Complexité avancée

Philippe Schnoebelen – Jean Goubault-Larrecq

2020-2021

Table des matières

Première partie

Philippe Schnoebelen

1 Classes en espace

Définition 1.1 Classe SPACE

Soit $f : \mathbb{N} \to \mathbb{N}$.
On définit $\mathbf{SPACE}(f(n))$ par l'ensemble des langages L tels qu'il existe une machine de Turing déterministe $\mathcal{M}$ qui termine sur toute entrée x avec un espace borné par $f(|x|)$, et qui accepte ssi $x \in L$.
De même on définit $\mathbf{SPACE}(f(n))$ avec des machines de Turing non déterministes.

Remarque 1.1

On ne compte ni la taille de l'entrée ni la taille de la sortie dans l'espace mémoire.

Définition 1.2 Classe NL

On définit la classe **NL** par

$$\mathbf{NL} = \bigcup_{k \in \mathbb{N}} \mathbf{NSPACE}(k \log(n)).$$

Théorème 1.1

Pour toute fonction f :
— $\mathbf{SPACE}(\mathcal{O}(f(n))) = \mathbf{SPACE}(f(n))$
— $\mathbf{NSPACE}(\mathcal{O}(f(n))) = \mathbf{NSPACE}(f(n))$

Preuve du théorème 1.1.

Pour montrer que $\mathbf{SPACE}(\alpha f(n)) \subseteq \mathbf{SPACE}(f(n))$x, on crée une machine avec plein d'éléments en plus dans l'alphabet pour prendre moins de place sur la bande.

Voilà !

1.1 Problème d'accessibilité dans les graphes

Définition 1.3 GAP(Graph Accessibility Problem)

GAP est le problème suivant :
— **Entrées :** graphe orienté G, sommets s, t
— **Question :** existe-t-il un chemin de s à t dans G ?

Lemme 1.2 Complexité spatiale non déteministe de GAP

Le problème GAP est **NL**.

Proposition 1.3 Complexité spatiale déterministe de GAP

Le problème GAP est $\mathbf{SPACE}(\log^2(n))$.

Preuve du proposition 1.3.

Il suffit de montrer que l'algorithme suivant termine en espace $\log^2(n)$.

Algorithm 1: path(s,t,L)

```
si L = 1 alors
    retourner s = t ∨ s → t ∈ E
sinon
    pour chaque x ∈ V faire
        si path(s,x,⌊L/2⌋) ∧ path(x,t,⌈L/2⌉) alors retourner Vrai;
    retourner Faux;
```

Youpi !

Lemme 1.4

NL $\subseteq$ **P**

Preuve du lemme 1.4.

Le nombre de configurations possibles d'une machine ayant un espace logarithmique est polynomial. Donc si la machine termine, elle ne boucle pas, donc on a au plus un nombre polynomial d'étapes.

cskifo

Lemme 1.5 Majoration du nombre de configurations

Une machine de Turing (déterministe ou non) tournant en espace **SPACE**$(f(n))$ a un nombre de configurations sur toute entrée x de taille n en $\mathcal{O}(n \cdot f(n)^{1+\log(|\Gamma|)})$.

Définition 1.4 Fonction propre

Une fonction $f : \mathbb{N} \to \mathbb{N}$ est propre ssi f est croissante et il existe une machine de Turing finissant sur toute entrée de taille n en écrivant $f(n)$ blancs sur sa bande de sortie en temps $\mathcal{O}(n + f(n))$ et en espace $\mathcal{O}(f(n))$.

Théorème 1.6 Théorème de Savitch

Soit f une fonction propre telle que $f(n) \geqslant \log(n)$, alors **NSPACE**$(f(n))$ $\subseteq$ **SPACE**$(f^2(n))$.

Preuve du théorème 1.6.

Puisque f est propre on peut réserver une bande de la machine pour écrire $f(n)$ en unaire en espace **SPACE**$(f(n))$.
Alors on peut appliquer la fonction path définie plus haut pour vérifier que l'on peut aller de la configuration initiale à la configuration finale dans le graphe des configurations possibles de la machine.

Youpi !

Corollaire 1.7

PSPACE $=$ **NPSPACE**

1.2 Theorème d'Immerman-Szelepcsényi

Théorème 1.8 Theorème d'Immerman-Szelepcsényi

GAP $\in$ **coNL**

Preuve du théorème 1.8.

On s'intéresse au problème coGAP :
— **Entrées :** graphe G, sommets s et t
— **Question :** Est-il vrai qu'il n'existe pas de chemin de s à t ?

On pose $J_i = \{u \in V \mid s \xrightarrow{<i} u\}$ et on va utiliser l'algorithme suivant.

Algorithm 2: copath(s)

$J \leftarrow [s]$;
pour i *allant de* 1 *à* $|V|-1$ **faire**
  $K \leftarrow []$;
  pour chaque $z \in V$ **faire**
    trouvé $\leftarrow$ **Faux**;
    pour chaque $y \in V$ **faire**
      si $y \in J \wedge (y = z \vee y \to z$ **alors** trouvé $\leftarrow$ **Vrai**;
    si *trouvé* **alors** $K \leftarrow K \cup \{z\}$;
  $J \leftarrow K$;

Malheureusement il n'est pas logspace, donc on va le modifier un peu.

Algorithm 3: copath(s, t, G)

$j \leftarrow 1$;
ok $\leftarrow$ **Faux**;
pour i *allant de* 1 *à* $|V|-1$ **faire**
  $k \leftarrow 0$;
  pour chaque $z \in V$ **faire**
    trouvé $\leftarrow$ **Faux**;
    $q \leftarrow 0$;
    pour chaque $y \in V$ **faire**
      deviner $b \in \{$**Vrai**,**Faux**$\}$; // $b = y \in J$
      si b **alors**
        deviner un chemin $s \xrightarrow{<i} y$ dans G, sinon rejeter;
        $q \leftarrow q + 1$;
        si $y = z \vee y \to z$ **alors**
          trouvé $\leftarrow$ **Vrai** **si** $z = t$ **alors** ok $\leftarrow$ **Vrai**;
    si $q < j$ **alors** rejeter;
    si *trouvé* **alors** $K \leftarrow K \cup \{z\}$;
    si *ok* **alors** accepter;
  $J \leftarrow K$;

C'est ce que je voulais !

Corollaire 1.9

coNL = NL

2 Réductions et classes de complexité

Définition 2.1 **Réduction**

On dit que $L_1 \subseteq A^*$ se réduit en $L_2 \subseteq B^*$ s'il existe une fonction $r : A^* \to B^*$ telle que
— $\forall x \in A^*, x \in L_1 \iff r(x) \in L_2$
— r est calculable en espace logarithmique.
On note alors $L_1 \leqslant L_2$.

Remarque 2.1

La relation de réductibilité est un pré-ordre. On a la transitivité par composition, et n'a pas l'antisymétrie mais on a des classes d'équivalences de problèmes d'égale complexité.

Définition 2.2 **Problème complet**

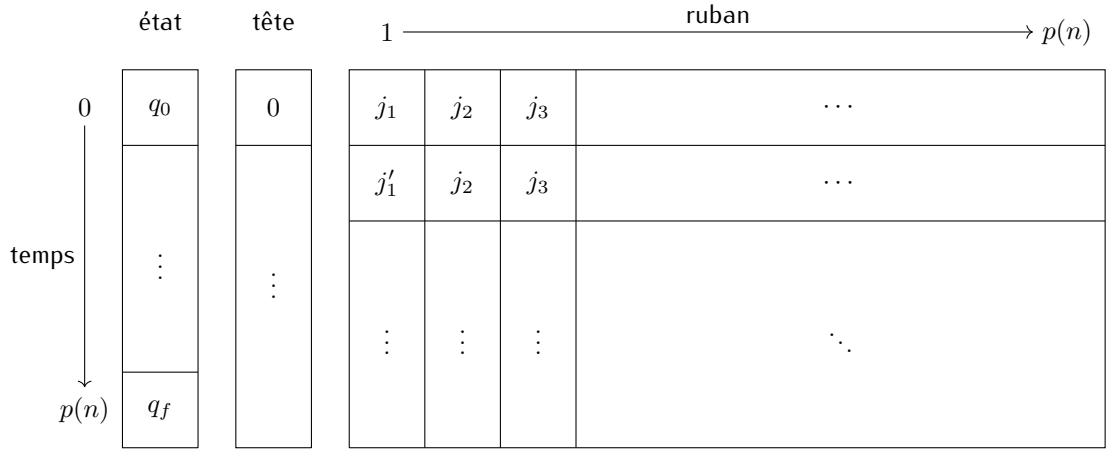
Un problème L est complet pour une classe de problèmes $\mathcal{C}$ si :
— $L \in \mathcal{C}$
— $\forall L' \in \mathcal{C}, L' \leqslant L$.

Exemple 2.1 **Exemples de problèmes complets**

— GAP est **NL**-complet.
— SAT est **NP**-complet.
— QBF (formules booléennes quantifiées) est **PSPACE**-complet.

Preuve du ??.

Montrons que GAP est **NL**-complet.
1. GAP $\in \mathbf{NL}$.
2. Soit $L \in \mathbf{NL}$, $L \subseteq A^*$ et M une machine de Turing non déterministe reconnaissant L en espace logarithmique. Soit $r : x \mapsto G_x = (V_x, E_x), s_x, t_x$ avec :
— $V_x = \{(q \in Q_M, w \in A^{\leqslant \log(n)}, p_0 \in [\![-1, n+1]\!], p_1 \in [\![-1, \log(n)+1]\!]\}$
— $E_x = \{(c, c') \mid M$, en calculant sur x, peut aller en 1 étpape de c à $c'\}$
— $s_x = (q_0, \varepsilon, 1, 1)$
— $t_x = (q_f, \varepsilon, 1, 1)$.
Montrons que SAT est **NP**-complet.
On rappelle que SAT est l'ensemble des formules booléennes en forme normale conjonctive satifiables.
1. En devinant une valuation des variables et en vérifiant en temps polynomial si la valuation satisfait la formule, on a que SAT $\in \mathbf{NP}$.
2. Soit $L \in \mathbf{NP}$, $L \subseteq A^*$ et M une machine de Turing non déterministe reconnaissant L en temps polynomial $p(n)$.
On va décrire le calcul de M par une formule. Ci-après est le diagramme du calcul de x par M.



La formule φ_x décrit un calcul de x dans le diagramme.
Les variables, pour $1 \leqslant i \leqslant p(n)$, $j \in \Sigma$, $0 \leqslant k \leqslant p(n)$ et $q \in Q$:
— $C_{i,j,k}$ qui vaut vrai si la cellule i contient le symbole j à l'étape k.
— $T_{i,k}$ qui vaut vrai si la tête de lecture est en i à l'étape k.
— $Q_{q,k}$ qui vaut vrai si on est dans l'état q à l'étape k.
Et φ_x est la conjonction de :
— $C_{i,j_i,0}$ pour tout i, les valeurs initiales du ruban.
— $Q_{q_0,0}$, l'état initial.
— $T_{0,0}$, la position initiale de la tête de lecture.
— $\neg C_{i,j,k} \vee \neg C_{i,j',k}$, pour tous $i, j \neq j', k$, pour préciser qu'on n'a qu'un seul symbole par cellule à chaque instant.
— $\bigvee_{j \in \Sigma} C_{i,j,k}$ pour tous i, k, pour préciser qu'on a au moins un symbole par cellule à chaque instant.
— $C_{i,j,k} \wedge C_{i,j',k+1} \Rightarrow T_{i,j}$ pour tous $i, j \neq j', k$, pour préciser qu'on doit avoir la tête de lecture à la bonne place pour écrire.
— $\neg Q_{q,k} \vee \neg Q_{q',k}$, pour tous $q \neq q', k$, pour préciser qu'on est dans un seul état à chaque instant.
— $\neg T_{i,k} \vee \neg T_{i',k}$, pour tous $i \neq i', k$, pour préciser qu'on n'est qu'à un seul endroit à chaque instant.
— $T_{i,k} \wedge Q_{q,k} \wedge C_{i,j,k} \Rightarrow \bigvee_{((q,j),(q',j',d)) \in \delta} (T_{i+d,k+1} \wedge Q_{q',k+1} \wedge C_{i,j',k+1})$, pour tous i, j, k, pour donner toutes les transitions possibles.
— $\bigvee_{0 \leqslant k \leqslant p(n)} \bigvee_{f \in F} Q_{f,k}$ pour terminer sur un état final.

Eh ben on y va, on mouline !

$$\varphi_x = \bigwedge_{1\leqslant i\leqslant p(n)} C_{i,j_i,0} \wedge Q_{q_0,0} \wedge T_{0,0} \wedge \bigwedge_{1\leqslant i\leqslant p(n)} \bigwedge_{j\neq j'\in\Sigma} \bigwedge_{0\leqslant k\leqslant p(n)} \neg C_{i,j,k} \vee \neg C_{i,j',k}$$
$$\wedge \bigwedge_{1\leqslant i\leqslant p(n)} \bigwedge_{j\in\Sigma} \bigwedge_{0\leqslant k\leqslant p(n)} \bigvee_{j\in\Sigma} C_{i,j,k} \wedge \bigwedge_{1\leqslant i\leqslant p(n)} \bigwedge_{j\neq j'\in\Sigma} \bigwedge_{0\leqslant k\leqslant p(n)} C_{i,j,k} \wedge C_{i,j',k+1} \Rightarrow T_{i,j}$$
$$\wedge \bigwedge_{q\neq q'\in Q} \bigwedge_{0\leqslant k\leqslant p(n)} \neg Q_{q,k} \vee \neg Q_{q',k} \wedge \bigwedge_{0\leqslant i<i'\leqslant p(n)} \bigwedge_{0\leqslant k\leqslant p(n)} \neg T_{i,k} \vee \neg T_{i',k}$$
$$\wedge \bigwedge_{1\leqslant i\leqslant p(n)} \bigwedge_{j\in\Sigma} \bigwedge_{0\leqslant k\leqslant p(n)} T_{i,k} \wedge Q_{q,k} \wedge C_{i,j,k} \Rightarrow \bigvee_{((q,j),(q',j',d))\in\delta} (T_{i+d,k+1} \wedge Q_{q',k+1} \wedge C_{i,j',k+1})$$
$$\wedge \bigvee_{0\leqslant k\leqslant p(n)} \bigvee_{f\in F} Q_{f,k}.$$

Voilà !

3 Alternation

Pour implémenter un algorithme non déterministe de NP, il faut exécuter l'algorithme en parallèle pour chaque choix possible, et s'arrêter dès qu'un calcul accepte, ou attendre que tous refusent. Cependant cela requiert un nombre exponentiel d'ordinateurs ou de cœurs.

Si, au contraire, on attend que tous les calculs acceptent pour accepter, alors on a une implémentation d'algorithmes de coNP pour résoudre par exemple VALIDITY (SAT avec des $\forall$ au lieu de $\exists$).

On va définir une généralisation permettant d'avoir un peu des deux en même temps.

3.1 Définition et inclusions

Définition 3.1 — Machine de Turing alternante (ATM)

Une machine de Turing alternante $(Q, \Delta, ..., F \subseteq Q, R \subseteq Q, Q = Q_\exists \uplus Q_\forall)$. Avec F (resp. R) les états acceptants (resp. rejetants) et $Q_\exists$ (resp. $Q_\forall$) les états se demandant qu'il existe un choix acceptant (resp. que tous les choix soient acceptants).
On note $c = (q, p_0, w_1, p_1, ...)$ une configuration de la machine.

Définition 3.2 — Configuration acceptante

c est acceptante ssi une des conditions suivantes est vérifiée :
— $q \in F$
— $q \in Q_\exists$ et $\exists c \to c', c'$ acceptante
— $q \in Q_\forall$ et $\forall c \to c', c'$ acceptante

Définition 3.3 — Classe ATIME, classe ASPACE

On définit les classes $\mathbf{ATIME}(f)$ et $\mathbf{ASPACE}(f)$ avec des machines alternantes.

Proposition 3.1 — Inclusions générales

$\mathbf{TIME}(f) \subseteq \mathbf{NTIME}(f) \subseteq \mathbf{ATIME}$
$\mathbf{TIME}(f) \subseteq \mathbf{coNTIME}(f) \subseteq \mathbf{ATIME}$
$\mathbf{ATIME}(f) \subseteq \mathbf{ASPACE}(f) \subseteq \mathbf{TIME}(2^{\mathcal{O}(f)})$

3.2 Temps alternant $\simeq$ espace

Définition 3.4 — Classe AP

On définit la classe $\mathbf{AP}$ (ou $\mathbf{APTIME}$) par $\mathbf{ATIME}(n^{\mathcal{O}(1)})$.

Lemme 3.2

QBF $\in$ AP

Preuve du lemme 3.2.

On s'arrange pour avoir un état de $Q_\exists$ pour chaque choix de variable avec un $\exists$ et un état de $G_\forall$ pour chaque choix de variable avec un $\forall$. De cette manière on calcule en temps linéaire.

C'est ce que je voulais !

Corollaire 3.3

PSPACE $\subseteq$ AP

Preuve du corollaire 3.3.

QBF est **PSPACE**-complet.

Voilà !

On a même égalité entre les deux.

Théorème 3.4

PSPACE = AP

Preuve du théorème 3.4.

Il suffit de montrer que **AP** $\subseteq$ **PSPACE**.
Soit $L \in$ **AP** et $\mathcal{M}$ une ATM reconnaissant L en temps polynomial $p_{\mathcal{M}}(n)$.
Avec un programme récursif vérifiant si un état est acceptant en utilisant la définition récursive de l'acceptance, il suffit alors d'un espace mémoire $\mathcal{O}(p_{\mathcal{M}}(n)^2)$, ce qui est toujours un polynôme.

Et là c'est le plus beau jour de ma vie.

Et ce résultat est lui-même encore plus général.

Théorème 3.5 **ATIME est presque toujours SPACE**

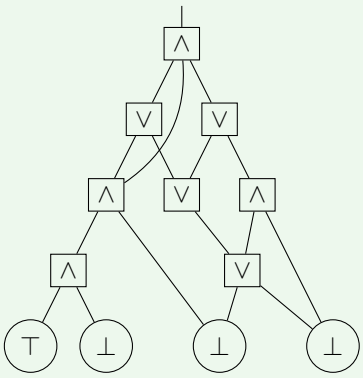
Soit f une fonction telle que $f(n) \geqslant n$.
Alors **ATIME**(poly(f)) = **SPACE**(poly(f)).

3.3 Espace alternatif $\simeq$ temps exponentiel

Le problème de l'évaluation d'une formule booléenne sans variable, avec des $\bot$ et $\top$, est dans **L**. Ce problème est l'évaluation d'un arbre. Avec l'alternation on va pouvoir évaluer un graphe.

Exemple 3.1 **Exemple de circuit booléen**

Un circuit est un graphe sans cycle, c'est à dire un arbre ayant des nœuds se recoupant.



Définition 3.5 **Classe AL**

On définit la classe **AL** par **ASPACE**($\log n$).

Définition 3.6 **Problème MonotoneCircuitValue (MCV)**

Le problème MCV est celui du calcul d'un circuit booléen avec seulement des $\vee$ et $\wedge$ (pas de $\neg$).

Proposition 3.6

MCV $\in$ **AL**

Preuve du proposition 3.6.

On utilise la fonction récursive suivante pour l'évaluation d'un circuit booléen :

$$\text{eval}(x) = \begin{cases} \text{Faux} & \text{si } x = \bot \\ \text{Vrai} & \text{si } x = \top \\ \text{eval}(x_1) \wedge \text{eval}(x_2) & \text{si } x = x_1 \wedge x_2 \\ \text{eval}(x_1) \vee \text{eval}(x_2) & \text{si } x = x_1 \vee x_2. \end{cases}$$

Voilà !

Définition 3.7 **Problème CircuitValue (CV)**

Le problème circuit value est celui du calcul d'un circuit booléen avec des $\vee$, $\wedge$ et $\neg$.

Proposition 3.7

CV $\in$ **AL**

Preuve du proposition 3.7.

Il suffit de montrer CV $\leqslant$ MCV.

Voilà !

Théorème 3.8

MCV est **PTIME**-complet.

Preuve du théorème 3.8.

L'idée est la même que pour la preuve que SAT est **NP**-complet, mais puisque l'on a une machine déterministe on va réussir à créer un circuit booléen sans quantificateurs.

Voilà !

Corollaire 3.9

$\mathbf{P} \subseteq \mathbf{AL}$

On a ici aussi égalité.

Théorème 3.10

$\mathbf{P} = \mathbf{AL}$

Preuve du théorème 3.10.

On va montrer que MCV est **AL**-complet.
On crée un circuit dont les nœuds sont les configurations, avec un $\vee$ pour les états existentiels et avec un $\wedge$ pour les états universels.

C'est ce que je voulais !

Théorème 3.11 **ASPACE(f) est presque toujours EXPTIME(f)**

Soit f une fonction telle que $f(n) \geqslant \log n$.
Alors $\mathbf{TIME}(2^{\mathcal{O}(f(n))} = \mathbf{ASPACE}(f)$.

4 Machines oracles et hierarchie polynomiale

4.1 Machine de Turing oracle

Une machine oracle est une machine de Turing qui a accès à un oracle.

Définition 4.1 — Machine de Turing oracle

Soit $O \subseteq A^*$.
Une machine de Turing oracle $\mathcal{M}^O$ est une machine de Turing avec :
— un ruban « requête » qui est en écriture seule
— trois états spéciaux : $q_{\text{requête}}$, q_{oui} et q_{non} tels que $q_{\text{requête}}$ renvoie vers q_{oui} ou q_{non} selon si $w \in O$ ou non, et efface le ruban de requête

Exemple 4.1 — Oracle SAT

On prend $O = \text{SAT}$. Alors $\mathcal{M}^O$ peut résoudre en temps polynomial :
— SAT, et même UNSAT qui est coNP-complet
— SAT-UNSAT : φ_1, φ_2 accepté ssi φ_1 est satifiable et φ_2 non satisfiable
— UNIQSAT : φ accepté ssi φ est satisfiable avec une unique valuation
— MINSAT : φ accepté ssi φ est satisfiable et la valuation la plus faible à l'ordre lexicographique rend la dernière variable vraie

Définition 4.2 — Classe C^O

Soit $O \subseteq A^*$ et C une classe de complexité.
On définit la classe C^O par l'ensemble des langages décidables en complexité C par une machine du Turing oracle sur le langage O.

Proposition 4.1

— $\mathbf{NP} \subseteq \mathbf{P}^{\text{SAT}}$
— $\mathbf{coNP} \subseteq \mathbf{P}^{\text{SAT}}$
— Si $O \in \mathbf{P}$ alors $\mathbf{P}^O = \mathbf{P}$

4.2 Hierarchie polynomiale temporelle

Définition 4.3

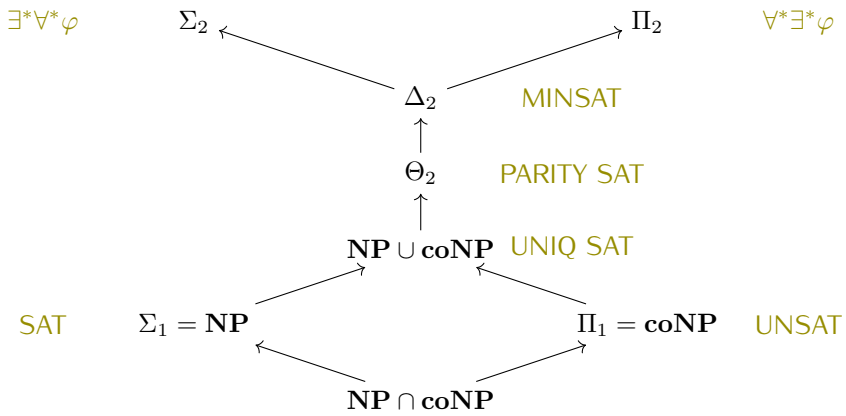
$$\Sigma_0 = \Pi_0 = \Theta_0 = \Delta_0 = \mathbf{P}$$

$$\Sigma_{i+1} = \mathbf{NP}^{\Sigma_i} \qquad \Pi_{i+1} = \mathbf{co}\Sigma_{i+1} \qquad \Delta_{i+1} = \mathbf{P}^{\Sigma_i} \qquad \Theta_{i+1} = \mathbf{P}_{\|}^{\Sigma_i} = \mathbf{P}^{\Sigma_i[O \log n]}$$

où $\mathbf{P}^{\Sigma_i[O \log n]}$ dit qu'on ne peut faire qu'un nombre de requêtes logarithmique.

Théorème 4.2

$$\mathbf{PH} = \bigcup_{i \in \mathbb{N}} \Sigma_i = \bigcup_{i \in \mathbb{N}} \Pi_i = \bigcup_{i \in \mathbb{N}} \Delta_i = \bigcup_{i \in \mathbb{N}} \Theta_i$$



Théorème 4.3

— Σ_iBF est Σ_i-complet (on fait i alternations en commençant par $\exists$)
— Π_iBF est Π_i-complet (on fait i alternatons en commençant par $\forall$)

Théorème 4.4

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

Preuve du théorème 4.4.

On montre par récurrence que pour tout i, $\Sigma_i \subseteq \mathbf{PSPACE}^{\mathbf{PSPACE}} = \mathbf{PSPACE}$.

Et là c'est le plus beau jour de ma vie.

Théorème 4.5

Si $\mathbf{PH} = \mathbf{PSPACE}$ alors $\exists k, \mathbf{PH} = \Sigma_k$.

Corollaire 4.6

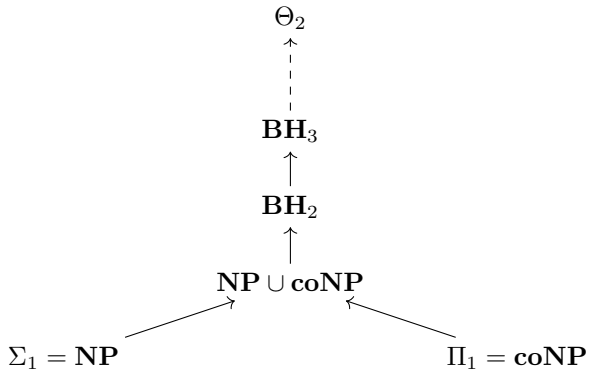
PH n'a pas de problème complet, sauf si $\mathbf{PH} = \mathbf{PSPACE}$.

Définition 4.4 Hierarchie booléenne

On définit récursivement :
— $\mathbf{BH}_1 = \mathbf{NP}$
— $\mathbf{BH}_{2k} = \mathbf{coNP} \wedge \mathbf{BH}_{2k-1}$ (une instance de **coNP** et une instance de $\mathbf{BH_{2k-1}}$)
— $\mathbf{BH}_{2k+1} = \mathbf{coNP} \vee \mathbf{BH}_{2k}$
Et on pose $\mathbf{BH} = \bigcup \mathbf{BH}_k$.

Théorème 4.7

$\mathbf{BH} \subseteq \Theta_2$



Deuxième partie

Jean Goubault-Larrecq

5 Randomized Turing machines

Définition 5.1 Machine de Turing randomisée

Une machine de Turing randomisée est une machine de Turing avec deux rubans en lecture seule : le ruban d'entrée et le ruban aléatoire.
Le ruban aléatoire est rempli uniformément aléatoirement et on ne peut pas aller à gauche dessus.

On va s'intéresser à la probabilité que la machine accepte sur x, r.
Il faut alors avoir une entrée aléatoire r de taille au moins du nombre d'itérations.

5.1 Classe RP

Définition 5.2 RP

Un langage L est dans $\mathbf{RP}$ ssi il existe une machine de Turing $\mathcal{M}$ (normale) terminant sur toute entrée x en au plus $p(|x|)$ polynôme et :
— si $x \in L$ alors $\mathbb{P}_r(\mathcal{M}(x,r) \text{ accepte}) \geqslant \frac{1}{2}$
— si $x \notin L$ alors $\mathcal{M}(x,r)$ n'accepte pour aucun r.

Il n'y a pas de machines $\mathbf{RP}$, c'est bien une machine de Turing en temps polynomial.

Définition 5.3 Classe coRP

L est dans $\mathbf{coRP}$ ssi il existe une machine de Turing $\mathcal{M}$ (normale) terminant sur toute entrée x en au plus $p(|x|)$ polynôme et :
— si $x \in L$ alors $\mathcal{M}(x,r)$ accepte pour tout r.
— si $x \notin L$ alors $\mathbb{P}_r(\mathcal{M}(x,r) \text{ accepte}) \leqslant \frac{1}{2}$

Exemple 5.1 Test de primalité

On considère le problème PRIMALITY de savoir un un entier p (en binaire) est premier.
Ce problème est clairement dans $\mathbf{coNP}$, il a été montré en 1975 qu'il est dans $\mathbf{NP}$ et en 2004 par Agrawal, Kayal et Saxena qu'il est dans $\mathbf{P}$.
Avec de la randomisation on peut le vérifier bien plus efficacement, en utilisant le test de primalité de Fermat par exemple (en négligeant les nombres de Carmichael).
Sinon il faut utiliser le test de Rabin-Miller qui marche vraiment, pour montrer que PRIMALITY $\in \mathbf{coRP}$.

Définition 5.4 Classe $\mathbf{RP}(\varepsilon)$

On définit la classe $\mathbf{RP}(\varepsilon)$ avec la même définition que $\mathbf{RP}$ mais avec la probabilité supérieure à $1 - \varepsilon$.

Théorème 5.1

$\forall \varepsilon \in]0,1[, \mathbf{RP}(\varepsilon) = \mathbf{RP}$

Preuve du théorème 5.1.

On montre que si $\eta < \varepsilon$ alors $\mathbf{RP}(\eta) = \mathbf{RP}(\varepsilon)$. Pour cela on prend une machine qui va faire plein d'exécutions avec des autres entrées aléatoires pour faire diminuer la probabilité d'erreur, c'est-à-dire qu'on montre que $\mathbf{RP}(\varepsilon) \subseteq \mathbf{RP}(\varepsilon^k)$ pour tout k, en ayant une machine k fois plus lente.

cskifo

On peut même rendre l'erreur exponentiellement petite en étant log plus lent.

Théorème 5.2

$\mathbf{P} \subseteq \mathbf{RP} \subseteq \mathbf{NP}$

Preuve du théorème 5.2.

$\mathbf{P} = \mathbf{RP}(0) \subseteq \mathbf{RP}$
Pour montrer l'inclusion dans $\mathbf{NP}$ on devine une entrée aléatoire qui marche.

cskifo

5.2 Classe ZPP

ZPP veut dire Zero Probability of error Polynomial-time.
C'est la classe des langages décidables en temps polynomial en moyenne (et pas dans le pire des cas).

Définition 5.5 Classe ZPP

$\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$

Théorème 5.3 Définition alternative

ZPP est la classe des langages pouvant être décidés en moyenne en temps polynômial avec une probabilité nulle d'erreur.

6 P/poly

Définition 6.1 Uniform P/poly

Un langage L est dans uniform P/poly ssi pour tout n on peut construire un circuit C_n en espace $\mathcal{O}(\log n)$ et tel que pour toute entrée x de taille n, $x \in L \Leftrightarrow C_n[x] = 1$.

Proposition 6.1

$\mathbf{P}$ = uniform P/poly

Définition 6.2 P/poly

Un langage L est dans $\mathbf{P/poly}$ ssi pour tout n il y a un circuit C_n de taille n tel que pour toute entrée x de taille n, $x \in L \Leftrightarrow C_n[x] = 1$.

Alors $\mathbf{P/poly}$ contient des langages indécidables (par exemple HALT). Il contient même d'ailleurs un nombre indénombrable de langages.

7 Classe BPP

Définition 7.1 Classe BPP

BPP est l'ensemble des langages L tels qu'il existe une machine de Turing $\mathcal{M}$ polynômiale telle que pour toute entrée x de taille n :
— $x \in L \Longrightarrow \mathbb{P}_r(\mathcal{M}(x,r) \text{ accepte}) \geqslant 2/3$
— $x \notin L \Longrightarrow \mathbb{P}_r(\mathcal{M}(x,r) \text{ accepte}) \leqslant 1/3$.

On peut également changer la constante de probabilité.

Définition 7.2 Classe $\mathbf{BPP}(\varepsilon)$

$\mathbf{BPP}(\varepsilon)$ est l'ensemble des langages L tels qu'il existe une machine de Turing $\mathcal{M}$ polynômiale telle que pour toute entrée x de taille n :
— $x \in L \Longrightarrow \mathbb{P}_r(\mathcal{M}(x,r) \text{ accepte}) \geqslant 1 - \varepsilon$
— $x \notin L \Longrightarrow \mathbb{P}_r(\mathcal{M}(x,r) \text{ accepte}) \leqslant \varepsilon$.

Théorème 7.1

Pour $\varepsilon \in]0, 1/2[$, $\mathbf{BPP}(\varepsilon) = \mathbf{BPP}$.

8 Merlin et Arthur

8.1 Classes AM et MA

8.2 Preuves interactives

Définition 8.1 **Classe IP**

Un langage L est dans $\mathbf{IP}[1]$ ssi pour tout polynôme g on a
— une machine de Turing polynômiale $\mathcal{A}$
— une fonction Merlin $M : \Sigma^* \to \Sigma^*$ produisant des sorties de taille polynômiale
— un langage D décidable en temps polynômial
tels que
— si $x \in L$, alors $\mathbb{P}_r(x\#q\#r\#y \in D) \geqslant 1/2^{g(n)}$, où $q = \mathcal{A}(x, r), y = M(x\#q\#r)$
— si $x \notin L$, alors pour toute fonction de merlin M', $\mathbb{P}_r(x\#q\#r\#y \in D) \leqslant 1/2^{g(n)}$, où $q = M'(x\#q\#r)$

Proposition 8.1

GNI $\in \mathbf{IP}[1]$

Proposition 8.2

GNI $\in \mathbf{AM}$

8.3 Classe BP·

Définition 8.2 **Classe BP $\cdot C$**

Soit C une classe de complexité

On va avoir besoin de faire de la réduction d'erreur. On va faire ça de manière brute en utilisant de la redondance.

Définition 8.3 **Classe démocratique**

Une classe C est démocratique si pour tout $L \in C$, $\{w_1\#...\#w_k \mid \text{une majorité des } w_i \text{ est dans } L\}$ est dans C.

Théorème 8.3

$\mathbf{AM} = \mathbf{BP} \cdot \mathbf{NP}$

Théorème 8.4 **De Babai**

$\mathbf{MA} \subseteq \mathbf{AM}$

Lemme 8.5

$\mathbf{MAM} \subseteq \mathbf{AM}$

8.4 Problèmes à promesse

On va avoir des problèmes où on promet quelque chose, par exemple pour SAT on peut promettre que le mot représente bien une clause.

9 Lemmes de codage de Sipser

Définition 9.1 Fonction de hachage

Une fonction de hachage h est une fonction de $[\![0, N-1]\!]$ dans quelque chose, où le quelque chose est stocké dans un tableau appelé table de hachage.

Pour avoir une bonne fonction de hachage en fait il faut tirer la table de hachage au hasard.

Définition 9.2 Fonction de hachage linéaire

Une fonction de hachage linéaire est une application linéaire de $(\mathbb{Z}/2\mathbb{Z})^m \to (\mathbb{Z}/2\mathbb{Z})^{m'}$ que l'on utilise comme une fonction de hachage.

Définition 9.3 Collision

Soit $H := (h_1, ..., h_l) : \Sigma^m \to \Sigma^{m'}$, où $\Sigma = \mathbb{Z}/2\mathbb{Z}$.
Une collision x dans X est un point de X tel que les points $y_1, ..., y_l$
— sont dans X
— sont distincts de x
— tels que $h_1(x) = h_1(y_l), ..., h_l(x) = h_l(y_l)$.

Lemme 9.1 Premier lemme de codage

Si X est assez petit, alors tirer au sort $H := (h_1, ..., h_l)$ donne avec haute probabilité il n'y aura aucune collision.

Lemme 9.2 Deuxième lemme de codage

Si $|X| > l \cdot 2^{m'}$, où $l \geqslant m'$, alors quelque soit la manière de tirer au sort $H := (h_1, ..., h_l)$, X aura forcément une collision pour H.

Théorème 9.3

$\mathbf{AM} \subseteq \Pi p_2$

Théorème 9.4 Goldwasser-Sipser

Pour tout $k \geqslant 1$, $\mathbf{IP}[k] \subseteq \mathbf{AM}[k+1]$.

10 Théorème de Boppana-Hastad-Zachos

Théorème 10.1 Théorème de Boppana-Hastad-Zachos

Si $\mathbf{coNP} \subseteq \mathbf{AM}$ alors $\mathbf{PH}$ s'effondre au niveau 2.

Preuve du théorème 10.1.

cskifo

Corollaire 10.2

Si GI est $\mathbf{NP}$-complet alors $\mathbf{PH}$ s'effondre au niveau 2.

Preuve du corollaire 10.2.

$\mathbf{AM}$ est clos par réductions polynômiales.

Et là c'est le plus beau jour de ma vie.

Index des définitions

Index des résultats