# Classes de complexité

François Schwarzentruber

ENS Rennes

## Contents

**1 Définition des classes de complexité** **2**
1.1 Classes non stables par modèle de calcul . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Classes stables par modèle de calcul . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**2 Théorème de Savitch** **3**

**3 PSPACE** **4**
3.1 QBF : formules booléennes quantifiées . . . . . . . . . . . . . . . . . . . . . . . . . . 4
3.2 Jeux à deux joueurs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**4 Langages rationnels** **13**
4.1 Langage vide . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
4.2 Langage universel . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

**5 EXPTIME et NEXPTIME** **16**
5.1 $P \neq EXPTIME$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
5.2 Des jeux de plateau . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
5.3 Concision . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

**6 LOGSPACE et NLOGSPACE** **18**
6.1 Définitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
6.2 NL-complétude . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
6.3 NL-complétude de l'accessibilité . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
6.4 $NL \subseteq P$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
6.5 $NL =$ co-$NL$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
6.6 2SAT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
6.7 Problèmes $P$-complets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

# 1 Définition des classes de complexité

## 1.1 Classes non stables par modèle de calcul

**Définition 1 ()**
[Sip06, p. 229] Soit $f : \mathbb{N} \to \mathbb{R}^+$.

- $TIME(f) = \{L \mid L$ est décidé par une MT dét en temps $O(f(n))\}$;
- $NTIME(f) = \{L \mid L$ est décidé par une MT non dét en temps $O(f(n))\}$.

**Définition 2 ()**
[Sip06, p. 308] Soit $f : \mathbb{N} \to \mathbb{R}^+$ telle que $f(n) \geq n$.

- $SPACE(f) = \{L \mid L$ est décidé par une MT dét en espace $O(f(n))\}$;
- $NSPACE(f) = \{L \mid L$ est décidé par une MT non dét en espace $O(f(n))\}$.

## 1.2 Classes stables par modèle de calcul

**Définition 3 ()**

- $P = \bigcup_k TIME(n \mapsto n^k)$
- $NP = \bigcup_k NTIME(n \mapsto n^k)$
- $EXPTIME = \bigcup_k TIME(n \mapsto 2^{n^k})$
- $NEXPTIME = \bigcup_k NTIME(n \mapsto 2^{n^k})$
- $PSPACE = \bigcup_k SPACE(n \mapsto n^k)$
- $NPSPACE = \bigcup_k NSPACE(n \mapsto n^k)$
- $EXPSPACE = \bigcup_k SPACE(n \mapsto 2^{n^k})$
- $NEXPSPACE = \bigcup_k NSPACE(n \mapsto 2^{n^k})$

Stables par nombre de rubans, etc.

**Définition 4 ()**
co- $= \{L \subseteq \Sigma^* \mid \overline{L} \in$ $\}$.

**Définition 5 ()**
$L$ est -dur ssi tout problème dans se réduit à $L$ en temps polynomial.

**Définition 6 ()**
$L$ est -complet ssi $L$ est dans et est -dur.

## 2 Théorème de Savitch

**Théorème 1 (de Savitch)** *[Sip06, p. 310] Soit $f : \mathbb{N} \to \mathbb{R}^+$ telle que $f(n) \geq n$*[1] *et $f(n)$ calculable en espace $O(f(n))$. $NSPACE(f) \subseteq SPACE(f^2)$.*

Démonstration.

Soit $L$ un langage dans $NSPACE(f)$. Il existe une machine de Turing $M$ non déterministe qui décide $L$ avec un espace $f(n)$ quitte à multiplier $f$ par une constante. Sans perte de généralité, on suppose que $M$ efface son ruban et place le curseur à gauche avant d'accepter un mot : cette configuration s'appelle $c_{accept}$ de $M$. Voici un algorithme déterministe qui décide $L$ :

```
procedure deciderL(x)
    if  [c_accept accessible depuis c_ini(x) dans le graphe des configurations de M]
        accepter
    else
        rejeter
```

Implémentation de [ ]. Dans le graphe des configuration de $M$, seules les configurations $M$ de taille de ruban $f(|x|)$ au plus sont accessibles depuis $c_{ini}(x)$. Il y en a au plus

$$T(|x|) = |Q| \times |\Sigma|^{f(|x|)} \times f(|x|).$$

Il existe $d$ tel que $T(|x|) \leq 2^{df(|x|)}$.

$c_{accept}$ **accessible** depuis $c_{ini}(x)$ ssi il existe un chemin de $c_{ini}(x)$ à $c_{accept}$ de longueur au plus $2^{df(|x|)}$.

On implémente [ ] alors [chemin?$(c_{ini}(x), c_{accept}, 2^{df(|x|)})$] où chemin? est une fonction conçue avec **diviser pour régner** :

```
function chemin?(c1, c2, t)
    if t = 1
        return vrai si c1 = c2 ou c1 →^M c2 ; non, sinon
    else
        for c configuration de M de taille de ruban au plus f(n) do
            if chemin?(c1, c, t/2) et chemin?(c, c2, t/2) then
                return vrai
        return faux
```

La complexité spatiale de *chemin?*$(c_1, c_2, t)$ est $C(t) = O(f(|x|)) + C(\frac{t}{2})$, c'est à dire $C(t) = O(log_2 t f(|x|))$. D'où une complexité spatiale pour deciderL$(x)$ de $C(2^d f(|x|)) = \underbrace{O(f(|x|)}_{\text{calcul de } f(|x|)} + O(f(|x|)^2) = O(f|x|)^2$. ■

**Corollaire 1** *PSPACE = NPSPACE.*

[1] $f(n) \geq n$ car on doit comptabiliser au moins la taille de l'entrée.

# 3 PSPACE

## 3.1 QBF : formules booléennes quantifiées

Une formule propositionnelle $\varphi$ est **satisfiable** ssi **il existe** une valuation $\nu$ telle que $\nu \models \varphi$.

**SAT**

- entrée : une formule propositionnelle $\varphi$
- sortie : oui si $\varphi$ est satisfiable ; non sinon.

est **NP**-complet.

Une formule propositionnelle $\varphi$ est **valide** ssi **pour toute** valuation $\nu$ telle que $\nu \models \varphi$.

**VALIDE**

- entrée : une formule propositionnelle $\varphi$
- sortie : oui si $\varphi$ est valide ; non sinon.

est co-**NP**-complet.

But : définir un problème PSPACE-complet, **TQBF**, qui généralise **SAT** et **VALIDE**

### 3.1.1 Syntaxe

**Définition 7 (formule booléenne quantifiée)**
Une **formule booléenne quantifiée** (sous forme prénexe) est une formule de la forme

$$\exists p_1 \forall p_2 \exists p_3 \dots Q_n p_n \chi$$

où $Q_{2k} = \forall$ et $Q_{2k+1} = \exists$ et $\chi$ est une formule de la logique propositionnelle.

Une formule booléenne quantifiée est **close** si toutes les variables sont sous la portée d'un quantificateur.

### 3.1.2 Sémantique

**Définition 8 (conditions de vérité)**

- $\nu \models \chi$ comme en logique propositionnelle si $\chi$ est propositionnelle ;
- $\nu \models \exists x\varphi$ ssi $\nu[x := 0] \models \varphi$ ou $\nu[x := 1] \models \varphi$ ;
- $\nu \models \exists x\varphi$ ssi $\nu[x := 0] \models \varphi$ et $\nu[x := 1] \models \varphi$.

Lorsque $\varphi$ est close, $\nu \models \varphi$ ne dépend pas de $\nu$ et on dira que $\varphi$ est vraie s'il existe telle que $\nu \models \varphi$.

### 3.1.3 Problème de décision

**TQBF**

- entrée : une formule booléenne quantifiée close $\varphi$
- sortie : oui si $\varphi$ est vraie ; non sinon.

#### 3.1.4 Dans PSPACE

**Théorème 2** *[Sip06](p. 285-287)* **TQBF** *est dans PSPACE.*

Démonstration.
Voici un algorithme qui vérifie que $\nu \models \varphi$ en temps polynomial en $|\nu|$ et $|\varphi|$ :

```
function tqbf(ν, φ)
  match φ
    ∃pψ: tqbf(ν[p := 0], ψ ou tqbf(ν[p := 1], ψ
    ∀pψ: tqbf(ν[p := 0], ψ et tqbf(ν[p := 1], ψ
    ψ propositionnelle: oui si ν ⊨ ψ ; non sinon.
```

■

#### 3.1.5 PSPACE-dur

**Théorème 3** **TQBF** *est NPSPACE-dur.*

Démonstration.
Soit $L$ un problème NPSPACE. On réduit $L$ à **TQBF** en temps polynomial. Il existe donc une machine de Turing $M$...

on reprend la démonstration du théorème de Savitch...

Pour toute instance $x$ de $L$, on crée une formule booléenne quantifiée qui exprime que 'il existe un chemin de $c_{ini}(x)$ à $c_{accept}$ de longueur au plus $2^{df(|x|)}$' :

$$tr(x) := (\vec{c_{ini}} = \text{config initiale avec } x) \wedge (\vec{c_{acc}} = \text{config acceptante}) \wedge ch?(\vec{c_{ini}}, \vec{c_{acc}}, 2^{df(|x|)}).$$

où $ch?(\vec{c}_1, \vec{c}_2, 2^k)$ exprime 'chemin?$(c_1, c_2, 2^k)$ renvoie vrai', où $\vec{c}_1, \vec{c}_2$ sont des collections de propositions atomiques qui représentent les configurations $c_1$ et $c_2$ et est définie par induction sur $k$ :

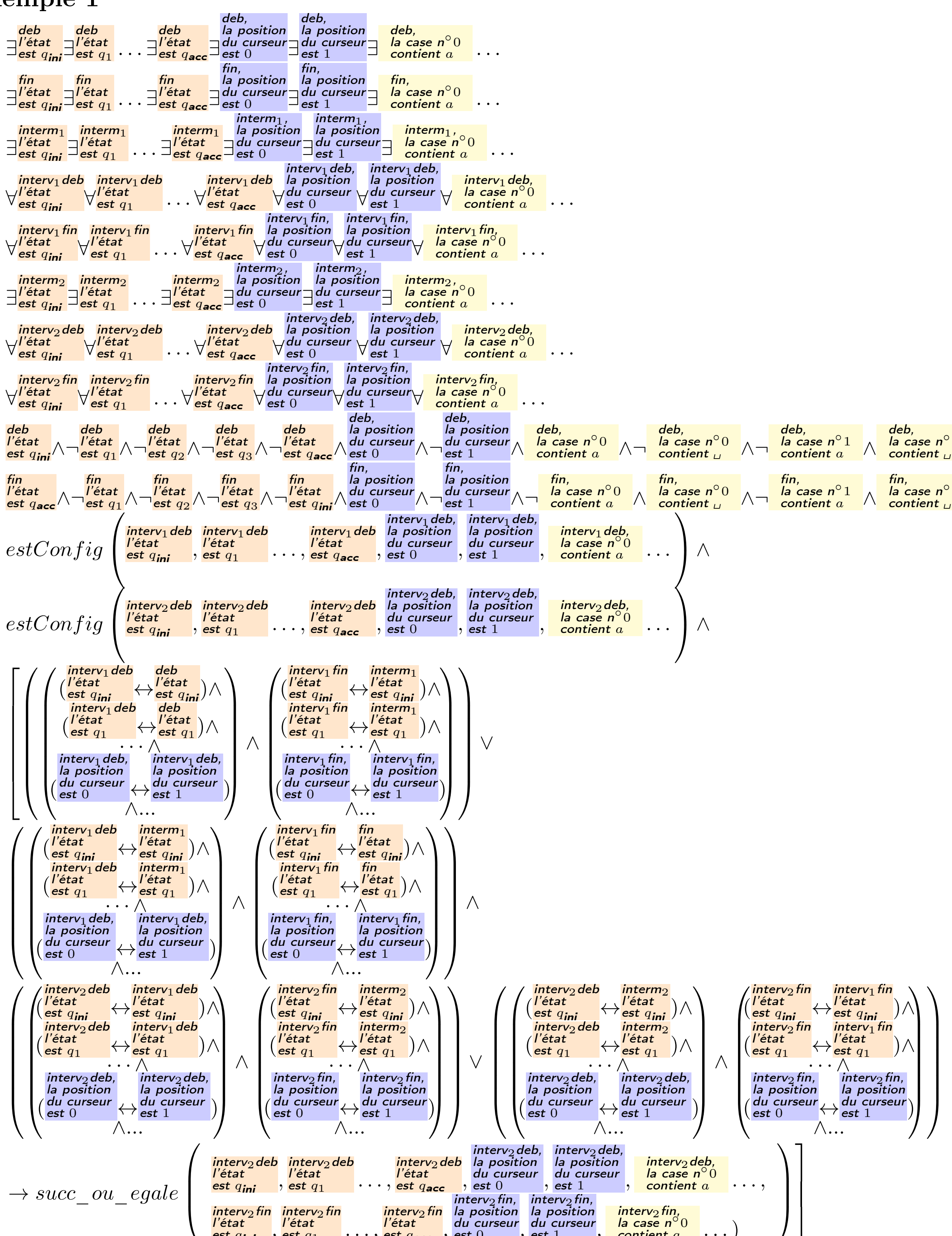
- $ch?(\vec{c}_1, \vec{c}_2, 2^0) = (\vec{c}_1 = \vec{c}_2) \vee succ(\vec{c}_1, \vec{c}_2)$ ;
- Si $k > 0$,

$$ch?(\vec{c}_1, \vec{c}_2, 2^k) = \cancel{\exists \vec{c}, estConfig(\vec{c}) \wedge ch?(\vec{c}_1, \vec{c}, 2^{k-1}) \wedge ch?(\vec{c}, \vec{c}_2, 2^{k-1})}$$

$$= \exists \vec{c}, estConfig(\vec{c}) \wedge \left(\forall (\vec{d}, \vec{d'}) \in \{(\vec{c}_1, \vec{c}), (\vec{c}, \vec{c}_2)\} ch?(d, d', 2^{k-1})\right)$$

Par construction, $x \in L$ ssi $tr(x)$ est QBF-vraie. On peut écrire un algorithme qui calcule $tr(x)$ en temps polynomial en $|x|$. ■

## Exemple 1



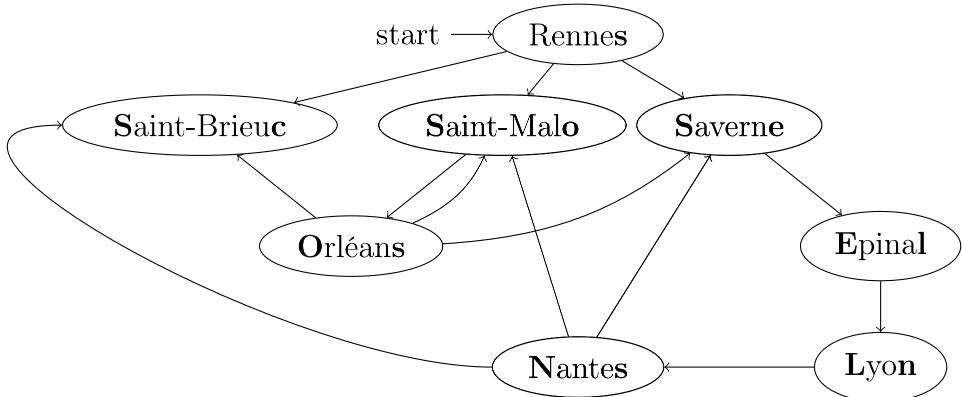
## 3.2 Jeux à deux joueurs

On utilise **TQBF** pour montrer la PSPACE-dureté de problèmes :

- model checking du premier ordre ;
- Jeu de géographie généralisé, etc., puis le Go ;
- Reversi, Hex, etc.

Á l'inverse, on peut modéliser des jeux à deux joueurs avec **TQBF** par exemple les échecs ([KS08], chap. 9, p. 209).

### 3.2.1 Jeu de géographie généralisé




On considère le jeu à deux joueurs suivant. Soit $G$ un graphe fini et $s$ un sommet de départ. Un jeton est placé dans $s$. Une action d'un joueur consiste à supprimer le sommet du graphe où il y a le jeton et à le déplacer dans l'un des successeurs. Un joueur perd lorsque le jeton est dans un sommet sans successeur.

On considère le problème de décision ([Sip06], p. 289):

**GEOGRAPHIE**

- entrée : un graphe $G$, un sommet $s$ de $G$ ;
- sortie : oui si le joueur 1 a une stratégie gagnante à partir de $G, s$ ; non, sinon.

### 3.2.2 GEOGRAPHIE est PSPACE-complet

**Proposition 1** ***GEOGRAPHIE*** *est dans PSPACE.*

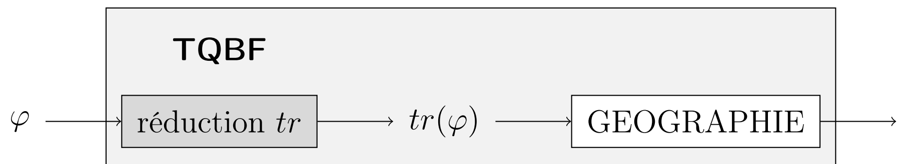
DÉMONSTRATION.

```
function joueurgagne(G, s)
    for t successeur de s dans G do
        if joueurgagne(G \ {t}, t) = faux
            return vrai
    return faux
```

■

**Proposition 2** ***GEOGRAPHIE*** *est PSPACE-dur.*

Démonstration.




Soit $\varphi$ une formule booléenne quantifiée close. On suppose qu'elle est de la forme

$$\exists p_1 \forall q_1 \ldots \exists p_k \forall q_k \psi$$

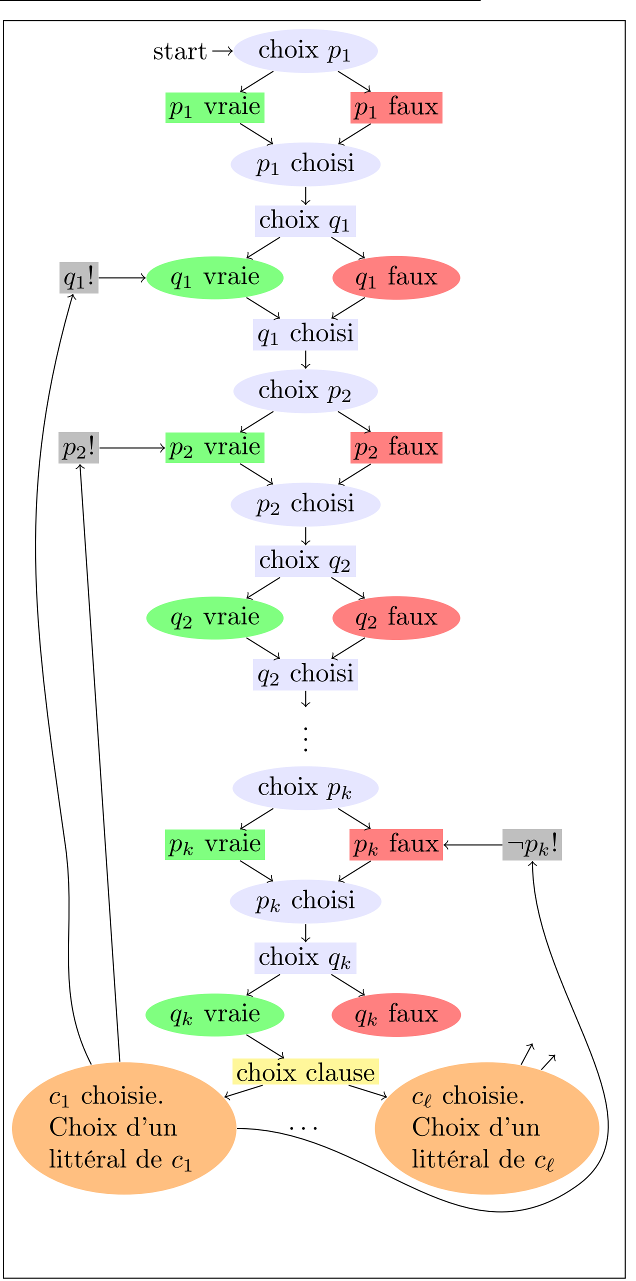
où $\psi$ est une forme normale conjonctive, c'est à dire

$$\psi = (c_1 \wedge \cdots \wedge c_\ell)$$

où $c_i$ est une clause.
$tr(\varphi)$ est donné par le graphe dessiné sur la droite où le sommet initial est 'choix $p_1$'. Sur le dessin, on a pris l'exemple

$$c_1 = (q_1 \vee p_2 \vee \neg p_k).$$

- $tr(\varphi)$ est calculable en temps polynomial en $|\varphi|$.
- $\varphi$ est QBF-vraie ssi le joueur 1 a une stratégie gagnante au jeu de géographie avec le graphe pointé $tr(\varphi)$.




■

### 3.2.3 Jeu de géographie planaire

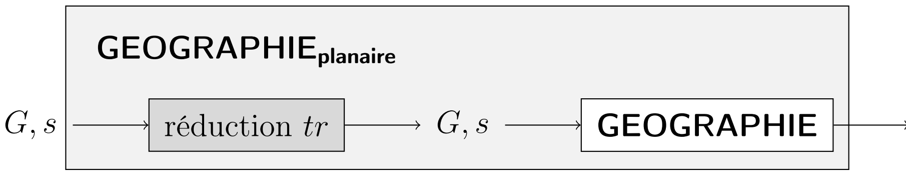
On s'intéresse au même problème de décision mais restreint aux graphes planaires.

**GEOGRAPHIE$_{\text{planaire}}$**

- entrée : un graphe $G$ **planaire**, un sommet $s$ de $G$ ;
- sortie : oui si le joueur 1 a une stratégie gagnante à partir de $G, s$; non, sinon.

**Proposition 3** ***GEOGRAPHIE$_{\text{planaire}}$*** *est dans PSPACE.*

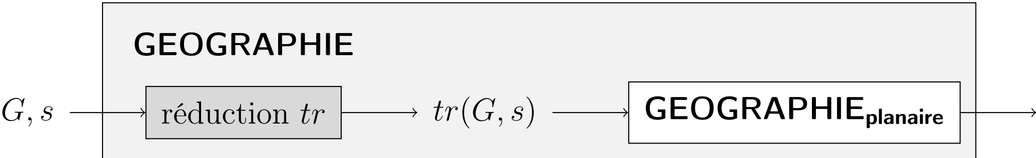
Démonstration.



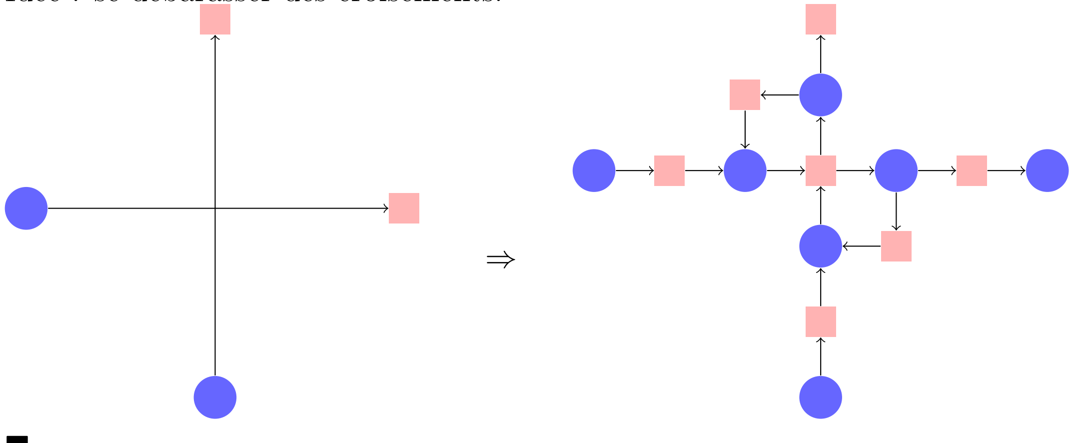

■

**Proposition 4** ***GEOGRAPHIE$_{\text{planaire}}$*** *est PSPACE-dur.*

Démonstration.




Idée : se débarasser des croisements.



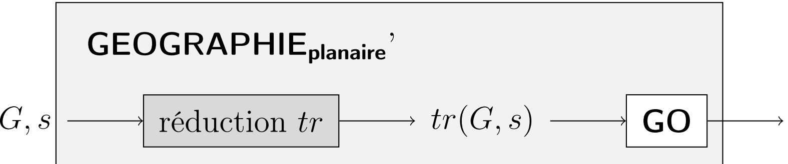

■

### 3.2.4 Jeu de go

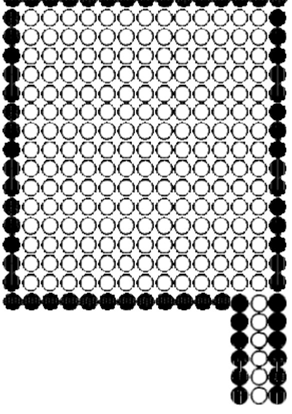
**GO**
- entrée : un plateau de Go ;
- sortie : oui si le joueur 1 (noir) a une stratégie gagnante à partir de $G, s$; non, sinon.

**Théorème 4** ***GO*** *est PSPACE-dur.*

Démonstration.



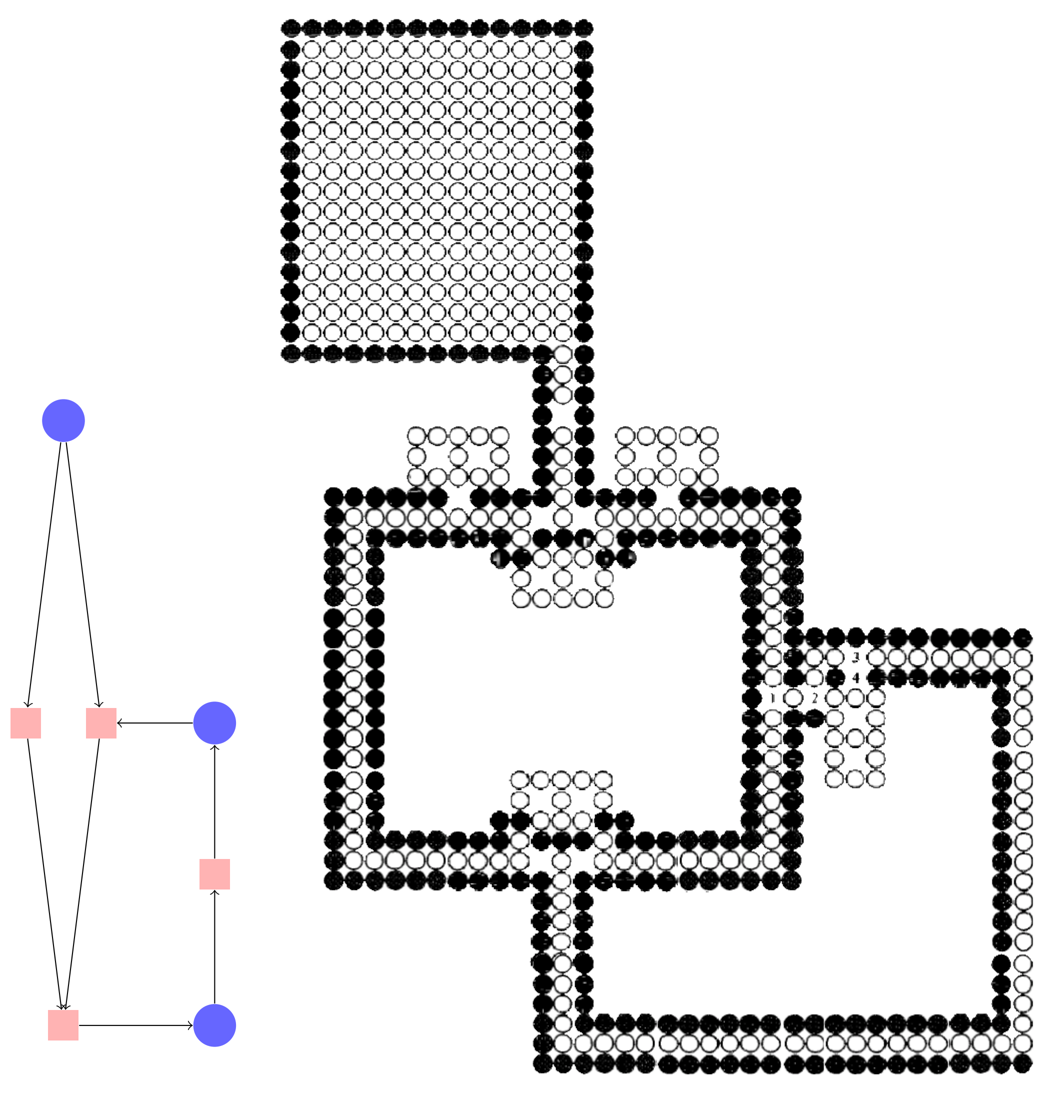

$tr(G, s)$ est le plateau de Go suivant :



ici on accroche le reste du plateau obtenu à partir de $G, s$

| Motif dans $G$ | Portion de plateau de Go correspondante |
| --- | --- |
| | |
| | |
| | |
| | |
| | |

**Exemple 2**




■

# 4 Langages rationnels

Réf : [AH74][p. 395, section 10.6]

## 4.1 Langage vide

**VIDE EXPRESSION RAT**

- entrée : Une expression régulière $e$
- sortie : oui si $L(e) = \emptyset$ ; non, sinon.

**Théorème 5** ***VIDE EXPRESSION RAT*** *est dans* $\boldsymbol{P}$*.*

Démonstration.

On réduit **VIDE EXPRESSION RAT** au problème de non-accessibilité en temps polynomial : on transforme l'expression rationnelle $e$ en un automate fini non-déterministe avec $\epsilon$-transitions $\mathcal{A}_e$ telle $L(\mathcal{A}_e) = L(e)$. $L(e) = \emptyset$ est équivalent au fait qu'aucun état final n'est accessible depuis l'état initial dans $\mathcal{A}_e$. Comme le problème de non-accessibilité est dans **P**, **VIDE EXPRESSION RAT** est dans **P**. ■

## 4.2 Langage universel

**UNIVERSALITE EXPRESSION RAT**

- entrée : Une expression régulière $e$
- sortie : oui si $L(e) = \Sigma^*$ ; non, sinon.

**Théorème 6** ***UNIVERSALITE EXPRESSION RAT*** *est dans PSPACE.*

Démonstration.

On construit un algorithme qui requiert un espace polynomial en $O(|e|)$ :

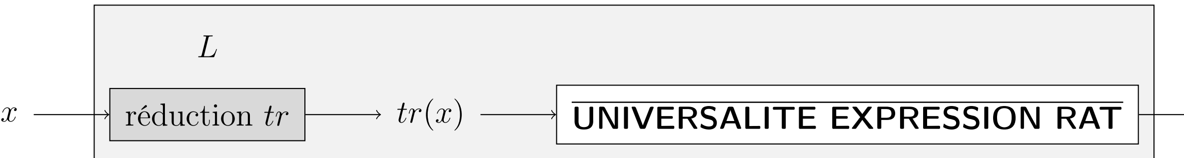
```
procedure nonuniverselle?(e : expression rationnelle)
    A := construire automate non-déterministe avec ε-transitions
                                              tel que L(A) = L(e);
    S := clotûre de {etat initial de A}
    tant que  S contient un état final
        choose lettre a
        S := cloture des a-sucesseurs des états dans S
    accepter
```

■

**Théorème 7** ***UNIVERSALITE EXPRESSION RAT*** *est PSPACE-dur.*

DÉMONSTRATION.

Soit $L$ un problème dans PSPACE. On réduit $L$ à $\overline{\textsf{UNIVERSALITE EXPRESSION RAT}}$ en temps polynomial.




Soit $M$ une machine de Turing qui accepte $L$. Soit $x$ une instance de $M$. Il existe un polynôme $f$ tel que, pour tout instance $x$, dans l'exécution depuis $x$, le ruban de la machine est borné par $f(|x|)$. Idée : $tr(x)$ est une expression régulière telle que

$$L(tr(x)) = \{\text{mots qui ne représentent pas une exécution acceptante de } M(x)\}.$$

**Convention de notations.** Une exécution de $M$ est un mot de la forme

$$▦ w_1 ▦ w_2 \dots w_k ▦$$

où $k \geq 1$, ▦ est un symbole qui sépare les configurations, $w_i$ sont des mots de longueur exactement $N = f(|x|)$.

**Exemple 3**

$$▦ \begin{bmatrix} a \\ q_0 \end{bmatrix} b\text{␣␣␣} ▦ b \begin{bmatrix} b \\ q \end{bmatrix} \text{␣␣␣} ▦ ba \begin{bmatrix} \text{␣} \\ q' \end{bmatrix} \text{␣␣} ▦ \dots$$

| | |
|---|---|
| $\Sigma$ | alphabet contenant l'alphabet du ruban, les couples $\begin{bmatrix} \text{lettre du ruban} \\ \text{état} \end{bmatrix}$ et ▦ |
| $\Delta$ | alphabet contenant l'alphabet du ruban, les couples 'lettre du ruban/état' |
| $R$ | alphabet du ruban |
| $C$ | alphabet des couples $\begin{bmatrix} \text{lettre du ruban} \\ \text{état} \end{bmatrix}$ |
| $A$ | alphabet des couples $\begin{bmatrix} \text{lettre du ruban} \\ acc \end{bmatrix}$ |

**Définition de $tr(x)$.** $tr(x)$ est l'union des expressions rationnelles suivantes qui résument le fait qu'un mot ne représente pas une exécution acceptante :

| Expression | Signification |
|---|---|
| $\Delta^*$ | Pas de symbole 🧱 |
| $\Delta^*$🧱$\Delta^*$ | Qu'un seul symbole 🧱 |
| $\Delta\Sigma^*$ | Ne commence pas avec 🧱 |
| $\Sigma^*\Delta$ | Ne termine pas avec 🧱 |
| $\Sigma^*$🧱$R^*$🧱$\Sigma^*$ | Configuration sans curseur |
| $\Sigma^*$🧱$\Delta^*C\Delta^*C\Delta^*$🧱$\Sigma^*$ | Configuration avec au moins 2 curseurs |
| $\Sigma^*$🧱🧱$\Sigma^*$ | Configuration avec un ruban de longueur 0 |
| $\Sigma^*$🧱$\Delta$🧱$\Sigma^*$ | Configuration avec un ruban de longueur 1 |
| $\Sigma^*$🧱$\Delta\Delta$🧱$\Sigma^*$ | Configuration avec un ruban de longueur 2 |
| $\vdots$ | $\vdots$ |
| $\Sigma^*$🧱$\Delta^{N-1}$🧱$\Sigma^*$ | Configuration avec un ruban de longueur $N-1$ |
| $\Sigma^*$🧱$\Delta^{N+1}\Delta^*$🧱$\Sigma^*$ | Configuration avec un ruban de longueur $\geq N+1$ |
| 🧱$(\Delta \setminus \{\left[\begin{smallmatrix} x_1 \\ q_0 \end{smallmatrix}\right]\}\Delta^*$🧱$\Sigma^*$ | 1ère case du ruban non conforme à la configuration initiale |
| 🧱$(\Delta^{i-1}(\Delta\setminus\{x_i\})\Delta^*$🧱$\Sigma^*$ où $x_i = \llcorner$ si $i > \|x\|$ | $i^e$ case du ruban non conforme à la configuration initiale |
| $(\Sigma \setminus A)^*$ | Pas d'états acceptants |
| $\Sigma^* c_1 c_2 c_3 \Sigma^{N-1}(\Sigma \setminus f(c_1c_2c_3))\Sigma^*$ | Transition non conforme |

où $f$ sont des fonctions qui correspondent aux transitions :

| $c_1$ | $c_2$ | $c_3$ |
|---|---|---|
| | $f(c_1c_2c_3)$ | |

**Exemple 4** *Par exemple :*

- $f($🧱$, \left[\begin{smallmatrix} a \\ q_0 \end{smallmatrix}\right], b) = c$ *si la transition depuis l'état $q_0$ en lisant $a$ écrit $c$ ;*
- $f(\left[\begin{smallmatrix} a \\ q_0 \end{smallmatrix}\right], b, d) = \left[\begin{smallmatrix} b \\ q \end{smallmatrix}\right]$ *si la transition depuis $q_0$ en lisant $a$ va dans $q$ ;*
- $f(b, c, d) = c$.

■

# 5 EXPTIME et NEXPTIME

## 5.1 $P \neq EXPTIME$

**Remarque 1** *Les démonstrations dans [Pap][p. 145] utilisent la notion de fonction propre*[2]*. C'est joli mais ça ne sert à rien car les fonctions manipulées sont des polynômes et sont donc propres.*

**Théorème 8** *[Pap][p. 145]* $P \subsetneq EXPTIME$.

DÉMONSTRATION.

**Lemme 1** $P \subseteq TIME(2^n) \subseteq TIME((2^{2n+1})^3) \subseteq EXPTIME$.

DÉMONSTRATION.

$P \subseteq TIME(2^n)$ provient du fait que tout polynôme est un $O(2^n)$.
$TIME((2^{2n+1})^3) \subseteq EXPTIME$ car $(2^{2n+1})^3 = O(2^{6n+3}) = O(2^{n^2})$.
■

**Lemme 2** $TIME(2^n) \subsetneq TIME((2^{2n+1})^3)$.

DÉMONSTRATION.

Soit $f(n) = 2^{2n+1}$. Considérons le problème de l'arrêt en temps $f$ :

**H**

- entrée : une machine de Turing $M$, déterministe, et une entrée $x$
- sortie : oui si $M$ accepte $x$ en moins de $f(|x|)$ étapes ; non, sinon.

On montre que **H** $\in TIME((2^{2n+1})^3)$ et **H** $\notin TIME(2^n)$.

**Première partie :** **H** $\in TIME((2^{2n+1})^3)$

Le problème se résout à l'aide de l'algorithme suivant :

```
arretEnTempsF(M, x)
    exécuter f(|x|) étapes de calcul de M(x)
    if l'exécution est acceptante
        accepter
    else
        rejeter
```

Dans [Pap][p. 142], on trouve la démonstration que l'on peut implémenter cet algorithme avec une machine de Turing qui utilise un temps $O(f(n)^3)$, où $n$ est la taille de $M$ plus la taille de $x$.

[2]c'est à dire des fonctions croissantes et calculables en temps $f$ et en espace $f$

**Deuxième partie : H $\notin TIME(2^n)$**

D'autre part, montrons qu'il n'existe aucune machine de Turing qui résout le problème **H** avec un temps $f(\lfloor\frac{n}{2}\rfloor)$. Supposons au contraire qu'il existe une machine de Turing $M_{\mathsf{H}}$ qui résout le problème **H** en temps $f(\lfloor\frac{n}{2}\rfloor)$. On construit alors la machine $D$.

```
function D(M)
  if M_H accepte (M, M)
  |  rejeter
  else
  |  accepter
```

$M_{\mathsf{H}}(M)$ requiert $f(\lfloor\frac{|M|+|M|+1}{2}\rfloor) = f(|M|)$ étapes de calcul. On a alors :

D rejette D ssi $M_{\mathsf{H}}(D, D)$ répond oui
ssi $D$ accepte $D$ en moins de $f(\lfloor\frac{|D|+|D|+1}{2}\rfloor)$ étapes.
ssi $D$ accepte $D$ en moins de $f(\lfloor|D|\rfloor)$ étapes.
ssi $D$ accepte $D$.

Contradiction. Donc il n'existe pas de telle machine $M_{\mathsf{H}}$. ■ ■

## 5.2 Des jeux de plateau

Sont EXPTIME-complets les jeux d'échecs [FL81], le jeu de Go (avec la règle Ko)[Rob83], etc. Pour la culture, $EXPTIME = APSPACE$ où $APSPACE$ est la classe des problèmes décidés par une machine de Turing alternante[CS76] en espace polynomial. Intuitivement, le 'alternant' aux jeux à deux joueurs et le 'espace polynomial' correspondant à un 'plateau' de taille polynomiale.

## 5.3 Concision

EXPTIME et NEXPTIME contiennent respectivement les versions exponentiellement plus concises des problèmes de P et NP ([Pap], p. 492).

**Théorème 9** *([Pap], p. 491, Th. 20.1) Si N = NP, alors EXPTIME = NEXPTIME.*

Démonstration.

Soit $L$ un problème dans NEXPTIME. Montrons qu'il est dans EXPTIME. Il existe une machine de Turing non déterministe qui décide $L$ en temps $O(2^{n^k})$. Soit

$$L' = \{x \underbrace{\textvisiblespace\textvisiblespace\cdots\textvisiblespace}_{\text{il y a } 2^{n^{|x|}} - |x| \text{ caractères `}\textvisiblespace\text{'}} \mid x \in L\}$$

On a $L' \in NP$ (même machine).
Donc $L' \in NP = P$.
Donc $L \in EXPTIME$.
■

# 6 LOGSPACE et NLOGSPACE

## 6.1 Définitions

Ici, on utilise le modèle de machine de Turing à plusieurs rubans et le ruban d'entrée est en lecture seule. On ne mesure que la mémoire utilisée sur les autres rubans.

**Définition 9 (L et NL)**

- **L** $= SPACE(\log n)$
- **NL** $= NSPACE(\log n)$

**L** et **NL** sont stables par changement de modèles de machine (plusieurs rubans etc.).

**Exemple 5** ***ACCESSIBILITE***

- entrée : *un graphe orienté $G$, deux sommets $s$ et $t$ ;*
- sortie : *oui, s'il existe un chemin de $s$ vers $t$ dans $G$.*

*On ne sait pas si* ***ACCESSIBILITE*** *est dans L. On pourrait imaginer faire comme dans Savitch (diviser pour régner) mais ça coûte $\log^2 n$ et pas $\log n$ ! Mais par contre :*

***Proposition 5*** *[Pap] (example 2.10 p. 48-49)* ***ACCESSIBILITE*** *est dans* ***NL****.*

DÉMONSTRATION.
Voici un algorithme non-déterministe pour résoudre **ACCESSIBILITE** :

```
function path?(G, s, t)
    w = s
    while w ≠ t do
        w := choisir un successeur de w
    accepter
```

■

## 6.2 NL-complétude

Une réduction polynomiale serait stupide ! On utilise ici :

**Définition 10 (réduction en espace logarithmique)**
Un problème $A$ se réduit à un autre $B$ en espace logarithmique si il existe une fonction calculable en espace logarithmique qui calcule $tr : \Sigma^* \to \Sigma^*$ telle que $x \in A$ ssi $f(x) \in B$.

$tr$ calculable en espace logarithmique, cela signifie qu'il existe un transducteur, c'est à dire une machine de Turing à trois rubans :

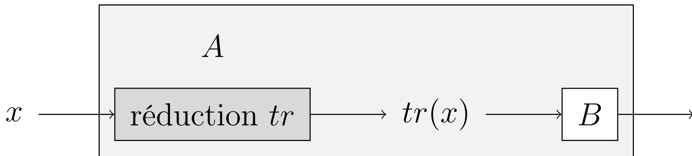
- le ruban d'entrée contenant $x$, en lecture seule ;
- un ruban de travail en lecture/écriture qui contient au plus $O(\log|x|)$ symboles ;
- un ruban de sortie qui contient $tr(x)$ à la fin, en écriture seule.

**Définition 11 ()**
$P$ est NL-complet ssi dans $P \in NL$ et tout problème NL se réduit en espace logarithmique se réduit à $P$.

**Théorème 10**

1. *Si $A$ se réduit à $B$ en espace logarithmique et $B \in L$ alors $A \in L$ ;*
2. *Si $A$ se réduit à $B$ en espace logarithmique et $B \in NL$ alors $A \in NL$ ;*
3. *Si $A$ se réduit à $B$ en espace logarithmique et $B \in coNL$ alors $A \in coNL$.*

Démonstration.
[1.] Le schéma suivant ne donne pas directement un algorithme en espace $O(log|x|)$ :




On obtient un algorithme pour $A$ à partir de $tr$ et d'un algorithme pour $B$ : le squelette est l'algorithme de $B$ et quand on a besoin du $i$ symbole du mot $tr(x)$, on utilise la machine pour $tr$ comme sous-routine.
[2. 3.] Même principe. ■

**Théorème 11** *Si on montre qu'un problème NL-complet est dans L, alors ils le sont tous.*

## 6.3 NL-complétude de l'accessibilité

**Théorème 12** ***ACCESSIBILITE*** *est NL-complet.*

DÉMONSTRATION.
Dans NL cf proposition 5.

NL-dur Soit $A$ un problème NL. Il existe une machine non-déterministe $M$ (à deux rubans comme décrit plus haut) qui décide $A$ en espace $O(\log n)$. On construit une réduction $tr$ en espace logarithmique de $A$ vers **ACCESSIBILITE**. Sans perte de généralité, on suppose que $M$ n'a qu'une seule configuration acceptante.

On conçoit une machine qui écrit le graphe des configurations $G_x$ de la machine $M$ sur une entrée $x$, en espace logarithmique par rapport à $|x|$. Soit $s_x$ la configuration initiale de $M$ avec $x$ sur le ruban d'entrée. Soit $t$ l'unique configuration finale acceptante de $M$. On a $x \in A$ ssi $tr(x) := (G_x, s_x, t) \in$ **ACCESSIBILITE**.
■

En fait, considérons la restriction[3] :

**ACCESSIBILITE**$_{acyclique}$

- entrée : Un graphe $G$ **acyclique**, $s$, $t$
- sortie : oui, s'il existe un chemin de $s$ à $t$ dans $G$ ; non, sinon.

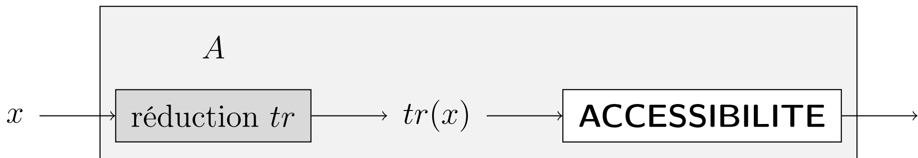
**Théorème 13** ***ACCESSIBILITE***$_{acyclique}$ *est NL-complet.*

DÉMONSTRATION.
Dans NL cf proposition 5. NL-dur Voir démonstration du théorème 12. En supposons que la machine $M$ ne boucle pas, le graphe $G_x$ est acyclique.
■

## 6.4 $NL \subseteq P$

**Théorème 14** $NL \subseteq P$.

DÉMONSTRATION.
Soit $A$ dans NL. Comme **ACCESSIBILITE** est NL-dur, on a la réduction en espace logarithmique :




Comme et **ACCESSIBILITE** dans **P** (parcours en profondeur par exemple), et que les calculs de la réduction peuvent être réalisé en temps polynomial, le schéma ci-dessus donne un algorithme en temps polynomial pour $A$. ■

[3]Dans [Pap], cette restriction est "diluée" dans la démonstration du théorème 16.3, p. 398

## 6.5 $NL =$ co-$NL$

**Proposition 6** $\overline{\textsf{ACCESSIBILITE}}$ *est dans NL.*

DÉMONSTRATION.

On écrit un algorithme de la forme :

```
procedure nonpath?(G, s, t)
|   nonpathnb?(G, s, t, getacc(G, s)).
```

où

1. $nonpathnb?(G, s, t, c)$ est appelé pour $c =$ 'nombre de sommets accessibles dans $G$ depuis $s$' et a une branche acceptante ssi il n'y a pas de chemin de $s$ à $t$.
2. $getacc(G, s)$ a une branche acceptante et les seules branches acceptantes retourne le nombre de sommets accessibles dans $G$ depuis $s$.

1.

```
function nonpathnb?(G, s, t, c)
|   n := 0
|   for u sommet de G différent de t do
|   |   choose b ∈ {0, 1}
|   |   if b = 1
|   |   |   path?(G, s, u)
|   |   |   n := n + 1
|   if n ≠ c rejeter
|   accepter
```

L'algorithme est correct : $t$ n'est pas accessible depuis $s$ ssi l'ensemble des sommets accessibles (de taille $c$) est inclus dans $G \setminus \{t\}$ ssi il existe une branche de l'algorithme qui réussit.

2.

```
function getacc(G, s)
|   getnumber(G, s, |G|)
```

où $getnumber(G, s, i)$ est une fonction non-déterministe telles que :

- il existe au moins une exécution de $getnumber(G, s, i)$ qui n'échouent pas ;
- toutes les exécutions de $getnumber(G, s, i)$ qui n'échouent pas renvoient le nombre de sommets accessibles depuis $s$ en au plus $i$ étapes dans $G$.

Elle utilise $path?(G, s, t, i)$, une procédure qui décide en espace logarithmique le problème d'accessibilité suivant :

**ACCESSIBILITE**$_{etapes}$

- entrée : $G, s, t, i$
- sortie : oui si $t$ accessible depuis $s$ en au plus $i$ étapes.

**Remarque 2** *Contrairement à [Sip06], on écrit ici une fonction récursive. La fonction récursive n'est pas bonne car il faut stocker la pile d'appel. Je laisse le soin de dérécursifier cette fonction ou de lire [Sip06] afin de bien avoir un espace logarithmique. En tout cas, la fonction récursive présentée dans la suite suffit à comprendre pourquoi le non-déterminisme suffit à compter le nombre de sommets accessibles en au plus i étapes.*

```
function getnumber(G, s, i)
    if i = 0
        return 1
    else
        c' := getnumber(G, s, i − 1)
        n := 0
        for v sommet de G do
            n' := 0
            for u sommet de G do
                choose b ∈ {0, 1}
                if b = 1
                    path?(G, s, u, i − 1)
                    n' := n' + 1
                    if u →^G v
                        n := n + 1
                        break
            if n' ≠ c' rejeter
        return n
```

Si $i = 0$, c'est 1. Sinon, on suppose que l'on a le nombre de sommets accessibles en au plus $i - 1$ étapes. Ensuite, on parcourt les sommets $v$. On va tester pour chacun d'eux s'ils sont accessibles en $i$ étapes. $n$ est le compteur qui compte de tels sommets. Pour chaque $v$, on parcourt les sommets $u$ accessibles en $i - 1$ étapes. C'est toujours la même astuce. On ne peut a priori en espace logarithme. Sauf, que de la même façon, on devine pour chacun d'eux. On les compte avec $n'$ etc.

■

**Théorème 15** *NL = coNL. [Sip06][p. 331]*

Démonstration.

$\boxed{\subseteq}$ Soit $A$ dans NL. Comme **ACCESSIBILITE** est NL-complet, $A$ se réduit à **ACCESSIBILITE** en espace logarithmique. Par le théorème 10, comme **ACCESSIBILITE** est dans coNL, alors $A$ est dans coNL.

$\boxed{\supseteq}$ Soit $A$ dans coNL. Comme **ACCESSIBILITE** est NL-complet, $\overline{A}$ se réduit à **ACCESSIBILITE** en espace logarithmique. Par le théorème 10, comme **ACCESSIBILITE** est dans coNL, alors $\overline{A}$ est dans coNL. Donc $A$ est dans NL.

■

## 6.6 2SAT

**Proposition 7** *2SAT $\in$ NL.*

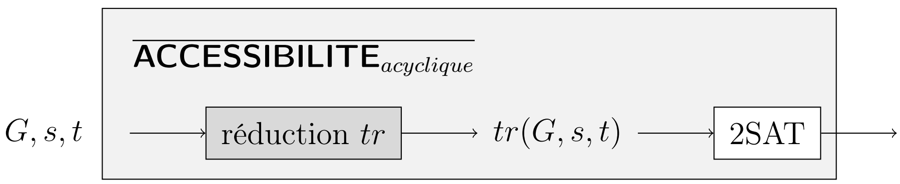
DÉMONSTRATION.
Comme coNL = NL, il suffit de montrer que $\overline{2SAT}$ est dans NL. Voici un algorithme non-déterministe en espace logarithmique qui accepte $\overline{2SAT}$ :

```
procedure 2sat‾(φ)
    choisir une variable propositionnelle p dans φ
    ℓ := p
    while ℓ ≠ ¬p do
        Choisir une clause de la forme ℓ → ℓ′
        ℓ := ℓ′
    while ℓ ≠ p do
        Choisir une clause de la forme ℓ → ℓ′
        ℓ := ℓ′
    accepter
```

■

**Proposition 8** *([Pap], p. 398, Th. 16.3) 2SAT est NL-dur.*

DÉMONSTRATION.
Comme NL = coNL, le problème $\overline{\textsf{ACCESSIBILITE}_{acyclique}}$ est aussi NL-dur.
On donne une réduction en espace logarithmique :




$$tr(G,s,t) = s \quad \wedge \quad \neg t \quad \wedge \quad \bigwedge_{\text{arc } (u,v) \text{ dans } G} (u \rightarrow v).$$

On a :

- $tr$ est calculable en espace logarithmique ;
- $(G,s,t) \in \overline{\textsf{ACCESSIBILITE}_{acyclique}}$ iff $tr(G,s,t) \in$ 2SAT.

■

## 6.7 Problèmes $P$-complets

**Définition 12 ()**
Un problème **A** est $P$-dur si tout problème **B** dans $P$ se réduit en espace logarithmique à **A**.

**Définition 13 ()**
Un problème est P-complet s'il est dans P et est P-dur.

**Proposition 9** *S'il existe un problème* ***A*** *qui est* $P$*-dur et dans* $NL$*, alors* $P = NL$*.*

**Théorème 16** *Le problème suivant est P-complet ([Pap], p. 81 et 168) :*

***CIRCUIT VALUE***

- *entrée : Un circuit logique avec des portes logiques et, ou et non, avec des entrées mises à vrai, faux et avec une sortie*
- *sortie : Oui, si la sortie est à vraie ; non, sinon.*

DÉMONSTRATION.

dans P L'algorithme parcourt le graphe acyclique du circuit et évalue les sorties des portes une à une.

P-dur On code l'exécution de la machine à partir d'une entrée par un circuit.
■

Autre exemple : HORN-SAT est P-complet.

# References

[AH74] A.V. Aho and J.E. Hopcroft. *Design & Analysis of Computer Algorithms.* Pearson Education India, 1974.

[CS76] Ashok K Chandra and Larry J Stockmeyer. Alternation. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pages 98–108. IEEE, 1976.

[FL81] Aviezri S Fraenkel and David Lichtenstein. *Computing a perfect strategy for n× n chess requires time exponential in n.* Springer, 1981.

[KS08] Daniel Kroening and Ofer Strichman. *Decision procedures: an algorithmic point of view.* Springer Science & Business Media, 2008.

[Pap] Christos H. Papadimitriou. *Computational complexity.*

[Rob83] John Michael Robson. The complexity of go. In *IFIP Congress*, pages 413–417, 1983.

[Sip06] M. Sipser. *Introduction to the Theory of Computation*, volume 27. Thomson Course Technology Boston, MA, 2006.