Calculabilité

François Schwarzentruber

Préparation à l'option informatique de l'agrégation de mathématiques
ÉNS Rennes

Table des matières

Chapitre 1

Machines de Turing

Points du programme de l'agrégation
Définitions des machines de Turing. Équivalence entre classes de machines (exemples : nombre de rubans, alphabet).

1.1 Problèmes de décision

Définition 1 (problème de décision)
Un **problème de décision** est une fonction qui à une **instance** associe oui ou non.

Exemple 1

Connexe
entrée : un graphe fini non orienté G ;
sortie : oui si G est connexe ; non, sinon.

*Un graphe fini non orienté G est une **instance** de **Connexe**.*

Définition 2 (instance positive, instance négative)
- Une **instance positive** est une instance dont la réponse est oui.
- Un **instance négative** est une instance dont la réponse est non.

Exemple 2
- *Un graphe fini non orienté G connexe est une **instance positive** de **Connexe**.*
- *Un graphe fini non orienté G non connexe est une **instance négative** de **Connexe**.*

Hilbert rêvait d'un algorithme qui décide ce problème de décision :

RACINES ENTIERES
entrée : une équation diophantienne[1] [[Sipser, 2006], p. 183] ;
sortie : oui s'elle admet des solutions entières ; non sinon.

En 1970, Yuri Matijasevic et al. ont démontré qu'**il n'existe pas** de tel algorithme.

1.2 Thèse de Church-Turing

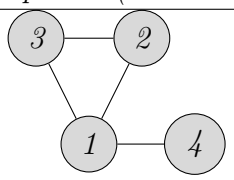
Une machine de Turing
modélise
un algorithme.

1. avec un nombre arbitraire d'inconnue et à coefficients entiers

1.3 Codage

instance	mot
ensemble des instances positives	langage

Exemple 3 ([Sipser, 2006], p. 186)

Graphe G (instance)	*Codage $\langle G \rangle$ (mot)*
	$(1,2,3,4)((1,2),(2,3),(3,1),(1,4))$

$L = \{\langle G \rangle \mid G \text{ est un graphe non orienté connexe}\}$.

1.4 Définition des machines de Turing

Définition 3 (machine de Turing non déterministe ([Sipser, 2006], p. 168))
Une **machine de Turing non-déterministe** est un tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc})$ où :
- Q est un ensemble fini non vide d'états ;
- Σ est l'alphabet fini des mots d'entrée tel que $\textvisiblespace \notin \Sigma$;
- Γ est l'alphabet fini du ruban avec $\Sigma \subseteq \Gamma$ et $\textvisiblespace \in \Gamma$;
- $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{\leftarrow, \rightarrow, \bullet\})$;
- $q_0 \in Q$ est l'état initial ;
- q_{acc} est l'état d'acceptation.

Définition 4 (machine de Turing déterministe)
Une machine de Turing $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc})$ est **déterministe** si δ est une fonction partielle

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow, \bullet\}.$$

Définition 5 (configuration)
Une **configuration** est un mot uqv où $u \in \Gamma^*$, $v \in \Gamma^\omega$ et $q \in Q$.

Définition 6 (configuration initiale)
Une **configuration initiale** est un mot $q_0 w \textvisiblespace\textvisiblespace \ldots$ où q_0 est l'état initial et $w \in \Sigma^*$.

Définition 7 (configuration acceptante)
Une configuration est **acceptante** si l'état de la configuration est q_{acc}.

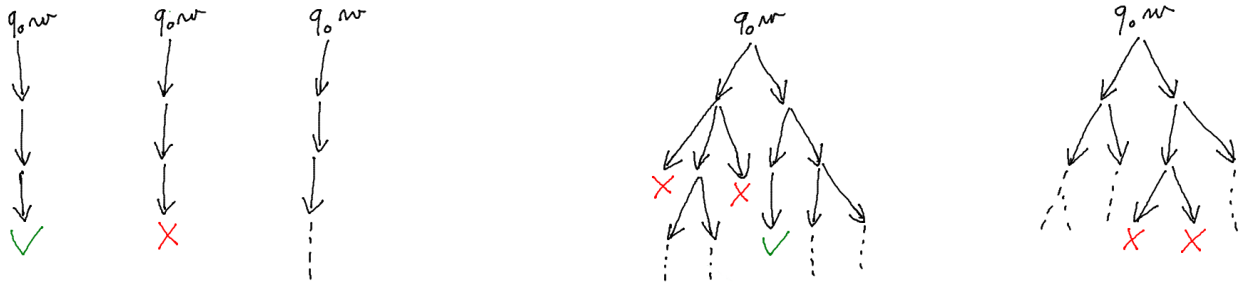
Définition 8 (un pas de calcul)
Pour tout $a, b, c \in \Sigma$, $u \in \Sigma^*, v \in \Sigma^\omega$,
- Si $(q, a, q', b, \leftarrow) \in \delta$ alors $ucqav$ devient $uq'cbv$;
- Si $(q, a, q', b, \bullet) \in \delta$, alors $uqav$ devient $uq'bv$;
- Si $(q, a, q', b, \rightarrow) \in \delta$, alors $uqav$ devient $ubq'v$;

Définition 9 (exécution)
Une **exécution** est une suite de configurations $C_1, \ldots, C_k$ telle que C_i devient C_{i+1}.

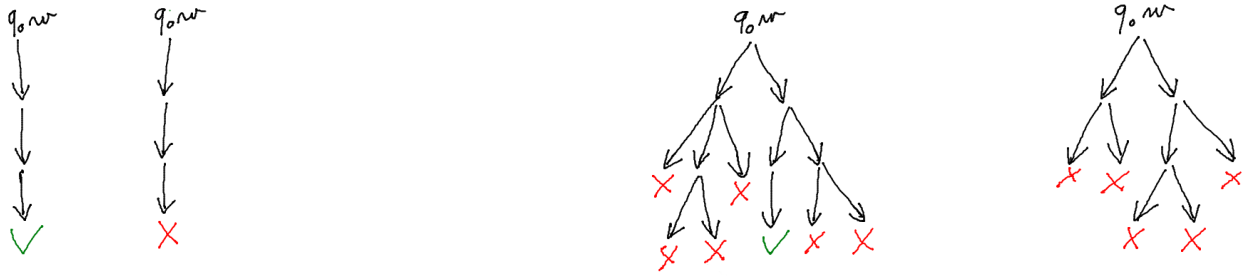
Définition 10 (exécution acceptante)
Une exécution est $C_1, \ldots, C_k$ **acceptante** si C_k est une configuration acceptante.

Définition 11 (acceptation d'un mot)
Une machine M **accepte** un mot w s'il existe une exécution acceptante $C_1, \ldots, C_k$
où C_1 est la configuration initiale $q_0 w \sqcup\sqcup \ldots$.

Définition 12 (langage accepté)
Le **langage accepté** par M, noté $L(M)$, est l'ensemble des mots w acceptés par M.



Définition 13 (décideur, langage décidé)
([Sipser, 2006], p. 170, p. 180) Une machine de Turing M qui s'arrête depuis $q_0 w$ pour tout mot w est appelé **décideur**. On dit que le langage accepté par M est **décidé** par M.



1.5 Equivalence entre machines de Turing

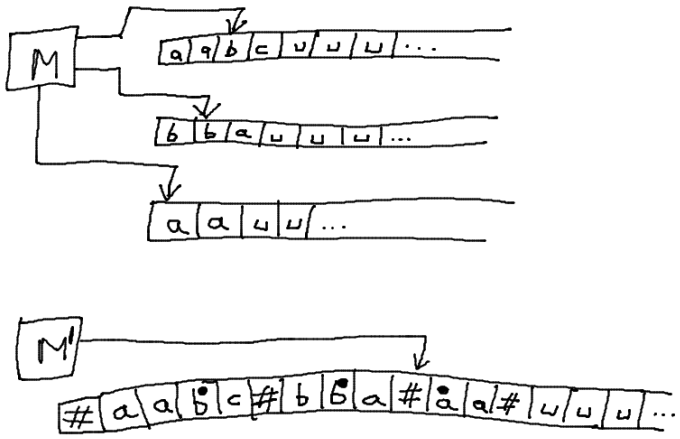
1.5.1 Plusieurs rubans

Définition 14 (machine de Turing déterministe à k rubans)
Une machine de Turing $(Q, \Sigma, \Gamma, \delta, q_0, acc)$ est déterministe si δ est une fonction partielle

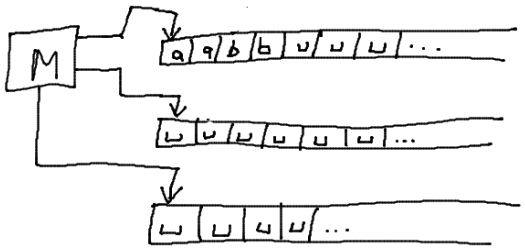
$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\leftarrow, \rightarrow, \bullet\}^k.$$

Théorème 1 *([Sipser, 2006], p. 177) Une machine de Turing déterministe à k rubans est équivalente à une machine de Turing déterministe à un ruban.*

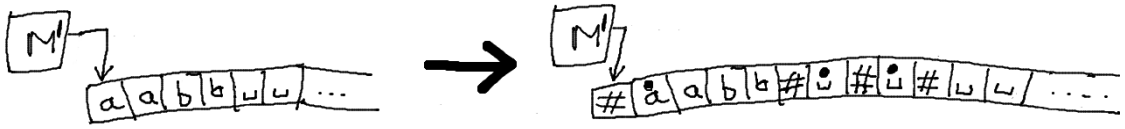
Idée de la Démonstration.
Soit M une machine de Turing déterministe à k rubans. On construit M' à un ruban qui simule M. L'idée est de concaténer les k rubans en les séparant par $\#$ et en marquant les cases sous le curseur avec $\bullet$.



La configuration initiale de M est :



Ainsi, la machine M' met d'abord son ruban dans le format qui représente les k rubans :



■

Corollaire 1 *Un langage est accepté par une machine de Turing déterministe à k rubans ssi il est accepté par une machine de Turing déterministe à un ruban.*

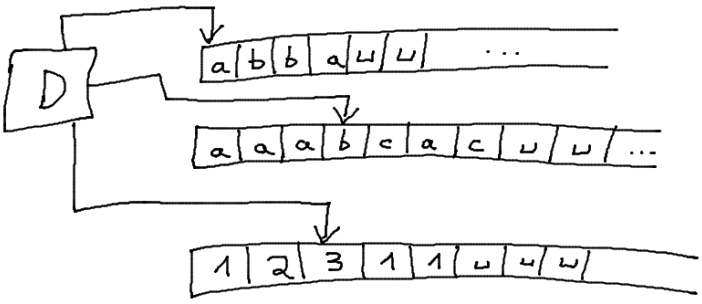
Corollaire 2 *Un langage est décidé par une machine de Turing (décideuse) déterministe à k rubans ssi il est décidé par une machine de Turing déterministe (décideuse) à un ruban.*

1.5.2 Déterministe vs non-déterministe

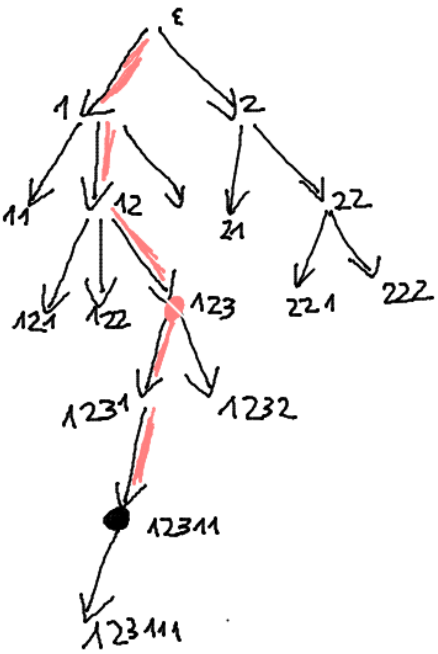
Théorème 2 *([Sipser, 2006], p. 179) Une machine de Turing non-déterministe est équivalente à une machine de Turing déterministe.*

Idée de la Démonstration.

Soit M une machine de Turing non-déterministe. L'idée est de construire une machine déterministe D qui simule un **parcours en largeur** de l'arbre de calcul de M. Quitte à la transformer en une machine à un ruban (via le théorème 1), on construit D à trois rubans :



1. Le premier ruban contient le mot d'entrée et est en lecture seule ;
2. Le deuxième ruban est le ruban de travail et représente la configuration temporaire de M ;
3. Le troisième ruban contient l'adresse de la configuration courante à calculer dans l'arbre de calcul de M. Son alphabet est $\{1, \ldots, b\}$ où b est le branchement dans l'arbre de calcul de M :



procédure $D(w)$
  pour $n := 0, 1, 2, \ldots$
    pour tout mot de longueur n sur $\{1, \ldots, b\}$
      Écrire u sur le 3^e ruban
      tant que pas à la fin du troisième ruban ou simulation impossible
        soit i l'entier sous le curseur du 3^e ruban
        simuler la i^{me} transition de M sur le deuxième ruban
        avancer le curseur du troisième ruban vers la droite
      Si la simulation de M est acceptante **accepter**
    Si pas de configuration courante de profondeur n **rejeter**

■

Corollaire 3 *Un langage est accepté par une machine de Turing non-déterministe ssi il est accepté par une machine de Turing déterministe.*

Corollaire 4 *Un langage est décidé par une machine de Turing (décideuse) non-déterministe ssi il est décidé par une machine de Turing déterministe (décideuse).*

Idée de la Démonstration.

$\boxed{\Rightarrow}$ Considérons un langage décidé par une machine de Turing M (décideuse) non-déterministe. L'arbre de calcul de M est à branchement fini et n'admet que des branches finies.

Lemme 1 (de König) *Un arbre à branchement fini qui n'a que des branches finis est fini.*

D'après le lemme de König, l'arbre de calcul de M est fini. Montrons que $D(w)$ où D est donné dans la démonstration du théorème 2 termine. Comme l'arbre est fini, il existe une n' une profondeur sans nœuds. Par l'absurde, supposons que $D(w)$ ne termine pas. Lorsque $n = n'$, on arrive donc dans **rejeter** . Contradiction. Donc $D(w)$ termine.

■

1.5.3 Autres variantes

— Ruban infini dans les deux sens ([Wolper, 2006], p. 110) ;
— Éliminer le surplace • ;
— Machines de Turing à deux états seulement ;
— Uniquement 2 lettres (en plus de ␣).

1.6 Notes bibliographiques

Dans [Papadimitriou,], il utilise un caractère spécial ▷ comme butoir pour le côté gauche du ruban. Les configurations sont des triplets (u, q, v) où $u, v \in \Sigma^*$ et $q \in Q$. Il confond alphabet d'entrée et alphabet de sortie.

Dans [Sipser, 2006], l'explication sur le ruban infini à droite est légère : on confond uqv et $uqv␣$.

Dans [Perifel, 2014], il donne directement la définition des machines à k rubans.

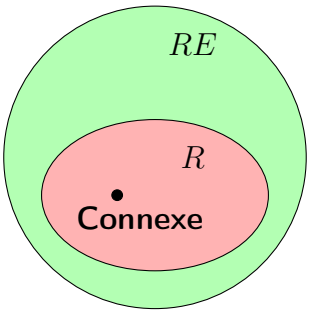
Chapitre 2

Indécidabilité

Points du programme de l'agrégation
Universalité. décidabilité, Indécidabilité. Théorème de l'arrêt. Théorème de Rice. Réduction de Turing. Définitions et caractérisations des ensembles récursifs, récursivement énumérables.

2.1 Classes R et RE

Définition 15 (décidable/récursif)
Un problème de décision **A** est décidable/récursif s'il existe une machine de Turing (décideuse) qui décide **A**.
On note R la classe des problèmes de décision décidables.

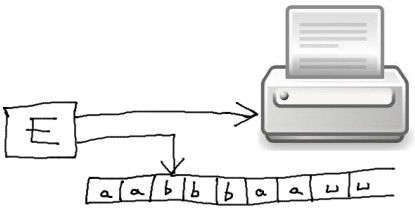


Exemple 4 ***Connexe*** *est décidable.* ***Connexe*** $\in R$.

Définition 16 (récursivement énumérable)
Un problème de décision **A** est récursivement énumérable s'il existe une machine de Turing qui accepte **A**.
On note RE la classe des problèmes de décision récursivement énumérables.

2.2 Enumérateurs

Définition 17 (énumérateur)
([Sipser, 2006], p. 18-181) Un **énumérateur** est une machine de Turing qui imprime/énumère des mots.



Théorème 3 *$L \in RE$ ssi il existe un énumérateur qui énumère des mots.*

Idée de la Démonstration.
$\boxed{\Leftarrow}$ Soit E un énumérateur de L. On construit une machine M qui accepte L :

procédure $M(w)$
  Boucle infinie
    Continuer l'exécution E
    Soit u le mot imprimé par E
    si $(u = w)$ **accepter**

■

$\boxed{\Rightarrow}$ Soit M une machine qui accepte L. On construit E qui énumère les mots de L :

procédure $E()$
  pour $n := 0, 1, 2, \ldots$
    pour tout mot w de longueur $\leq n$
      Exécuter n étapes de calcul de $M(w)$
      si $M(w)$ est acceptant **alors**
        imprimer w

2.3 Problème de l'arrêt

On s'intéresse au **problème de l'arrêt** :

Arrêt
entrée : une machine de Turing M, un mot w ;
sortie : oui si $M(w)$ s'arrête ; non, sinon.

2.3.1 ... est récursivement énumérable

Théorème 4 ***Arrêt*** *est récursivement énumérable.*

Idée de la Démonstration.

On donne une machine U qui accepte l'entrée M, w ssi M s'arrête sur w. La machine U est dite **universelle** car elle est capable de simuler l'exécution de n'importe quel machine de Turing M.

procédure $U(M, w)$
  simuler $M(w)$
  accepter

■

2.3.2 ... est indécidable

Théorème 5 ***Arrêt*** *est indécidable.*

Idée de la Démonstration.

Par l'absurde. Supposons que **Arrêt** est décidable. Il existe donc une machine de Turing A :
- qui s'arrête sur toute entrée M, w ;
- telle que A accepte M, w, ssi $M(w)$ termine.

procédure $A(M, w)$
  ...
  ...
  ...

On construit une machine *paradoxe* comme suit :

procédure *paradoxe*(M)
  si A accepte $(M, \langle M \rangle)$
    boucler ↻
  sinon
    accepter

$$\begin{gathered} \textit{paradoxe}(\textit{paradoxe}) \text{ ne termine pas} \\ \text{ssi} \\ A \text{ accepte } (\textit{paradoxe}, \langle \textit{paradoxe} \rangle) \\ \text{ssi} \\ \textit{paradoxe}(\textit{paradoxe}) \text{ termine.} \end{gathered}$$

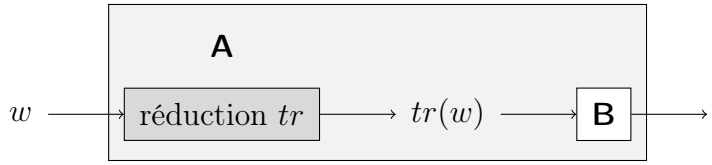
Contradiction.

■

2.4 Montrer l'indécidabilité par réduction

2.4.1 Réduction

Définition 18 (Réduction)
Une réduction d'un problème **A** à un problème **B** est une fonction tr calculable telle que pour toute instance w de **A**, w est instance positive de **A** ssi $tr(w)$ est instance positive de **B**.



On dit que **A** se réduit à **B** s'il existe une réduction de **A** à **B**.

Théorème 6 *Si **A** se réduit à **B** alors :*
— ***B** décidable implique **A** décidable.* *(le schéma donne un algorithme pour **A**)*
— ***A** indécidable implique **B** indécidable.* *(contraposée)*

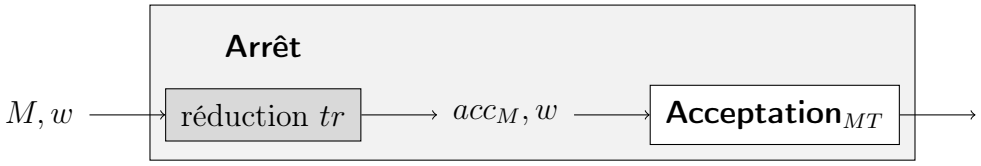
2.4.2 Exemple : acceptation par une machine de Turing

On s'intéresse au problème de l'acceptation d'un mot par une machine de Turing :

Acceptation$_{MT}$
entrée : une machine de Turing M, un mot w ;
sortie : oui si M accepte w ; non, sinon.

Théorème 7 ***Acceptation**$_{MT}$ est indécidable.*

Idée de la Démonstration.
On réduit **Arrêt** dans **Acceptation**$_{MT}$.



On pose $tr(M, w) = (acc_M, w)$
où acc_M est la machine construite effectivement à partir de M suivante :

```
procédure acc_M(w)
    M(w)
    accepter
```

tr est une réduction de **Arrêt** dans **Acceptation**$_{MT}$. En effet :
— tr est calculable ;
— M, w est une instance positive de **Arrêt**
ssi $M(w)$ s'arrête
ssi acc_M accepte w
ssi $tr(M, w)$ est une instance positive de **Acceptation**$_{MT}$.
Comme **Arrêt** est indécidable, **Acceptation**$_{MT}$ est indécidable. ■

2.5 Théorème de Rice

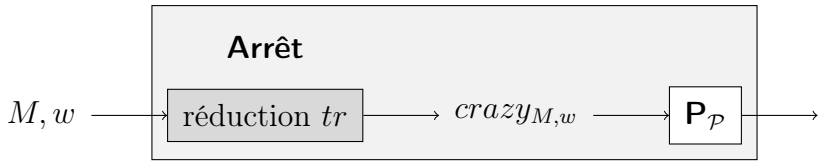
2.5.1 Enoncé

Théorème 8 (de Rice) *[Wolper, 2006] Soit $\mathcal{P}$ telle $\emptyset \subsetneq \mathcal{P} \subsetneq RE$. Le problème $\mathbf{P}_\mathcal{P}$ défini par*

$\mathbf{P}_\mathcal{P}$
entrée : une machine de Turing M ;
sortie : oui si $L(M) \in \mathcal{P}$; non, sinon.

est indécidable.

Idée de la Démonstration.
Sans perte de généralité, on suppose[1] que $\emptyset \notin \mathcal{P}$. Réduisons **Arrêt** à $\mathbf{P}_\mathcal{P}$.



Soit $\mathbf{G} \in \mathcal{P}$. Comme $\mathbf{G} \in RE$, il existe une machine G qui accepte $\mathbf{G}$. La réduction tr est définie par

$$tr(M, w) = crazy_{M,w}$$

où $crazy_{M,w}$ est la machine décrite à droite.

procédure $crazy_{M,w}(x)$
  Simuler $M(w)$
  si G accepte x
    accepter
  sinon
    rejeter

1. tr est une fonction calculable : on construit effectivement $crazy_{M,w}$ à partir de M et w ;
2. Comme $L(crazy_{M,w}) = \begin{cases} \mathbf{G} \text{ si } M \text{ s'arrête sur } w \\ \emptyset \text{ sinon} \end{cases}$, on a :

$$M, w \text{ est une instance positive de } \mathbf{Arrêt}$$
$$\text{ssi}$$
$$M \text{ s'arrête sur } w$$
$$\text{ssi}$$
$$L(crazy_{M,w}) \in \mathcal{P}$$
$$\text{ssi}$$
$$tr(M, w) = crazy_{M,w} \text{ est une instance positive de } \mathbf{P}_\mathcal{P}.$$

■

2.5.2 Exemples d'application

Exemple 5 *Avec $\mathcal{P} = \{\emptyset\}$, on a :*

Langagevide
entrée : une machine de Turing M
sortie : oui si $L(M) = \emptyset$; non, sinon.

est indécidable.

Exemple 6 *Avec $\mathcal{P} = \{L \mid \{\langle G \rangle \mid G \text{ est un graphe connexe}\} \subseteq L\}$, on a :*

TestSiGraphesConnexesAcceptes
entrée : une machine de Turing M
sortie : oui si $\{\langle G \rangle \mid G \text{ est un graphe connexe}\} \subseteq L(M)$; non, sinon.

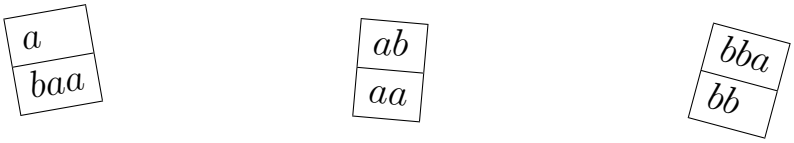
est indécidable.

1. Sinon prendre $RE \setminus \mathcal{P}$ à la place de $\mathcal{P}$ dans la suite de la démonstration.

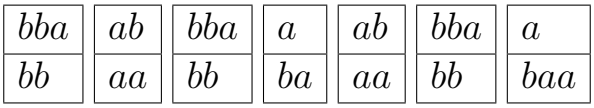
2.6 Problèmes de correspondance de Post

2.6.1 Définitions

Le problème de Post prend en entrée un système de tuiles comme



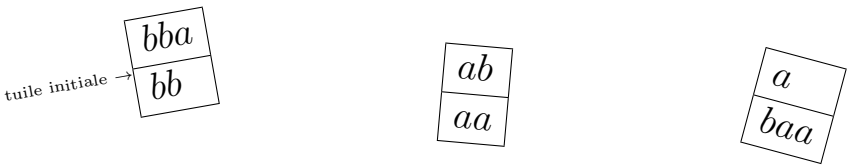
et répond oui si on peut mettre bout à bout des tuiles du système (on peut réutiliser plusieurs fois la même tuile) pour que les mots du haut et du bas soient identiques.



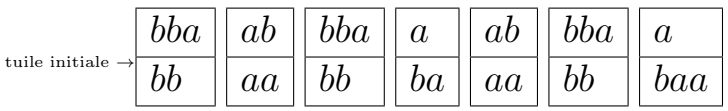
Définition 19 (Problème de correspondance de Post)

Post
entrée : un alphabet fini Σ, une famille finie de couples de mots $((h_i, b_i))_{i=1..n}$ sur Σ ;
sortie : oui s'il existe $i_1, \ldots, i_p \in \{1, \ldots, n\}$ tels que $p \geq 1$ et $h_{i_1} \ldots h_{i_p} = b_{i_1} \ldots b_{i_p}$; non, sinon.

Le problème de Post marqué est similaire mais il impose de commencer la suite de tuiles par la tuile initiale. Si l'entrée est :



et répond oui si on peut mettre bout à bout des tuiles du système (on peut réutiliser plusieurs fois la même tuile) pour que les mots du haut et du bas soient identiques en partant de la tuile initiale.



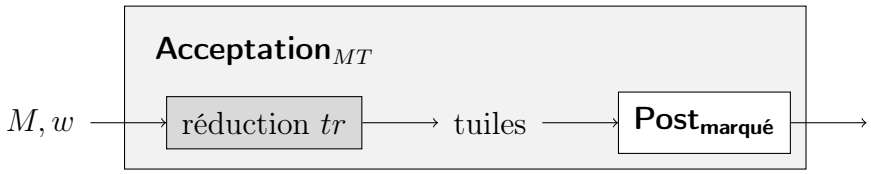
Définition 20 (Problème de correspondance de Post marqué)

Post$_{\text{marqué}}$
entrée : un alphabet fini Σ, une famille finie de couples de mots $((h_i, b_i))_{i=1..n}$ sur Σ ;
sortie : oui s'il existe $i_2, \ldots, i_p \in \{1, \ldots, n\}$ tels que $p \geq 1$ et $h_{\mathbf{1}} h_{i_2} \ldots h_{i_p} = b_{\mathbf{1}} b_{i_2} \ldots b_{i_p}$; non, sinon.

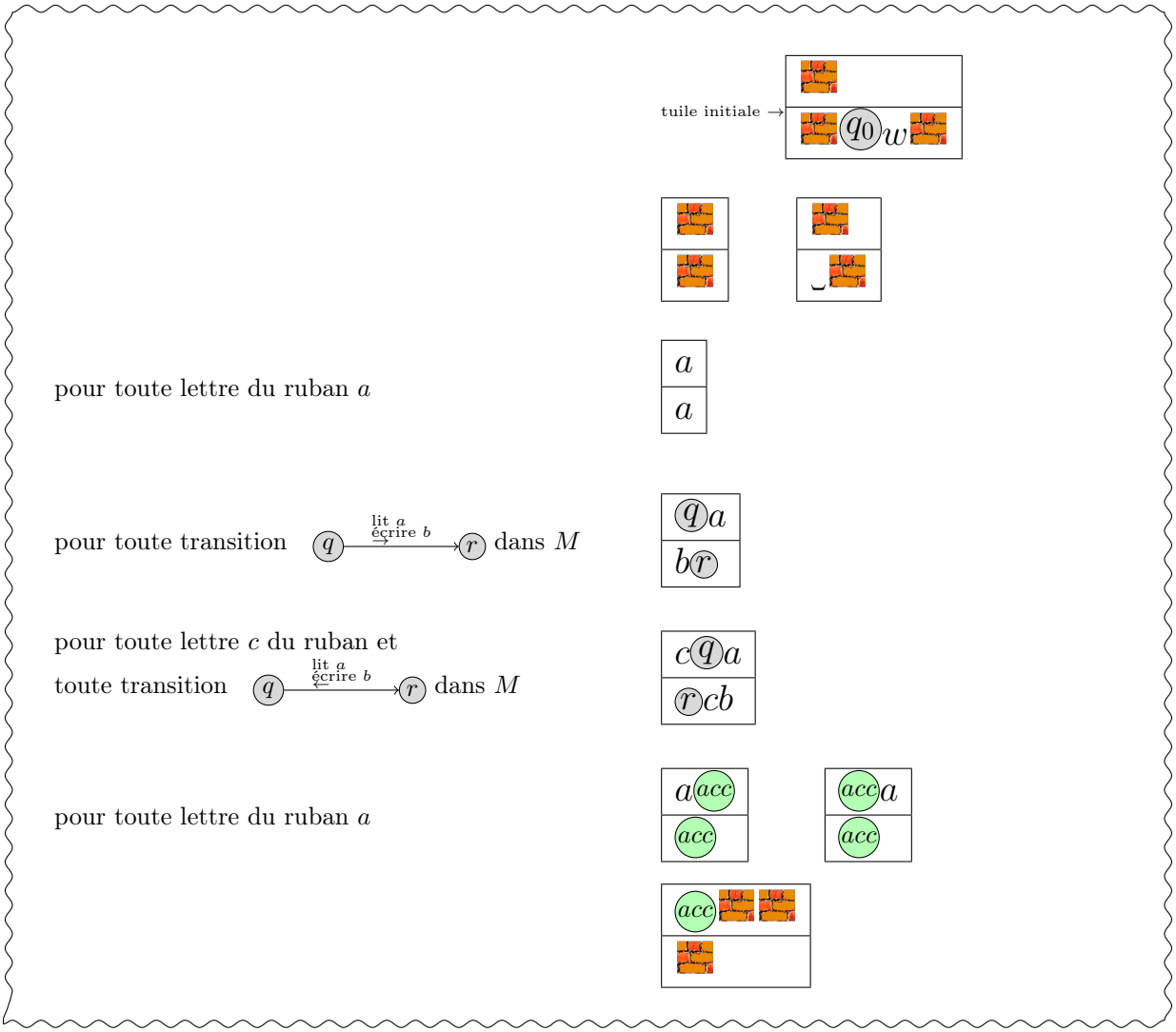
2.6.2 Démonstrations d'indécidabilité

Théorème 9 ***Post*$_{\text{marqué}}$** *est indécidable.*

IDÉE DE LA DÉMONSTRATION.
[Sipser, 2006]
Définissons une réduction tr de **Acceptation**$_{MT}$ dans **Post**$_{\text{marqué}}$.



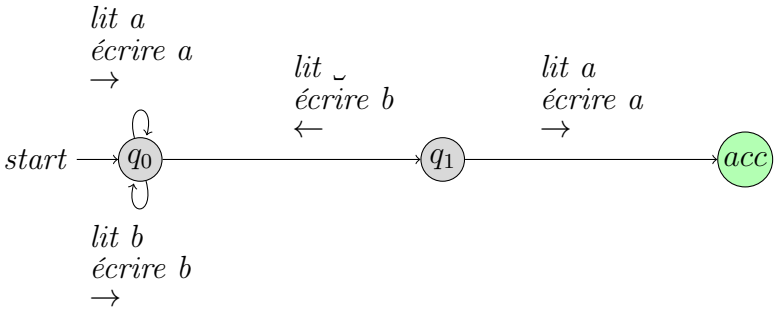
$tr(M, w)$ est le système de tuiles ci-dessous :



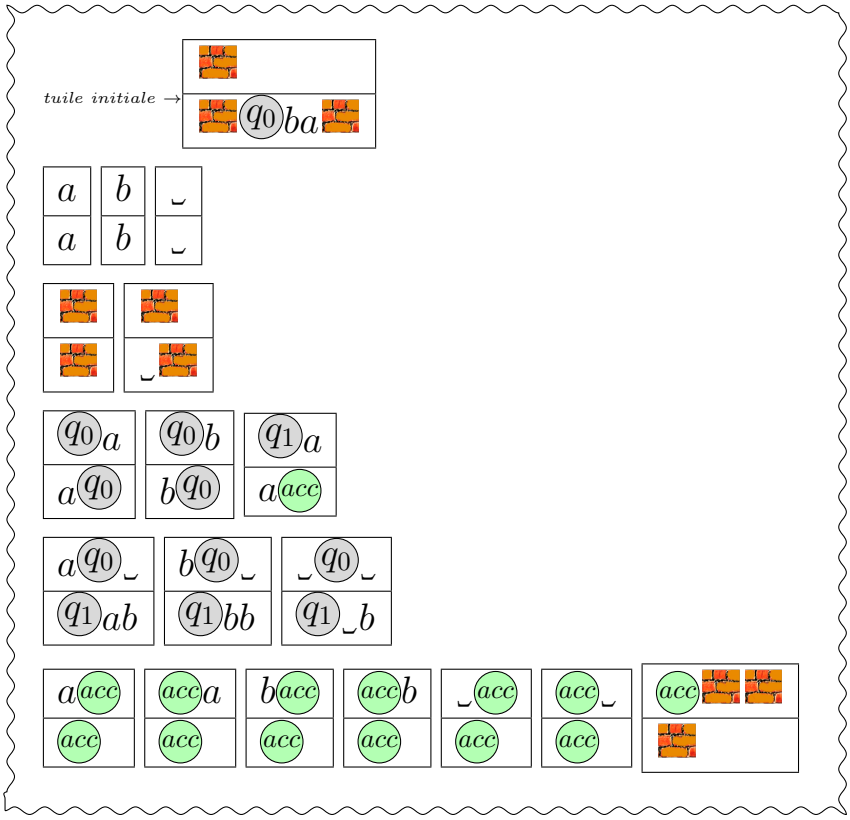
1. tr est une fonction calculable ;
2. M accepte w ssi $tr(M, w)$ est une instance positive de **Post**$_{\text{marqué}}$.

Comme **Acceptation**$_{MT}$ est indécidable, **Post**$_{\text{marqué}}$ est indécidable.
■

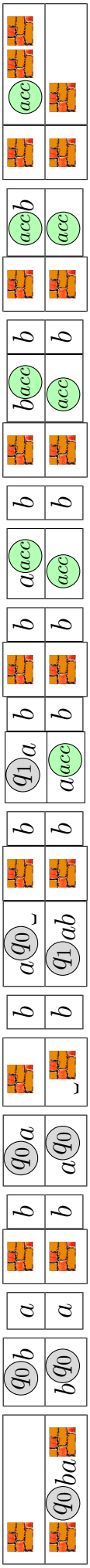
Exemple 7 *Considérons la machine de Turing M suivante qui accepte les mots qui finissent par un a :*



et le mot $w = ba$.

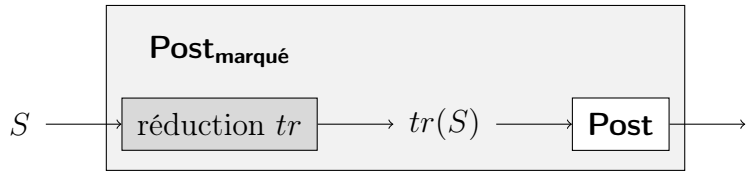


L'exécution acceptante de M sur aba est représentée par la suite de tuiles sur le bord droit de la page.



Théorème 10 ***Post*** *est indécidable.*

IDÉE DE LA DÉMONSTRATION.
[Sipser, 2006] Définissons une réduction de $\textbf{Post}_{\textbf{marqué}}$ dans **Post**.



Pour tout système de tuiles S, $tr(S)$ est définie comme :

Si la tuile initiale de S est $\begin{array}{|c|}\hline h_1h_2\ldots h_\ell \\ \hline b_1b_2\ldots b_k \\ \hline \end{array}$ alors $tr(S)$ contient $\begin{array}{|c|}\hline \star h_1\star h_2\ldots\star h_\ell \\ \hline \star b_1\star b_2\star\ldots\star b_k\star \\ \hline \end{array}$

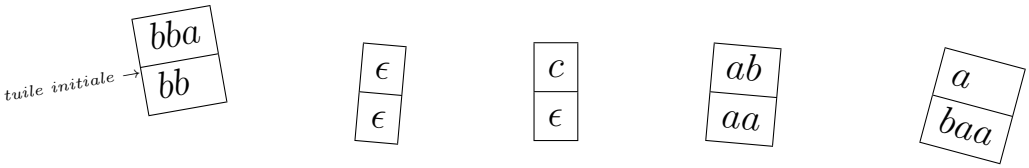
Si S contient $\begin{array}{|c|}\hline h_1h_2\ldots h_\ell \\ \hline b_1b_2\ldots b_k \\ \hline \end{array}$ alors $tr(S)$ contient $\begin{array}{|c|}\hline \star h_1\star h_2\ldots\star h_\ell \\ \hline b_1\star b_2\star\ldots\star b_k\star \\ \hline \end{array}$

$\begin{array}{|c|}\hline \star \\ \hline \\ \hline \end{array}$

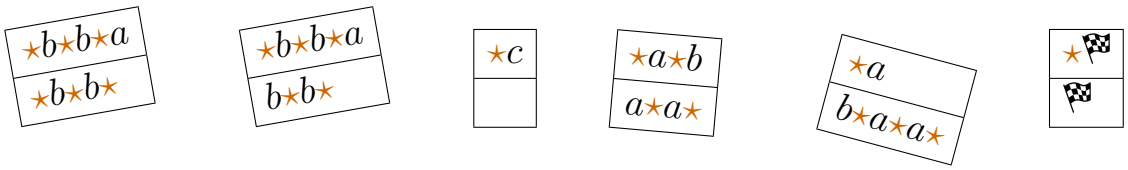
1. tr est une fonction calculable ;
2. S instance positive de $\textbf{Post}_{\textbf{marqué}}$ ssi $tr(S)$ instance positive de **Post**.

Comme $\textbf{Post}_{\textbf{marqué}}$ est indécidable, **Post** est indécidable.
■

Exemple 8 *L'instance de* ***Post***$_{\textbf{marqué}}$



est transformée en l'instance de ***Post*** *suivante :*



2.7 Bilan

Définition 21 (problème dual)
Soit **A** un problème de décision. Le **problème dual** de **A** est :

$\overline{\mathbf{A}}$
entrée : une instance w ;
sortie : oui si $w \notin \mathbf{A}$; non sinon.

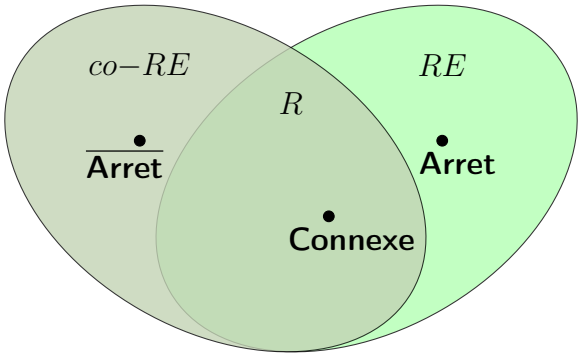
Définition 22 ($co-RE$)
$co-RE = \{\mathbf{A} \mid \overline{\mathbf{A}} \in RE\}$.

Théorème 11 $RE \cap co-RE = R$

Idée de la Démonstration.
Supposons $\mathbf{A} \in RE$ et $\overline{\mathbf{A}} \in RE$. Soit A et $\overline{A}$ des semi-algorithmes respectifs pour **A** et $\overline{\mathbf{A}}$. L'algorithme B défini ci-dessous décide **A** :

procédure $B(i)$
Lancer en parallèle $A(i)$ et $\overline{A}(i)$ et s'arrêter quand l'un des proccessus a accepté i
accepter si A a accepté i ; **rejeter** si $\overline{A}$ a accepté i.

■



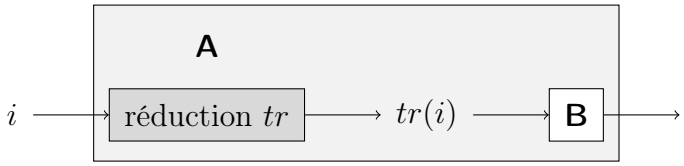
2.8 Notes bibliographiques

Problème de l'arrêt

Dans [Wolper, 2006], il introduit le langage LU, etc. La présentation est longue mais intéressante avant d'arriver... au problème de l'arrêt. J'ai préféré ici une démonstration plus directe. Dans [Sipser, 2006], le problème présenté dans la section "problème de l'arrêt" est le problème de l'acceptation. Le véritable problème de l'arrêt est présenté après.

Réduction

Je me suis inspiré de [Dasgupta et al., 2006] pour les schémas de réductions :



Chapitre 3

NP-complétude

Points du programme de l'agrégation
Complexité en temps et en espace : classe P. Machines de Turing non déterministes : classe NP. Acceptation par certificat. Réduction polynomiale. NP-complétude. Théorème de Cook.

3.1 Problèmes de décision dits "de recherche" [1]

3.1.1 Vocabulaire

Exemple 9

3-COLORATION

entrée : *Un graphe $G = (S, A)$ non orienté ;*
sortie : *oui s'il existe une fonction $c : S \to \{$, , $\}$ qui est un coloriage[2] de G ; non sinon.*

Exemple 10

SAT

entrée : *Une formule φ de la logique propositionnelle ;*
sortie : *oui si φ est satisfiable, c'est à dire il existe une valuation ν telle que $\nu \models \varphi$; non sinon.*

Problème de décision	Instance w	Certificat c	Propriété
3-COLORATION	Un graphe non orienté $G = (S, A)$	$c : S \to \{$, , $\}$	c est une coloration
SAT	Une formule φ de la logique propositionnelle	une valuation ν	$\nu \models \varphi$

Les **problèmes de décision de recherche** peuvent être mis sous la forme suivante :

A

entrée : une **instance** w ;
sortie : oui s'il existe un **certificat** c telle que $\mathcal{P}(w, c)$ est vraie ; non sinon.

où vérifier si $\mathcal{P}(w, c)$ s'effectue en temps polynomial en $|w|$.

1. terminologie de [Dasgupta et al., 2006]
2. c'est-à-dire une fonction c telle que pour tout $(s, t) \in A$, $c(s) \neq c(t)$.

3.1.2 D'un problème d'optimisation à un problème de décision

Exemple 11

SAC-A-DOS-OPTIMISATION

- *entrée :*
  - *Des valeurs* $(v_1, \ldots, v_n)$ *;*
  - *des poids* $(p_1, \ldots, p_n)$ *;*
  - *un poids maximal* P *;*
- *sortie : la valeur maximale de* $\sum_{i=1}^{n} x_i v_i$ *telle que* $\sum_{i=1}^{n} x_i p_i \leq P$ *avec* $(x_1, \ldots, x_n) \in \{0,1\}^n$.

⇕

SAC-A-DOS

entrée :
- *Des valeurs* $(v_1, \ldots, v_n)$ *;*
- *des poids* $(p_1, \ldots, p_n)$ *;*
- *un poids maximal* P *;*
- *une valeur seuil* V *;*

sortie : oui s'il existe $(x_1, \ldots, x_n) \in \{0,1\}^n$ *telle que* $\sum_{i=1}^{n} x_i v_i \geq V$ *et* $\sum_{i=1}^{n} x_i p_i \leq P$.

On obtient un algorithme pour **SAC-A-DOS-OPTIMISATION** à partir d'un algorithme S pour **SAC-A-DOS** : dichotomie sur V et appel à S.

3.2 Classe P

Définition 23 (classe P)
La classe **P** est la classe des problèmes de décision **A** tel qu'il existe une machine de Turing déterministe A telle que
— $L(A) = \mathbf{A}$;
— et il existe un polynôme f telle que pour toute instance w,
l'exécution $A(w)$ est de longueur au plus $f(|w|)$.

Thèse Cobham–Edmonds [Arora and Barak, 2009] : problème facile = problème dans **P**

Techniques algorithmiques pour montrer que certains problèmes de recherche sont dans **P**

Algorithmes sur les graphes.
— Satisfiabilité d'un ensemble de 2-clauses en logique propositionnelle [Dasgupta et al., 2006] ;

Gloutons.
— Satisfiabilité d'un ensemble de clauses de Horn en logique propositionnelle [Dasgupta et al., 2006] ;
— Arbre couvrant minimum [Dasgupta et al., 2006]

Flots.
— Couplage maximal ;
— Elimination d'une équipe au baseball.

Programmation dynamique.
— Plus longue sous-suite commune [Cormen, 2009] ;
— Appartement d'un mot au langage engendré par une grammaire algébrique.

Programmation linéaire (réelle).
— Maximiser la vente de chocolats.

3.3 EXPTIME

Définition 24 (classe EXPTIME)
La classe **EXPTIME** est la classe des problèmes de décision **A** tel qu'il existe une machine de Turing déterministe A telle que
— $L(A) = \mathbf{A}$;
— et il existe un polynôme f telle que pour toute instance w,
l'exécution $A(w)$ est de longueur au plus $2^{f(|w|)}$.

Proposition 1 *3-COLORIAGE est dans EXPTIME.*

Idée de la Démonstration.
Voici un algorithme qui décide **3-COLORIAGE** en temps exponentiel :

procédure 3col(G)
  pour tout $c : S \to \{$  , ,  $\}$
    si c est un 3-coloriage **alors**
      accepter
  rejeter

■

Mais la classe **EXPTIME** semble trop grosse pour caractériser nos problèmes de recherche.

3.4 Classe NP

3.4.1 Intuition : jeu à un joueur

G est une instance positive de **3-COLORATION** ssi le jeu suivant admet une stratégie gagnante :

procédure 3-coloration(G)
  pour $s \in S$
    choisir $c[s]$ dans {🔴, 🟢, 🟡} ;
  si c est un 3-coloriage **alors**
    accepter (gagné)
  sinon
    rejeter (perdu)

3.4.2 Définition de la classe NP

Jeu à un joueur	Machine non-déterministe
Partie du jeu	Exécution
Partie du jeu gagnante	Exécution acceptante
Stratégie	Choix des transitions durant une exécution = certificat
Stratégie gagnante	Choix des transitions qui donne une exécution acceptante
Existence d'une stratégie gagnante pour w	w est accepté

Définition 25 (classe NP)
La classe **NP** est la classe des problèmes de décision **A** tel qu'il existe une machine de Turing non-déterministe A telle que
- $L(A) = \mathbf{A}$;
- et il existe un polynôme f telle que pour toute instance w, toutes les exécutions de $A(w)$ sont de longueur au plus $f(|w|)$.

3.4.3 Définition alternative : vérifieur par certificat

Définition 26 (vérifieur)
([Sipser, 2006], p. 243) Un **vérifieur** pour un problème de décision **A** est une machine de Turing déterministe V tel que w est une instance positive de **A** ssi il existe c tel que $V(w, c)$ est une exécution acceptante.

Proposition 2 (classe NP, définition alternative) *([Sipser, 2006], p. 244)*

A $\in$ ***NP*** *ssi* *il existe un vérifieur V pour **A** et un polynôme f tels que pour toute instance w, pour tout certificat c, la longueur de l'exécution $V(w, c)$ est $\leq f(|w|)$.*

Idée de la Démonstration.

⇐

procédure $M(w)$
  Choisir c de longueur $f(|w|)$
  si $V(w, c)$ accepte
  alors
    accepter
  sinon
    rejeter

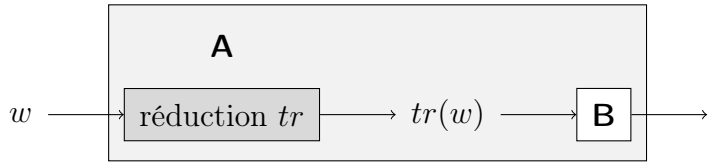
⇒

procédure $V(w, c)$
  Simuler l'exécution $M(w)$ en prenant les choix non-déterministe conseillé par c
  si l'exécute accepte
  alors
    accepter
  sinon
    rejeter

■

3.4.4 Réductions polynomiales

Définition 27 (Réduction)
Une **réduction polynomiale** d'un problème **A** à un problème **B** est une fonction tr calculable en temps polynomial telle que pour toute instance w de **A**, i est instance positive **A** ssi $tr(w)$ est instance positive **B**.



On dit que **A** se réduit en temps polynomial à **B** s'il existe une réduction polynomiale de **A** à **B**. Intuitivement, si **A** se réduit en temps polynomial **B**, alors **B** est plus dur que **A**.

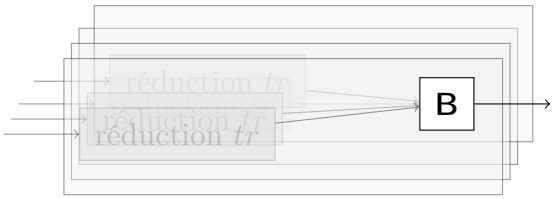
Théorème 12 *Si **A** se réduit en temps polynomial **B**, alors :*

1. $\boldsymbol{B} \in P$ *implique* $\boldsymbol{A} \in P$ *;*
2. $\boldsymbol{B} \in NP$ *implique* $\boldsymbol{A} \in NP$*.*

Idée de la Démonstration.
Le schéma ci-dessus donne un algorithme en temps polynomial pour **A**. ■

3.4.5 NP-dureté

Définition 28 (NP-dur)
Un problème **B** est **NP-dur** si pour tout problème **A** $\in$ **NP**, **A** se réduit en temps polynomial à **B**.



Définition 29 (NP-complet)
Un problème est **NP-complet** s'il est dans **NP** et **NP**-dur.

3.4.6 Le problème ouvert P $\stackrel{?}{=}$ NP

Proposition 3 *Si un problème **NP**-dur est dans **P** alors **P** = **NP**.*

Idée de la Démonstration.
Par le théorème 12, 1. ■

3.5 Mon premier problème NP-complet : SAT

CNF-SAT
entrée : Une formule φ de la logique propositionnelle en forme normale conjonctive ;
sortie : oui si φ est satisfiable ; non sinon.

Théorème 13 (de Cook) *SAT et CNF-SAT sont NP-complets.*

Proposition 4 *Si* $\mathbf{SAT} \in \mathbf{P}$ *alors* $\mathbf{P} = \mathbf{NP}$.

3.5.1 Dans NP

Théorème 14 *SAT est dans NP.*

Idée de la Démonstration.

Voici un algorithme non-déterministe qui décide **SAT** en temps polynomial.

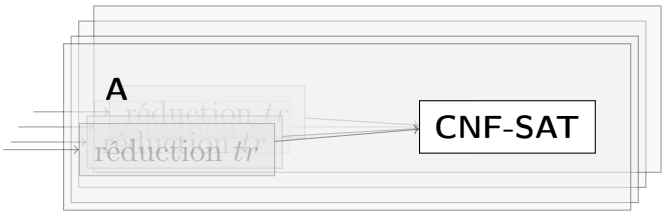
procédure sat(φ)
  pour toute proposition atomique p dans φ
    choisir $\nu[p]$ dans $\{faux, vrai\}$;
  si $\nu \models \varphi$ **alors**
    accepter (gagné)
  sinon
    rejeter (perdu)

■

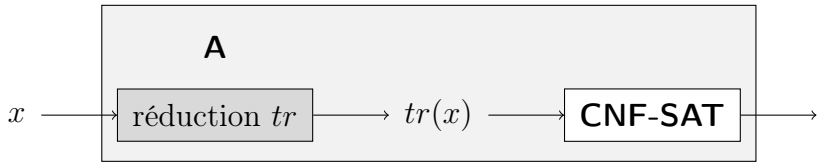
3.5.2 NP-dur

Théorème 15 *CNF-SAT est NP-dur.*

Idée de la Démonstration.
On montre que tout problème dans **NP** s'y réduit en temps polynomial. Soit $\mathbf{A} \in \mathbf{NP}$.



On va donner une réduction polynomiale tr tel que x est une instance positive de **A** ssi $tr(x)$ est une formule satisfiable.



Soit $M = (\Sigma, \Gamma, Q, \delta, q_0, acc)$ une machine de Turing non-déterministe qui décide **A** en temps polynomial. Il existe un polynôme f tel que, sur une entrée x, toutes les exécutions de la machine de Turing sont de longueur au plus $f(|x|)$. Sans perte de généralité, on suppose que l'état final acceptant acc est un état qui boucle.

On définit la réduction tr transforme une **A**-instance x en une formule $tr(x)$ qui exprime

"il existe une exécution acceptante de M sur le mot x en temps $f(|x|)$"

Soit $\mathcal{C} = \{0, \ldots, f(|x|)\}$. Pour écrire $tr(x)$, on introduit les propositions atomiques suivantes :

au temps t, l'état est q | au temps t, la position du curseur est i | au temps t, la case n° i contient a | au temps t, on tire la transition τ

où $t, i \in \mathcal{C}$, $q \in Q$, $a \in \Gamma$ et $\tau \in \delta$.

Notons que seules les $f(|x|)$ cases du ruban sont pertinentes car la tête de lecture n'a pas le temps d'aller au delà de la $f(|x|)^{\text{ème}}$ case.
Définissons à présent $tr(x)$ comme la conjonction des formules suivantes.

Unicité et existence des valeurs.

1. $\bigwedge_{t\in\mathcal{C}} \bigvee_{q\in Q}$ au temps t, l'état est q — La machine est dans un état à tout instant t

2. $\bigwedge_{t\in\mathcal{C}} \bigwedge_{q,q'\in Q|q\neq q'} \Big(\neg$ au temps t, l'état est q $\vee \neg$ au temps t, l'état est $q'\Big)$ — La machine n'est jamais dans deux états à la fois

3. $\bigwedge_{t\in\mathcal{C}} \bigvee_{i\in\mathcal{C}}$ au temps t, la position du curseur est i — Le curseur est positionné quelque part à tout instant t

4. $\bigwedge_{t\in\mathcal{C}} \bigwedge_{i,i'\in\mathcal{C}|i\neq i'} \Big(\neg$ au temps t, la position du curseur est i $\vee \neg$ au temps t, la position du curseur est $i'\Big)$ — Le curseur n'a jamais deux positions différentes

5. $\bigwedge_{t\in\mathcal{C}} \bigwedge_{i\in\mathcal{C}} \bigvee_{a\in\Sigma}$ au temps t, la case n° i contient a — À tout instant, toute case du ruban contient une lettre

6. $\bigwedge_{t\in\mathcal{C}} \bigwedge_{a,b\in\Sigma|a\neq b} \Big(\neg$ au temps t, la case n° i contient a $\vee \neg$ au temps t, la case n° i contient $b\Big)$ — Une case ne contient au plus qu'une lettre

7. $\bigwedge_{t\in\mathcal{C}} \bigvee_{\tau\in\delta}$ au temps t, on tire la transition τ — 'A tout instant t, on tire une transition pour aller vers l'instant $t+1$

8. $\bigwedge_{\tau,\tau'\in\delta|\tau\neq\tau'} \Big(\neg$ au temps t, on tire la transition τ $\vee \neg$ au temps t, on tire la transition $\tau'\Big)$ — On ne tire jamais plus d'une transition

Configuration initiale

9. au temps 0, la case n° 0 contient $\sqcup$ $\wedge$ au temps 0, la case n° 1 contient x_1 $\wedge \ldots \wedge$ au temps 0, la case n° n contient x_n $\wedge$ au temps 0, la case n° $n+1$ contient $\sqcup$ $\wedge \ldots \wedge$ au temps 0, la case n° $f(|x|)$ contient $\sqcup$ — À l'instant 0, le ruban contient $\sqcup x_1 \ldots x_n \sqcup \ldots \sqcup$

10. au temps 0, l'état est q_0 $\wedge$ au temps 0, la position du curseur est 1 — À l'instant 0, la machine est dans l'état initial q_0 à l'instant 0 et le curseur est à la position 1.

Exécution acceptante

11. $\bigvee_{t\in\mathcal{C}}$ au temps t, l'état est acc — La machine atteint l'état d'acceptation acc

Exécution des transitions

12. $\bigwedge_{t\in\mathcal{C}}\bigwedge_{i\in\mathcal{C}}\bigwedge_{a\in\Sigma}\Big(\big(\neg$ au temps t, la position du curseur est i $\wedge$ au temps t, la case n° i contient $a\big)\rightarrow$ au temps $t+1$, la case n° i contient $a\Big)$ — on ne change pas le contenu du ruban si le curseur n'y est pas

13. $\bigwedge_{t\in\mathcal{C}\setminus\{f(|x|)\}}\ \bigwedge_{(q,a,e,b,d)\in\delta}\Big($au temps t, on tire la transition $q,a,e,b,d\rightarrow$ au temps t, l'état est $q\Big)$

14. $\bigwedge_{t\in\mathcal{C}\setminus\{f(|x|)\}}\ \bigwedge_{(q,a,e,b,d)\in\delta}\ \bigwedge_{i\in\mathcal{C}}\Big[\big($au temps t, on tire la transition $q,a,e,b,d\big)\wedge$ au temps t, la position du curseur est $i\big)\rightarrow$ au temps t, la case n° i contient $a\Big]$

15. $\bigwedge_{t\in\mathcal{C}\setminus\{f(|x|)\}}\ \bigwedge_{(q,a,e,b,d)\in\delta}\Big($au temps t, on tire la transition $(q,a,e,b,d)\rightarrow$ au temps $t+1$, l'état est $e\Big)$

16. $\bigwedge_{t\in\mathcal{C}\setminus\{f(|x|)\}}\ \bigwedge_{(q,a,e,b,d)\in\delta}\ \bigwedge_{i\in\mathcal{C}}\Big[\big($au temps t, on tire la transition $(q,a,e,b,d)\big)\wedge$ au temps t, la position du curseur est $i\big)\rightarrow$ au temps $t+1$, la case n° i contient $b\Big]$

17. $\bigwedge_{t\in\mathcal{C}\setminus\{f(|x|)\}}\ \bigwedge_{(q,a,e,b,d)\in\delta}\ \bigwedge_{i\in\mathcal{C}|i+d\in\mathcal{C}}\Big($au temps t, on tire la transition $(q,a,e,b,d)\big)\wedge$ au temps t, la position du curseur est $i\big)\rightarrow$ au temps $t+1$, la position du curseur est $i+d$

La formule $tr(x)$ est bien en forme normale conjonctive. Aussi, on peut la calculer en temps polynomial en $|x|$ à partir de x . On a bien x est une instance positive de **A** ssi $tr(x)$ est une formule satisfiable.

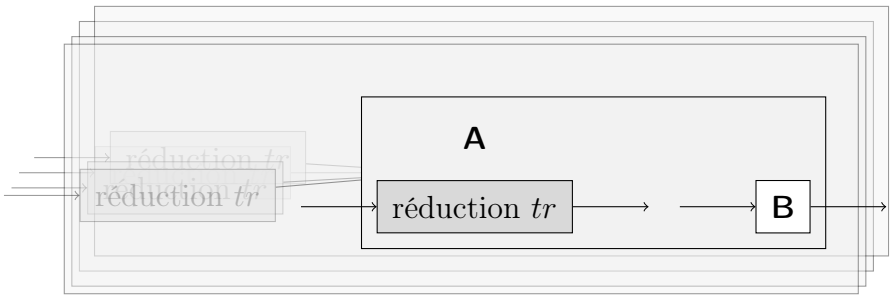
$\boxed{\Rightarrow}$ Soit x une instance positive de **A**. Alors il existe une exécution de M acceptante sur le mot x en temps $f(|x|)$ et sur une longueur de ruban de $f(|x|)$. On construit une valuation qui satisfait $tr(x)$.

$\boxed{\Leftarrow}$ Si $tr(x)$ est satisfiable. Soit ν une valuation qui satisfait $tr(x)$. On construit à partir de ν une exécution de M acceptante sur le mot x. Donc le mot x est une instance positive de **A**.
■

3.6 Réductions polynomiales pour montrer qu'un problème est NP-dur

Proposition 5 *Si* **A** *se réduit polynomialement à* **B** *et que* **A** *est NP-dur alors* **B** *est NP-dur.*

Idée de la Démonstration.



■

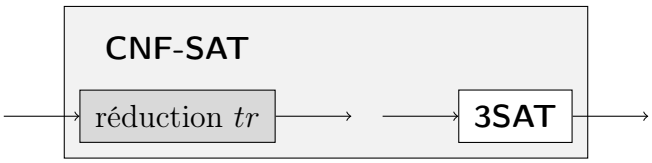
3.6.1 3-SAT

3-SAT
entrée : Une formule φ de la logique propositionnelle en 3-forme normale conjonctive ;
sortie : oui si φ est satisfiable ; non sinon.

Proposition 6 ***3-SAT*** *est* ***NP****-dur.*

Idée de la Démonstration.

Nous allons réduire polynomialement **CNF-SAT** à **3SAT**.



Si φ est une forme normale conjonctive, $tr(\varphi)$ est obtenue à partir de φ en remplaçant chaque clause par un ensemble de 3-clauses en introduisant des variables supplémentaires.

Exemple 12 $(a \vee b \vee c \vee d \vee e) \quad \rightsquigarrow \quad (a \vee b \vee \alpha) \wedge (\neg\alpha \vee c \vee \beta) \wedge (\neg\beta \vee d \vee e)$.

Montrons que φ est satisfiable ssi $tr(\varphi)$ satisfiable.

$\boxed{\Leftarrow}$ Supposons que φ est vraie pour une certaine valuation. En particulier, chaque clause $(a \vee b \vee c \vee d \vee e)$ est vraie. Montrons que l'on peut donner des valeurs aux variables intermédiaires de sorte que $(a \vee b \vee \alpha) \wedge (\neg\alpha \vee c \vee \beta) \wedge (\neg\beta \vee d \vee e)$ soit vraies. L'une des variables a, b, c, d, e est à vraie. Par exemple, c est à vraie. Il suffit de mettre les premières variables intermédiaires à vraies jusqu'à être dans la 3-clause où apparaît c. Ici : α à vraie. Puis on met les variables intermédiaires suivantes à faux.

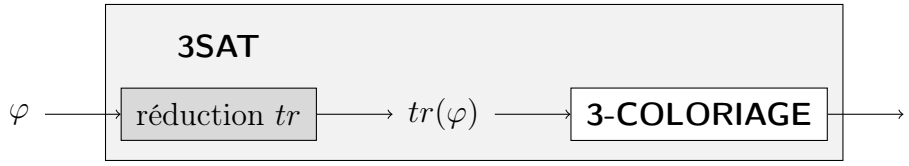
$\boxed{\Rightarrow}$ Supposons que $tr(\varphi)$ soit satisfiable pour une certaine valuation. En particulier, chaque sous-conjunction de clauses $(a \vee b \vee \alpha) \wedge (\neg\alpha \vee c \vee \beta) \wedge (\neg\beta \vee d \vee e)$, obtenu à partir d'une clause de l'instance de SAT est vraie. Montrons que l'une des variables a, b, c, d, e est mise à vraie. Par l'absurde, supposons que toutes les variables a, b, c, d, e sont à faux. On a alors $\alpha, \beta, \neg\beta$ à vraie. Contradiction.

■

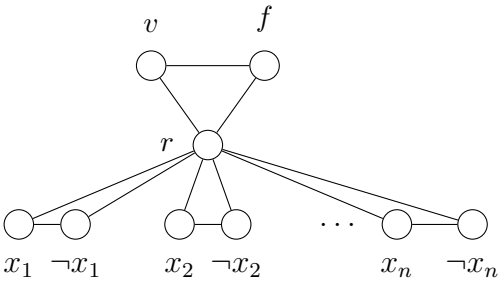
3.6.2 3-coloration

Théorème 16 ***3-COLORIAGE*** *est NP-dur.*

IDÉE DE LA DÉMONSTRATION.



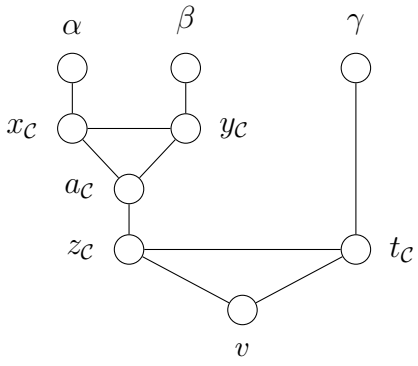
On définit une réduction tr qui à toute **3SAT**-instance φ associe une **3-COLORIAGE**-instance $tr(\varphi)$. $tr(\varphi)$ est un graphe basé sur le **gadget** suivant :



où $x_1, x_2, \ldots, x_n$ sont les propositions atomiques qui apparaissent dans φ. On code le fait qu'un littéral est vrai par le fait qu'il a la couleur du sommet v et faux s'il a la couleur du sommet f. La couleur de $\neg x$ est toujours différente de celle de x.

Ensuite, pour chaque 3-clause $(\alpha \vee \beta \vee \gamma)$ de φ, $tr(\varphi)$ contient aussi le gadget à ci-dessous. Pour coder une 2-clause $\alpha \vee \beta$, on prend le même gadget mais avec f à la place de γ.

Lemme 2 *Dans tout coloriage, l'un des sommets α, β, γ a avoir la couleur de v.*



IDÉE DE LA DÉMONSTRATION.
Par l'absurde, supposons qu'ils ont tous la couleur de f (on rappelle qu'ils ne peuvent pas avoir la couleur de r car la structure initiale ne le permet pas). Dans ce cas, comme γ est de la couleur de f, pour sûr t_C est de la couleur d'r donc z_C est de la couleur de f. D'autre part, comme α et β sont de la couleur de f, pour sûr, x_C et y_C ne sont pas de couleur de f donc a_C est de couleur de f. Contradiction. ■

Lemme 3 *On peut compléter tout coloriage où les sommets α, β, γ sont de la couleur de f ou v et l'un d'eux est de la couleur de v.*

IDÉE DE LA DÉMONSTRATION.
Faire tous les cas. ■

Lemme 4 *φ satisfiable ssi $tr(\varphi)$ est 3-coloriable.*

IDÉE DE LA DÉMONSTRATION.
$\boxed{\Rightarrow}$ Supposons que φ est satisfiable et soit ν une valuation telle que $\nu \models \varphi$. On colorie les noeuds de la façon suivante :
— $c[r] =$; $c[v\,] =$; $c[f\,] =$;
— $c[p] =$ si $\nu[p] = 1$; sinon.
— $c[\neg p] =$ si $\nu[p] = 1$; sinon.
Par le lemme 3, on complète le coloriage pour tous les gadgets. Donc $tr(\varphi)$ est 3-coloriable.
$\boxed{\Leftarrow}$ Supposons que $tr(\varphi)$ est 3-coloriable. On construit la valuation ν :
— $\nu[p_i] = 1$ si p_i est colorié avec la couleur de v : 0 sinon.
Par le lemme 2, $\nu \models \varphi$. ■ ■

3.7 Exemples de problèmes

3.7.1 NP-complets

Voyageur de commerce	Faire des trous dans une taule en déplaçant le moins possible le forêt
Arbres de Steiner	Connexion de circuits
k-coloration de graphes si $k \geq 3$	Fréquences radio

21 problèmes de Karp : [Karp, 1972], [Dasgupta et al., 2006] Autre référence : [Garey and Johnson, 1979]

3.7.2 Encore non classés

ISOMORPHISME DE GRAPHES
entrée : Deux graphes G_1, G_2
sortie : oui si G_1 et G_2 sont isomorphes ; non, sinon.

FACTORISATION D'ENTIERS
entrée : un entier n, un entier $1 < m < n$
sortie : oui s'il existe un facteur $d \in \{1, \ldots, m\}$ de n ; non, sinon.

3.7.3 Démonstration de l'appartenance à P récente

Attention, contrairement à ce qui est dit dans [Garey and Johnson, 1979][p. 154], la primalité [Agrawal et al., 2004] et la programmation linéaire ([Khachiyan, 1980], [Wright, 2005]) ont été montrés dans **P**.

3.8 NP-complétude en pratique

3.8.1 Branch and bound

Exemple du voyageur de commerce : lire [Dasgupta et al., 2006].

3.8.2 Algorithmes d'approximation

Lire le chapitre correspondant dans [Dasgupta et al., 2006] ou la référence [Vazirani, 2013]. On a des fois des schémas d'approximation en temps polynomial. Par exemple, pour le voyageur de commerce, il existe un algorithme *vdcapprox* tel que $vdcapprox(G, \epsilon)$ retourne un voyage de longueur $\geq \ell_{opt} + \epsilon$ où ℓ_{opt} est la longueur optimale, en temps polynomial en G et ϵ.

3.8.3 Réduction à SAT ou à la programmation linéaire entière

Exemple 13 *logimage, sudoku, etc.*

- Compétitions SAT : `http://www.satcompetition.org/`
- Outil maison pour SAT : `http://people.irisa.fr/Francois.Schwarzentruber/dpll_demo/`
- Outil pour la programmation linéaire entière : `https://www.gnu.org/software/glpk/`

Chapitre 4

Classes de complexité

4.1 Définition des classes de complexité

4.1.1 Classes non stables par modèle de calcul

Définition 30 ($TIME(f), NTIME(f)$)
[Sipser, 2006, p. 229] Soit $f : \mathbb{N} \to \mathbb{R}^+$.
- $TIME(f) = \{L \mid L \text{ est décidé par une MT dét en temps } O(f(n))\}$;
- $NTIME(f) = \{L \mid L \text{ est décidé par une MT non dét en temps } O(f(n))\}$.

Définition 31 ($SPACE(f), NSPACE(f)$)
[Sipser, 2006, p. 308] Soit $f : \mathbb{N} \to \mathbb{R}^+$ telle que $f(n) \geq n$.
- $SPACE(f) = \{L \mid L \text{ est décidé par une MT dét en espace } O(f(n))\}$;
- $NSPACE(f) = \{L \mid L \text{ est décidé par une MT non dét en espace } O(f(n))\}$.

4.1.2 Classes stables par modèle de calcul

Définition 32 ($P, NP, EXPTIME, NEXPTIME, PSPACE$,**etc.**)

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

Stables par nombre de rubans, etc.

Définition 33 (co-☁)
co-☁ $= \{L \subseteq \Sigma^* \mid \overline{L} \in$ ☁$\}$.

Définition 34 (☁**-dur**)
L est ☁-dur ssi tout problème dans ☁ se réduit à L en temps polynomial.

Définition 35 (☁**-complet**)
L est ☁-complet ssi L est dans ☁ et est ☁-dur.

4.2 Théorème de Savitch

Théorème 17 (de Savitch) *[Sipser, 2006, p. 310] Soit $f : \mathbb{N} \to \mathbb{R}^+$ telle que $f(n) \geq n$ [1] et $f(n)$ calculable en espace $O(f(n))$. $NSPACE(f) \subseteq SPACE(f^2)$.*

Idée de la 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 :

procédure deciderL(x)
  si c_{accept} **accessible** depuis $c_{ini}(x)$ dans le **graphe des configurations** de M
    accepter
  sinon
    rejeter

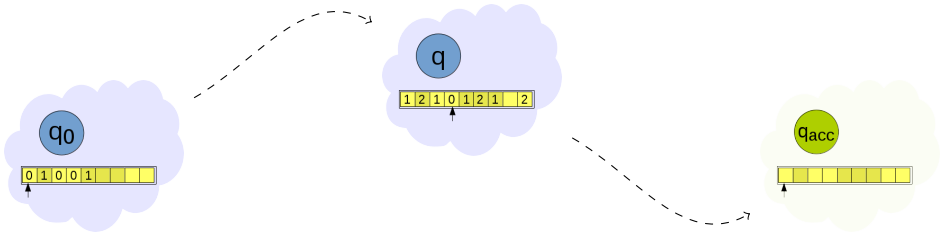
Dans le graphe des configuration de M, seules les configurations M de taille de ruban $f(|x|)$ au plus sont accessibles depuis $c_{ini}(x)$. Il y en a au plus

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

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



fonction chemin ?(c_1, c_2, t)
  si $t = 1$
    retourner vrai si $c_1 = c_2$ ou $c_1 \to^M c_2$; non, sinon
  sinon
    pour c configuration de M de taille de ruban au plus $f(n)$
      si *chemin*?$(c_1, c, \frac{t}{2})$ et *chemin*?$(c, c_2, \frac{t}{2})$ **alors**
        retourner vrai
    retourner 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 5 *PSPACE = NPSPACE.*

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

4.3 PSPACE

4.3.1 QBF : formules booléennes quantifiées

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

SAT
entrée : une formule propositionnelle φ
sortie : oui si φ est satisfiable ; non sinon.

est **NP**-complet.

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

VALIDE
entrée : une formule propositionnelle φ
sortie : oui si φ est valide ; non sinon.

est co-**NP**-complet.

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

Syntaxe

Définition 36 (formule booléenne quantifiée)
Une **formule booléenne quantifiée** (sous forme prénexe) est une formule de la forme

$$Q_1 p_1 \dots Q_n p_n \chi$$

où $Q_k \in \{\exists, \forall\}$ et χ est une formule de la logique propositionnelle. Une formule booléenne quantifiée est **close** si toutes les variables sont sous la portée d'un quantificateur.

Sémantique

Définition 37 (conditions de vérité)
- $\nu \models \chi$ comme en logique propositionnelle si χ 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 φ est close, $\nu \models \varphi$ ne dépend pas de ν et on dira que φ est vraie s'il existe ν telle que $\nu \models \varphi$.

Problème de décision

TQBF
entrée : une formule booléenne quantifiée close φ
sortie : oui si φ est vraie ; non sinon.

Dans PSPACE

Théorème 18 *[Sipser, 2006](p. 285-287)* ***TQBF*** *est dans PSPACE.*

Idée de la Démonstration.
Voici un algorithme qui vérifie que $\nu \models \varphi$ en espace polynomial en $|\nu|$ et $|\varphi|$:

fonction $tqbf(\nu, \varphi)$
  match φ
    $\exists p\psi$: $tqbf(\nu[p := 0], \psi)$ ou $tqbf(\nu[p := 1], \psi)$
    $\forall p\psi$: $tqbf(\nu[p := 0], \psi)$ et $tqbf(\nu[p := 1], \psi)$
    ψ propositionnelle : oui si $\nu \models \psi$; non sinon.

■

PSPACE-dur

Théorème 19 ***TQBF*** *est NPSPACE-dur.*

IDÉE DE LA 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...

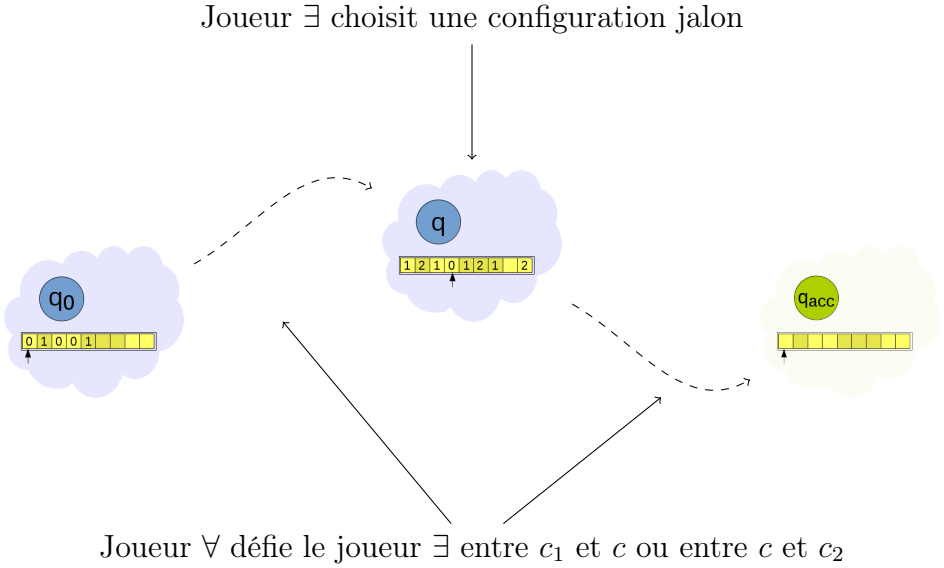
Pour toute instance x de L, on crée une formule booléenne quantifiée $tr(x)$ qui exprime 'il existe un chemin de $c_{ini}(x)$ à c_{accept} de longueur au plus $2^{df(|x|)}$' :

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

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 respectivement les configurations c_1 et c_2.

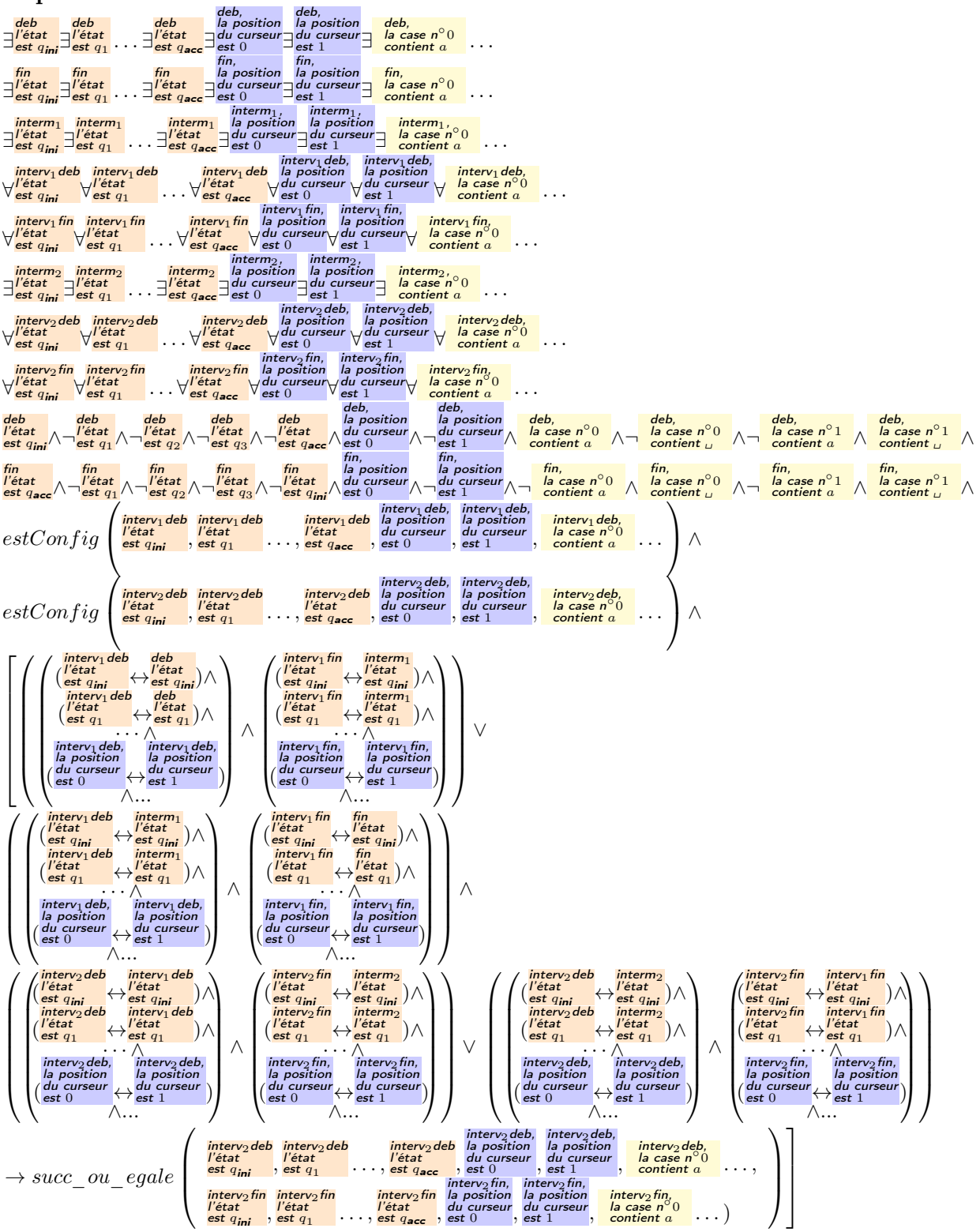
$ch?(\vec{c}_1, \vec{c}_2, 2^k)$ est définie par induction sur k :
— $ch?(\vec{c}_1, \vec{c}_2, 2^0) = (\vec{c}_1 = \vec{c}_2) \vee succ(\vec{c}_1, \vec{c}_2)$;
— Si $k > 0$,

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



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 14

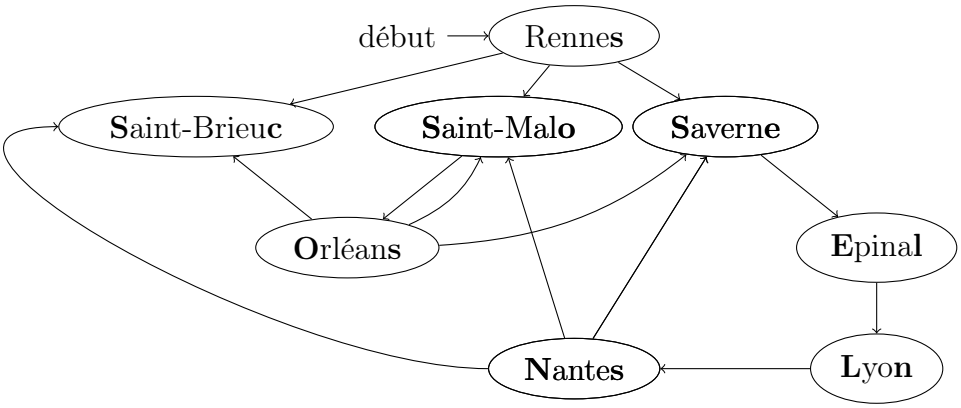


4.3.2 Jeux à deux joueurs

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 ([Kroening and Strichman, 2008], chap. 9, p. 209).

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 ([Sipser, 2006], 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.

GEOGRAPHIE est PSPACE-complet

Proposition 7 ***GEOGRAPHIE*** *est dans PSPACE.*

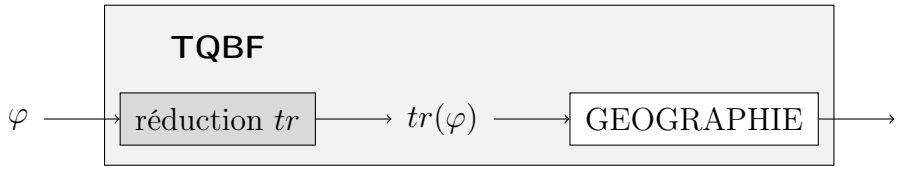
Idée de la Démonstration.

```
fonction joueurgagne(G, s)
    pour t successeur de s dans G
        si joueurgagne(G \ {t}, t) = faux
            retourner vrai
    retourner faux
```

■

Proposition 8 ***GEOGRAPHIE*** *est PSPACE-dur.*

Idée de la Démonstration.



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

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

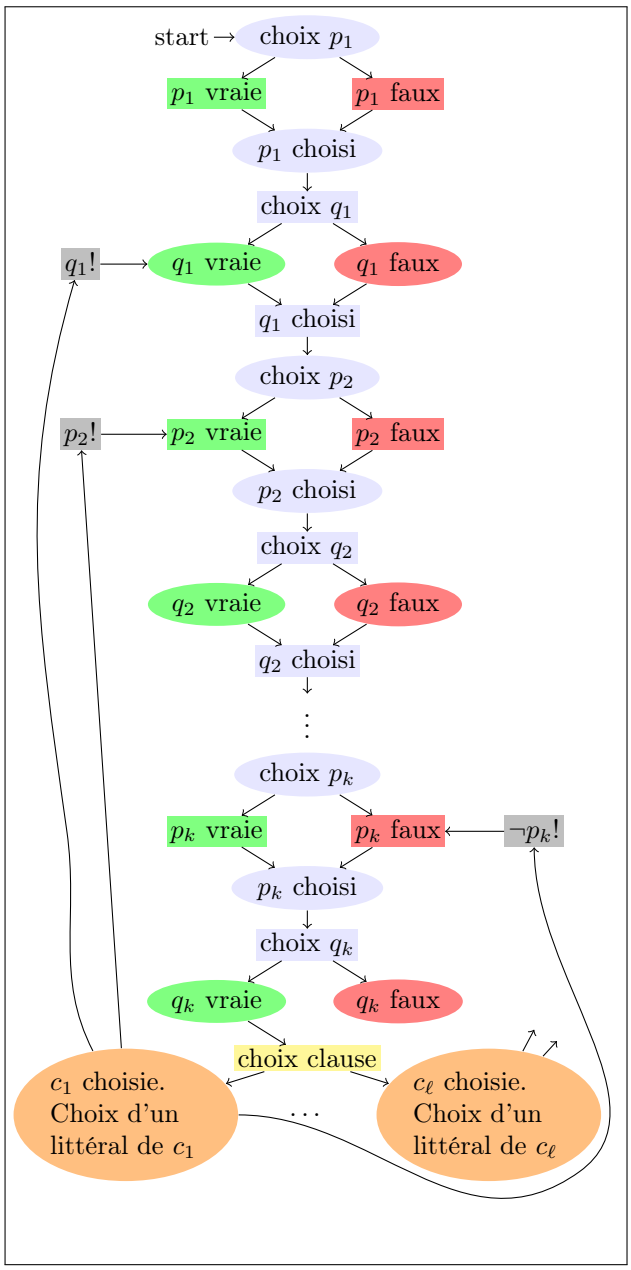
où ψ est une forme normale conjonctive, c'est à dire

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

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



■

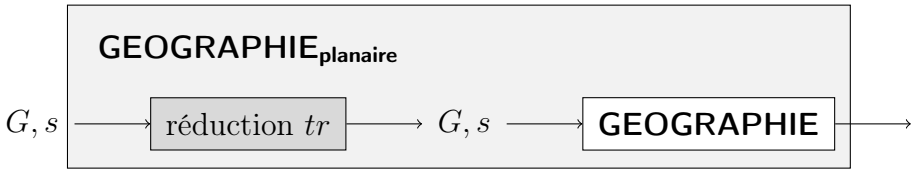
Jeu de géographie planaire

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

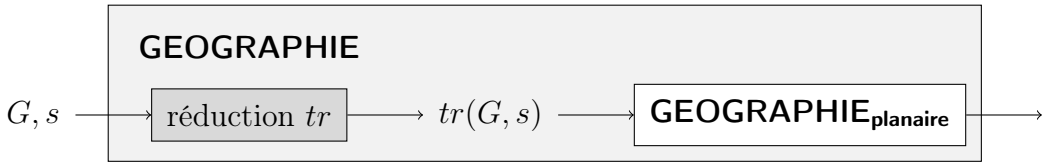
Idée de la Démonstration.



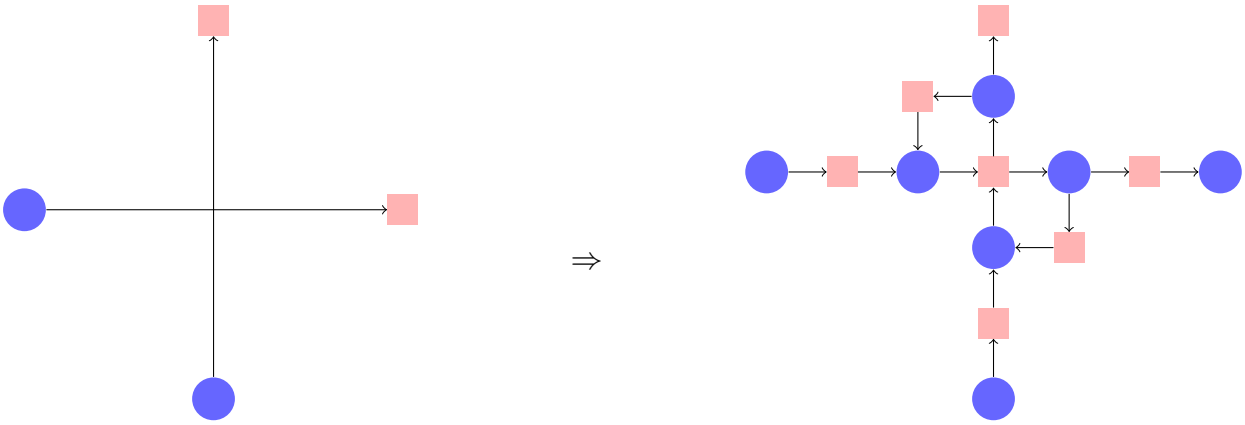
■

Proposition 10 ***GEOGRAPHIE*$_{\textbf{planaire}}$** *est PSPACE-dur.*

IDÉE DE LA DÉMONSTRATION.



Idée : se débarasser des croisements.



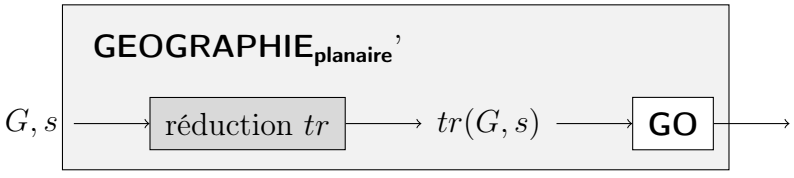
■

Jeu de go

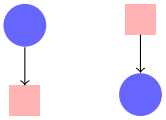
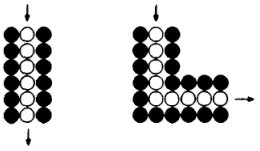
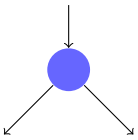
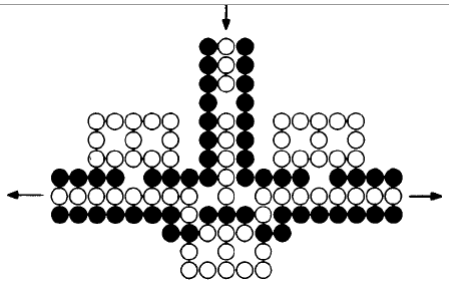
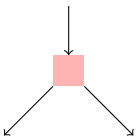
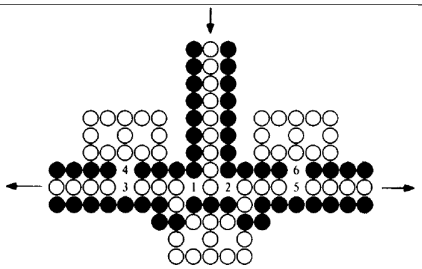
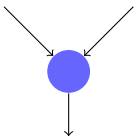
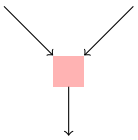
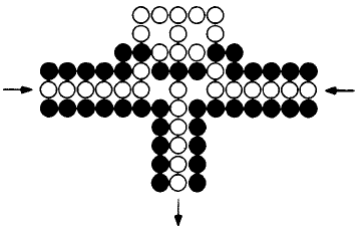
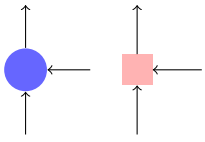
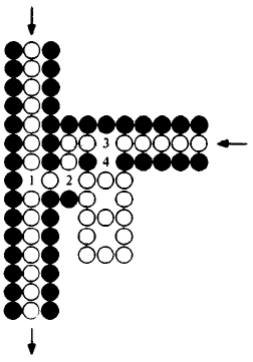
GO
entrée : un plateau de Go ;
sortie : oui si le joueur 1 (noir) a une stratégie gagnante à partir de G, s ; non, sinon.

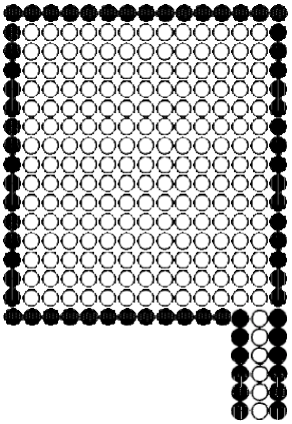
Théorème 20 ***GO*** *est PSPACE-dur.*

IDÉE DE LA DÉMONSTRATION.



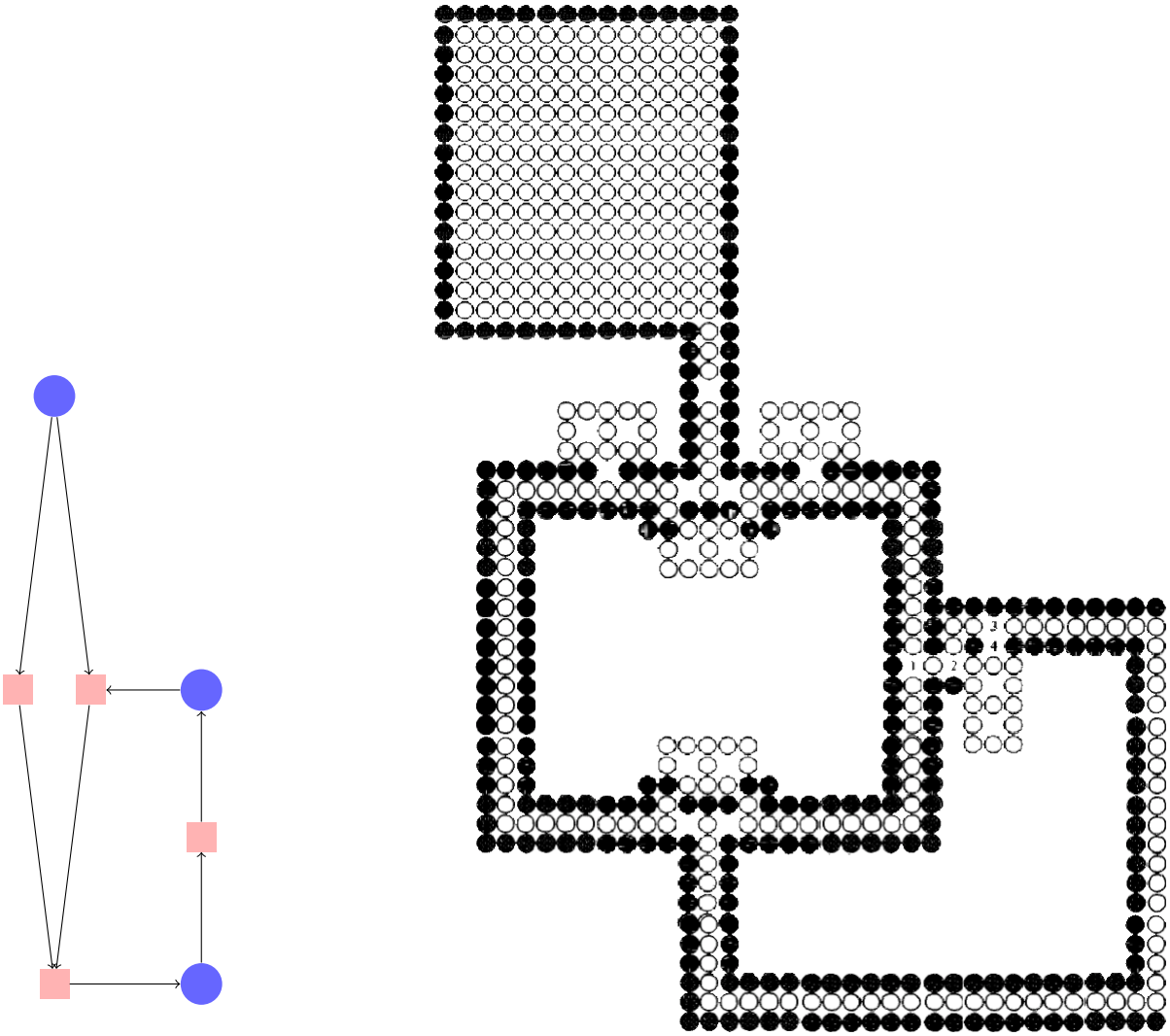
$tr(G, s)$ est le plateau de Go suivant :

Motif dans G	Portion de plateau de Go correspondante
	
	
	
 	
	



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

Exemple 15



■

4.4 Langages rationnels

Réf : [Aho and Hopcroft, 1974][p. 395, section 10.6]

4.4.1 Langage vide

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

Théorème 21 ***VIDE EXPRESSION RAT*** *est dans* ***P****.*

Idée de la 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 ϵ-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**. ■

4.4.2 Langage universel

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

Théorème 22 ***UNIVERSALITE EXPRESSION RAT*** *est dans PSPACE.*

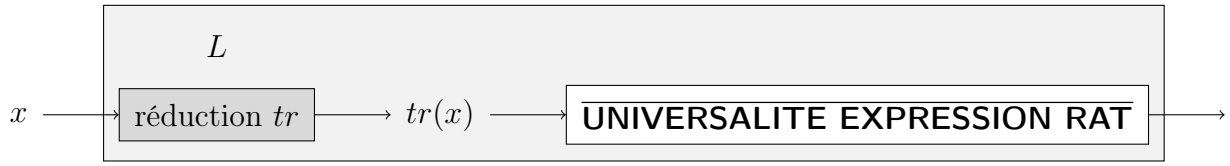
Idée de la Démonstration.
On construit un algorithme qui requiert un espace polynomial en $O(|e|)$:

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

■

Théorème 23 ***UNIVERSALITE EXPRESSION RAT*** *est PSPACE-dur.*

IDÉE DE LA DÉMONSTRATION.
Soit L un problème dans PSPACE. On réduit L à **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

$$\square w_1 \square w_2 \dots w_k \square$$

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

Exemple 16

$$\square \begin{bmatrix} a \\ q_0 \end{bmatrix} b \sqcup\sqcup \square b \begin{bmatrix} b \\ q \end{bmatrix} \sqcup\sqcup \square ba \begin{bmatrix} \sqcup \\ q' \end{bmatrix} \sqcup \square \dots$$

Σ	alphabet contenant l'alphabet du ruban, les couples $\begin{bmatrix} \text{lettre du ruban} \\ \text{état} \end{bmatrix}$ et $\square$
Δ	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 :

Δ^*	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 $x_i = \textvisiblespace$ si $i > \lvert x\rvert$	i^e case du ruban non conforme à la configuration initiale
$(\Sigma \setminus A)^*$	Pas d'états acceptants
$\Sigma^* c_1c_2c_3\Sigma^{N-1}(\Sigma \setminus f(c_1c_2c_3))\Sigma^*$	Transition non conforme

où f sont des fonctions qui correspondent aux transitions :

c_1	c_2	c_3
	$f(c_1c_2c_3)$	

Exemple 17 *Par exemple :*

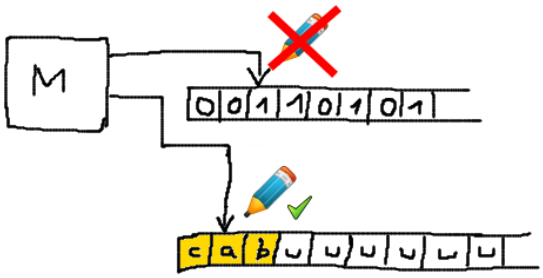
- $f(, \left[\begin{smallmatrix} a \\ q_0 \end{smallmatrix}\right], b) = c$ *si la transition depuis l'état* q_0 *en lisant* a *écrit* c *;*
- $f(\left[\begin{smallmatrix} a \\ q_0 \end{smallmatrix}\right], b, d) = \left[\begin{smallmatrix} b \end{smallmatrix}\right] q$ *si la transition depuis* q_0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

■

4.5 LOGSPACE et NLOGSPACE

4.5.1 Définitions

Ici, on utilise le modèle de machine de Turing à deux rubans : le ruban d'entrée en lecture seule et un ruban de travail. On ne comptabilise que la mémoire utilisée sur le ruban de travail.



Définition 38 (L et NL)

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

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

4.5.2 Accessibilité

ACCESSIBILITE
entrée : un graphe orienté G, deux sommets s et t ;
sortie : oui, s'il existe un chemin de s vers t dans G ; non, sinon.

Proposition 11 *([Papadimitriou,], example 2.10 p. 48-49)* ***ACCESSIBILITE*** *est dans* ***NL****.*

Idée de la Démonstration.
Voici un algorithme non-déterministe pour résoudre **ACCESSIBILITE** :

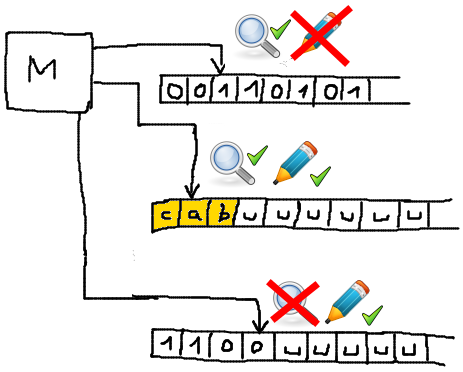
fonction $path?(G, s, t)$
  $w = s$
  tant que $w \neq t$
    $w :=$ **choisir** un successeur de w
  accepter

■

4.5.3 NL-complétude

Définition 39 (réduction en espace logarithmique)
Un problème A **se réduit** à un autre B **en espace logarithmique** si il existe une fonction $tr : \Sigma^* \to \Sigma^*$ telle que :

- $x \in A$ ssi $f(x) \in B$.
- tr calculable en espace logarithmique, i.e. il existe une machine de Turing à trois rubans :
  - le ruban d'entrée en lecture seule contenant x ;
  - un ruban de travail en lecture/écriture qui contient au plus $O(\log |x|)$ symboles ;
  - un ruban de sortie en écriture seule sur lequel est $tr(x)$ à la fin de l'exécution.

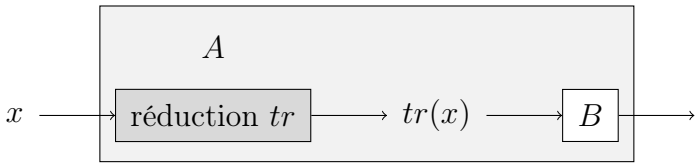


Définition 40 (NL-complet)
A est **NL**-complet ssi dans $A \in$ **NL** et tout problème de **NL** se réduit en espace logarithmique se réduit à A.

Théorème 24

1. *Si A se réduit à B en espace logarithmique et $B \in \boldsymbol{L}$ alors $A \in \boldsymbol{L}$;*
2. *Si A se réduit à B en espace logarithmique et $B \in \boldsymbol{NL}$ alors $A \in \boldsymbol{NL}$;*
3. *Si A se réduit à B en espace logarithmique et $B \in$ co-$\boldsymbol{NL}$ alors $A \in$ co-$\boldsymbol{NL}$.*

Idée de la 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 $i^{\text{ème}}$ symbole du mot $tr(x)$, on utilise la machine qui calcule tr comme sous-routine.
2. 3. Même principe. ■

Théorème 25 *Si on montre qu'un problème **NL**-complet est dans **L**, alors **L** = **NL**.*

Théorème 26 ***ACCESSIBILITE*** *est **NL**-complet.*

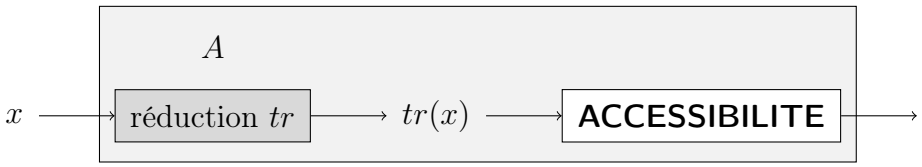
Idée de la Démonstration.
Dans NL cf proposition 11.
NL-dur Soit A un problème **NL**. Il existe une machine non-déterministe M (à deux rubans comme décrit plus haut) qui décide A en espace $O(\log n)$. On construit une réduction tr en espace logarithmique de A vers **ACCESSIBILITE**. Sans perte de généralité, 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**.
■

4.5.4 NL ⊆ P

Théorème 27 ***NL*** $\subseteq$ ***P***.

Idée de la Démonstration.
Soit A dans **NL**. Comme **ACCESSIBILITE** est **NL**-dur, A se réduit **ACCESSIBILITE** en espace logarithmique :



Comme **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. ■

4.5.5 NL = co-NL

Proposition 12 $\overline{\textbf{\textit{ACCESSIBILITE}}}$ *est dans* ***NL***.

Idée de la Démonstration.
On écrit un algorithme de la forme :

procédure 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.

fonction $nonpathnb?(G, s, t, c)$
| $n := 0$
| **pour** u sommet de G différent de t
| | **choisir** $b \in \{0, 1\}$
| | **si** $b = 1$
| | | $path?(G, s, u)$
| | | $n := n + 1$
| **si** $n \neq c$ **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.

fonction $getacc(G, s)$
| $getnumber(G, s, |G|)$

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 1 *Contrairement à [Sipser, 2006], on écrit ici une fonction récursive pour comprendre que le non-déterminisme suffit à compter le nombre de sommets accessibles en au plus i étapes. Attention, elle n'est pas en espace logarithmique car il faut stocker la pile d'appel. Le livre [Sipser, 2006] donne une version itérative en espace logarithmique.*

```
fonction getnumber(G, s, i)
    si i = 0
        retourner 1
    sinon
        c' := getnumber(G, s, i − 1)
        n := 0
        pour v sommet de G
            n' := 0
            pour u sommet de G
                choisir b ∈ {0, 1}
                si b = 1
                    path?(G, s, u, i − 1)
                    n' := n' + 1
                    si u →^G v
                        n := n + 1
                        break
            si n' ≠ c' rejeter
        retourner 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 28 ***NL*** = *co-**NL**. [Sipser, 2006][p. 331]*

Idée de la Démonstration.

$\boxed{\subseteq}$ Soit A dans **NL**. Comme **ACCESSIBILITE** est **NL**-complet, A se réduit à **ACCESSIBILITE** en espace logarithmique. Par le théorème 24, comme **ACCESSIBILITE** est dans co-**NL**, alors A est dans co-**NL**.

$\boxed{\supseteq}$ Soit A dans co-**NL**. Comme **ACCESSIBILITE** est NL-complet, $\overline{A}$ se réduit à **ACCESSIBILITE** en espace logarithmique. Par le théorème 24, comme **ACCESSIBILITE** est dans co-**NL**, alors $\overline{A}$ est dans co-**NL**. Donc A est dans **NL**.

■

4.5.6 2SAT

Proposition 13 *$2SAT \in$ **NL**.*

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

procédure $\overline{2sat}(\varphi)$
  choisir une variable propositionnelle p dans φ
  $\ell := p$
  tant que $\ell \neq \neg p$
    Choisir une clause de la forme $\ell \to \ell'$
    $\ell := \ell'$
  tant que $\ell \neq p$
    Choisir une clause de la forme $\ell \to \ell'$
    $\ell := \ell'$
  accepter

■

Proposition 14 *([Papadimitriou,], p. 398, Th. 16.3) 2SAT est **NL**-dur.*

Idée de la Démonstration.

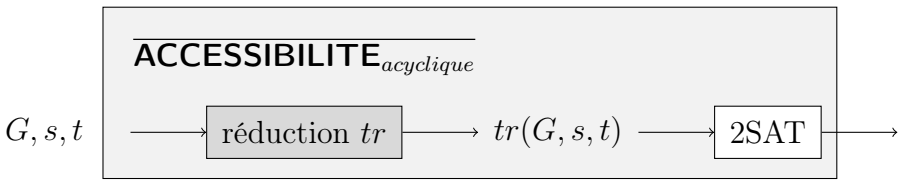
ACCESSIBILITE$_{acyclique}$
entrée : Un graphe G **acyclique**, s, t
sortie : oui, s'il existe un chemin de s à t dans G ; non, sinon.

Lemme 5 ***ACCESSIBILITE***$_{acyclique}$ *est **NL**-complet.*

Idée de la Démonstration.
Dans **NL** cf proposition 11. **NL**-dur Voir démonstration du théorème 26. En supposons que la machine M ne boucle pas, le graphe G_x est acyclique.

■

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



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

On a :
— tr est calculable en espace logarithmique ;
— $(G,s,t) \in \overline{\textbf{ACCESSIBILITE}_{acyclique}}$ iff $tr(G,s,t) \in$ 2SAT.

■

4.5.7 Problèmes P-complets

Définition 41 (P-dur)
Un problème **A** est **P**-dur si tout problème **B** dans **P** se réduit en espace logarithmique à **A**.

Définition 42 (P-complet)
Un problème est **P**-complet s'il est dans **P** et est **P**-dur.

Proposition 15 *S'il existe un problème* ***A*** *qui est* ***P****-dur et dans* ***NL****, alors* $\boldsymbol{P = NL}$.

Théorème 29 *Le problème suivant est* ***P****-complet ([Papadimitriou,], 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.

Idée de la 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.

Chapitre 5

Théorie des fonctions récursives

Points du programme de l'agrégation
Définition des fonctions primitives récursives ; schémas primitifs (minimisation bornée). Définition des fonctions récursives ; fonction d'Ackerman. Équivalence avec les fonctions récursives.

5.1 Fonctions primitives récursives

5.1.1 Syntaxe d'un langage de programmation fonctionnelle

Définition 43 (Langages des expressions des fonctions primitives récursives)
Le langage $\mathcal{L}_{FPR}$ des **expressions des fonctions primitives récursives** est défini par induction :

- $\mathbb{O}$ est une expression d'arité 0 ;
- σ est une expression d'arité 1 ;
- π_i^n où $n \in \mathbb{N}^*$ et $i \in \{1, \dots, n\}$ est d'arité n ;
- Si F est une expression d'arité n et $G_1, \dots, G_n$ sont des expressions d'arité ℓ alors : $\circ(F, G_1, \dots, G_n)$ est une expression d'arité ℓ ;
- Si F est une expression d'arité n et G est une expression d'arité $n+2$ alors $rec\,(F, G)$ est une expression d'arité $n+1$.

5.1.2 Sémantique

Définition 44 (sémantique d'une expression)
On définit la fonction $\mathsf{sem} : \mathcal{L}_{FPR} \to \bigcup_{k\in\mathbb{N}} \mathcal{F}(\mathbb{N}^k, \mathbb{N})$ par induction structurelle :

- $\mathsf{sem}(\mathbb{O}) = \begin{array}{ccc} \mathbb{N}^0 & \to & \mathbb{N} \\ / & \mapsto & 0 \end{array}$;
- $\mathsf{sem}(\sigma) = \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ x & \mapsto & x+1 \end{array}$;
- $\mathsf{sem}(\pi_i^n) = \begin{array}{ccc} \mathbb{N}^n & \to & \mathbb{N} \\ (x_1, \dots, x_n) & \mapsto & x_i \end{array}$;
- Si F est une expression d'arité n et $G_1, \dots, G_n$ sont des expressions d'arité ℓ alors $\mathsf{sem}(\circ(F, G_1, \dots, G_n)) = \begin{array}{ccc} \mathbb{N}^\ell & \to & \mathbb{N} \\ \vec{x} & \mapsto & \mathsf{sem}(F)(\mathsf{sem}(G_1)(\vec{x}), \dots, \mathsf{sem}(G_n)(\vec{x})) \end{array}$;
- Si F est une expression d'arité ℓ et G est une expression d'arité $\ell+2$ alors $\mathsf{sem}(rec\,(F, G)) = g$ où $g : \mathbb{N}^{\ell+1}$ est la fonction définie par récurrence par :

$$g(\vec{x}, k) = \begin{cases} \mathsf{sem}(F)(\vec{x}) & \text{si } k = 0 \\ \mathsf{sem}(G)(\vec{x}, k-1, g(\vec{x}, k-1)) & \text{si } k > 0. \end{cases}$$

Remarque 2 *On peut fabriquer la fonction nulle d'arité 1 comme suit :* $\mathbb{O}^1 = rec\,(\mathbb{O}, \pi_2^2)$.
Puis la fonction nulle d'arité 2 : $\mathbb{O}^2 = \circ(\mathbb{O}^1, \pi_1^3)$, *etc.*
Par simplicité, on note $\mathbb{O}$ *la fonction nulle quelque soit l'arité.*

Le programme $\circ(\sigma, \pi_3^4)$ renvoie $x_3 + 1$ si on lui donne (x_1, x_2, x_3, x_4) en entrée. La fonction

$$\begin{array}{ccc} \mathbb{N}^4 & \to & \mathbb{N} \\ (x_1, x_2, x_3, x_4) & \mapsto & x_3 + 1 \end{array}$$

est le **sens** du programme $\circ(\sigma, \pi_3^4)$, noté $\mathsf{sem}(\circ(\sigma, \pi_3^4))$.

Définition 45 (fonction récursive primitive)
Une fonction $\mathbb{N}^k \to \mathbb{N}$ est **récursive primitive** si elle est dans $\mathsf{sem}(\mathcal{L}_{FPR})$. On note $FPR = \mathsf{sem}(\mathcal{L}_{FPR})$.

Proposition 16 *L'ensemble des fonctions récursives primitives est dénombrable.*

Idée de la Démonstration.
Le langage $\mathcal{L}_{FPR}$ est dénombrable. Donc $\mathsf{sem}(\mathcal{L}_{FPR})$ est dénombrable. ■

5.1.3 Schémas primitifs dans un langage de programmation impératif

Fonction nulle $\mathbb{O}$

```
fonction nul()
|   retourner 0
```

Fonction successeur σ

```
fonction succ(x)
|   retourner x + 1
```

Fonction projection σ

```
fonction proj(x_1, ..., x_n)
|   retourner x_i
```

Composition $\circ(F, G_1, \ldots, G_n)$

```
fonction compFG1Gn(x⃗)
|   retourner F(G_1(x⃗), ..., G_n(x⃗))
```

en supposant que les fonctions $F, G_1, ...G_n$ soient définies.

Récursivté

```
fonction recFG(x⃗, k)
|   r := F(x⃗)
|   pour i := 1 à k
|   |   r := G(x⃗, i − 1, r);
|   retourner r
```

en supposant que les fonctions F et G soient définies.

5.2 Exemples de fonctions récursives primitives

Exemple 18

$$+ : \begin{array}{ccc} \mathbb{N}^2 & \to & \mathbb{N} \\ (x, y) & \mapsto & x + y \end{array}$$

$$\wr$$

$$+(x, k) = \begin{cases} x = \mathsf{sem}(\pi_1^1) & si\ k = 0 \\ +(x, k-1) + 1 = \mathsf{sem}(\circ(\sigma, \pi_3^3)(x, k-1, +(x, k-1)) & si\ k > 0. \end{cases}$$

$$\wr$$

$$+ = \mathsf{sem}(rec\,(\pi_1^1, \circ(\sigma, \pi_3^3)))$$

Voir : `http://people.irisa.fr/Francois.Schwarzentruber/recursive_functions/`

$$a+b = a+\underbrace{1+\cdots+1}_{b \text{ examplaires}} \qquad a^b = a\uparrow^1 b = \underbrace{a\times\cdots\times a}_{b \text{ exemplaires}}$$

$$a\times b = \underbrace{a+\cdots+a}_{b \text{ exemplaires}} \qquad a\uparrow^2 b = \underbrace{a^{a^{\cdot^{\cdot^{\cdot^{a}}}}}}_{b \text{ exemplaires}}.$$

Définition 46 (flèche de Knuth)

— $a\uparrow^1 b = a^b$;
— Pour $n>1$, $a\uparrow^n b = \begin{cases} 1 & \text{si } b=0 \\ a\uparrow^{n-1}(a\uparrow^n (b-1)) & \text{sinon} \end{cases}$

Proposition 17 *Pour tout n, la fonction* $\uparrow^n : \begin{array}{ccc} \mathbb{N}^2 & \to & \mathbb{N} \\ (a,b) & \mapsto & a\uparrow^n b \end{array}$ *est récursive primitive.*

5.2.1 Prédicats

Définition 47 (prédicat)
Soit F expression d'arité ℓ.
F est un **prédicat** d'arité ℓ si $\mathsf{sem}(F) : \mathbb{N}^\ell \to \{0,1\}$.

Notation 1 *On écrit* $\overset{ifthen}{else}(cond, F_{iftrue}, F_{iffalse})$ *pour l'expression :*

$$\circ(plus, \circ(mult, cond, F_{iftrue}), \circ(mult, \circ(not, cond), F_{iffalse}))$$

où :
— *cond est un prédicat d'arité ℓ ;*
— *F_{iftrue} et $F_{iffalse}$ sont des expressions d'arité ℓ.*

Proposition 18 $\mathsf{sem}($ $\overset{ifthen}{else}(cond, F_{iftrue}, F_{iffalse}))$ *est la fonction :*

$$\begin{array}{ccc} \mathbb{N}^\ell & \to & \mathbb{N} \\ \vec{x} & \mapsto & \begin{cases} \mathsf{sem}(F_{iftrue})(\vec{x}) & \text{if } \mathsf{sem}(cond)(\vec{x}) = 1 \\ \mathsf{sem}(F_{iffalse})(\vec{x}) & \text{if } \mathsf{sem}(cond)(\vec{x}) = 0 \end{cases} \end{array}$$

5.2.2 Minimisation bornée

Définition 48 (minimisation bornée)
Soit F expression d'arité $\mathbb{N}^{\ell+1}$. On note $minbornee_{\ell+1}(F)$ l'expression suivante :

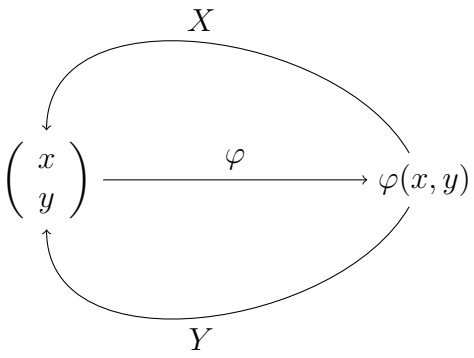
$$rec\left(\begin{array}{ll} \overset{ifthen}{else}(\circ(F, \pi_1^{\ell+1}, \ldots, \pi_\ell^{\ell+1}, \mathbb{O}), & , \quad \overset{ifthen}{else}(\circ(\leq, \pi_{\ell+2}^{\ell+2}, \pi_{\ell+1}^{\ell+2}), \\ \mathbb{O}^\ell, \quad 1^\ell) & \pi_{\ell+2}^{\ell+2}, \quad \overset{ifthen}{else}(\circ(F, \pi_1^{\ell+1}, \ldots, \pi_\ell^{\ell+1}, \circ(\sigma, \pi_{\ell+1}^{\ell+2})), \\ & \circ(\sigma, \pi_{\ell+1}^{\ell+2}), \quad \circ(\sigma, \circ(\sigma, \pi_{\ell+1}^{\ell+2}))) \end{array}\right)$$

Proposition 19 *Soit F expression d'arité $\mathbb{N}^{\ell+1}$.*
$\mathsf{sem}(minbornee_{\ell+1}(F))$ *est la fonction suivante :*

$$\begin{array}{ccc} \mathbb{N}^{\ell+1} & \to & \mathbb{N} \\ (\vec{x}, n) & \mapsto & \begin{cases} \text{le plus petit } i \in \{0,\ldots,n\} \text{ tel que } \mathsf{sem}(F)(\vec{x}, i) \neq 0 \\ n+1 \text{ si un tel } i \text{ n'existe pas.} \end{cases} \end{array}$$

5.3 Codage par un entier

5.3.1 Bijection de $\mathbb{N}^2$ dans $\mathbb{N}$



Théorème 30 ([Dehornoy, 2000], p. 181) *Il existe une bijection φ de $\mathbb{N}^2$ dans $\mathbb{N}$ qui est primitive récursive et dont l'inverse est aussi primitive récursive.*

Idée de la Démonstration.
Définition Soit $\varphi(x,y) = \underbrace{2^x \times (2y+1)}_{\psi(x,y)} - 1$.

φ est une bijection La fonction ψ est bijection de $\mathbb{N}^2$ sur $\mathbb{N}^*$ car tout nombre non nul se décompose de façon unique en le produit d'une puissance de deux et d'un nombre impair. Ainsi, φ est une bijection de $\mathbb{N}^2$ dans $\mathbb{N}$.

φ est récursive primitive La fonction φ est bien récursive primitive comme composition de fonctions qui le sont.

φ^{-1} est une bijection La réciproque est de ψ est $\psi^{-1}(n) = (X(n), Y(n))$ avec :
— $X(n)$ = l'exposant de 2 dans la décomposition en nombre premier de n ;
— $Y(n)$ = la partie entière du nombre impair dans la décomposition en nombre premier de n.

φ^{-1} est récursive primitive **Retrouver l'abscisse.** X est le nombre de pas de divisions par 2 à réaliser à partir de n jusqu'à obtenir un entier impair. Un pas de calcul, c'est à dire une division lorsque c'est pair est $D(n) = \frac{n}{2}$ si n est pair et $= n$ si n est impair.

k pas de calcul se faire en itérant : on calcule n, puis $D(n)$, puis $D^2(n)$, etc. puis on s'arrête lorsque $D^k(n) = D^{k+1}(n)$. Soit $I(n,k)$ défini par :
— $I(n,0) = n$
— $I(n,k+1) = D(I(n,k))$.
On a :

$$X(n) = \sum_{i=0}^{n} 1_<(I(n,i+1), I(n,i)).$$

Retrouver l'ordonnée. Une chose sûre, quand on itère trop (n fois par exemple !) le calcul précédent, on stagne sur le nombre impair. Il n'y a plus qu'à le rediviser par deux encore une fois pour avoir 'y' :

$$Y(n) = \lfloor \frac{I(n,n)}{2} \rfloor.$$

■

5.3.2 Récurrence multiple

Définition 49 (fonction récursive primitive à valeurs dans $\mathbb{N}^q$)
On dit qu'une fonction $\vec{g} : \mathbb{N}^\ell \to \mathbb{N}^q$ est **récursive primitive** si pour tout $i \in \{1, \ldots, q\}$, g_i est récursive primitive.

Proposition 20 *si $\vec{g} : \mathbb{N}^\ell \to \mathbb{N}^2$ est récursive primitive et $\vec{h} : \mathbb{N}^{\ell+2+1} \to \mathbb{N}^2$ est récursive primitive, alors, $\vec{f} : \mathbb{N}^{\ell+1} \to \mathbb{N}^2$ définie par :*
- $\vec{f}(\vec{n}, 0) = \vec{g}(\vec{n})$;
- $\vec{f}(\vec{n}, k+1) = \vec{h}(\vec{n}, k, \vec{f}(\vec{n}, k))$

est primitive récursive.

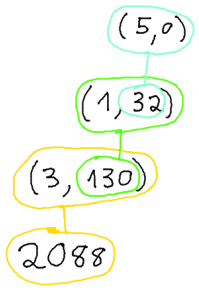
Idée de la Démonstration.
On écrit :
- $c(\vec{n}, 0) = \varphi(\vec{g}(\vec{n}))$;
- $c(\vec{n}, k+1) = \varphi(\vec{h}(\vec{n}, k, X(c(\vec{n}, k)), Y(c(\vec{n}, k))))$

■

Exemple 19 *La fonction fib définie par :*
- $fib(0) = fib(1) = 1$;
- $fib(k+2) = fib(k+1) + fib(k)$

est primitive récursive. En effet, on pose $\vec{f}(k) = (fib(k), fib(k+1))$. On a alors :
- $\vec{f}(0) = (1, 1)$;
- $\vec{f}(k+1) = \vec{h}(k, \vec{f}(k))$ *où* $\vec{h}(k, \vec{x}) = (x_2, x_1 + x_2)$.

5.3.3 Structure de données : exemple des listes



Codage de la liste $[3, 1, 5]$

Encodage

On encode une liste d'entiers par un entier :
- $c([]) = 0$
- $c(x :: L) = 1 + \varphi(x, c(L))$

Opérations

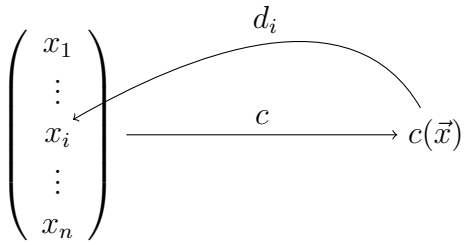
Si x est un entier et ℓ est un entier représentant une liste :
- $listevide() = 0$;
- $cons(x, \ell) = 1 + \varphi(x, \ell)$;
- $tete(\ell) = X(\ell - 1)$;
- $queue(\ell) = Y(\ell - 1)$.

5.4 Langage de programmation impératif jouet vers fonctions primitives récursives

var x_1 ; //entrée
var $x_2 = 0$;
⋮
var $x_n = 0$;

bloc du programme

sortie : x_1



Bloc de programme P	Expression $tr(P)$
$x_i := F(\vec{x})$	$\circ(c, d_1, \ldots, d_{i-1}, \circ(F, d_1, \ldots, d_n), d_{i+1}, \ldots, d_n)$
$bloc_1; bloc_2$	$\circ(tr(bloc2), tr(bloc1))$
si $x_i \neq 0$ **alors** | *bloc*	$\underset{\text{else}}{\text{ifthen}}(\circ(notzero, d_i), tr(bloc), \pi_1^1)$
répéter x_i fois | *bloc*	$\circ(rec\,(\pi_1^1, \circ(tr(bloc), \pi_3^3)), \pi_1^1, d_i)$

L'expression d'un programme constitué d'un bloc de programme *bloc* est

$$\circ(d_1, \circ(tr(bloc), \circ(c, \pi_1^1, \mathbb{O}^1, \ldots, \mathbb{O}^1)))$$

.

5.5 Limite des fonctions récursives primitives

5.5.1 Preuve via un argument diagonal

Théorème 31 *Il existe des fonctions calculables*[1] *mais non primitives récursives.*

Idée de la Démonstration.

Considérons une **énumération effective** $(e_n)_{n\in\mathbb{N}}$ des expressions du langage $\mathcal{L}_{FPR}$ d'arité 1. On considère

$$g : \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & \mathsf{sem}(e_n)(n) + 1 \end{array}$$

.

La fonction g est calculable. Voici un algorithme qui calcule g :

1. par une machine de Turing

procédure *calculerg*(n)
  Enumérer des expressions $e_1, e_2, \ldots$ jusqu'à obtenir e_n ;
  Interpréter l'expression e_n pour calculer $r := \mathsf{sem}(e_n)(n)$
  retourner $r + 1$

La fonction g n'est pas primitive récursive. Par l'absurde. Supposons que g est récursive primitive. Il existe k tel que $g(x) = \mathsf{sem}(e_k)(x)$ pour tout x. En particulier, $g(k) = \mathsf{sem}(e_k)(k) = \mathsf{sem}(e_k)(k) + 1$. Contradiction.
■

5.5.2 Vers une fonction trop rapide : Ackermann-Péter

Théorème 32

$$\begin{array}{rccl} f: & \mathbb{N}^3 & \to & \mathbb{N} \\ & (a,b,n) & \mapsto & a \uparrow^n b \end{array}$$

n'est pas primitive récursive.

Pour simplifier, supprimons a comme argument :

Définition 50 (fonction Ackermann-Péter)
$A : \mathbb{N}^2 \to \mathbb{N}$ définie par inductivement [2] :
— $A(0, m) = m + 1$
— $A(k + 1, 0) = A(k, 1)$
— $A(k + 1, m + 1) = A(k, A(k + 1, m))$

Proposition 21

$$A(k, n) = 2 \uparrow^{k-2} (n + 3) - 3.$$

Idée de la Démonstration.
Par récurrence. ■

Proposition 22 *A est calculable.*

Idée de la Démonstration.
Sa définition donne un algorithme. ■

Théorème 33 *$A \notin FPR$.*

Idée de la Démonstration.
[[Dehornoy, 2000], p. 192] Résumé de la démonstration :

1. On montre que les fonctions primitives récursives sont assez lentes (par induction structurelle).
2. Par ailleurs, $n \mapsto A(n, n)$ n'est pas lente. Donc $A \notin FPR$.

Par '$f : \mathbb{N}^\ell \to \mathbb{N}$ est lente', on entend la propriété $P(f)$ suivante :

il existe un entier k tel que pour tout $(n_1, \ldots, n_\ell) \in \mathbb{N}^\ell$, on a

$$f(n_1, \ldots, n_\ell) \leq A(k, \sum_{i=1}^{\ell} n_i).$$

■

2. L'induction est ici défini sur l'ordre lexicographique sur $\mathbb{N}^2$

5.6 Fonctions μ-récursives partielles

5.6.1 Syntaxe

Définition 51 (Langages des expressions des fonctions μ-récursives)
Le langage $\mathcal{L}_{F\mu R}$ des **expressions des fonctions μ-récursives** sont définies par induction :

- $\mathbb{O}$ est une expression d'arité 1 ;
- σ est une expression d'arité 1 ;
- π_i^n où $n \in \mathbb{N}^*$ et $i \in \{1, \dots, n\}$ est d'arité n ;
- Si F est une expression d'arité n et $G_1, \dots, G_n$ sont des expressions d'arité ℓ alors : $\circ(F, G_1, \dots, G_n)$ est une expression d'arité ℓ ;
- Si F est une expression d'arité n et G est une expression d'arité $n+2$ alors $rec\,(F, G)$ est une expression d'arité $n+1$;
- Si F est d'arité $\ell + 1$ alors $mu(F)$ est une expression d'arité ℓ.

5.6.2 Sémantique

Définition 52 (sémantique)
On définit la fonction $\mathsf{sem} : \mathcal{L}_{FPR} \to \bigcup_{k\in\mathbb{N}} \mathcal{F}_{partielle}(\mathbb{N}^k, \mathbb{N})$ par induction structurelle :

- $\mathsf{sem}(\mathbb{O}) = \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ x & \mapsto & 0 \end{array}$;
- $\mathsf{sem}(\sigma) = \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ x & \mapsto & x+1 \end{array}$;
- $\mathsf{sem}(\pi_i^n) = \begin{array}{ccc} \mathbb{N}^n & \to & \mathbb{N} \\ (x_1, \dots, x_n) & \mapsto & x_i \end{array}$;
- Si F est une expression d'arité n et $G_1, \dots, G_n$ sont des expressions d'arité ℓ alors

$$\mathsf{sem}(\circ(F, G_1, \dots, G_n)) = \begin{array}{ccc} \mathbb{N}^\ell & \to & \mathbb{N} \\ \vec{x} & \mapsto & \begin{cases} \mathsf{sem}(F)(\mathsf{sem}(G_1)(\vec{x}), \dots, \mathsf{sem}(G_n)(\vec{x})) \\ \quad \text{si } \mathsf{sem}(G_1)(\vec{x}), \dots, \mathsf{sem}(G_n)(\vec{x}) \text{ sont définis} \\ \text{non défini sinon} ; \end{cases} \end{array}$$

- Si F est une expression d'arité ℓ et G est une expression d'arité $\ell + 2$ alors $\mathsf{sem}(rec\,(F, G)) = g$ où $g : \mathbb{N}^{\ell+1}$ est la fonction définie par récurrence par :

$$g(\vec{x}, k) = \begin{cases} \mathsf{sem}(F)(\vec{x}) & \text{si } k = 0 \\ \mathsf{sem}(G)(\vec{x}, k-1, g(\vec{x}, k-1)) & \text{si } k > 0 \text{ et } g(\vec{x}, k-1) \text{ définie.} \\ \text{non défini} & \text{sinon} \end{cases}$$

- Si F est d'arité $\ell + 1$ alors

$$\mathsf{sem}(mu(F)) = \begin{array}{ccc} \mathbb{N}^\ell & \to & \mathbb{N} \\ \vec{x} & \mapsto & \begin{cases} \text{le plus petit } i \in \mathbb{N} \text{ tel que } \mathsf{sem}(F)(\vec{x}, i) \neq 0 \\ \qquad\qquad \text{si un tel } i \text{ existe} \\ \text{non défini} \qquad \text{si un tel } i \text{ n'existe pas.} \end{cases} \end{array}$$

Remarque 3 *On note parfois*

$$\mu i.q(\vec{n}, i) = \begin{cases} \textit{le plus petit } i \in \mathbb{N} \textit{ tel que } q(\vec{n}, i) = 1 \\ \textit{non défini si un tel } i \textit{ n'existe pas} \end{cases}$$

Définition 53 (μ-récursive partielle)
Une fonction partielle $\mathbb{N}^k \to \mathbb{N}$ est μ-récursive partielle si elle est dans $\mathsf{sem}(\mathcal{L}_{F\mu R})$.

5.6.3 Minimisation non bornée dans un langage de programmation impératif

Minimisation non borné $mu(F)$

```
fonction muF(x⃗)
    i := 0;
    tant que F(x⃗, i) = 0
     |   i := i + 1
    retourner i
```

en supposant que la fonction F est définie.

5.7 Langage de programmation impératif avec while vers fonctions μ-récursives partielles

5.7.1 Constructions des programmes

On ajoute aux constructions données en sous-section 5.4 la boucle tant que :

Bloc de programme P	Expression $tr(P)$
tant que $x_i = 0$ | *bloc*	$\circ(rep, \pi_1^1, mu(\circ(d_i, rep)))$ où — $rep = rec\,(\pi_1^1, \circ(tr(bloc), \pi_3^3))$ est interprété comme la fonction qui prend en argument l'environnement e et un nombre d'itérations i et qui retourne l'environnement après i itérations de *bloc*.

5.8 Fonctions μ-récursives totales

C'est la minimisation non bornée qui est dangereuse : le calcul effectif de $\mathsf{sem}(mu(F))(\vec{x})$ peut ne pas terminer et donc la fonction n'est pas définie en $\vec{x}$. Pour éviter cela, nous nous restreignons aux fonctions q dites **sûres**.

Définition 54 (fonction sûre)
Une fonction $q : \mathbb{N}^{\ell+1} \to \mathbb{N}$ est **sûre**[3] si pour tout $\vec{n} \in \mathbb{N}^{\ell}$, il existe $i \in \mathbb{N}$ tel que $q(\vec{n}, i) \neq 0$.

Définition 55 (μ-récursive totale)
Une fonction **μ-récursive totale** est une fonction totale de la forme $\mathsf{sem}(e)$ où $e \in \mathcal{L}_{F\mu R}$.

Proposition 23 *Il existe des fonctions totales non μ-récursives totales.*

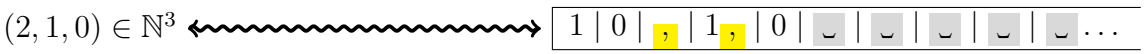
Idée de la Démonstration.
L'ensemble des fonctions est non dénombrable alors que l'ensemble des fonctions μ-récursives totales est dénombrable. ■

Remarque 4 *Attention, l'argument diagonal ne fonctionne pas. Existe-t-il une énumération* ***effective*** $(f_n)_{n\in\mathbb{N}}$ *des fonctions μ-récursives totales ?*

3. [Wolper, 2006] introduit la notion de prédicat sûr.

5.9 Équivalence avec les machines de Turing

5.9.1 Des fonctions μ-récursives aux machines de Turing



Théorème 34 *Toute fonction récursive partielle est calculable par une machine de Turing.*

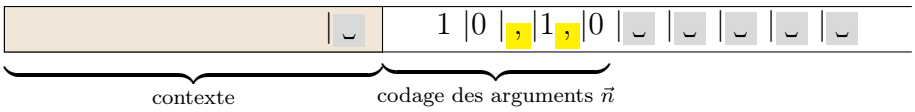
Soit e l'expression d'une fonction μ-récursive d'arité ℓ. Il existe une machine de Turing M_e déterministe tel que pour tout $\vec{x} \in \mathbb{N}^\ell$,

- *Si* $\mathsf{sem}(e)$ *est définie, l'exécution depuis la configuration initiale avec le codage de $\vec{x}$ sur le ruban termine avec le codage de* $\mathsf{sem}(e)$ *sur le ruban ;*
- *Si* $\mathsf{sem}(e)$ *n'est pas définie, l'exécution depuis la configuration initiale avec le codage de $\vec{x}$ sur le ruban ne termine pas.*

Idée de la Démonstration.

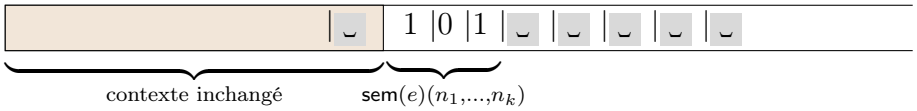
Pour tout expression $e \in \mathcal{L}_{F\mu R}$, on définit la propriété $\mathcal{P}(e)$ suivante :

Si e est d'arité k, alors il existe une machine de Turing M_e à un ruban tel que pour tout $\vec{n} \in \mathbb{N}^k$, depuis toute configuration initiale avec sur le ruban



où le contexte est soit le mot vide ou un mot sur $\{0, 1, |, ␣, ,\}$ avec jamais deux espaces ␣ consécutifs et qui finit avec un symbole ␣,

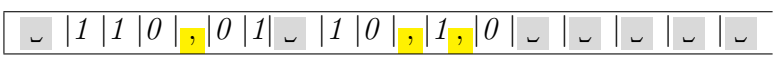
- la machine s'arrête avec le ruban suivant



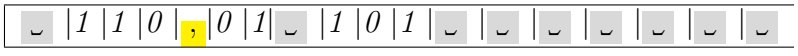
  si $\mathsf{sem}(e)(n_1, \ldots, n_k)$ est définie ;
- la machine boucle si $\mathsf{sem}(e)(n_1, \ldots, n_k)$ n'est pas définie.

␣	symbole blanc aussi utilisé pour séparer les appels de fonctions
0 et 1	symboles utilisées pour coder les entiers en binaire
,	symbole pour séparer les arguments.

Exemple 20 *Si le ruban est comme suit :*



et que $f(2, 1, 0) = 5$ alors après l'exécution de M_f, on a :



Fonction nulle. Voici $M_{\mathbb{O}}$:

Remplace les arguments (il n'y a rien) par 0 tout à la fin

d'où $\mathcal{P}(\mathbb{O})$.

Fonction successeur. Voici M_σ :

//il n'y a qu'un seul nombre après l'éventuel dernier ␣
Remplacer le dernier nombre écrit par lui-même plus 1

d'où $\mathcal{P}(\sigma)$.

$i^{\text{ème}}$ projection à n arguments. Voici $M_{\pi_i^n}$:

Des n derniers nombres après le dernier ␣
Ajouter un ␣ et réécrire le $i^{\text{ème}}$ nombre après
Recopier le nombre à la fin au début de la zone des arguments
Effacer toute la suite

d'où $\mathcal{P}(\pi_i^n)$.

Composition. Soit $g \in \mathcal{L}_{F\mu R}$ d'arité ℓ et $h_1, \ldots, h_\ell \in \mathcal{L}_{F\mu R}$ à d'arité k telles que $\mathcal{P}(g)$ et $\mathcal{P}(h_1), \ldots, \mathcal{P}(h_\ell)$ soient vraies. Voici la machine $M_{\circ(g,h_1,\ldots,h_\ell)}$.

pour $i := 1$ à ℓ
⎪ Recopier $\vec{n}$ (écrit $i-1$ portions de ruban avant) tout à la fin, précédé d'un ␣
⎪ Appeler M_{h_i} (il remplace le $\vec{n}$ que l'on vient de recopier par $\mathsf{sem}(h_i)(\vec{n})$)
Décaler les résultats $\mathsf{sem}(h_1)(\vec{n})$ ␣ … ␣ $\mathsf{sem}(h_\ell)(\vec{n})$ sur les $\vec{n}$ encore restants et efface la fin
Remplacer les $\ell - 1$ ␣ par des ,
Appeler M_g

d'où $\mathcal{P}(\circ(g, h_1, \ldots, h_\ell))$.

Récursion. Soit $g \in \mathcal{L}_{F\mu R}$ d'arité ℓ et $h \in \mathcal{L}_{F\mu R}$ d'arité $\ell + 2$ telles que $\mathcal{P}(g)$ et $\mathcal{P}(h)$ soient vraies. Voici la machine $M_{rec(g,h)}$.

Ecrire ␣
Ecrire 0 (appelons cette portion i)
Ecrire ␣
Recopier k ici
Ecrire ␣
Recopier $\vec{n}$
Appeler M_g
tant que $i < k$
⎪ Décaler $\mathsf{sem}(rec\,(g,h))(\vec{n}, i)$ pour insérer $\vec{n}$, i , avant
⎪ Appeler M_h
⎪ Incrémenter i de 1
Décaler le résultat $\mathsf{sem}(rec\,(g,h))(\vec{n}, i)$ vers la gauche en effaçant $\vec{n}$, k, i

d'où $\mathcal{P}(rec\,(g,h))$.

Minimisation non bornée Soit $q \in \mathcal{L}_{FPR}$ d'arité $\ell + 1$ tel que $\mathcal{P}(q)$.

Ecrire ␣
Ecrire 0 (appelons cette portion i)
Ecrire ␣
Recopier $\vec{n}$, 0
Appeler M_q
tant que le résultat de M_q est 0
  Incrémenter i de 1
  Recopier $\vec{n}$, i à la place du précédent résultat de M_q
  Appeler M_q
Supprimer le résultat de M_q
Décaler i vers la gauche (et supprimer $\vec{n}$)

d'où $\mathcal{P}(mu(q))$. ■

Corollaire 6 *Toute fonction récursive totale est calculable par une machine de Turing (qui s'arrête sur toutes les entrées).*

Idée de la Démonstration.
Seules les minimisations non bornées avec prédicats non sûrs donnent des exécutions infinies.
■

5.9.2 Des machines de Turing aux fonctions μ-récursives

Codage des entiers

— Attention, la représentation binaire d'un mot est ambigue. Par exemple, les mots 1, $01 \in \{0,1\}^*$ représentent tous l'entier 1.
— Une possibilité [Wolper, 2006][p. 173] est de considérer que l'alphabet $\{1,2\}$ et de travailler en base 3. Ainsi, $w = w_0, \ldots w_\ell \in \{1,2\}^*$ est codé par le nombre :

$$gd(w) = \sum_{i=0}^{\ell} 3^i w_i.$$

— Un autre alternative est d'utiliser la structure de données Liste déjà vu pour encoder un mot.

Théorème 35 *Toute fonction $\Sigma^* \to \Sigma^*$ calculable par machine de Turing déterministe avec éventuellement des exécutions infinies (où alors la fonction n'est pas définie) est μ-récursive partielle.*
Toute fonction $\Sigma^* \to \Sigma^*$ calculable par machine de Turing est récursive totale.

Idée de la Démonstration.
Voici le comportement de la machine décrit dans le langage donné à la section 5.7.1.

var mot ; // entrée
$config = configinit(mot)$;
tant que $configfinale?(config)$
  $config = configsuivante(config)$
$mot = ruban(config)$ //sortie

Donnons une démonstration proche de celle donnée dans [Wolper, 2006] Les entiers représentent les mots et les configurations de la machine. On construit les fonctions primitives récursives suivantes :

— $configinit : \mathbb{N} \to \mathbb{N}$: qui associe la représentation d'un mot x à la représentation de la configuration initiale ;
— $ruban : \mathbb{N} \to \mathbb{N}$ qui associe à la représentation d'une configuration c la représentation du mot sur le ruban.
— $suivante : \mathbb{N} \to \mathbb{N}$: qui associe la représentation d'une configuration c à la représentation de la configuration suivante de c ;
— $suivante^* : \mathbb{N}^2 \to \mathbb{N}$ qui à une configuration c et un entier n associe la configuration que l'on a dans n étapes à partir de c ;
— $configfinale? : \mathbb{N} \to \mathbb{N}$ un prédicat qui dit, étant donné une configuration c dit si elle est finale.

On utilise alors la minimisation non bornée pour définir la fonction suivante

$$nbEtapes : x \mapsto \mu i.configfinale?(suivante^*(configinit(x), i))$$

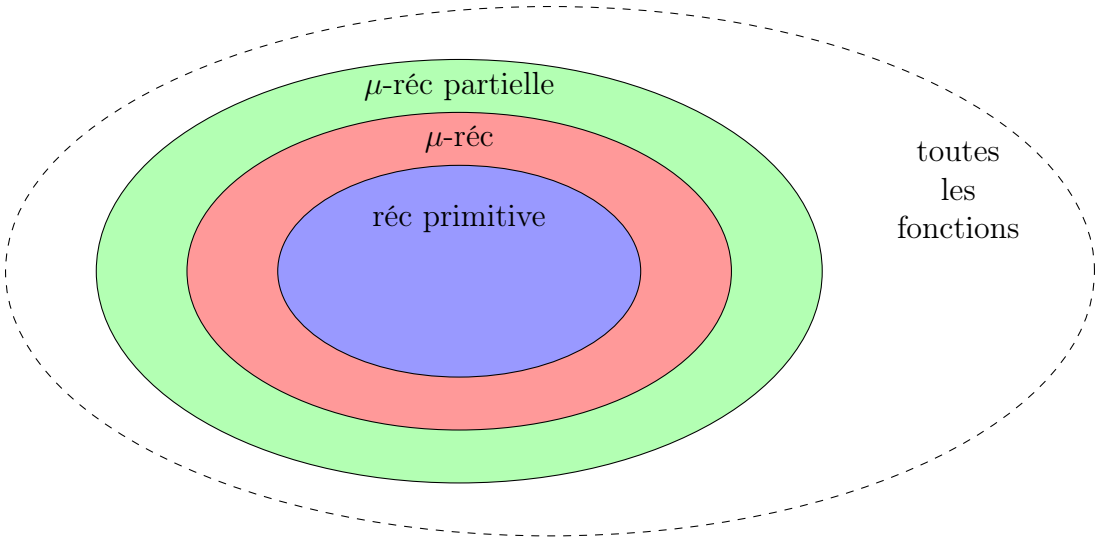
qui associe au mot d'entrée le nombre d'étapes de calcul pour arriver dans un état final.

Voici alors la fonction qui calcule la même chose que la machine de Turing :

$$f : x \mapsto ruban(suivante^*(configinit(x), nbEtapes(x))).$$

nbEtapes est a priori une **fonction partielle**. Le prédicat $configfinale?(suivante^*(configinit(x), i)$ n'est **pas forcément sûr** dans le cas général. En fait, il est sûr ssi la machine s'arrête quelque soit l'entrée. ■

5.10 Bilan



Fonctions récursives primitives	Fonctions μ-récursives totales	Fonctions μ-récursives partielles
on sait "syntaxiquement" que leurs calculs terminent toujours	on sait "sémantiquement" que leurs calculs terminent toujours	leurs calculs ne terminent pas forcément
programmes sans boucle *while*	programmes avec boucle *while*	
totales		partielles

5.11 Notes bibliographiques

Syntaxe VS sémantique

Certains livres ([Wolper, 2006], p. 155 ; [Dehornoy, 2000], p. 171) ne donnent pas de syntaxe pour les fonctions primitives récursives ou les fonctions μ-récursives partielles. Le manque d'une syntaxe s'en ressent :

— l'outil `http://people.irisa.fr/Francois.Schwarzentruber/recursive_functions/` a besoin d'une syntaxe.
— Des justifications basées sur la syntaxe qui sont vagues (dans [Wolper, 2006], p. 129, "chaque fonction primitive récursive peut être décrite par une chaîne de caractères".)
— Difficulté d'écrire les traductions vues en sous-section 5.4 et 5.7.

Martin-Löf et Kleene semblent aussi introduire une syntaxe TODO : étude bibliographie à compléter.

Prédicats

Dans son livre, Wolper [Wolper, 2006] introduit la notion de prédicat : c'est une fonction $\mathbb{N}^k \to \{0,1\}$. Intuitivement, on interprète 0 comme faux et 1 comme vrai. Je n'aime pas car les définitions ne sont plus syntaxiques. La minimisation bornée et non bornée ne sont définies que pour les prédicats mais c'est une restriction sémantique.

Fonctions à valeurs dans $\mathbb{N}^k$

Ici, nos fonctions sont à valeurs dans $\mathbb{N}$ et au besoin nous codons un tuple de $\mathbb{N}^k$ avec la technique de la sous-section 5.3. Dans [Brookshear, 1989], les fonctions sont à valeurs dans $\mathbb{N}^k$ et l'auteur introduit (p. 201) le schéma de **combinaison** : à partir de $f : \mathbb{N}^k \to \mathbb{N}^n$ et $g : \mathbb{N}^k \to \mathbb{N}^m$ on construit $f \times g : \begin{array}{ccc} \mathbb{N}^k & \to & \mathbb{N}^{n+m} \\ \vec{x} & \mapsto & (f(\vec{x}), g(\vec{x}) \end{array}$.

Fonctions partielles

Dans [Wolper, 2006], la définition de la minimisation non bornée ne donnent pas une fonctions partielles car lorsqu'un i n'existe pas, on définit $\mu i.q(\vec{n}, i) = 0$. La fonction est donc totale alors que le "calcul" ne devrait pas terminer. C'est confus.

Langages de programmation jouets

Sans while

Bucle. Dans [Hofstadter et al., 1985], il y a la présentation d'un langage appelé Bucle, sans boucle **while**. Le voici :

— Les variables sont entières positives ;
— des affectations $x =$ quelque chose de primitif récursif ;
— les variables déclarées non initialisées sont à 0 ;
— L'incrément ;
— Déclaration de fonctions ;
— Appel par valeur SANS récursion ;
— Boucles for du type : répéter x fois le bloc B (où x n'est pas modifié dans le bloc B)
— Le retour de fonction **retourner** .

Théorème 36 *Une fonction f est récursive primitive ssi il existe une fonction de Bucle qui calcule f.*

Modèle FOR. Un langage équivalent est montré dans ([Olivier Ridoux,], chap. 6) appelé le modèle FOR.

Avec while

Mucle. Dans [Hofstadter et al., 1985], il y a la présentation d'un langage appelé Mucle, qui est une extension de Bucle avec une boucle MU.

Modèle WHILE. Un langage équivalent est montré dans [Calculateurs, calculs, calculabilité, Olivier Ridoux, Gives Lesventes, chap. 6] appelé le modèle WHILE.

Langage de programmation "Bare-bones". Langage de programmation "Bare-bones" [[Brookshear, 1989] (p. 227)]

- incr x
- decr x
- boucle while $x \neq 0$
- entiers positifs
- clear x : mettre x à 0 (peut-être réalisé avec while $x \neq 0\{$ decr $x\}$)
- Pas de fonctions, pas d'appels.

Fonctions d'Ackermann

Dans la plupart des ouvrages, on parle de la fonction d'Ackermann. Il s'agit d'une simplification avec deux variables seulement dûe à Rózsa Péter.

Aller vraiment plus loin

Hiérarchie de Grzegorczyk.

Bibliographie

[Agrawal et al., 2004] Agrawal, M., Kayal, N., and Saxena, N. (2004). Primes is in p. *Annals of mathematics*, pages 781–793.

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

[Arora and Barak, 2009] Arora, S. and Barak, B. (2009). *Computational complexity : a modern approach*. Cambridge University Press.

[Brookshear, 1989] Brookshear, J. G. (1989). *Theory of computation : formal languages, automata, and complexity*. Benjamin-Cummings Publishing Co., Inc.

[Cormen, 2009] Cormen, T. H. (2009). *Introduction to algorithms*. MIT press.

[Dasgupta et al., 2006] Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. V. (2006). Algorithms.

[Dehornoy, 2000] Dehornoy, P. (2000). *Mathématiques de l'informatique : cours et exercices corrigés*. Dunod.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). Computers and intractability : a guide to np-completeness.

[Hofstadter et al., 1985] Hofstadter, D. R., Henry, J., and French, R. (1985). *Gödel, Escher, Bach : les brins d'une guirlande éternelle*. InterEditions.

[Karp, 1972] Karp, R. M. (1972). *Reducibility among combinatorial problems*. Springer.

[Khachiyan, 1980] Khachiyan, L. G. (1980). Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1) :53–72.

[Kroening and Strichman, 2008] Kroening, D. and Strichman, O. (2008). *Decision procedures : an algorithmic point of view*. Springer Science & Business Media.

[Olivier Ridoux,] Olivier Ridoux, G. L. *Calculateurs, calculs, calculabilité*. Dunod.

[Papadimitriou,] Papadimitriou, C. H. *Computational complexity*.

[Perifel, 2014] Perifel, S. (2014). *Complexité algorithmique*. Ellipses.

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

[Vazirani, 2013] Vazirani, V. V. (2013). *Approximation algorithms*. Springer Science & Business Media.

[Wolper, 2006] Wolper, P. (2006). *Introduction à la calculabilité*. Dunod.

[Wright, 2005] Wright, M. (2005). The interior-point revolution in optimization : history, recent developments, and lasting consequences. *Bulletin of the American mathematical society*, 42(1) :39–56.