Notes de cours préliminaires
Cours de complexité, second semestre, première année, magistère STIC

Ralf Treinen
Département Informatique
ENS de Cachan
treinen@lsv.ens-cachan.fr

20 mai 2004

Table des matières

Chapitre 1

Introduction

Objectif : étudier les classes de complexité des problèmes calculatoires.

1.1 Pré-requis

- du cours *calculabilité*
  - les machines de Turing, être habitué à des manipulations des machines de Turing.
  - Les classes P et NP devraient être connues (même si les définitions seront rappelées dans le cours).
  - NP-complétude, la preuve du théorème de Cook (SAT est NP-complet), et quelques autres problèmes NP-complets (3-SAT, circuit Hamiltonien, etc.)
- du cours *algorithmique*
  - la capacité de déterminer la consommation d'espace et de temps des algorithmes
  - connaître quelques algorithmes efficaces sur les graphes, notamment le problème d'accessibilité est en P
- du cours *logique*
  - logique propositionnelle, les circuits
  - un peu de logique du deuxième ordre (sera expliqué en cours)
- du cours *langages*
  - les langages réguliers est sous contexte

1.2 Plan

1. Le modèle de machine de Turing que nous allons utiliser dans ce cours comme modèle de référence. Consommation d'espace et de temps des machines déterministes et non-déterministes.

   Variantes, notamment la réduction du nombre de rubans.

   Les facteurs constants n'importent pas, même pour ce modèle de calcul très précis.

   Conséquence : on peut passer à une notion $O(f)$.

   Par contre, les classes de complexité sont différentes quand les fonctions sont suffisamment différentes (par exemple, $DTIME(n^4)$ est strictement plus grand que $DTIME(n^3)$). Ce sont les *théorèmes de hierarchie*.

2. Puis nous allons nous intéresser à des relations entre des *mesures* de complexité. Si on utilise $f(n)$ d'espace , qu'est-ce qu'on peut en conclure sur la consommation en temps, et inversement ? On va étudier cette question en conjonction avec la question : combien est-ce qu'on perd (dans le pire des cas) en espace et en temps dans la déterminisation des machines ?
3. Puis nous allons passer à une classification plus grossière des classes de complexité et ne considérer que des classes comme logarithmique, polynomial, exponentiel, etc. À partir de ce moment la question du modèle de calcul pert son importance (quand il reste "raisonable", comme des RAM, et séquentiel)

   Pour commencer ce sont les classes P et NP qui nous intéressent. Nous allons étudier les notions différentes de réduction d'un problème vers un autre (puisqu'il y en a bien plusieures qui ont toutes leur intérêt), et étudier la question de la relation entre P et NP en plus de profondeur que c'était le cas dans le cours de calculabilité.
4. Puis nous allons regarder les classes de complexité liées aux modèles de calcul parallèle (circuits, PRAM), et leur relation avec la classe (séquentielle) LOGSPACE.
5. Puis nous jetterons un regard vers le haut : Le hierarchie polynomiale, qui parle par exemple des problèmes qu'on peut résoudre en temps polynomiale supposant qu'on ait un oracle qui sait résoudre un problème en NP, puis la classe des problèmes qu'on peut résoudre en temps polynomial si on dispose d'un oracle pour cette dernière classe, etc. Toutes ces classes sont incluses dans la classe PSPACE des problèmes qu'on peut résoudre en espace polynomial. Cette classe peut être caractérisée par un problème d'existence d'une stratégie gagnante dans une certaine classe de jeux.
6. Pour terminer nous regarderons quelques résultats pour les classes alternantes. On a par exemple que la classe "PSPACE par des machines alternantes" est égale à la classe "EXPTIME par des machines déterministes".

1.3 Remarques divers

Théorie A et Théorie B. Dans la communauté de l'informatique théorique il y a une dichotomie entre "informatique théorique à l'Européenne", et "l'informatique théorique à l'Américaine". Le premier terme dénote une approche où on considère comme le coeur de l'informatique théorique la logique, les automates, les méthodes formelles, tandis que le deuxième dénote une approche qui prend comme centre d'intérêt l'algorithmique et la complexité. On peut bien constater cette dichotomie dans les différences dans les cursus d'informatique en Europe et en Amérique, et aussi dans les activités de recherche principales dans ces deux mondes. On parle aussi parfois de "théorie A" et "théorie B", en allusion au "Handbook of Theoretical Computer Science" qui consiste d'un tome A (Algorithms and Complexity) et d'un tome B (Formal Methods and Semantics).

Pourtant cette séparation est en partie artificielle, et il est important de connaître les résultats principaux des deux domaines. Les questions de complexité sont bien sûr centrales pour toutes les questions algorithmiques en logique, et inversement la logique joue un rôle important dans l'étude de la complexité. Il y a même une branche de la complexité qui essaye de caractériser les classes de complexité par des logiques différentes et de voir ainsi les relations entre classes de complexité comme des relations entres des logiques différentes (*descriptive complexity theory*). Nous allons voir un théorème fondateur de ce domaine, c'est le théorème de Fagin.

L'ouvrage principal conseillé pour ce cours [Pap94] suit une approche conciliatrice et insiste sur l'importance de la logique pour la complexité.

Quelques mots sur l'histoire. La plupart des résultats date des années 70. Le grand coup de ce domaine était évidemment le théorème de Cook qui a déclenché énormément de travaux. À cause du grand âge (pour l'informatique) de ce domaine et son importance il y a énormément de résultats et une prolifération effrayante de définitions (voir par exemple [Aar04]). Il peut étonner que, malgré tout l'apparat technique qui a été développé, les questions centrales (et très simples) sont toujours ouvertes, comme P=NP (nous verrons dans ce cours des autres du même genre). Malgré l'âge du domaine il y a aussi des résultats fondamentaux et relativement récents, comme par exemple le théorème d'Immerman qui était par exemple inconnu au moment où Hopcroft et Ullman écrivaient leur ouvrage classique [HU79] (en fait ce théorème fait quasiment obsolète une petite partie de la théorie développée en Hopcroft/Ullman). Des autres résultats récents viennent de l'algorithmique, par exemple l'algorithme polynomial pour tester si un nombre est premier a fait grand bruit en août 2002.

***Complex* ne veut pas forcement dire *mauvais*.** Un aspect important est qu'une grande complexité n'est pas toujours une mauvaise chose. C'est notamment dans la cryptographie où on aimera bien avoir une certitude que certains problèmes sont nécessairement coûteux à résoudre. Actuellement on a pas cette certitude - en fait on utilise en cryptographie des problèmes dont on ne sait parfois même pas s'ils sont NP-durs (nous en verrons un exemple, c'est le problème d'isomorphisme de graphes).

Les preuves. Le style des preuves dans ce domaine est, en comparaison avec les preuves qu'on a normalement dans les cours "langages" ou “logiques”, différent. Les preuves seront souvent dans un style qui rappelle plutôt les preuves vues en “calculabilité”. La raison est que les constructions et manipulations des machines de Turing sont souvent tellement casse-pied à écrire formellement qu'on se contente souvent de les écrire de façon informelle. Pourtant il faut être sûr qu'il n'y pas de “trous cachés” dans l'argumentation.

Chapitre 2

Préliminaires

Une fonction $f \colon \mathbb{N} \to \mathbb{N}$ est dite *en* $O(g)$ pour une fonction $g \colon \mathbb{N} \to \mathbb{N}$ s'il y a un $n_0 \in \mathbb{N}$ et une constante $c \in \mathbb{N}$ tels

$$\text{pour tout } n \geq n_0 : f(n) \leq c * g(n)$$

Machines de Turing : Q est l'ensemble d'états, Σ l'alphabet d'entrée, Γ l'alphabet de ruban.

Si $Prop$ est une propriété des ensembles (qui sont des sous-ensembles d'un univers U fixé en avance) alors on dit qu'un ensemble est « co-$Prop$ » si son complément par rapport à U a la propriété $Prop$ (ou « est $Prop$ »). Par exemple on dit qu'un langage $L \subseteq \Sigma^*$ est co-fini ssi $\Sigma^* - L$ est fini.

log est le logarithme à base 2. $\log^i x$ dénote $(\log x)^i$.

$\lceil x \rceil$ est le plus petit entier qui est supérieur ou égal à x.

Chapitre 3

Classes de complexités basées sur les machines de Turing

3.1 Définitions

Notre modèle de référence d'un machine de Turing :

– nombre quelconque de rubans (mais voir au-dessous pour des restrictions possibles du nombre de rubans)
– les rubans sont infinis en deux directions (pas essentiel)
– l'entrée est au début sur le premier ruban. Parfois (voir au-dessous) on considère des machines *off-line* où l'entrée est notée sur un ruban à part qui est en lecture seulement.

Definition 1 *Soit* $f\colon \mathbb{N} \to \mathbb{R}$*. Une machine* M *avec alphabet d'entrée* Σ *est*

1. f-bornée en temps *si pour tout* $x \in \Sigma^*$ *aucun calcul de* M *sur* x *prend plus temps que* $\lceil f(|x|) \rceil$
2. f-bornée en espace *si pour tout* $x \in \Sigma^*$ *aucun calcul de* M *sur* x *n'utilise sur tous les rubans confondus plus de cases que* $\max(1, \lceil f(|x|\rceil)$*. Pour des machines offline l'espace occupé par le ruban d'entrée ne compte pas.*

Remarques :

– $|x|$ est la longueur du mot x.
– On dit *n-borné* au lieu de $\lambda n.n$-borné, etc.
– Pour la complexité en temps on ne considère que des fonctions f avec $f(n) \geq n$, les machines qui ne regardent pas toute leur entrée sont donc excluses.
– Pour la consommation d'espace on prend le max avec 1 parce que tout calcul évidemment touche au moins une case, ça nous permet de dire qu'une machine est n-bornée au lieu de $\max(1, n)$-bornée.
– Le nombre de rubans n'entre pas dans la consommation en espace, ni la taille de l'alphabet ou le nombre d'états.
– Le fait que pour les machines offline le ruban d'entrée ne compte pas nous permet d'étudier des machines qui prennent par exemple espace $\log(n)$.
– En ce qui concerne les machines non-déterministes on trouve parfois une définition différente dans la littérature, où on prend le minimum sur tous les calculs acceptants (notamment dans [GJ79]).

Definition 2 *Un langage $L \subseteq \Sigma^*$ est en*
- *DTIME(f) s'il est accepté par une machine déterministe f-bornée dans le temps.*
- *NTIME(f) s'il est accepté par une machine non-déterministe f-bornée dans le temps.*
- *DSPACE(f) s'il est accepté par une machine déterministe f-bornée dans l'espace. On considère des machines offline quand $f(n) < n$ pour quelque n.*
- *NSPACE(f) s'il est accepté par une machine non-déterministe f-bornée dans l'espace (même remarque pour des machines offline).*

Remarques :
- Pour l'instant on n'utilise pas la notation $O(\cdot)$, en fait ça sera une conséquence des résultats de ce chapitre que ça n'a pas de sens de distinguer entre des classes f et $c * f$ où c est une constante.
- Pour l'instant on parle de complexité d'un langage et pas de complexité d'un *problème* (comme 3-SAT). Un tel problème algorithmique se traduit dans un langage formel par un schéma de traduction. Normelement il y a plusieurs traductions possibles pour un problème donné qui diffèrent en longueur du codage résultant par un facteur constant. Dès qu'on saurra que les facteurs constantes n'importent pas on pourra tranquillement aussi parler de la complexité d'un problème sans d'aller trop dans les détails de la question comment représenter une instance de ce problème par un mot de Σ^* (il y a quand-même une chose très importante qu'il ne faut pas oublier : les constantes numériques sont toujours représentées en *binaire*).
- Pour la consommation d'espace on pourrait toujours considérer des machines offline mais quand $f(n) \geq n$ ça n'importe pas (Question : pourquoi ?)

3.2 Classes de complexité en espace

Exemple 1 *Le langage $L = \{wcw^R \mid w \in \{a,b\}^*\}$ où w^R dénote le miroir du mot w (par exemple, $aab^R = baa$) est en DSPACE(n) : On copie la partie de l'entrée jusqu'au c sur un deuxième ruban, puis on compare le reste de l'entrée avec le contenu du deuxième ruban en allant en derrière (simulation d'un automate à pile).*

Exercice 1
Montrer que le langage L de l'exemple 1 est même en DSPACE(log(n)).

Exercice 2
À quelle classes de langages bien connues correspondent DSPACE(1) et NSPACE(1) ? Même question pour DSPACE(c) et NSPACE(c) où $c \in \mathbb{N}$ est une constante quelconque ?

On voit déjà ici l'intérêt des machines offline.

Lemme 1 *Si L est accepté par une machine M_1 f-bornée en espace et $f(n) \geq \log_2 n$ pour tous n alors L est aussi accepté par une machine M_2 f-bornée en espace qui s'arrête sur toutes les entrées.*
Si M_1 est déterministe alors M_2 l'est aussi.

Exercice 3
Montrer le lemme 1. Attention au fait que le calcul d'une valeur $f(n)$ peut prendre plus de place que $f(n)$. Pourquoi l'hypothèse $f(n) \geq \log_2 n$?

3.2.1 Compression du ruban

Théorème 1 *Pour toute constante c,* $DSPACE(f(n)) = DSPACE(f(n) - c)$ *et* $NSPACE(f(n)) = NSPACE(f(n) - c)$.

Remarque : noter que la définition de DSPACE et NSPACE fait le max avec 1.
Démonstration: Représenter les c premiers cases dans l'état. □

Théorème 2 *Si* $r > 0$ *alors* $DSPACE(f(n)) = DSPACE(r * f(n))$ *et* $NSPACE(f(n)) = NSPACE(r * f(n))$.

Démonstration: Il suffit de considérer le cas $r > 1$ (sinon on passe de r à $\frac{1}{r}$), et de supposer (par théorème 3) que la machine est à un seul ruban. L'inclusion $\subseteq$ est évidente. Pour l'autre inclusion compresser $\lceil r \rceil$ symboles du ruban dans un seul (on passe donc du l'alphabet Γ à l'alphabet Γ^r. Représenter la position relative dans un r-bloc dans l'état. □

Corollaire 1 *Si* $f \in O(g)$ *alors* $DSPACE(f) \subseteq DSPACE(g)$ *et* $NSPACE(f) \subseteq NSPACE(g)$.

Démonstration: Si $f \in O(g)$ alors il y a un n_0 tel que que pour tous les $n \geq n_0$: $f(n) \leq c * g(n)$ pour une constante $c \in \mathbb{N}$. Soit $d = max(f(n) \mid n < n_0)$. Par le théorème 1, L est aussi dans DSPACE(f(n)-d). Par le théorème 2, L est même dans $DSPACE(\frac{f(n)-d}{c})$. Cette fonction est partout plus petite que g ; il en suit l'inclusion. □

3.2.2 Réduction du nombre de rubans

Théorème 3 *Si L est accepté par une machine M avec k rubans qui est f-bornée dans l'espace, alors il existe une machine M' avec 1 ruban* sémi-infini *f-bornée en espace qui accepte également L.*
Si M est déterministe alors M' l'est aussi.

Démonstration: Plier les rubans à la position 0 donne $2k$ rubans sémi-infinis.
Passer à un ruban sur un alphabet $(\Gamma \cup \{\blacklozenge\})^{2k}$, où $\blacklozenge$ marque toujours la position de la tête simulée pour ce ruban. Représenter dans l'état quel ruban on est en train de lire ou écrire.
(Cette preuve devrait être bien connue du cours *calculabilité*.) □
Remarque : cette simulation est bien coûteuse en temps. Il faudra faire plus d'effort pour obtenir un résultat analogue pour la complexité en temps.

3.3 Classes de complexité en temps

3.3.1 Accélération

Pour une fonction $f : \mathbb{N} \to \mathbb{R}$ et une constante $c \in \mathbb{N}$ nous notons f_c la fonction

$$f_c(n) = \begin{cases} n & \text{si } n \leq c \\ f(n) + 2c & \text{si n > c} \end{cases}$$

Théorème 4 *Pour toute constante c,* $DTIME(f) = DTIME(f_c)$ *et* $NTIME(f) = NTIME(f_c)$.

Démonstration: Il y a un nombre fini d'entrées de longueur $\leq c$ pour lesquelles on peut donc représenter le résultat dans les transitions de la machine. On peut donc d'abord lire jusqu'à c caractères de l'entrée et, si on trouve la fin, donner la réponse toute de suite. Sinon on revient au début (ce qui nous prend alors temps $2c$) et fait le travail de la machine de départ. □

Remarque : Si on sait que la machine commence son calcul par une lecture complète de son entrée, c'est-à-dire que les $|x|$ premiers étapes d'un calcul sur entrée x bougent la tête sur le ruban d'entrée à droite, alors on peut se passer du "+2c" dans la définition de f_c.

On défini

$$\underline{\lim}_{n\to\infty} f(n) := \lim_{n\to\infty}(\{f(n), f(n+1), f(n+2), \ldots\})$$

Théorème 5 *Soit L accepté par une machine déterministe* M_1 *qui est* f*-bornée en temps et utilise* $k > 1$ *rubans, avec* $\underline{\lim}_{n\to\infty} \frac{f(n)}{n} = \infty$.
Alors pour tout $c > 0$ *il existe une machine déterministe* M_2 *à* k *rubans qui est* $c * f(n)$*-bornée en temps et qui accepte L.*

Démonstration: Première idée : prendre la construction de la preuve du théorème de compression (théorème 2). Cela ne marche pas parce que cette machine fait toujours le même nombre de pas.

Deuxième idée : pré-calculer les étapes de calcul qui se passent à l'intérieur d'un bloc de caractères. Ça « marche presque » mais il est toujours possible que la machine passe son temps à traverser les frontières entre blocs, dans quel cas on ne gagne presque rien.

La bonne idée : on reprend la deuxième idée mais on prend, dans le pré-calcul, de la machine M_1, les voisins à gauche et à droite en considération. Puis, la machine M_2 va simuler une séquence de pas de la machine M_1 qui fait sur au moins un ruban quitter le voisin gauche vers la gauche, ou le voisin droit vers la droite. Dans ce cas M_2 a simulé, dans 8 étapes (4 pour lire, 4 pour écrire), autant d'étapes de M_1 qu'est la taille des blocs (qui reste à déterminer).

Tout au début, on copie l'entrée x vers un autre ruban (rappel : $k > 1$) en compressant en blocs de m caractères ($|x|$ étapes), puis on remet la tête du ruban sur lequel on a copié l'éntrée au debut ($\lceil \frac{|x|}{m} \rceil$ étapes), et puis on commence à simuler la machines M_1 en respectant le fait que l'entrée compressée est maintenant sur un autre ruban. On copiant le mot on a écrasé le mot d'entrée.

Donc le temps utilisé par M_2 pour un mot d'entrée x de longueur n est

$$\begin{array}{ll} & n + \lceil n/m \rceil + 8 * \lceil f(n)/m \rceil \\ \leq & n + n/m + 8 * f(n)/m + 9 \end{array}$$

Puisque $\underline{\lim}_{n\to\infty} \frac{f(n)}{n} = \infty$ on a

$$\forall d \exists n_d \forall n \geq n_d : n \leq \frac{f(n)}{d}$$

Donc, si $n \geq 9$ et $n \geq n_d$ (nous allons déterminer la valeur de d plus tard), le temps pris par M_2 est majoré par

$$2\frac{f(n)}{d} + \frac{f(n)}{dm} + 8\frac{f(n)}{m} = f(n)\left(\frac{8}{m} + \frac{2}{d} + \frac{1}{md}\right)$$

Nous choisissons m tel que $cm \geq 16$ et d tel que $2m + 1 \leq 8d$. On a alors

$$\begin{aligned} & 2m+1 \leq 8d \\ \Rightarrow\quad & \frac{2m}{d} + \frac{1}{d} \leq 8 \\ \Rightarrow\quad & 8 + \frac{2m}{d} + \frac{1}{d} \leq 16 \leq cm \\ \Rightarrow\quad & \frac{8}{m} + \frac{2}{d} + \frac{1}{md} \leq c \end{aligned}$$

C'est bien ce qu'on veut. Par le théorème 4 et la remarque, les petites valeurs n'importent pas. □

Corollaire 2 *Si* $\underline{\lim}_{n\to\infty} \frac{f(n)}{n} = \infty$ *et* $c > 0$ *alors* $DTIME(f(n)) = DTIME(c * f(n))$.

Démonstration: Suit du théorème précédent. Une machine avec un ruban passe aussi comme une machine à deux rubans. □

Corollaire 3 *Si* $f(n) = d * n$ *pour un* $d > 1$ *alors pour tout* $\epsilon > 0$ *on a que* $DTIME(f(n)) = DTIME((1+\epsilon)f(n))$.

Démonstration: Comme au-dessus. Il faut alors que

$$\begin{aligned} & n + \frac{n}{m} + 8\frac{dn}{m} + 2 \leq (1+\epsilon)n \\ \Leftrightarrow\quad & \frac{2}{n} + \frac{1}{m} + \frac{8d}{m} \leq \epsilon \end{aligned}$$

Donc c'est bon si $m \geq \frac{1+8d}{\epsilon}$ (quand n est suffisamment grand). □

En fait on a les mêmes résultats d'accélération pour NTIME.

Corollaire 4 *Si* f *en* $O(g)$*, et les conditions du théorème d'accélération s'appliquent à* g*, alors* $DTIME(f) \subseteq DTIME(g)$ *et* $NTIME(f) \subseteq NTIME(g)$.

Démonstration: Soit L accepté par une machine f-bornée en temps. Si $f \in O(g)$ alors il y a un n_0 tel que pour tous les $n \geq n_0$: $f(n) \leq c * g(n)$.

Si $n_0 = 0$ alors L est acceptée par une machine $c*g(n)$-bornée en temps, alors par le théorème 5 aussi par une machine $\frac{1}{c} * c * g(n) = g(n)$-bornée en temps (si g satisfait les conditions du théroème 5 alors $c * g$ les satisfait aussi).

Si $n \geq 1$ alors, par le théorème 4, L est accepté par une machine f_{n_0}-bornée en temps. Puisque $n_0 \geq 1$ et $g(n) \geq n$ partout on a que $f_{n_0}(n) \leq 3 * g(n)$ pour tout n, donc L est accepté par une machine $3 * g(n)$-bornée en temps, et en conclut comme dans le premier cas. □

3.3.2 Réduction du nombre de rubans

Exemple 2 *Nous reprenons l'exemple 1. On peut reconnaître ce langage en temps linéaire avec 2 rubans, mais avec 1 ruban il faut un temps quadratic (nous nous passons de la preuve qu'on ne peut pas faire mieux que* n^2 *avec un seul ruban).*

Théorème 6 *Si $L \in DTIME(f)$ alors L est accepté par une machine déterministe avec un seul ruban $f(n)^2$-bornée en temps. (pareil pour NTIME)*

Démonstration: La construction est la même que pour la compression du ruban. On se débrouille pour traverser le ruban une seule fois pour retrouver toutes les têtes. Notez que, on temps $f(n)$, on ne peut consommer que l'espace $f(n)$ (c'est une autre observation triviale mais importante). On obtient donc une simulation avec un seul ruban en temps $c * f(n)^2$ pour une constante c. Puis, on applique l'accélération pour se débarasser du facteur constant. □

Théorème 7 *Si L est accepté par une machine (non-)déterministe à k rubans qui est $f(n)$-bornée en temps, alors L est aussi accepté par une machine à 2 rubans qui est $f(n) * \log(f(n))$-bornée en temps.*

Démonstration: Voir [HU79] page 292. □

Pour résumer la morale de l'histoire jusqu'à ici :
- Les constantes additives et les facteurs constants n'importent pas.
- Les classes en espace sont plus robustes que les classes en temps.

Chapitre 4

Les théorèmes de hiérarchie

4.1 Une infinité de classes

Théorème 8 *Soit $f: \mathbb{N} \to \mathbb{N}$ une fonction récursive et totale. Alors il existe un langage récursif $L \notin DTIME(f)$ (respectivement $DSPACE(f)$, $NTIME(f)$, $NSPACE(f)$).*

Démonstration: La preuve n'est ici donnée que pour DTIME (pour les autres classes la preuve est analogue).

Il y a une fonction totale et récursive qui à chaque mot $x \in \Sigma^*$ associe une machine de Turing T_x, fonction qui est surjective (dans le sens que : pour toute machine de Turing M il existe un mot $x \in \Sigma^*$ tel que $T_x = M$ à quelques détails syntaxiques près qui ne concernent pas la sémantique d'une machine de Turing, comme les noms des états, des noms des symboles du ruban, etc.).

$$L := \{x \in \Sigma^* \mid T_x \text{ n'accepte pas } x \text{ en } \leq f(|x|) \text{ étapes } \}$$

Le langage L est évidemment récursif : étant donné x, on construit T_x et commence à la simuler sur l'entrée x, en décrémentant à chaque étape un compteur qu'on avait initialisé avec la valeur $f(|x|)$. Si on atteint un état acceptant avant que le compteur ne soit 0 alors on accepte, sinon on rejette.

Remarquez que cette simulation est aussi possible dans le cas des machines non-déterministes, il suffit d'exploiter toutes les exécutions possible de la machine T_x, en coupant après temps $f(|x|)$.

Supposons pour l'absurde que L soit accepté par une machine déterministe M qui est f-bornée en temps. Soit $x \in \Sigma^*$ tel que $T_x = M$ (dans le même sens qu'au-dessus), donc T_x est aussi f-bornée en temps et accepte L. On a donc

$$\begin{array}{ll} & x \in L(T_x) \\ \Leftrightarrow & T_x \text{ accepte } x \\ \Leftrightarrow & T_x \text{ accepte } x \text{ en } f(|x|) \text{ étapes} \\ \Leftrightarrow & x \notin L = L(T_x) \end{array}$$

Ce qui est bien absurde. □

Corollaire 5 *Il y a une séquence infinie de classes*

$$DTIME(f_1) \subsetneqq DTIME(f_2) \subsetneqq DTIME(f_3) \subsetneqq \ldots$$

où toutes les fonctions f_i sont récursives. Pareil pour DSPACE, NTIME, NSPACE.

Démonstration: On commence par une fonction quelconque récursive f_1. Si on a f_i on applique le théorème 8, et prend f_{i+1} telle que $L \in DTIME(f_{i+1})$. □

4.2 Le théorème de hiérarchie pour l'espace

Le théorème de hiérarchie pour l'espace dit que les classes $DSPACE(f)$ et $DSPACE(g)$ sont différentes même pour des « petites différences » entre f et g. Pourtant il faut que les fonctions ne soient pas trop farfelues.

Definition 3 *Une fonction f est dite*

1. constructible en espace *s'il y a une machine déterministe T f-bornée en espace et pour tout $n \in \mathbb{N}$ un mot $x \in \Sigma^n$ tel que le calcul de T sur entrée x utilise exactement n cases mémoire.*
2. entièrement constructible en espace *s'il y a une machine déterministe T f-bornée en espace telle que pour tout mot $x \in \Sigma^*$ le calcul de T sur entrée x utilise exactement $f(|x|)$ cases mémoire.*

Notez que la machine T dans le premier cas peut ne pas terminer sur certaines entrées.

En fait toutes les fonctions intéressantes sont entièrement constructibles en espace : tous les polynômes, $\log n$, $n!$, 2^n. De plus, l'ensembles des fonctions entièrement constructibles en espace est clos par multiplication, addition, exponentation, etc.

Exercice 4
Montrer que $\log n$, $n!$, et 2^n sont entièrement constructibles en espace. Montrer que si f et g sont entièrement constructibles en espace alors $f * g$ l'est aussi.

Exercice 5
Montrer : Toute fonction f avec $f(n) \geq n$ pour tous n qui est constructible en espace est aussi *entièrement* constructible en espace.

Théorème 9 (hiérarchie pour DSPACE) *Soient*

1. *f_2 entièrement constructible en espace*
2. *$\forall n : f_1(n), f_2(n) \geq \log_2(n)$*
3. *$\underline{\lim}_{n\to\infty} \frac{f_1(n)}{f_2(n)} = 0$*

Alors il y a un langage $L \in DSPACE(f_2) - DSPACE(f_1)$

Par conséquence, on a par exemple

$$DSPACE(\log n) \subsetneqq DSPACE(n * \log n) \subsetneqq DSPACE(n^2) \subsetneqq DSPACE(n^2 * \log n) \subsetneqq \ldots$$

puisque les inclusions larges sont triviales.

Démonstration: Idée : Diagonalisation (on s'en sert beaucoup dans la théorie de complexité). Construire une machine M f_2-bornée en espace qui diffère de toute machine f_1-bornée en espace.

Il y a une fonction récursive et totale ϕ qui associe à chaque mot $x \in \Sigma^*$ une machine déterministe M_x telle que
– ϕ est surjective (comme dans la preuve du théorème 8)
– pour toute machine M il y a un nombre infini de $x \in \Sigma^*$ avec $M = M_x$
La machine M fonctionne comme suit : sur entrée x,

1. marquer $f_2(|x|)$ cases sur un ruban (c'est possible parce que f_2 est entièrement constructible en espace)
2. simuler M_x sur entrée x. Si la simulation dépasse la région des rubans marquée dans l'étape (1) alors rejeter, sinon accepter si et seulement si M_x rejette. Pour cela nous avons besoin d'un compteur pour détecter des boucles, ce qui est possible en espace $f_2(n)$ parce que $f_2(n) \geq \log n$.

Évidemment M est f_2-bornée en espace. Supposons pour l'absurde que $L(M)$ soit accepté par une machine M_1 f_1-bornée en espace. Puisque $f_1(n) \geq \log n$ nous pouvons supposer (lemme 1) que M_1 s'arrête toujours.

En général, la simulation d'une machine M f_1-bornée en espace avec un alphabet de ruban de cardinalité t prend espace $\lceil \log t \rceil * f_1(n)$. Par la dernière hypothèse il existe n_0 tel que pour tous les $n > n_0$: $\lceil \log t \rceil * f_1(n) \leq f_2(n)$. Par l'hypothèse à la fonction ϕ il existe une $x \in \Sigma^*$ avec $|x| \geq n_0$ et $M_1 = M_x$. M_1 a donc suffisamment de place pour simuler le calcul de M_x sur x, et on obtient que $x \in L(M_1) \Leftrightarrow x \notin L(M_x)$, ce qui est en contradiction avec $M_1 = M_x$. □

Remarque : on peut affaiblir les hypothèses du théorème, voir [HU79] page 298 :

1. f_2 constructible en espace
2. $\forall n : f_1(n) \geq \log_2(n)$
3. $\underline{\lim}_{n\to\infty} \frac{f_1(n)}{f_2(n)} = 0$

Est-ce que la même preuve marche aussi pour NSPACE ? En fait nous avons dans la preuve au-dessus utilisé à un seul endroit que les machines sont déterministes, c'est le fait que la machine M_2 peut simuler M_1 (dans les bornes d'espace expliquées) et accepter si et seulement si la machine simulée refuse. Si la machine à simuler est non-déterministe ça pose évidemment un problème parce qu'il faudra essayer *tous* les calculs possibles, et il n'est pas clair combien ça nous coûtera. Si on suit la même construction on arrive à une machine M_2 qui est $co-NSPACE(f_2)$ au lieu de $NSPACE(f_2)$.

Heureusement, le théorème d'Immerman (théorème 20) nous assure que $NSPACE(f) = co-NSPACE(f)$ si $f(n) \geq \log n$. Il en suit :

Théorème 10 (hiérarchie pour NSPACE) *Le théorème 9 est aussi vrai pour NSPACE.*

4.3 Le théorème de hiérarchie pour le temps

Definition 4 *Une fonction f est* entièrement constructible en temps *s'il y a une machine déterministe M qui pour toute entrée $x \in \Sigma^*$ fait $f(|x|)$ pas.*

Toutes les fonctions « habituelles » sont entièrement constructibles en temps.

Théorème 11 (hiérarchie pour DTIME) *Soient*

1. *f_2 entièrement constructible en temps x*
2. $\underline{\lim}_{n\to\infty} \frac{f_1(n)*\log f_2(n)}{f_2(n)} = 0$

Alors il existe un langage $L \in DTIME(f_2(n)) - DTIME(f_1(n))$

On a donc que $DTIME(n) \subsetneq DTIME(n^2)$ et même $DTIME(n*\log n) \subsetneq DTIME(n^2)$ mais le théorème ne nous dit pas si l'inclusion $DTIME(n) \subseteq DTIME(n*\log n)$ est stricte ou pas.

Démonstration: La preuve est comme la preuve du théorème 9 avec les différences suivantes :

– Au lieu de marquer $f_2(n)$ cases mémoires on fait maintenant en parallèle un calcul de la valeur $f_2(n)$, et on arrête la simulation quand ce calcul de $f_2(n)$ est achevé.

– La machine M_2 a un nombre fixe de rubans mais est censée simuler des machines avec un nombre quelconque de ruban. Pour l'espace ça ne pose pas de problème, à cause du théorème 3. Pour le temps, nous n'avons que le théorème 7, ce qui explique aussi le facteur logarithmique dans l'énoncé du théorème. □

Il y a également un théorème de hiérarchie pour NTIME, voir [Pap94] pour des références.

4.4 Autre résultats intéressants

Si on ne fait pas l'hypothèse que les fonctions soient entièrement constructibles alors on arrive à un résultat bizarre :

Théorème 12 (Borodin Gap Theorem) *Soit g une fonction totale et récursive avec $g(n) \geq n$. Alors il existe une fonction totale et récursive f telle que*

$$DSPACE(f(n)) = DSPACE(g(f(n)))$$

(analogue pour NSPACE, DTIME, NTIME.)

Autrement dit, il y a dans la hiérarchie des classes de complexité des « écarts arbitrairement grands ». Par exemple, il y une fonction f telle que $DSPACE(f(n)) = DSPACE(2^{f(n)})$. L'apparente contradiction avec le théorème de hiérarchie pour DSPACE s'explique par le fait que f n'est pas entièrement constructible en espace.

Démonstration: Soit $M_1, M_2, \ldots$ une énumération des machines de Turing. Soit $S_i(n)$ le nombre maximal de nombre de cases utilisées par M_i sur une entrée de taille n. Notez que $S_i(n)$ peut être ∞ dans le cas où M_i ne termine pas sur certaines entrées de taille n. Par contre, étant donnée une valeur k, on peut facilement décider si $M_i(n) = k$.

Nous choisissons $f(n)$ telle que pour tous les i, $1 \leq i \leq n$: soit $S_i(n) \leq f(n)$, soit $g(f(n)) < S_i(n)$. Le problème est que nous ne pouvons pas calculer les valeurs $S_i(n)$. Pour trouver une valeur pour $f(n)$ nous exécutons le « programme » suivant :

```
j := 1
while exists i (1 <= i <= n) with j+1 <= S_i(n) <= g(j)
do
  j := g(j)
done
return j
```

Évidemment, ce programme termine après au maximum $n+1$ itérations de la boucle avec une valeur comme souhaitée. □

Exercice 6
Compléter la preuve du théorème 12.

Remarquons qu'on peut dans le théorème de Borodin même pour toute fonction récursive f_0 choisir f telle que $f(n) \geq f_0(n)$ pour tous n, il suffit d'initialiser dans la preuve la variable j par $f_0(n)$ au lieu de 1.

Le théorème suivant dit qu'il y a des langages pour lesquelles il n'y a pas de programme optimal :

Théorème 13 (Blum's Speedup Theorem) *Soit r une fonction totale et récursive. Alors il existe une langage L telle que pour toute machine M_1 f_1-bornée en temps qui accepte L, il existe une machine M_2 f_2-bornée en temps qui accepte également L, telle que $r(f_2(n)) \leq f_1(n)$ pour presque tous $n \in \mathbb{N}$.*

Ce théorème se montre par diagonalisation, voir [HU79], section 12.6.

Exercice 7
Montrer : il existe un langage infini et récursif qui ne contient aucun langage infini et régulier.

Indication : construire son complément par diagonalisation.

Chapitre 5

Relations entre mesures de complexité

5.1 Temps contre espace, déterminisme contre non-déterminisme

Théorème 14 *Pour toute fonction* $f\colon \mathbb{N} \to \mathbb{N}$ *:*

$$\begin{aligned} DTIME(f) &\subseteq DSPACE(f) \\ NTIME(f) &\subseteq NSPACE(f) \end{aligned}$$

Démonstration: Une machine peut en temps $f(n)$ utiliser espace au maximum $f(n)+1$. On conclut en appliquant le lemme 1. □

Théorème 15 *Pour toute fonction* $f\colon \mathbb{N} \to \mathbb{N}$ *telle que* $\forall n : f(n) \geq \log n$ *:*

$$\textit{si } L \in DSPACE(f) \textit{ alors existe } c \in \mathbb{N} \textit{ telle que } L \in DTIME(c^{f(n)})$$

Démonstration: Comme la preuve du théorème 1. □

Théorème 16 *Pour toute fonction* $f\colon \mathbb{N} \to N$ *entièrement constructible en espace :*

$$NTIME(f) \subseteq DSPACE(f)$$

Démonstration: utiliser un ruban supplémentaire pour énumérer tous les choix possibles de la machine non-déterministe. Base du compteur = nombre maximal de choix dans un état donné de la machine. Borner la longueur du compteur par $f(|x|)$. □

Théorème 17 *Pour toute fonction* $f\colon \mathbb{N} \to N$ *entièrement constructible en espace avec* $\forall n$: $f(n) \geq \log n$ *:*

$$\textit{si } L \in NSPACE(f) \textit{ alors existe } c \in \mathbb{N} \textit{ telle que } L \in DTIME(c^{f(n)})$$

Démonstration: Si une machine f-bornée en espace accepte une entrée x alors il y a une séquence d'exécution de longueur $\leq d^{f(n)}$ pour une constante d (comme dans la preuve du lemme 1, c'est ici qu'on utilise que $f(n) \geq \log n$). De plus nous pouvons supposer qu'il y a une unique configuration acceptante.

On construit le graphe de toutes les configurations possibles de la machine, avec une arrête de c_1 vers c_2 ssi la machine peut aller en un pas de c_1 vers c_2. Ce graphe a $d^{f(n)}$ noeuds, on peut donc décider si la configuration acceptante est atteignable à partir de la configuration initiale pour une entrée x en temps $k * (d^{f(n)})^2 \leq (k * d^2)^{f(n)}$.

(L'hypothèse qu'il y a une seule configuration acceptante se discute, mais on voit facilement qu'on peut aussi décider en temps quadratique si une configuration quelconque dont l'état est acceptant, est accessible.)

En fait on n'a pas besoin de construire en avance la matrice d'adjacence du graphe puisqu'on peut décider facilement s'il y a une arrête entre deux nœuds. □

Definition 5 1. $\mathbf{P} = \bigcup_{p \text{ polynôme}} DTIME(p)$

2. $\mathbf{NP} = \bigcup_{p \text{ polynôme}} NTIME(p)$
3. $\mathbf{PSPACE} = \bigcup_{p \text{ polynôme}} DSPACE(p)$
4. $\mathbf{NPSPACE} = \bigcup_{p \text{ polynôme}} NSPACE(p)$

Nous avons donc que

$$DSPACE(\log n) \subseteq NSPACE(\log n) \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$$

et, par le théorème de hiérarchie en espace,

$$DSPACE(\log n) \subsetneq \mathbf{PSPACE}$$

Donc au moins une des inclusions dans la ligne au-dessus est stricte mais on ne sait pas laquelle (c'est une question ouverte).

On obtient donc aussi que $NSPACE(f(n)) \subseteq DSPACE(c^{f(n)})$, mais on peut faire beaucoup mieux :

Théorème 18 (Savitch) *Le problème* accessibilité *est en $DSPACE((\log n)^2)$, où n est le nombre de nœuds du graphe.*

Démonstration: Considérons le programme suivante :

```
procedure path(x,y,i)
(* est-ce qu'il y a un chemin de x vers y de longueur ≤ 2^i ? *)
  if i = 0
  then return (x=y) || adjacent(x,y)
  else  forall nodes z do
    if path(x,z,i-1) && path(z,y,i-1)
    then return true
  done;
  return false
end
path(x,y,log n).
```

où $adjacent(x, y)$ est vrai si et seulement s'il y a une arrête de x vers y.

Pour implanter la procédure récursive il nous faut une pile. Cette pile sert à sauvegarder les paramètres (x, y, i) et variables locales (z) de la procédure. Pour l'appel $\texttt{path}(x, y, \lceil logn) \rceil$ la pile contiendra $\log n$ cadres, et chaque cadre prend place $4 * \log n$. Donc, en total $\log n * 4 * \log n$. □

Corollaire 6 (Savitch) *Si f est entièrement constructible en espace et $\forall n : f(n) \geq \log n$ alors $NSPACE(f(n)) \subseteq DSPACE((f(n))^2)$.*

Démonstration: Accessibilité sur le graphe des configuration, puis théorème 18. □

Corollaire 7

$$PSPACE = NPSPACE$$

Exercice 8
Montrer : il existe une fonction totale et récursive f telle que

$$DTIME(f) = NTIME(f) = DSPACE(f) = NSPACE(f)$$

Indication : la fonction f n'est pas entièrement constructible en temps, voir théorème 12.

En fait le résultat $DTIME(f) \subseteq DSPACE(f)$ peut être amélioré :

Théorème 19 (Hopcroft, Paul, Vaillant) *Si f est entièrement constructible en espace alors*

$$DTIME(f(n)) \subseteq DSPACE\left(\frac{f(n)}{\log f(n)}\right)$$

Ce théorème peut étonner puisqu'il est bien possible qu'une machine f-bornée en temps touche vraiment $f(n)$ cases du ruban. Mais une machine qui en temps $f(n)$ touche $f(n)$ cases du ruban n'est en vérité pas plus puissante qu'un automate fini (pourquoi ?). Un ruban n'est utile que quand on peut revenir à ce qu'on a écrit, dans quel cas on touche forcement moins que $f(n)$ cases différentes en temps $f(n)$.

5.2 Compléments des classes non-déterministes

Definition 6 *Une fonction $f \colon \Sigma^* \to \Sigma*$ est* calculable par une machine non-déterministe f-bornée en espace *s'il y a une machine non-déterministe f-bornée en espace telle que pour toute entrée x*
- *il y une séquence de calcul qui atteint un état acceptant*
- *toutes séquence de calcul qui se termine dans un état acceptant produit sur le ruban de sortie $f(x)$.*

Donc, la machine non-déterministe peut faire des mauvaises choix, mais dans ce cas-là elle doit le détecter plus tard et aller dans un état non-acceptant. La taille du ruban d'entrée et du ruban de sortie ne comptent pas dans la consommation d'espace.

Lemme 2 (Immerman) *On peut calculer en $NSPACE(log(n))$ le nombre de nœuds d'un graphe G qui sont accessibles à partir d'un nœud x, où n est le nombre de nœuds de G.*

Démonstration: Soit $A(i)$ l'ensemble de nœuds accessibles à partir de x par un chemin de longueur $\leq i$, et $S(i) = |A(i)|$. Évidemment $A(0) = \{x\}$ et $S(0) = 1$, et la valeur cherchée est $S(n-1)$.

Nous donnons d'abord un programme naïf pour calculer la *liste* de nœuds accessibles à partir de x. Ce programme est bien trop coûteux en espace pour répondre à l'énoncé du lemme, mais il nous servira comme point de départ dans le programme de Immerman.

```
input x (* the starting node *)
input n (* the number of nodes *)

J := [x];
for i = 1 to n-1 do
  (* J = A(i-1) *)
  K := [];
  foreach node z do
      (* check whether z ∈ A(i) *);
      found := false;
      foreach node y do
        if y ∈ J then
          if y=z || adjacant(y,z) then found := yes fi
        fi
      done
      if found then k := k+1;
  done;
  J := K;
done;
return J
```

Si on veut faire la même récurrence avec les valeurs $S(i)$ au lieu des ensembles $A(i)$ on est bien sûr embêté pour tester $y \in J$.

C'est là où le non-déterminisme vient au secours : Nous devinons les nœuds qui sont en J. Il reste à assurer que notre algorithme non-déterministe va détecter dans le cours d'une exécution s'il s'était trompé dans ses choix plus tôt. Il y a deux possibilités comment l'algorithme non-déterministe peut se tromper :

– il peut deviner qu'un nœud est en J mais en vérité ce n'est pas la cas. Nous vérifions donc que, quand un nœud est choisi d'être en J, il est bien accessible en $i-1$ étapes. Pour cela nous utilisons un algorithme non-déterministe qui dévine simplement le chemin d'accès et le vérifie à la volée.
– il peut deviner qu'un nœud n'est pas en J mais un vérité il l'est. Nous utilisons le fait que nous connaissons la cardinalité de J, donc nous comptons simplement le nombre de choix positifs et vérifions à la fin que ce nombre de choix correspond bien à la cardinalité de J.

```
input x (* the starting node *)
input n (* the number of nodes *)

procedure check-path(y,l)
(* check existance of a path from x to y of length ≤ l *)
```

```
  while l ≠ 0 do
    guess a node z;
    if not(z=y || adjacent(z,y)) then fail fi;
    l := l-1;
  done;
  if z ≠ x then fail fi;
end;

j := 1;
for i = 1 to n-1 do
  (* j = S(i-1) *)
  k := 0
  foreach node z do
      (* check whether z ∈ A(i) *);
      found := false;
      l := j;
      foreach node y do
        guess non-deterministically whether y ∈ A(i-1)
        if guess=yes then
          check-path(y,i-1)
          l := l-1;
          if y=z || adjacant(y,z) then found := yes fi
        fi
      done
      if l ≠ 0 then fail;
      (* here we know that we have guessed correctly A(i-1) *)
      if found then k := k+1;
  done;
  j := k;
done;
return j
```

□

Théorème 20 (Immerman) *Si f avec $\forall n : f(n) \geq \log n$ est entièrement constructible en espace alors $NTIME(f) = co - NTIME(f)$.*

Démonstration: Puisque $f(n) \geq \log n$ il y a $c^{f(n)}$ configurations possibles pour un mot d'entrée x de taille n, et on peut les toutes énumérer puisque f est entièrement constructible en espace. Cela nous donne une représentation implicite du graphe des configurations, sur lequel on applique l'algorithme du lemme 2. Si on trouve qu'à un moment une configuration acceptante est atteignable on répond “non”, sinon on répond “oui”. □

Un petit rappel :

Definition 7 *Un langage $L \subseteq \Sigma^*$ est* sous contexte *s'il est engendré par une grammaire où pour toutes les règles $\alpha \to \beta$ on a que $|\alpha| \leq |\beta|$, ou α est l'axiome S, $\beta = \epsilon$, et S ne parait jamais à droite dans une règle.*

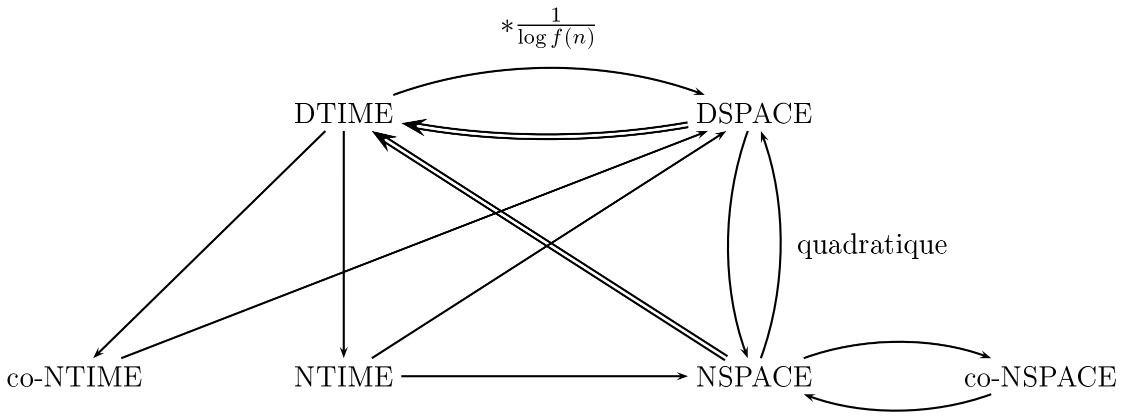

FIG. 5.1 – Relations entre les mesures de complexité. Une single flèche sans décoration dénote une réduction linéaire, une double flèche une réduction à perte exponentielle. Les hypothèses sont omises.

La classe CSL est la classe de tous les langages sous contexte (pour un alphabet Σ donné).

Lemme 3 $CSL = NSPACE(n)$

Exercice 9
Montrer le lemme 3.

Corollaire 8 *CSL est clos sous complément.*

Démonstration: Lemme 3 plus théorème 20. □

Un théorème analogue pour les classes en temps n'est pas connu. Remarquons que, si NP était différent de $co - NP$ alors P serait différent de NP.

Les résultats principaux de ce chapitre sont résumés dans la figure 5.2.

Chapitre 6

Réductions, problèmes durs et complets

6.1 Réduction par traduction

Une notion de *réduction* sert à faire un raisonnement du style suivant :

Si $L_1 \leq L_2$ (L_1 se réduit à L_2)
et si L_2 est dans la classe de complexité C
alors L_1 est aussi dans la classe C

Comment définir la notion de réduction ? La notion de réduction doit nous donner un moyen pour obtenir d'un algorithme pour L_2 en classe C un algorithme pour L_1 en classe C.

Definition 8 *Soient $L_1, L_2 \in \Sigma^*$ deux langages.*

1. L_1 est réductible à L_2 en temps polynômial, $L_1 \leq_m^{ptime} L_2$, *s'il y a une fonction $f\colon \Sigma^* \to \Sigma^*$ qui est calculable en temps polynômial, telle que pour tout $x \in \Sigma^*$:*

$$x \in L_1 \qquad \textit{si et seulement si} \qquad f(x) \in L_2$$

2. L_1 est réductible à L_2 en espace logarithmique, $L_1 \leq_m^{logspace} L_2$ *s'il y a une fonction $f\colon \Sigma^* \to \Sigma^*$ qui est calculable en espace logarithmique, telle que pour tout $x \in \Sigma^*$:*

$$x \in L_1 \qquad \textit{si et seulement si} \qquad f(x) \in L_2$$

Nous rappelons que ni la taille du ruban d'entrée, ni la taille du ruban de sortie comptent pour la consommation d'espace. De plus on suppose que la tête sur le ruban de sortie ne bouge jamais à gauche. Le calcul d'une telle fonction f est toujours fait par une machine déterministe.

On peut ici parler d'une *réduction par traduction* puisqu'on « traduit » une instance d'un problème dans une instance d'un autre problème, puis on travaille sur la traduction pour résoudre le problème de départ. Dans la littérature anglophone on parle de « many-one reducibility » puisque la fonction de traduction peut ne pas être injective[1]. La réduction en temps polynômial est aussi appelée *réduction de Karp*.

[1] L'analogue pour des fonctions de traduction injectives, « one-one reducibility » en anglais, n'a pas beaucoup d'intérêt pour la théorie de complexité mais est utilisé dans la théorie de calculabilité

La première définition est la définition vue dans le cours de calculabilité. Elle est suffisante pour montrer que si $L_1 \leq_{\mathrm{m}}^{\mathrm{ptime}} L_2 \in \mathbf{P}$ alors $L_1 \in \mathbf{P}$ puisque la taille de $f(x)$ est polynômiallement bornée. Par contre l'analogue n'est pas vraie pour la classe **L**.

Lemme 4 *Pour tous langages $A, B \in \Sigma^*$: $A \leq_m^{ptime} B$ implique $\bar{A} \leq_m^{ptime} \bar{B}$.*

En fait ce lemme est vrai pour toutes les notions de réductibilité présentées dans cette section.

Definition 9
1. *$\mathbf{L}$ est la classe des langages reconnaissables en espace déterministe logarithmique.*
2. *$\mathbf{NL}$ est la classe des langages reconnaissables en espace non-déterministe logarithmique.*

Lemme 5 *Les classes $\mathbf{L}$, $\mathbf{NL}$, $\mathbf{P}$, $\mathbf{NP}$ et $\mathbf{PSPACE}$ sont clos sous réduction en espace logarithmique, c'est-à-dire si $L_1 \leq_m^{logspace} L_2 \in C$ pour une classe C de cette liste alors aussi $L_1 \in C$.*

Exercice 10
Le montrer.

Lemme 6 *Si $L_1 \leq_m^{logspace} L_2 \in \mathbf{L}$ alors $L_1 \leq_m^{ptime} L_2 \in P$.*

Démonstration: Comme la preuve du lemme 1. □

Puisque nous nous intéressons principalement pour les classes **L** et **P**, nous allons utiliser la relation $\leq_{\mathrm{m}}^{\mathrm{logspace}}$ comme relation de réduction par défaut. Notez que les réductions en temps polynômial que vous avez vues dans le cours de calculabilité sont même en espace logarithmique, puisqu'en général il suffit de réaliser quelques compteurs sur les rubans de travail.

Par exemple on peut réduire le problème Graphe Hamiltonien vers le problème Satisfiabilité en espace logarithmique : étant donné un graphe G avec n nœuds, on construit l'ensemble suivant des clauses, où la variable x_i^j est censée exprimer que nœud i est le j-ème sur un circuit Hamiltonien :

$$\begin{array}{ll}
x_i^1 \vee \ldots \vee x_i^n & \forall 1 \leq i \leq n \\
x_1^j \vee \ldots \vee x_n^j & \forall 1 \leq j \leq n \\
\neg x_i^j \vee \neg x_i^k & \forall 1 \leq i \leq n, \forall 1 \leq j < k \leq n \\
\neg x_i^j \vee \neg x_k^j & \forall 1 \leq i < k \leq n, \forall 1 \leq j \leq n \\
\neg x_i^j \vee \neg x_k^{j+1} & \text{si pas d'arrête entre } i \text{ et } k
\end{array}$$

On voit bien que quelques compteurs sont suffisants pour construire cet ensemble de clauses.

Il est convenable qu'une relation de réduction soit transitive.

Lemme 7 *$\leq_m^{ptime}$ et $\leq_m^{logspace}$ sont transitives.*

Démonstration: Pour ptime c'est évident, pour logspace l'idée est la même que pour l'exercice 10. □

6.2 Problèmes durs et complets

Les définitions suivantes sont rélatives à une notion de réduction qui est transitive :

Definition 10 *Soit C une classe de complexité.*

1. *Un langage L est C-*dur *(pour une notion de réduction $\leq$) si $L' \leq L$ pour tout $L' \in C$.*
2. *Un langage L est C-*complet *(pour une notion de réduction $\leq$) si $L \in C$ et L est dur pour C.*

Les problèmes C-complets sont les problèmes qui sont « le moins probables » d'être dans une classe plus petite.

Lemme 8 *Le langage*

$$L_{NP} := \{(M, x, 1^n) \mid \textit{la machine } M \textit{ accepte } x \textit{ en au maximum } n \textit{ étapes}\}$$

est NP-complet (pour reductions $\leq_m^{logspace}$).

Exercice 11
Le montrer.

Lemme 9 *Si*

1. *L est dur pour C_1*
2. *C_2 est clos sous réduction*
3. *$L \in C_2$*

alors $C_1 \subseteq C_2$

Démonstration: Soit $L_1 \in C_1$ quel conque. Docn $L_1 \leq L \in C_2$ puisque L est dur pour C_1. Donc $L_1 \in C$ puisque C est clos sous réduction. □

Exemple : S'il y a un algorithme en **P** pour un problème NP-complet alors **P**=**NP**.

Le lemme suivant donne la façon canonique pour montrer qu'un problème est C-complet une fois qu'on connait déjà un autre probleme C-complet :

Lemme 10 *Si*

1. *L_1 est C-complet*
2. *$L_2 \in C$*
3. *$L_1 \leq L_2$*

alors L_2 est C-complet.

Démonstration: Pour montrer que L_2 est C-dur : Soit $L' \in C$ qule conque, alors $L' \leq L_1 \leq L_2$ parsque L_1 est C-dur, donc $L' \leq L_2$ pusique $\leq$ est transitive. □

Exemple : On a pour le langage L_{NP} du lemme 8 : $L_{NP} \leq_{\mathrm{m}}^{\mathrm{logspace}} 3SAT$, donc $3SAT$ est NP-complet (théorème de Cook, voir le cours de *Calculabilité*). Vous avez vu plein d'autres exemples de cette technique dans ce cours.

Exercice 12
La notion de *réductibilité en temps linéaire* est définie comme la réductibilité en temps polynomiâle sauf que la fonction f est calculée en temps linéaire. Montrer qu'il n'y a aucun langage qui est **P**-complet pour réduction en temps linéaire.

6.3 Réduction par table de vérité

Definition 11 *Un langage L_1 est* réductible par table de vérité *à un langage L_2, $L_1 \leq_{tt}^{ptime} L_2$ s'il y a un $n \in \mathbb{N}$, une fonction $f \colon \Sigma \to (\Sigma^*)^n$ calculable en temps polynômial, et une fonction booléenne h à n arguments tel que pour tout $x \in \Sigma^*$*

$$x \in L_1 \qquad ssi \qquad h(f(x)) = \text{TRUE}$$

On définit $\leq_{tt}^{logspace}$ de façon analogue.

Autrement dit, une réduction par table de vérité calcule d'abord n « questions » à L_2, et puis utilise une table de vérité pour trouver la réponse selon les réponses données par la machine de L_2. La notion $\leq^{\mathrm{tt}}$ vient de « truth table » (anglais pour table de vérité). Notez que toutes les « questions » sont calculées en avance et ne peuvent pas dépendre des réponses données pour des questions précédentes.

Exercice 13
Lesquelles des classes **L**, **NL**, **P**, **NP**, **PSPACE** sont closes sous $\leq_{\mathrm{tt}}^{\mathrm{logspace}}$? Attention, il y a une classe dans cette liste pour laquelle la question est équivalente à une question ouverte.

Lemme 11 *Pour tout langage $A \in \Sigma^*$: $A \leq_{tt}^{ptime} \bar{A}$.*

C'est aussi vrai pour la réductibilité de Turing (voir au-dessous).

Exercice 14
Montrer que $\leq_{\mathrm{tt}}^{\mathrm{ptime}}$ est transitive.

6.4 Réduction par oracle

Aussi appelé reduction de *Turing*. C'est la notion de reduction la plus faible, et d'ailleurs la notion utilisée dans le résultat original du théoreme de Cook.

Definition 12 *Une* machine de Turing à oracle *$M^?$ est définit comme une machine de Turing habituelle, avec la différence qu'une telle machine dispose d'un ruban de travail distingué, et a parmi ses états trois états distingués $q^?$, q^y, et q^n. Pour un langage $A \in \Gamma^*$, l'exécution de la machine $M^?$ avec oracle A, notée M^A, a par rapport à une machine habituelle la particularité suivante : quand la machine entre dans l'état $q^?$ et si le mot sur le ruban distingué est en A alors à l'étape suivante la machine se trouve dans l'état q^y, et sinon dans l'état q^n.*

Un langage L_1 est Turing-reductible *à un langage L_2 s'il y a une machine déterministe à oracle $M^?$ telle que M^{L_2} accepte le langage L_1, et $M^?$ est polynomiallement bornée en temps (c'est à dire, M^A est polynomiallement bornée en temps pour tous les oracles A).*

Autrement dit : la machine à oracle peut à tout moment formuler une question à l'oracle (en écrivant la question sur le ruban distingué), et puis continuer le calcul suivant la réponse obtenue de l'oracle. C'est la notion la plus générale d'une réduction entre deux langages.

Exercice 15
Montrer que $\leq_{\mathrm{T}}^{\mathrm{ptime}}$ est transitive.

6.5 Rélations entre notions de réduction

Lemme 12

$$\leq_m^{logspace} \quad \subseteq \quad \leq_m^{ptime} \quad \subseteq \quad \leq_{tt}^{ptime} \quad \subseteq \quad \leq_T^{ptime}$$

La preuve est évidente. On peut montrer qu'en fait

Lemme 13

$$\leq_m^{ptime} \quad \subsetneq \quad \leq_{tt}^{ptime} \quad \subsetneq \quad \leq_T^{ptime}$$

mais les langages connus qui servent comme exemples pour montrer que les inclusions sont propres, sont tous décidables en temps exponentiel. Donc on ne sait pas si les notions de **NP**-complétude sont différentes pour les différentes notions de réductibilité. Pourtant il y a des travaux qui montrent que ces notions de **NP**-complétude sont différentes mais ils se basent sur des hypothèses supplémentaires.

Exercice 16
Montrer : s'il y a deux langages A, B avec $B \in \mathbf{NP}$, $B \neq \emptyset$ et $B \neq \Sigma^*$ tels que $A \not\leq_{\mathrm{m}}^{\mathrm{ptime}} B$ et $A \leq_{\mathrm{T}}^{\mathrm{ptime}} B$ alors $\mathbf{P} \neq \mathbf{NP}$.

Voir l'article [Pav03] pour un survol des résultats et questions ouvertes concernants les notions différentes de reduction.

Chapitre 7

P et NP

Exercice 17
Montrer que **NP** et **P** sont clos sous l'étoile de Kleene :
1. Si $L \in \mathbf{NP}$ alors $L^* \in \mathbf{NP}$
2. Si $L \in \mathbf{P}$ alors $L^* \in \mathbf{P}$

Donner une borne supérieure pour le degré du polynôme de la complexité de L^*, en fonction du degré du polynôme de la complexité de L.

Exercice 18
L'argument suivant pour $\mathbf{P} \neq \mathbf{NP}$ est **faux** :

> *Supposons pour l'absurde que* $\mathbf{P} = \mathbf{NP}$. *Donc, SAT est dans* $\mathbf{P}$, *et il y a un entier* k *tel que* $SAT \in DTIME(n^k)$. *Puisque SAT est complet pour* $\mathbf{NP}$, *tout problème* L *en* $DTIME(n^{k+1}) \subseteq \mathbf{NP}$ *est réductible en espace logarithmique vers SAT, et donc* $DTIME(n^{k+1}) \subseteq DTIME(n^k)$. *Or, par le théorème de hiérarchie pour le temps, il existe un langage* $L \in DTIME(n^{k+1}) - DTIME(n^k)$. *Absurde, donc* $\mathbf{P} \neq \mathbf{NP}$.

Indiquez l'erreur dans cet argument !

7.1 Une caractérisation logique de NP

Le théorème principale de cette section appartient au domaine de la *complexité descriptive*. Dans ce domaine on cherche à caractériser les classes de complexité par des logiques, et donc à ramener la comparaison de classes de complexité vers la comparaison de l'expressivité des logiques.

On peut exprimer des propriétés des graphes par des formules logiques dans le langage suivant que nous appelons le *le langage des graphes* :
- $\mathcal{F} = \emptyset$, il n'y a donc pas de constantes ni symboles de fonctions,
- $\mathcal{P} = \{A/2, = /2\}$, les seules symboles de prédicat sont le symbole binaire A et un symbole binaire $=$.

Tout graphe (non-orienté) G est une structure de ce langage : l'univers est l'ensemble de nœuds de G, et la relation $A(a, b)$ est vraie quand le graphe contient une arrête de a vers b.

Dans la logique du premier ordre on peut exprimer des propriétés comme *il y a un nœud isolé*

$$\exists x \forall y \neg A(x, y)$$

ou *il y a exactement un nœud isolé*

$$\exists x \forall y \neg A(x,y) \wedge \forall x_1, x_2 \Big(\forall y (\neg A(x_1,y) \wedge \neg A(x_2,y)) \to x_1 = x_2\Big)$$

Mais l'expressivité est quand-même très restreinte si on reste dans la logique du premier ordre. Pour exprimer des propriétés plus intéressantes il faut passer à la logique du deuxième ordre.

Definition 13 *Une* formule existentielle du deuxième ordre *du langage $\mathcal{F}, \mathcal{P}$ est une formule de la forme*

$$\exists P_1, \ldots, P_n \phi$$

où $P_1, \ldots, P_n$ sont des variables de prédicat (d'une arité quelconque), et ϕ une formule de la logique du premier ordre du langage $\mathcal{F}, \mathcal{P} \cup \{P_1, \ldots, P_n\}$.

Un ensemble L de graphes finis est dit en Σ^1_1 s'il y a une formule existentielle ϕ du deuxième ordre et dans la langage des graphes telle que pour tout graphe fini G : $G \in L$ ssi $G \models \phi$.

Par exemple, pour tout k, l'ensemble des graphes coloriables avec k couleurs est en Σ^1_1 :

$$\begin{aligned} \exists P_1, \ldots, P_k \quad \Big(& \bigwedge_{i \neq j} \neg \exists x (P_i(x) \wedge P_j(x)) \\ \wedge & \ \forall x \bigvee_i P_i(x) \\ \wedge & \bigwedge_i \neg \exists x, y (P_i(x) \wedge P_i(y) \wedge A(x,y))\Big) \end{aligned}$$

Exercice 19
Montrer que les ensembles suivants de graphes sont en Σ^1_1 :

1. L'ensemble des graphes finis Hamiltoniens
2. L'ensemble des graphes finis connexes.

Lemme 14 *Pour toute formule ϕ du premier ordre et du langage de graphes, le problème « étant donné un graphe G et une valuation α, est-ce que $G, \alpha \models \phi$? » est en* **P**.

Démonstration: Par induction sur la structure de la formule. Notez que la formule est fixe et ne fait pas partie de l'entrée (sinon on aurait une autre complexité). □

Théorème 21 (Fagin) *Pour tout ensemble L de graphes finis :*

$$L \in \mathbf{NP} \Leftrightarrow L \in \Sigma^1_1$$

Démonstration: Pour montrer que tout $L \in \Sigma^1_1$ est en **NP** : deviner les valeurs des variables de prédicats, puis appliquer le lemme 14.

Pour l'autre direction nous allons construire pour toute machine M polynômiallement bornée en temps une formule Φ_M telle que M accepte un graphe G ssi $G \models \phi_M$. Nous supposons que le graphe G est présenté en forme d'une matrice d'adjacence, c'est à dire un graphe avec n nœuds est représenté comme un mot en $\{0,1\}^{n*n}$.

L'idée de la construction ressemble dans quelque sens à la preuve du théorème de Cook : nous expressions à l'aide d'une formule logique qu'il y a un tableau dont toutes les lignes contiennent des configurations de la machine M, la première est une configuration initiale, la dernière une configuration acceptante, et toutes les configurations successives sont construites selon les instructions de la machine de Turing. Une grande différence avec la preuve de Cook est que la construction de cette formule est maintenant uniforme pour la machine M et ne dépend pas du mot d'entrée x. Par contre nous pouvons toujours utiliser notre connaissance du polynôme qui borne le temps de la machine M.

Pour simplifier la preuve nous pouvons supposer que la machine M est n^k bornée en temps pour une constante k (qui dépend de M), et que la machine boucle simplement dès qu'elle arrive dans un état acceptant. Dans ce cas il nous suffit alors d'assurer qu'il y a un tableaux de taille $n^k \times n^k$ dans le sens décrit au-dessus.

La seule difficulté consiste à représenter les coordonnées des $n^k \times n^k$ cases (l'univers de la structure G ne contient que n éléments) tel que nous puissions exprimer à l'aide d'une formule logique la relation de voisinage à gauche, en haut, etc. entre deux cases.

Nous utilisons la même technique qu'en exercice 19 et utilisant un ordre total (quelconque) sur les nœuds pour compter de 1 à n. Une case du tableau est donc représenté par $2k$ nœuds. Les k-uplets de nœuds sont lexicographiquement ordonnés, et nous pouvons facilement donner des formules $min_k(x_1, \ldots, x_n)$ et $max_k(x_1, \ldots, x_n)$, exprimant que le vecteur $\vec{x}$ est le k-uplet minimal, resp. maximal, et une formule $next_k(x_1, \ldots, x_k, y_1, \ldots, y_k)$ qui exprime que le vecteur $\vec{y}$ est le successeur du vecteur $\vec{x}$.

Le « contenu » du tableau est représenté par des variables de prédicats P_σ, une pour chacun symbole σ de la machine (états et symboles de ruban), et d'arité $2k$.

La formule Φ_M a donc la structure suivante (les détails sont omis) : $\exists P_{\sigma_1}, \ldots, P_{\sigma_l}$ suivie de la conjonction de formules suivantes :

- P est un ordre total sur les nœuds (comme dans l'exercice 19)
- Les prédicats P_σ font une partition des cases du tableau (comme dans l'exemple que k-colorabilité est en Σ^1_1)
- Le contenu du tableau représente un calcul de la machine M (similaire à la preuve du théorème de Cook). C'est ici qu'on se sert de la formule $next_k$.
- La dernière configuration est acceptante :

$$\exists \vec{x}, \vec{y}(max_k(\vec{x}) \wedge \bigvee_{q \in F} P_q(\vec{x}, \vec{y}))$$

- le contenu du ruban de la première configuration du ruban est la matrice d'adjacence du graphe. Nous trouvons donc sur la première ligne dans la première colonne l'état initial q_0, puis dans les n^2 colonnes suivantes la matrice d'ajacence, et après que des blancs :

$$\begin{aligned}
\exists \vec{x}\Big(min_k(\vec{x}) \wedge \forall \vec{y} \qquad & min_k(\vec{y}) \rightarrow P_{q_0}(\vec{x}, \vec{y}) \\
\wedge \quad & min_{k-2}(y_1, \ldots, y_{k-2}) \rightarrow \exists \vec{z}\big(succ_k(\vec{y}, \vec{z}) \\
& \qquad \wedge (A(y_{k-1}, y_k) \rightarrow P_1(\vec{x}, \vec{z})) \\
& \qquad \wedge (\neg A(y_{k-1}, y_k) \rightarrow P_0(\vec{x}, \vec{z}))\big) \\
\wedge \quad & \neg min_{k-2}(y_1, \ldots, y_{k-2}) \rightarrow \exists \vec{z}(succ_k(\vec{x}, \vec{z}) \wedge P_\square(\vec{x}, \vec{z}))\Big)
\end{aligned}$$

□

7.2 co-NP

La classe co-**NP** est la classe des langages dont le complément est en **NP**. Puisque **NP** est clos sous réductions (par traduction) la classe co-**NP** l'est aussi.

Lemme 15 *Un langage L est* **NP**-*complet ssi $\bar{L}$ est co*-**NP**-*complet.*

Démonstration: La fonction de traduction est la même pour L et pour $\bar{L}$. □

Donc, l'ensemble des formules booléennes *valides* est co-**NP**-complet puisque cet ensemble se réduit vers l'ensemble des formules booléennes non-satisfaisables, et inversement (les deux traductions consistent simplement en application d'une négation).

De la même façon on obtient que l'ensemble des formules booléennes valides qui sont en forme disjonctive normale où chaque clause contient trois littéraux, est co-**NP**-complet.

Lemme 16 *S'il y a un $L \in \mathbf{NP}$ qui est co*-**NP**-*complet alors $\mathbf{NP} = co - \mathbf{NP}$.*

Démonstration: Soit $L \in \mathbf{NP}$ et co-**NP**-complet

Pour montrer que $co - \mathbf{NP} \subseteq \mathbf{NP}$: Soit $L' \in co - \mathbf{NP}$, on a donc $L' \leq_{\mathrm{m}}^{\mathrm{logspace}} L \in \mathbf{NP}$, donc $L' \in \mathbf{NP}$.

Pour montrer que $\mathbf{NP} \subseteq co - \mathbf{NP}$: On a que $\bar{L} \in co - \mathbf{NP}$ et $\bar{L}$ est **NP**-complet. Soit $L' \in \mathbf{NP}$. On a que $L' \leq_{\mathrm{m}}^{\mathrm{logspace}} \bar{L} \in co - \mathbf{NP}$, donc $L' \in co - \mathbf{NP}$. □

La question « $\mathbf{NP} \stackrel{?}{=} co - \mathbf{NP}$ » est un problème ouvert. Si $\mathbf{NP} \neq co - \mathbf{NP}$ alors on a évidemment que $\mathbf{P} \neq \mathbf{NP}$.

Bien sûr on a que $\mathbf{P} \subseteq (\mathbf{NP} \cap co - \mathbf{NP})$. Pour longtemps un « candidat » pour $(\mathbf{NP} \cap co - \mathbf{NP}) - \mathbf{P}$ était PRIMES, c'est-à-dire l'ensemble des nombres premiers (écrits en notation binaire). PRIMES est trivialement en co-**NP**(pour montrer qu'un nombre n'est *pas* premier il suffit de deviner une décomposition en deux facteurs), et V. Pratt a montré en 1975 que PRIMES est aussi en **NP**. Très récemment, le résultat suivant faisait beaucoup de bruit :

Théorème 22 (Agrawal, Kayal, et Saxena) *PRIMES est en* **P**.

7.3 NP minus P

Théorème 23 *Si $\mathbf{P} \neq \mathbf{NP}$ alors il y a langage $L \in \mathbf{NP} - \mathbf{P}$ qui n'est pas* **NP**-*complet.*

Démonstration: Remarquons d'abord qu'il y a une énumération effective $M_1, M_2, \ldots$ de toutes les machines déterministes polynômiallement bornées en temps : on énumère en parallèle toutes les machines déterministes et tous les entiers, et construit pour machine M et entier k la machine qui simule M en incrémentant un compteur à chaque étape, et qui coupe le calcul quand le conteur atteint $|x|^k$. On obtient de façon analogue une énumération $R_1, R_2, \ldots$ de toutes les machines qui écrivent sur un ruban et qui sont logarithmiquement bornées en espace (c'est donc une énumération effective de toutes les réductions en logspace).

Soit S une machine déterministe qui décide 3SAT, par exemple en temps exponentiel (la complexité n'a aucune importance pour la preuve).

Nous définissons le langage L simultanément avec une fonction $f : \mathbb{N} \to \mathbb{N}$. La fonction f sera définie par une machine qui représente les entiers en notation unaire (n est représenté comme un mot de longueur n). La question si $x \in L$ ne dépend de la valeur de f que pour l'argument $|x|$, et la valeur de $f(|x|)$ ne dépendra de l'appartenance à L que pour des mot strictement plus courts que x.

$$L := \{x \in \Sigma^* \mid x \in 3SAT \text{ et } f(|x|) \text{ est pair}\}$$

En fait f sera calculable en temps linéaire, ce qui rend L immédiatement en **NP**.

La fonction f est définie par récurrence :

1. $f(0) = 1$
2. pour calculer $f(n+1)$ il y a deux étapes :
   (a) La première étape consiste à calculer $f(0), f(1), \ldots$, mais ce calcul est stoppé quand n étapes sont passés. Soit $f(j) = k$ le dernier calcul achevé (on a donc $j < n$, et quand n tend vers ∞ la valeur correspondante de j tend également vers ∞, mais beaucoup plus lentement). La valeur de $f(n+1)$ peut alors être soit k, soit $k+1$, comme déterminé dans la deuxième étape :
   (b) Dans la deuxième étape il y a deux cas :
      i. si k est pair, $k = 2i$:
      On commence à énumérer tous les mots dans un ordre de longueur croissante. Pour chaque mot z on cherche une différence entre l'appartenance de z à L d'une part et l'acceptation de z par M_i d'autre part. Cette recherche est stoppée après n pas de calcul. Si on a trouvé une différence alors on met $f(n+1) = k+1$, sinon $f(n+1) = k$.
      ii. Si k est impair, $k = 2i + 1$:
      On commence à énumérer tous les mots dans un ordre de longueur croissante. Pour chaque mot z on cherche une différence entre l'acceptation de z par S d'une part et l'appartenance du résultat de R_i sur entrée z à L d'autre part. Cette recherche est stoppée après n pas de calcul. Si on a trouvé une différence alors on met $f(n+1) = k+1$, sinon $f(n+1) = k$.

On peut montrer par induction que pour tous les n, soit $f(n+1) = f(n)$, soit $f(n+1) = f(n) + 1$.

Puis nous montrons que $\lim_{n\to\infty} f(x) = \infty$. Supposons pour l'absurde que ce n'est pas le cas. On a alors que $\exists n_0 \forall n \geq n_0 : f(n) = f(n_0)$.

1. Si $f(n_0)$ est pair il n'y aucune différence entre L et M_i. Mais puisque $f(x)$ est pair pour presque tous les x, $L = 3SAT$ à un nombre fini de mots près (ceux pour lesquels $f(x)$ est impairs). Donc $3SAT$ est décidé par une machine déterministe et polynômiallement bornée en temps (à un nombre fini de mots près, ce qui ne dérange pas), ce qui est en contradiction avec $\mathbf{P} \neq \mathbf{NP}$.
2. Si $f(n_0)$ est impair il n'y aucune différence entre $3SAT$ et $x \mapsto L(R_i(x))$, R_i est donc une réduction en logspace de 3SAT vers L. Or $f(x)$ est impair pour presque tous les x, donc L est un langage fini. On a donc une réduction en logspace de 3SAT vers un langage fini, ce qui est aussi en contradiction avec $\mathbf{P} \neq \mathbf{NP}$.

Donc, pour tous les i, on trouve une différence entre L et le i-ème langage décidable en temps polynômial, ce qui montre que L n'est pas en **P**.

Pareil, pour tous les i, on trouve une différence entre L et la i-ème réduction en logspace de 3SAT ce qui montre que 3SAT n'est pas logspace-réductible vers L, et L n'est donc pas **NP**-complet. □

Bien que cette preuve est constructive on aimerai quand-même voir un problème « naturel » qui est en $\mathbf{NP} - \mathbf{P}$ sans être **NP**-complet. Bien sûr il n'y a aucun problème pour lequel on puisse actuellement démontrer qu'il est dans cette classe (sinon on aurait une preuve de $\mathbf{P} \neq \mathbf{NP}$) mais il y a un candidat : C'est le problème GRAPHE-ISOMORPHISME - état donné deux graphes, est-ce qu'il sont isomorphes ? Ce problème est évidemment en **NP**, mais pour l'instant on ne connaît ni un algorithme en **P** ni une preuve de **NP**-complétude. Le problème d'isomorphisme de graphes est utilisé pour quelques protocoles cryptographiques, notamment pour les « Zero Knowledge Proofs »

Le problème n'est pas à confondre avec le problème SOUSGRAPHE-ISOMORPHISME - étant donné deux graphes, est-ce que le premier est isomorphe à un sous-graphe du deuxième ? Ce deuxième problème est connu d'être **NP**-complet.

7.4 P et NP avec des oracles

Théorème 24 *Il y a un oracle A tel que* $\mathbf{P}^A = \mathbf{NP}^A$

Démonstration: Il suffit de choisir pour A un langage **PSPACE**-complet quelconque (nous en verrons des exemples plus tard). □

Exercice 20
Montrer que $\mathbf{P}^A = \mathbf{NP}^A$ pour tout langage A qui est **PSPACE**-complet.

Théorème 25 *Il y a un oracle B tel que* $\mathbf{P}^B \neq \mathbf{NP}^B$

Démonstration: Nous allons définir un langage B, et puis

$$L := \{0^n \mid \exists x \in B \text{ tel que } |x| = n\}$$

Évidemment on aura que $L \in \mathbf{NP}^B$ puisqu'il suffit de deviner le mot x, et de puis faire vérifier par l'oracle que c'est le bon.

Nous construisons une séquence $(B_i)_{i\geq 0}$ de langages ; et puis $B = \bigcup_{i\geq 0} B_i$. Chaque ensemble B_i ne contiendra que de mots de longueur $\leq i$, et tous les mots en $B_{i+1} - B_i$ sont de longueurs $i+1$. Nous mettons $B_0 = \emptyset$.

Soit $M_1, M_2, \ldots$ une énumération de toutes les machines déterministes à oracle qui sont polynômiallement bornées, et telle que toute machine apparaît un nombre infini de fois dans cette énumération.

Pour construire B_i nous simulons la machine M_i sur entrée 0^i. Pendant cette simulation nous comptons les pas de calcul, et arrêtons la simulation dès que $i^{\log i}$ pas ont été faits. Le problème est que nous ne connaissons pas encore B, nous ne connaissons que B_{i-1}, c'est à dire tous les mots en B qui sont de longueur $< i$. Nous posons simplement que l'oracle répond à toutes les questions de longueur $\geq i$ négativement. Les questions de longueur $\geq i$ sont stockées dans un ensemble X de mots à exclure de B, et notre construction assura que B sera disjoint de la valeur finale de X.

Si la simulation s'arrête avant $i^{\log i}$ pas dans un état non-acceptant, alors nous mettons

$$B_i := B_{i-1} \cup \{x \in \{0,1\}^* \mid |x| = i, x \notin X\}$$

Dans les deux autres cas (la simulation accepte dans la borne du temps, ou elle ne s'arrête pas avant que le temps est épuisé) nous mettons

$$B_i := B_{i-1}$$

Il reste à montrer qu'aucune des machines M_i n'accepte L.

1. Si dans la construction de l'ensemble B_i on avait refusé 0^i alors $0^i \notin L(M_i^B)$, et $0^i \in L$ puisque B contient au moins un mot de longueur i. Ceci est dû au fait qu'à la i-ème étape, la cardinalité de X est au maximum $\sum_{j=1}^{i} j^{\log j} < 2^i$.
2. Si dans la construction de l'ensemble B_i la simulation avait accepté 0^i alors on a que $0^i \in L(M_i^B)$ mais $0^i \notin L$ puisque $B_i = B_{i-1}$
3. Si dans la construction de l'ensemble B_i la simulation ne s'est pas arrêtée dans le bon temps alors c'est parce que cette machine est bornée par un n^k dans le temps mais $i^k > i^{\log i}$. Mais il y aura un $j > i$ tel que $j^k < j^{\log j}$ et on retrouvera cette machine plus tard quand on pourra achever la simulation, et donc se retrouver dans un des deux premiers cas.

□

La conséquence de ces deux théorèmes est que toute preuve de $\mathbf{P} = \mathbf{NP}$ ou de $\mathbf{P} \neq \mathbf{NP}$ ne s'appliquera pas aux machines à oracle.

Chapitre 8

Les classes de complexité parallèles

Dans ce chapitre nous nous intéressons à des classes de complexité qu'on obtient avec des modèles de calculs en parallèle. On supposera qu'on dispose d'un nombre illimité de processeurs. Cette hypothèse est peu réaliste mais nous permettra d'étudier qu'est ce qu'on peut gagner dans le mieux des cas par un calcul parallèle. Il se trouve qu'il y a une connexion intéressante avec les classes **L** et **NL**.

8.1 Algorithmes parallèles

8.1.1 Multiplication de matrices

Pour l'instant on prend un modèle de calcul informel avec un parallélisme non-borné. Notre premier exemple est le problème de multiplier deux matrices d'entiers de dimension $n \times n$.

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j} \qquad i, j = 1, \ldots, n$$

Sur une machine séquentielle on peut faire ce calcul en temps $O(n^3)$ en supposant que chaque opération arithmétique prend un temps constant : il y a n^3 produits à calculer, et $n^2 * (n-1)$ sommes. Toute version parallèle de cet algorithme ferra donc certainement un *travail* (c'est à dire la somme des temps de calculs sur tous les processeurs) qui est $\geq O(n^3)$.

Notre première solution parallèle consiste à calculer tous les n^3 produits en parallèle à l'aide de n^3 processeurs, et puis de calculer les n^2 sommes en parallèle (mais chaque somme est calculée séquentiellement en temps $n-1$). On obtient donc un algorithme parallèle avec n^3 processeurs, en temps $O(n)$, et qui fait en total un travail $O(n^3)$.

La deuxième solution consiste à calculer les sommes sous forme d'un arbre binaire. Dans ce cas nous calculons une somme à n composants avec $\frac{n}{2}$ processeurs en temps $O(\log n)$ avec un travail $O(n)$. Cela donne en total un algorithme en temps $O(\log n)$, avec n^3 processeurs, et travail $O(n^3)$.

Est-ce qu'on peut faire encore mieux ?

– Dans le calcul des sommes il y a n valeurs à combiner à l'aide d'un opérateur binaire, ce qu'on ne peut pas faire plus rapidement que $\log n$. $O(\log n)$ est alors optimal pour le temps.
– Le travail fait par une machine parallèle ne peut pas être inférieur au travail fait par une implantation séquentielle du même algorithme. $O(n^3)$ est donc aussi optimal pour le travail.

– Si on fait un travail $O(n^3)$ en temps $O(\log n)$ on a besoin d'au moins $\frac{n^3}{\log n}$ processeurs. Il reste alors une marge pour le nombre de processeurs.

Notre troisième solution utilise effectivement que $\frac{n^3}{\log n}$ processeurs et fait travail $O(n^3)$ en temps $O(\log n)$. D'abord, chacun des processeurs calcule $\log n$ des produits (cela prend alors temps $\log n$). Puis, on calcule les sommes en forme d'un arbre binaire, sauf que les sous-arbres terminales de hauteur $\log\log n$ sont chacun évalués par un seul processeur. Dans chaque somme, il y a $\frac{n}{\log n}$ de tels arbres, et chacun d'eux a $\log n$ nœuds. Le temps total utilisé est alors $O(\log n)$.

8.1.2 Accessibilité dans un graphe

Il est plus étonnant qu'on peut aussi trouver un algorithme parallèle efficace pour le problème d'accessibilité dans les graphes. Remarquons d'abord qu'une parallélisation naïve de l'algorithme séquentiel, en explorant tous les chemins possibles en parallèle, ne nous fait pas avancer parc que un tel algorithme ne peut pas être plus rapide que le chemin le plus court entre les deux nœuds, et ce chemin peut être de longueur n dans le pire des cas.

Soit $A(G)$ la matrice d'adjacence du graphe G où de plus nous avons mis 1 sur la diagonale, Autrement dit, $A(G)(i,j) = 1$ ssi il y a un chemin de i vers j de longueur ≤ 1. Il est facile de voir que $A^2(G)(i,j) \geq 1$ ssi il y a un chemin de i vers j de longueur ≤ 2. Plus généralement, $A^i(G)(i,j) \geq 1$ ssi il y a un chemin de i vers j de longueur $\leq 2^i$ (on fait, pour ce problème, il aurait suffit de faire une multiplication booléenne de deux matrices, c.-à-d. de considérer que $1 + 1 = 1$). Pour savoir s'il y a un chemin de i vers j il suffit donc de calculer $A^{\lceil \log i \rceil}(G)$, et nous avons vu que cela se fait avec un algorithme parallèle on temps $O(\log^2 i)$ et avec travail $O(n^3 * \log n)$.

8.2 Les PRAM

La *Parallel Random Access Machine* est la version parallèle de la RAM vue en cours de *Calculabilité*. Un programme PRAM pour k processeurs est une séquence $(\pi_1, \ldots, \pi_n)$ de k programmes de RAM. Les particularités sont :

– Chaque processeur à son propre compteur de programme.
– Tous les processeurs avancent de manière synchrone.
– Chaque processeur à son accumulateur, dénoté r_i.
– Tous les registres sont partagés.
– Si deux processeurs essayent d'écrire au même moment dans le même registre alors celui avec l'index le plus petit a la priorité.

Ce modèle de calcul ressemble à un langage de programmation parallèle mais est idéalisé en plusieurs respects :

– L'écriture sur des registres est difficile à gérer (à cause de la stratégie de résolution de conflit)
– La communication entre processeurs est instantanée (par mémoire partagée) mais en vérité elle pose un problème pour des grands réseaux de processeurs.
– La taille des entiers n'est pas bornée (donc une opération arithmétique prend temps constant indépendant de la taille des arguments). C'est un problème hérité des RAM séquentiels.

L'avantage de ce modèle est qu'il permet d'exprimer facilement les algorithmes parallèles comme le programme pour la multiplication de matrices discuté au-dessus.

Le programme doit dépendre du nombre d'arguments, et aussi de la longueur total de tous les arguments. Pour éviter des cas dégénerés on ne considère que des *familles uniformes de programmes PRAM*, ou plus courts des *PRAM uniformes*. Une PRAM uniforme est une fonction calculable en espace logarithmique (par exemple par une machine de Turing) qui à une paire (n, m) en notation unaire associe un programme PRAM (le premier argument est le nombre des arguments au programme, le deuxième la somme des longueurs des tous les arguments en notation binaire).

Definition 14 *Une fonction $f : \mathbb{N}^n \to \mathbb{N}^k$ est calculée par une PRAM uniforme P en temps parallèle f avec g processeurs ssi pour tout $(a_1, \ldots, a_n) \in \mathbb{N}^n$ avec $m = \sum_{i=1}^{n} \lceil \log a_i \rceil$:*
- *$P(n, m)$ utilise $\leq g(m)$ processeurs*
- *le calcul de $P(n, m)$ avec initialement les valeurs $a_1, \ldots, a_n$ dans les registres $r_1, \ldots, r_n$ s'arrête en $\leq f(m)$ pas de calcul, et à la fin les valeurs des registres $r_1, \ldots, r_k$ sont $F(a_1, \ldots, a_n)$.*

Par exemple on peut facilement réaliser des PRAM uniformes pour la multiplication de matrices ou pour le problème d'accessibilité dans les graphes.

8.3 Les circuits

Definition 15 *Un* circuit à n entrées *est un graphe orienté et acyclique (V, E) tel que*
- *$V = \{1, \ldots, n\}$*
- *si $(i, j) \in E$ alors $i < j$*
- *tout nœud a une sorte qui peut être un de $\wedge$, $\vee$, $\neg$, 1, 0, ou une variable propositionnelle $x_1, x_2, \ldots, x_n$, et le nombre d'entrées d'un nœud (0, 1, ou 2) correspond à sa sorte.*
- *pour tous les i il y a exactement un nœud avec sorte x_i.*
- *le nœud n n'a pas de sortie*

La *taille* d'un circuit est son nombre de nœuds.

Un circuit *accepte* $(i_1, \ldots, i_n)$ (avec $i_j \in \{0, 1\}$) si l'évaluation du circuit, où chaque nœud avec sorte x_j a la valeur i_j, donne au nœud n la valeur 1 (la définition exacte est omise).

Une *famille uniforme de circuits C* est une fonction calculable en espace logarithmique (par exemple par une machine de Turing) qui sur entrée 1^n construit un circuit $C(i)$ avec n entrées.

Le langage accepté par une famille uniforme de circuits C est $\{w \in \{0, 1\} \mid C_{|w|} \text{ accepte } w\}$.

Le problème Circuit-Value est : étant donné un circuit sans entrées, est-ce que sa valeur est 1 ?

Théorème 26 Circuit-Value *est* **P***-complet.*

Exercice 21
Démontrer le théorème 26.

Le problème Monotone Circuit-Value est la restriction de Circuit-Value aux circuits qui ne contiennent pas le symbole de négation.

Théorème 27 Monotone Circuit-Value *est* **P***-complet.*

Exercice 22
Démontrer théorème 27.

Les circuits ne traitent que des langages $\subseteq \{0,1\}^*$ mais ce n'est pas une vraie restriction puisqu'on peut recoder tout langage avec un alphabet binaire.

Definition 16 *Un langage $L \in \{0,1\}^*$ est accepté par une famille de circuits de taille polynômiale s'il y a une famille uniforme de circuits C et une constante k telles que pour tout x : $x \in L$ ssi $C_{|x|}$ accepte x et est de taille $\leq n^k$.*

Théorème 28 *$L \in \mathbf{P}$ ssi L est accepté par une famille de circuits de taille polynômiale.*

Démonstration: Si L est en $\mathbf{P}$ alors on associe à n le circuit comme dans la preuve du théorème 26.
L'autre direction est une conséquence du théorème 26. □

8.4 Les circuits comme modèle de calcul parallèle

Si on considère les circuits uniformes comme modèle de calcul parallèle alors la taille du circuit correspond au travail total fait, et sa profondeur (la longueur maximale d'un chemin vers le dernier nœud) au temps de calcul.

Definition 17 *$\mathbf{PT/WK}(f(n), g(n)$ est la classe des langages acceptés par des circuits uniformes de profondeur $O(f(n))$ et de taille $O(g(n))$.*

Le nom vient de « Parallel Time and Work ». On remarque qu'on utilise toute de suite la notion $O(\cdot)$ puisqu'il n'y a pas de théorème d'accélération comme pour les machines séquentielles.
Par exemple, Accessibilité $\in \mathbf{PT/WK}(\log^2 n, n^3 * \log n)$.

8.5 Relations entre PRAM et Circuits

Théorème 29 *Si $L \in \mathbf{PT/WK}(f(n), g(n))$ alors il y a une PRAM uniforme qui calcule la fonction caractéristique de L en temps parallèle $O(f(n))$ et avec $O(\frac{g(n)}{f(n)})$ processeurs.*

Démonstration: On obtient d'abord une solution avec $g(n)$ processeurs en définissant pour chaque porte du circuit un processeur avec un programme qui attend que toutes ces entrées sont disponibles et puis simule la porte. Par exemple pour une porte AND avec entrées i et j, et où d est la longueur maximale d'un chemin vers cette porte :

$$\begin{array}{rl}
1 & SKIP \\
\vdots & \vdots \\
3d & SKIP \\
3d+1 & LOAD\ j \\
3d+2 & JZERO\ 3d+5 \\
3d+3 & LOAD\ k \\
3d+5 & HALT
\end{array}$$

On peut optimiser le nombre de processeurs utilisés en gérant une file d'attente des portes à simuler. □

Théorème 30 *Soit la fonction F calculée par une PRAM uniforme en temps parallèle $f(n)$ avec $g(n)$ processeurs, où*
– $f(n)$ et $g(n)$ peuvent être calculées à partir de n en notation unaire en espace logarithmique
– la machine qui calcule la n-ème PRAM à partir de n est n^k-bornée en temps
Alors la représentation binaire de F est calculée par une famille uniforme de circuits de
– profondeur $O(f(n) * (f(n) + \log n))$
– taille $O(g(n) * f(n) * (n^k * f(n) + g(n)))$

Voir [Pap94] chapitre 15 pour la preuve.

8.6 La classe NC

Definition 18

$$\begin{aligned} \mathbf{NC}_i &:= \bigcup_{k\geq 1} \mathbf{PT/WK}(\log^i n, n^k) \\ \mathbf{NC} &:= \bigcup_{i\geq 1} \mathbf{NC}_i \end{aligned}$$

Par exemple nous avons vu que ACCESSIBILITÉ $\in \mathbf{NC}_2$.
Évidemment on a que

$$\mathbf{NC}_1 \subseteq \mathbf{NC}_2 \subseteq \ldots \subseteq \mathbf{NC} \subseteq \mathbf{P}$$

mais on ne connaît aucun théorème de hiérarchie qui puisse montrer que ces inclusions sont propres.

Exercice 23

1. Montrer : tout langage régulier $L \in \{0,1\}^*$ est en $\mathbf{NC}_1$.
2. Est-ce que tous les langages en $\mathbf{NC}_1$ sont réguliers ?

Théorème 31 *Les classes $\mathbf{NC}_i$ pour $i \geq 2$ sont closes par $\leq_m^{logspace}$.*

Démonstration: Soit $L' \in \mathbf{NC}_i$, $i \geq 2$, et $L \leq_{\mathrm{m}}^{\mathrm{logspace}} L'$. On peut construire une machine déterministe qui, sur entrée (x, i), dit en espace $\log|x|$ si le i-ème bit du résultat de la réduction est 1. Cette machine a un graphe de configurations de taille n^k pour une constante k. En utilisant le fait que ACCESSIBILITÉ est en $\mathbf{NC}_2$ on peut donc résoudre le problème « est-ce que le i-ème bit du résultat de la réduction de x est 1 ? » en $\mathbf{NC}_2$. Le circuit pour L en $\mathbf{NC}_i$ consiste alors en le circuit pour L', où pour tous les i la i-ème entrée est remplacée par le circuit qui détermine le i-ème bit de la réduction. □

8.7 NC et espace logarithmique séquentiel

Exercice 24
Montrer que $\mathbf{NL} \subseteq \mathbf{NC}_2$

Théorème 32

$$\mathbf{NC}_1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{NC}_2$$

Démonstration: $\mathbf{L} \subseteq \mathbf{NL}$ est évident.

$\mathbf{NL} \subseteq \mathbf{NC}_2$: exercice 24.

Pour $\mathbf{NC}_1 \subseteq L$: Soit $L \in \mathbf{NC}_1$. On construit un algorithme pour L en $\mathbf{L}$ comme la composition de deux fonctions qui sont chacune calculable en espace logarithmique (on se rappelle que les fonctions calculables en espace logarithmique sont closes par compositions, voir lemme 5). Étant donné un mot x,

1. La première fonction calcule le circuit $C_{|x|}$ (cela se fait en espace logarithmique, selon la définition d'un circuit uniforme).
2. La deuxième fonction évalue le circuit avec un algorithme décendant. Cet algorithme ne garde en « mémoire » qu'un chemin de l'arbre. Cette évaluation se fait alors aussi en espace logarithmique.

   Remarquons qu'on évalue ici le circuit comme s'il était un arbre, c'est-à-dire on ne profite pas du partage de valeurs.

□

Le théorème 32 nous donne alors une relation forte entre temps parallèle et espace logarithmique.

Théorème 33 Accessibilité *est* $\mathbf{NL}$*-complet*

Démonstration: L'appartenance à $\mathbf{NL}$ est évidente.

Si $L \in \mathbf{NL}$ alors on peut, pour un x donné, construire en espace logarithmique son graphe de configurations (qui est de taille polynômiale en $|x|$). Le problème $x \in L$ est alors un problème d'accessibilité dans ce graphe. □

Corollaire 9 Accessibilité *restreinte à des graphes acycliques est* $\mathbf{NL}$*-complet.*

Démonstration: Le graphe de configurations dans la preuve du théorème 33 est acyclique. □

Une *2CNF* est une formule propositionnelle en forme conjonctive normale, où chaque clause consiste de deux littéraux au maximum. 2SAT est l'ensemble des 2CNF qui sont satisfaisables.

Théorème 34 2SAT *est* $\mathbf{NL}$*-complet.*

Démonstration: Pour montrer que 2SAT $\in \mathbf{NL}$:

Nous montrons d'abord une équivalence entre 2SAT et un problème sur les graphes : étant donnée une 2CNF, on construit un graphe où les nœuds sont les littéraux de C, et où il y a une arrête de α vers β, noté $\alpha \to \beta$ dans la suite, quand $(\neg\alpha \vee \beta)$ est une clause de C (modulo commutativité de $\vee$ et la loi de double négation). Donc, si $\alpha \to \beta$ alors on a aussi que $\neg\beta \to \neg\alpha$. Nous montrons que C est satisfaisable si et seulement s'il n'y a aucun littéral α tel que $\alpha \to^* \neg\alpha$ et $\neg\alpha \to^* \alpha$.

– Si un tel littéral α existe alors C implique que α et $\neg\alpha$ sont équivalents, ce qui implique que C est non-satisfaisable.
– Si un tel littéral α n'existe pas alors on peut construire une solution pour C. Tant que toutes les variables ne soient pas encore affectées soit α un littéral qui n'a pas encore de valeur, tel que $\alpha \not\to \neg\alpha$. On donne à α et tous les nœuds atteignables par α la valeur TRUE, et à $\neg\alpha$ et tous les nœuds à partir desquels on peut atteindre $\neg\alpha$ la valeur FALSE.

On obtient donc que 2SAT est en co-**NL** : pour montrer que la formule est non-satisfaisable on choisit un chemin $\alpha \to^* \neg\alpha$ et un chemin $\neg\alpha \to^* \alpha$. Or, par le théorème d'Immerman, co-**NL**=**NL**.

Pour montrer que 2SAT est **NL**-dur :

Nous montrons que co-ACCESSIBILITÉ (qui est **NL**-complet parce que $\mathbf{NL} = co - \mathbf{NL}$) $\leq_{\mathrm{m}}^{\mathrm{logspace}}$ 2SAT. Étant donné un graphe G avec deux nœuds *start* et *goal*. Nous construisons la 2CNF dont les variables sont les nœuds de G, et les clauses sont *start*, $\neg$*goal*, et $(\neg x \vee y)$ pour toutes les arrêtes (x, y) de G. On voit que cette 2CNF est satisfaisable si et seulement si *goal* n'est pas atteignable à partir de *start*. □

Exercice 25
Donner un problème **L**-complet (pour réduction $\leq_{\mathrm{m}}^{\mathrm{logspace}}$)

8.8 Exercice : les classes PolyL et SC

On définit la classe

$$\mathbf{PolyL} := \bigcup_{k\geq 1} DSPACE(log^k n)$$

De plus, on définit la classe **SC** comme la classe des langages qui sont reconnus par des machines déterministes qui sont *à la fois* n^k-bornées en temps et $log^l n$-bornées en espace pour des constantes k et l.

Exercice 26 (facile)
Montrer que $\mathbf{NL} \subseteq \mathbf{PolyL} \cap \mathbf{P}$

Exercice 27
Montrer que $\mathbf{PolyL} \neq \mathbf{P}$.

Indication : est-ce qu'il y a des problèmes qui sont **PolyL**-complets ?

Exercice 28
Pourquoi est-ce qu'on ne peut pas conclure facilement, comme dans l'exercice 26, que $\mathbf{NL} \subseteq \mathbf{SC}$? (la réponse tient sur 15 lignes maximum !)

Indication : Estimer le temps pris par une machine déterminisée selon Savitch.

Pour votre culture : $\mathbf{NL} \stackrel{?}{\subseteq} \mathbf{SC}$ est un problème ouvert.

On peut définir les *circuits de largeur polylog* comme suit : Un circuit est de largeur polylog si on peut partionner ses portes en ensembles $S_0, S_1, \ldots$ tels que

– S_0 est l'ensemble des portes d'entrée
– tous les entrées des portes en S_{i+1} sont en $S_0 \cup S_i$
– il y a un k tel que pour tous les $i \geq 1$: $|S_i| \leq \log^k n$
(attention, la définition donnée dans [Pap94], exercice 16.4.3, est différente)

Exercice 29
Montrer que tout langage régulier est reconne par un famille uniforme de circuits de largeur polylog.

Exercice 30
Montrer que la classe des langages reconnus par des circuits uniformes de largeur polylog est incluse dans **SC**.

Est-ce que l'inverse est vraie ?

On trouvera plus sur la complexité des circuits dans les ouvrages [Vol99] et [Str94].

Chapitre 9

La hiérarchie polynômiale et PSPACE

9.1 La hiérarchie polynômiale

Definition 19 *Les classes de la* hiérarchie polynômiale *sont*

$$\Delta_0\mathbf{P} = \Sigma_0\mathbf{P} = \Pi_0\mathbf{P} = \mathbf{P}$$

et pour tout $i \geq 0$:

$$\begin{aligned} \Delta_{i+1}\mathbf{P} &= \mathbf{P}^{\Sigma_i\mathbf{P}} \\ \Sigma_{i+1}\mathbf{P} &= \mathbf{NP}^{\Sigma_i\mathbf{P}} \\ \Pi_{i+1}\mathbf{P} &= co-\mathbf{NP}^{\Sigma_i\mathbf{P}} \end{aligned}$$

Voir la section 6.4 pour la définition des machines à oracle.

Lemme 17 *On a*
- $\Delta_1\mathbf{P} = \mathbf{P}$
- $\Sigma_1\mathbf{P} = \mathbf{NP}$
- $\Pi_1\mathbf{P} = co-\mathbf{NP}$

et pour tous les $i \geq 0$:
- $\Delta_i\mathbf{P} \subseteq \Sigma_i\mathbf{P}$
- $\Delta_i\mathbf{P} \subseteq \Pi_i\mathbf{P}$
- $\Sigma_i\mathbf{P} \subseteq \Delta_{i+1}\mathbf{P}$
- $\Pi_i\mathbf{P} \subseteq \Delta_{i+1}\mathbf{P}$

Voir la figure 9.1. Il n'est pas connu si ces inclusions sont strictes ou pas.

Definition 20 *Une relation $R \subseteq \Sigma^* \times \Sigma^*$ est*
- polynômiallement décidable *s'il y a une machine déterministe et polynômiallement bornée en temps qui accepte $\{x\#y \mid (x, y) \in R\}$*
- polynômiallement balancée *s'il existe un $k \geq 1$ tel que $(x, y) \in R$ implique que $|y| \leq |x|^k$*

Théorème 35 *Soit $i \geq 1$. $L \in \Sigma_i\mathbf{P}$ ssi il existe une relation R polynômiallement balancée telle que*

$$
\begin{tikzcd}
{} & \vdots & {} \\
& \Delta_2\mathbf{P} \arrow[ul] \arrow[ur] & \\
\Sigma_2\mathbf{P} = \mathbf{NP}^{\mathbf{NP}} \arrow[ur] & & \Pi_2\mathbf{P} = co-\mathbf{NP}^{\mathbf{NP}} \arrow[ul] \\
& \Delta_1\mathbf{P} \arrow[ul] \arrow[ur] & \\
\Sigma_1\mathbf{P} = \mathbf{NP} \arrow[ur] & & \Pi_1\mathbf{P} = co-\mathbf{NP} \arrow[ul] \\
& \Delta_0\mathbf{P} = \mathbf{P} \arrow[ul] \arrow[ur] &
\end{tikzcd}
$$

FIG. 9.1 – Le début de la hiérarchie polynômiale

1. $L = \{x \mid \exists y : (x, y) \in R\}$

2. $R \in \Pi_{i-1}\mathbf{P}$

Ce théorème dit alors pour le cas $i = 1$: $L \in \mathbf{NP}$ ssi il existe une relation R polynômiallement balancée et polynômiallement décidable telle que $L = \{x \mid \exists y : (x, y) \in R\}$. On peut voir ce mot y comme un « certificat » de l'appartenance de x à L.

Exercice 31
Montrer le théorème 35.

Corollaire 10 *Soit $i \geq 1$. $L \in \Pi_i\mathbf{P}$ ssi il existe une relation R polynômiallement balancée et $k \geq 1$ telle que*

1. $L = \{x \in \Sigma^* \mid \forall y : |y| \leq |x|^k \Rightarrow (x, y) \in R\}$

2. $R \in \Sigma_{i-1}\mathbf{P}$

Minimal-Circuit est le problème suivant : Étant donné un circuit, est-il minimal ? (un circuit est minimal s'il n'y a pas de circuit plus petit qui calcule la même fonction). On a donc que Minimal-Circuit $\in \Pi_2\mathbf{P}$, on choisissant

$$R = \{(C, C') \mid |C| \leq |C'| \text{ ou } C' \text{ calcule une fonction différente de } C\}$$

La relation « C et C' calculent des fonctions différentes » est en $\Sigma_1\mathbf{P} = \mathbf{NP}$ puisqu'il suffit de deviner des argument pour lesquels les deux circuits calculent des résultats différents.

Definition 21 *Une relation $R \subseteq (\Sigma^*)^{i+1}$ est* polynômiallement balancée *s'il existe un k tel que*

$$(x, y_1, \ldots, y_i) \in R \Rightarrow |y_1|, \ldots |y_i| \leq |x|^k$$

Corollaire 11 *Soit $i \geq 1$. Alors $L \in \Sigma_i\mathbf{P}$ ssi il existe une relation $R \in (\Sigma^*)^{i+1}$ polynômiallement décidable et polynômiallement balancée telle que*

$$L = \{x \mid \exists y_1 \forall y_2 \exists y_3 \ldots (x, y_1, \ldots, y_i) \in R\}$$

Démonstration: Par induction, en utilisant le théorème 35 et le corollaire 10. □

Le problème QSat$_i$ est : étant donnée une formule propositionnelle quantifiée (les quantificateurs portent sur les valeurs booléennes des variables propositionnelles) avec i blocs de quantificateurs, est-elle valide ? Par exemple, une instance de QSat$_3$ est

$$\exists P_1, P_2 \, \forall P_3, P_4, P_5 \, \exists P_6 : \; P_1 \, \wedge (P_2 \vee P_3 \rightarrow \ldots)$$

Théorème 36 *Pour tout $i \geq 1$,* QSat$_i$ *est $\Sigma_i\mathbf{P}$-complet.*

Pour ce théorème on peut même supposer que la matrice de la formule est en forme conjonctive normale.

Démonstration: D'abord QSat$_i \in \Sigma_i\mathbf{P}$ est évident par corollaire 11.

Soit $L \in \Sigma_i\mathbf{P}$. Il y a donc une relation R polynômiallement balancée et polynômiallement décidable telle que

$$x \in L \Leftrightarrow \exists y_1 \forall y_2 \ldots \exists y_i R(x, y_1, \ldots, y_i)$$

(nous supposons ici que i est impair.) Soit ϕ l'expression booléenne qui correspond à l'existence d'un calcul acceptant de la machine pour R (comme dans la preuve du théorème de Cook). Nous pouvons supposer que sur le ruban d'entrée de la machine les arguments $x, y_1, \ldots, y_i$ commencent sur des positions fixes (on peut le supposer puisque leur longueur est bornée). Étant donnée une valeur w pour x, soit $\hat{\phi}_w$ l'expression ϕ où nous avons substitué pour les variables de x les valeurs booléennes correspondantes à w. Donc, w est traduit vers

$$\exists Y_1 \forall Y_2 \ldots \exists Y_i \exists Z. \hat{\phi}_w$$

où les Y_i sont les variables booléennes qui correspondent au i-ème argument, et Z toutes les autres variables du tableau.

Dans le cas que i est pair on prend le complément $\bar{L} \in \Pi_i\mathbf{P}$. On le réduit de la même manière à QVAL_i (définie comme QSAT_i sauf que le premier quantificateur est $\forall$), donc L se réduit vers QSAT_i. □

Théorème 37 *S'il y a un $i \geq 1$ tel que $\Sigma_i\mathbf{P} = \Pi_i\mathbf{P}$ alors $\Sigma_j\mathbf{P} = \Pi_j\mathbf{P} = \Delta_j\mathbf{P}$ pour tout $j \geq i$.*

On dit dans ce cas que la hiérarchie polynômiale *s'effondre au niveau i*.

Démonstration: Il suffit de montrer que $\Sigma_i\mathbf{P} = \Sigma_{i+1}\mathbf{P}$. Soit $L \in \Sigma_{i+1}\mathbf{P}$. Alors il existe une relation $R \in \Pi_i\mathbf{P} = \Sigma_i\mathbf{P}$ telle que $L = \{x \mid \exists y.(x, y) \in R\}$.

Donc il y a une relation $S \in \Pi_{i-1}\mathbf{P}$ telle que $(x, y) \in R \Leftrightarrow \exists z.(x, y, z) \in S$, c'est-à-dire $x \in L \Leftrightarrow \exists y, z.(x, y, z) \in S$. Donc $L \in \Sigma_i\mathbf{P}$. □

Definition 22

$$\mathbf{PH} = \bigcup_{i \geq 0} \Delta_i\mathbf{P}$$

On montre facilement que toutes les classes de la hiérarchie polynômiale sont closes par $\leq_{\mathrm{m}}^{\mathrm{logspace}}$. Il en suit

Théorème 38 *S'il y a un problème qui est **PH**-complet alors la hiérarchie polynômiale s'effondre à un certain niveau.*

9.2 PSPACE

Théorème 39

$$\mathbf{PH} \subseteq \mathbf{PSPACE}$$

Démonstration: Pour tout i, $\text{QSAT}_i \in \mathbf{PSPACE}$ □

Il n'est pas connu si l'inclusion est stricte.

Le problème QBF (pour *Quantified Boolean Formula*) est : étant donnée une formule booléenne quantifiée

$$\exists x_1 \forall x_2 \exists x_3 \ldots \phi$$

est-elle valide ? Par rapport au problèmes QSAT_i il y a à remarquer que dans QBF nous ne permettons pas de blocs de quantificateurs mais demandons une alternance stricte. Ce n'est pas une vraie restriction puisque nous pouvons toujours ajouter des variables « bidon » pour forcer une alternance stricte des quantificateurs.

Théorème 40 QBF *est* **PSPACE**-*complet.*

On peut même suppose que la matrice de la formule est en forme disjonctive normale (ou en forme conjonctive normale).

Démonstration: Il est facile de voir que QBF est en **PSPACE** : il suffit de réaliser une récurrence sur toutes les variables de la formule (ce qui se fait avec une pile de taille linéaire dans la taille de la formule).

Pour montrer que QBF est **PSPACE**-dur : Soit un langage L reconnu par une machine M qui est n^k-bornée en espace. Il y a donc $c^{(n^k)}$ configurations possibles, une construction à la Cook nous donnerait alors une formule de taille exponentielle. L'idée pour construire une formule de taille polynômiale est la même que dans la preuve du théorème de Savitch (théorème 18) : On peut représenter une configuration de la machine M par $m = c' * n^k$ variables booléennes (pour une constante c'). On cherche à définir une formule

$$\phi_i(x_1, \ldots, x_m, y_1, \ldots, y_m)$$

qui exprime qu'on peut accéder la configuration $y_1, \ldots, y_m$ à partir de la configuration $x_1, \ldots, x_m$ en $\leq 2^i$ pas. Une première idée pour la définition de ϕ_i est : la formule ϕ_0 est une variante triviale de la table de transition de la machine M, et puis

$$\phi_{i+1}(\bar{x}, \bar{y}) = \exists \bar{z}(\phi_i(\bar{x}, \bar{z}) \wedge \phi_i(\bar{z}, \bar{y}))$$

mais avec cette construction on obtient encore une formule ϕ_i de taille exponentielle en i. La bonne idée est d'utiliser des quantificateurs *universelles* pour exprimer dans la définition de ϕ_{i+1} deux « appels » à la « procédure » ϕ_i :

$$\phi_{i+1}(\bar{x}, \bar{y}) = \exists \bar{z}(\forall \bar{v} \forall \bar{w}((\bar{x} = \bar{v} \wedge \bar{z} = \bar{w}) \vee (\bar{z} = \bar{v} \wedge \bar{y} = \bar{v}) \to \phi_i(\bar{v}, \bar{w})))$$

On obtient donc une formule dont la taille est linéaire en i, donc polynômiale en n. □

Il y a une forte connexion entre les problèmes en **PSPACE** et les jeux. Par exemple, QBF peut être vu comme une classe de jeux entre deux joueurs E et A. Pour une formule booléenne quantifiée

$$\exists x_1 \forall y_1 \exists x_2 \forall y_2 \exists x_n \forall y_n \ldots \ \phi$$

il y a n tours. Au i-ème tour, c'est d'abord le joueur E qui choisit une valeur pour la variable x_i, puis c'est le joueur A qui choisit une valeur pour la variable y_i. Si à la fin la formule ϕ est vraie pour la valuation construite alors c'est E qui gagne, sinon c'est A qui gagne. La formule est vraie si et seulement si le joueurs E a une stratégie gagnante dans ce jeu, c'est à dire à chaque étape une fonction qui lui dit en fonction des choix précédents de son adversaire comment choisir.

On peut voir de même façon les formules de la logique du premier ordre comme des jeux.

Exercice 32
À quoi correspond une stratégie dans un tel jeu pour la logique du premier ordre ?

Le jeu GÉOGRAPHIE est comme suit : une instance de ce jeu est donnée par un graphe fini et orientée et un nœud de départ. Il y a deux joueurs I et II. Chaque joueur à son tour choisit une arrête qui mène vers un nœud pas encore visité. Le premier joueur qui n'arrive plus à choisir perd.

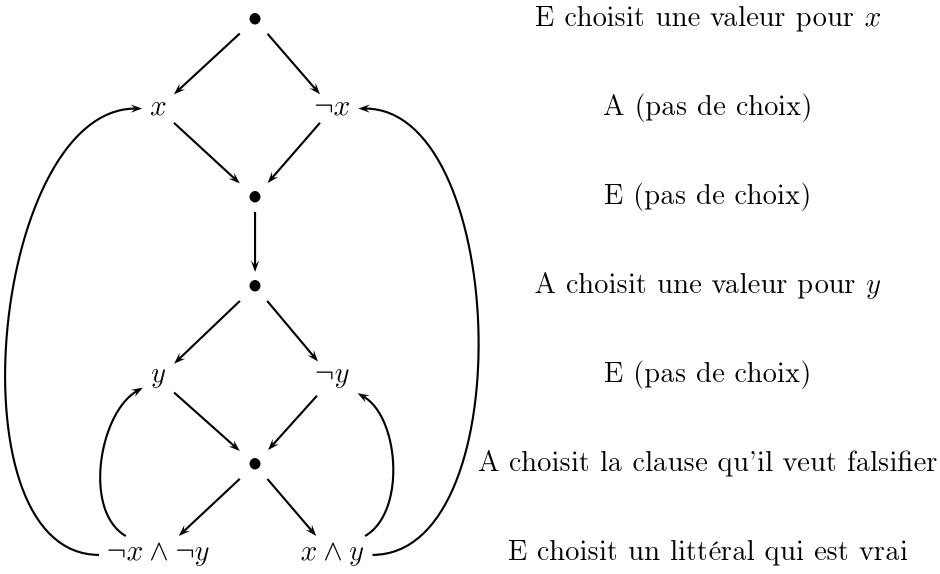

FIG. 9.2 – Le graphe pour la formule $\exists x \forall y ((x \vee y) \wedge (\neg x \vee \neg y))$

Théorème 41 *Le problème «étant donnée une instance de* GÉOGRAPHIE*, est-ce que le joueur I a une stratégie gagnante » est* **PSPACE***-complet.*

Démonstration: On montrer comme facilement comme dans la preuve du théorème 40 que le problème est en **PSPACE** (la longueur d'un match est limitée par le nombre de nœuds du graphe).

Pour voir que le problème est **PSPACE**-dur : On peut réduire QBF vers GÉOGRAPHIE comme suit : Étant donnée une formule en forme conjonctive normale, on construit un graphe consistant d'un losange par variable (voir la figure 9.2), puis un nœud par clause, et de chaque clause une arrête vers les compléments de ses littéraux. □

Le problème IN-PLACE-ACCEPTANCE est : étant donnée une machine *déterministe* M et un mot x, est-ce que M accepte x sans de jamais quitter les $|x|+1$ premières cases des rubans ? (La case numéro $|x|+1$ contient le marqueur de la fin de l'entrée)

Théorème 42 IN-PLACE-ACCEPTANCE *est* **PSPACE***-complet.*

Démonstration: Le problème est trivialement en **PSPACE**, il suffit de simuler une variante de la machine qui ne boucle pas (voir le lemme 1).

Pour montrer que le problème est **PSPACE**-dur : Soit $L \in$ **PSPACE** accepté par une machine M déterministe en temps n^k. Donc M accepte x si et seulement si une variante évidente M' de M accepte le mot $pad(x) := x\Box^{(n^k)-|x|}$ en place. La réduction de (M, x) vers $(M', pad(x))$ se fait en espace logarithmique. □

Les expressions rationnelles sur un alphabet $\Sigma = \{a_1, \ldots, a_n\}$ sont définies par la grammaire abstraite suivante :

$$r ::= a_1 \mid \ldots \mid a_n \mid \epsilon \mid r_1 \cup r_2 \mid r_1 r_2 \mid r^*$$

où ϵ dénote le mot vide, $\cup$ l'union de deux langages, juxtaposition leur concaténation, et $*$ l'itération. On admet qu'on peut en espace logarithmique traduire une telle expression rationnelle en un automate *non-déterministe* qui reconnaît le même langage. $L(r)$ dénote le langage reconnu par l'expression rationnelle r. Une expression r est Σ-*universelle* si $L(r) = \Sigma^*$.

Exercice 33
Montrer que le problème « Étant donnée une signature Σ et expression rationnelle r, est-ce que r est Σ-universelle ? » est dans **PSPACE**.

Indication : Il convient de montrer que le problème est dans co-**NPSPACE**. Justifier pourquoi votre algorithme n'utilise qu'un espace polynomial.

Exercice 34
Montrer que le problème « Étant donnée une signature Σ et une expression rationnelle r, est-ce que r est Σ-universelle ? » est **PSPACE**-complet.

Indication : On réduit le problème IN-PLACE-ACCEPTANCE vers le problème en question : Pour chaque machine de Turing déterministe M et mot x, on peut construire une expression rationnelle qui dénote le langage des mots qui ne sont *pas* des traces d'un calcul de T sur x en espace $|x| + 1$. Il suffit de donner cette expression rationnelle, et de la justifier. Vous pouvez utiliser pour la construction de l'expression rationnelle les abréviations qui vous conviennent si vous justifiez pourquoi la réduction se fait en espace logarithmique.

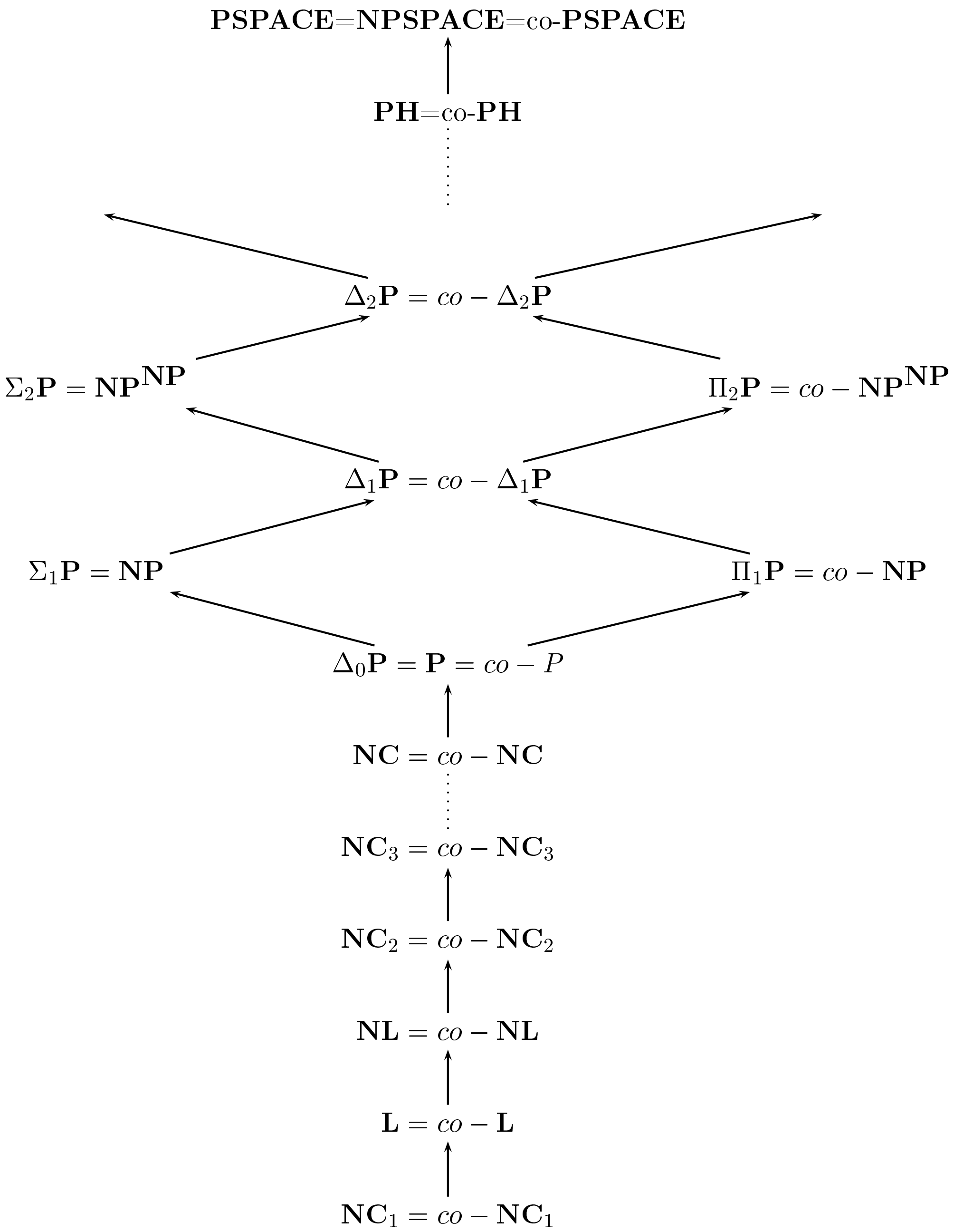

FIG. 9.3 – Résumé des classes importantes vues dans ce cours

Bibliographie

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Pap94] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Aar04] Scott Aaronson. The complexity zoo, 2004. Available at `http://www.cs.berkeley.edu/~aaronson/zoo.html`.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5) :935–938, 1988.

[Pav03] A. Pavan. Comparison of reductions and completeness notions. *ACM SIGACT News*, 34(2) :27–41, June 2003.

[Str94] Howard Straubing. *Finite Automata, Formal Logic and Circuit Complexity.* Birkäuser, 1994.

[Vol99] Heribert Vollmer. *Introduction to Circuit Complexity.* Texts in Theoretical Computer Science. Springer-Verlag, 1999.