Complexité de Turing

Bruno Grenet
D'après le cours de Marianne Delorme

Septembre 2007 - Janvier 2008
Dernière modification : 4 décembre 2009

Table des matières

1 Divers modèles de calcul **1**
- 1.1 Machine de Turing . . . 1
  - 1.1.1 Définitions . . . 1
  - 1.1.2 Complexité temporelle . . . 3
  - 1.1.3 Complexité spatiale . . . 4
  - 1.1.4 Relation entre complexité en temps et complexité en espace . . . 5
- 1.2 Autres modèles de calcul . . . 5
  - 1.2.1 Algorithmes de Markov . . . 5
  - 1.2.2 Machines RAM (*Random Access Memory*) . . . 6
  - 1.2.3 Circuits booléens . . . 6
  - 1.2.4 Automates cellulaires de dimension 1 à voisinage de premiers voisins . . . 7

2 Systèmes acceptables de programmation **9**
- 2.1 Définition d'un SAP . . . 9
- 2.2 Résultats préliminaires . . . 9
- 2.3 Traductions entre SAP . . . 11
- 2.4 Mesure de complexité abstraite . . . 12

3 P et NP **15**
- 3.1 La classe P . . . 15
- 3.2 Machine de Turing non déterministe . . . 17
- 3.3 La classe NP . . . 18
- 3.4 Conjecture de Hartmanis-Berman . . . 21
- 3.5 La classe CoNP . . . 25

4 La classe PSPACE **27**
- 4.1 Quelques exemples . . . 27
- 4.2 Théorème de Savitch . . . 28
- 4.3 Existence de problèmes PSPACE-complets . . . 29
- 4.4 Machines de Turing avec Oracle . . . 29
- 4.5 Hiérarchie polynomiale . . . 33
  - 4.5.1 Caractérisation . . . 33
  - 4.5.2 Théorèmes d'effondrement . . . 36
  - 4.5.3 Autres résultats . . . 36

5 Hiérarchies **39**
- 5.1 Comparaisons immédiates . . . 39
- 5.2 Théorèmes de hiérarchie déterministe . . . 41
- 5.3 Conséquences des théorèmes . . . 43
- 5.4 Lemmes de translation . . . 46
- 5.5 Théorèmes d'union . . . 47

Bibliographie **51**

Introduction

Ce polycopié essaie de reproduire le plus fidèlement possible le cours de Marianne Delorme, donné aux étudiants de Master 1 à l'École Normale Supérieure de Lyon entre septembre 2007 et janvier 2008. À ce titre, il s'adresse en premier lieu aux étudiants suivant l'équivalent de ce cours donné maintenant par d'autres professeurs. Quelques tout petits ajouts ont été effectués çà et là, mais ils seront malheureusement moins nombreux que les oublis et imprécisions qui se trouvent forcément disséminés dans le texte.

Ce cours de complexité ne prétend pas à l'exhaustivité, mais ouvre la porte à ce vaste domaine. En particulier, un grand nombre des techniques de preuves classiques sont abordés au cours des chapitres. Pour se rendre compte de l'étendue du domaine, le lecteur est invité à consulter les quelques ouvrages indiqués. Une bibliographie complète est disponible dans certains de ces ouvrages.

Les pré-requis pour ce cours sont une familiarité certaine avec les machines de Turing, des connaissances de base sur les fonctions récursives et les principaux résultats les concernant. Ces pré-requis correspondent très exactement aux cours de Fondements de l'Informatique de première année à l'ÉNS Lyon.

L'initiative de ce poly revient à Nicolas Estibals, qui a également assuré toute la partie technique (architecture LaTeX), et l'hébergement du projet. La transcription du cours n'est pas l'œuvre d'une seule personne. Je tiens à remercier ici tous ceux qui m'ont aidé dans cette tâche, en me passant des notes de cours, en écrivant certains chapitres, ou en effectuant un ingrat travail de relecture. Ce polycopié n'existerait bien entendu pas sans le fabuleux cours de Marianne. Toute la matière lui est due, mais le texte ne peut reproduire exactement les séances du vendredi matin. Merci pour ces cours, même si l'horaire fut parfois dur...

Ce cours étant truffé d'erreurs typographiques, orthographiques et mathématiques, toute correction ou suggestion est la bienvenue à l'adresse `bruno.grenet@ens-lyon.fr`. Tout ou partie de cet ouvrage est copiable à des fins non commerciales[1]. Un petit mot pour prévenir est néanmoins apprécié.

Bonne lecture !

Bruno Grenet, Automne 2008

[1] Cela dit, bonne chance à ceux qui voudraient faire de l'argent avec !

Chapitre 1

Divers modèles de calcul

Dans ce chapitre, nous introduisons la notion importante de *machine de Turing*. Il est des présentations plus intuitives de ces machines, mais le but ici est de donner une définition rigoureuse mathématiquement afin de pouvoir ensuite définir les notions de *complexités en temps et en espace*. Le lecteur habitué des machines de Turing reconnaîtra tout de même dans les définitions les objets dont il a l'intuition. La plupart des théorèmes énoncés dans ce chapitre ne seront pas démontrés, car ils sont vus comme un rappel de résultats déjà connus du lecteur.

La deuxième partie du chapitre est dédiée à d'autres modèles de calculs afin que l'on se rende compte que les machines de Turing sont *une* approche de la complexité et qu'il en existe d'autres. La complexité des circuits booléens est en particulier un sujet d'étude fécond. Ces autres modèles ne sont que mentionnés, et on ne considèrera dès le chapitre suivant que les machines de Turing.

1.1 Machine de Turing

1.1.1 Définitions

Cette partie peut paraître quelque peu rébarbative. Elle définit de manière rigoureuse ce que sont une machine de Turing, une configuration d'une telle machine, un pas de calcul, ainsi que les langages *reconnus* et *décidés* par ces machines.

Définition 1.1. Une *machine de Turing* est un septuplet[1]

$$\mathcal{M} = (Q, \Sigma, \Gamma, B, q_0, \delta, F)$$

où :
- Q est un ensemble fini d'états,
- Σ un alphabet fini,
- B un symbole dit « blanc » tel que $B \in \Gamma$ et $B \notin \Sigma$,
- Γ l'alphabet de travail $\Sigma \subsetneq \Gamma$,
- $q_0 \in Q$ l'état initial,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{G, D, S\}$ la fonction de transition,
- $F \subset Q$ l'ensemble des états d'arrêt.

Définition 1.2. La *configuration* d'une machine de Turing est décrite par

$$\mathcal{C} = uqv,\ u, v \in \Gamma^*,$$

où q est l'état courant de la machine, la tête de lecture étant placée sur la première lettre de v. Le ruban ne contient en dehors de u et v que des caractères blancs. Le mot uv est donc le mot contenu sur le ruban de la machine.

[1] Ne soyons pas si catégorique, la littérature regorge de variations sur ces définitions.

Définition 1.3. Un *pas de calcul* (ou dérivation simple) sur une machine de Turing

$$(\mathcal{C} = uqv) \vdash\!\!\!- (\mathcal{C}' = u'q'v')$$

est défini de la manière suivante : supposons que $u = wx$ avec w un mot et x une lettre, et que $v = aw'$ avec a une lettre et w' un mot.

- Si $\delta(q, a) = (q', a', G)$, alors $u' = w$ et $v' = xa'w'$.
- Si $\delta(q, a) = (q', a', S)$, alors $u' = u$ et $v' = a'w'$.
- Si $\delta(q, a) = (q', a', D)$, alors $u' = ua'$ et $v' = w'$.

Définition 1.4. La *dérivation* sur les configurations est définie par

$$\mathcal{C} \vdash\!\!\!-^* \mathcal{C}' \iff \exists\, (\mathcal{C}_i)_{0 \leqslant i \leqslant n} \text{ telle que } \mathcal{C}_0 = \mathcal{C}, \mathcal{C}_n = \mathcal{C}' = uq_fv \text{ avec } q_f \in F, \text{ et } \forall\, i\ , \mathcal{C}_i \vdash\!\!\!- \mathcal{C}_{i+1}.$$

Ainsi la dérivation n'est définie que pour les configurations où $\mathcal{M}$ s'arrête. Le lecteur avisé aura reconnu dans ces définitions les définitions classiques des machines de Turing déterministes. La présentation un peu fastidieuse qui en est faite, avec les configurations présentées comme des mots, permet une définition précise des notions de complexité. On verra dans la suite des variations à ces définitions, et en particulier les définitions de machine de Turing non déterministe et de machine de Turing avec oracle.

Une machine de Turing peut-être considérée comme calculant une fonction $f : \Sigma^* \to \Sigma^* \cup \{\bot\}$ (non nécessairement totale). Le mot u est dans le domaine de f si et seulement si, partant d'une configuration initiale q_0u, $\mathcal{M}$ s'arrête au bout d'un temps fini dans une configuration d'arrêt $vq_{\text{arrêt}}v'$, où $q_{\text{arrêt}} \in F$. vv' est alors l'image de u par f.

Une machine de Turing peut aussi être vue comme reconnaissant des langages. Le mot $u \in \Sigma^*$ est reconnu par $\mathcal{M}$ si et seulement si, partant d'une configuration initiale q_0u, $\mathcal{M}$ termine dans une configuration d'acceptation. Le langage reconnu par $\mathcal{M}$ est alors

$$\mathcal{L}(\mathcal{M}) = \left\{ u \in \Sigma^* \;\middle|\; q_0u \vdash\!\!\!-^* u'q_{\text{acceptation}}v' \right\}$$

Définition 1.5. Les langages reconnus par machine de Turing sont dits *récursivement énumérables.*

Définition 1.6. On dit qu'une machine $\mathcal{M}$ décide un langage L si

$$\begin{cases} F = \{q_{\text{acceptation}}, q_{\text{rejet}}\} \\ u \in L \iff q_0u \vdash\!\!\!-^* u'q_{\text{acceptation}}v' \\ u \notin L \iff q_0u \vdash\!\!\!-^* u'q_{\text{rejet}}v' \\ \mathcal{M} \text{ termine sur chaque entrée.} \end{cases}$$

On note alors $L = \mathcal{L}(\mathcal{M})$.

Remarque. On a noté de la même façon les langages *reconnus* et *décidés* par une machine de Turing. Le contexte ou une indication précise permettront de savoir de quoi on parle.

Remarque (Digression sur la notion de problème). Si on ne s'intéresse qu'aux problèmes de décision, un problème est un ensemble de données, un codage pour ces données et une question fermée. Par exemple :

- Primalité : un entier n codé binaire est-il premier ?
- `SAT` : une formule binaire convenablement codée est-elle satisfiable ?
- Correspondance de Post : soit $(u_i, v_i) \in ((A^*)^2)^p$, existe-t-il une suite d'indices $i_1, \ldots, i_k$ tel que $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$
- Problème de l'arrêt : un programme en `C`, une entrée n, ce programme s'arrête-t-il sur n ?

La solution du problème est alors représentée par une partition sur l'ensemble des mots :

– l'ensemble des instances positives, *i.e.* les mots pour lesquels la machine s'arrête et donne une réponse positive ;
– l'ensemble des instances négatives, *i.e.* les mots pour lesquels la machine s'arrête et donne une réponse négative ;
– l'ensemble des mots qui ne sont pas une instance, *i.e.* les mots pour lesquels la machine ne s'arrête pas.

On donne dès à présent une variation de notre définition de machine de Turing.

Définition 1.7. Une machine de Turing à k rubans est un octuplet $(Q, \Sigma, \Gamma, B, q_0, \delta, F, k)$ où
– Q est un ensemble fini d'états,
– Σ un alphabet fini,
– B un symbole dit « blanc » tel que $B \in \Gamma$ et $B \notin \Sigma$,
– Γ l'alphabet de travail $\Sigma \subsetneq \Gamma$,
– $q_0 \in Q$ l'état initial,
– $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{G, D, S\}^k$ la fonction de transition,
– $F \subset Q$ l'ensemble des états d'arrêt.

La seule variation ici est que la fonction de transition n'opère plus sur un caractère à la fois, mais sur un k-uplet. Il est important de noter ici que les rubans sont indépendants, c'est-à-dire que le déplacement de la tête de lecture est différent pour chaque ruban. Les configurations peuvent toujours être représentées par des mots, et les définitions précédentes restent valables. On attire également l'attention du lecteur sur le fait que les rubans n'ayant pas d'origine, on peut considérer que ce sont ceux-ci qui se déplacent, et non la tête de lecture. Ceci afin de pouvoir continuer à écrire nos configurations avec un seul emplacement pour la tête de lecture malgré la présence de plusieurs rubans.

On note alors $(u_1, \dots, u_k)q(v_1, \dots, v_k)$ pour signifier que la machine est dans l'état q et que le ruban i contient le mot $u_i v_i$, la tête de lecture de ce ruban étant placée au dessus de la première lettre de v_i.

1.1.2 Complexité temporelle

On s'intéresse ici à la complexité en temps d'une machine de Turing, à savoir le nombre d'étapes de calcul nécessaires à la machine pour donner le résultat.

Soit une machine de Turing $\mathcal{M} = (Q, \Sigma, \Gamma, B, q_0, \delta, F)$. On définit deux fonctions pour caractériser sa complexité temporelle :

$$t_{\mathcal{M}} : \begin{cases} \Sigma^* & \to \mathbb{N} \\ u & \mapsto \begin{cases} \text{le nombre de transitions réalisées} & \text{si } \mathcal{M} \text{ termine sur } u, \\ \bot & \text{sinon} \end{cases} \end{cases}$$

$$T_{\mathcal{M}} : \begin{cases} \mathbb{N} & \to \mathbb{N} \\ n & \mapsto \max\{t_{\mathcal{M}}(u) \mid |u| = n \text{ et } \mathcal{M} \text{ termine sur } u\} \text{ si c'est défini.} \end{cases}$$

Définition 1.8. On dit qu'une machine de Turing $\mathcal{M}$ travaille en temps $t : \mathbb{N} \to \mathbb{N}$ si pour tout $u \in \Sigma^*$, $t_{\mathcal{M}}(u) \leqslant t(|u|)$.

Définition 1.9. Soit $t : \mathbb{N} \to \mathbb{N}$. On définit la classe de complexité en temps $\mathsf{DTIME}(t)$ comme étant la classe des langages décidés par une machine de Turing qui travaille en temps $t(n)$.

Remarque. Dans cette définition, la machine de Turing considérée doit bien entendu être *déterministe*. On fera dans la suite l'abus de notation suivant : au lieu de noter la classe $\mathsf{DTIME}(t)$, elle sera très souvent notée $\mathsf{DTIME}(t(n))$, c'est-à-dire qu'on parlera de classes de complexité telles que $\mathsf{DTIME}(n^2)$ en lieu et place de $\mathsf{DTIME}(n \mapsto n^2)$.

Théorème 1.10 (accélération linéaire)**.** *Soit $t : \mathbb{N} \to \mathbb{N}$, c une constante strictement positive. On suppose que $\lim_{n\to\infty} \frac{t(n)}{n} = \infty$ alors :*

$$\mathsf{DTIME}(ct) = \mathsf{DTIME}(t)$$

Preuve. On ne donne que l'idée de cette preuve classique. On peut supposer dans perte de généralité que $c < 1$. Supposons que L soit reconnu en temps t par une machine $\mathcal{M}$ qui travaille sur l'alphabet Σ. On construit la nouvelle $\mathcal{M}'$ de la façon suivante. L'alphabet utilisé est Σ^m pour un certain m, chaque symbole pour $\mathcal{M}'$ codant m symboles de Σ. On peut vérifier qu'en huit pas de calcul, $\mathcal{M}'$ peut simuler m pas de calcul de $\mathcal{M}$. On obtient le résultat en choisissant correctement la valeur de m en fonction de la constante c souhaitée. □

Théorème 1.11. *Si L est décidé par une machine de Turing à k rubans en temps t, alors il existe une machine à un ruban qui décide L en temps $5kt^2$.*

Preuve. Soit M la machine à k rubans qui décide L, et construisons un machine $\bar{M}$ à un ruban qui décide le même langage. L'idée consiste simplement à regrouper les cases du ruban de $\bar{M}$ par paquet de k cases. Ainsi, les cases $1, k+1, 2k+1, \ldots$ représenteront le premier ruban de M, les cases $2, k+2, 2k+2, \ldots$ le deuxième ruban, et ainsi de suite. Pour noter l'emplacement de la tête de lecture de M sur chaque ruban, il faut doubler l'alphabet. Pour chaque lettre a de l'alphabet de M, l'alphabet de $\bar{M}$ contient les deux lettres a et $\bar{a}$. La lettre a joue le rôle habituel, tandis que si la case $r \cdot k + c$ contient la lettre $\bar{a}$, cela signifie que la case c du ruban r contient la lettre a et que la tête de lecture pointe sur cette case.

La définition de la fonction de transition de $\bar{M}$ est alors assez claire, et on peut vérifier que $\bar{M}$ travaille en temps au plus $5kt^2$. □

Remarque. Cette borne ne peut pas être améliorée en général. En effet, on peut montrer que le langage `Pal` des palindromes ($x \in \texttt{Pal} \iff \exists u, x = u\bar{u}$, où $\bar{u}$ représente le miroir de u) peut être reconnu en temps linéaire par une machine à deux rubans mais nécessite un temps quadratique sur une machine à un ruban.

D'autre part, on peut aussi montrer qu'une machine de Turing à k rubans peut être simulée par une machine à deux rubans en temps $\mathcal{O}(t \cdot \log t)$.

1.1.3 Complexité spatiale

Contrairement à ce qui précède, ce qui suit a pour objet la complexité *en espace* des machines de Turing. Alors qu'on s'est précédemment intéressés au temps que mettait une machine pour donner le résultat, on essaie ici de savoir l'espace occupé sur les rubans lors du fonctionnement de la machine.

On considère des machines de Turing avec ruban(s) d'entrée, ruban(s) de sortie et ruban(s) de travail. On n'évalue l'espace utilisé pendant un calcul que sur les rubans de travail. Si l'on a k rubans de travail, l'espace comptabilisé sur le ruban i pendant le calcul C est le nombre maximum de cases visité sur ce ruban ; on le note $l_i(C)$.

De même que pour la complexité temporelle, on définit deux fonctions auxiliaires :

$$s_{\mathcal{M}} : \begin{cases} \Sigma^* & \to \mathbb{N} \\ u & \mapsto \sum_{j=1}^{k} l_j(C_u) \text{ si le calcul termine} \end{cases}$$

$$S_{\mathcal{M}} : \begin{cases} \mathbb{N} & \to \mathbb{N} \\ n & \mapsto \max\{s_{\mathcal{M}}(u) \mid |u| = n \text{ et } \mathcal{M} \text{ termine sur } u\} \end{cases}$$

$S_{\mathcal{M}}$ est la complexité en espace de $\mathcal{M}$. Un machine de Turing $\mathcal{M}$ fonctionne en espace $s : \mathbb{N} \to \mathbb{N}$ si $\forall n \in \mathbb{N},\ S_{\mathcal{M}}(n) \leqslant s(n)$.

Définition 1.12. La classe de complexité en espace $\mathsf{DSPACE}(s(n))$ définie par la fonction $s : \mathbb{N} \to \mathbb{N}$ est la classe des langages décidés par une machine de Turing déterministe qui travaille en espace $s(n)$.

Théorème 1.13 (compression linéaire)**.** *Soit $s : \mathbb{N} \to \mathbb{N}$, soit c une constante strictement positive, alors :*

$$\mathsf{DSPACE}(s) = \mathsf{DSPACE}(cs)$$

Preuve. On peut supposer que $c < 1$ sans perdre en généralité.

On a alors trivialement : $\mathsf{DSPACE}(cs) \subseteq \mathsf{DSPACE}(s)$.

Soit $L \in \mathsf{DSPACE}(s)$, il existe une machine de Turing $\mathcal{M}$ qui peut décider L en espace s. On cherche à en déduire une machine $\mathcal{M}'$ qui le décide en temps cs. On crée des macro-cellules de taille $k = \lceil \frac{1}{c} \rceil$. On adapte la fonction de transition pour qu'elle effectue le calcul uniquement sur les macro-cellules. On se convainc assez facilement que la nouvelle machine calcule L en espace $\frac{s}{k} \leqslant cs$.

□

1.1.4 Relation entre complexité en temps et complexité en espace

Les deux complexités précédemment définies sont liées d'après le théorème ci-après. C'est un premier pas dans l'étude des classes de complexité, au cours de laquelle il y aura des va-et-vient sans cesse entre les deux types de complexité.

Théorème 1.14.

$$s_{\mathcal{M}}(u) \leqslant t_{\mathcal{M}}(u) \leqslant 2^{c_{\mathcal{M}} s_{\mathcal{M}}(u)}$$

où $c_{\mathcal{M}}$ est une constante dépendant uniquement de $\mathcal{M}$.

Preuve. La première inégalité se justifie par le fait qu'en t étapes, on ne peut lire au plus que t cases. En effet, on ne lit à chaque étape qu'une seule case.

Pour la deuxième, comme $\mathcal{M}$ termine sur u, elle ne peut passer deux fois par la même configuration. Or le nombre de configurations possibles de $\mathcal{M}$ atteignable depuis le mot u peut être calculé de la façon suivante. Une configuration est définie par

- un état ($|Q|$ possibilités),
- un mot de longueur au plus $s_{\mathcal{M}}(u)$ ($|\Gamma|^{s_{\mathcal{M}}(u)}$ possibilités),
- la position de la tête de lecture, parmi $s_{\mathcal{M}}(u) + c$ positions.

Les possibilités sont donc au total de l'ordre de $2^{c_{\mathcal{M}} s_{\mathcal{M}}(u)}$. □

1.2 Autres modèles de calcul

Cette partie présente quelques uns des autres modèles de calcul existants. La particularité de ceux qui sont présentés est la possibilité de définir des notions de complexité dessus. Ils sont un autre angle d'attaque pour défricher les questions de complexité. Ils ne sont cependant cités qu'à titre informatif, et le reste des chapitres les ignorera. Le lecteur intéressé est une fois de plus invité à se reporter à la bibliographie pour trouver plus d'informations sur ces modèles, certains définissant des domaines de recherche très actifs.

1.2.1 Algorithmes de Markov

Définition 1.15. Soit un alphabet Σ, un algorithme de Markov $\mathcal{M}_a$ est une suite finie de règles $p_i \to_{(t)} q_i$ avec $(p_i, q_i) \in (\Sigma^*)^2$, où t, quand il figure, signifie que c'est une règle de terminaison.

Le calcul de $\mathcal{M}_a$ sur un mot u est une suite de substitutions dans le mot courant. A chaque étape , on applique la première transformation de la liste qui convient, *i.e.* la règle dont le mot de gauche est un facteur du mot courant. On peut donc constater qu'un algorithme de Markov $\mathcal{M}_a$ est déterministe.

Plus formellement on peut définir une relation de dérivation $\vdash$ comme précédemment sur les machines de Turing :

$$\begin{aligned} \mathcal{M}_a : \quad u \vdash_0 v \quad &\iff v = u \\ u \vdash_{n+1} v \quad &\iff \exists w \in \Sigma^* \begin{cases} u & \vdash_n w \\ w & \vdash_1 v \end{cases} \end{aligned}$$

1.2.2 Machines RAM (*Random Access Memory*)

On se donne un alphabet $\Sigma = \{a_1, \ldots, a_n\}$. Une machine RAM consiste en un nombre potentiellement infini de registres qui sont nommés et qui contiennent chacun un mot sur Σ (éventuellement le mot vide), et d'un programme au moyen de 7 types d'instructions.

- 1_j : X ADD$_j$ Y $(1 \leqslant j \leqslant k)$
- 2 : X DEL Y
- 3 : X CLEAR Y
- 4 : X $Y \leftarrow Z$
- 5 : X JMP X'
- 6 : X Y JMP$_j$ X'
- 7 : X CONTINUE

Dans ces instructions, X et X' représentent des noms de lignes et Y et Z des noms de registres. La signification des instructions est donnée ci-dessous.

- 1_j : Concaténer a_j à a fin du mot contenu par Y.
- 2 : Supprimer le premier caractère du mot contenu par Y.
- 3 : Effacer le contenu de Y (remplacer par le mot vide).
- 4 : Substituer au contenu de Y le contenu de Z (sans modifier Z).
- 5 : Aller à la ligne X'.
- 6 : Aller à la ligne X' si le mot contenu dans Y commence par a_j.
- 7 : Instruction de fin de programme.

1.2.3 Circuits booléens

Ce modèle de calcul s'appuie sur l'équivalence entre une fonction à valeur booléenne et une formule booléenne.

Proposition 1.16. *Si $f : \{0,1\}^n \to \{0,1\}$, on sait lui associer de façon canonique une formule booléenne φ_f telle que $f(v) = 1 \iff v$ satisfait φ_f. Et réciproquement, toute formule booléenne à n variables définit une fonction $f : \{0,1\}^n \to \{0,1\}$ telle que φ est satisfiable si et seulement si $f(v) = 1$.*

Définition 1.17. Un *circuit booléen* est un graphe acyclique $C = (V, E)$. Les sommets du graphe $V = \{1, \ldots, n\}$ sont les portes du circuit. Toute arête peut être représentée par (i, j), $i < j$. Ces portes sont de degré entrant 0, 1 ou 2. La seule porte de degré sortant 0 est n, et est dite porte de sortie, les autres portes ayant un degré sortant 1.

Ces portes ont un type appartenant à $\{0, 1, \wedge, \vee, \neg\} \cup \{x_i\}$. Si i est une porte, son type est noté $s(i)$. Si $s(i) \in \{0,1\} \cup \{x_i\}$, $\deg^-(i) = 0$. Si $s(i) = \neg$, $\deg^-(i) = 1$. Si $s(i) \in \{\wedge, \vee\}$, $\deg^-(i) = 2$.

La complexité d'un circuit est donnée par sa taille en nombre de portes. La proposition suivante est un exemple de résultat qu'on peut prouver sur les circuits booléens.

Proposition 1.18. *Pour tout $n \geqslant 2$, il existe une fonction booléenne $f : \{0,1\}^n \to \{0,1\}$ telle qu'aucun circuit de taille inférieure à $2^n/2n$ ne la calcule.*

Preuve. A VOIR !

Soit m la taille d'un circuit : pour chaque porte i, on a $m^2 \cdot (n+5)$ choix possibles. Donc avec m portes, on a $\left(m^2 \cdot (n+5)\right)^m$ choix. Or il y a 2^{2^n} fonction booléenne de $\{0,1\}^n$ dans $\{0,1\}$. Si $m = 2^n/2n$, alors

$$\log\left((m^2 \cdot (n+5))^m\right) = 2^n \cdot \frac{2n - \log \frac{(2n)^2}{n+5}}{2n} < 2^n.$$

Certaines fonctions ne sont donc pas calculées par de tels circuits. □

Ce modèle de calcul permet de calculer toute sorte d'entrée, et non seulement celles d'une taille donnée. C'est l'objet de la prochaine définition.

Définition 1.19. Une fonction $f : \{0,1\}^* \to \{0,1\}$ est dite *calculable par circuit* s'il existe une famille (C_n) de circuits tels que $f_{|\{0,1\}^n}$ soit calculée par C_n.

Sans restriction sur la famille, on peut *calculer par circuits* des fonctions non récursives.

Proposition 1.20. *Pour tout langage L sur $\{0,1\}$ tel que $\varepsilon \notin L$, il existe une famille de circuit qui le décide.*

Preuve. Posons $L_n = L \cap \{0,1\}^n$. Pour chaque $w \in L_n$, on sait construire le circuit C_w le caractérisant. C_n est obtenu comme

$$C_n = \bigvee_{w \in L_n} C_w.$$

□

1.2.4 Automates cellulaires de dimension 1 à voisinage de premiers voisins

On considère un ruban bi-infini, dont chaque case contient un automate fini. Un automate cellulaire $\mathcal{A}$ est un couple (Q, δ) où Q est un ensemble fini d'état, et $\delta : Q^3 \to Q$. Une configuration de $\mathcal{A}$ est une fonction $c : \mathbb{Z} \to Q$. L'évolution de $\mathcal{A}$ à partir d'une configuration c est une suite de configuration (c_i), avec $c_0 = c$, et telle que pour tout i, $c_i(z) = \delta(c_{i-1}(z-1), c_{i-1}(z), c_{i-1}(z+1))$.

On peut restreindre le fonctionnement de $\mathcal{A}$ à ses configurations « finies » en ajoutant à Q un état dit *quiescent* e qui satisfait $\delta(e,e,e) = e$. Une configuration finie est alors une configuration c telle que $c(z) = e$ sauf pour un nombre fini d'entiers z.

On peut prouver que les automates cellulaires restreints à leurs configurations finies ont exactement la puissance de calcul des machines de Turing.

Chapitre 2

Systèmes acceptables de programmation

Ce chapitre introduit la notion de *système acceptable de programmation* qui donne une structure à l'ensemble des algorithmes. L'objectif est de démontrer le *théorème d'isomorphisme* de Rogers et de définir la notion de *mesure abstraite de complexité*.

2.1 Définition d'un SAP

Définition 2.1. Un système acceptable de programmation (SAP) est une énumération $(\varphi_i)_{i\geqslant 0}$ des fonctions partielles calculables de $\mathbb{N}$ dans $\mathbb{N}$ telle que :

(i) il existe une fonction universelle φ_{univ} telle que $\varphi_{univ}(\langle i, j\rangle) = \varphi_i(j)$[1] pour tous i et j dans $\mathbb{N}$,

(ii) il existe un théorème *s-n-m*.

Dans la suite, un élément d'un SAP sera appelé un programme. Un programme calcule une fonction. On distingue les deux notions, le programme du SAP qui est un objet *informatique*, et la fonction calculée qui est un objet *mathématique*. Ainsi, on pourra dire que deux programmes distincts calculent la même fonction.

2.2 Résultats préliminaires

Cette partie montre l'existence d'une infinité de programmes calculant la même fonction dans un même SAP. De plus, les indices de ces programmes dans le SAP sont calculables.

Ce premier résultat démontre l'existence de deux programmes d'indices différents calculant la même fonction.

Proposition 2.2. *Soit (φ_i) un SAP. Pour tout entier k, il existe un entier j différent de k tel que $\varphi_j = \varphi_k$.*

Preuve. Considérons $\langle x, y\rangle \mapsto \begin{cases} \varphi_k(y) & \text{si } x \neq k \\ \varphi_{x+1}(y) & \text{sinon} \end{cases}$.

On définit ainsi une fonction partielle calculable. Donc il existe $a \in \mathbb{N}$ tel que cette fonction soit calculée par φ_a.

$$\varphi_a(\langle x, y\rangle) = \begin{cases} \varphi_k(y) & \text{si } x \neq k \\ \varphi_{x+1}(y) & \text{sinon} \end{cases}$$

[1] Soit les deux fonctions divergent, soit elles sont égales.

Par le théorème *s-n-m*, il existe une fonction totale calculable s_1^1 telle que $\varphi_a(\langle x, y\rangle) = \varphi_{s_1^1(\langle a,x\rangle)}(y)$.

Soit $f : x \mapsto s_1^1(\langle a, x\rangle)$. Cette fonction est totale calculable, donc d'après le théorème du point fixe de Kleene, il existe e tel que $\varphi_{f(e)} = \varphi_e$.

$$\left.\begin{array}{l}\text{Si } e \neq k, \text{ on choisit } j = e \\ \text{Si } e = k, \text{ on choisit } j = e + 1\end{array}\right\} \text{ ce qui donne bien } \varphi_j = \varphi_k.$$

□

Le lemme suivant généralise la proposition précédente. Il stipule que pour chaque programme du SAP, il existe une infinité de programmes calculant la même fonction. Sa démonstration construit explicitement une fonction totale récursive calculant les indices des programmes calculant la même fonction.

Lemme 2.3 (Lemme de remplissage)**.** *Soit (φ_i) un SAP. Il existe une fonction totale récursive ρ qui à tout programme φ_i associe une infinité de programmes distincts calculant la même fonction.*

Plus précisément, ρ est telle que

(i) pour tout $i \geqslant 0$, $x \mapsto \rho(\langle i, x\rangle)$ est injective,

(ii) pour tout i et tout x, $\varphi_{\rho(\langle i,x\rangle)} = \varphi_i$.

Preuve constructive. On va construire la fonction ρ récursivement, en se souvenant que la seule variable est le deuxième argument, i étant fixé.

On pose $\rho(\langle i, 0\rangle) = i$.

Supposons que $\rho(\langle i, k\rangle)$ soit défini pour $0 \leqslant k < x$ et que $\varphi_{\rho(\langle i,k\rangle)} = \varphi_i$. Il s'agit de définir $\rho(\langle i, x\rangle)$.

On considère la fonction partielle calculable suivante :

$$\langle i, j, z\rangle \mapsto \begin{cases} \varphi_{\max\{\rho(\langle i,k\rangle) \mid 0\leqslant k<x\}+1}(z) & \text{si } j \in \{\rho(\langle i, k\rangle) \mid 0 \leqslant k < x\} \\ \varphi_i(z) & \text{sinon} \end{cases}$$

Cette fonction a un numéro dans l'énumération. Soit a ce numéro. D'après le théorème *s-n-m*, $\varphi_a(\langle i, j, z\rangle) = \varphi_{s_2^1(\langle a,i,j\rangle)}(z)$.

Si on considère $f : j \mapsto s_2^1(\langle a, i, j\rangle)$ et si on lui applique le théorème de Kleene, on obtient l'existence d'un entier e tel que $\varphi_{f(e)} = \varphi_e$. D'où

$$\varphi_e(z) = \varphi_{f(e)}(z) = \varphi_a(\langle i, e, z\rangle) = \begin{cases} \varphi_{\max\{\rho(\langle i,k\rangle) \mid 0\leqslant k<x\}+1}(z) & \text{si } e \in \{\rho(\langle i, k\rangle) \mid 0 \leqslant k < x\} \\ \varphi_i(z) & \text{sinon} \end{cases}$$

Si $e \in \{\rho(\langle i, k\rangle) \mid 0 \leqslant k < x\}$, on pose $\rho(\langle i, x\rangle) = \max\{\rho(\langle i, k\rangle) \mid 0 \leqslant k < x\} + 1$. Sinon, $\rho(\langle i, x\rangle) = e$.

Montrons maintenant que ρ vérifie les deux conditions du lemme :

(i) $x \mapsto \rho(\langle i, x\rangle)$ est injective car :
- $p(\langle 0, 0\rangle) = 0$.
- Soit $x \neq x'$, par exemple $x < x'$. Si $\rho(\langle i, x\rangle) = \rho(\langle i, x'\rangle)$, alors soit $\rho(\langle i, x'\rangle) = \max\{\rho(\langle i, k\rangle) \mid 0 \leqslant k < x\}$ + 1 et il y a contradiction avec le fait que $\rho(\langle i, x\rangle) \in \{\rho(\langle i, k\rangle) \mid 0 \leqslant k < x\}$, soit $\rho(\langle i, x\rangle) \notin \{\rho(\langle i, k\rangle) \mid 0 \leqslant k < x\}$, et donc $\rho(\langle i, x\rangle)$ non plus, ce qui est absurde.

(ii) $\varphi_{\rho(\langle i,x\rangle)} = \varphi_i$ est vrai car :
- Si $\rho(\langle i, x\rangle) = \max\{\rho(\langle i, k\rangle) \mid 0 \leqslant k < x\} + 1$, c'est que e appartient à cet ensemble, donc il existe k_e tel que $e = \langle i, k_e\rangle$, avec $k_e < x$. Donc, $\varphi_{\rho(\langle i,x\rangle)} = \varphi_e = \varphi_{\rho(\langle i,k_e\rangle)} = \varphi_i$.
- Sinon, $\rho(\langle i, x\rangle) = e$. Alors $\varphi_{\rho(\langle i,x\rangle)} = \varphi_e$ et comme $\varphi_e = \varphi_i$, on peut conclure.

□

2.3 Traductions entre SAP

Cette partie s'intéresse à l'égalité de deux programmes de deux SAP différents, et aboutit sur le théorème d'isomorphisme de Rogers qui affirme que deux SAP sont toujours isomorphes et que l'isomorphisme est une fonction calculable. On peut alors définir ce qu'est une *mesure abstraite de complexité*.

Cette première proposition affirme qu'il existe une fonction calculable permettant de passer d'un SAP à un autre.

Proposition 2.4. *Soit (φ_i) et (ψ_i) deux SAP. Alors il existe une fonction totale calculable t telle que $\varphi_i = \psi_{t(i)}$ pour tout i.*

Preuve. Comme (φ_i) est un SAP, il existe φ_{univ} telle que $\varphi_{univ}(\langle i, j\rangle) = \varphi_i(j)$. Et comme (ψ_i) est un SAP, il existe k tel que $\varphi_{univ}(\langle i, j\rangle) = \psi_k(\langle i, j\rangle) = \varphi_i(j)$.

Comme (ψ_i) est un SAP, on a un théorème $s - n - m$. Donc il existe s_1^1 totale récursive telle que $\psi_k(\langle i, j\rangle) = \psi_{s_1^1(\langle k,i\rangle)}(j)$.

On définit alors t comme $i \mapsto s_1^1(\langle k, i\rangle)$. □

La prochaine proposition étend le résultat précédent en précisant la forme prise par la fonction permettant de passer d'un SAP à un autre.

Proposition 2.5. *Soit (φ_i) et (ψ_i) deux SAP. Alors il existe des fonctions totales récursives g et h telles que*

(i) $g(0) > 0$ *et* $h(0) > 0$,

(ii) g *et* h *sont strictement croissantes,*

(iii) $\forall\ i,\ \varphi_i = \psi_{g(i)}$ *et* $\psi_i = \varphi_{h(i)}$.

Preuve. D'après la proposition précédente, il existe une fonction totale récursive t telle que $\varphi_i = \psi_{t(i)}$. On va définir la fonction g. La fonction h se définit exactement de la même façon.

On pose

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

où ρ est la fonction du lemme de remplissage. Montrons que $g(0)$ est bien défini, sa stricte positivité étant évidente.

Par définition, $\rho(\langle t(0), 0\rangle) = t(0)$ est défini.

– Soit $t(0) = 0$ et comme $x \mapsto \rho(\langle 0, x\rangle)$ est injective, il existe un $x > 0$ dans l'ensemble $\{y | \rho(\langle t(0), y\rangle) > 0\}$.
– Soit $t(0) \neq 0$ et $t(0)$ est élément de ce même ensemble.

Supposons maintenant $g(0)$, $g(1)$, …, $g(n-1)$ définis. On veut définir $g(n)$. On pose

$$g(n) = \rho(\langle t(n), \min\{y | \rho(\langle t(n), y\rangle) > g(n-1)\}\rangle)$$

g est clairement strictement croissante.

De plus, $\varphi_i = \psi_{t(i)} = \psi_{\rho(\langle t(i),y\rangle)}$ pour un y qui permet de définir $g(i)$. Donc $\varphi_i = \psi_{g(i)}$. □

Les deux précédentes propositions permettent de démontrer ce résultat central sur les SAP, à savoir que deux SAP sont toujours isomorphes.

Théorème 2.6 (Théorème d'isomorphisme de Rogers)**.** *Soit (φ_i) et (ψ_i) deux SAP. Alors il existe une fonction totale calculable f, bijective, telle que f^{-1} est calculable, et telle que $\varphi_i = \psi_{f(i)}$ et $\psi_i = \varphi_{f^{-1}(i)}$ pour tout i.*

Preuve. On sait qu'il existe des fonctions g et h données par la proposition précédente. De ses propriétés résulte que $g(x) > x$ et $h(x) > x$ pour tout x.

On veut définir f. Soit x un entier. Si x admet un antécédent par g, il est unique. Et de même pour h. Posons

$$\mathcal{C}_x = \left\{x, h^{-1}(x), g^{-1}h^{-1}(x), h^{-1}g^{-1}h^{-1}(x),\ \dots\ , h^{-1}(g^{-1}h^{-1})^i(x), (g^{-1}h^{-1})^{i+1}(x),\ \dots\right\}$$

$\mathcal{C}_x$ est une partie non vide de $\mathbb{N}$, majorée par x donc finie. Son plus petit élément est soit de la forme $h^{-1}(g^{-1}h^{-1})^i(x)$, soit de la forme $(g^{-1}h^{-1})^i(x)$. Dans le premier cas, on pose $f(x) = h^{-1}(x)$ et dans le second cas, on pose $f(x) = g(x)$. On vérifie que f est bijective.

Supposons que $f(x) = f(y)$. Si $f(x) = h^{-1}(x)$ et $f(y) = h^{-1}(y)$, alors clairement $x = y$. De même, si $f(x)$ et $f(y)$ sont les images de x et y par g, g étant strictement croissante, $x = y$. Enfin, si $f(x) = h^{-1}(x)$ et $f(y) = g(y)$, on obtient $h^{-1}(x) = g(y)$. On va arriver à une contradiction. Soit i et j tels que $\min C_x = h^{-1}(g^{-1}h^{-1})^i(x)$ et $\min C_y = (g^{-1}h^{-1})^j(y)$, et notons $z = \min C_y$. Alors

$$\begin{aligned}(g^{-1}h^{-1})^j(y) &= (g^{-1}h^{-1})^j(g^{-1}g(y)) \\ &= (g^{-1}h^{-1})^{j+1}(x).\end{aligned}$$

Si $j + 1 \leqslant i$, alors $(g^{-1}h^{-1})^{j+1}(x)$ admet un antécédent par h. Mais alors l'existence de cet antécédent contredit la minimalité de z. Si au contraire, $j + 1 > i$, c'est $\min C_x$ qui n'est pas un minimum. Donc cette situation, où $f(x)$ et $g(y)$ n'ont pas la même forme, est impossible.

On va maintenant montrer la surjectivité de f. La démonstration, qui va construire de fait un inverse à f, montrera également que f^{-1} est calculable. Soit $y \in \mathbb{N}$. On veut calculer son antécédent x par f. Examinons $z = \min C_{h(y)}$:

- si $z = h^{-1}(g^{-1}h^{-1})^i(h(y))$, alors $f(h(y)) = h^{-1}(h(y)) = y$ donc $f^{-1}(y) = h(y)$.
- sinon, $z = (g^{-1}h^{-1})^i(h(y)) = (g^{-1}h^{-1})^{i-1}(g^{-1}(y))$. Ceci nous assure que y admet un antécédent x par g. Un raisonnement similaire au précédent impose que $f(x)$ soit bien $g(x)$. Et donc $f^{-1}(y) = g^{-1}(y)$. Comme $g^{-1}(y) < y$, on peut bien calculer sa valeur : il suffit de tester pour chaque valeur inférieure à y si son image par g est y.

□

2.4 Mesure de complexité abstraite

Le théorème d'isomorphisme de Rogers permet un niveau d'abstraction plus élevé encore. On peut en effet raisonner sur un SAP particulier et obtenir des résultats qui resteront valables pour tout SAP. Cette idée permet de définir une *mesure de complexité abstraite.*

Définition 2.7. Une *mesure de complexité abstraite* pour un SAP (φ_i) est une énumération de fonctions partielles récursives (θ_i) telle que :

(i) $\theta_i(x)$ est défini $\iff$ $\varphi_i(x)$ est défini.

(ii) $\{\langle i, x, y\rangle \ | \theta_i(x) \leqslant y\}$ est récursif.

Exemple 2.8. La fonction $t_{\mathcal{M}} : u \mapsto$ « nombre de pas de calcul de $\mathcal{M}$ sur u » définit une mesure de complexité abstraite.

Remarque. (φ_i) n'est pas une mesure de complexité pour (φ_i) : la condition (ii) n'est pas satisfaite à cause du théorème de l'arrêt.

Remarque. On peut définir des mesures de complexité un peu extravagantes. Soit (φ_i) un SAP. On se donne une fonction f totale calculable. Il existe j tel que $f = \varphi_j$. On se donne une mesure de complexité (θ_i). On peut alors construire la mesure (μ_i) telle que pour $i \neq j$, $\mu_i = \theta_i$ et $\mu_j = 0$.

Le théorème suivant montre que deux mesures de complexité sont toujours intimement[2] liées, et ne sont donc pas totalement quelconques.

Théorème 2.9 (Comparaison des mesures de complexité abstraites)**.** *Soit (φ_i) et (ψ_i) deux SAP, et soit respectivement (θ_i) et (μ_i) des mesures de complexité abstraites sur ces SAP. Soit t une traduction récursive de (φ_i) dans (ψ_i),* i.e. *pour tout i, $\varphi_i = \psi_{t(i)}$.*

Alors il existe une fonction récursive totale à deux arguments r telle que :

$$\theta_i(x) \leqslant r(x, \mu_{t(i)}(x)) \text{ et } \mu_{t(i)}(x) \leqslant r(x, \theta_i(x))$$

pour presque tout x[3].

Preuve. On considère la fonction h définie par :

$$(i, x, y) \mapsto \begin{cases} \max(\theta_i(x), \mu_{t(i)}(x)) & \text{si } \theta_i(x) = y \text{ ou } \mu_{t(i)}(x) = y \\ 0 & \text{sinon} \end{cases}$$

h est bien définie et elle est totale calculable. On pose alors

$$r(x, y) = \max_{i \leqslant x}(\max_{z \leqslant y}(h(i, x, z)))$$

□

On en déduit rapidement la proposition suivante, qui majore le résultat d'un calcul en fonction de sa complexité. C'est en fait un cas particulier en prenant un seul et même SAP muni d'une seule et même mesure de complexité abstraite.

Proposition 2.10. *Pour toute mesure de complexité abstraite (θ_i) relative à un SAP (φ_i), il existe une fonction totale récursive r telle que pour tout i*

$$\varphi_i(x) \leqslant r(x, \theta_i(x)) \text{ presque partout.}$$

On va maintenant prouver qu'il existe beaucoup de fonctions ne pouvant être des mesures de complexité. Ce résultat, démontré indépendamment par Trakhtenbrot en 1964 puis Borodin en 1972, est appelé le théorème de la lacune.

Théorème 2.11 (Gap theorem)**.** *Soit (θ_i) une mesure de complexité abstraite. Soit g une fonction totale récursive à deux arguments telle que $y < g(x, y)$ pour presque tous x et y.*

Alors il existe une fonction récursive totale t, récursivement constructible, telle que pour tout i et tout x, si $t(x) < \theta_i(x) < g(x, t(x))$ alors $x \leqslant i$.

Preuve. La condition du théorème est équivalente à la suivante :

$$i < x \implies \theta_i(x) \leqslant t(x) \text{ ou } g(x, t(x)) \leqslant \theta_i(x)$$

Posons $A_x = \{y | \forall\, i,\ i < x \implies \theta_i(x) \leqslant y \text{ ou } g(x, y) \leqslant \theta_i(x)\}$, et $t(x) = \min A_x$.
Si $A_x \neq \emptyset$, t est bien définie, totale et récursive. On va alors montrer que A_x est non vide.
Pour cela, considérons la suite $(y_n)_{n \in \mathbb{N}}$ définie récursivement par

$$y_0 = 0,\ y_{k+1} = g(x, y_k)$$

La suite $(y_n)_{n \in \mathbb{N}}$ est strictement croissante. De plus, il y a au plus x valeurs de $\theta_i(x)$ donc il existe j, $0 \leqslant j \leqslant x + 1$, tel que $\theta_i(x) \leqslant y_j$ ou $g(x, y_j) \leqslant \theta_i(x)$. □

[2] C'est-à-dire récursivement :-p.
[3] *Id est* pour tout x sauf un nombre fini.

Il n'existe pas systématiquement de borne inférieure pour la complexité d'une fonction donnée. Ce résultat est appelé théorème de compression.

Théorème 2.12 (Théorème de compression)**.** *Soit (θ_i) une mesure de complexité abstraite. Il existe des fonctions totales récursives f et g telles que pour tout i et tout x, si $\theta_i(x)$ est défini, alors $\varphi_{f(i)}(x)$ l'est également et vérifie $\varphi_{f(i)}(x) \leqslant x$, et telles que*

(i) pour tout j, si $\varphi_j = \varphi_{f(i)}$, alors $\theta_j(x) \geqslant \theta_i(x)$ presque partout,

(ii) $\theta_{f(i)}(x) \leqslant g(x, \theta_i(x))$ presque partout.

Preuve. On introduit $C(i,x) = \{j | j < x \text{ et } \theta_j(x) \leqslant \theta_i(x)\}$. On définit ensuite la fonction φ_a de la façon suivante :

$$\varphi_a : \langle i, x\rangle \mapsto \begin{cases} \min\{y | \forall\, j \in C(i,x),\ y \neq \varphi_j(x)\} & \text{si } \varphi_i(x) \text{ est défini} \\ \text{Non défini} & \text{sinon} \end{cases}$$

Si $\varphi_i(x)$ est défini, $\theta_i(x)$ l'est également. De plus, $C(i,x)$ étant un ensemble fini bien défini, $\{y | \forall\, j \in C(i,x),\ y \neq \varphi_j(x)\}$ est infini, donc en particulier non vide. Donc φ_a est bien définie et calculable. Par le théorème *s-n-m*, on déduit que $\varphi_a(\langle i, x\rangle) = \varphi_{s_1^1(\langle a,x\rangle)}(x)$. On définit alors f par $f(x) = s_1^1(\langle a, x\rangle)$.

Ceci définit bien f, et assure la relation $\varphi_{f(i)}(x) \leqslant x$. Il faut maintenant vérifier les deux autres conditions du théorème.

(i) est satisfaite : si $\varphi_j = \varphi_{f(i)}$ alors partout où ces fonctions sont définies, on a $\varphi_j(x) = \varphi_{f(i)}(x)$. Donc $j \notin C(i,x)$. Comme $j < x$, $\theta_j(x) > \theta_i(x)$ presque partout.

(ii) est satisfaite. On construit g :

$$\text{Posons } h(i,x,y) = \begin{cases} \theta_{f(i)}(x) & \text{si } \theta_i(x) = y \\ 0 & \text{sinon} \end{cases}$$

h est bien définie, totale et récursive. Il suffit alors de poser

$$g(x,y) = \max_{i \leqslant x} h(i,x,y)$$

et ça marche !

□

Le résultat suivant, appelé *théorème d'accélération* est dû à Blum. Il stipule que certaines fonctions peuvent être accélérées autant que l'on veut, et le nombre de fois que l'on veut. Sa démonstration étant longue et un peu techinque, elle sera omise ici.

Théorème 2.13 (Speed-up theorem)**.** *Pour toute mesure de complexité abstraite $(\theta_i)_i$ et toute fonction récursive totale g telle que pour tout x et tout y, $g(x,y) \leqslant g(x, y+1)$, il existe une fonction récursive totale f telle que pour tout x, $f(x) \leqslant x$ et telle que si $f = \varphi_i$, alors il existe j tel que $\varphi_j = f$ et $g(x, \theta_j(x)) \leqslant \theta_i(x)$ presque partout.*

Chapitre 3

P et NP

Ce chapitre a pour objet l'étude des deux classes de complexité certainement les plus connues. En particulier, le problème de savoir si $\mathsf{P} = \mathsf{NP}$ est l'une des plus importantes questions ouvertes en complexité et en informatique en général, si ce n'est la plus importante. La supposition que ces deux classes sont différentes, ou plus exactement le fait qu'on ne sait actuellement pas résoudre les problèmes de NP en temps polynomial est à la base de toute la théorie de la cryptographie, et de tous les systèmes de cryptage actuels. Un autre exemple prouvant l'importance de ce problème est le fait qu'il apparaisse dans les problèmes du millénaire de la fondation Clay, doté d'un prix d'un million de dollars.

Le but de ce chapitre n'est bien entendu pas de résoudre cette question ouverte, mais de donner le plus précisément possible la structure de ces deux classes. On étudiera également la classe complémentaire de NP, appelée CoNP. On verra également une conjecture qui, si elle était vérifiée, montrerait que $\mathsf{P} \neq \mathsf{NP}$. Cette conjecture fait partie des nombreux problèmes équivalents au problème $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$.

3.1 La classe P

Définition 3.1.

$$\mathsf{P} = \bigcup_{k \geqslant 1} \mathsf{DTIME}(n^k)$$

Un exemple de problème dans P est le test de primalité : le nombre n donné en entrée est-il premier ? L'appartenance de ce problème à la classe P a été prouvée en août 2002 par Agrawal, Kayal et Saxena. Leur algorithme AKS s'effectue en $\mathcal{O}(\log(n)^{12})$. La borne a été depuis améliorée, mais c'est l'appartenance à P de ce problème réputé difficile qui est importante.

On démontre une première propriété d'inclusion de deux classes. Bien que ce soit un résultat facile, il est intéressant de le mentionner et de le démontrer effectivement, car l'idée utilisée l'est souvent dans ce type de résultat. Il se trouve de plus que l'on énoncera dans ce chapitre mais surtout dans les chapitres suivants une foultitude de résultats de ce genre, dont certaines démonstrations seront laissées à la sagacité du lecteur. En cas de blocage, il ne faudra pas hésiter à se souvenir de cette courte démonstration !

Définition 3.2.

$$\mathsf{LOGSPACE} = \mathsf{DSPACE}(\log n)$$

Proposition 3.3.

$$\mathsf{LOGSPACE} \subseteq \mathsf{P}$$

Preuve. Si $L \in \mathsf{LOGSPACE}$, il existe une machine de Turing M qui le décide en espace $\log n$. Le temps d'exécution de cette machine est au plus de $2^{c \log n}$ (théorème 1.14). □

On démontrera dans la suite que $\mathsf{DTIME}(n^i) \subsetneq \mathsf{DTIME}(n^j)$ pour $i < j$.

Mais peut-on en dire plus sur la structure de P ? La méthode générale et classique[1] consiste à définir une relation de pré-ordre[2] sur la classe P. Cette relation donne canoniquement une relation d'équivalence, et qui induit un ordre sur les classes. La première idée consiste à définir la réduction polynomiale $\leqslant^p$.

Définition 3.4. On dit que A se réduit polynomialement à B, et on note $A \leqslant^p B$, s'il existe une fonction totale récursive f, calculable en temps polynomial, telle que $x \in A \iff f(x) \in B$.

Le problème de cette réduction est qu'elle ne donne rien sur P ! En fait, on considère sur P la relation $\leqslant^{\log}$ dite de réduction en *espace logarithmique*.

Définition 3.5. On dit que A se réduit à B en espace logarithmique, et on note $A \leqslant^{\log} B$, s'il existe une fonction totale récursive f, calculable en espace logarithmique, telle que $x \in A \iff f(x) \in B$.

Remarque (fondamentale). Le modèle de machine de Turing qui sert à cette évaluation est le suivant : un ruban d'entrée de pure lecture, une ruban de sortie de pure écriture, la tête ne se déplaçant que vers la droite, et des rubans de travail sur lesquels on évalue la complexité.

Proposition 3.6. *$\leqslant^p$ et $\leqslant^{\log}$ sont des pré-ordres.*

Preuve. La réflexivité de chacune de ces deux relations est évidentes. La transitivité exige une démonstration. Pour cela on prend les notations suivantes. On suppose qu'on a trois problèmes A, B et C tels que $A \leqslant^p B$ et $B \leqslant^p C$. On note f la fonction qui réalise la réduction polynomiale entre A et B et M_1 la machine qui implémente f, et g et M_2 la fonction et la machine pour B et C. On va construire la machine M_3 qui montrera que $A \leqslant^p C$. On adopte les mêmes notations pour la réduction en espace logarithmique.

$A \leqslant^p B$ et $B \leqslant^p C$ signifie qu'on passe de A à B par f et de B à C par g. Donc on passe de A à C par $g \circ f$. Donc $\leqslant^p$ est transitive.

La même preuve ne fonctionne pas pour $\leqslant^{\log}$ car le ruban de sortie de M_1 n'a pas une taille logarithmique. Puisqu'il porte $f(x)$ si $x \in A$, sa longueur est bornée par $2^{c \log n} = n^c$. On peut donc représenter une position sur ce ruban par un mot en $\mathcal{O}(\log n)$.

On va construire M_3 avec un ruban d'entrée, un ruban de travail R_{pos} contenant l'expression de ces positions, un ruban de sortie et les rubans de travail de M_1 et M_2.

M_3 fonctionne de la façon suivante :

- Si M_2 doit viser le i^{e} symbole de $f(x)$, M_3 contient sur le ruban R_{pos} l'expression de i et simule le calcul de M_1 sur x jusqu'à obtenir ce i^{e} symbole sur lequel M_3 simule le fonctionnement de M_2.
- Ensuite, M_3 ajuste son état en fonction de la transition de M_2 et modifie le ruban des positions en la représentation de $(i-1)$ ou $(i+1)$.

□

On peut prouver de manière analogue la proposition suivante.

Proposition 3.7. *Pour tout k, si $L \in \mathsf{DSPACE}(\log^k n)$ et $L' \leqslant^{\log} L$, alors $L' \in \mathsf{DSPACE}(\log^k n)$.*

Finalement, on considérant la relation d'équivalence[3] $\equiv^{\log}$, on obtient des classes, un ordre sur ces classes qui a un plus petit élément qui est LOGSPACE et un plus grand élément qui est… la classe des langages P-complets.

Définition 3.8. Un langage L est dit P-complet si

[1] Et efficace de surcroît !

[2] Un pré ordre est une relation réflexive et transitive.

[3] $A \equiv^{\log} B \iff A \leqslant^{\log} B$ et $B \leqslant^{\log} A$.

(i) $L \in \mathsf{P}$,

(ii) pour tout $A \in \mathsf{P}$, $A \leqslant^{\log} L$.

Proposition 3.9. LOGSPACE *est le plus petit élément de l'ordre induit par* $\leqslant^{\log}$ *sur les classes d'équivalences de* $\equiv^{\log}$.

Preuve. LOGSPACE est une classe d'équivalence : soit A et B dans LOGSPACE, on définit la fonction f qui permet d'affirmer que $A \leqslant^{\log} B$. Pour cela, pour une entrée x, on teste l'appartenance de x à A, et selon la réponse, on renvoie un élément de B ou un élément qui n'est pas dans B. Puisque A est dans LOGSPACE, f fonctionne bien en espace logarithmique. De même, on définit g qui permet de passer de B à A. Ceci prouve que $A \equiv^{\log} B$. De plus, tout problème équivalent à un problème de LOGSPACE est bien dans LOGSPACE avec la même construction.

Maintenant que l'on sait que c'est une classe d'équivalence, il suffit de remarquer que c'est le plus petit élément de l'ordre induit. En effet, si A est un problème quelconque et $B \in$ LOGSPACE, $A \leqslant^{\log} B$ implique que A est dans LOGSPACE. Pour savoir si $x \in A$, on calcule $f(x)$ où f est le témoin de la relation entre les deux problèmes, on décide si $f(x) \in B$, et on répond en fonction de la réponse pour $f(x)$. Tout cela se fait bien en espace logarithmique. □

La figure **??** résume la structure obtenue pour P. La partie centrale est inconnue à ce jour : on ne sait pas s'il existe des problèmes ou non, et si oui, le nombre qu'il y en a.

3.2 Machine de Turing non déterministe

On définit ce qu'est une machine de Turing non déterministe *à un ruban* pour des questions de simplicité. Cette définition s'étend bien entendu de la même manière que dans le cas des machines déterministes à des machines à plusieurs rubans.

Définition 3.10. Une machine de Turing non déterministe est un septuplet $M = (Q, \Sigma, \Gamma, B, \delta, q_0, F)$ où

- Q est l'ensemble fini des états ;
- Σ est un alphabet fini ;
- Γ est l'alphabet de travail ($\Sigma \subsetneq \Gamma$) ;
- B est un symbole dit « blanc » tel que $B \in \Gamma$ et $B \notin \Sigma$;
- $q_0 \in Q$ est l'état initial ;
- $\delta : Q \times \Sigma \to \mathcal{P}(Q \times \Gamma \times \{G, D, S\})$ est la fonction de transition ;
- $F \subseteq Q$ est un ensemble d'états dits « d'arrêt ».

Cette définition est essentiellement similaire à celle d'une machine de Turing déterministe. Le seul changement concerne la fonction de transition. Elle est à valeurs dans l'ensemble des parties de $Q \times \Gamma \times \{G, D, S\}$, c'est-à-dire qu'à partir de chaque configurations, plusieurs évolutions sont possibles. Il est équivalent de remplacer cette fonction de transition par une relation de transition

$$\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{G, D, S\})$$

indiquant les transitions autorisées.

On a maintenant un arbre de calcul sur une entrée. Cet arbre peut avoir des branches finies et possiblement des branches infinies.

On dit qu'un mot $u \in \Sigma^*$ est reconnu par une machine de Turing non déterministe M s'il existe un calcul acceptant de M sur u, c'est-à-dire une branche finie dans l'arbre des calcul qui termine sur un état acceptant.

On note $L(M)$ le langage reconnu par M, *i.e.* l'ensemble des mots sur Σ sur lesquels M réussit un calcul acceptant.

Une fonction partielle f est dite calculée par une machine de Turing non déterministe M lorsque

(i) M entre dans un état d'arrêt sur u si et seulement si $u \in \mathrm{dom} f$,

(ii) sur tout $u \in \mathrm{dom} f$, il existe un calcul de M qui donne $f(u)$,

(iii) tout calcul sur $u \in \mathrm{dom} f$ aboutissant à un état d'arrêt sort $f(u)$.

On définit maintenant la complexité en temps d'une machine de Turing non déterministe.

Définition 3.11. La complexité en temps d'une machine de Turing non déterministe est définie par les deux fonctions suivantes :

$$t_M : \begin{cases} \Sigma^* & \to \mathbb{N} \\ n & \mapsto \min \{\text{nombre de transitions pour que } M \text{ aboutisse à un état acceptant sur } u\} \end{cases}$$

si ce minimum existe. Sinon, $t_M(u)$ n'est pas défini.

$$T_M : \begin{cases} \mathbb{N} & \to \mathbb{N} \\ n & \mapsto \max \{t_M(u) \mid |u| = n\} \end{cases}$$

si ce maximum existe. Sinon, $T_M(n)$ n'est pas défini.

La complexité en temps d'un machine de Turing non déterministe est alors $n \mapsto \max(T_M(n), n+1)$.

Les définitions sont analogues pour l'espace.

Proposition 3.12. *Si L est reconnu par une machine de Turing non déterministe M, il est reconnu par une machine de Turing déterministe M'. De plus, si M travaille en temps $t(n)$, M' travaille en temps au plus $2^{\mathcal{O}(t(n))}$.*

Preuve. Soit M une machine de Turing non déterministe reconnaissant L. On construit M' à trois rubans :
- un ruban d'entrée E,
- un ruban « chemin de calculs » C,
- un ruban pour simuler M sur le chemin inscrit sur le ruban C.

On définit $m = \max |\{\delta(q,a) \mid (q,a) \in \mathrm{dom}\delta\}|$.

On considère $\Gamma = \{1, \ldots, m\}$; Γ^* est bien ordonné par l'ordre hiérarchique. Ce sont ses éléments que l'on énumère sur C.

Fonctionnement de M′ : sur l'entrée x, M' énumère successivement les éléments de Γ^* en les effaçant. Puis M' simule le fonctionnement de M sur le chemin indiqué par C, si c'est possible. Sinon, elle considère le successeur de C[4]. Elle s'arrête dans un état acceptant si M s'arrête dans un état acceptant. □

3.3 La classe NP

Définition 3.13.

$$\mathsf{NP} = \bigcup_{k \geqslant 1} \mathsf{NTIME}(n^k)$$

où $\mathsf{NTIME}(f)$ est la classe des langages reconnus en temps f par une machine de Turing non déterministe.

La définition bien connue de NP étant donnée, on donne une première caractérisation de cette classe de complexité. Cette caractérisation est en fait plus générale que ce cas précis, ce que l'on verra dans un chapitre suivant sous le nom de hiérarchie polynomiale.

[4] On confond ici le ruban et ce qu'il contient.

Proposition 3.14. *Un langage L est dans* NP *si et seulement s'il existe un polynôme p et un langage $B \in \mathsf{P}$ tels que*

$$x \in L \iff \exists\, y,\ |y| \leqslant p(|x|), \langle x, y\rangle \in B.$$

Preuve. Si $L \in \mathsf{NP}$, il existe une machine de Turing non déterministe qui le reconnaît en temps polynomial p. Et $x \in L$ si et seulement s'il existe un chemin acceptant dans l'arbre des calculs de M sur x. On choisit pour B le langage suivant, dont il est facilement vérifiable qu'il appartient à P.

$$B = \{\langle x, y\rangle \mid y \text{ code un chemin acceptant de } M \text{ sur } x\}$$

Réciproquement, supposons qu'il existe B et p vérifiant les conditions du théorème. On considère l'algorithme suivant : sur l'entrée x,

- on devine y tel que $|y| \leqslant p(|x|)$;
- on code $\langle x, y\rangle$;
- on teste l'appartenance de $\langle x, y\rangle$ à B.

□

On va maintenant s'intéresser de plus près à la structure de NP. On revient à la relation de pré-ordre $\leqslant^p$ et à la relation d'équivalence associée $\equiv^p$, et on montre que P est la plus petite classe de NP *modulo* $\equiv^p$ pour l'ordre induit sur les classes par $\leqslant^p$.

Proposition 3.15. *Soit $A \in \mathsf{P}$, alors pour tout $B \in \mathsf{NP}$, $A \leqslant^p B$.*

Preuve. On construit la fonction f qui à un élément de A renvoie un élément de B[5], et qui à un élément n'appartenant pas à A renvoie un élément n'appartenant pas à B. Il est clair qu'une telle fonction existe, et qu'elle est calculable en temps polynomial. Enfin, on a clairement l'équivalence

$$x \in A \iff f(x) \in B.$$

□

Et de plus, on sait qu'il existe un plus grand élément à cet ordre, que l'on appelle la classe des problèmes NP-complets.

Définition 3.16. Un langage L est dit NP-complet si

(i) $L \in \mathsf{NP}$,

(ii) pour tout $A \in \mathsf{NP}$, $A \leqslant^p L$.

Remarque. Si on a uniquement (ii), L est dit NP-dur.

Remarque. S'il existe L NP-complet tel que $L \in \mathsf{P}$, alors $\mathsf{P} = \mathsf{NP}$. De même, si L est P-complet et que $L \in \mathsf{LOGSPACE}$, alors $\mathsf{P} = \mathsf{LOGSPACE}$.

Preuve de la remarque. Trivialement, $\mathsf{P} \subseteq \mathsf{NP}$. Maintenant, si $A \in \mathsf{NP}$, comme L et NP-complet, $A \leqslant^p L$. Et comme $L \in \mathsf{P}$, nécessairement $A \in \mathsf{P}$. □

On va maintenant démontrer le fameux théorème de Cook[6] qui affirme l'existence d'un problème NP-complet. Pour ne pas verser dans trop de classicisme, nous démontrerons la NP-complétude non pas de SAT mais d'un autre problème. Le lecteur frustré trouvera très facilement la preuve pour le problème SAT dans la littérature.

Théorème 3.17 (Cook-Levin)**.** *Il existe des langages* NP*-complets.*

[5] Il nous est bien égal de savoir lequel !

[6] On devrait plutôt dire, *théorème de Cook-Levin*, Levin étant un logicien, élève de Kolmogorov, qui a exhibé au même moment que Cook un autre problème NP-complet. Mais étant en U.R.S.S., personne n'en a rien su !

Preuve. Le problème auquel on va s'intéresser s'appelle `MTND-T` :

$$\begin{cases} \text{Entrée} & : M \text{ une machine de Turing non déterministe, } t \in \mathbb{N},\ t \leqslant |Q| \\ \text{Question} & : M \text{ réalise-t-elle sur } \varepsilon \text{ un calcul de longueur inférieure à } t \text{ (pour le temps) ?} \end{cases}$$

ε représente ici le mot vide. Le langage associé à ce langage est

$$L = \{\langle M, t\rangle \mid M \text{ entre dans un état d'arrêt en au plus } t \text{ transitions.}\}$$

que nous allons montrer être NP-complet.

Tout d'abord, montrons que $L \in \mathsf{NP}$. On devine pour cela $\langle M, t\rangle$, on calcule à partir de cela M et t, et on simule M sur ε pendant au plus t transitions.

Prouvons maintenant que pour tout $A \in \mathsf{NP}$, $A \leqslant^p$ `MTND-T`. Pour cela, on montre l'existence d'une fonction calculable en temps polynomial telle que $x \in A$ si et seulement si $f(x) \in$ `MTND-T`.

Comme $A \in \mathsf{NP}$, il existe une machine de Turing non déterministe M qui le reconnaît en temps polynomial p. Considérons la fonction $x \mapsto \langle M_x, t_x\rangle$, où M_x est la machine convenable déduite de la machine M' suivante : sur l'entrée ε, M' écrit l'entrée x, revient en configuration initiale pour M, et simule le fonctionnement de M sur x. M' a besoin des états de M, ainsi qu'un état initial et des états nécessaires pour écrire x. Cette machine a donc $2|x| + p(|x|) + 2$ états. M_x est obtenue à partir de M' en ajoutant à M' autant d'état que nécessaires pour que $2|x| + p(|x|) + 2 \leqslant |Q_{M_x}|$. □

On donne maintenant un exemple de problème NP-complet un peu différent. Il fait partie d'un type de problèmes NP-complets qui ont eus une certaine importance dans l'étude de NP, tout comme `SAT` et ses dérivés, ou les problèmes du type de `MTND-T` qui parlent de propriétés des machines de Turing.

Exemple 3.18 (pavage fini du plan)**.** On définit le problème du pavage fini du plan `PFP` comme suit :

$$\begin{cases} \text{Entrée} & : \begin{array}{l} - \text{ Un ensemble fini } C \text{ de couleurs, dont une couleur dite } \textit{blanche}. \\ - \text{ Un ensemble fini } T \text{ de tuiles carrées, dont les bords sont colorés par les éléments de } C, \text{ dont une tuile dont les quatre bords sont blancs, dite } \textit{tuile blanche}. \\ - \text{ Un entier } n \leqslant |C|. \end{array} \\ \text{Question} & : \text{Est-il possible de paver à l'aide de tuiles de } T \text{ un carré de taille } n \times n, \text{ par un pavage non trivial et valide ?} \end{cases}$$

On entend par pavage non trivial un pavage contenant au moins une tuile non blanche. Un pavage est dit valide si les bords de deux tuiles adjacentes ont toujours même couleur.

Proposition 3.19. *Le problème* `PFP` *est* NP*-complet.*

Preuve. Il est tout d'abord clair que `PFP` est dans NP. En effet, on peut vérifier très facilement, étant donné un pavage, qu'il est valide, non trivial et qu'il est contenu dans un carré de taille $n \times n$.

On va maintenant montrer qu'il est NP-difficile en se ramenant à `MTND-T`. On explicite en premier lieu la réduction d'une instance de `MTND-T` en une instance de `PFP`, puis on prouve que l'instance de `PFP` ainsi construite satisfait le problème si et seulement si l'instance de départ satisfaisait `MTND-T`.

Le pavage construit va représenter l'évolution d'une machine de Turing non déterministe sur un calcul terminant. Informellement, si on considère notre carré $n \times n$ comme un ensemble

de n lignes de longueur n les unes au-dessus des autres, la ligne tout en bas représentera la configuration initiale de notre machine, au dessus d'elle se trouvera la configuration de la machine après une étape, et ainsi de suite jusqu'à la configuration finale. Plus précisément, la configuration de la machine à l'état t est codée par la séquence des couleurs se trouvant en haut de chaque tuile de la ligne t du pavage (la ligne 0 étant celle du bas). Pour un calcul complet en t étapes, notre pavage aura donc une hauteur $t+1$, avec une dernière ligne fermant le pavage. Sa largeur sera par conséquent également $t+1$, ce qui suffit car en t étapes, une machine ne peut visiter au plus que t cases.

Il faut maintenant expliciter comment est codée une configuration de la machine par une séquence de couleurs. Une séquence de couleurs représente le ruban de notre machine de Turing, chaque tuile représentant une case du ruban. Il faut représenter en plus l'état q de la machine, ainsi que la position de la tête de lecture. Ceci peut-être fait car rien n'oblige les tuiles à ne porter comme information que la valeur du ruban. Ainsi, pour signifier que la tête de lecture est au dessus de la case c qui contient le caractère α, et que l'état courant est q, la couleur du haut de la tuile représentant c ne sera plus simplement α mais (q, α). Cette tuile étant la seule de sa ligne à avoir comme couleur du haut un couple, on sait que la tête de lecture se trouve à cet emplacement.

Maintenant que les idées sont en place, il suffit de décrire formellement la réduction. On considère donc une instance de `MTND-T`, c'est-à-dire une machine de Turing non déterministe M et un entier t. Le jeu de tuile est décrit par la figure 3.1, en rajoutant bien entendu la tuile blanche. Il y a quatre sortes de tuiles, et on retrouve dans chaque tuile des tuiles spéciales, qui permettent de délimiter les bords du pavages. Les traits dessinés sur les tuiles ne sont là que pour aider à la compréhension, les couleurs étant représentées par des chaînes de caractères. La couleur blanche est représentée par la chaîne vide, c'est-à-dire par l'absence d'écriture sur la tuile.

Maintenant que le jeu de tuile a été présenté, il ne reste plus qu'à prouver que la machine M effectue un calcul de longueur inférieure à t si et seulement si le jeu de tuile permet de paver le plan $(t+1) \times (t+1)$ de façon non triviale.

Supposons qu'un tel pavage existe. Les tuiles d'initialisation étant séparées en tuiles *de gauche* et tuiles *de droite*, et la tuile de l'état initial étant la seule à pouvoir faire la liaison entre ces deux typs de tuiles, on est assuré que cet état initial est présent sur la première ligne. Ceci évite d'avoir comme pavage un carré, valide, non trivial, mais ne représentant qu'un ruban blanc sans évolution. Cet état initial étant en place, on est assuré d'avoir des tuiles de transition, et ce à chaque ligne jusqu'à l'état final. En effet, mises à part les tuiles des états initial et final, seules les tuiles de transition possède des couples comme couleur. Ainsi, le pavage est forcément de la même forme que celui de la figure 3.2. Il suffit alors de « lire » sur le pavage l'évolution de la machine. Cette évolution démarre bien de l'état initial, seul état pouvant se trouver à la base du pavage, et se termine bien par l'état final, seul état pouvant se trouver en haut du pavage.

Réciproquement, si on a un calcul de longueur inférieure à t, ce calcul décrit une manière de paver le carré. Pour cela, il suffit de recopier les configurations successives avec le codage explicité préalablement, et de compléter par des tuiles blanches.

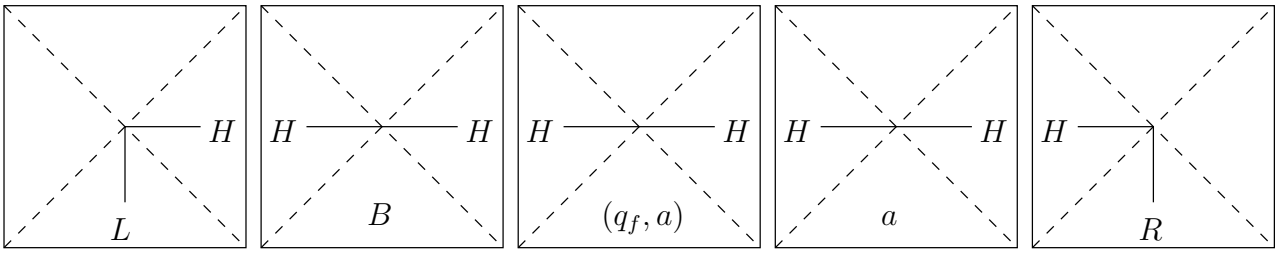
□

3.4 Conjecture de Hartmanis-Berman

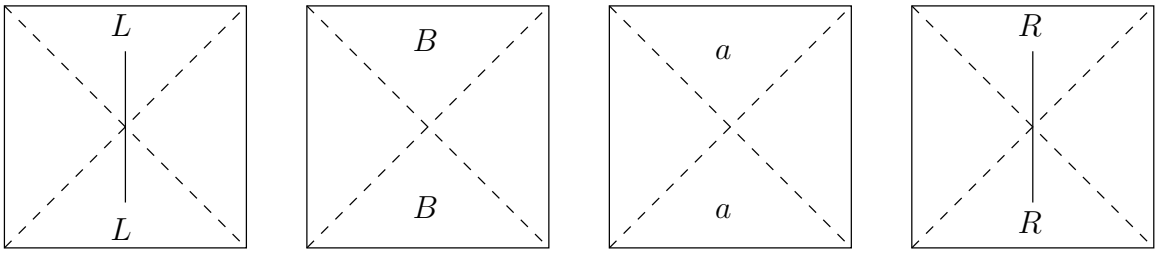
Cette conjecture concerne la structure interne de NP. Mais si elle est vérifiée, alors nécessairement $\mathsf{P} \neq \mathsf{NP}$.

Conjecture 3.20 (Hartmanis-Berman, 1977)**.** *Les langages* NP*-complets sont p-isomorphes deux à deux.*

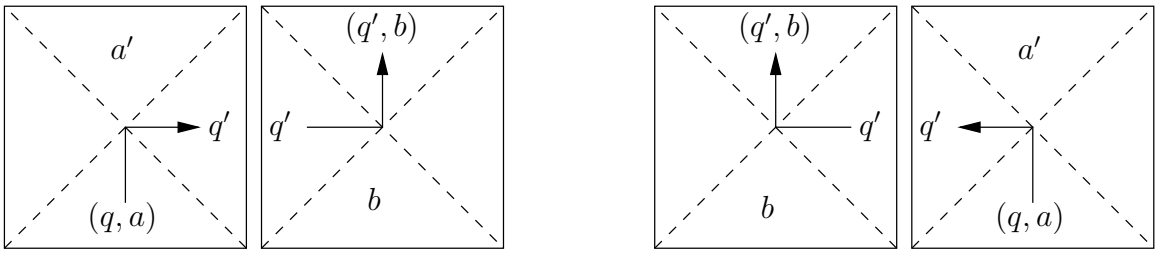
Définition 3.21. Deux langages L et L' sont dits p-isomorphes s'il existe une fonction f bijective,



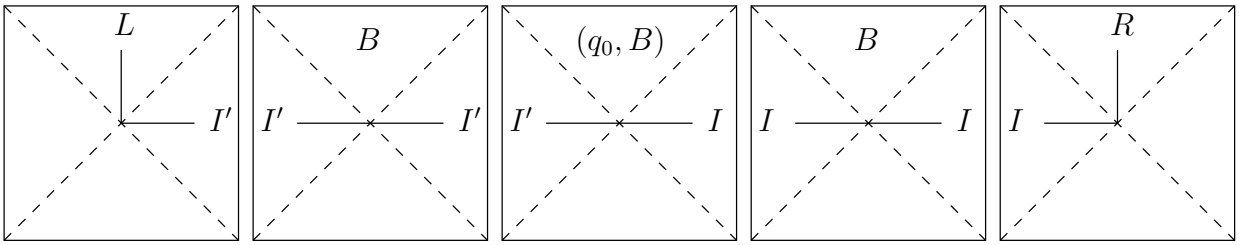
(a) Tuiles d'arrêt



(b) Tuiles d'information



(c) Tuiles de transition



(d) Tuiles d'initialisation

FIG. 3.1: Le jeu de tuiles, auquel il faut ajouter la tuile blanche.

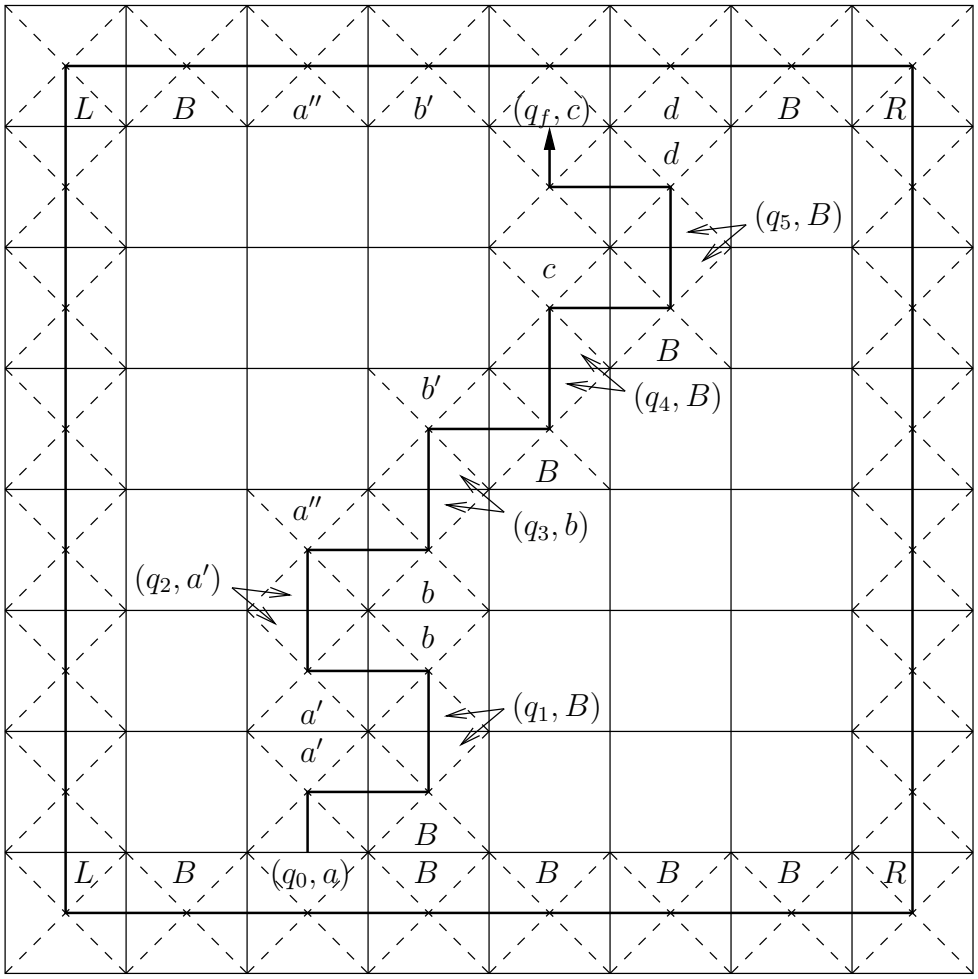


FIG. 3.2: Exemple de pavage

calculable en temps polynomial, qui réalise $L \leqslant^p L'$ et telle que f^{-1} est également calculable en temps polynomial et réalise $L' \leqslant^p L$.

On va maintenant donner des exemples de langages p-isomorphes.

$$\texttt{IS} \text{ (Independant Set)} = \begin{cases} \text{Entrée} & : G = (V, E) \text{ un graphe fini et } k \geqslant 0 \\ \text{Question} & : G \text{ a-t-il un ensemble indépendant de taille au moins } k\,? \end{cases}$$

Un ensemble indépendant est un ensemble $V' \subseteq V$ tel que pour tous x et x' dans V', $\{x, x'\} \notin E$.

$$\texttt{VC} \text{ (Vertex Cover)} = \begin{cases} \text{Entrée} & : G = (V, E) \text{ un graphe fini et } k \geqslant 0 \\ \text{Question} & : G \text{ a-t-il une couverture de sommets de taille au plus } k\,? \end{cases}$$

Une couverture est un ensemble $V' \subseteq V$ tel que pour tout $\{x, x'\} \in E$, x ou x' est dans E.

$$\texttt{Clique} = \begin{cases} \text{Entrée} & : G = (V, E) \text{ un graphe fini et } k \geqslant 0 \\ \text{Question} & : \text{Existe-t-il une clique de taille } k \text{ pour } G\,? \end{cases}$$

Lemme 3.22. *`Clique` et `IS` sont p-isomorphes.*

Preuve. Il suffit d'utiliser la fonction suivante :

$$\begin{array}{rcl} \texttt{IS} & \to & \texttt{Clique} \\ (G = (V, E), k) & \mapsto & (G^c = (V, E'), k) \end{array}$$

où $\{x, y\} \in E' \iff \{x, y\} \notin E$. □

On peut donner une autre caractérisation du problème $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ en terme d'existence d'une fonction particulière.

Proposition 3.23. $\mathsf{P} \neq \mathsf{NP} \iff$ *Il existe une fonction* à sens unique.

Définitions 3.24.

- Une fonction f (possiblement partielle) est dite *honnête* s'il existe un polynôme q tel que $\forall\, y \in Imf, \exists\, x, y = f(x)$ et $|x| \leqslant q(|y|)$.
- Une fonction f (possiblement partielle) est dite *p-inversible* s'il existe une fonction g (possiblement partielle) calculable en temps polynomial et telle que pour tout $y \in Imf$, $g(y)$ existe, $g(y) \in Domf$ et $f(g(y)) = y$.
- Une fonction est dite *à sens unique si*
  - (i) elle est honnête,
  - (ii) elle est calculable en temps polynomial,
  - (iii) elle n'est pas inversible.

Preuve. On montre d'abord le sens réciproque. Supposons qu'il existe une fonction à sens unique. On exhibe un langage de NP qui n'appartient pas à P. A cette fin, considérons $Pref_{inv}(f) = \{\langle y, t\rangle \mid \text{il existe } x \text{ tel que } |x| + |t| \leqslant q(|y|) \text{ et } f(tx) = y\}$, où q est un polynôme témoin de ce que f est honnête.

$Pref_{inv}(f) \in \mathsf{NP}$: on devine x tel que $|x| + |t| \leqslant q(|y|)$ et on calcule éventuellement $f(tx) = y$.

$Pref_{inv}(f) \notin \mathsf{P}$: on montre en fait que si f est calculable en temps polynomial et honnête, et que $Pref_{inv}(f) \in \mathsf{P}$, alors f est p-inversible, en construisant un *p-inverse* g. On part de y. Considérons $\langle y, \varepsilon\rangle$. On sait décider en temps polynomial si $\langle y, \varepsilon\rangle \in Pref_{inv}(f)$. Si non, $y \notin Imf$. Si oui, on sait calculer $f(\varepsilon)$ en temps polynomial. Et si $y = f(\varepsilon)$, on pose $g(y) = \varepsilon$.

Si $y \neq f(\varepsilon)$, c'est qu'il existe $x \neq \varepsilon$ tel que $f(x) = y$. Cet élément x commence par 0 ou par 1. On s'interroge sur l'appartenance de $\langle y, 0\rangle$ ou $\langle y, 1\rangle$ à $Pref_{inv}(f)$. L'un ou l'autre lui appartient, supposons par exemple qu'il s'agisse de $\langle y, 0\rangle$. On calcule $f(0)$. Si $f(0) = y$, on pose $g(y) = 0$. Sinon, on continue à examiner $\langle y, 00\rangle$, $\langle y, 01\rangle$, ...

Le processus étant défini, $g(y)$ est bien défini.

On s'intéresse maintenant au sens direct. Supposons que $\mathsf{P} \neq \mathsf{NP}$. Montrons qu'il existe alors une fonction à sens unique. Puisque $\mathsf{P} \neq \mathsf{NP}$, il existe A tel que $A \in \mathsf{NP}$ et $A \notin \mathsf{P}$. Puisque $A \in \mathsf{NP}$, il existe un polynôme p et un langage $B \in \mathsf{P}$ tels que $x \in A \iff \exists\, y(|y| \leqslant p(|x|)$ et $\langle x, y\rangle \in B)$.

On définit f de la façon suivante :

$$\langle x, w\rangle \mapsto \begin{cases} 0x & \text{si } \langle x, w\rangle \in B \\ 1x & \text{sinon.} \end{cases}$$

f est bien définie, et calculable en temps polynomial. On vérifie alors qu'elle est à sens unique.

- f est honnête : il existe un polynôme r tel que pour tout $z \in Im(f)$, il existe x tel que $|x| \leqslant r(z)$ et $z = f(x)$. Si $z \in Im(f)$, alors z s'écrit soit $0y$ s'il existe w tel que $z = f(\langle y, w\rangle)$ avec $\langle y, w\rangle \in B$, soit $1y$ s'il existe w tel que $z = f(\langle y, w\rangle)$ avec $\langle y, w\rangle \notin B$.

$$|\langle y, w\rangle| \leqslant |y| + |w| \leqslant |y| + p(|y|)$$

- f n'est pas p-inversible : raisonnons par l'absurde et montrons que dans le cas inverse, A serait dans P. Supposons qu'il existe un « p-inverse » de f, que l'on note g. Alors pour tout $y \in \Sigma^*$, on sait calculer $g(0y)$ en un temps inférieur à $r(|0y|)$ où r est le polynôme témoignant du fait que g est calculable en temps polynomial.
  Si $g(0y)$ est défini, on calcule $f(g(Oy)) = 0y$ et donc $y \in A$. Si au contraire, $g(0y)$ n'est pas défini, $0y \notin Im(f)$, donc pour tout $w \in \Sigma^*$, $\langle y, w\rangle \notin B$, et donc $y \notin A$. Donc A est décidable en temps polynomial. C'est absurde.

□

3.5 La classe CoNP

On considère le classique problème `SAT`, dont la question est de savoir si une formule est satisfiable. On peut considérer le problème voisin `VALID` dont la question est de savoir si une formule est satisfaite par toute instanciation des variables. On dit qu'une formule φ est valide si pour toute valeur v donnée au variable, $\varphi(v) = 1$. Alors φ est non valide s'il existe une instanciation v des variables telle que $\varphi(v) = 0$, autrement dit s'il existe v telle que $(\neg\varphi)(v) = 1$. Mais ce problème est une instance de `SAT`, avec comme formule considérée $\neg\varphi$. On dit alors que `VALID`, dont la négation est dans NP, est dans la classe CoNP. On donne une définition précise de cette classe.

Définition 3.25.

$$\mathsf{CoNP} = \left\{ L \subset \Sigma^* \,\middle|\, \bar{L} \in \mathsf{NP} \right\}$$

Comme on l'a explicité préalablement, un exemple de problème dans CoNP est le problème `VALID`. On peut aussi remarquer que $\mathsf{P} \subset \mathsf{CoNP}$, de la même façon qu'on a $\mathsf{P} \subset \mathsf{NP}$. Une autre analogie avec la classe NP est l'existence dans CoNP d'une classe particulière de problèmes, plus durs que tous les autres. On appelle cette classe la classe des langages CoNP-complets.

Définition 3.26. Une langage L est dit CoNP-complet s'il appartient à CoNP et si pour tout langage $H \in \mathsf{CoNP}$, $H \leqslant^p L$.

Cette définition toute naturelle qu'elle soit demande tout de même une justification. Il faut en effet s'assurer que cette classe n'est pas vide. C'est encore `SAT` qui va nous donner la solution.

Exemple 3.27. VALID est CoNP-complet.

Preuve. Soit $L \in \mathsf{CoNP}$. Alors par définition, $\bar{L} \in \mathsf{NP}$ et donc $\bar{L} \leqslant^p \mathtt{SAT}$, car SAT est NP-complet. Soit f un témoin de cette inégalité. Par définition, $x \in \bar{L} \iff f(x) \in \mathtt{SAT}$. On a alors la suite d'équivalences suivante :

$$\begin{aligned} x \in L &\iff f(x) \notin \mathtt{SAT} \\ &\iff f(x) \text{ n'est pas satisfiable} \\ &\iff \forall\, v,\ f(x)(v) = 0 \\ &\iff \neg f(x) \in \mathtt{VALID} \end{aligned}$$

Donc $L \leqslant^p \mathtt{VALID}$, grâce à la fonction $x \mapsto \neg f(x)$. □

La méthode utilisée semble plus générale que le résultat démontré. En effet, elle l'est, et nous donne la caractérisation des problèmes CoNP-complets en fonction des problèmes NP-complets à laquelle on s'attend.

Proposition 3.28. *Soit L langage sur Σ^*. Alors*

$$L \textit{ est } \mathsf{NP}\textit{-complet} \iff \bar{L} \textit{ est } \mathsf{CoNP}\textit{-complet.}$$

Preuve. Montrons le sens direct. Soit pour cela L un langage NP-complet. Alors en particulier, $L \in \mathsf{NP}$ et par définition $\bar{L} \in \mathsf{CoNP}$. Soit alors K un langage quelconque de CoNP. Or $\bar{K} \in \mathsf{NP}$ et L est NP-complet, donc $\bar{K} \leqslant^p L$, et une certaine fonction f témoigne de cela. On reprend la même démonstration que dans l'exemple de la CoNP-complétude de VALID.

$$x \in \bar{K} \iff f(x) \in L \iff f(x) \notin \bar{L}$$

Mais on sait également que $x \in \bar{K}$ si et seulement si $x \notin K$. On a donc l'équivalence

$$x \notin K \iff f(x) \notin \bar{L},$$

d'où on déduit la variante positive équivalente, ce qui nous assure que

$$K \leqslant^p \bar{L}.$$

Le langage K ayant été choisi de manière totalement quelconque, cela suffit à assurer que $\bar{L}$ est CoNP-complet.

La démonstration réciproque se fait de manière absolument identique. □

On montre enfin deux propositions quasiment évidentes qui mettent en relation les classes NP et CoNP.

Proposition 3.29. *S'il existe $L \in \mathsf{NP}$ qui est* CoNP*-complet, alors* $\mathsf{NP} = \mathsf{CoNP}$.

Preuve. Montrons que sous cette hypothèse, $\mathsf{NP} \subseteq \mathsf{CoNP}$. Soit $H \in \mathsf{NP}$. Alors $\bar{H} \in \mathsf{CoNP}$, donc $\bar{H} \leqslant^p L$ car L est supposé CoNP-complet. Mais $L \in \mathsf{NP}$, donc ceci implique que $\bar{H} \in \mathsf{NP}$ et donc que $H \in \mathsf{CoNP}$.

L'autre inclusion se montre de manière similaire. □

Proposition 3.30. *La classe* NP *est close par complément si et seulement s'il existe un langage L* NP*-complet tel que $\bar{L}$ soit dans* NP.

Preuve. Le sens direct est clair. Supposons alors qu'il existe un tel langage L, et considérons un langage H dans NP. Alors L étant NP-complet, $H \leqslant^p L$, et par conséquent $\bar{H} \leqslant^p \bar{L}$. Mais $\bar{L}$ étant dans NP, cela suffit à conclure qu'il en est de même pour $\bar{H}$. □

Chapitre 4

La classe PSPACE

On s'intéresse dans ce chapitre à la classe PSPACE qui caractérise les langages reconnaissables en espace polynomial. On démontrera entre autres le théorème de Savitch dont une conséquence est l'égalité de NPSPACE et de PSPACE.

Définition 4.1.

$$\mathsf{PSPACE} = \bigcup_k \mathsf{DSPACE}(n^k)$$

4.1 Quelques exemples

On donne dans cette partie trois exemples de problèmes de PSPACE et NPSPACE. Tout d'abord le fameux SAT, puis $\overline{\mathtt{A_{ND}}}$ que l'on montrera appartenir à NPSPACE, et enfin le problème de l'accessibilité dans un graphe, qui sera un exemple d'application du théorème de Savitch qui sera vu dans la partie suivante.

1. SAT est décidable en espace linéaire. En effet, l'algorithme naïf consistant à tester toutes les valuations possibles fonctionne en espace linéaire.
2. $\overline{\mathtt{A_{ND}}} = \{\langle A\rangle \mid A \text{ est un automate fini non déterministe tel que } L(A) \neq \Sigma^*\}$. Ce langage est décidable et reconnu en espace non déterministe linéaire.

   On considère l'algorithme suivant :
   - On place un marqueur sur l'état initial de A.
   - Répéter au plus $2^{|Q_A|}$ fois :
     - Deviner un caractère de Σ.
     - Placer des marqueurs sur les états atteints.
     - Si un état d'acceptation est marqué, refuser l'entrée.
     - Sinon, accepter.

   Cet algorithme est non déterministe, et fonctionne bien en espace linéaire. On doit encore vérifier que $\overline{\mathtt{A_{ND}}} = L(M)$ si M est la machine de Turing non déterministe qui réalise l'algorithme précédent.
   - Si $\langle A\rangle \in L(M)$, on doit montrer que $L(A) \neq \Sigma^*$. Or $\langle A\rangle \in L(M)$ implique qu'il existe un calcul acceptant de M sur $\langle A\rangle$ qui exhibe un mot de longueur inférieure à $2^{|Q_A|}$ qui n'appartient pas à $L(A)$. Donc $L(A) \neq \Sigma^*$.
   - Inversement, supposons que $\langle A\rangle \in \overline{\mathtt{A_{ND}}}$. Alors il existe un mot $u \in \Sigma^*$ tel que $u \notin L(A)$. Si $u \geqslant 2^{|Q_A|}$, il existe certainement par le lemme de l'étoile un mot de longueur inférieure à $2^{|Q_A|}$ qui n'appartient pas à $L(A)$ et qui sert de témoin au fait que $\langle A\rangle \in L(M)$.
3. Accessibilité dans un graphe :

   $$\begin{cases} \langle G, s, t\rangle \text{ où } G \text{ est un graphe fini et } s \text{ et } t \text{ deux sommets.} \\ \text{Existe-t-il un chemin dans } G \text{ de } s \text{ à } t\,? \end{cases}$$

On donne l'entrée de la façon suivante : on donne le nombre d'états n, une matrice d'adjacence du graphe, et les sommets s et t. Il existe donc un algorithme facile en espace $\mathcal{O}(n \log n)$. On va montrer mieux. Il existe en effet un algorithme non déterministe en espace $\mathcal{O}(\log n)$, et un algorithme déterministe en espace $\mathcal{O}((\log n)^2)$.

Pour l'algorithme non déterministe, on énumère les sommets par les entiers entre 1 et n, en fixant $s = 1$ et $t = n$. On se donne deux rubans qui vont chacun contenir le code d'un sommet. L'espace utilisé sera donc toujours logarithmique. Le premier ruban contient le sommet courant i. Sur le second, on devine un sommet j qui est le successeur de i dans le chemin de s à t. En commençant par le sommet 1 (donc s) et en s'arrêtant dès qu'on est bloqué ou qu'on arrive au sommet n (donc t), on décide bien le problème de l'accessibilité dans le graphe.

L'algorithme déterministe repose sur une remarque : si on désigne par $chem(x, y, i)$ la propriété qui dit que y est accessible depuis x par un chemin de longueur inférieure à 2^i, alors

$$chem(x, y, i) \iff \exists\, z, chem(x, z, i-1) \wedge chem(z, y, i-1)$$

On en tire alors un algorithme récursif. Dans l'algorithme, i devra varier de 0 à $\log n$. A chaque étape, on va évaluer $chem(x, y, i)$, pour obtenir à la fin $chem(s, t, \log n)$.

– Si $i = 0$, vérifier si $x = y$.
– Si $i \geqslant 1$, évaluer pour tous les sommets z $chem(x, z, i-1)$ et $chem(z, y, i-1)$.

En mémoire, on doit garder

$$(x, y, i), (x, z_1, i-1), (x, z_2, i-2), \ldots, (x, z_i, 1), (z_i, z_{i+1}, 1)$$

Il y a $(i+1)$ triplets de taille chacun $\mathcal{O}(\log n)$. D'où une complexité en espace $\mathcal{O}((\log n)^2)$.

4.2 Théorème de Savitch

Le résultat démontré pour le problème d'accessibilité dans un graphe peut s'appliquer en réalité à beaucoup d'autres problèmes. C'est le sujet de ce théorème, qui a une conséquence très importante. Il permet de déduire l'égalité des classes de complexité PSPACE et NPSPACE.

Théorème 4.2. *Soit s une fonction telle que pour tout n, $s(n) \geqslant \log n$, Turing-calculable en espace $s(n)$. Alors*

$$\mathsf{NSPACE}(s(n)) \subseteq \mathsf{DSPACE}(s(n)^2)$$

Preuve. Soit $L \in \mathsf{NSPACE}(s(n))$. Il existe une machine de Turing non déterministe M qui reconnaît L de telle sorte qu'aucun calcul acceptant ne nécessite un espace strictement supérieur à $s(n)$. La longueur d'un chemin acceptant dans M est donc bornée par $2^{cs(n)}$ où c est une constante dépendant de M. Un chemin acceptant de M sur u est une suite de configurations.

$$c_0 = q_0 u \vdash c_1 \vdash \cdots \vdash c_{arrêt}$$

On peut transformer la machine M de telle manière que l'on ait une seule configuration d'arrêt.

Dans ces conditions, savoir si $u \in L(M)$ revient à savoir s'il existe un chemin de $c_0 = q_0 u$ à la configuration d'arrêt dans le graphe des configurations distinctes de M de longueur bornée par $cs(n)$.

D'où une complexité en $\mathcal{O}((\log 2^{cs(n)})^2)$ donc en $\mathcal{O}(s(n)^2)$. □

Corollaire 4.3. PSPACE = NPSPACE.

La preuve de ce corollaire est immédiate. Il suffit de prouver que NPSPACE ⊆ PSPACE. Or si $L \in \mathsf{NPSPACE}$, il existe un polynôme p tel que $L \in \mathsf{NSPACE}(p(n))$. Donc d'après le théorème 4.2, $L \in \mathsf{DSPACE}(p(n)^2)$. Et comme p^2 est un polynôme, $L \in$ PSPACE.

4.3 Existence de problèmes PSPACE-complets

On démontre ici un équivalent du théorème de Cook, mais pour la classe PSPACE. Le problème utilisé peut être vu comme une adaptation du célèbre `SAT` utilisé pour démontrer le théorème de Cook.

Théorème 4.4. *Il existe des langages* PSPACE*-complets, comme par exemple* `TQBF`*.*

Avant de démontrer ce théorème, il faut en expliciter les termes. On dit qu'un langage est PSPACE-complet s'il appartient à PSPACE et que tout problème de PSPACE se réduit polynomialement à ce dernier. On a donc une définition complètement similaire à la définition de NP-complétude, avec le même ordre de réduction polynomial, noté $\leqslant^p$. Le langage `TQBF` est le langage des formules booléennes totalement quantifiées vraies. C'est donc un sous-ensemble des formules de `SAT`, dont on a quantifié les variables.

Preuve. On remarque d'abord que `TQBF` est clairement dans PSPACE. Il reste maintenant à montrer qu'il est PSPACE-complet, c'est-à-dire que pour tout langage L dans PSPACE, on a $L \leqslant^p$ `TQBF`. On doit donc trouver une fonction f Turing-calculable en temps polynomial telle que $x \in L \iff f(x) \in$ `TQBF`.

On va en fait exprimer le fait que $x \in L$ compte tenu du fait que L est décidable par une machine de Turing M en espace polynomial. On cherche à écrire qu'il existe un chemin réussi dans M à partir de q_0x : $q_0x \vdash\!\!\!-^* c_{\text{accept}}$. Notons p le polynôme qui témoigne du fait que $L \in$ PSPACE. M effectue au plus $2^{cp(n)}$ pas de calcul avant de s'arrêter.

Pour c, c' et c'' des configurations, on définit le prédicat $\varphi(c, c', t)$ qui est vrai si et seulement s'il existe un chemin de c à c' de longueur inférieure à t. Pour cela on dit que $\varphi(c, c', 1)$ signifie que $c \vdash\!\!\!- c'$, et par induction,

$$\varphi(c, c', t) \iff \exists\, c'',\ \varphi(c, c'', t/2) \wedge \varphi(c'', c', t/2).$$

En remplaçant chaque formule à droite de l'équivalence par son équivalent, on obtient par induction une formule de taille $\mathcal{O}(2^t)$, ce qui ne nous convient pas. Le problème vient du fait que le nombre d'occurrence de φ est doublé à chaque étape. On va exprimer $\varphi(c, c', t)$ à l'aide d'une formule un peu différente, mais qui ne fait intervenir qu'une seule occurrence de φ. L'astuce consiste à exprimer les deux φ en un seul, mais avec des variables quantifiées :

$$\varphi(c, c', t) \iff \exists\, c'' \forall\, z_1 \forall\, z_2 \left((z_1 = c \wedge z_2 = c'') \vee (z_1 = c'' \wedge z_2 = c') \rightarrow \varphi(z_1, z_2, t/2)\right).$$

A chaque pas, on ne rajoute qu'une seule formule. Or puisqu'on diminue par deux la valeur de t à chaque étape, et que le temps maximal est en $2^{\mathcal{O}(p(n))}$, le nombre d'étape de calcul est en $\mathcal{O}(p(n))$. Chaque formule ayant une taille en $\mathcal{O}(p(n))$, on obtient finalement une formule de taille $\mathcal{O}(p(n)^2)$.

Pour $x \in L$, on pose alors $f(x) = \varphi(q_0x, c_{\text{accept}}, 2^{cp(|x|)})$. □

Ce dernier théorème nous donne déjà une structure de PSPACE relativement satisfaisante, mais on peut faire encore mieux !

4.4 Machines de Turing avec Oracle

La clé du succès pour définir une structure plus fine de PSPACE consiste à passer par les machines de Turing avec oracle. Celles-ci permettront de définir la notion de Turing-réduction, grâce à laquelle on peut en apprendre davantage sur la classe PSPACE.

Une *machine de Turing avec oracle* est une machine classique dans laquelle on peut dès que l'on souhaite interroger un *oracle*. Celui-ci nous permet d'avancer beaucoup plus vite dans un calcul. On imposera des limites dans l'utilisation que l'on fait de ces machines, car leur puissance est potentiellement beaucoup trop élevée. On exprime la définition formelle ci-après.

Définition 4.5. Une *machine de Turing avec oracle* est une machine de Turing *classique* à laquelle on ajoute un ruban, appelé *ruban d'oracle*, et trois nouveaux états notés (généralement) $q_?$, q_{oui} et q_{non}.

L'oracle est un langage A.

Le fonctionnement de la machine est le suivant :

- sur une entrée, la machine calcule de façon classique (en particulier, elle utilise le ruban d'oracle comme un ruban normal), jusqu'à entrer dans l'état d'interrogation $q_?$,
- de l'état $q_?$, elle passe en une transition soit dans l'état q_{oui}, soit dans l'état q_{non}, selon que le mot inscrit sur le ruban d'oracle appartienne ou non au langage d'oracle A,
- puis elle reprend son fonctionnement ordinaire, avec éventuellement d'autres passages par l'état d'interrogation de l'oracle.

Bien entendu, une machine de Turing avec oracle peut ne jamais utiliser son oracle, et fonctionne alors exactement de la même façon qu'une machine de Turing classique.

Exemple 4.6. Le langage $\left\{0^i1^i0^i1^j0^j1^j0 \,\middle|\, i,j \geqslant 1\right\}$ est reconnu par une machine de Turing avec l'oracle $\{a^nb^nc^n \,|\, n \geqslant 1\}$ en temps linéaire.

Comme on l'a dit en introduction, les machines de Turing avec oracle peuvent être extrêmement puissantes, allant jusqu'à reconnaître des langages indécidables ! On va donner un exemple d'un langage indécidable, reconnu par une machine de Turing avec oracle. Pour montrer la puissance de ces machines, on choisit même un langage non récursivement énumérable, ce qui montre un résultat encore plus fort. Cet exemple va motiver la définition de la notion de Turing-réduction, qui permet de contrôler un minimum l'utilisation que l'on fait de ces machines trop puissantes pour nous.

Considérons les langages L_u et L_v suivants :

$$\begin{aligned} L_u &= \{\langle M, w\rangle \,|\, M \text{ accepte } w\} \\ L_v &= \{\langle M\rangle \,|\, L(M) = \emptyset\} \end{aligned}$$

On va d'abord montrer que L_u est récursivement énumérable, mais non récursif.

- $L_u = \text{dom}\varphi_{\text{univ}}$ où $\varphi_{\text{univ}}(\langle M, w\rangle) = M(w)$ donc L_u est récursivement énumérable.
- L_u est non récursif est une conséquence directe du théorème de l'arrêt.

Ceci nous prouve donc que $\overline{L_u}$ n'est pas récursivement énumérable, d'après le théorème de Post.

Montrons maintenant que L_v n'est pas récursivement énumérable, en montrant que son complémentaire n'est pas récursif mais uniquement récursivement énumérable. L'application du théorème de Post nous donnera le résultat.

- On construit une énumération de $\overline{L_v}$ de la façon suivante : on énumère les crochets $\langle i, j\rangle$, et on simule M sur l'entrée i pendant au plus j étapes. Si M entre dans un état d'arrêt, l'entrée est acceptée, sinon, on passe au crochet suivant.
- $\overline{L_v} = \{\langle M\rangle \,|\, L(M) \neq \emptyset\}$ n'est pas récursif, par une application directe du théorème de Rice.

Le théorème de Post nous permet de conclure que L_v n'est pas récursivement énumérable.

Cependant, on va pouvoir reconnaître L_v grâce à une machine de Turing avec oracle.

Proposition 4.7. *L_v est reconnu par une machine de Turing avec l'oracle L_u.*

Preuve. Soit T la machine dont le fonctionnement est le suivant : sur l'entrée $\langle M\rangle$,

1. T détermine le code de la machine N_M suivante : sur l'entrée x, N_M énumère les $\langle i, j\rangle$ et simule M sur l'entrée i pendant au plus j étapes. Si M rentre dans un état d'acceptation, N_M accepte x, et sinon, N_M passe au successeur de $\langle i, j\rangle$.
2. T interroge l'oracle pour savoir si $\langle N_M, \varepsilon\rangle$ appartient à L_u. Le cas échéant, elle rejette l'entrée. Sinon, elle l'accepte.

$\langle M \rangle \in L_v \iff L(M) = \emptyset$ donc si $\langle M \rangle \in L_v$, alors N_M ne s'arrête sur aucune entrée. Donc $\langle N_M, \varepsilon \rangle \notin L_u$. Donc T accepte $\langle M \rangle$.

Réciproquement, si T accepte $\langle M \rangle$, c'est que $\langle N_M, \varepsilon \rangle \notin L_u$. Donc N_M n'accepte pas ε. Donc M ne s'arrête sur aucun i, et $L(M) = \emptyset$. □

On a donc prouvé que $L_v = L(T, L_u)$[1]. On montre maintenant la réciproque qui stipule que L_u est reconnu par une machine de Turing avec l'oracle L_v. On cherche à prouver l'existence d'une telle machine, et on la note S. On définit S de la manière suivante : sur l'entrée $\langle M, w \rangle$,

- S construit la machine M' qui accepte $\emptyset$ si $w \notin L(M)$, et Σ^* dans le cas contraire. Sur l'entrée x, M', ignorant son entrée, simule M sur w, et accepte si M accepte w.
- S interroge l'oracle sur $\langle M' \rangle$: S accepte l'entrée si et seulement si la réponse de l'oracle est non.

Ceci nous permet de construire une hiérarchie.

$$\begin{cases} S_1 = L_v \\ S_{i+1} = \{ \langle M \rangle \mid L(M, S_i) = \emptyset \} \end{cases}$$

On peut montrer avec ces outils des résultats intéressants, qui seront repris plus en détail dans la suite. On énonce seulement un résultat de manière informelle pour avoir un exemple du type de résultats que l'on obtient. La notion d'équivalence est l'idée que l'on s'en fait intuitivement au vu de ce qui a été fait, à savoir que deux langages sont dits équivalents s'ils peuvent jouer chacun le rôle d'oracle pour reconnaître l'autre. On a vu avec L_u et L_v qu'un langage récursivement énumérable et un langage non récursivement énumérable peuvent être équivalents.

Proposition 4.8. $\{ \langle M \rangle \mid L(M) = \Sigma^* \}$ *est « équivalent » à* S_2.

On définit maintenant la notion de Turing-réduction, très utile pour mettre à jour une structure de PSPACE.

Définition 4.9. A et B étant deux langages, A est dit B-récursif s'il existe une machine de Turing avec l'oracle B qui décide A. De la même façon, A est dit B-récursivement énumérable s'il existe une machine de Turing avec l'oracle B qui reconnaît A.

Dans cette définition, les termes sont importants. Il ne faut pas confondre la *décision* d'un langage, et sa *reconnaissance*. On rappelle qu'un langage L est décidé lorsqu'il existe une machine de Turing qui prenant en entrée un mot w, répond 1 (ou oui) si $w \in L$, et 0 (ou non) si $w \notin L$. Le langage L est reconnu s'il existe une machine de Turing qui sur l'entrée w répond 1 (ou oui) si $w \in L$ et boucle sinon. Dans la suite, on s'intéressera essentiellement à la décision de langages.

On adopte deux notations pour cette notation de récursivité. On note $A = L(M, B)$ lorsque A est B-récursif. La proposition suivante justifie la notation $A \leqslant_T B$.

Proposition 4.10. *La relation* $\leqslant_T$ *est un pré-ordre.*

Un pré-ordre est une relation réflexive et transitive.

Preuve. $\leqslant_T$ est clairement réflexive. Posons alors $A = L(M, B)$ et $B = L(M', C)$, ce qui signifie que $A \leqslant_T B$ et $B \leqslant_T C$. On construit une machine M'' avec l'oracle C qui décide A. Sur une entrée u, M'' simule M jusqu'à entrer dans l'état d'interrogation de l'oracle. Elle simule alors M', avec oracle C. La réponse de M' permet de continuer la simulation de M comme si on avait utilisé l'oracle B. □

On rappelle une autre notion de réduction. On note $A \leqslant_r B$ lorsqu'il existe une fonction totale récursive telle que

$$x \in A \iff f(x) \in B.$$

On donne quelque relation entre les deux pré-ordres.

[1] La notation signifie que L_v est le langage décidé par la machine T avec l'oracle L_u.

Proposition 4.11. *Si $A \leqslant_r B$, alors $A \leqslant_T B$.*

Ce résultat est complètement trivial, tout comme ceux qui viennent.

Proposition 4.12.

(i) Si $A \leqslant_T B$ et si B est récursif, alors A est récursif.

(ii) Si $A \leqslant_r B$ et si B est C-récursif (resp. récursivement énumérable), alors A est C-récursif (resp. récursivement énumérable).

(iii) Si $A \leqslant_r B$ et si A n'est pas récursivement énumérable, alors B ne l'est pas non plus.

Au vu de ces résultats faciles, il est naturel de se demander la chose suivante : si $A \leqslant_T B$, et si A n'est pas récursivement énumérable, que peut-on dire de B ? La réponse, fort décevante, est que l'on ne peut rien dire. En effet, on peut pour s'en convaincre reprendre l'exemple précédent, et remarquer que $\overline{L_u} \leqslant_T L_u$.

De même que pour $\leqslant_r$ où on s'était intéressé à la réduction en un temps donné, et on particulier en temps polynomial ce qui nous donnait la réduction $\leqslant^p$, on s'intéresse ici à la Turing-réduction polynomiale.

Définition 4.13. On dit[2] que A se Turing-réduit polynomialement à B, et on note $A \leqslant_T^p B$ s'il existe une machine de Turing M qui décide A avec l'oracle B de telle sorte que tout calcul de M sur u soit de longueur bornée par un polynôme p et que les mots d'oracle soient de longueur bornée par ce même polynôme.

Cette Turing-réduction polynomiale a quelques propriétés évidentes que l'on énonce ici sans les démontrer.

Proposition 4.14.

(i) $\forall A,\ \overline{A} \leqslant_T^p A$.

(ii) $\forall A, B,\ A \leqslant^p B \implies A \leqslant_T^p B$.

(iii) Si $\mathsf{NP} \neq \mathsf{CoNP}$, $\overline{\texttt{SAT}}$ *n'est pas polynomialement réductible à* SAT.

Voyons maintenant quelques propriétés de clôture des classes usuelles par la relation $\leqslant_T^p$.

Proposition 4.15. P *et* PSPACE *sont closes pour* $\leqslant_T^p$.

Preuve. Si $B \in \mathsf{P}$ et $A \leqslant_T^p B$, alors $A \in \mathsf{P}$, car il suffit le simuler l'interrogation à l'oracle en temps polynomial. La démonstration est analogue pour PSPACE. □

Proposition 4.16. $\mathsf{NP} = \mathsf{CoNP} \iff \mathsf{NP}$ *est close pour* $\leqslant_T^p$.

Preuve. Supposons NP close pour $\leqslant_T^p$. Si $A \in \mathsf{NP}$, alors $\overline{A} \in \mathsf{CoNP}$. Mais $\overline{A} \leqslant_T^p A$, donc $\overline{A} \in \mathsf{NP}$, et donc $A \in \mathsf{CoNP}$. L'autre sens est analogue.

Réciproquement, on suppose que $\mathsf{NP} = \mathsf{CoNP}$ et on veut montrer que si $A \leqslant_T^p B$ et $B \in \mathsf{NP}$, alors $A \in \mathsf{NP}$. Il suffit d'utiliser la caractérisation suivante de NP : $B \in \mathsf{NP}$ si et seulement s'il existe un langage $C \in \mathsf{P}$ et un polynôme p tels que

$$x \in B \iff \exists\, y,\ |y| \leqslant p(|x|) \text{ et } \langle x, y\rangle \in C.$$

On peut sans difficulté simuler l'oracle en temps polynomial sur une machine non déterministe. □

[2] Ou plutôt on évite de le dire !

4.5 Hiérarchie polynomiale

Dans cette partie nous allons décrire une hiérarchie de PSPACE et démontrer quelques unes de ses propriétés afin de mettre à jour une *bonne* structure de PSPACE.

Définition 4.17. On note $(\Sigma_k^{\mathsf{P}}, \Pi_k^{\mathsf{P}}, \Delta_k^{\mathsf{P}})_{k\geqslant 0}$ la hiérarchie polynomiale définie par induction comme suit :

- Pour $k = 0$, $\Sigma_0^{\mathsf{P}} = \Pi_0^{\mathsf{P}} = \Delta_0^{\mathsf{P}} = \mathsf{P}$
- $\Sigma_{k+1}^{\mathsf{P}} = \mathsf{NP}^{\Sigma_k^{\mathsf{P}}}$, *i.e.* la classe des langages reconnus par une machine de Turing non déterministe en temps polynomial avec un oracle dans Σ_k^{P}.
- $\Pi_{k+1}^{\mathsf{P}} = \mathsf{Co}\Sigma_{k+1}^{\mathsf{P}}$
- $\Delta_{k+1}^{\mathsf{P}} = \mathsf{P}^{\Sigma_k^{\mathsf{P}}}$

4.5.1 Caractérisation

Cette partie va mettre au jour une caractérisation des classes de complexité définies ci-dessus. A fin de voir plus clair dans cette hiérarchie, la première proposition nous en donne une structure globale.

Proposition 4.18.

$$\forall k \in \mathbb{N}, \begin{cases} \Delta_{k+1}^{\mathsf{P}} & \subseteq \Sigma_{k+1}^{\mathsf{P}} \cap \Pi_{k+1}^{\mathsf{P}} \\ \Sigma_k^{\mathsf{P}} \cup \Pi_k^{\mathsf{P}} & \subseteq \Delta_{k+1}^{\mathsf{P}} \end{cases}$$

Preuve. Soit $\mathcal{L}$ un langage de $\Delta_{k+1}^{\mathsf{P}}$, il existe ainsi un oracle A dans Σ_k^{P} tel que $\mathcal{L} \in \mathsf{P}^A$. Or $\mathsf{P} \subseteq \mathsf{NP}$, ce qui permet de déduire que $\mathcal{L} \in \mathsf{P}^A \subseteq \mathsf{NP}^{\Sigma_k^{\mathsf{P}}} = \Sigma_{k+1}^{\mathsf{P}}$. De même, et comme $\mathsf{P} \subseteq \mathsf{NP}$, $\mathcal{L} \in \Pi_{k+1}^{\mathsf{P}}$.

Soit $\mathcal{L} \in \Sigma_k^{\mathsf{P}}$, alors $\mathcal{L} \in \mathsf{P}^{\mathcal{L}} \subseteq \mathsf{P}^{\Sigma_k^{\mathsf{P}}} = \Delta_{k+1}^{\mathsf{P}}$. □

La proposition 4.18 nous permet d'exhiber la structure suivante.

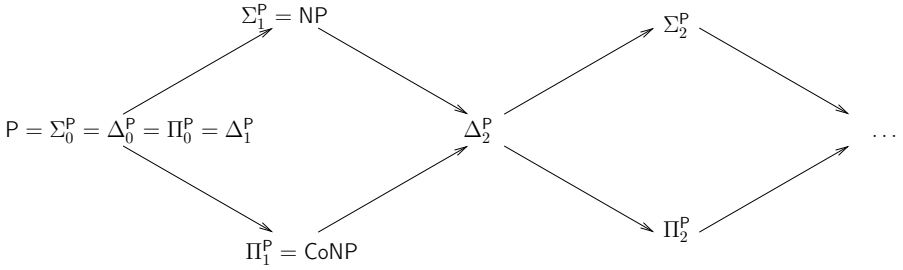


FIG. 4.1: Structure de PSPACE par la hiérarchie polynomiale

Le théorème suivant donne une caractérisation des classes préalablement définies à partir de problèmes de P. C'est une généralisation de la Proposition 3.14.

Théorème 4.19. *Pour tout $k \in \mathbb{N}$,*

(i) $A \in \Sigma_k^{\mathsf{P}}$ si et seulement si il existe un langage $B \in \mathsf{P}$ et un polynôme p tels que :

$$x \in A \iff \exists^{p(|x|)} y_1, \forall^{p(|x|)} y_2, \dots Q_k^{p(|x|)} y_k, \langle x, y_1, y_2, \dots y_k \rangle \in B$$

où $Q_j = \begin{cases} \exists & \text{si } j \text{ est impair,} \\ \forall & \text{sinon.} \end{cases}$

(ii) $A \in \Pi_k^{\mathsf{P}}$ *si et seulement s'il existe un langage* $B \in \mathsf{P}$ *et un polynôme* p *tels que :*

$$x \in A \iff \forall^{p(|x|)} y_1, \exists^{p(|x|)} y_2, \dots Q_k^{p(|x|)} y_k, \langle x, y_1, y_2, \dots y_k \rangle \in B$$

$$\text{où } Q_j = \begin{cases} \forall & \text{si } j \text{ est impair,} \\ \exists & \text{sinon.} \end{cases}$$

Remarque (Notations). Définissons les notations précédemment utilisées :
- $\exists^{p(n)} x, R(x)$ signifie qu'il existe un mot de longueur au plus $p(n)$ tel que $R(x)$ soit satisfait.
- Soit $\mathcal{C}$ une classe de complexité, $\exists\mathcal{C}$ désigne la classe des langages A tels qu'il existe $B \in \mathcal{C}$ et un polynôme p tels que :

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

- Les définitions sont similaires pour $\forall$.
- Si A est un langage, on note $A^{(*)} = \{\langle y_1, y_2, \dots y_k \rangle \mid \forall i, y_i \in A\}$

On définit maintenant les outils nécessaires à la démonstration du Théorème 4.19.

Définition 4.20. Une classe de langage $\mathcal{C}$ est dite *close par paire*, si le fait que A appartienne à $\mathcal{C}$ implique que $\{\langle x, y \rangle \mid x \in A, y \text{ quelconque}\}$ soit dans $\mathcal{C}$.

Proposition 4.21. *Pour toute classe de langage* $\mathcal{C}$*,*

(i) $A \in \exists\mathcal{C} \iff \overline{A} \in \forall\mathcal{C}$

(ii) Si $\mathcal{C}$ *est close par paire, alors* $\mathcal{C} \subseteq \forall\mathcal{C}$ *et* $\mathcal{C} \subseteq \exists\mathcal{C}$*.*

Preuve. Si A est dans $\exists\mathcal{C}$ alors il existe $B \in \mathcal{C}$ et un polynôme p tels que :

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

D'où :

$$\begin{aligned} x \in \overline{A} &\iff x \notin A \\ &\iff \forall^{p(|x|)} y \, \langle x, y \rangle \notin B \\ &\iff \forall^{p(|x|)} y \, \langle x, y \rangle \in \overline{B}. \end{aligned}$$

Ainsi $\overline{A} \in \forall\mathcal{C}$. On prouve l'autre implication de façon similaire.

La proposition (ii) est triviale. □

Proposition 4.22. *Soit* $\mathcal{C}$ *l'une des classes* $\Sigma_k^{\mathsf{P}}, \Pi_k^{\mathsf{P}}$ *ou* Δ_k^{P}*, alors pour tout langage* A *de* $\mathcal{C}$*,*

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

Preuve. Supposons, par exemple, que $A \in \Sigma_k^{\mathsf{P}}$. Il existe donc une machine de Turing non déterministe $\mathcal{M}$ qui reconnaît A avec un oracle $B \in \Sigma_{k-1}^{\mathsf{P}}$. Il suffit alors de construire la machine qui prend en entrée un k-uplet $\langle y_1, y_2, \dots, y_k \rangle$ et qui le décode en $y_1, y_2, \dots y_k$. Elle simule ensuite $\mathcal{M}$ sur chacun des y_i. L'acceptation a lieu si chaque y_i est accepté.

Pour l'autre inclusion, on remarque que $A \leqslant^{\mathsf{P}} A^{(*)}$ grâce à la fonction $x \mapsto \langle x \rangle$. □

Ce théorème contient toute l'information dont on a besoin pour le théorème 4.19. Certains points ont déjà été vus ((i) et (ii)) et certains sont conséquences directes des autres points ((iv) et (vi)).

Théorème 4.23.

(i) $\exists\mathsf{P} = \mathsf{NP} = \Sigma_1^\mathsf{P}$
(ii) $\forall\mathsf{P} = \mathsf{CoNP} = \Pi_1^\mathsf{P}$
(iii) $\forall k \in \mathbb{N}^*, \exists\Sigma_k^\mathsf{P} = \Sigma_k^\mathsf{P}$
(iv) $\forall k \in \mathbb{N}^*, \forall\Pi_k^\mathsf{P} = \Pi_k^\mathsf{P}$
(v) $\forall k \in \mathbb{N}, \exists\Pi_k^\mathsf{P} = \Sigma_{k+1}^\mathsf{P}$
(vi) $\forall k \in \mathbb{N}, \forall\Sigma_k^\mathsf{P} = \Pi_{k+1}^\mathsf{P}$

Preuve. (i) et (ii) sont déjà prouvés (*cf* Proposition 3.14).

(iii) Remarquons que Σ_k^P est close par paire. Par la proposition 4.21,

$$\Sigma_k^\mathsf{P} \subseteq \exists\Sigma_k^\mathsf{P}.$$

Réciproquement, soit $A \in \exists\Sigma_k^\mathsf{P}$. Il existe donc $B \in \Sigma_k^\mathsf{P}$ et un polynôme p tels que

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

Puisque $B \in \Sigma_k^\mathsf{P}$, il existe une machine $\mathcal{M}$ et un oracle D tels que $B = \mathcal{L}(\mathcal{M}, D)$. En considérant alors l'algorithme suivant, nous montrons finalement que $A \in \Sigma_k^\mathsf{P}$. Sur l'entrée x, on devine y tel que $|y| \leqslant p(|y|)$. On code $\langle x, y\rangle$ et on fait fonctionner $\mathcal{M}$ sur $\langle x, y\rangle$. On accepte x si et seulement si $\mathcal{M}$ accepte.

(v) Remarquons en premier lieu que $\Sigma_{k+1}^\mathsf{P} = \mathsf{NP}^{\Pi_k^\mathsf{P}}$. En effet, par définition $\Sigma_{k+1}^\mathsf{P} = \mathsf{NP}^{\Sigma_k^\mathsf{P}}$. Il existe donc une machine de Turing non déterministe $\mathcal{M}$ et un oracle $B \in \Sigma_k^\mathsf{P}$ tels que $A = \mathcal{L}(\mathcal{M}, B)$. On considère alors la machine $\mathcal{M}'$ avec l'oracle $\overline{B}$ qui simule $\mathcal{M}$ sauf sur les configurations d'interrogation de l'oracle où elle répond le contraire de $\mathcal{M}$.

Supposons que $A \in \exists\Pi_k^\mathsf{P}$. Alors il existe $B \in \Pi_k^\mathsf{P}$ et un polynôme p tels que $x \in A \iff \exists^{p(|x|)} y(\langle x, y\rangle \in B)$. On considère l'algorithme suivant : sur l'entrée x,

- deviner y tel que $|y| \leqslant p(|x|)$;
- coder $\langle x, y\rangle$;
- en utilisant le fait que $B \in \Pi_k^\mathsf{P} = \mathsf{Co}\Sigma_k^\mathsf{P}$, décider si $\langle x, y\rangle \in B$ ou non.

Donc $A \in \mathsf{NP}^{\Pi_k^\mathsf{P}} = \Sigma_{k+1}^\mathsf{P}$.

Réciproquement, on cherche à prouver que $\Sigma_{k+1}^\mathsf{P} \subseteq \exists\Pi_k^\mathsf{P}$. Pour cela, on raisonne par récurrence sur k. Si $k = 0$, on a $\Sigma_1^\mathsf{P} = \mathsf{NP}$ et $\exists\Pi_0^\mathsf{P} = \exists\mathsf{P} = \mathsf{NP}$. Supposons alors que $\Sigma_k^\mathsf{P} \subseteq \exists\Pi_{k-1}^\mathsf{P}$. Soit $A \in \Sigma_{k+1}^\mathsf{P}$. Il existe donc une machine de Turing non déterministe en temps polynomial $\mathcal{M}$ et un oracle $B \in \Sigma_k^\mathsf{P}$ tels que $A = \mathcal{L}(\mathcal{M}, B)$. Donc $x \in A$ revient à dire qu'il existe un calcul de $\mathcal{M}$ sur x menant de la configuration initiale q_0x à la configuration d'acceptation.

$$x \in A \iff \alpha_0 = q_0x \vdash \alpha_1 \vdash \alpha_2 \vdash \cdots \vdash \alpha_{\text{acc}},$$

les α_i étant les configurations successives de $\mathcal{M}$. Parmi les α_i, il existe des configurations d'interrogation de l'oracle. Notons $(z_1, \ldots, z_q)$ les mots du ruban d'oracle qui donnent lieu à une réponse positive et $(w_1, \ldots, w_m)$ ceux donnant lieu à une réponse négative. Alors α_0 est une configuration initiale, α_{acc} une configuration acceptante, et $\alpha_i \vdash \alpha_{i+1}$ si l'une des trois conditions suivantes est vérifiée :

- $\alpha_i \not\supset q_?$;
- $\alpha_i \supset q_?$ et la réponse est oui (sur un z_i) ;
- $\alpha_i \supset q_?$ et la réponse est non (sur un w_i).

Remarquons alors que $\langle z_1, \ldots, z_q\rangle \in B^{(*)}$ et $\langle w_1, \ldots, w_m\rangle \in \overline{B}^{(*)}$. Comme $B \in \Sigma_k^\mathsf{P}$, l'hypothèse de récurrence assure que $B \in \exists\Pi_{k-1}^\mathsf{P}$, et donc $B^{(*)}$ également. De même, $\overline{B} \in \Pi_k^\mathsf{P}$ donc $\overline{B}^{(*)}$ également.

Ceci montre que $A \in \exists\Pi_k^\mathsf{P}$.

(iv) et (vi) se déduisent respectivement de (iii) et (v). □

On démontre maintenant le théorème initial. La preuve consiste essentiellement à remarquer que le travail a déjà été fait !

Preuve du Théorème 4.19. La preuve se fait par récurrence sur k.

Pour $k = 0$, $\Sigma_k^{\mathsf{P}} = \Pi_k^p = \Delta_k^{\mathsf{P}} = \mathsf{P}$.

On suppose les deux résultats vrais au rang k. Soit $A \in \Sigma_{k+1}^{\mathsf{P}}$. On vient de prouver que $\Sigma_{k+1}^{\mathsf{P}} = \exists\Pi_k^{\mathsf{P}}$. On peut alors utiliser l'hypothèse de récurrence pour rajouter le quantificateur existentiel manquant et obtenir (i). On fait de même pour (ii). □

4.5.2 Théorèmes d'effondrement

On ne sait pas si cette hiérarchie polynomiale est réelle, ou si tous le niveaux sont égaux, au moins à partir d'un certain rang. Ces questions sont en fait équivalentes à des questions ouvertes très connues telles que $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$. Cependant, on peut sous certaines hypothèses prouver que la hiérarchie s'effondre. C'est l'objet des théorèmes de cette partie.

On commence par montrer que s'il y a effondrement à un niveau, alors la hiérarchie s'arrête à ce niveau.

Théorème 4.24. *S'il existe k tel que $\Sigma_k^{\mathsf{P}} = \Pi_k^{\mathsf{P}}$, alors pour tout $j \leqslant 0$,*

$$\Sigma_{k+j}^{\mathsf{P}} = \Pi_{k+j}^{\mathsf{P}}.$$

Preuve. On montre ce résultat par récurrence sur j. Le cas de base est trivial, et on suppose le résultat au rang $j-1$. Alors

$$\Sigma_{k+j}^{\mathsf{P}} = \exists\Pi_{k+(j-1)}^{\mathsf{P}} \stackrel{HR}{=} \exists\Sigma_{k+(j-1)}^{\mathsf{P}} = \Pi_{k+j}^{\mathsf{P}}.$$

□

Comme corollaires de ce résultat, on peut voir le lien entre l'effondrement de la hiérarchie et le grand problème ouvert de la complexité.

Corollaire 4.25. *S'il existe $k > 0$ tel que $\Sigma_0^{\mathsf{P}} \neq \Sigma_k^{\mathsf{P}}$, alors $\mathsf{P} \neq \mathsf{NP}$.*

Corollaire 4.26.

$$\mathsf{P} \neq \mathsf{NP} \iff \mathsf{P} \neq \mathsf{PH},$$

où $\mathsf{PH} = \bigcup_k \Sigma_k^{\mathsf{P}} = \bigcup_k \Pi_k^{\mathsf{P}} = \bigcup_k \Delta_k^{\mathsf{P}}$.

Il est relativement clair que $\mathsf{PH} \subseteq \mathsf{PSPACE}$. En fait, tout niveau de la hiérarchie est dans PSPACE (récurrence sur k). Mais ce qu'on ne sait pas, c'est si cette hiérarchie englobe tout PSPACE. Si c'était le cas, alors la hiérarchie serait finie.

Théorème 4.27. *Si $\mathsf{PH} = \mathsf{PSPACE}$, alors il existe k tel que $\Sigma_{k+1}^{\mathsf{P}} = \Sigma_k^{\mathsf{P}}$.*

Preuve. En effet, PSPACE contient un langage PSPACE-complet A. Si $\mathsf{PH} = \mathsf{PSPACE}$, il existe k tel que $A \in \Sigma_k^{\mathsf{P}}$. Mais alors si $B \in \mathsf{PSPACE}$, $B \leqslant^p A$, donc $B \in \Sigma_k^{\mathsf{P}}$. □

4.5.3 Autres résultats

D'autres résultats sont intéressants à partir de cette hiérarchie. Le premier concerne la hiérarchie elle-même, avec l'existence de problèmes complets pour chaque classe Σ_k^{P}.

Théorème 4.28. *Pour tout k, il existe une famille de langages Σ_k^{P}-complets.*

Preuve. Pour un ensemble X_i de variables, on note $\tau_i : X_i \to \{0,1\}$ une fonction qui associe à chaque variable une valeur booléenne.

$$\mathtt{SAT}_k = \begin{cases} \text{Entrée} & : X_1, \ldots, X_k \text{ des ensembles de variables, } F \text{ une formule booléenne} \\ & \quad \text{sur } X = \bigcup_i X_i. \\ \text{Question} & : \text{Est-il vrai que } \exists\tau_1 \forall\tau_2 \ldots Q_k \tau_k (F_{|\tau_1 \ldots \tau_k} = 1) \,? \end{cases}$$

Pour $k \geqslant 1$, $\mathtt{SAT}_k$ est Σ_k^{P}-complet. □

On donne maintenant quelques exemples de problèmes de PH, en indiquant plus précisément dans quelle classe ils se trouvent.

Exemple 4.29.

$$\texttt{EC} = \{\langle G, k\rangle \mid G \text{ est un graphe fini dont la clique de taille maximale est de taille } k.\}$$

Le problème EC (Exact-Clique) est dans Σ_2^{P}. En effet, il est défini comme l'ensemble des graphes vérifiant

$$\exists C \forall C' (C \text{ est une clique de taille } k \text{ et } C' \text{ est un clique}) \implies \left|C'\right| \leqslant k.$$

On pourrait encore caractériser un élément de EC par le fait que si S est un ensemble de sommet de taille supérieure à $k+1$, S n'est pas une clique et il existe une clique de taille k. Donc $\texttt{EC} \in \Pi_2^{\mathsf{P}}$.

D'où $\texttt{EC} \in \Delta_2^{\mathsf{P}}$.

Exemple 4.30. Deux formules booléennes sont dites équivalentes lorsqu'elles sont satisfiables sur le même ensemble de valuation.

$$\texttt{Min}\ F = \{\langle \varphi\rangle \mid \varphi \text{ n'est équivalente à aucune formule de taille plus petite.}\}$$

On peut définir ce problème par la formule

$$\forall \psi \exists v (|\psi| < |\varphi| \implies \psi(v) \neq \varphi(v)).$$

Donc $\texttt{Min}\ F \in \Pi_2^{\mathsf{P}}$.

Exemple 4.31. Soit $G = (V, E)$ un graphe fini complet dont les arêtes sont partiellement 2-colorées par c ($c : E \to \{1, 2, *\}$). Soit $k \geqslant 0$. Est-il vrai que pour toute 2-coloration c' qui est une restriction de c, il existe une clique de taille k qui soit monochrome ?

Ce problème est Π_2^{P}-complet.

Les deux prochains résultats sont primordiaux pour la méthodologie d'attaque du problème $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$. En effet, ils montrent qu'on ne peut utiliser d'oracles. Cela signifie qu'il faut l'attaquer quasiment de front. C'est sans doute ce qui lui permet de résister depuis longtemps.

Proposition 4.32. *Il existe A tel que $\mathsf{P}^A = \mathsf{NP}^A$.*

Preuve. Soit A un langage PSPACE-complet. Montrons que $\mathsf{NP}^A \subseteq \mathsf{P}^A$.

Si $L \in \mathsf{NP}^A$, il existe une machine de Turing non déterministe $\mathcal{M}$ en temps polynomial telle que $L = \mathcal{L}(\mathcal{M}, A)$. D'où $\mathsf{NP}^A \subseteq \mathsf{NPSPACE}$. Or $\mathsf{NPSPACE} = \mathsf{PSPACE}$ et $\mathsf{PSPACE} \subseteq \mathsf{P}^A$. En effet, soit $L \in \mathsf{PSPACE}$. Alors $L \leqslant^p A$. Donc $L \in \mathsf{P}^A$ grâce à l'algorithme suivant : sur l'entrée x, on calcule $f(x)$ où f est le témoin de $L \leqslant^p A$, puis on interroge A.

Donc $\mathsf{NP}^A = \mathsf{P}^A$. □

Proposition 4.33. *Il existe B tel que $\mathsf{P}^B \neq \mathsf{NP}^B$.*

Remarque. Il existe une énumération effective des machines de Turing en temps polynomial qui caractérise la classe des langages décidés en temps polynomial.

Preuve. On désigne par $(\mathcal{M}_i)$ une telle énumération. Soit p_i le polynôme associé à la machine $\mathcal{M}_i$. On construit B par induction. Plus précisément, on construit une suite (B_i, n_i) où B_i est un langage sur $\{0, 1\}$ et n_i un entier positif.

- On part de $B_0 = \emptyset$ et $n_0 = 0$.
- On suppose B_{i-1} et n_{i-1} définis, pour $i \geqslant 1$. On définit n_i comme le plus petit entier n tel que $2^n \geqslant p_i(n)$ et $n \geqslant 2^{n_{i-1}}$. Donc $2^{n_i} > p_i(n_i) > n_i > 2^{n_{i-1}}$.
  On définit B_i par l'algorithme suivant. On fait fonctionner la machine $\mathcal{M}$ sur 0^{n_i} avec l'oracle B_{i-1}. Si 0^{n_i} est accepté, on pose $B_i = B_{i-1}$. Sinon, il existe un mot y de longueur n_i qui n'a pas été interrogé lors du calcul. Il en existe un plus petit, que l'on note encore y. On pose $B_i = B_{i-1} \cup \{y\}$.

On pose alors

$$B = \bigcup_{i \geqslant 0} B_i.$$

Soit $L_B = \{0^n \mid \text{il existe } x \in B \text{ tel que } |x| = n\}$. On montre que $L_B \in \mathsf{NP}^B$ mais que $L_B \notin \mathsf{P}^B$.

- $L_B \in \mathsf{NP}^B$: sur l'entrée 0^n, on devine x tel que $|x| = n$ et on interroge B.
- $L_B \notin \mathsf{P}^B$: remarquons d'abord que $0^{n_i} \in \mathcal{L}(\mathcal{M}_i, B) \iff 0^{n_i} \in \mathcal{L}(\mathcal{M}_i, B_{i-1})$. Donc

$$0^{n_i} \in \mathcal{L}_B \iff 0_i^n \notin \mathcal{L}(\mathcal{M}_i, B) \iff 0^{n_i} \notin \mathcal{L}(\mathcal{M}_i, B_{i-1}).$$

□

Chapitre 5

Hiérarchies

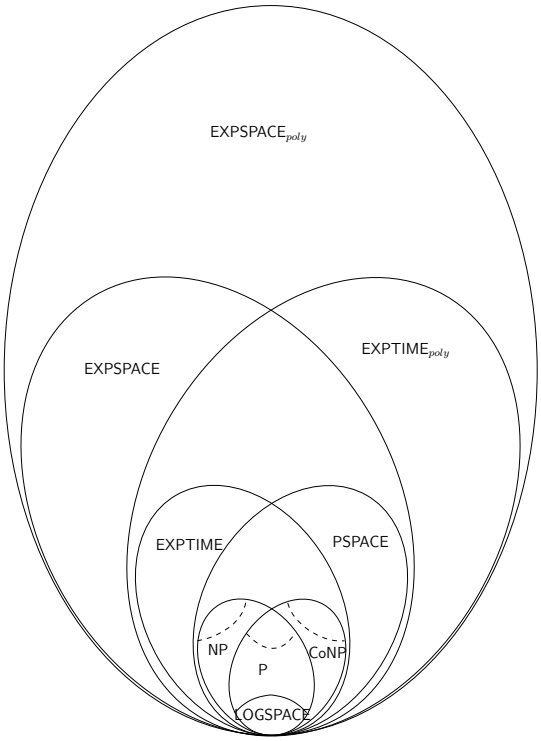


FIG. 5.1: Contenu du chapitre. Les pointillés indiquent les problèmes complets pour la classe considérée. Tout n'est pas prouvé dans ce chapitre, certains résultats venant de chapitres précédents.

Après s'être intéressés à la hiérarchie polynomiale, on va dans ce chapitre s'intéresser de manière plus générale aux différentes hiérarchies qui existent au niveau des classes de complexité. L'une des questions qui sous-tend ce chapitre est de savoir ce qu'il faut rajouter en complexité pour passer d'une classe à une autre de façon stricte. Autrement dit, on se demande quels ingrédients on doit ajouter pour avoir strictement plus de problèmes dans une classe de complexité.

5.1 Comparaisons immédiates

Proposition 5.1. *Soit f une fonction de $\mathbb{N}$ dans $\mathbb{N}$. Alors*
(i) si $f(n) \geqslant n$ pour tout n,

$$\mathsf{DTIME}(f(n)) \subseteq \mathsf{NTIME}(f(n)) \subseteq \mathsf{DSPACE}(f(n)),$$

(ii) si $f(n) \geqslant \log n$ *pour tout* n*,*

$$\mathsf{DSPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n)) \subseteq \bigcup_{c>0} \mathsf{DTIME}(2^{cf(n)}).$$

Preuve.

(i) La première inclusion est un résultat bien connu, on se contente donc de prouver que dans les conditions d'application de la proposition, $\mathsf{NTIME}(f(n)) \subseteq \mathsf{DSPACE}(f(n))$.

Soit M une machine de Turing non déterministe en temps $f(n)$. On sait construire une machine de Turing déterministe *équivalente* en temps $2^{cf(n)}$ où c est une constante qui dépend de M. On munit M de trois rubans :

R_1 **:** ruban d'entrée,

R_2 **:** ruban d'énumération des *chemins* possibles dans la d'exécution de la machine M,

R_3 **:** calcul de M selon le mot courant de R_2.

On peut aisément vérifier que cette machine travaille en espace $f(n)$, car en particulier, les chemins dans l'arbre d'exécution de M ne peuvent dépasser cette taille.

(ii) De même, la seule chose à prouver est l'inclusion $\mathsf{NSPACE}(f(n)) \subseteq \bigcup_{c>0} \mathsf{DTIME}(2^{cf(n)})$.

On énumère sur un ruban les configurations distinctes possibles. Ce nombre de configurations est borné par $2^{\mathcal{O}(f(n))}$. Le temps nécessaire à énumérer ces configurations est également borné par $2^{\mathcal{O}(f(n))}$. Ce qui achève la preuve.

□

On donne ici l'énoncé d'un théorème, dit *théorème de la lacune*, prouvé par Bordin. La démonstration de ce théorème est un peu longue mais sans être très complexe. Elle utilise des techniques classiques du domaine.

Théorème 5.2. *Soit* g *une fonction totale récursive telle que* $g(n) \geqslant n$ *pour tout* n*. Alors il existe une fonction totale récursive* s *telle que*

$$\mathsf{DSPACE}(s(n)) = \mathsf{DSPACE}(g(s(n))).$$

Preuve. Soit $(M_i)_i$ une énumération des machines de Turing. Soit s_i la complexité en espace de M_i. On cherche à construire s.

On va le faire de telle sorte que pour tout $k \geqslant 0$,
- soit $s_k(n) \leqslant s(n)$ presque partout,
- soit $g(s(n)) < s_k(n)$ infiniment souvent.

Ce qui impliquera qu'aucune fonction s_k ne peut satisfaire $s(n) \leqslant s_k(n) \leqslant g(s(n))$ infiniment souvent.

Soit n un entier positif. On considère les machines $M_1, \ldots, M_n$. On définit $s(n)$ par l'algorithme suivant :

```
1 j := 1;
2 tant que il existe i, 0 ⩽ i ⩽ n, tel que j + 1 ⩽ s_i(n) ⩽ g(j) faire
3 |  j := s_i(n);
4 s(n) := j;
```

i étant donné, on connaît M_i. Pour une entrée de longueur n, on fait fonctionner M_i en espace borné par $g(j)$. Si M_i s'arrête dans cet espace pour tout n, sa complexité est inférieure à $g(j)$. S'il existe une entrée sur laquelle M_i ne s'arrête pas dans cet espace, $s_i(n) > g_j(n)$ ou $s_i(n)$ n'est pas défini. Et on n'a pas $j + 1 \leqslant s_i(n) \leqslant g(j)$! On a donc soit $s_i(n) \leqslant s(n)$, soit $g(n) < s_i(n)$.

Il suffit de vérifier que $\mathsf{DSPACE}(g(s(n))) \subseteq \mathsf{DSPACE}(s(n))$. Supposons le contraire, donc qu'il existe $L \in \mathsf{DSPACE}(g(s(n)))$ tel que $L \notin \mathsf{DSPACE}(s(n))$.

Il existe M_k telle que $L = L(M_k)$, et $s_k(n) \leqslant g(s(n))$ presque partout. En particulier pour tout entier $n \leqslant k$. On sait que pour tout i, $0 \leqslant i \leqslant n$, $s_i(n) \leqslant s(n)$ ou $g(s(n)) < s_i(n)$. En particulier, $s_k(n) \leqslant s(n)$ ou $g(s(n)) < s_k(n)$.

Donc $s_k(n) \leqslant s(n)$ pour n assez grand. □

L'intitulé de ce chapitre parlant de hiérarchies, le résultat suivant va être particulièrement intéressants. Il justifie le chapitre en assurant qu'il existe des hiérarchies. Ce n'est pas son propos exact mais ce sera un corollaire du théorème.

Théorème 5.3. *Soit f une fonction totale récursive. Alors il existe un langage récursif qui n'appartient pas à* $\mathsf{DTIME}(f(n))$.

Il est intéressant de noter qu'il existe un résultat analogue pour l'espace. On se contente ici de démontrer le résultat sur le temps.

Preuve. Soit M une machine qui calcule f. Soit (M_i) une énumération des machines de Turing. Soit (x_i) une énumération des mots sur $\{0,1\}$. On pose

$$L = \{x_i \,|\, M_i \text{ ne s'arrête pas en moins de } f(|x_i|) \text{ pas de calcul ou rejette } x_i\}.$$

L est récursif : sur l'entrée x,
- chercher i tel que $x = x_i$,
- calculer $|x_i|$ puis $f(|x_i|)$,
- lancer M_i sur x_i pour au plus $f(|x_i|)$ pas de calcul.

On accepte un mot si M_i s'arrête sur un état de rejet ou ne s'arrête pas, et on le refuse sinon, c'est-à-dire si M_i s'arrête sur un état d'acceptation.

On montre maintenant que $L \notin \mathsf{DTIME}(f(n))$. En effet, dans le cas contraire, il existe une machine M_j qui s'arrête sur toute entrée en temps $f(n)$ et qui reconnaît L. Mais si $x_j \in L$, alors $x_j \in L(M_j)$ et donc $x_j \notin L$. Et de la même façon, si $x_j \notin L$, on en déduit que $x_j \in L$. Cette contradiction montre termine la preuve. □

Il résulte de ce théorème qu'il existe des hiérarchies (strictes !) de classe de complexité. On va maintenant en construire une.

On part d'une fonction f totale récursive. Soit $L \notin \mathsf{DTIME}(f(n))$ récursif. Soit $\hat{f}$ la complexité en temps d'une machine qui décide L. Soit f' définie par $f'(n) = \max(f(n), \hat{f}(n))$. Clairement, $L \in \mathsf{DTIME}f'(n)$. On a donc construit f' telle que

$$\mathsf{DTIME}(f(n)) \subsetneq \mathsf{DTIME}(f'(n)).$$

5.2 Théorèmes de hiérarchie déterministe

Maintenant que l'on sait qu'il existe des hiérarchies strictes de classes de complexité, on va énoncer et prouver deux théorèmes, l'un pour l'espace et l'autre pour le temps, qui établissent des conditions suffisantes pour que deux fonctions définissent deux classes de complexité déterministe différentes. Ces théorèmes ne sont valables que pour des fonctions particulières, à savoir les fonctions constructibles pour le temps et celles constructibles pour l'espace.

Définitions 5.4.

(i) Une fonction f est dite constructible pour le temps s'il existe une machine de Turing M et un entier n_0 tels que pour tout $n \geqslant n_0$, M s'arrête sur l'entrée 1^n en temps exactement $f(n)$.

(ii) Une fonction f est dite constructible pour l'espace s'il existe une machine de Turing M et un entier n_0 tels que pour tout $n \geqslant n_0$, M s'arrête sur l'entrée 1^n en laissant exactement $f(n)$ cases non blanches et en n'ayant pas visité plus de $f(n)$ cases.

Remarque. Si f est constructible pour le temps, alors elle l'est pour l'espace.

On va maintenant énoncer sans les démontrer deux lemmes assez simples donnant des conditions suffisantes pour qu'une fonction soit constructibles pour le temps et pour l'espace, respectivement. Le premier est connu sous le nom de *théorème de Kobayashi*.

Lemme 5.5. *Soit $f : \mathbb{N} \to \mathbb{N}$ telle que $f(n) \geqslant n$ pour tout n, et telle qu'il existe un réel positif ε tel que $f(n) \geqslant (1+\varepsilon)n$ presque partout. Alors les deux assertions suivantes sont équivalentes :*

(i) f est calculable en temps $\mathcal{O}(f(n))$.

(ii) f est constructible pour le temps.

Lemme 5.6. *Soit $f : \mathbb{N} \to \mathbb{N}$ telle que $f(n) \geqslant \log n$ pour tout n assez grand. Alors les deux assertions suivantes sont équivalentes :*

(i) f est calculable en espace $\mathcal{O}(f(n))$.

(ii) f est constructible pour l'espace.

Ces pré-requis vont nous permettre d'énoncer les deux théorèmes souhaités, l'un pour l'espace et l'autre pour le temps.

Théorème 5.7. *Soient deux fonctions s_1 et s_2 de $\mathbb{N}$ dans $\mathbb{N}$ telles que*

(i) s_2 soit constructible pour l'espace et $s_2(n) \geqslant s_1(n) \geqslant \log n$ pour tout n,

(ii)

$$\lim_{n\infty} \frac{s_1(n)}{s_2(n)} = 0.$$

Alors

$$\mathsf{DSPACE}(s_1(n)) \subsetneq \mathsf{DSPACE}(s_2(n)).$$

Preuve. On veut exhiber un langage qui est dans $\mathsf{DSPACE}(s_2(n))$ mais pas dans $\mathsf{DSPACE}(s_1(n))$.

On considère une énumération (M_x) des machines de Turing à un ruban de travail sur $\{0, 1, B\}$ où x est le code de la machine M_x. On considère la machine U qui réalise l'algorithme suivant : sur l'entrée x, si x n'est pas le code d'une machine de l'énumération, rejeter l'entrée, et sinon

- déterminer une plage de $s_2(n)$ cases (où $|x| = n$),
- constituer un compteur à partir de ces cases, en transformant les caractères du ruban précédent en 0 et en les faisant précéder de 1 (ce qui représente alors $2^{s_2(n)}$ en binaire),
- simuler M_x en diminuant le compteur à chaque pas de calcul.

Si la simulation précédente exige de sortir de l'espace marqué de $s_2(n)$ cases, l'entrée est rejetée. Sinon, si M_x est entrée dans un état d'acceptation avant que le compteur soit à 0, on rejette l'entrée. Sinon, c'est-à-dire si M_x n'est pas entrée dans un état d'arrêt à la fin du décompte, ou si cet état d'arrêt est un état de rejet, alors on accepte l'entrée.

Clairement, $L(U)$ est récursif, et est dans $\mathsf{DSPACE}(s_2(n))$. On veut prouver maintenant que $L(U) \notin \mathsf{DSPACE}(s_1(n))$. Supposons donc le contraire. Alors il existe une machine de Turing M_x qui le décide en espace $c_1 s_1(n)$ et en temps $t_1(n)$. On a donc $L(U) = L(M_x)$.

La condition

$$\lim_{n\infty} \frac{s_1(n)}{s_2(n)} = 0$$

du théorème assure qu'il existe n_1 tel que

$$n \geqslant n_1 \implies c_1 s_1(n) \leqslant s_2(n).$$

On sait par ailleurs qu'il existe une constante (dépendant de M_x) telle que $t_1(n) \leqslant 2^{c_2 s_1(n)}$. Donc il existe un entier n_0 tel que

$$\forall\, n \geqslant n_0,\ c_1 s_1(n) < s_2(n) \text{ et } t_1(n) < t_2(n).$$

On peut conclure si l'énumération (M_x) est telle que toute machine a une infinité de codes de la forme $1^k 0x$ avec $k \geqslant 1$. On laisse le lecteur se convaincre de l'existence d'une telle énumération. □

Théorème 5.8. *Soient deux fonctions t_1 et t_2 de $\mathbb{N}$ dans $\mathbb{N}$ telles que*

(i) t_2 soit constructible pour le temps et $t_2(n) \geqslant t_1(n) \geqslant n$ pour tout n,

(ii)

$$\lim_{n\infty} \frac{t_1(n) \log t_1(n)}{t_2(n)} = 0.$$

Alors

$$\mathsf{DTIME}(t_1(n)) \subsetneq \mathsf{DTIME}(t_2(n)).$$

La démonstration de ce théorème est similaire à la démonstration précédente, et n'est donc pas explicitée ici. Le facteur $\log n$ vient du problème suivant : quand on parle de la complexité en temps d'une machine de Turing, on ne précise pas le nombre de ruban de cette machine. Seulement, pour faire une démonstration, on a besoin de connaître ce nombre, sans quoi on ne peut pas raisonner. Or on sait que n'importe quelle machine de Turing à k rubans peut être simulée par une machine de Turing à deux rubans *modulo* une augmentation de la complexité d'un facteur $\log n$.

5.3 Conséquences des théorèmes

Ces deux théorèmes sont relativement riches en conséquences. La plupart des résultats de hiérarchies sont obtenus comme corollaires plus ou moins lointains de ses résultats. Un résultat important découlant de ces théorèmes est le fait que la classe PSPACE soit différente de la classe des langages reconnus en temps exponentiel. Par contre, il est amusant de remarquer que si l'on sait que ces deux classes sont différentes, on ne sait en revanche pas quelle relation d'inclusion il y a entre les deux, ni même s'il y en a une !

On commence par quelques conséquences immédiates. Puisque pour tous k et k' strictement supérieurs à 1,

$$\lim_{n\infty} \frac{n^k \log n}{2^{k'n}} = 0,$$

le théorème de hiérarchie déterministe pour le temps assure que la classe des langages reconnus en temps polynomial est strictement plus petite que celle des langages reconnus en temps exponentiel. En effet, pour tous k et k', le théorème assure que

$$\mathsf{DTIME}(n^k) \subsetneq \mathsf{DTIME}(2^{k'n}).$$

En faisant l'union sur tous les k et k', on obtient le résultat souhaité. On définit d'abord la classe EXPTIME.

Définition 5.9.

$$\mathsf{EXPTIME} = \bigcup_{k>1} \mathsf{DTIME}(2^{kn})$$

Proposition 5.10.

$$\mathsf{P} \neq \mathsf{EXPTIME}$$

On définit également une classe au dessus de EXPTIME, contenant les langages décidés en $\mathcal{O}(2^{n^k})$ pour un certain k.

Définition 5.11.

$$\mathsf{EXPTIME}_{poly} = \bigcup_{k>1} \mathsf{DTIME}(2^{n^k})$$

La même preuve que précédemment nous donne le résultat suivant.

Proposition 5.12.

$$\mathsf{EXPTIME} \subsetneq \mathsf{EXPTIME}_{poly}$$

Ces résultats triviaux étant énoncés, le lecteur est bien convaincu que l'on pourrait continuer, mais également que ça ne présenterait que peu d'intérêt. On va donc énoncer des résultats un peu plus fort et moins directement liés aux théorèmes.

On dit qu'une fonction f est superpolynomiale lorsque pour tout i,

$$\liminf_{n\infty} \frac{n^i}{f(n)} = 0.$$

Proposition 5.13. *Si f est une fonction vérifiant*

$$\lim_{n\infty} \frac{n^i}{f(n)} = 0$$

pour tout i, alors

(i) $\mathsf{P} \subsetneq \mathsf{DTIME}(f(n))$,

(ii) $\mathsf{PSPACE} \subsetneq \mathsf{DSPACE}(f(n))$.

Preuve.

(i) Si f vérifie les conditions du théorème, f est superpolynomiale, et $f^{1/2}$ également[1]. En appliquant le théorème de hiérarchie déterministe pour le temps aux deux fonctions, on obtient deux relations.

$$\mathsf{DTIME}(f^{1/2}(n)) \subsetneq \mathsf{DTIME}(f(n))$$
$$\mathsf{DTIME}(n^i) \subsetneq \mathsf{DTIME}(f^{1/2}(n))$$

D'où le résultat, obtenu par union sur i.

(ii) La démonstration est complètement similaire dans le cas de l'espace.

□

On montre maintenant quelques résultats sur des classes de complexité célèbres. On en définit une de plus, qui est l'équivalent exact de EXPTIME pour l'espace.

Définition 5.14.

$$\mathsf{EXPSPACE} = \bigcup_{k>1} \mathsf{DSPACE}(2^{nk})$$
$$\mathsf{EXPSPACE}_{poly} = \bigcup_{k>1} \mathsf{DSPACE}(2^{n^k})$$

Proposition 5.15.

(i) $\mathsf{NLOGSPACE} \subsetneq \mathsf{PSPACE} \subsetneq \mathsf{EXPSPACE}$

(ii) $\mathsf{PSPACE} \subsetneq \mathsf{EXPSPACE}_{poly}$

(iii) $\mathsf{P} \subsetneq \mathsf{EXPTIME}_{poly}$

[1] $f^{1/2}$ désigne la fonction $n \mapsto \sqrt{f(n)}$.

Preuve. On ne montre que le premier point. Le troisième est déjà connu, et la démonstration pour le deuxième est semblable à celle pour le troisième.

D'après le théorème 4.2 de Savitch, on a l'inclusion $\mathsf{NLOGSPACE} \subseteq \mathsf{DSPACE}(\log(n)^2)$. Et d'après le théorème de hiérarchie déterministe en espace, on l'inclusion stricte $\mathsf{DSPACE}(\log(n)^2) \subsetneq \mathsf{DSPACE}(n)$. Ceci nous donne la première inclusion. Pour la seconde, il suffit de remarquer avec ce même théorème que pour tout k, $\mathsf{DSPACE}(n^k) \subsetneq \mathsf{DSPACE}(2^n)$. □

On va maintenant démontrer le théorème le plus marquant de cette partie. Il est marquant pour plusieurs raisons. Tout d'abord le résultat établi est un résultat très intéressant. Mais il est également marquant car s'il assure que deux classes de complexité sont différentes, il ne donne aucune indication sur leur position l'une par rapport à l'autre. Actuellement, on ne sait rien dire de plus sur ces classes.

Théorème 5.16.

$$\mathsf{EXPTIME} \neq \mathsf{PSPACE}$$

Preuve. On remarque d'abord que $2^{kn} = \mathcal{O}(2^{n^{3/2}})$ pour tout k. Par conséquent, $\mathsf{EXPTIME} \subseteq \mathsf{DTIME}(2^{n^{3/2}})$. De plus, d'après le théorème de hiérarchie en temps, $\mathsf{DTIME}(2^{n^{3/2}}) \subsetneq \mathsf{DTIME}(2^{n^2})$. Donc

$$\mathsf{EXPTIME} \subsetneq \mathsf{DTIME}(2^{n^2}).$$

Supposons maintenant que $\mathsf{EXPTIME} = \mathsf{PSPACE}$. On va montrer sous cette hypothèse que $\mathsf{DTIME}(2^{n^2}) \subseteq \mathsf{PSPACE}$. Puisqu'on a l'inclusion stricte inverse, on aboutira à une contradiction.

Soit L un langage dans $\mathsf{DTIME}(2^{n^2})$. On considère

$$L' = \left\{ x\#^t \;\middle|\; x \in L \text{ et } t = |x|^2 - x \right\}.$$

$L' \in \mathsf{DTIME}(2^n)$. Comme on a supposé que $\mathsf{EXPTIME} = \mathsf{PSPACE}$, il existe k tel que $L' \in \mathsf{DSPACE}(n^k)$. Soit M une machine de Turing qui décide L' en espace n^k. On définit la machine M' suivante : sur l'entrée x,

- copier x et ajouter $|x|^2 - |x|$ dièses,
- faire fonctionner M sur le mot obtenu.

$L = L(M')$ et la machine M' fonctionne en espace $\mathcal{O}(n^{2k})$. Donc $L \in \mathsf{PSPACE}$. □

On montre maintenant un résultat supplémentaire dépendant de la réponse du problème central de la complexité : $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$! On peut voir une fois de plus que le problème se pose de beaucoup de façons différentes.

Proposition 5.17. *Si* $\mathsf{P} = \mathsf{NP}$*, alors* $\mathsf{EXPTIME}_{poly} = \mathsf{NEXPTIME}_{poly}$.

Définition 5.18.

$$\mathsf{NEXPTIME}_{poly} = \bigcup_k \mathsf{NTIME}(2^{n^k})$$

Preuve. Supposons $\mathsf{P} = \mathsf{NP}$ et soit $L \in \mathsf{NEXPTIME}_{poly}$. On associe à L le langage

$$L' = \left\{ x\#^{2^{|x|^k - |x|}} \;\middle|\; x \in L \right\}$$

et donc, $L' \in \mathsf{NP}$. Comme on a supposé $\mathsf{P} = \mathsf{NP}$, il existe une machine de Turing déterministe qui décide L' en temps polynomial. On conclue de la même façon que pour le théorème précédent. □

5.4 Lemmes de translation

On sait que $\mathsf{DTIME}(2^n) \subseteq \mathsf{DTIME}(n2^n)$. On se demande si l'inclusion est stricte ou non. La réponse nous sera donnée par l'un des lemmes de translation que nous allons exprimer. Ils sont valables pour les complexités en temps et en espace, déterministes et non déterministes. On n'explicite que la version en espace non déterministe. Les énoncés sont quasiment les mêmes, et les preuves des adaptations proches de celles que l'on présentent.

Lemme 5.19. *Soit s_1, s_2 et f des fonctions constructibles pour l'espace, telles que $s_2(n) \geqslant n$ et $f(n) \geqslant n$ pour tout n, alors*

$$\mathsf{NSPACE}(s_1(n)) \subseteq \mathsf{NSPACE}(s_2(n)) \implies \mathsf{NSPACE}(s_1(f(n))) \subseteq \mathsf{NSPACE}(s_2(f(n))).$$

Preuve. Supposons que $\mathsf{NSPACE}(s_1(n)) \subseteq \mathsf{NSPACE}(s_2(n))$ $(*)$ et que $L \in \mathsf{NSPACE}(s_1(f(n)))$. Il existe une machine de Turing non déterministe M_1 qui le reconnaît en espace $s_1(f(n))$. On considère alors le langage $L' = \{x\#^i \mid M_1 \text{ accepte } x \text{ en espace } s_1(|x|+i)\}$.

$L' \in \mathsf{NSPACE}(s_1(n))$: sur l'entrée $x\#^i$, on marque $s_1(|x|+i)$ cellules et on simule M_1 sur x dans l'espace marqué. On accepte si et seulement si M_1 accepte x. Alors comme $(*)$, L' appartient à $\mathsf{NSPACE}(s_2(n))$. Donc il existe une machine de Turing non déterministe M_2 qui le reconnaît en espace $s_2(n)$.

On va considérer la machine M_3 dont on explicite ci-après le fonctionnement. Sur l'entrée x, M_3 marque $f(|x|)$ cellules puis $s_2(f(|x|))$ cellules (ce qui est possible car les deux fonctions sont constructibles pour l'espace). Puis M_3 simule M_2 sur $x\#^i$ pour $i = 0, 1, \ldots$ en comptant le nombre de dièses du mot considéré dans un compteur dont l'espace est au plus $\log i$, et de telle sorte que M_3 accepte x si et seulement si M_2 accepte $x\#^i$ pour un compteur qui n'excède pas $s_2(f(|x|))$.

M_3 fonctionne en espace $s_2(f(n))$.

$L = L(M_3)$: si $x \in L$, alors il existe $i = f(|x|) - |x|$ tel que $x\#^i \in L(M_2)$. Et le fonctionnement de M_3 sur $x\#^i$ montre que $x \in L(M_3)$. Et réciproquement ! □

Avant d'utiliser ce lemme pour répondre à notre question, on utilise cette version que l'on vient de démontrer pour énoncer un théorème de hiérarchie en espace non déterministe.

Théorème 5.20. *Pour tous entiers s et t,*

$$\mathsf{NSPACE}(n^{s/t}) \subseteq \mathsf{NSPACE}(n^{(s+1)/t}).$$

Preuve. Supposons que ce ne soit pas, alors $\mathsf{NSPACE}(n^{s/t}) = \mathsf{NSPACE}(n^{(s+1)/t})$. Donc en particulier $\mathsf{NSPACE}(n^{(s+1)/t}) \subseteq \mathsf{NSPACE}(n^{s/t})$. On considère alors les fonctions f définies par $f(n) = n^{(s+i)/t}$ pour $i \geqslant 0$.

Grâce au lemme de translation, on déduit que $\mathsf{NSPACE}(n^{(s+1)(s+i)}) \subseteq \mathsf{NSPACE}(n^{s(s+i)})$ pour $i = 0, \ldots, s$ $(*)$. Comme pour $i \geqslant 1$, $s(s+i) \leqslant (s+1)(s+i-1)$,

$$\mathsf{NSPACE}(n^{s(s+i)}) \subseteq \mathsf{NSPACE}(n^{(s+1)(s+i-1)}) \ (**).$$

On a ensuite les inclusions suivantes :

$$\begin{array}{lll}
(*) \text{ avec } i = s & \Rightarrow & \mathsf{NSPACE}(n^{(s+1)2s}) \subseteq \mathsf{NSPACE}(n^{2s^2}) \\
(**) \text{ avec } i = s & \Rightarrow & \mathsf{NSPACE}(n^{2s^2}) \subseteq \mathsf{NSPACE}(n^{(s+1)(2s-1)}) \\
(*) \text{ avec } i = s-1 & \Rightarrow & \mathsf{NSPACE}(n^{(s+1)(2s-1)}) \subseteq \mathsf{NSPACE}(n^{s(2s-1)})
\end{array}$$

En continuant ainsi à appliquer nos deux équations, on arrive à la conclusion que $\mathsf{NSPACE}(n^{2s(s+1)}) \subseteq \mathsf{NSPACE}(n^{s^2})$.

D'après le théorème de Savitch, $\mathsf{NSPACE}(n^{s^2}) \subseteq \mathsf{DSPACE}(n^{2s^2})$. Et d'après le théorème de hiérarchie déterministe,

$$\mathsf{DSPACE}(n^{2s^2}) \subsetneq \mathsf{DSPACE}(n^{2s(s+1)}) \subseteq \mathsf{NSPACE}(n^{2s(s+1)}).$$

D'où

$$\mathsf{DSPACE}(n^{2s(s+1)}) \subsetneq \mathsf{DSPACE}(n^{2s(s+1)})\,!$$

□

Voyons maintenant si l'inclusion $\mathsf{DTIME}(2^n) \subseteq \mathsf{DTIME}(n2^n)$ est stricte. On utilise le lemme de translation en version temps déterministe.

Proposition 5.21.

$$\mathsf{DTIME}(2^n) \subsetneq \mathsf{DTIME}(n2^n)$$

Preuve. Supposons que l'inclusion n'est pas stricte. Alors $\mathsf{DTIME}(n2^n) \subseteq \mathsf{DTIME}(2^n)$ $(*)$.

On considère la fonction f définie par $f(n) = 2^n$. En utilisant le lemme de translation pour le temps déterministe, on déduit de $(*)$ que

$$\mathsf{DTIME}(2^n 2^{2^n}) \subseteq \mathsf{DTIME}(2^{2^n}).(**)$$

On considère la fonction f définie par $f(n) = n + 2^n$. De même, on déduit du lemme de translation que

$$\mathsf{DTIME}((n+2^n)2^{n+2^n}) \subseteq \mathsf{DTIME}(2^{n+2^n}).(***)$$

De $(**)$ et $(***)$, on déduit que

$$\mathsf{DTIME}((n+2^n)2^{n+2^n}) \subseteq \mathsf{DTIME}(2^{2^n}).$$

On peut appliquer le théorème de hiérarchie déterministe pour le temps, avec $t_1(n) = 2^{2^n}$ et $t_2(n) = (n+2^n)2^{n+2^n}$, qui donne que

$$\mathsf{DTIME}(2^{2^n}) \subseteq \mathsf{DTIME}((n+2^n)2^{n+2^n})\ldots$$

□

Ces théorèmes de translation sont assez puissants puisqu'ils permettent, quand on ne peut pas directement utiliser les théorèmes de hiérarchies, de triturer les données afin de les appliquer.

5.5 Théorèmes d'union

Dans le domaine de la complexité, et entre autres dans ce cours, on parle souvent de façon approximative de certaines choses. Par exemple, n'importe quel mathématicien qui se respecte aurait les cheveux dressés sur la tête s'il nous entendait parler des classes de complexité $\mathsf{DTIME}(t(n))$, $\mathsf{NSPACE}(s(n))$, ... En effet, $s(n)$ et $t(n)$ ne sont pas des fonctions, et on devrait parler plutôt de $\mathsf{DTIME}(t)$. Heureusement, les mathématiciens ne se penchent pas là-dessus, ce qui nous permet d'utiliser ces notations moins lourdes lorsqu'il s'agit par exemple de parler de $\mathsf{DTIME}(n(\log n)^2)$ sans avoir spécialement envie de donner un nom à cette fonction.

Après cette petite digression, nous allons revenir sur une imprécision qui peut paraître elle réellement gênante. Cela dit, le résultat que l'on démontre est rarement connu et les gens ne se posent que peu la question. On parle des classes de complexité P, NP, PSPACE à tour de bras. Cependant, sont-ce réellement des classes de complexité ? La définition d'une classe requiert l'existence d'une fonction telle que les langages de la classe soient reconnus en temps ou espace borné par cette fonction. En d'autres termes, existe-t-il des fonctions récursives t_{P}, t_{NP}, s_{PSPACE} définissant ces classes ?

Pour répondre à cette question, on peut déjà remarquer que toutes ces classes plus générales sont définies de la même façon : ce sont des unions de classes *bien définies*. Une autre propriété est également vérifiée, c'est que la suite des fonctions utilisées pour faire l'union est strictement croissantes. Ceci nous donne exactement les conditions des théorèmes d'union, existant pour le cas du temps et celui de l'espace. On énonce ici celui pour l'espace.

Théorème 5.22. *Soit $(f_i)_i$ une suite récursivement énumérable de fonctions totales récursives telles que pour tout i et tout n, $f_i(n) < f_{i+1}(n)$. Alors il existe une fonction totale récursive s telle que*

$$\mathsf{DSPACE}(s(n)) = \bigcup_i \mathsf{DSPACE}(f_i(n)).$$

Corollaire 5.23. PSPACE *est une classe de complexité !*

Preuve du théorème. On cherche à construire une fonction s totale récursive telle que

(i) $f_i(n) \leqslant s(n)$ presque partout pour tout i (ce qui assure l'inclusion $\bigcup_i \mathsf{DSPACE}(f_i(n)) \subseteq \mathsf{DSPACE}(s(n))$),

(ii) si s_j est la complexité en espace d'une machine de Turing M_j telle que pour tout i, $s_j(n) > f_i(n)$ infiniment souvent, alors $s_j(n) > s(n)$ infiniment souvent (ce qui assure l'égalité).

Construction de s : On part d'une énumération M_j des machines de Turing. On désigne par s_j la complexité en espace de M_j. On a deux suites récursivement énumérables (M_j) et (f_k). On construit s par l'intermédiaire d'une liste L de couples (i_j, k) où i_j représente une machine de Turing et k une fonction. L'algorithme est le suivant :

```
1 L = ∅;
2 pour n ⩾ 1 faire
3     si pour tout (i_j, k) ∈ L, s_j(n) ⩽ f_k(n) alors
4         ajouter (i_n, n) à L et poser s(n) = f_n(n);
5     sinon
6         c'est qu'il existe dans L un couple (i_j, k) tel que s_j(n) > f_k(n). Le nombre de ces
          couples ayant cette propriété est fini. On choisit le plus petit indice k_0 parmi eux,
          puis le plus petit i_{j_0} correspondant. ;
7         poser s(n) = f_{k_0}(n);
8         remplacer (i_{j_0}, k_0) par le couple (i_{j_0}, n);
9         ajouter (i_n, n) à L;
```

On définit bien ainsi une fonction s récursive totale. Reste à vérifier qu'elle satisfait bien les conditions (i) et (ii).

(i) Il s'agit de vérifier que pour tout i, $f_i(n) \leqslant s(n)$ presque partout. $s(n)$ est défini soit au pas 2 soit au pas 2. On peut se contenter de ne vérifier l'inégalité que pour $i < n$ car elle doit être vérifiée presque partout. Si $s(n)$ est défini en 2, $s(n) = f_n(n) > f_i(n)$. La difficulté vient du 2. $s(n)$ est défini par un couple (i_j, k). Dans la liste L, à ce stade du calcul, figure un couple (i_j, i) car $i < n$. Soit on l'utilise pour définir $s(n)$ et alors $s(n) = f_i(n)$. Soit on ne l'utilise pas, et on utilise un couple (i_j, k) tel que $k < i$. On calcule en fait $s(n+1)$, $s(n+2)$, ... jusqu'à ce que le calcul de $s(m)$ $(m \geqslant n+1)$ utilise un couple (i_j, i). Et alors $s(m) = f_i(m)$.

(ii) Si j est tel que pour tout i, $s_j(n) > f_i(n)$ infiniment souvent, il existe une suite $n_0, n_1, \ldots$ telle que pour tout m, $s_j(n_m) > f_i(n_m)$.

Considérons $s(n_m)$ pour un élément de cette suite. Comme $s_j(n_m) > f_i(n_m)$, le couple (i_j, i) appartient à la liste obtenue dans le processus de définition de s. Soit c'est un tel

couple qui définit $s(n_m)$ et alors $s(n_m) = f_i(n_m) < s_j(n_m)$. Soit c'est un couple (i_j, k) avec $k < i$ qui définit $s(n_m)$ et alors $s(n_m) = f_k(n_m) < f_i(n_m) < s_j(n_m)$.

□

Bibliographie

[1] Sanjeev Arora and Boaz Barak. *Computational Complexity : A Modern Approach*. Cambridge University Press, 2009.

[2] Lane A. Hemaspaandra and Mitsunori Ogihara. *The Complexity Theory Companion*. Springer, 2002.

[3] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. pas la bonne edition.

[4] Dexter Kozen. *Theory of Computation*. Springer, 2006.

[5] Piergiorgio Odifreddi. *Classical Recursion Theory*. North-Holland, Amsterdam, Vol. 1, 1989, Vol. 2, 1999.

[6] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley Reading, Mass, 1994.

[7] Michael Sipser. *Introduction to the Theory of Computation*. 1997.