HAL open science

Kit de survie - Calculabilité

François Schwarzentruber

To cite this version:

François Schwarzentruber. Kit de survie - Calculabilité. Master. France. 2018. cel-01947220

HAL Id: cel-01947220

https://cel.hal.science/cel-01947220

Submitted on 6 Dec 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Kit de survie - Calculabilité

François Schwarzentruber

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

Ces notes de cours survolent le programme en calculabilité et théorie de la complexité de l'option informatique de l'agrégation de mathématiques. Ce document a été débuté en 2016 à l'aide des élèves agrégatifs à l'ÉNS Rennes que je remercie. Elles sont volontairement laconiques mais illustrées.

Table des matières

1 Machines de Turing **5**
1.1 Problèmes de décision . . . 5
1.2 Codage . . . 5
1.3 Thèse de Church-Turing . . . 6
1.4 Définition des machines de Turing . . . 6
1.5 Equivalence entre machines de Turing . . . 7
1.5.1 Plusieurs rubans . . . 7
1.5.2 Déterministe vs non-déterministe . . . 8
1.5.3 Autres variantes . . . 10
1.6 Notes bibliographiques . . . 10

2 Indécidabilité **11**
2.1 Classes R et RE . . . 11
2.2 Enumérateurs . . . 11
2.3 Problème de l'arrêt . . . 12
2.3.1 ... est récursivement énumérable . . . 12
2.3.2 ... est indécidable . . . 12
2.4 Montrer l'indécidabilité par réduction . . . 13
2.4.1 Réduction . . . 13
2.4.2 Exemple : acceptation par une machine de Turing . . . 13
2.5 Problèmes de correspondance de Post . . . 14
2.5.1 Définitions . . . 14
2.5.2 Démonstrations d'indécidabilité . . . 15
2.6 Théorème de Rice . . . 18
2.6.1 Enoncé . . . 18
2.6.2 Exemples d'application . . . 18
2.7 Bilan . . . 19
2.8 Notes bibliographiques . . . 19

3 NP-complétude **21**
3.1 Classe P . . . 21
3.2 Classe **NP** . . . 21
3.2.1 Définition . . . 21
3.2.2 Définition alternative : vérifieur par certificat . . . 22
3.2.3 Réductions polynomiales . . . 22
3.2.4 **NP**-dureté . . . 22
3.3 Théorème de Cook . . . 23
3.3.1 Dans NP . . . 23
3.3.2 NP-dur . . . 23
3.4 Réductions polynomiales pour montrer la NP-dureté . . . 26
3.4.1 3-SAT . . . 26
3.4.2 3-coloration . . . 27
3.5 Panoramas de problèmes . . . 28
3.5.1 Problèmes dans NP conjecturés ni dans P, ni NP-complets . . . 29
3.5.2 Démonstration de l'appartenance à P récente . . . 29
3.6 NP-complétude en pratique . . . 29
3.6.1 Branch and bound . . . 29
3.6.2 Algorithmes d'approximation . . . 29
3.6.3 Réduction à SAT ou à la programmation linéaire entière . . . 29

4 Classes de complexité **31**
4.1 Définition des classes de complexité . . . 31
4.1.1 Classes non stables par modèle de calcul . . . 31
4.1.2 Classes stables par modèle de calcul . . . 31
4.2 Théorème de Savitch . . . 32
4.3 PSPACE . . . 33
4.3.1 Rappel : SAT et VALIDE . . . 33
4.3.2 QBF : formules booléennes quantifiées . . . 33
4.3.3 Jeux à deux joueurs . . . 35
4.4 Universalité d'un langage rationnel . . . 37
4.5 LOGSPACE et NLOGSPACE . . . 39
4.5.1 Définitions . . . 39
4.5.2 Accessibilité . . . 39
4.5.3 NL-complétude . . . 39
4.5.4 **NL** $\subseteq$ **P** . . . 40
4.5.5 Théorème de Immerman-Szelepcsenyi : **NL** = co-**NL** . . . 41
4.5.6 2SAT . . . 42
4.5.7 Problèmes **P**-complets . . . 43

5 Théorie des fonctions récursives **45**
5.1 Fonctions primitives récursives . . . 45
5.1.1 Schémas primitifs . . . 45
5.1.2 Syntaxe d'un langage de programmation fonctionnelle . . . 45
5.1.3 Sémantique . . . 46
5.2 Exemples de fonctions récursives primitives . . . 47
5.2.1 Prédicats . . . 47
5.2.2 Minimisation bornée . . . 47
5.3 Codage par un entier . . . 48
5.3.1 Bijection de $\mathbb{N}^2$ dans $\mathbb{N}$ . . . 48
5.3.2 Application 1 : suites définies par récurrence . . . 49
5.3.3 Application 2 : structure de données (exemple des listes) . . . 49
5.4 (*) Langage de programmation impératif jouet vers fonctions primitives récursives . . . 50
5.5 Limite des fonctions récursives primitives . . . 50
5.5.1 Preuve via un argument diagonal . . . 50
5.5.2 Fonction d'Ackermann-Péter . . . 51
5.6 Fonctions μ-récursives partielles . . . 51
5.6.1 Minimisation non bornée . . . 51
5.6.2 Syntaxe . . . 52
5.6.3 Sémantique . . . 52
5.7 Fonctions μ-récursives totales . . . 52
5.8 (*) Langage de programmation impératif avec while vers fonctions μ-récursives partielles . . . 53
5.9 Équivalence avec les machines de Turing . . . 53
5.9.1 Des fonctions μ-récursives aux machines de Turing . . . 53
5.9.2 Des machines de Turing aux fonctions μ-récursives . . . 55
5.10 Bilan . . . 56
5.11 Notes bibliographiques . . . 56

A PRIMES est dans NP **59**

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 entrée associe oui ou non.

Exemple 2 ***Connexe***
entrée : un graphe fini non orienté G ;
sortie : oui si G est connexe ; non, sinon.

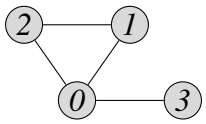
Définition 3 (instance)
Une entrée d'un problème de décision s'appelle une **instance**.

Définition 4 (instance positive, instance négative)
- Une instance est **positive** si sa sortie est oui.
- Une instance est **négative** si sa sortie est non.

Exemple 5
- *Un graphe fini non orienté G est une **instance** de **Connexe**.*
- *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**.*

1.2 Codage

Exemple 6 ([Sipser, 2006], p. 186)

Instance G 	*Codage par le mot* $\langle G \rangle$ = 100#0−1#0−10#0−11#1−10
Problème de décision ***Connexe***	*Langage* $L = \{\langle G \rangle \mid G$ *est un graphe non orienté connexe*$\}$.

1.3 Thèse de Church-Turing

Une machine de Turing
modélise
un algorithme.

1.4 Définition des machines de Turing

`http://people.irisa.fr/Francois.Schwarzentruber/turing_machine_simulator/`

Définition 7 (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 $\sqcup \notin \Sigma$;
- Γ est l'alphabet fini du ruban avec $\Sigma \subseteq \Gamma$ et $\sqcup \in \Gamma$;
- $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{\leftarrow, \rightarrow, \bullet\})$;
- $q_0 \in Q$ est l'**état initial** ;
- $q_{acc} \in Q$ est l'**état d'acceptation** ;
- $q_{rej} \in Q$ est l'**état de rejet** avec $q_{acc} \neq q_{rej}$.

Définition 8 (machine de Turing déterministe)
Une machine de Turing $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc})$ est **déterministe** si δ est une fonction partielle

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

Définition 9 (configuration)
Une **configuration** est un mot uqv où $u \in \Gamma^*$, $v \in \Gamma^\omega$ et $q \in Q$.

Définition 10 (configuration initiale)
Une **configuration initiale** est un mot $q_0 w \sqcup\sqcup \ldots$ où q_0 est l'état initial et $w \in \Sigma^*$.

Définition 11 (configuration acceptante)
Une configuration est **acceptante** si l'état de la configuration est q_{acc}.

Définition 12 (un pas de calcul)
Pour tout $a, b, c \in \Gamma$, $u \in \Gamma^*$, $v \in \Gamma^\omega$,
- 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 $ubq'v$;

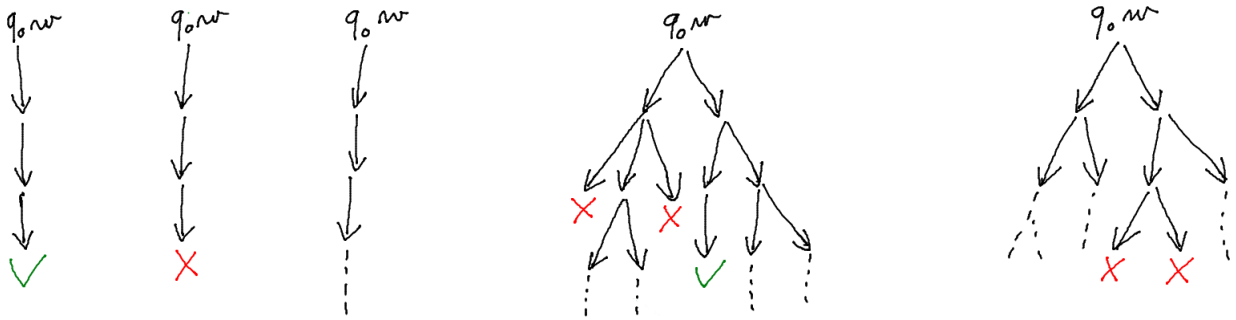
Définition 13 (exécution)
Une **exécution** est une suite maximale de configurations $C_1, \ldots, C_k$ telle que C_i devient C_{i+1}.

Définition 14 (exécution acceptante)
Une exécution est $C_1, \ldots, C_k$ **acceptante** si C_k est une configuration acceptante.

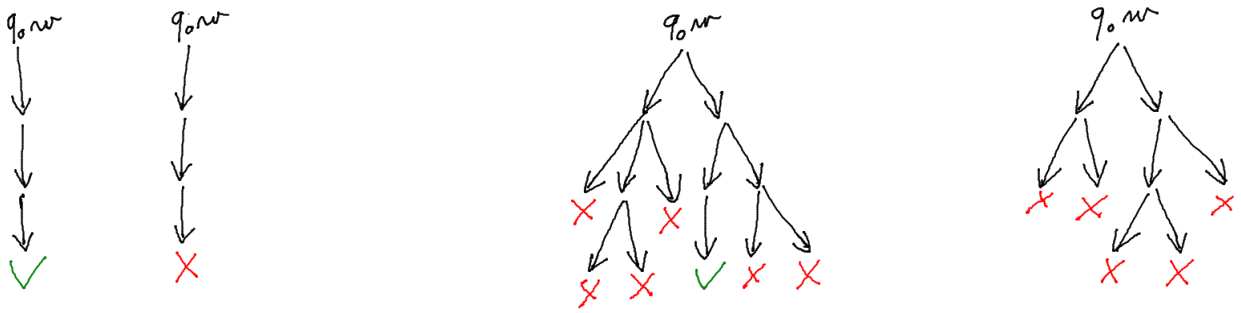
Définition 15 (acceptation d'un mot)
Une machine M **accepte** un mot w s'il existe une exécution acceptante $C_1, \ldots, C_k$
où C_1 est la configuration initiale $q_0 w \sqcup\sqcup \ldots$.

Définition 16 (arbre de calcul)
L'**arbre de calcul** depuis $q_0 w \sqcup \ldots$ est l'arbre de racine $q_0 w \sqcup \ldots$ et t.q. les fils de tout nœud C sont les configurations C' où C devient C'.

Définition 17 (langage accepté)
Le **langage accepté** par M, noté $L(M)$, est l'ensemble des mots w acceptés par M.



Définition 18 (décideur, langage décidé)
([Sipser, 2006], p. 170, p. 180) Une machine de Turing M qui s'arrête depuis q_0w pour tout mot w est appelé **décideur**. On dit que le langage accepté par M est **décidé** par M.



Définition 19 (M décide L en temps f)
Soit $f : \mathbb{N} \to \mathbb{N}$. M décide L en temps f si
- M décide L ;
- pour tout w, la hauteur de l'arbre de calcul depuis q_0w est $\leq f(|w|)$.

1.5 Equivalence entre machines de Turing

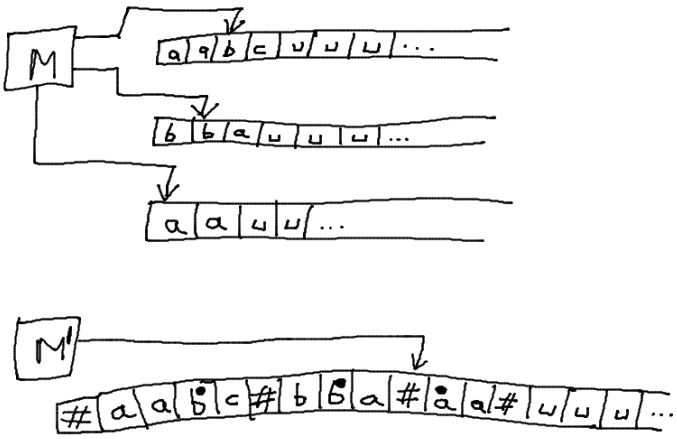
1.5.1 Plusieurs rubans

Définition 20 (machine de Turing déterministe à k rubans)
Une machine de Turing $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc})$ est déterministe si δ est une fonction partielle

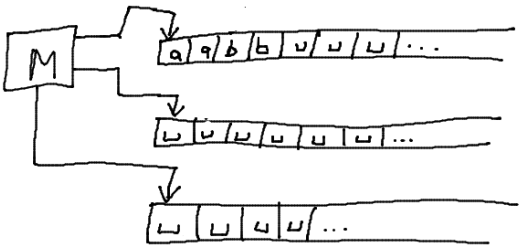
$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\leftarrow, \rightarrow, \bullet\}^k.$$

Théorème 21 *([Sipser, 2006], p. 177) Une machine de Turing déterministe à k rubans est équivalente à une machine de Turing déterministe à un ruban.*

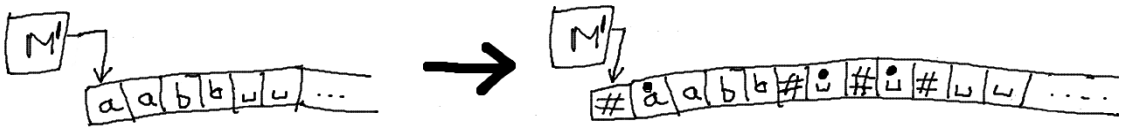
Idée de la Démonstration.
Soit M une machine de Turing déterministe à k rubans. On construit M' à un ruban qui simule M. L'idée est de concaténer les k rubans en les séparant par # et en marquant les cases sous le curseur avec $\bullet$.



La configuration initiale de M est :



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



■

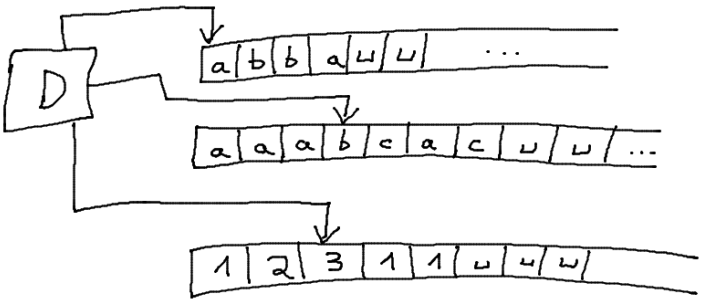
Corollaire 22 *Un langage est accepté par une machine de Turing déterministe à k rubans ssi il est accepté par une machine de Turing déterministe à un ruban.*

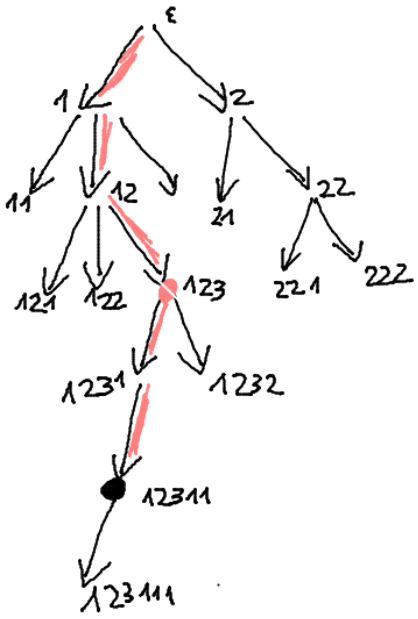
Corollaire 23 *Un langage est décidé par une machine de Turing (décideuse) déterministe à k rubans ssi il est décidé par une machine de Turing déterministe (décideuse) à un ruban.*

1.5.2 Déterministe vs non-déterministe

Théorème 24 *([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. On construit une machine déterministe D qui simule un **parcours en largeur** de l'arbre de calcul de M à trois rubans (quitte à la transformer en une machine à un ruban avec le théorème 21).





1. Le 1er ruban, en lecture seule, contient le mot d'entrée ;
2. Le 2e ruban est le ruban de travail et représente la configuration temporaire de M ;
3. Le 3e ruban contient l'adresse courante de la configuration à calculer dans l'arbre de calcul de M. Son alphabet est $\{1, \ldots, b\}$ où b est l'arité de 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 3e ruban
      tant que pas à la fin du 3e ruban ou simulation impossible
        soit i l'entier sous le curseur du 3e ruban
        simuler la i^{me} transition tirable de M sur le 2e ruban
        avancer le curseur du 3e ruban vers la droite
      si la simulation de M est acceptante **alors** **accepter**
    si pas de configuration courante de profondeur n **alors** **rejeter**

■

Corollaire 25 *Un langage est accepté par une machine de Turing non-déterministe ssi il est accepté par une machine de Turing déterministe.*

Corollaire 26 *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 27 (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 24 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 $\bullet$;
— Machines de Turing à deux états seulement ;
— Machine où les actions du curseurs sont 'se déplacer d'une case veres la droite' ou 'retourner tout au début du ruban, tout à gauche' [Sipser, 2006] ;
— Uniquement 2 lettres (en plus de $\sqcup$) ;
— Automate à file
— 2-tag

1.6 Notes bibliographiques

Dans [Papadimitriou, 2003], il utilise un caractère spécial $\triangleright$ 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 $uqv\sqcup$.

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

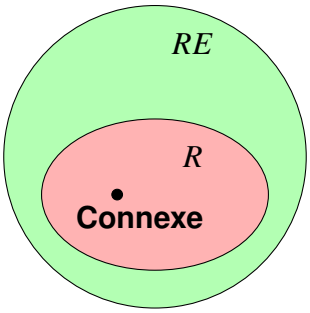
Chapitre 2

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 28 (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.

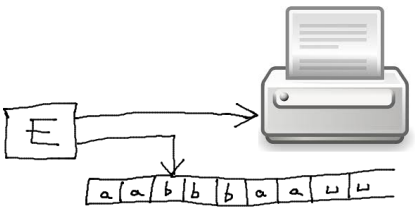


Exemple 29 ***Connexe*** *est décidable, i.e.* ***Connexe*** $\in R$.

Définition 30 (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 31 (énumérateur)
([Sipser, 2006], p. 18-181) Un **énumérateur** est une machine de Turing qui imprime/énumère des mots.



Théorème 32 *$L \in RE$ ssi il existe un énumérateur qui énumère des mots.*

Idée de la Démonstration.

$\boxed{\Leftarrow}$ Soit E un énumérateur de L.
Voici 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 config atteinte est acceptante **alors**
        imprimer w

■

2.3 Problème de l'arrêt

Définition 33 (problème de l'arrêt)
Le **problème de l'arrêt** est le problème de décision :

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

2.3.1 ... est récursivement énumérable

Théorème 34 ***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 simule l'exécution de n'importe quel machine de Turing M.

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

■

2.3.2 ... est indécidable

Théorème 35 ***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.

On construit une machine *paradoxe* comme suit :

procédure $A(M, w)$
  ...
  ...
  ...

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

paradoxe(*paradoxe*) ne termine pas
ssi
A accepte (*paradoxe*, ⟨*paradoxe*⟩)
ssi
paradoxe(*paradoxe*) termine.

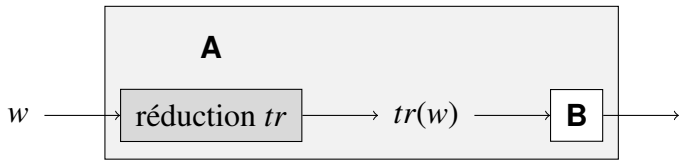
Contradiction.

■

2.4 Montrer l'indécidabilité par réduction

2.4.1 Réduction

Définition 36 (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 37 *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)*

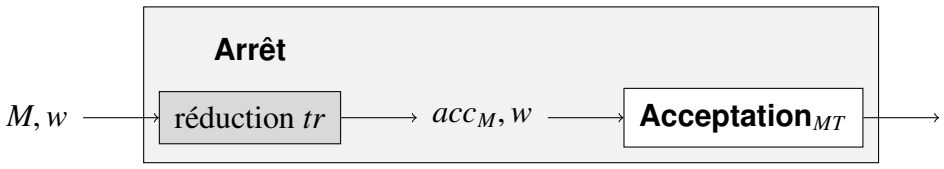
2.4.2 Exemple : acceptation par une machine de Turing

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 38 ***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 suivante :

procédure $acc_M(w)$
  $M(w)$
  accepter

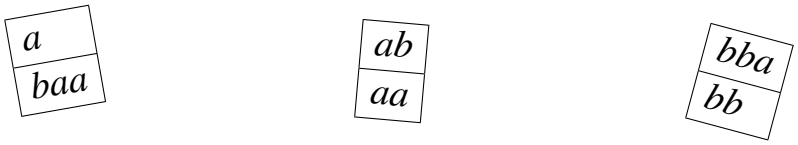
tr est une réduction de **Arrêt** dans **Acceptation**$_{MT}$. En effet :
— tr est calculable (acc_M est construite effectivement à partir de M) ;
— $(M, w) \in$ **Arrêt** ssi $M(w)$ s'arrête ssi acc_M accepte w ssi $tr(M, w) \in$ **Acceptation**$_{MT}$.

Comme **Arrêt** est indécidable, **Acceptation**$_{MT}$ est indécidable. ■

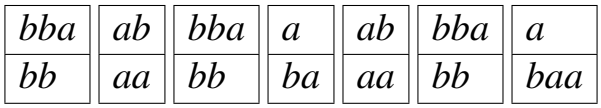
2.5 Problèmes de correspondance de Post

2.5.1 Définitions

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



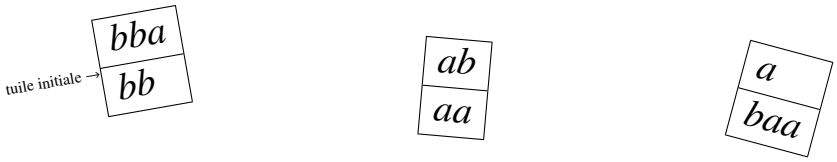
et répond oui ssi on peut mettre bout à bout des tuiles du système (on peut réutiliser plusieurs fois la même tuile) pour que les mots du haut et du bas soient identiques.

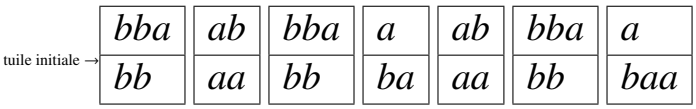


Définition 39 (Problème de correspondance de Post)

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 impose de commencer une tuile initiale.





Définition 40 (Problème de correspondance de Post marqué)

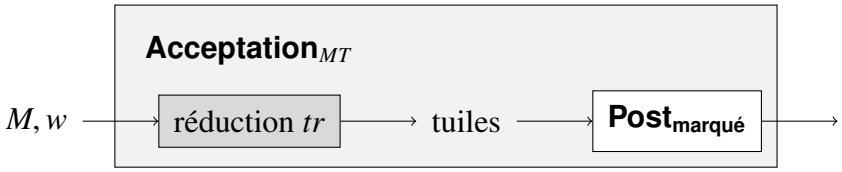
Post$_{\text{marqué}}$
entrée : un alphabet fini Σ, une famille finie de couples de mots non vides $((h_i, b_i))_{i=1..n}$ sur Σ ;
sortie : oui s'il existe $i_2, \ldots, i_p \in \{1, \ldots, n\}$ tels que $p \geq 1$ et $h_{\mathbf{1}} h_{i_2} \ldots h_{i_p} = b_{\mathbf{1}} b_{i_2} \ldots b_{i_p}$; non,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

2.5.2 Démonstrations d'indécidabilité

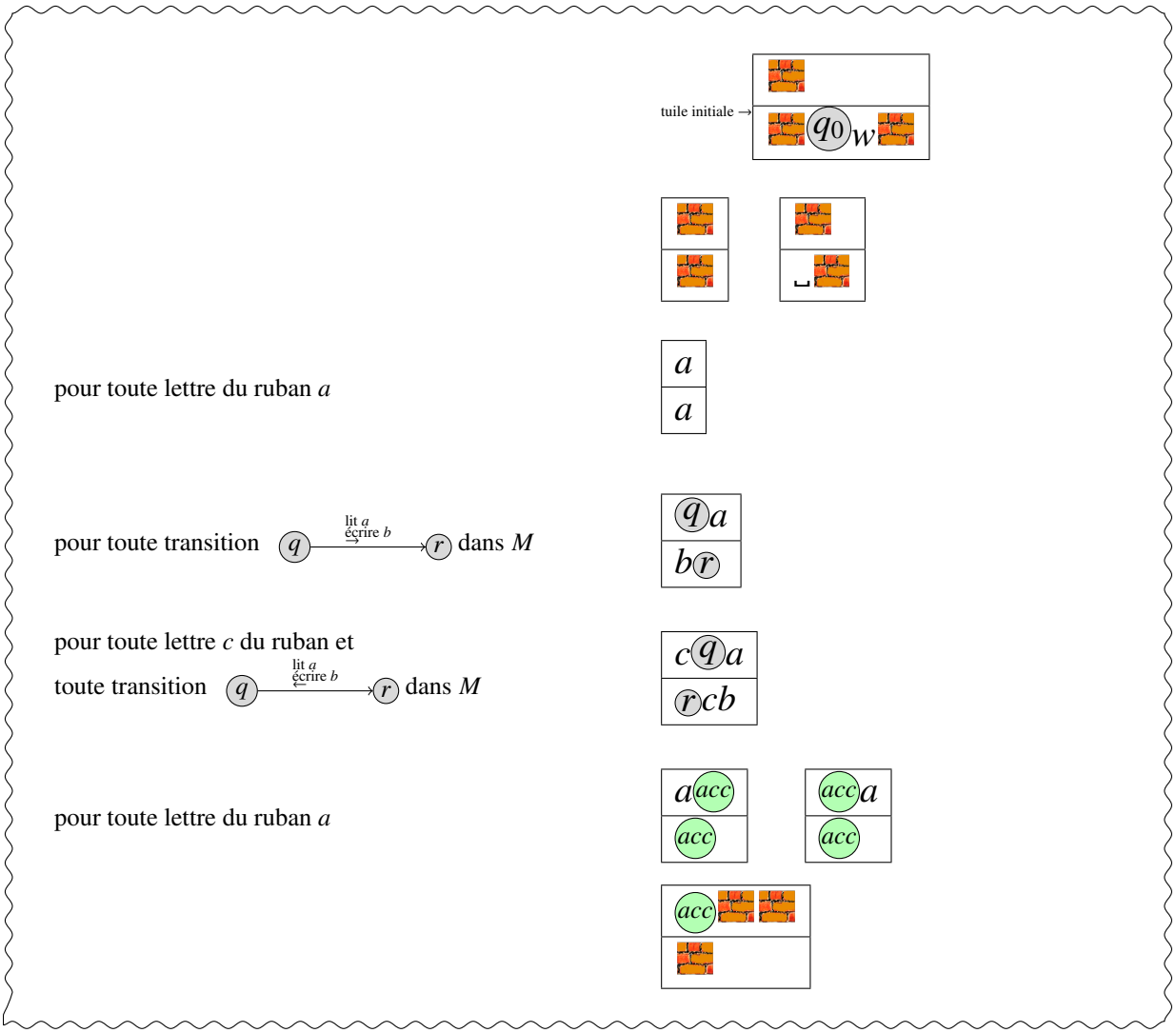
Théorème 41 ***Post*$_{\textbf{marqué}}$** *est indécidable.*

IDÉE DE LA DÉMONSTRATION.
[Sipser, 2006]

Définissons une réduction tr de **Acceptation**$_{MT}$ dans **Post**$_{\textbf{marqué}}$.



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

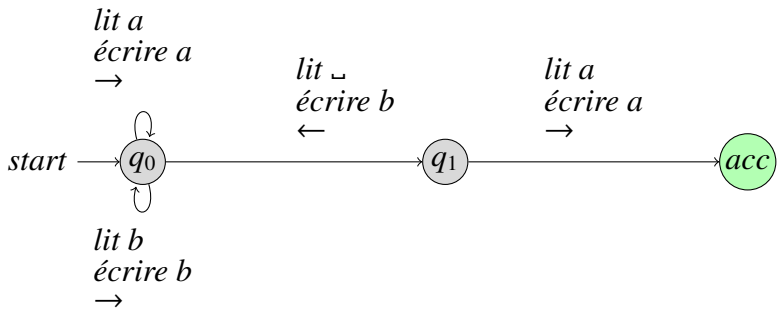


1. tr est une fonction calculable ;
2. M accepte w ssi $tr(M, w)$ est une instance positive de **Post**$_{\textbf{marqué}}$.

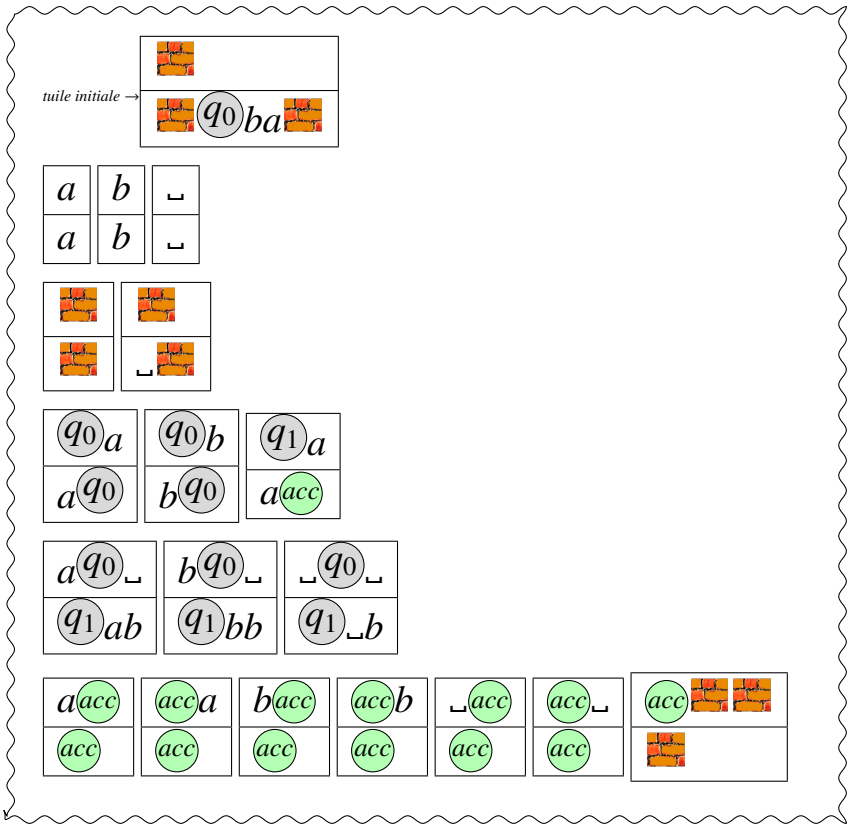
Comme **Acceptation**$_{MT}$ est indécidable, **Post**$_{\textbf{marqué}}$ est indécidable.

■

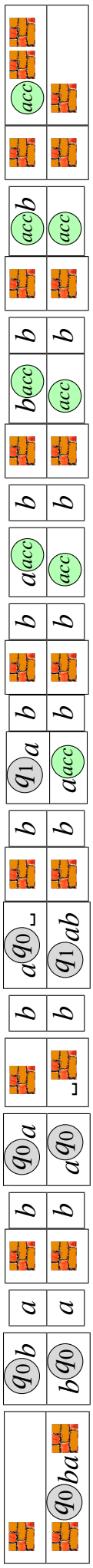
Exemple 42 *Considérons la machine de Turing M suivante qui accepte les mots qui finissent par un a :*



et le mot w = ba.

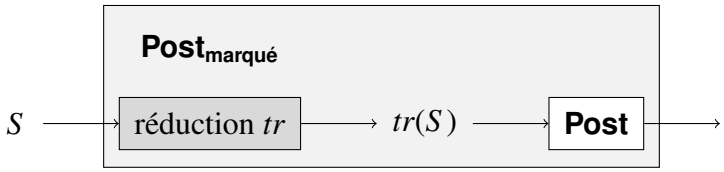


L'exécution acceptante de M sur aba est représentée par la suite de tuiles sur le bord droit de la page.



Théorème 43 ***Post*** *est indécidable.*

IDÉE DE LA DÉMONSTRATION.
[Sipser, 2006] Définissons une réduction de **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 **Post**$_{\textbf{marqué}}$ ssi $tr(S)$ instance positive de **Post**.

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

■

Exemple 44 *L'instance de* ***Post***$_{\textbf{marqué}}$

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

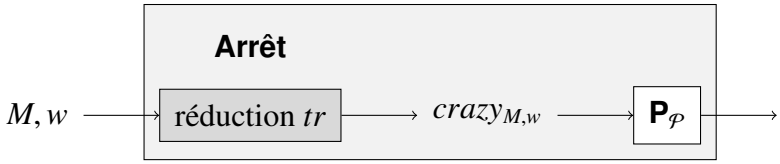
2.6 Théorème de Rice

2.6.1 Enoncé

Théorème 45 (de Rice) *[Wolper, 2006] Soit $\mathcal{P}$ telle $\emptyset \subsetneq \mathcal{P} \subsetneq RE$.*
Le problème $\mathbf{P}_{\mathcal{P}}$ suivant est indécidable :

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

Idée de la Démonstration.
Sans perte de généralité, on supposeque $\emptyset \notin \mathcal{P}$. 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 $crazy_{M,w}(x)$
  $M(w)$
  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. M, w instance positive de **Arrêt**
   ssi
   M s'arrête sur w $\quad L(crazy_{M,w}) = \begin{cases} \mathbf{G} \text{ si } M \text{ s'arrête sur } w \\ \emptyset \text{ sinon} \end{cases}$
   ssi
   $L(crazy_{M,w}) \in \mathcal{P}$
   ssi
   $tr(M, w) = crazy_{M,w}$ instance positive de $\mathbf{P}_{\mathcal{P}}$.

■

2.6.2 Exemples d'application

Les problèmes de décision suivants sont indécidables :

Exemple 46 *Avec $\mathcal{P} = \{\emptyset\}$:*

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

Exemple 47 *$\mathcal{P} = \{L \mid \{\langle G \rangle \mid G \text{ est un graphe connexe}\} \subseteq L\}$:*

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.*

2.7 Bilan

Définition 48 (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.

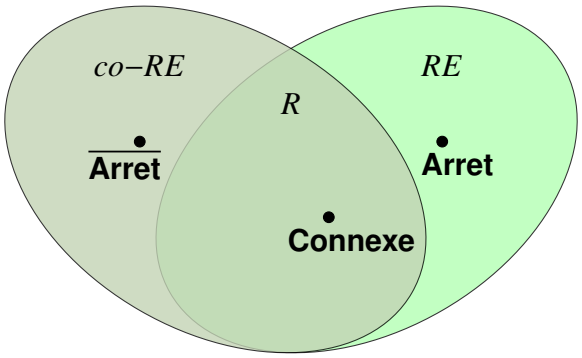
Définition 49 ($co{-}RE$)
$co{-}RE = \{\mathbf{A} \mid \overline{\mathbf{A}} \in RE\}$.

Théorème 50 $RE \cap co{-}RE = R$

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 A' défini ci-dessous décide **A** :

procédure $A'(w)$
Lancer en parallèle $A(w)$ et $\overline{A}(w)$ et s'arrêter dès lors qu'un des proccessus a accepté w
accepter si A a accepté w ; **rejeter** si $\overline{A}$ a accepté w.

■



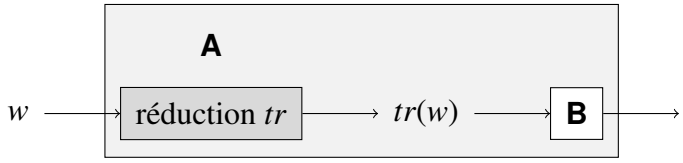
2.8 Notes bibliographiques

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 présentation plus directe. Dans [Sipser, 2006], le problème présenté dans la section "problème de l'arrêt" du livre est en fait le problème de l'acceptation. Le véritable problème de l'arrêt est présenté plus tard dans le livre.

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 Classe P

Définition 51 (classe P)
La classe **P** est la classe des problèmes de décision **A** tels qu'il existe une machine de Turing déterministe M et un polynôme f tels que M décide **A** en temps f.

Thèse Cobham–Edmonds [Arora and Barak, 2009] : **P** = classe des problèmes faciles

3.2 Classe NP

3.2.1 Définition

Définition 52 (classe NP)
La classe **NP** est la classe des problèmes de décision **A** tels qu'il existe une machine de Turing non-déterministe M et un polynôme f telle que M décide **A** en temps f.

3-COLORATION
entrée : Un graphe non orienté $G = (S, A)$;
sortie : oui si G est 3-coloriable ; non sinon.

Proposition 53 ***3-COLORATION*** $\in$ ***NP***.

Idée de la Démonstration.

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

■

3.2.2 Définition alternative : vérifieur par certificat

Définition 54 (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 accepte (w, c).

Définition 55 (certificat)
Un tel c s'appelle **certificat** de l'appartenance de w à **A** ex : une 3-coloration

Proposition 56 *([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 c, la longueur de l'exécution $V(w, c)$ est $\leq f(|w|)$.*

Idée de la Démonstration.

$\Leftarrow$

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

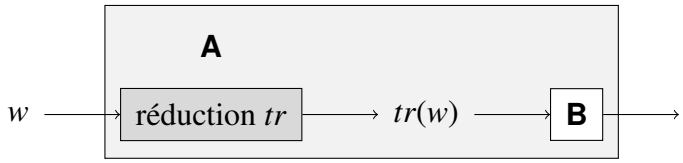
$\Rightarrow$

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

■

3.2.3 Réductions polynomiales

Définition 57 (Réduction)
Une **réduction polynomiale** d'un problème **A** à un problème **B** est une fonction $tr : \Sigma^* \to \Sigma^*$ calculable en temps polynomial telle que pour tout $w \in \Sigma^*$ de **A**, $w \in$ **A** ssi $tr(w) \in$ **B**.



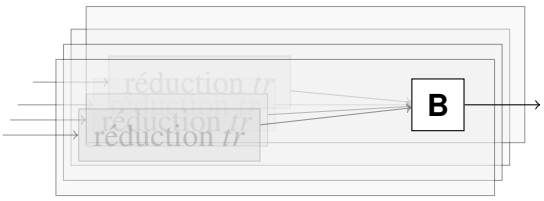
On dit que **A se réduit en temps polynomial** à **B** s'il existe une réduction polynomiale de **A** à **B** (intuitivement, **A** est plus facile que **B**).

Théorème 58 *Si **A** se réduit en temps polynomial **B**, alors :*

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

3.2.4 NP-dureté

Définition 59 (NP-dur)
Un problème est **NP-dur** si tout problème de **NP** s'y réduit en temps polynomial.



Définition 60 (NP-complet)
Un problème est **NP-complet** s'il est dans **NP** et **NP**-dur.

Proposition 61 *Si un problème **NP**-dur est dans **P** alors **P** = **NP**.*

3.3 Théorème de Cook

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

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

Théorème 62 (de Cook) ***SAT*** *et* ***CNF-SAT*** *sont* ***NP****-complets.*

3.3.1 Dans NP

Théorème 63 ***SAT*** *est dans* ***NP****.*

Idée de la Démonstration.

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

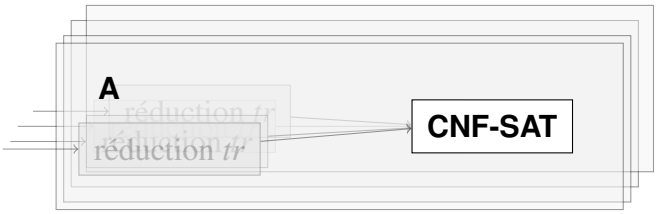
```
procédure sat(φ)
    pour toute proposition atomique p dans φ
        choisir v[p] dans {faux, vrai} ;
    si v ⊨ φ alors
        accepter (gagné)
    sinon
        rejeter (perdu)
```

■

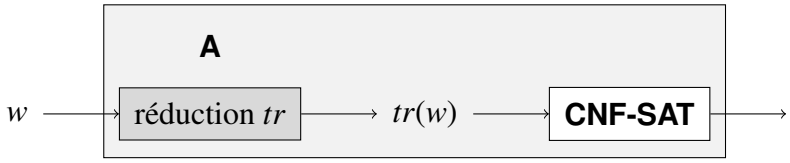
3.3.2 NP-dur

Théorème 64 ***CNF-SAT*** *est* ***NP****-dur.*

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



On va définir une réduction polynomiale tr tel que $w \in$ **A** ssi $tr(w)$ est une formule satisfiable.



Soit $M = (\Sigma, \Gamma, Q, \delta, q_0, q_{acc})$ une machine de Turing non-déterministe qui décide **A** en temps polynomial. Il existe un polynôme f tel que, pour toute entrée w, toutes les exécutions de M soient de longueur au plus $f(|w|)$. On modifie M pour que l'état final acceptant acc est un état sur lequel on boucle.

La formule $tr(w)$ exprime

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

Définition de $tr(w)$. Soit $C = \{0, \ldots, f(|w|)\}$. On introduit les propositions atomiques :

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 $t, i \in C$, $q \in Q$, $a \in \Gamma$ et $\tau \in \delta$.

Seules les $f(|w|)$ cases du ruban sont pertinentes car la tête de lecture ne va jamais au-delà. Définissons $tr(w)$ comme la conjonction les formules 1-17 suivantes.

Unicité et existence des valeurs.

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

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

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

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

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

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

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

8. $\bigwedge_{\tau,\tau' \in \delta \mid \tau \neq \tau'} \left(\neg$ [au temps t, on tire la transition τ] $\vee \neg$ [au temps t, on tire la transition τ'] $\right)$ — 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] $\wedge \ldots$ [au temps 0, la case n° n contient x_n] $\wedge$ [au temps 0, la case n° $n+1$ contient $\sqcup$] $\wedge \ldots$ [au temps 0, la case n° $f(|w|)$ 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. $\boxed{\text{au temps f(|w|), l'état est } q_{acc}}$

La machine atteint l'état d'acceptation q_{acc}

Exécution des transitions

12. $$\bigwedge_{t\in C}\bigwedge_{i\in C}\bigwedge_{a\in\Sigma}\left(\left(\neg\boxed{\begin{array}{l}\text{au temps } t,\\ \text{la position}\\ \text{du curseur est } i\end{array}}\wedge\boxed{\begin{array}{l}\text{au temps } t,\\ \text{la case n° } i\\ \text{contient } a\end{array}}\right)\rightarrow\boxed{\begin{array}{l}\text{au temps } t+1,\\ \text{la case n° } i\\ \text{contient } a\end{array}}\right)$$

on ne change pas le contenu du ruban si le curseur n'y est pas

13. $$\bigwedge_{t\in C\setminus\{f(|w|)\}}\bigwedge_{(q,a,q',b,d)\in\delta}\left(\boxed{\begin{array}{l}\text{au temps } t, \text{ on tire}\\ \text{la transition } (q,a,q',b,d)\end{array}}\rightarrow\boxed{\begin{array}{l}\text{au temps } t,\\ \text{l'état est } q\end{array}}\right)$$

14. $$\bigwedge_{t\in C\setminus\{f(|w|)\}}\bigwedge_{(q,a,q',b,d)\in\delta}\bigwedge_{i\in C}\left[\left(\boxed{\begin{array}{l}\text{au temps } t, \text{ on tire}\\ \text{la transition } (q,a,q',b,d)\end{array}}\wedge\boxed{\begin{array}{l}\text{au temps } t,\\ \text{la position}\\ \text{du curseur est } i\end{array}}\right)\rightarrow\boxed{\begin{array}{l}\text{au temps } t,\\ \text{la case n° } i\\ \text{contient } a\end{array}}\right]$$

15. $$\bigwedge_{t\in C\setminus\{f(|w|)\}}\bigwedge_{(q,a,q',b,d)\in\delta}\left(\boxed{\begin{array}{l}\text{au temps } t, \text{ on tire}\\ \text{la transition } (q,a,q',b,d)\end{array}}\rightarrow\boxed{\begin{array}{l}\text{au temps } t+1,\\ \text{l'état est } q'\end{array}}\right)$$

16. $$\bigwedge_{t\in C\setminus\{f(|w|)\}}\bigwedge_{(q,a,q',b,d)\in\delta}\bigwedge_{i\in C}\left[\left(\boxed{\begin{array}{l}\text{au temps } t,\\ \text{on tire}\\ \text{la transition}\\ (q,a,q',b,d)\end{array}}\wedge\boxed{\begin{array}{l}\text{au temps } t,\\ \text{la position}\\ \text{du curseur est } i\end{array}}\right)\rightarrow\boxed{\begin{array}{l}\text{au temps } t+1,\\ \text{la case n° } i\\ \text{contient } b\end{array}}\right]$$

17. $$\bigwedge_{t\in C\setminus\{f(|w|)\}}\bigwedge_{(q,a,q',b,d)\in\delta}\bigwedge_{i\in C|i+d\in C}\left(\boxed{\begin{array}{l}\text{au temps } t,\\ \text{on tire}\\ \text{la transition}\\ (q,a,q',b,d)\end{array}}\wedge\boxed{\begin{array}{l}\text{au temps } t,\\ \text{la position}\\ \text{du curseur est } i\end{array}}\right)\rightarrow\boxed{\begin{array}{l}\text{au temps } t+1,\\ \text{la position}\\ \text{du curseur est } i+d\end{array}}$$

La formule $tr(w)$ est bien en forme normale conjonctive.

1. $tr(w)$ est calculable en temps polynomial en $|w|$ à partir de w ;
2. $w \in \mathbf{A}$ ssi $tr(w)$ est une formule satisfiable.

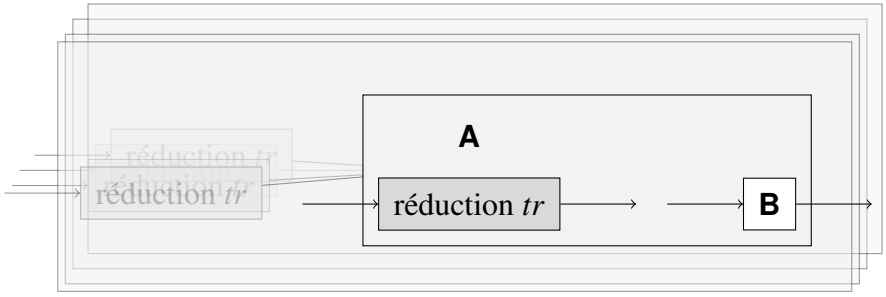
$\boxed{\Rightarrow}$ Soit $w \in \mathbf{A}$. Alors il existe une exécution de M acceptante sur le mot w en temps $f(|w|)$ et sur une longueur de ruban de $f(|w|)$. On construit une valuation qui satisfait $tr(w)$.

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

3.4 Réductions polynomiales pour montrer la NP-dureté

Proposition 65 *Si **A** se réduit polynomialement à **B** et que **A** est NP-dur alors **B** est NP-dur.*

IDÉE DE LA DÉMONSTRATION.



■

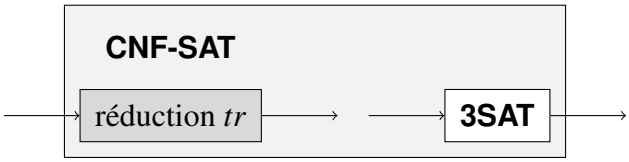
3.4.1 3-SAT

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

Proposition 66 ***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 67 $(a \vee b \vee c \vee d \vee e) \quad \rightsquigarrow \quad (a \vee b \vee \alpha) \wedge (\neg\alpha \vee c \vee \beta) \wedge (\neg\beta \vee d \vee e)$.

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.

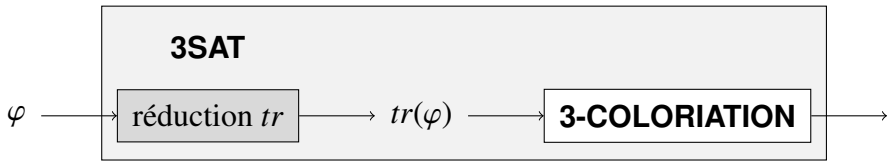
$\boxed{\Rightarrow}$ Supposons que $tr(\varphi)$ soit satisfiable pour une certaine valuation. En particulier, chaque sous-conjunction de clauses $(a \vee b \vee \alpha) \wedge (\neg\alpha \vee c \vee \beta) \wedge (\neg\beta \vee d \vee e)$, obtenu à partir d'une clause de l'instance de SAT est vraie. 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.

■

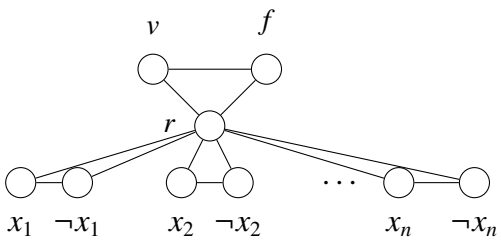
3.4.2 3-coloration

Théorème 68 ***3-COLORIATION*** *est NP-dur.*

IDÉE DE LA DÉMONSTRATION.



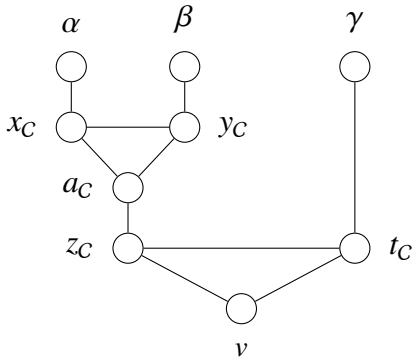
On définit une réduction tr en temps polynomial qui à toute **3SAT**-instance φ associe une **3-COLORIATION**-instance $tr(\varphi)$. $tr(\varphi)$ est un graphe basé sur le **gadget** suivant :



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 C de la forme $(\alpha \vee \beta \vee \gamma)$ de φ, on ajoute le gadget ci-dessous. Pour coder une 2-clause C de la forme $(\alpha \vee \beta)$, on prend le même gadget mais avec f à la place de γ. La fonction tr est calculable en temps polynomial.

Lemme 69 *Dans toute coloration, l'un des sommets α, β, γ a 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 de 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 70 *On peut compléter toute coloriation 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 71 *φ satisfiable ssi $tr(\varphi)$ est 3-coloriable.*

IDÉE DE LA DÉMONSTRATION.
$\boxed{\Rightarrow}$ Supposons que φ est satisfiable et soit v une valuation telle que $v \models \varphi$. On colorie les noeuds de la façon suivante :

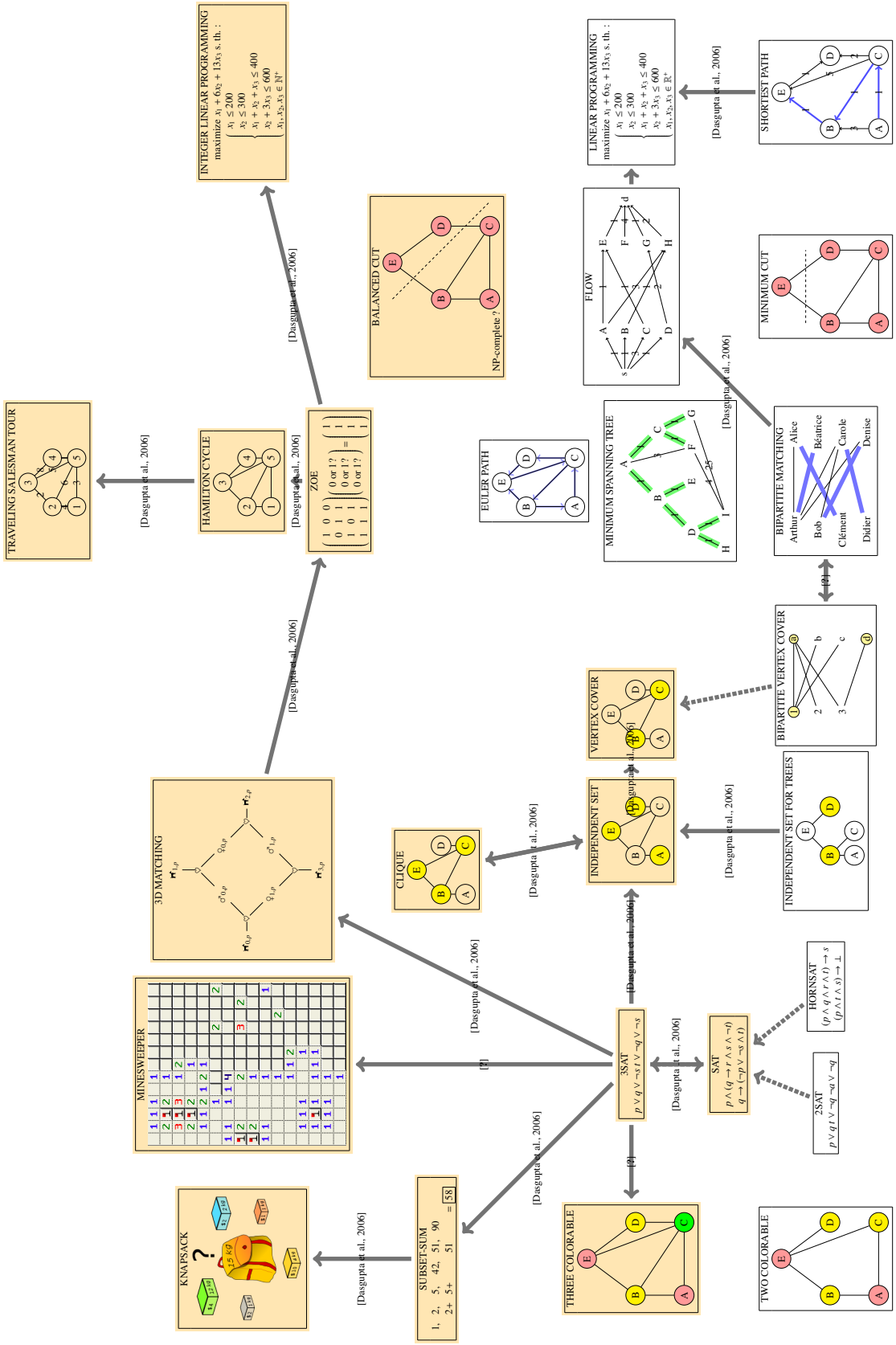
— $c[r] =$; $c[v] =$; $c[f] =$;
— $c[p] =$ si $v[p] = 1$; sinon.
— $c[\neg p] =$ si $v[p] = 1$; sinon.

Par le lemme 70, on complète la coloriation pour tous les gadgets. Donc $tr(\varphi)$ est 3-coloriable.
$\boxed{\Leftarrow}$ Supposons que $tr(\varphi)$ est 3-coloriable. On construit la valuation v :
— $v[p_i] = 1$ si p_i est colorié avec la couleur de v : 0 sinon.
Par le lemme 69, $v \models \varphi$. ■ ■

3.5 Panoramas de problèmes

21 problèmes de Karp : [Karp, 1972], [Dasgupta et al., 2006]

Autre référence : [Garey and Johnson, 1979]



3.5.1 Problèmes dans NP conjecturés ni dans P, ni NP-complets

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.

3.5.2 Démonstration de l'appartenance à P récente

⚠ 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**.

3.6 NP-complétude en pratique

3.6.1 Branch and bound

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

3.6.2 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.

3.6.3 Réduction à SAT ou à la programmation linéaire entière

— Compétitions SAT : `http://www.satcompetition.org/`
— Outil pédagogique pour l'algorithme DPLL :
`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 72 ($TIME(f), NTIME(f)$)
[Sipser, 2006, p. 229] Soit $f : \mathbb{N} \to \mathbb{R}^+$.

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

Définition 73 (décidé en espace f)
M décide L en espace f si M décide L et pour tout w, au plus $f(|w|)$ premières cases du ruban ont été utilisés dans les configurations de l'arbre de calcul depuis $q_0 w$.

Définition 74 ($SPACE(f), NSPACE(f)$)
[Sipser, 2006, p. 308] Soit $f : \mathbb{N} \to \mathbb{R}^+$ telle que $f(n) \geq n$.

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

4.1.2 Classes stables par modèle de calcul

Définition 75 ($P, NP, EXPTIME, NEXPTIME, PSPACE$, **etc.**)

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

Définition 76 (co-)
co- $= \{L \subseteq \Sigma^* \mid \overline{L} \in$ $\}$.

Définition 77 (-dur)
L est -dur ssi tout problème dans se réduit à L en temps polynomial.

Définition 78 (-complet)
L est -complet ssi L est dans et est -dur.

4.2 Théorème de Savitch

Théorème 79 (de Savitch) *[Sipser, 2006, p. 310] Soit $f : \mathbb{N} \to \mathbb{N}$ 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 son entrée : on note c_{accept} cette configuration acceptante. Voici un algorithme déterministe qui décide L :

```
procédure deciderL(w)
    si q0w →* c_accept dans le graphe des configurations de M
        accepter
    sinon
        rejeter
```

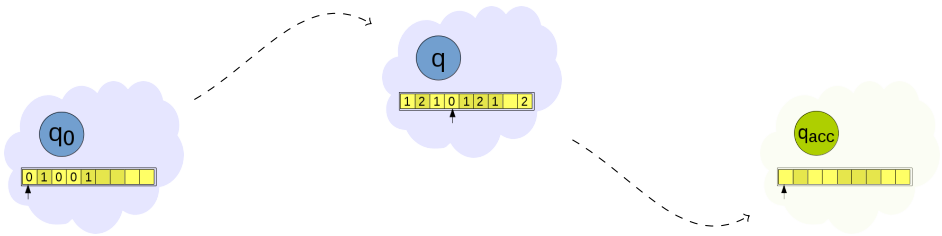
Lemme 80 *Il existe $d \in \mathbb{N}$ t.q. le nombre de config. accessibles depuis q_0w est majoré par $2^{df(|w|)}$.*

Idée de la Démonstration.
Le nombre de configurations accessibles depuis q_0w est majoré par $|Q| \times |\Sigma|^{f(|w|)} \times f(|w|)$. ■

Lemme 81 $q_0w \to^* c_{accept}$ *ssi* $q_0w \to^{\leq 2^{df(|w|)}} c_{accept}$

On implémente le test $q_0w \to^* c_{accept}$ avec chemin ?$(q_0w, c_{accept}, 2^{df(|w|)})$ où chemin ? est :

```
fonction chemin ?(c1, c2, t)
    si t = 1
        retourner (c1 →^≤1 c2)
    sinon
        pour toute config. c de M de taille de ruban au plus f(|w|)
            si chemin ?(c1, c, t/2) et chemin ?(c, c2, t/2) alors
                retourner vrai
        retourner faux
```



La fonction chemin ? suit **diviser pour régner**. La complexité spatiale de chemin ?(c_1, c_2, t) est $C(t) = O(f(|w|)) + C(\frac{t}{2})$, c'est à dire $C(t) = O(f(|w|)log_2 t)$. D'où une complexité spatiale pour deciderL(w) de

$$\underbrace{O(f(|w|)}_{\text{calcul de } f(|w|)} + \underbrace{C(2^{df(|w|)})}_{O(f(|w|)^2)} = O(f|w|)^2.$$

■

Corollaire 82 *PSPACE = NPSPACE.*

1. $f(n) \geq n$ car on doit comptabiliser au moins la taille de l'entrée. Le théorème fonctionne aussi pour $f(n) \geq \log n$ maisil faut alors utiliser des machines de Turing à plusieurs rubans.

4.3 PSPACE

4.3.1 Rappel : SAT et VALIDE

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

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 v, $v \models \varphi$.

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

est co-**NP**-complet.

4.3.2 QBF : formules booléennes quantifiées

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

Syntaxe

Définition 83 (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 84 (conditions de vérité)
- $v \models \chi$ comme en logique propositionnelle si χ est propositionnelle ;
- $v \models \exists p\varphi$ ssi $v[p := 0] \models \varphi$ ou $v[p := 1] \models \varphi$;
- $v \models \forall p\varphi$ ssi $v[p := 0] \models \varphi$ et $v[p := 1] \models \varphi$.

Si φ est close, $v \models \varphi$ ne dépend pas de v et on dit que φ est **vraie** s'il existe v t.q. $v \models \varphi$.

Problème de décision

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

Théorème 85 *[Sipser, 2006](p. 285-287)* ***TQBF*** *est dans PSPACE.*

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

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

■

Théorème 86 ***TQBF*** *est NPSPACE-dur.*

IDÉE DE LA DÉMONSTRATION.
Soit L un problème NPSPACE. On réduit L à **TQBF** en temps polynomial.

on reprend la démonstration du théorème de Savitch...

Pour toute instance w de L, on crée une formule booléenne quantifiée $tr(w)$ qui exprime

'il existe un chemin de $q_0 w$ à c_{accept} de longueur au plus $2^{df(|w|)}$'

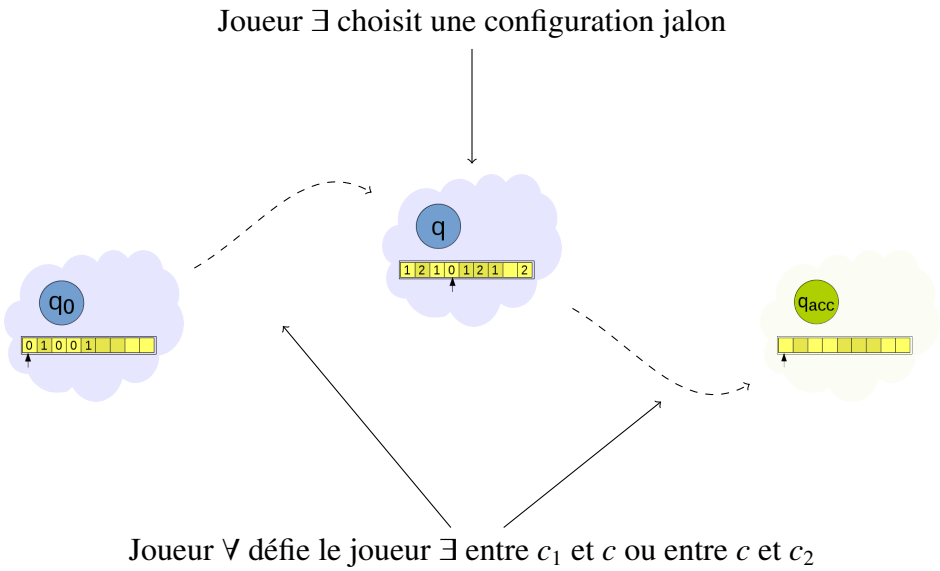
$$tr(w) := (\vec{c}_{ini} = \text{ config initiale avec } w) \wedge (\vec{c}_{acc} = \text{ config acceptante}) \wedge ch?(\vec{c}_{ini}, \vec{c}_{acc}, 2^{df(|w|)}).$$

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 $k > 0$,

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

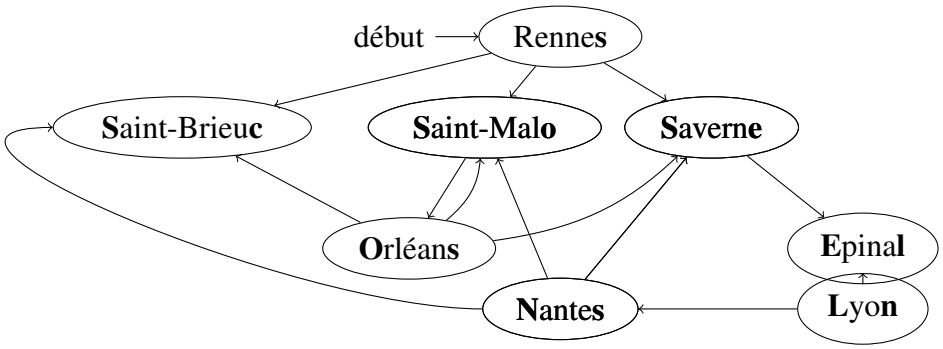


Par construction, $w \in L$ ssi $tr(w)$ est QBF-vraie. On peut écrire un algorithme qui calcule $tr(w)$ en temps polynomial en $|w|$. ■

4.3.3 Jeux à deux joueurs

On utilise **TQBF** pour montrer la PSPACE-dureté de problèmes :
— model checking de la logique du premier ordre ;
— Jeu de géographie généralisé, etc., puis le Go ;
— Reversi, Hex, etc.
Á l'inverse, on modélise des jeux à deux joueurs (exemple : les échecs) avec **TQBF** ([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. Tour à tour, chaque joueur supprime le sommet où il y a le jeton puis déplace le jeton 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.

Proposition 87 ***GEOGRAPHIE*** *est dans PSPACE.*

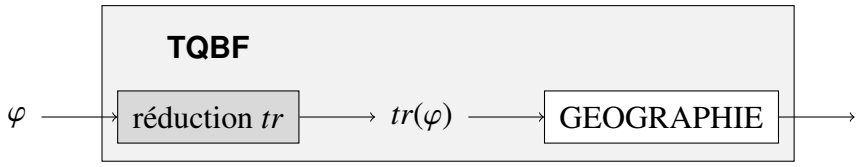
Idée de la Démonstration.

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

■

Proposition 88 ***GEOGRAPHIE*** *est PSPACE-dur.*

Idée de la Démonstration.



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

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

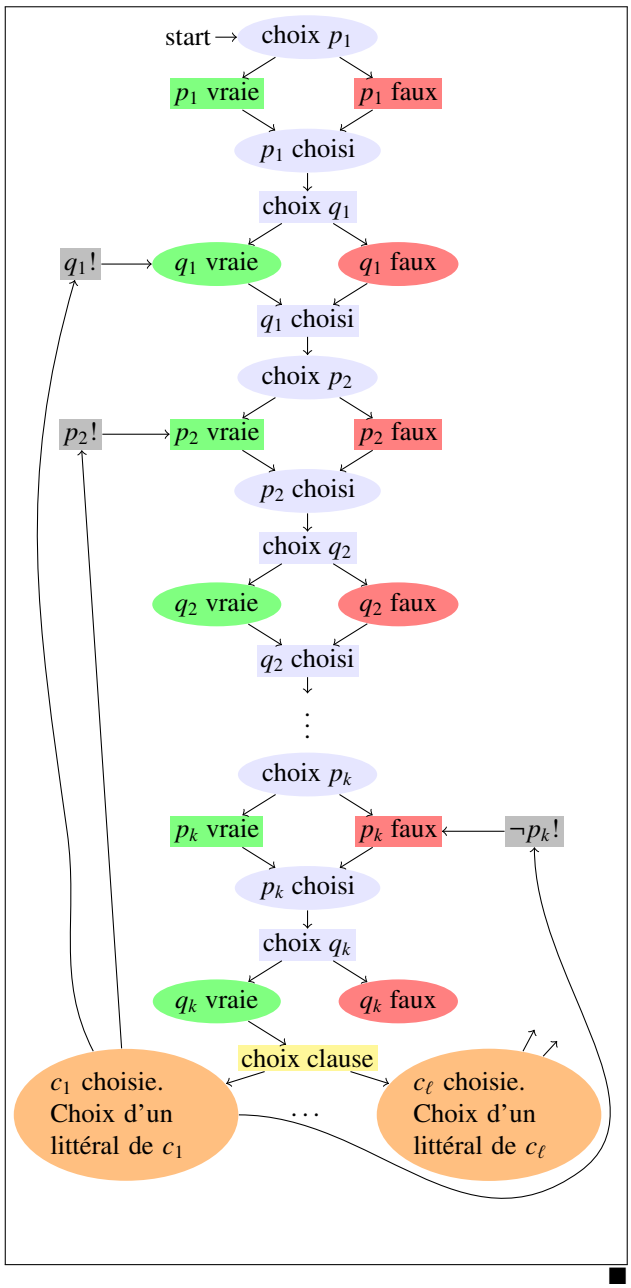
où ψ est une forme normale conjonctive, c'est à dire

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

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 $tr(\varphi)$.



■

4.4 Universalité d'un langage rationnel

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

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

Théorème 89 ***UNIVERSALITE*** *est dans PSPACE.*

Idée de la Démonstration.
Voici une machine qui décide $\overline{\textbf{UNIVERSALITE}}$ en espace polynomial :

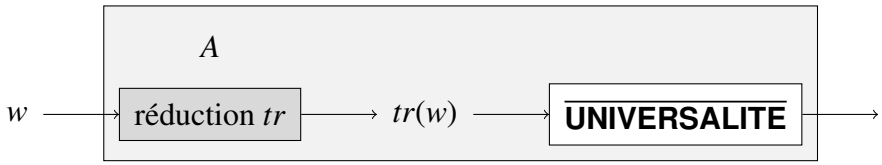
```
procédure nonuniverselle ?(e : expression rationnelle)
    𝒜 := construire un automate non-déterministe tel que L(𝒜) = L(e) ;
    S := clotûre de {état initial de 𝒜}
    tant que S contient un état final
        choisir lettre a
        S := cloture des a-successeurs des états dans S

    accepter
```

■

Théorème 90 ***UNIVERSALITE*** *est PSPACE-dur.*

Idée de la Démonstration.
Soit $A \in$ PSPACE. On réduit A à $\overline{\textbf{UNIVERSALITE}}$ en temps polynomial.



Soit M qui décide A en espace polynomial. Soit f un tel polynôme. Idée :

$$L(tr(w)) = \{\text{mots ne représentant pas une exécution acceptante de } M(w)\}$$

.

1. tr calculable en temps polynomial ;
2. $w \in A$ ssi $L(tr(w)) \neq \Omega^*$ où Ω est un alphabet défini plus bas.

Représentation des exécutions acceptantes. Un mot de la forme

$$\square C_1 \square C_2 \square \ldots\ldots \square C_k \square$$

représente une exécution de M, avec $k \geq 1$, les C_i sont des mots de longueur $N = f(|w|)$ représentant les configurations, et $\square$ est un symbole frais séparant les mots C_i.

Exemple 91 $\square \begin{bmatrix} a \\ q_0 \end{bmatrix} bc\sqcup\sqcup \square b \begin{bmatrix} b \\ q \end{bmatrix} c\sqcup\sqcup \square ba \begin{bmatrix} c \\ q' \end{bmatrix} \sqcup\sqcup \square \ldots\ldots \square \begin{bmatrix} a \\ q_{acc} \end{bmatrix} aad\sqcup \square$

Notations.

Γ = alphabet du ruban

C = alphabet des couples $\left[\begin{array}{l}\text{lettre du ruban}\\ \text{état}\end{array}\right]$

A = alphabet des couples $\left[\begin{array}{l}\text{lettre du ruban}\\ q_{acc}\end{array}\right]$

$\Delta = \Gamma \cup C$

$\Omega = \Delta \cup \{$ $\}$

Définition de $tr(w)$. L'expression rationnelle $tr(w)$ est l'union des expressions suivantes, chacune spécifiant une malformation dans un mot censé représenter une exécution acceptante :

$\Delta\Omega^*$	Ne commence pas avec
$\Omega^*\Delta$	Ne termine pas avec
Ω^* Γ^* Ω^*	Configuration sans curseur
Ω^* $\Delta^*C\Delta^*C\Delta^*$ Ω^*	Configuration avec au moins 2 curseurs
Ω^* Ω^*	Configuration avec un ruban de longueur 0
Ω^* Δ Ω^*	Configuration avec un ruban de longueur 1
Ω^* $\Delta\Delta$ Ω^*	Configuration avec un ruban de longueur 2
$\vdots$	$\vdots$
Ω^* Δ^{N-1} Ω^*	Configuration avec un ruban de longueur $N-1$
Ω^* $\Delta^{N+1}\Delta^*$ Ω^*	Configuration avec un ruban de longueur $\geq N+1$
$(\Delta \setminus \{\left[\begin{array}{c} w_1 \\ q_0 \end{array}\right]\}\Delta^*$ Ω^*	1ère case du ruban non conforme à la configuration initiale
$(\Delta^{i-1}(\Delta\setminus\{w_i\})\Delta^*$ Ω^* où $w_i = \sqcup$ si $i > \lvert w\rvert$	i^e case du ruban non conforme à la configuration initiale
$(\Omega \setminus A)^*$	Pas d'états acceptants
$\Omega^*\alpha_1\alpha_2\alpha_3\Omega^{N-1}(\Omega \setminus f(\alpha_1\alpha_2\alpha_3))\Omega^*$	Transition non conforme

où f sont des fonctions qui correspondent aux transitions :

α_1	α_2	α_3
	$f(\alpha_1\alpha_2\alpha_3)$	

Exemple 92

— $f($ $, \left[\begin{array}{c} a \\ q_0 \end{array}\right], b) = c$ *si la transition depuis l'état q_0 en lisant a écrit c ;*

— $f(\left[\begin{array}{c} a \\ q_0 \end{array}\right], b, d) = \left[\begin{array}{c} b \\ q \end{array}\right]$ *si la transition depuis q_0 en lisant a déplace le curseur à droite et va dans q ;*

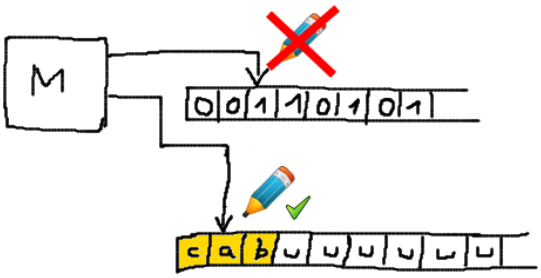
— $f(b, c, d) = c$.

■

4.5 LOGSPACE et NLOGSPACE

4.5.1 Définitions

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



Définition 93 (L et NL)

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

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 94 *([Papadimitriou, 2003], example 2.10 p. 48-49)* ***ACCESSIBILITE*** *est dans* ***NL****.*

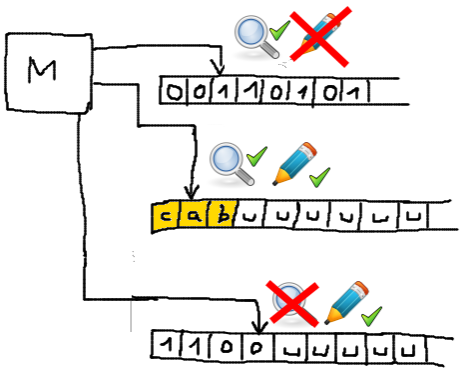
Idée de la Démonstration.

```
fonction path?(G, s, t)
    tant que s ≠ t
        s := choisir un successeur de s
    accepter
```

■

4.5.3 NL-complétude

Définition 95 (réduction en espace logarithmique)
Une **réduction en espace logarithmique** de A à B est une fonction $tr : \Sigma^* \to \Sigma^*$ telle que :
— $w \in A$ ssi $tr(w) \in B$.
— tr calculable en espace logarithmique, i.e. il existe une machine de Turing à trois rubans :
  — le ruban d'entrée en lecture seule contenant w ;
  — un ruban de travail en lecture/écriture qui contient au plus $O(\log |w|)$ symboles ;
  — un ruban de sortie en écriture seule sur lequel est écrit $tr(w)$ à la fin de l'exécution.



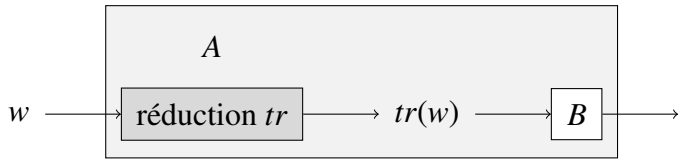
Proposition 96 *Une réduction en espace logarithmique est une réduction en temps polynomial.*

Définition 97 (NL-complet)
A est **NL**-complet ssi $A \in$ **NL** et tout problème de **NL** se réduit en espace logarithmique à A.

Théorème 98

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

Idée de la Démonstration.
Le schéma suivant ne donne pas directement un algorithme pour A en espace $O(log|w|)$:



En voici un : lancer l'algorithme pour B avec un ruban d'entrée $tr(w)$ virtuel : quand on a besoin du $i^{\text{ème}}$ symbole du mot $tr(w)$ on appelle la machine qui calcule tr. ■

Théorème 99 *S'il existe un problème **NL**-complet dans **L**, alors **L** = **NL**.*

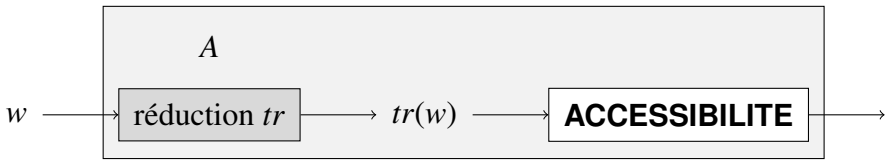
Théorème 100 ***ACCESSIBILITE*** *est **NL**-complet.*

Idée de la Démonstration.
Dans NL cf proposition 94.
NL-dur Soit A un problème **NL**. Il existe une machine non-déterministe M (à deux rubans) qui décide A en espace $O(\log n)$. Sans perte de généralité, on suppose que M n'a qu'une seule configuration acceptante. On construit une réduction tr en espace logarithmique de A vers **ACCESSIBILITE** : $tr(w) := (G_w, s_w, t_w)$ où est le graphe des configurations de la machine M sur l'entrée w, s_w la configuration initiale de M avec w sur le ruban d'entrée, et t_w l'unique configuration finale acceptante de M. On a $w \in A$ ssi $tr(w) \in$ **ACCESSIBILITE**. ■

4.5.4 NL ⊆ P

Théorème 101 *NL ⊆ **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 la réduction est aussi en temps polynomial, A est dans **P**. ■

4.5.5 Théorème de Immerman-Szelepcsenyi : NL = co-NL

Théorème 102 $\overline{\textbf{ACCESSIBILITE}}$ *est dans NL.*

Idée de la Démonstration.

Soit A_ℓ l'ensemble des sommets accessibles depuis s en au plus i étapes.
Soit $A := A_{|G|}$ est l'ensemble des sommets accessibles depuis s.

```
procédure path̄?(G, s, t)
    i := 0                                          i compte A
    pour tout sommet u ≠ t de G                     test si u ∈ A
        choisir téméraire ∈ {0, 1}                  tentons-nous le test u ∈ A ?
        si téméraire = 1
            path?(G, s, u)                          peut échouer si trop téméraire
            i := i + 1                              u ∈ A trouvé
    si i ≠ #(G, s) alors rejeter                    si peu téméraire ou t ∈ A, rejet
    accepter                                        t ∉ A
```

où # est une fonction non-déterministe telle que :
— $\#(G, s)$ échoue ou retourne le nombre de sommets accessibles depuis s dans G ;
— il existe au moins une exécution de $\#(G, s)$ qui n'échoue pas.

Lemme 103 *Pas de chemin de s à t dans G ssi $\overline{path}$? accepte (G, s, t).*

```
fonction #(G, s)                                    retourne |A|
    n = 1                                           |A_0| = 1
    pour ℓ := 1 à |G|                               calcul de |A_ℓ| à partir de |A_{ℓ−1}|
        n := #(G, s, ℓ, n)
    retourner n                                     |A_{|G|}|
```

```
fonction #(G, s, ℓ, n)                              retourne |A_ℓ| en sachant que n = |A_{ℓ−1}|
    k := 0                                          k compte A_ℓ
    pour tout sommet v de G                         test si v ∈ A_ℓ
        j := 0                                      j recompte A_{ℓ−1}
        preuve_que_v_dedans := false                booléen vraie si preuve de v ∈ A_ℓ trouvé
        pour tout sommet u de G                     test si u ∈ A_{ℓ−1}
            choisir téméraire ∈ {0, 1}              tentons-nous le test u ∈ A_{ℓ−1}
            si téméraire = 1
                path?(G, s, u, ℓ − 1)
                j := j + 1                          u ∈ A_{ℓ−1} trouvé
                si u →^G v
                    preuve_que_v_dedans := true
        si j ≠ n alors rejeter                      si peu téméraire sur le comptage de A_{ℓ−1} alors rejet
                                                    ici : preuve que v ∈ A_ℓ ssi v ∈ A_ℓ
        si preuve_que_v_dedans alors k := k + 1     si v ∈ A_ℓ trouvé, on le compte
    retourner k                                     |A_ℓ|
```

où $path?(G, s, u, \ell - 1)$ a une exécution qui n'échoue pas ssi $s \rightarrow^{\leq \ell - 1} u$

■

Corollaire 104 *NL = co-NL. [Sipser, 2006][p. 331]*

4.5.6 2SAT

Proposition 105 *2SAT ∈ **NL**.*

Idée de la Démonstration.
Comme co-**NL** = **NL**, il suffit de montrer que $\overline{2SAT} \in$ **NL** :

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 $\ell \to \ell'$ dans φ
    $\ell := \ell'$
  tant que $\ell \neq p$
    choisir une clause de la forme $\ell \to \ell'$ dans φ
    $\ell := \ell'$
  accepter

■

Proposition 106 *([Papadimitriou, 2003], p. 398, Th. 16.3) 2SAT est **NL**-dur.*

Idée de la Démonstration.

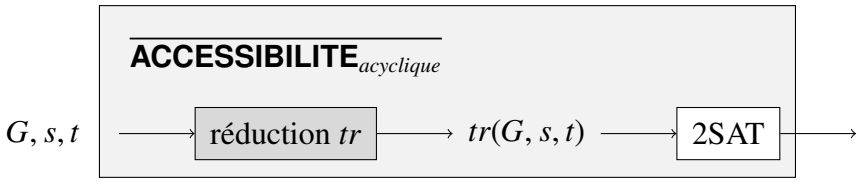
ACCESSIBILITE$_{acyclique}$
entrée : Un graphe G **acyclique**, s, t
sortie : oui, s'il existe un chemin de s à t dans G ; non, sinon.

Lemme 107 ***ACCESSIBILITE***$_{acyclique}$ *est **NL**-complet.*

Idée de la Démonstration.
Dans **NL** cf proposition 94.
NL-dur Reprendre la démonstration du théorème 100 en supposons que la machine M ne boucle pas (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 :



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

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

Définition 109 (P-complet)
Un problème est **P**-complet s'il est dans **P** et est **P**-dur.

Proposition 110 *S'il existe un problème **A** qui est **P**-dur et dans **NL**, alors **P** = **NL**.*

Théorème 111 *Le problème suivant est **P**-complet ([Papadimitriou, 2003], 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 évalue les sorties des portes une à une.
P-dur On code l'exécution de longueur polynomiale d'une machine comme un circuit.

■

Théorème 112 *HORN-SAT est **P**-complet.*

Proposition 113 *dans **P** Algorithme glouton (cf. [Dasgupta et al., 2006]).*
__P__-dur La réduction du théorème de Cook est en espace logarithmique et si M est déterministe alors la formule est une formule de Horn.

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 Schémas primitifs

Fonction nulle

fonction $\mathbb{O}()$
| **retourner** 0

Fonction successeur

fonction $\sigma(x)$
| **retourner** $x+1$

Fonction projection

fonction $\pi_{\mathbf{i}}^{\mathbf{n}}(x_1, \ldots, x_n)$
| **retourner** x_i

Composition de $\mathbf{F}$ avec $\mathbf{G_1}, \ldots, \mathbf{G_n}$

fonction $(\vec{x})$
| **retourner** $\mathbf{F}(\mathbf{G_1}(\vec{x}), \ldots, \mathbf{G_n}(\vec{x}))$

Récursivité avec $\mathbf{F}$ et $\mathbf{G}$

fonction $r(\vec{x}, k)$
| **si** $k = 0$
| | **retourner** $\mathbf{F}(\vec{x})$
| **sinon**
| | **retourner** $\mathbf{G}(\vec{x}, k-1, r(\vec{x}, k-1))$

5.1.2 Syntaxe d'un langage de programmation fonctionnelle

Définition 114 (Langages des expressions des fonctions primitives récursives)
Le langage $\mathcal{L}_{PR}$ 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 ;
- $\pi_{\mathbf{i}}^{\mathbf{n}}$ où $n \in \mathbb{N}^*$ et $i \in \{1, \ldots, n\}$ est d'arité n ;
- Si $\mathbf{F}$ est une expression d'arité n et $\mathbf{G_1}, \ldots, \mathbf{G_n}$ sont des expressions d'arité ℓ alors :
  $\circ(\mathbf{F}, \mathbf{G_1}, \ldots, \mathbf{G_n})$ est une expression d'arité ℓ ;
- Si $\mathbf{F}$ est une expression d'arité n et $\mathbf{G}$ est une expression d'arité $n+2$ alors
  $\mathbf{rec}\,(\mathbf{F}, \mathbf{G})$ est une expression d'arité $n+1$.

5.1.3 Sémantique

Définition 115 (sémantique d'une expression)
On définit la fonction $[\![\ ]\!] : \mathcal{L}_{PR} \to \bigcup_{k\in\mathbb{N}} \mathcal{F}(\mathbb{N}^k, \mathbb{N})$ par induction structurelle :

— $[\![\mathbb{O}]\!] = \begin{array}{ccc} \mathbb{N}^0 & \to & \mathbb{N} \\ / & \mapsto & 0 \end{array}$;

— $[\![\sigma]\!] = \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ x & \mapsto & x+1 \end{array}$;

— $[\![\pi_{\mathbf{i}}^{\mathbf{n}}]\!] = \begin{array}{ccc} \mathbb{N}^n & \to & \mathbb{N} \\ (x_1, \ldots, x_n) & \mapsto & x_i \end{array}$;

— Si $\mathbf{F}$ est une expression d'arité n et $\mathbf{G_1}, \ldots, \mathbf{G_n}$ sont des expressions d'arité ℓ alors

$$[\![\circ(\mathbf{F}, \mathbf{G_1}, \ldots, \mathbf{G_n})]\!] = \begin{array}{ccc} \mathbb{N}^\ell & \to & \mathbb{N} \\ \vec{x} & \mapsto & [\![\mathbf{F}]\!](\vec{x}, [\![\mathbf{G_1}]\!](\vec{x}), \ldots, [\![\mathbf{G_n}]\!](\vec{x})) \end{array} ;$$

— Si $\mathbf{F}$ est une expression d'arité n et $\mathbf{G}$ est une expression d'arité $n+2$ alors
$[\![\mathbf{rec}\,(\mathbf{F}, \mathbf{G})]\!] = g$ où $g : \mathbb{N}^{n+1} \to \mathbb{N}$ est la fonction définie par récurrence par :

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

Exemple 116 $[\![\circ(\sigma, \pi_3^3)]\!] : \begin{array}{ccc} \mathbb{N}^3 & \to & \mathbb{N} \\ (x_1, x_2, x_3) & \mapsto & x_3 + 1. \end{array}$

Exemple 117 $[\![\mathbf{rec}\left(\pi_1^1, \circ(\sigma, \pi_3^3)\right)]\!] = + : \begin{array}{ccc} \mathbb{N}^2 & \to & \mathbb{N} \\ (x, y) & \mapsto & x + y \end{array}$. *En effet :*

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

Remarque 118 *On construit la fonction nulle d'arité 1 comme suit :* $\mathbb{O}^1 = \mathbf{rec}\left(\mathbb{O}, \pi_2^2\right)$. *Puis celle d'arité* $k \geq 2$ *:* $\mathbb{O}^{\mathbf{k}} = \circ(\mathbb{O}^1, \pi_1^{\mathbf{k}})$. *Par simplicité, on les note toutes* $\mathbb{O}$ *quelque soit l'arité.*

Définition 119 (fonction récursive primitive)
Une fonction $\mathbb{N}^k \to \mathbb{N}$ est **récursive primitive** si une expression de $\mathcal{L}_{PR}$ la dénote. On note *PR* l'ensemble des fonctions récursives primitives.

Proposition 120 *L'ensemble des fonctions récursives primitives est dénombrable.*

Idée de la Démonstration.
Car le langage $\mathcal{L}_{PR}$ est dénombrable. ■

5.2 Exemples de fonctions récursives primitives

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 121 (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 122 *Pour tout n, la fonction $\uparrow^n$ est récursive primitive.*

5.2.1 Prédicats

Définition 123 (prédicat)
Une fonction est appelée **prédicat** si elle est à valeur dans $\{0, 1\}$.

$<$ et
estpair
non
$=$ $\leq$

Proposition 124 *Si* **cond** *est un prédicat,* **F**, **G** *sont des fonctions, tous-tes d'arité ℓ et p.r.*

alors

```
fonction (x⃗)
    si cond(x⃗) alors
        retourner F(x⃗)
    sinon
        retourner G(x⃗)
```

est primitive récursive.

Idée de la Démonstration.
∘(**plus**, ∘(**mult**, **cond**, **F**), ∘(**mult**, ∘(**not**, **cond**), **G**)). ■

5.2.2 Minimisation bornée

Définition 125 (minimisation bornée)
Soit f un prédicat d'arité $\ell + 1$. La fonction suivante s'appelle **minimisation bornée** de f :

```
fonction (x⃗, n)
    pour i := 0 à n
        si f(x⃗, i) ≠ 0 alors retourner i
    retourner n + 1
```

noté $\mu i \leq n.f(\vec{x}, i)$

Proposition 126 *La minimisation d'un prédicat p. r. est aussi p. r..*

5.3 Codage par un entier

Définition 127 (fonction récursive primitive à valeurs dans $\mathbb{N}^q$)
On dit qu'une fonction $\mathbf{g} : \mathbb{N}^\ell \to \mathbb{N}^q$ est **récursive primitive**
si pour tout $i \in \{1, \ldots, q\}$, $\mathbf{g}_i$ est récursive primitive.

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

$$\begin{tikzcd}
\begin{pmatrix} x \\ y \end{pmatrix} \arrow[r, "\varphi"] \arrow[r, bend left=60, "X"] & \varphi(x,y) \arrow[l, bend left=60, "Y"]
\end{tikzcd}$$

Théorème 128 ([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.
$\boxed{\text{Définition}}$ Soit $\varphi(x, y) = \underbrace{2^x \times (2y + 1)}_{\psi(x,y)} - 1$.

$\boxed{\varphi \text{ est récursive primitive}}$ ... comme composition de fonctions qui le sont.

$\boxed{\varphi \text{ est une bijection}}$ 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. On note la réciproque de ψ est $\psi^{-1}(n) = (X(n), Y(n))$. Ainsi, φ est une bijection de $\mathbb{N}^2$ dans $\mathbb{N}$.

$\boxed{\varphi^{-1} \text{ est récursive primitive}}$

1. $X(n)$ est le nombre de divisions par 2 à partir de n jusqu'à obtenir un entier impair.

$$D(n) = \begin{cases} \frac{n}{2} \text{ si } n \text{ est pair} \\ \text{n si } n \text{ est impair} \end{cases}$$

   On calcule n, puis $D(n)$, puis $D^2(n)$, etc. puis on s'arrête lorsque $D^k(n) = D^{k+1}(n)$.

$$interD(n, k) = \begin{cases} n \text{ si } k = 0 \\ D(interD(n, k-1)) \text{ sinon} \end{cases}$$

   On a :

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

2. En itèrant trop (n fois par exemple), $interD(n, k)$ vaut $2Y(n) + 1$. Ainsi :

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

■

5.3.2 Application 1 : suites définies par récurrence

Exemple 129 *La fonction Fibonacci* fib *est définie par :*
- $\mathsf{fib}(0) = \mathsf{fib}(1) = 1$;
- $\mathsf{fib}(k+2) = \mathsf{fib}(k+1) + \mathsf{fib}(k)$.

Définition 130 (schéma de récurrence multiple)
Si $\mathbf{g} : \mathbb{N}^{\ell} \to \mathbb{N}^2$, $\mathbf{h} : \mathbb{N}^{\ell+2+1} \to \mathbb{N}^2$,
le schéma de récursivité multiple définit la fonction : $\mathbf{f} : \mathbb{N}^{\ell+1} \to \mathbb{N}^2$:

$$\mathbf{f}(\vec{x}, k) = \begin{cases} \mathbf{g}(\vec{x}) & \text{si } k = 0 \\ \mathbf{h}(\vec{x}, k-1, \mathbf{f}(\vec{x}, k-1)) & \text{si } k > 0. \end{cases}$$

Exemple 131 *On pose* $\mathbf{f}(k) = (\mathsf{fib}(k), \mathsf{fib}(k+1))$. *On a alors :*

$$\mathbf{f}(0) = \begin{cases} (1,1) & \textit{si } k = 0 \\ \mathbf{h}(k-1, \mathbf{f}(k-1)) & \textit{si } k > 0. \end{cases}$$

où $\mathbf{h}(k, \vec{x}) = (x_2, x_1 + x_2)$.

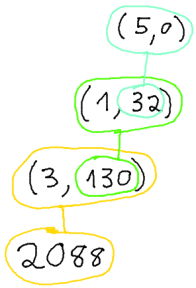
Proposition 132 *La classe des fonctions primitives récursives est stable par schéma de récursivité multiple.*

Idée de la Démonstration.
On pose :

$$\tilde{f}(\vec{x}, k) = \begin{cases} \varphi(\mathbf{g}(\vec{x})) & \text{si } k = 0 \\ \varphi(\mathbf{h}(\vec{x}, k-1, \underbrace{X(\tilde{f}(\vec{x}, k-1)), Y(\tilde{f}(\vec{x}, k-1))}_{\mathbf{f}(\vec{x}, k-1)})) & \text{si } k > 0. \end{cases}$$

■

5.3.3 Application 2 : 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

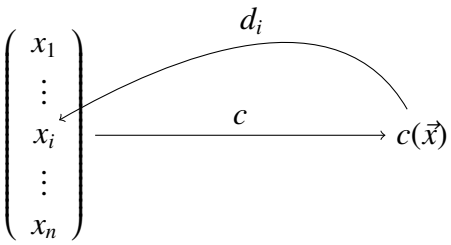
Si x est un entier et ℓ est un entier représentant une liste :
- $\mathit{listevide}() = 0$;
- $\mathit{cons}(x, \ell) = 1 + \varphi(x, \ell)$;
- $\mathit{tete}(\ell) = X(\ell - 1)$;
- $\mathit{queue}(\ell) = Y(\ell - 1)$.

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 $tr(P)$
$x_i := F(\vec{x})$	$\circ(\mathbf{c}, \mathbf{d_1}, \ldots, \mathbf{d_{i-1}}, \circ(\mathbf{F}, \mathbf{d_1}, \ldots, \mathbf{d_n}), \mathbf{d_{i+1}}, \ldots, \mathbf{d_n})$
$bloc_1; bloc_2$	$\circ(tr(bloc2), tr(bloc1))$
si $x_i \neq 0$ **alors** | $bloc$	${}^{\text{ifthen}}_{\text{else}}(\circ(\mathbf{notzero}, \mathbf{d_i}), tr(bloc), \pi^1_1)$
répéter x_i fois | $bloc$	$\circ(\mathbf{rec}\left(\pi^1_1, \circ(tr(bloc), \pi^3_3)\right), \pi^1_1, \mathbf{d_i})$

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

$$\circ(\mathbf{d_1}, \quad \circ(tr(bloc), \ \circ(\mathbf{c}, \pi^1_1, \mathbb{O}^1, \ldots, \mathbb{O}^1)))$$

.

5.5 Limite des fonctions récursives primitives

5.5.1 Preuve via un argument diagonal

Théorème 133 *Il existe une fonction non p. r. mais calculable par une machine de Turing.*

Idée de la Démonstration.
Soit $(e_n)_{n\in\mathbb{N}}$ une **énumération effective** des expressions de $\mathcal{L}_{PR}$ d'arité 1.

fonction $g(n)$
  Énumérer des expressions $e_1, e_2, \ldots$ jusqu'à obtenir e_n ;
  Interpréter l'expression e_n pour calculer $[\![e_n]\!](n)$
  retourner $[\![e_n]\!](n) + 1$

La fonction g n'est pas p. r. Par l'absurde, si elle l'était, il existe k tel que $g = [\![e_k]\!]$.
Ainsi, $g(k) = [\![e_k]\!](k) = [\![e_k]\!](k) + 1$. Contradiction.

■

5.5.2 Fonction d'Ackermann-Péter

Théorème 134 $\begin{array}{rccc} f: & \mathbb{N}^3 & \to & \mathbb{N} \\ & (a,b,n) & \mapsto & a \uparrow^n b \end{array}$ *n'est pas primitive récursive.*

Pour simplifier, supprimons a comme argument :

Définition 135 (fonction Ackermann-Péter)
$A : \mathbb{N}^2 \to \mathbb{N}$ définie par inductivement [1] :
— $A(0,m) = m+1$
— $A(k+1,0) = A(k,1)$
— $A(k+1,m+1) = A(k, A(k+1,m))$

Proposition 136 $A(k,n) = 2 \uparrow^{k-2} (n+3) - 3$.

Proposition 137 *A est calculable.*

Théorème 138 *A n'est pas primitive récursive.*

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.

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

■

5.6 Fonctions μ-récursives partielles

5.6.1 Minimisation non bornée

Définition 139 (minimisation non bornée)
Soit f un prédicat d'arité $\ell + 1$.
La fonction partielle suivante s'appelle **minimisation non bornée** de f :

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

$\mu i. f(\vec{x}, i)$

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

5.6.2 Syntaxe

Définition 140 (Langages des expressions des fonctions μ-récursives)
Le langage $\mathcal{L}_{\mu R}$ des **expressions des fonctions μ-récursives** est défini par induction :

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

5.6.3 Sémantique

Définition 141 (sémantique)
On définit la fonction $[\![\ ]\!] : \mathcal{L}_{\mu R} \to \bigcup_{k \in \mathbb{N}} \mathcal{F}_{partielle}(\mathbb{N}^k, \mathbb{N})$ par induction structurelle :

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

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

- Si $\mathbf{F}$ est d'arité $\ell + 1$ alors

$$[\![\mathbf{mu}(\mathbf{F})]\!] = \begin{array}{ccc} \mathbb{N}^\ell & \to & \mathbb{N} \\ \vec{x} & \mapsto & \mu i.[\![\mathbf{F}]\!](\vec{x}, i) \end{array}$$

$$\text{où } \mu i.[\![\mathbf{F}]\!](\vec{x}, i) = \begin{cases} \text{le plus petit } i \in \mathbb{N} \text{ tel que } [\![\mathbf{F}]\!](\vec{x}, i) \neq 0 & \text{si un tel } i \text{ existe} \\ \textbf{non défini} & \text{si un tel } i \text{ n'existe pas.} \end{cases}$$

Définition 142 (μ-récursive partielle)
Une fonction partielle $\mathbb{N}^k \to \mathbb{N}$ est μ-récursive partielle si une expression de $\mathcal{L}_{\mu R}$ la dénote.

5.7 Fonctions μ-récursives totales

Définition 143 (μ-récursive totale)
Une fonction **μ-récursive totale** est une fonction totale de la forme $[\![e]\!]$ où $e \in \mathcal{L}_{\mu R}$.

Si on se restreint les minimisations non bornées aux fonctions **sûres**, le calcul effectif de $[\![F]\!](\vec{x})$ termine et la fonction est totale.

Définition 144 (fonction sûre)
Une fonction $q : \mathbb{N}^{\ell+1} \to \mathbb{N}$ est **sûre**[2] si pour tout $\vec{x} \in \mathbb{N}^\ell$, il existe $i \in \mathbb{N}$ tel que $q(\vec{x}, i) \neq 0$.

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

5.8 (*) Langage de programmation impératif avec while vers fonctions μ-récursives partielles

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

Programme P	Expression $tr(P)$
tant que $x_i = 0$ | *bloc*	$\circ(rep, \pi_1^1, \mathbf{mu}(\circ(\mathbf{d_i}, rep)))$ où $rep = \mathbf{rec}\left(\pi_1^1, \circ(tr(bloc), \pi_3^3)\right)$ s'interprète comme la fonction qui prend en argument l'environnement e et un entier i et qui retourne l'environnement après i itérations de *bloc*.

5.9 Équivalence avec les machines de Turing

5.9.1 Des fonctions μ-récursives aux machines de Turing

$(2, 1, 0) \in \mathbb{N}^3 \longleftrightarrow$ | 1 | 0 | , | 1 | , | 0 | ␣ | ␣ | ␣ | ␣ | ␣ ...

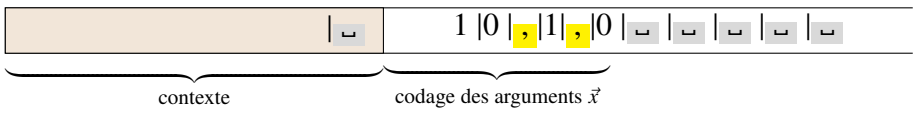
Théorème 145 *Toute fonction μ-récursive partielle est calculable par une machine de Turing.*

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

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

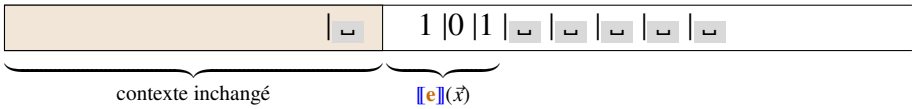
Idée de la Démonstration.
Pour tout expression $\mathbf{e} \in \mathcal{L}_{\mu R}$, on définit la propriété $\mathcal{P}(\mathbf{e})$ suivante :

Si **e** est d'arité k, alors il existe une machine de Turing $M_{\mathbf{e}}$ à un ruban tel que pour tout $\vec{x} \in \mathbb{N}^k$, 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 ␣,

- si $[\![\mathbf{e}]\!](\vec{x})$ est définie, la machine s'arrête avec le ruban suivant :



- si $[\![\mathbf{e}]\!](\vec{x})$ n'est pas définie, la machine boucle.

␣	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 146 *Si le ruban est comme suit :*

| ␣ |1 |1 |0 | , |0 |1| ␣ |1 |0 | , |1 | , |0 | ␣ | ␣ | ␣ | ␣ | ␣

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

| ␣ |1 |1 |0 | , |0 |1| ␣ |1 |0 |1 | ␣ | ␣ | ␣ | ␣ | ␣ | ␣ | ␣

Fonction nulle. Voici $M_{\mathbb{O}}$: remplacer les arguments par 0. d'où $\mathcal{P}(\mathbb{O})$.

Fonction successeur. Voici M_{σ} : incrémenter le dernier nombre écrit de 1. d'où $\mathcal{P}(\sigma)$.

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

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

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

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

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

Ecrire ␣ ; Ecrire 0 (appelons cette portion i) ; Ecrire ␣ ; Recopier k ici ; Ecrire ␣
Recopier $\vec{x}$
Appeler $M_{\mathbf{g}}$
tant que $i < k$
Décaler $[\![\mathbf{rec}\,(\mathbf{g}, \mathbf{h})]\!](\vec{x}, i)$ pour insérer $\vec{x}$, i , avant
Appeler $M_{\mathbf{h}}$
Incrémenter i de 1
Décaler le résultat $[\![\mathbf{rec}\,(\mathbf{g}, \mathbf{h})]\!](\vec{x}, i)$ vers la gauche en effaçant $\vec{x}$, k, i

d'où $\mathcal{P}(\mathbf{rec}\,(\mathbf{g}, \mathbf{h}))$.

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

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

d'où $\mathcal{P}(\mathbf{mu}(\mathbf{q}))$. ■

Corollaire 147 *Toute fonction récursive totale est calculable par une machine de Turing (qui s'arrête sur toutes les entrées).*

Idée de la Démonstration.
Le calcul des minimisations non bornées termine. ■

5.9.2 Des machines de Turing aux fonctions μ-récursives

Codage des entiers

⚠ La représentation binaire est ambigue, e.g. les mots 0^*1 représentent tous l'entier 1.

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

— Une alternative est d'utiliser la structure de données Liste pour encoder un mot.

(sous-section 5.3.3)

Théorème 148

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 (qui s'arrête sur toutes les entrées) est récursive totale.

Idée de la Démonstration.
On décrit le calcul d'une machine de Turing M avec le programme suivant (voir section 5.8) :

```
var w;
c = configinit(w);
tant que configFinale ?(c)
 |   c = suivante(c)
sortie = getRuban(c)
```

où :

— configinit : $\mathbb{N} \to \mathbb{N}$ associe la représentation d'un mot w à la représentation de la configuration initiale ;

— getRuban : $\mathbb{N} \to \mathbb{N}$ associe à la représentation d'une configuration c la représentation du mot sur le ruban.

— suivante : $\mathbb{N} \to \mathbb{N}$ associe la représentation d'une configuration c à la représentation de la configuration suivante de c ;

— configFinale ? : $\mathbb{N} \to \mathbb{N}$ un prédicat qui teste si une configuration c est finale.

[Wolper, 2006] définit une fonction (a priori partielle) qui utilise la minimisation non bornée pour simuler la boucle **tant que** en calculant le nombre d'itérations nécessaires :

$$\text{nbEtapes} : w \mapsto \mu i.\text{configFinale ?}(\text{suivante}^*(\text{configinit}(w), i))$$

où

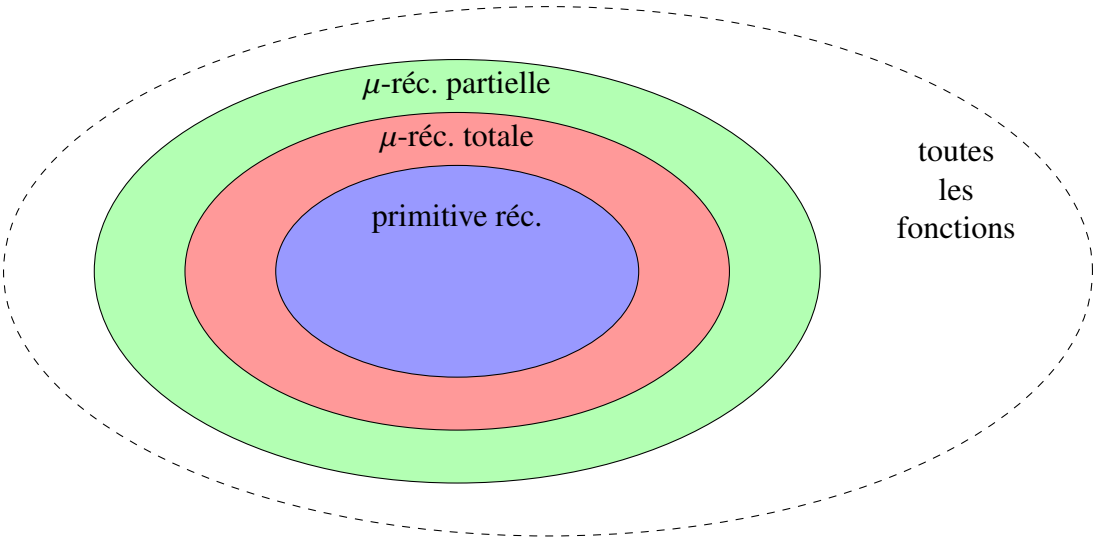
— suivante* : $\mathbb{N}^2 \to \mathbb{N}$ associe, à une configuration c et un entier n, la configuration après n étapes de calcul depuis c.

Voici alors la fonction calculée par la machine de Turing M :

$$f : w \mapsto \text{getRuban}(\text{suivante}^*(\text{configinit}(w), \text{nbEtapes}(w))).$$

Le prédicat configFinale ?(suivante*(configinit(w), i)) est sûr ssi la machine s'arrête sur toute entrée. ■

5.10 Bilan



Fonctions récursives primitives	Fonctions μ-récursives	
programmes sans boucles *while*	programmes avec boucles *while*	
garantie 'syntaxique' de la terminaison	terminaison indécidable	
fonctions totales		fonctions partielles

5.11 Notes bibliographiques

Pas d'introduction d'une syntaxe

D'autres cours ([Wolper, 2006], p. 155 ; [Dehornoy, 2000], p. 171) ne donnent pas de syntaxe pour les fonctions primitives récursives ou les fonctions μ-récursives partielles. Du coup :

— Les 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 5.8 et la démonstration du théorème 145.

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

$$\begin{array}{rccc} f \times g : & \mathbb{N}^k & \to & \mathbb{N}^{n+m} \\ & \vec{x} & \mapsto & (f(\vec{x}), g(\vec{x})) \end{array}.$$

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 $\mu i.q(\vec{x}, i) = 0$. 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 149 *Une fonction f est r. p. ssi il existe une fonction de Bucle qui calcule f.*

Modèle FOR. Un langage équivalent est montré dans ([Olivier Ridoux, 2008], chap. 6) appelé le modèle FOR.

Langages de programmation jouets 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.

Annexe A

PRIMES est dans NP

In this chapter, we sum up the work of Pratt [Pratt, 1975]. Note that we know since 2004 that PRIMES is in P [Agrawal et al., 2004].

Théorème 150 *PRIMES is in NP.*

Idée de la Démonstration.

Lemme 151 *Let $n \geq 3$. n is prime iff there is a $g \in \mathbb{Z}/n\mathbb{Z}^* \setminus \{1\}$ such that $g^{n-1} = 1[n]$ and for all prime number p that divides $n-1$, we have $g^{\frac{n-1}{p}} \neq 1$.*

Idée de la Démonstration.

Integer n is prime	iff	$(\mathbb{Z}/n\mathbb{Z}^*, \times)$ is a (cyclic) group
	iff	there is $g \in \{2, \ldots, n-1\}$ such that $g^{n-1} = 1[n]$ and for all $k \in \{1, \ldots, n-2\}$, $g^k \neq 1[n]$
	iff	there is a $g \in \{2, \ldots, n-1\}$ such that $g^{n-1} = 1[n]$ and for all prime numbers p that divide $n-1$, we have $g^{\frac{n-1}{p}} \neq 1$.

The last iff from top to bottom is trivial. The last iff from bottom to top is proven as follows. As $g^{n-1} = 1[n]$, we consider the group generated by g and let k be the order of g. We prove that $k \leq n-2$ leads to a contradiction. Assume, $k \leq n-2$. Thus, there exists $\ell \geq 2$ such that $k = \frac{n-1}{\ell}$. Let p be a prime number such that $\ell = p\ell'$. We have $g^k = g^{\frac{n-1}{p\ell'}}$. As $g^{k\ell'} \neq 1$, $g^k \neq 1$. Contradiction with the fact that k is the order of g. ■

We design the following non-deterministic algorithm that takes an integer $n \geq 2$:

```
procédure prime(n)
si n = 2 alors -
sinon
    Guess g ∈ {2, ..., n − 1}
    si g^(n−1) ≠ 1 alors rejeter
    Guess numbers p_1, ..., p_k ≥ 2 such that Σ_{i=1}^k log p_i ≤ log (n − 1)
    si n − 1 ≠ Π_{i=1}^k p_i alors rejeter
    pour i = 1 to k
        prime(p_i)
        si g^((n−1)/p_i) = 1 alors rejeter
```

Lemme 152 *Procedure* prime *rejects exactly non-prime numbers $n \geq 2$.*

Idée de la Démonstration.
For all $n \geq 2$, let $\mathcal{P}(n)$: 'prime(n) does not reject n iff n is prime'.

$\mathcal{P}(2)$ is true.

Let $n \geq 3$ such that $\mathcal{P}(\ell)$ holds for all $\ell < n$, and let us prove $\mathcal{P}(n)$.

($\Rightarrow$) Let us consider an non-rejecting execution of prime(n). We exhibit executions of prime(p_i) for all $i \in \{1, \ldots, k\}$ that are sub-executions of prime(n). By $\mathcal{P}(p_i)$, p_i is prime for all i. Lemma 151 implies n is prime.

($\Leftarrow$) Suppose n is prime and consider an execution in which the guessed p_i are such $n-1 = \Pi_{i=1}^{k} p_i$, p_i is prime, and subcalls prime(p_i) never reach **rejeter** (such executions of prime(p_i) exists by $\mathcal{P}(p_i)$). Then in that executions, Lemma 151 implies that we never reach **rejeter** in that call.

■

Now, we prove that prime is running in polynomial time in $\log n$.

Lemme 153 *Each computations (recursive calls excluded) in any subcall* prime(..) *from* prime(n) *is in* $O(\log n^4)$.

Idée de la Démonstration.
Guessing g is in $O(\log n)$. The total number of bits to guess numbers $p_1, \ldots, p_k$ is $O(\log n)$ and $k \leq \log n$. In particular, the number of bits on a number p_i is $O(\log n)$. The recall that the multiplication of two numbers of $O(\log n)$ each is $O(\log^2 n)$. As $k = O(\log n)$, the computation of $\Pi_{i=1}^{k} p_i$ is in $O(\log^3 n)$. Computing $\frac{n-1}{p_i}$ is in $O(\log^2 n)$. Computing $g^{\frac{n-1}{p_i}}$ is in $O(\log^3 n)$: indeed, computing $\frac{n-1}{p_i}$ is in $O(\log^2 n)$, each multiplication costs $O(\log^2 n)$ and there are $O(\log n)$ multiplications. As $g^{\frac{n-1}{p_i}}$ is computed for all $i \in \{1, \ldots, k\}$, and $k = O(\log n)$, we get $O(\log^4 n)$. ■

Let $C(n)$ be the maximal number of calls prime(..) in an execution call prime(n).

Lemme 154 *For all* $n \geq 2$, $C(n) = O(\log n)$.

Idée de la Démonstration.
Let us consider the following property :

$$\mathcal{P}(n) : C(n) \leq 2 \log n - 1$$

First :

$$C(2) = 1 \geq 2 \log 2 - 1 = 1$$
$$C(3) = 2 \geq 2 \log 3 - 1 \sim 2.16...$$

so $\mathcal{P}(2)$, $\mathcal{P}(3)$ hold.

Now, let $n \geq 4$ such that $\mathcal{P}(\ell)$ holds for all $\ell < n$, and let us prove $\mathcal{P}(n)$. Let $p_1, \ldots, p_k$ that corresponds to the maximal number of calls in prime(n). We have :

$$\begin{aligned} C(n) &\leq 1 + \sum_{i=1}^{k} C(p_i) \\ &\leq 1 + \sum_{i=1}^{k} (2 \log(p_i) - 1) \\ &\leq 1 + 2 \log \left(\Pi_{i=1}^{k} p_i \right) - k \\ &\leq 2 \log n - 1 \end{aligned}$$

as $k \geq 2$ when $n \geq 4$.

■

By lemmas 153 and 154, we conclude that prime(n) is in time $O(\log^5 n)$.

■

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.

[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, 2008] Olivier Ridoux, G. L. (2008). *Calculateurs, calculs, calculabilité*. Dunod.

[Papadimitriou, 2003] Papadimitriou, C. H. (2003). *Computational complexity*. John Wiley and Sons Ltd.

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

[Pratt, 1975] Pratt, V. R. (1975). Every prime has a succinct certificate. *SIAM J. Comput.*, 4(3) :214–220.

[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.

Index

2SAT, 42
3SAT, 26

acceptante, 6
acceptation d'un mot, 6
accessibilité, 39
arbre de calcul, 6

certificat, 22
classe de complexité, 31
codage, 5
coloration, 27
configuration, 6
configuration acceptante, 6
configuration initiale, 6

décidable, 11
décideur, 7
décidé en espace, 31
décidé en temps, 7
déterministe, 6

énumerateur, 11
état d'acceptation, 6
état de rejet, 6
état initial, 6
exécution, 6
exécution acceptante, 6

flèche de Knuth, 47
fonction μ-récursive, 52
fonction μ-récursive totale, 52
fonction d'Ackermann-Péter, 51
fonction primitive récursive, 45
formule booléenne quantifiée, 33

gadget, 27
géographie, 35

indécidable, 12, 15, 18
instance, 5
instance négative, 5
instance positive, 5

L, 39
langage accepté, 7
langage décidé, 7
langage rationnel, 37
lemme de König, 9

machine de Turing déterministe, 6
machine de Turing non-déterministe, 6
machine universelle, 12
minimisation bornée, 47
minimisation non bornée, 51

NL, 39
NP, 21
NP-complet, 22
NP-dur, 22

P, 21
plusieurs rubans, 7
problème de correspondance de Post, 14
problème de décision, 5
problème de l'acceptation d'un mot, 13
problème de l'arrêt, 12
problème dual, 19
prédicat, 47
PSPACE, 33

récursif, 11
récursivement énumérable, 11
réduction, 13
réduction en espace logarithmique, 39
réduction polynomiale, 22, 27

SAT, 23
satisfiable, 23

thèse de Church-Turing, 6
théorème de Cook, 23
théorème de Immerman-Szelepcsenyi, 41
théorème de Rice, 18
théorème de Savitch, 32
TQBF, 34

universalité d'un langage rationnel, 37

vérifieur, 22