Introduction à la calculabilité

Pierre Wolper

Email: pw@montefiore.ulg.ac.be

URL: http: //www.montefiore.ulg.ac.be/~pw/
http: //www.montefiore.ulg.ac.be/
~pw/cours/calc.html

Référence

Pierre Wolper, *Introduction à la calculabilité - 2ième édition*, Dunod, 2001.

Chapitre 1

Introduction

1.1 Motivation

- Comprendre les limites de l'informatique.
- Distinguer problèmes solubles et insolubles par des algorithmes.
- Obtenir des résultats indépendants de la technologie employée pour construire les ordinateurs.

1.2 Problèmes et Langages

- Quels problèmes sont solubles par un programme exécuté sur un ordinateur ?

  Il faut préciser :

  – la notion de problème,

  – la notion de programme exécuté sur un ordinateur.

La notion de problème

Problème : question *générique*.

Exemples :

- trier un tableau de nombres ;
- déterminer si un programme écrit en C s'arrête quelles que soient les valeurs des données qui lui sont fournies (problème de l'arrêt) ;
- déterminer si une équation à coefficients entiers a des solutions entières (10ième problème de Hilbert).

La notion de programme

Procédure effective : programme pouvant être exécuté sur un ordinateur.

Exemples :

- Procédure effective : programme écrit en JAVA ;

- Procédure non effective : “pour résoudre le problème de l’arrêt, il faut déterminer si le programme n’a pas de boucles ou de séquences d’appels récursifs infinies.”

Problème de l'arrêt

```
recursive function threen (n: integer):integer;
begin
if (n = 1) then 1
           else if even(n) then threen(n ÷ 2)
                           else threen(3 × n + 1);
end;
```

1.3 La formalisation des problèmes

Comment représente-t-on les instances de problèmes ?

Alphabets et mots

Alphabet : ensemble fini de symboles.

Exemples

- $\{a, b, c\}$
- $\{\alpha, \beta, \gamma\}$
- $\{1, 2, 3\}$
- $\{\clubsuit, \diamondsuit, \heartsuit\}$

Mot sur un alphabet : séquence *finie* d'éléments de cet alphabet.

Exemples

- *a, abs, zt, bbbssnbnzzyyyyyddtrra, grosseguindaille* sont des mots sur l'alphabet $\{a, \ldots, z\}$.

- 4♣3♢5♡2♠, 12765, ♣♡ sont des mots sur l'alphabet $\{0, \ldots, 8, \clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$.

Mot vide : désigné par e, ε ou encore λ.

Longueur du mot w : $|w|$

$w = aaabbaaaabb$
$w(1) = a$, $w(2) = a$,. . . , $w(11) = b$

Représentation des problèmes

Encodage d'un problème

Considérons un problème binaire dont les instances sont encodées par des mots définis sur un alphabet Σ. L'ensemble de tous les mots définis sur Σ peut être partitionné en 3 sous-ensembles :

- *instances positives : réponse* oui (*positive instances*) ;
- *instances négatives : réponse* non (*negative instances*) ;
- mots ne représentant pas des instances du problème.

Ou encore :

- les mots représentant des instances du problème pour lesquelles la réponse est *oui*, *instances positives* ;

- les mots ne représentant pas des instances du problème ou représentant des instances du problème pour lesquelles la réponse est *non*, *instances négatives*.

Langages

Langage (*language*) : ensemble de mots définis sur le même alphabet.

Exemples

- $\{aab, aaaa, \varepsilon, a, b, ababababbbbbbbbbbbb\}$, $\{\varepsilon, aaaaaaa, a, bbbbbb\}$ et $\emptyset$ (l'ensemble vide) : langages sur l'alphabet $\{a, b\}$.
- pour l'alphabet $\{0, 1\}$,

  $\{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100,$
  $101, 110, 111, \ldots\}$: langage contenant tous les mots.
- langage $\emptyset \neq$ langage $\{\varepsilon\}$.
- ensemble des mots représentant les programmes C qui s'arrêtent toujours.

1.4 La description des langages

Opérations sur les langages

Soit deux langages L_1 et L_2.

- $L_1 \cup L_2 = \{w | w \in L_1 \text{ ou } w \in L_2\}$;
- $L_1 \cdot L_2 = \{w | w = xy, x \in L_1 \text{ et } y \in L_2\}$;
- $L_1^* = \{w | \exists k \geq 0 \text{ et } w_1, \ldots, w_k \in L_1 \text{ tels que } w = w_1 w_2 \ldots w_k\}$;
- $\overline{L_1} = \{w | w \notin L_1\}$.

Langages réguliers

$\mathcal{R}$ (ensemble des langages réguliers sur un alphabet Σ) est le plus petit ensemble de langages tel que :

1. $\emptyset \in \mathcal{R}$ et $\{\varepsilon\} \in \mathcal{R}$,

2. $\{a\} \in \mathcal{R}$ pour tout $a \in \Sigma$ et

3. si $A, B \in \mathcal{R}$, alors $A \cup B$, $A \cdot B$ et $A^* \in \mathcal{R}$.

Les expressions régulières

Notation pour représenter les langages réguliers.

1. $\emptyset$, ε et les éléments de Σ sont des expressions régulières.

2. Si α et β sont des expressions régulières, alors $(\alpha\beta)$, $(\alpha \cup \beta)$, $(\alpha)*$ sont des expressions régulières.

Les expressions régulières constituent un langage sur l'alphabet $\Sigma' = \Sigma \cup \{), (, \emptyset, \cup, *, \varepsilon\}$.

Langage dénoté par une expression régulière

1. $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$,

2. $L(a) = \{a\}$ pour tout $a \in \Sigma$,

3. $L((\alpha \cup \beta)) = L(\alpha) \cup L(\beta)$,

4. $L((\alpha\beta)) = L(\alpha) \cdot L(\beta)$,

5. $L((\alpha)*) = L(\alpha)^*$.

Théorème

Un langage est régulier

si et seulement si

il est dénoté par une expression régulière.

Langages réguliers : exemples

- L'ensemble de tous les mots sur $\Sigma = \{a_1, \ldots, a_n\}$ est dénoté par $(a_1 \cup \ldots \cup a_n)^*$ (ou encore Σ^*).

- L'ensemble de tous les mots non vides sur $\Sigma = \{a_1, \ldots, a_n\}$ est dénoté par $(a_1 \cup \ldots \cup a_n)(a_1 \cup \ldots \cup a_n)^*$ (ou encore $\Sigma\Sigma^*$, ou Σ^+).

- l'expression $(a \cup b)^* a (a \cup b)^*$ dénote le langage des mots composés de "a" et "b" qui contiennent au moins un "a".

Langages réguliers : exemples (suite)

$$(a^*b)^* \cup (b^*a)^* = (a \cup b)^*$$

Démonstration

- $(a^*b)^* \cup (b^*a)^* \subset (a \cup b)^*$ car $(a \cup b)^*$ dénote l'ensemble de tous les mots composés des caractères "a" et "b".

- considérons un mot arbitraire

  $$w = w_1 w_2 \dots w_n \in (a \cup b)^*.$$

  On peut distinguer les 4 cas suivants...

1. $w = a^n$ et donc $w \subset (\varepsilon a)^* \subset (b^*a)^*$;
2. $w = b^n$ et donc $w \subset (\varepsilon b)^* \subset (a^*b)^*$;
3. w contient des a et des b et se termine par b

$$w = \underbrace{a \ldots ab}_{a^*b} \underbrace{\ldots b}_{(a^*b)^*} \underbrace{a \ldots ab}_{a^*b} \underbrace{\ldots b}_{(a^*b)^*}$$

   $\Rightarrow w \in (a^*b)^* \cup (b^*a)^*$;
4. w contient des a et des b et se termine par $a \Rightarrow$ décomposition similaire à celle du cas 3.

1.5 Les langages non réguliers

Fait

Il n'y a pas assez d'expressions régulières pour représenter tous les langages !

Définition

Cardinalité d'un ensemble...

Exemple

Les ensembles $\{0, 1, 2, 3\}$, $\{a, b, c, d\}$, $\{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$ ont tous la même taille.
Ils peuvent être mis en bijection, par exemple $\{(0, \clubsuit), (1, \diamondsuit), (2, \heartsuit), (3, \spadesuit)\}$.

Les ensembles dénombrables

Définition

Un ensemble infini est *dénombrable* (*denumerable*) si il existe une bijection entre cet ensemble et l'ensemble des nombres naturels.

Remarque

Au sens courant de "dénombrable", tout ensemble fini est aussi dénombrable.

Ensembles dénombrables : exemples

1. L'ensemble des nombres pairs est dénombrable :

$$\{(0,0),(2,1),(4,2),(6,3),\ldots\}.$$

2. L'ensemble des mots sur l'alphabet $\{a,b\}$ est dénombrable :
   $\{(\varepsilon,0),(a,1),(b,2),(aa,3),(ab,4),(ba,5),$
   $(bb,6),(aaa,7)\ldots\}$.

3. Les nombres rationnels sont dénombrables :
   $\{(0/1,0),(1/1,1),(1/2,2),(2/1,3),(1/3,4),$
   $(3/1,5),\ldots\}$.

4. Les expressions régulières sont dénombrables.

La technique de la diagonale

Théorème

L'ensemble des sous-ensembles d'un ensemble dénombrable n'est pas dénombrable.

Démonstration

	a_0	a_1	a_2	a_3	a_4	$\cdots$
s_0	$\times$	$\times$		$\times$		
s_1	$\times$	$\square$		$\times$		
s_2		$\times$	$\times$		$\times$	
s_3	$\times$		$\times$	$\square$		
s_4		$\times$		$\times$	$\square$	
$\vdots$						

$$D = \{a_i \mid a_i \notin s_i\}$$

Conclusion

- L'ensemble des langages n'est pas dénombrable.
- L'ensemble des langages réguliers est dénombrable.
- Il y a donc (beaucoup) plus de langages que de langages réguliers.

1.6 Un aperçu de la suite...

- Notion de procédure effective (automates).
- Problèmes non solubles algorithmiquement.
- Problèmes non solubles efficacement.

Chapitre 2
Les automates finis

2.1 Introduction

- Automates finis : première modélisation de la notion de procédure effective.(Ont aussi d'autres applications).

- Dérivation de la notion d'automate fini de celle de programme exécuté sur un ordinateur : état, état initial, fonction de transition.

- Hypothèse du nombre d'états fini. Conséquence : séquences d'états finies ou cycliques.

- Problème de la représentation des données : nombre de données différentes limitées car nombre d'états initiaux possibles fini.

Représentation des données.

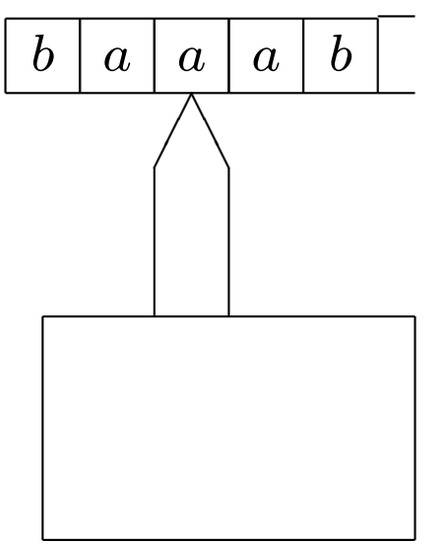
- Problème : reconnaître un langage.
- Données : mot.
- On supposera le mot fourni caractère par caractère, la machine traitant un caractère à chaque cycle et s'arrêtant à la fin du mot.

2.2 Description

- Ruban d'entrée.
- Ensemble d'états :
  - état initial,
  - états accepteurs.
- Mécanisme d'exécution.

ruban :

tête :



2.3 Formalisation

Un automate fini déterministe est défini par un quintuplet $M = (Q, \Sigma, \delta, s, F)$, où

- Q est un ensemble fini d'états,
- Σ est un alphabet,
- $\delta : Q \times \Sigma \to Q$ est la fonction de transition,
- $s \in Q$ est l'état initial,
- $F \subseteq Q$ est l'ensemble des états accepteurs.

Définition du langage accepté

- Configuration : $(q, w) \in Q \times \Sigma^*$.
- Configuration dérivable en une étape : $(q, w) \vdash_M (q', w')$.
- Configuration dérivable (en plusieurs étapes) : $(q, w) \vdash_M^* (q', w')$.
- Exécution d'un automate :

$$(s, w) \vdash (q_1, w_1) \vdash (q_2, w_2) \vdash \cdots \vdash (q_n, \varepsilon)$$

- Mot accepté :

$$(s, w) \vdash_M^* (q, \varepsilon)$$

  et $q \in F$.

- Langage accepté $L(M)$:

$$\{w \in \Sigma^* \mid (s, w) \vdash_M^* (q, \varepsilon) \text{ avec } q \in F\}.$$

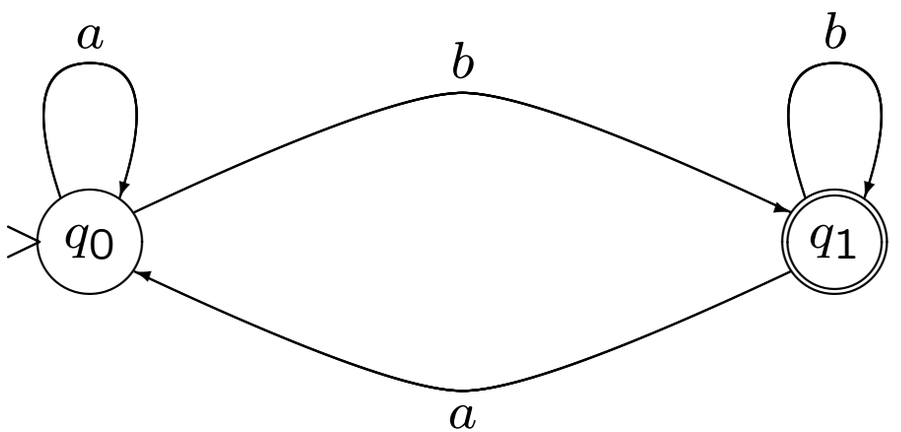
2.4 Exemples

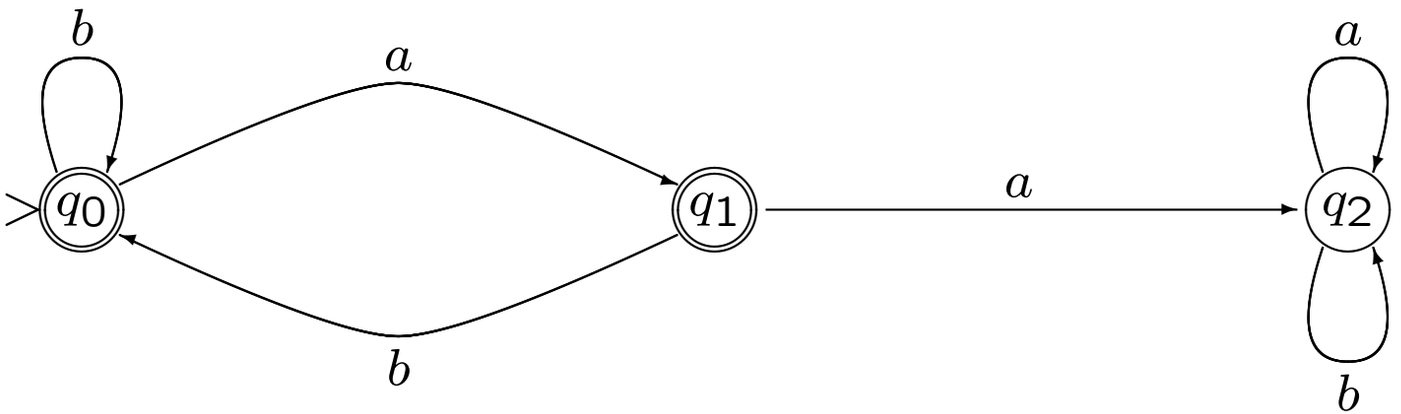
Mots se terminant par b :

δ :

q	σ	$\delta(q, \sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_0
q_1	b	q_1

$Q = \{q_0, q_1\}$
$\Sigma = \{a, b\}$
$s = q_0$
$F = \{q_1\}$





$\{w \mid w$ ne contient pas 2 a consécutifs$\}$.

2.5 Les automates finis non déterministes

Automates qui peuvent *choisir* parmi plusieurs transitions.

Motivation :

- Voir les conséquences de l'extension d'une définition donnée.
- Faciliter la description de langages par les automates finis.
- Le concept de non-déterminisme est généralement utile.

Description

Les automates finis non déterministes sont des automates finis où l'on permet :

- plusieurs transitions correspondant à la même lettre dans chaque état,
- des transitions sur le mot vide (c'est-à-dire sans avancer dans le mot d'entrée),
- des transitions sur des mots de longueur supérieure à 1 (regroupement de transitions).

Acceptent si au moins une exécution accepte.

Formalisation

Un automate fini non déterministe est défini par un quintuplet $M = (Q, \Sigma, \Delta, s, F)$, où

- Q est un ensemble d'états,
- Σ est un alphabet,
- $\Delta \subset (Q \times \Sigma^* \times Q)$ est la relation de transition,
- $s \in Q$ est l'état initial,
- $F \subseteq Q$ est l'ensemble des états accepteurs.

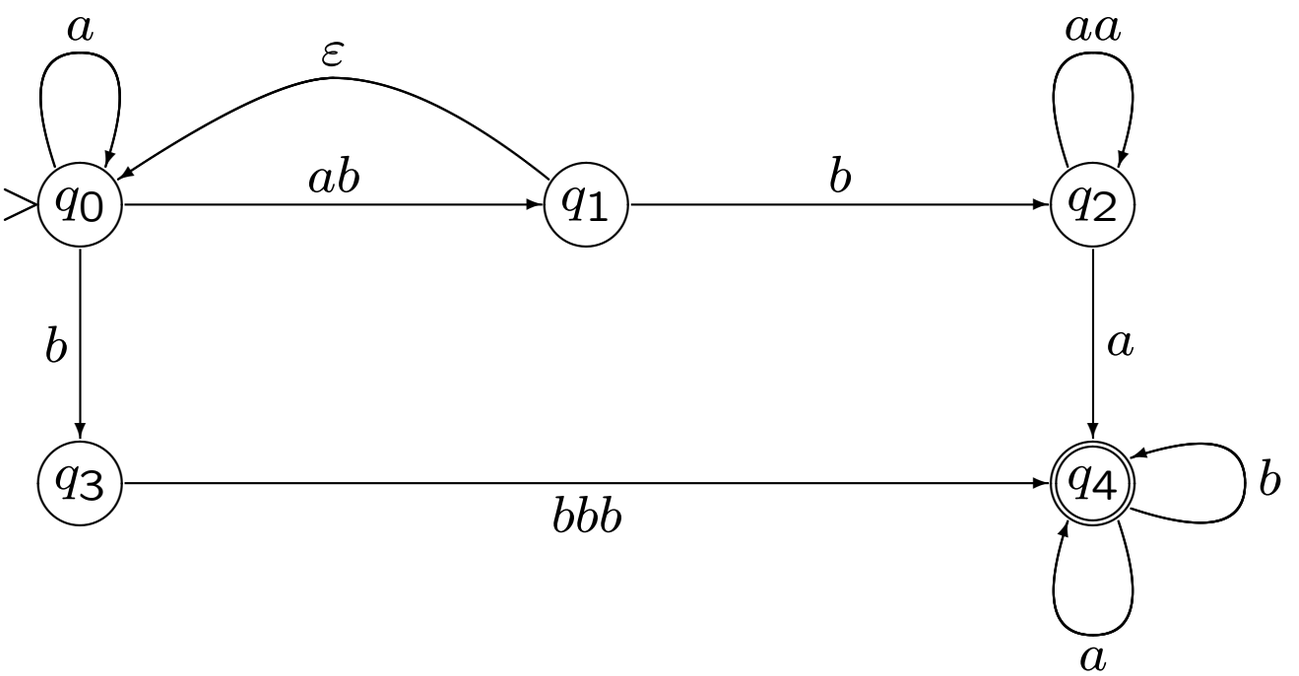
Définition du langage accepté

La configuration (q', w') est dérivable en une étape de la configuration (q, w) par la machine M ($(q, w) \vdash_M (q', w')$) si

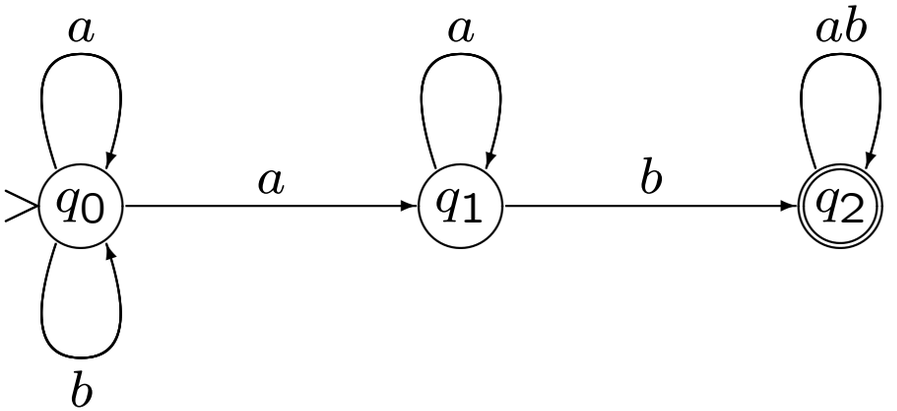
- $w = uw'$ (le mot w commence par un préfixe $u \in \Sigma^*$),
- $(q, u, q') \in \Delta$ (le triplet (q, u, q') est dans la relation de transition Δ).

Un mot est accepté s'il existe une exécution pour ce mot menant à un état accepteur.

Exemples



$$L(M) = ((a \cup ab)^* bbbb\Sigma^*) \cup ((a \cup ab)^* abb(aa)^* a\Sigma^*)$$



$$L(M) = \Sigma^* ab(ab)^*$$

Mots se terminant par au moins une répétition de ab.

2.6 Elimination du non-déterminisme

Définition

Deux automates M_1 et M_2 sont équivalents s'ils acceptent le même langage, c'est-à-dire si $L(M_1) = L(M_2)$.

Théorème

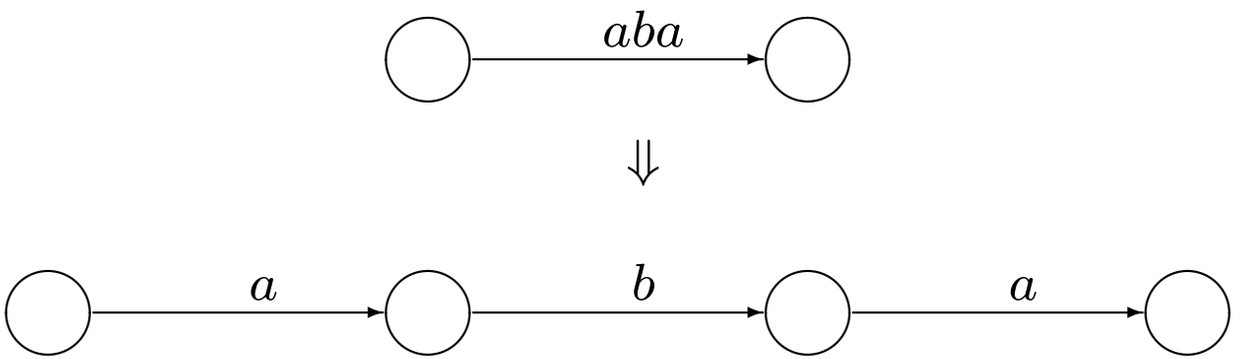
Pour tout automate non déterministe, il est possible de construire un automate déterministe équivalent.

2.6 Principe de la construction

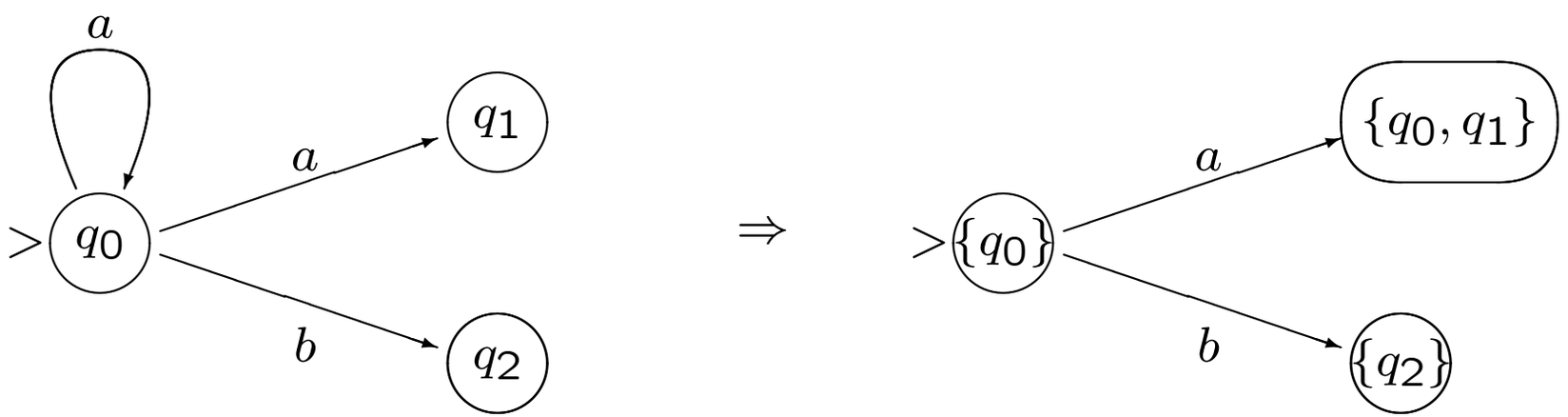
1. Eliminer les transitions de longueur supérieure à 1.

2. Eliminer les transitions compatibles.

Transitions de longueur supérieure à 1.



Transitions compatibles.



Formalisation

Automate non déterministe $M = (Q, \Sigma, \Delta, s, F)$. Construire un automate non déterministe équivalent $M' = (Q', \Sigma, \Delta', s, F)$ tel que $\forall (q, u, q') \in \Delta',\ |u| \leq 1$.

- Initialement $Q' = Q$ et $\Delta' = \Delta$.

- Pour chaque transition $(q, u, q') \in \Delta$ avec $u = \sigma_1 \sigma_2 \ldots \sigma_k$, $(k > 1)$:
  - on enlève cette transition de Δ',
  - on ajoute de nouveaux états $p_1,\ \ldots,\ p_{k-1}$ à Q',
  - on ajoute les nouvelles transitions (q, σ_1, p_1), (p_1, σ_2, p_2), $\ldots$, (p_{k-1}, σ_k, q') à Δ'

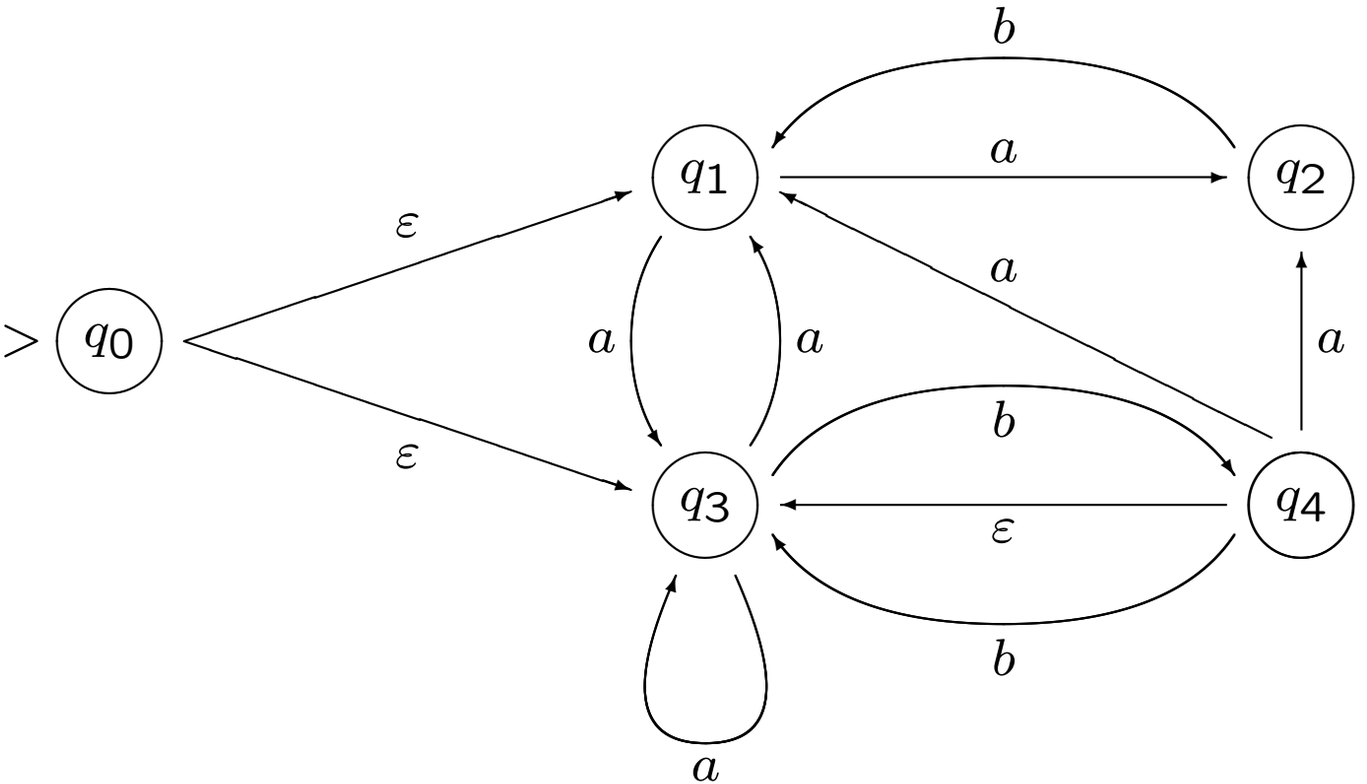
Formalisation

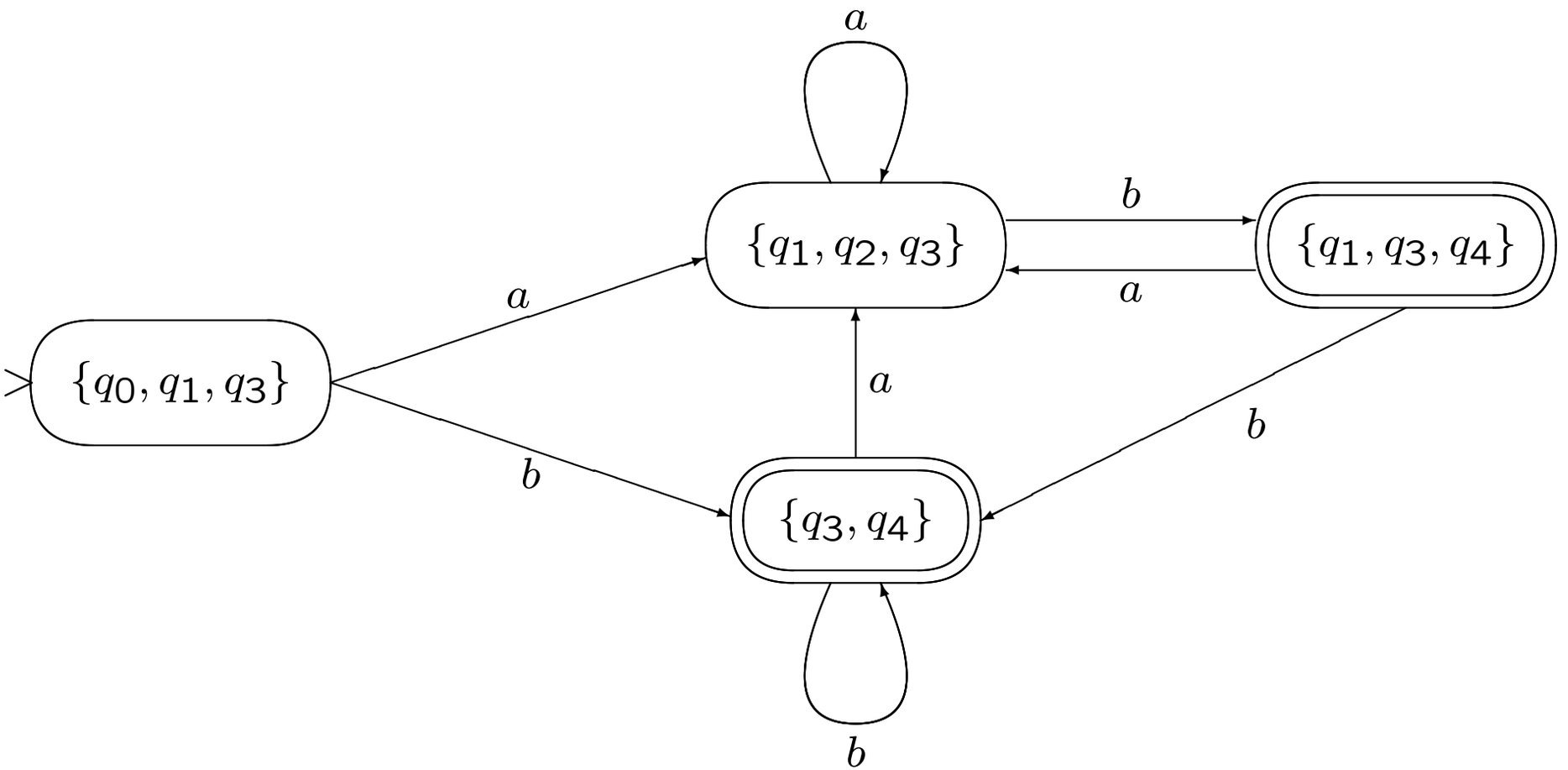
Automate non déterministe $M = (Q, \Sigma, \Delta, s, F)$ tel que $\forall (q, u, q') \in \Delta',\ |u| \leq 1$. Construire un automate déterministe équivalent $M' = (Q', \Sigma, \delta', s, F)$.

$$E(q) = \{p \in Q \mid (q, w) \vdash_M^* (p, w)\}.$$

- $Q' = 2^Q$.
- $s' = E(s)$.
- $\delta'(\mathbf{q}, a) = \bigcup\{E(p) \mid \exists q \in \mathbf{q}\ : (q, a, p) \in \Delta\}$.
- $F' = \{\mathbf{q} \in Q' \mid \mathbf{q} \cap F \neq \emptyset\}$.

Exemple




$\{q_0, q_1, q_3\}$
$\{q_1, q_2, q_3\}$
$\{q_1, q_3, q_4\}$
$\{q_3, q_4\}$
a
a
b
b
a
a
b
b

Autre construction : Uniquement états accessibles

1. Au départ Q' contient l'état initial s'.

2. Les opérations suivantes sont alors répétées jusqu'à ce qu'elles ne modifient plus l'ensemble Q'.

   (a) On choisit un état $\mathbf{q} \in Q'$ auquel l'opération (b) n'a pas encore été appliquée.

   (b) Pour chaque lettre $a \in \Sigma$ on calcule l'état $\mathbf{p}$ tel que $\mathbf{p} = \delta'(\mathbf{q}, a)$. L'état $\mathbf{p}$ est ajouté à Q'.

2.7 Automates finis et expressions régulières

Théorème
Un langage est régulier si et seulement si il est accepté par un automate fini.

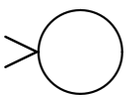
Nous démontrons :

1. Si un langage est dénoté par une expression régulière, il est accepté par un automate fini non déterministe.

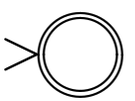
2. Si un langage est accepté par un automate fini non déterministe, il est régulier.

Des expressions aux automates

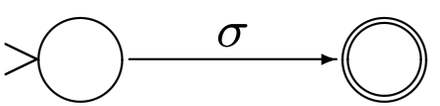
- $\emptyset$



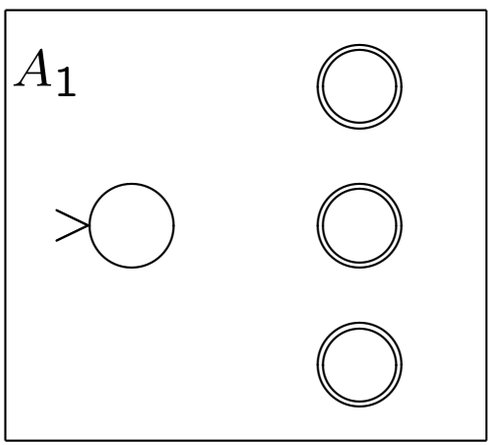
- ε

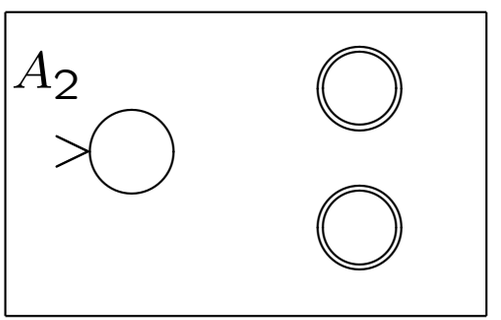


- $\sigma \in \Sigma$

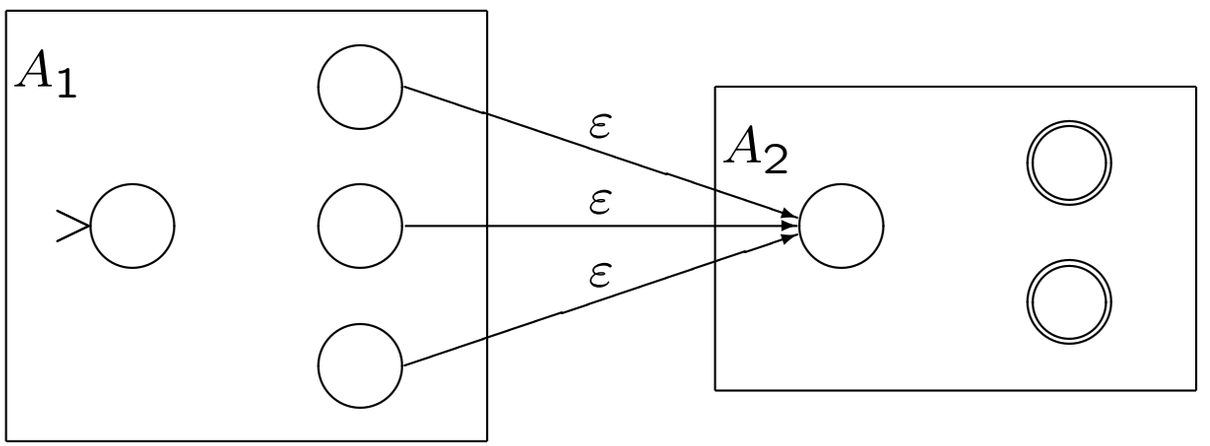


$\alpha_1 : A_1 = (Q_1, \Sigma, \Delta_1, s_1, F_1)$
$\alpha_2 : A_2 = (Q_2, \Sigma, \Delta_2, s_2, F_2)$





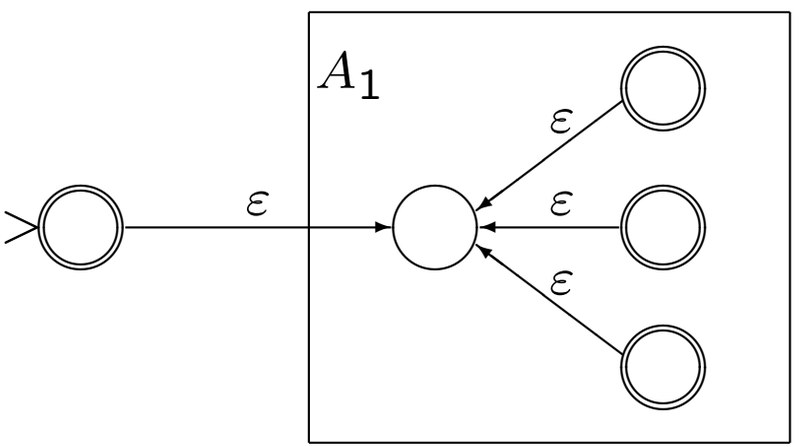
- $\alpha_1 \cdot \alpha_2$



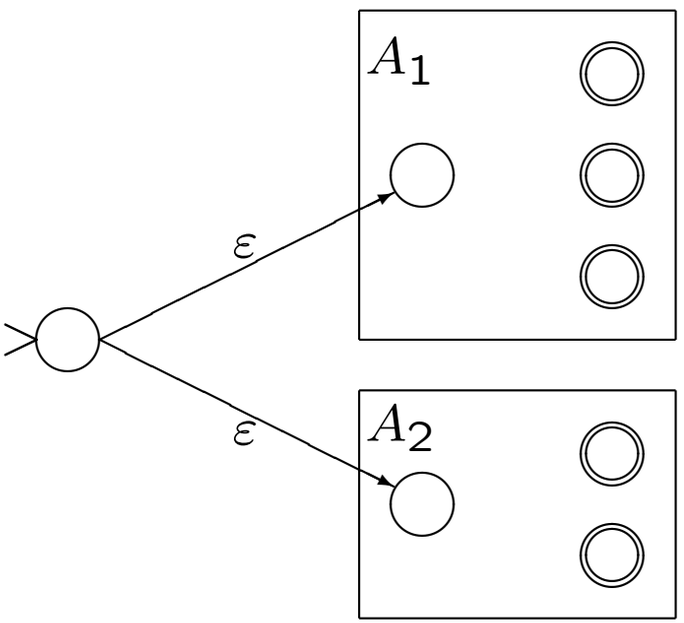
Formellement, $A = (Q, \Sigma, \Delta, s, F)$ où

- $Q = Q_1 \cup Q_2$,
- $\Delta = \Delta_1 \cup \Delta_2 \cup \{(q, \varepsilon, s_2) \mid q \in F_1\}$,
- $s = s_1$,
- $F = F_2$.

- $\alpha = \alpha_1^*$



- $\alpha = \alpha_1 \cup \alpha_2$



Des automates aux langages réguliers

Idée intuitive :

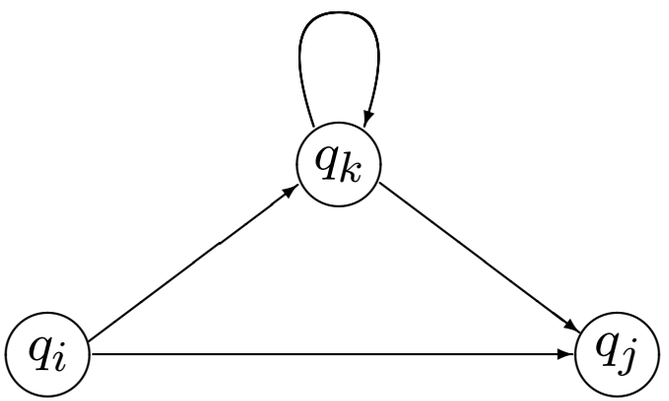
- Construire l'expression régulière correspondant à chaque chemin entre l'état initial et un état accepteur.

- Traiter les boucles à l'aide de l'opérateur $*$.

Définition
Soit un automate M et $Q = \{q_1, q_2, \ldots, q_n\}$ l'ensemble de ses états. Nous désignons par $R(i, j, k)$ l'ensemble des mots permettant de passer de l'état q_i à q_j en passant uniquement par des états $\{q_1, \ldots, q_{k-1}\}$.

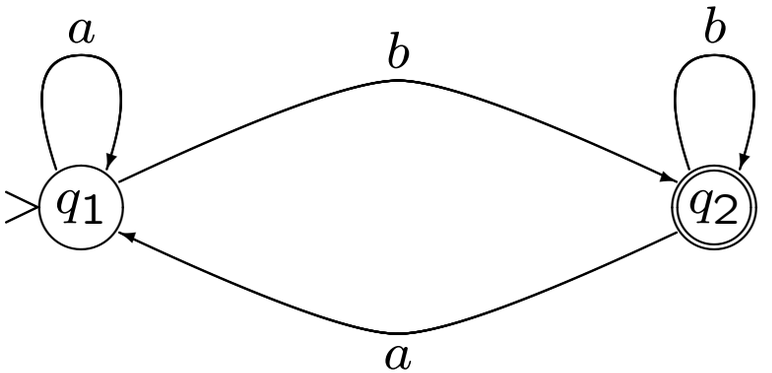
$$R(i,j,1) = \begin{cases} \{w \mid (q_i, w, q_j) \in \Delta\} & \text{si } i \neq j \\ \{\varepsilon\} \cup \{w \mid (q_i, w, q_j) \in \Delta\} & \text{si } i = j \end{cases}$$

$$\begin{aligned} R(i,j,k+1) &= R(i,j,k) \cup \\ &\quad R(i,k,k)R(k,k,k)^*R(k,j,k) \end{aligned}$$



$$L(M) = \bigcup_{q_j \in F} R(1,j,n+1).$$

Exemple



	$k=1$	$k=2$
$R(1,1,k)$	$\varepsilon \cup a$	$(\varepsilon \cup a) \cup (\varepsilon \cup a)(\varepsilon \cup a)^*(\varepsilon \cup a)$
$R(1,2,k)$	b	$b \cup (\varepsilon \cup a)(\varepsilon \cup a)^* b$
$R(2,1,k)$	a	$a \cup a(\varepsilon \cup a)^*(\varepsilon \cup a)$
$R(2,2,k)$	$\varepsilon \cup b$	$(\varepsilon \cup b) \cup a(\varepsilon \cup a)^* b$

Le langage accepté par l'automate est alors $R(1,2,3)$, soit

$$[b \cup (\varepsilon \cup a)(\varepsilon \cup a)^* b] \quad \cup \quad [b \cup (\varepsilon \cup a)(\varepsilon \cup a)^* b]\,[(\varepsilon \cup b) \cup a(\varepsilon \cup a)^* b]^*\,[(\varepsilon \cup b) \cup a(\varepsilon \cup a)^* b]$$

Chapitre 3

Les grammaires régulières

3.1 Introduction

Aspect différent de la notion de langage :

- pas formalisation de la notion de procédure effective,
- mais ensemble de mots satisfaisant certaines règles.
- Origine : formalisation du langage naturel.

Exemple

- une `phrase` est de la forme `sujet verbe`
- un `sujet` est un `pronom`
- un `pronom` est **il** ou **elle**
- un verbe est **dort** ou **écoute**

Phrases formées:

1. **il écoute**
2. **il dort**
3. **elle dort**
4. **elle écoute**

Grammaire

- Grammaire: description *générative* d'un langage
- Automate: description *analytique*
- Exemple: Langages de programmation définis par une grammaire (BNF), mais reconnus à l'aide d'une description analytique.
- La théorie des langages établit des correspondances entre descriptions analytiques et génératives.

3.2 Les grammaires

Une grammaire est un quadruplet $G = (V, \Sigma, R, S)$.

- V est un alphabet,
- $\Sigma \subseteq V$ est l'ensemble des *symboles terminaux* ($V - \Sigma$ est l'ensemble des *symboles non terminaux*),
- $R \subseteq (V^+ \times V^*)$ est un ensemble (fini) de *règles*,
- $S \in V - \Sigma$ est le *symbole de départ*.

Notation:

- Eléments de $V - \Sigma$: $A, B, \ldots$
- Eléments de Σ : $a, b, \ldots$.
- Règles $(\alpha, \beta) \in R$: $\alpha \rightarrow \beta$ ou encore $\alpha \underset{G}{\rightarrow} \beta$.
- Symbole de départ désigné par S.
- Mot vide : ε.

Exemple :

- $V = \{S, A, B, a, b\}$,
- $\Sigma = \{a, b\}$,
- $R = \{S \to A, S \to B, B \to bB, A \to aA, A \to \varepsilon, B \to \varepsilon\}$,
- S est le symbole de départ.

Mots générés : exemple

$aaaa$ fait partie du langage défini par la grammaire décrite précédemment :

S		
A	règle	$S \rightarrow A$
aA		$A \rightarrow aA$
aaA		$A \rightarrow aA$
$aaaA$		$A \rightarrow aA$
$aaaaA$		$A \rightarrow aA$
$aaaa$		$A \rightarrow \varepsilon$

Mots générés : définition

Soit $G = (V, \Sigma, R, S)$ et $u \in V^+,\ v \in V^*$. La grammaire G *permet de dériver* v *de* u *en une étape* (notation $u \underset{G}{\Rightarrow} v$) si et seulement si :

- $u = xu'y$ (u peut être décomposé en trois parties x, u' et y ; les parties x et y peuvent éventuellement être vides),
- $v = xv'y$ (v peut être décomposé en trois parties x, v' et y),
- $u' \underset{G}{\rightarrow} v'$ (la règle (u', v') est dans R).

Soit $G = (V, \Sigma, R, S)$ et $u \in V^+,\ v \in V^*$. La grammaire G *permet de dériver* v *de* u *en plusieurs étapes* (notation $u \overset{*}{\underset{G}{\Rightarrow}} v$) si et seulement si $\exists k \geq 0$ et $v_0 \dots v_k \in V^+$ tels que

- $u = v_0$,
- $v = v_k$,
- $v_i \underset{G}{\Rightarrow} v_{i+1}$ pour $0 \leq i < k$.

- Mots générés par une grammaire G: mots $v \in \Sigma^*$ (uniquement composés de symboles terminaux) tels que

$$S \overset{*}{\underset{G}{\Rightarrow}} v.$$

- Langage généré par une grammaire G (dénoté $L(G)$) est l'ensemble

$$L(G) = \{v \in \Sigma^* \mid S \overset{*}{\underset{G}{\Rightarrow}} v\}.$$

Exemple :
Le langage généré par la grammaire de l'exemple ci-dessus est celui comprenant tous les mots composés exclusivement de lettres a ou de lettres b.

Types de grammaires

Type 0: pas de restrictions sur les règles.

Type 1: grammaires *sensibles au contexte* (*context-sensitive*).

Les règles

$$\alpha \rightarrow \beta$$

satisfont la condition

$$|\alpha| \leq |\beta|.$$

Exception: la production

$$S \rightarrow \varepsilon$$

pour autant que S n'apparaisse pas dans le membre de droite d'une production.

Type 2: grammaires *hors-contexte* (*context-free*).

Productions de la forme

$$A \to \beta$$

où $A \in V - \Sigma$ et pas de restriction sur β.

Type 3: grammaires *régulières*.

Productions de la forme

$$A \to wB$$
$$A \to w$$

où $A, B \in V - \Sigma$ et $w \in \Sigma^*$.

3.3 Les grammaires régulières

Théorème:
Un langage est régulier si et seulement si il est généré par une grammaire régulière.

A. Si un langage est régulier, il est généré par une grammaire régulière.

Si L est régulier, il existe

$$M = (Q, \Sigma, \Delta, s, F)$$

tel que $L = L(M)$. A l'aide de M, on construit une grammaire régulière

$$G = (V_G, \Sigma_G, S_G, R_G)$$

qui génère L.

G est définie par :

- $\Sigma_G = \Sigma$,
- $V_G = Q \cup \Sigma$,
- $S_G = s$,
- $R_G = \left\{ \begin{array}{ll} A \to wB, & \text{pour tout } (A, w, B) \in \Delta \\ A \to \varepsilon & \text{pour tout } A \in F \end{array} \right\}$

B. Si un langage est généré par une grammmaire régulière, il est régulier.

Soit

$$G = (V_G, \Sigma_G, S_G, R_G)$$

la grammaire qui génère le langage L. Un automate fini non déterministe qui accepte L peut être défini comme suit :

- $Q = V_G - \Sigma_G \cup \{f\}$ (les états de M sont les non-terminaux de G plus un nouvel état f),
- $\Sigma = \Sigma_G$,
- $s = S_G$,
- $F = \{f\}$,
- $\Delta = \left\{ \begin{array}{ll} (A, w, B), & \text{pour tout } A \to wB \in R_G \\ (A, w, f), & \text{pour tout } A \to w \in R_G \end{array} \right\}$.

3.4 les langages réguliers

Quatre caractérisations des langages réguliers :

1. expressions régulières,

2. automates finis déterministes,

3. automates finis non-déterministes,

4. grammaires régulières.

Propriétés des langages réguliers

Soient L_1 et L_2, deux langages réguliers.

- $L_1 \cup L_2$ est régulier.
- $L_1 \cdot L_2$ est régulier.
- L_1^* est régulier.
- L_1^R est régulier.
- $\overline{L_1} = \Sigma^* - L_1$ est régulier.
- $L_1 \cap L_2$ est régulier.

$L_1 \cap L_2$ régulier ?

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Ou encore, si $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ accepte L_1 et $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ accepte L_2, l'automate suivant accepte $L_1 \cap L_2$:

- $Q = Q_1 \times Q_2$,
- $\delta((q_1, q_2), \sigma) = (p_1, p_2)$ si et seulement si $\delta_1(q_1, \sigma) = p_1$ et $\delta_2(q_2, \sigma) = p_2$,
- $s = (s_1, s_2)$,
- $F = F_1 \times F_2$.

- Soit Σ l'alphabet de définition de L_1 et $\pi : \Sigma \to \Sigma'$ une fonction de Σ vers un autre alphabet Σ'.

  Cette fonction dite de *projection* peut être étendue aux mots en l'appliquant à chaque symbole, *i.e.* pour $w = w_1 \dots w_k \in \Sigma^*$, $\pi(w) = \pi(w_1) \dots \pi(w_k)$.

  Pour L_1 régulier, le langage $\pi(L_1)$ est régulier.

Algorithmes

Les problèmes suivants sont solubles par un algorithme pour les langages réguliers :

- $w \in L$?
- $L = \emptyset$?
- $L = \Sigma^*$? $\qquad (\overline{L} = \emptyset)$
- $L_1 \subseteq L_2$? $\qquad (\overline{L_2} \cap L_1 = \emptyset)$
- $L_1 = L_2$? $\qquad (L_1 \subseteq L_2$ et $L_2 \subseteq L_1)$

3.5 Au-delà des langages réguliers

- Beaucoup de langages sont réguliers.

- Mais, tous les langages ne peuvent pas être réguliers. (argument de cardinalité)

- Nous allons montrer que certains langages particuliers ne sont pas réguliers en utilisant une autre technique.

Observations de base

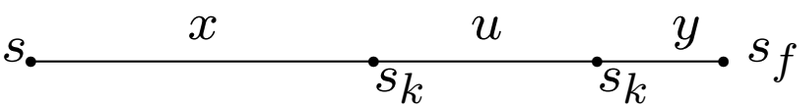
1. Tous les langages finis (comportant un nombre fini de mots) sont réguliers.

2. Un langage non régulier doit donc comporter un nombre infini de mots.

3. Si un langage comporte un nombre infini de mots, il n'y a pas de borne à la taille des mots faisant partie du langage.

4. Tout langage régulier est accepté par un automate fini comportant un nombre fixé d'états, soit m.

5. Considérons un langage régulier infini et un automate à m états acceptant ce langage. Pour tout mot de longueur supérieure à m, l'exécution de l'automate sur ce mot doit passer par un même état s_k au moins deux fois avec une partie non vide du mot séparant ces deux passages



6. Par conséquent, tous les mots de la forme xu^*y sont aussi acceptés par l'automate et font partie du langage.

Les théorèmes du "gonflement"

Première version :
Soit L un langage régulier infini. Alors, il existe $x, u, y \in \Sigma^*$, avec $u \neq \varepsilon$ tels que $xu^n y \in L \ \forall n \geq 0$.

Deuxième version :
Soit L un langage régulier infini et soit $w \in L$ tel que $|w| \geq |Q|$ où Q est l'ensemble d'états d'un automate déterministe acceptant L. Alors $\exists x, u, y$, avec $u \neq \varepsilon$ et $|xu| \leq |Q|$ tels que $xuy = w$ et, $\forall n$, $xu^n y \in L$.

Applications des théorèmes du gonflement

Le langage

$$a^n b^n$$

n'est pas régulier. En effet, il n'est pas possible de trouver des mots x, u, y tels que $xu^k y \in a^n b^n \ \forall k$ et donc que le théorème du gonflement est contredit.

$u \in a^*$: impossible.

$u \in b^*$: impossible.

$u \in (a \cup b)^* - (a^* \cup b^*)$: impossible.

Le langage

$$L = a^{n^2}$$

n'est pas régulier. En effet, le théorème du gonflement (2ième version) est contredit.

Soit $m = |Q|$ le nombre d'états d'un automate acceptant L. Considérons a^{m^2}. Puisque $m^2 \geq m$, il doit exister x, u et y telles que $|xu| \leq m$ et $xu^n y \in L\ \forall n$. Explicitement, on a

$$\begin{array}{ll} x = a^p & 0 \leq p \leq m-1, \\ u = a^q & 0 < q \leq m, \\ y = a^r & r \geq 0. \end{array}$$

Par conséquent $xu^2y \notin L$ puisque $p + 2q + r$ n'est pas un carré parfait. En effet,

$$m^2 < p + 2q + r \leq m^2 + m < (m+1)^2 = m^2 + 2m + 1.$$

Le langage

$$L = \{a^n \mid n \text{ est premier}\}$$

n'est pas régulier. Le premier théorème du gonflement implique qu'il existe des constantes p, q et r telles que $\forall k$

$$xu^k y = a^{p+kq+r} \in L,$$

autrement dit que $p + kq + r$ est premier pour tout k. Cela est impossible, puisque pour $k = p + 2q + r + 2$. Nous avons

$$p + kq + r = \underbrace{(q+1)}_{>1}\underbrace{(p+2q+r)}_{>1},$$

Les applications des langages réguliers

Problème : Trouver dans une (longue) chaîne de caractères w, toutes les instances d'une expression régulière α.

1. On considère l'expression régulière $\beta = \Sigma^* \alpha$.

2. On construit un automate non déterministe acceptant le langage décrit par β

3. A partir de cet automate, on construit un automate *déterministe* A_β.

4. On simule l'exécution de l'automate A_β sur le mot w. Lorsque cet automate est dans un état accepteur, on se trouve à la fin d'une occurrence de α dans le mot w.

Les applications des langages réguliers II: Le traitement de l'arithmétique

- Un nombre écrit en base r est un mot construit sur l'alphabet $\{0, \ldots, r-1\}$ ($\{0, \ldots, 9\}$ en décimal, $\{0, 1\}$ en binaire).

- Le nombre représenté par un mot $w = w_0 \ldots w_l$ est $nb(w) = \sum_{i=0}^{l} r^{l-i} nb(w_i)$

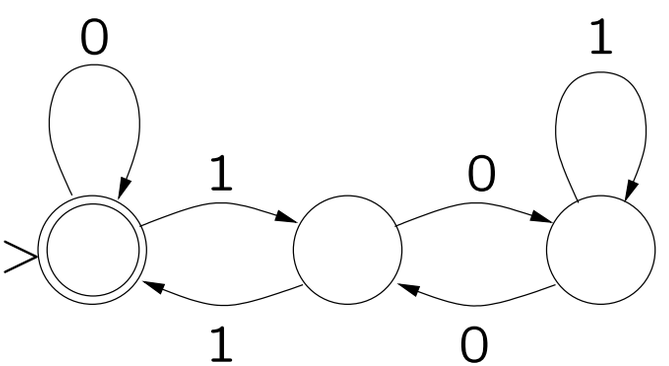
- Ajouter des "0" en tête de la représentation d'un nombre ne modifie pas la valeur représentée et donc un nombre a une infinité de représentations possibles.

- Les représentations sont lues en commençant par le chiffre la plus significatif et on considère tous les encodages possibles.

- **Exemple:** L'ensemble des représentations binaires du nombre 5 est le langage 0^*101.

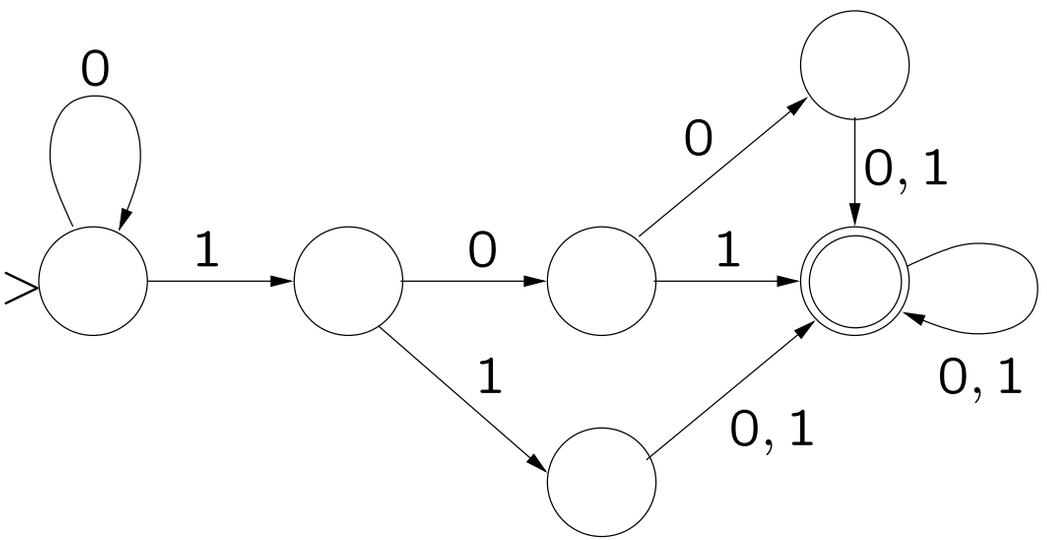
Quels ensembles de nombres sont réprésentés par des langages réguliers?

- Ensembles finis.
- L'ensemble des multiples de 2 est représenté par le langage $(0 \cup 1)^*0$.
- L'ensemble des puissances de 2 est représenté par le langage 0^*10^*, mais n'est pas représentable en base 3.
- L'ensemble des multiples de 3 est représenté par le langage accepté par l'automate ci-dessous.



Ensembles de nombres représentés par des langages réguliers (suite)

- L'ensemble des nombres $x \geq 5$ est représenté par l'automate



- En généralisant: ensembles de la forme $\{ax \mid x \in N\}$ ou $\{x \geq a \mid x \in N\}$ pour toute valeur fixée a.

Ensembles de nombres représentés par des langages réguliers (suite II)

- En combinant les deux catégories: ensembles de la forme $\{ax + b \mid x \in N\}$, pour a et b fixés.

- Union de tels ensembles: les *ensembles ultimement périodiques*.

- Intersection et complément n'apportent rien de plus.

- Les seuls ensembles représentables dans toutes les bases sont les ensembles ultimement périodiques.

Représentation de vecteurs de nombres

- Chaque nombre est représenté par un mot et on groupe les bits de position identique.

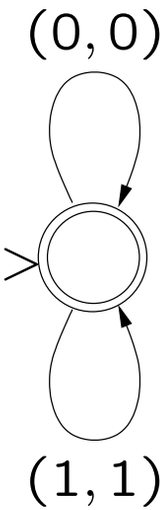
- **Exemple:**

  – le vecteur $(5, 9)$ est encodé par le mot $(0, 1)(1, 0)(0, 0)(1, 1)$ défini sur l'alphabet $\{0, 1\} \times \{0, 1\}$.

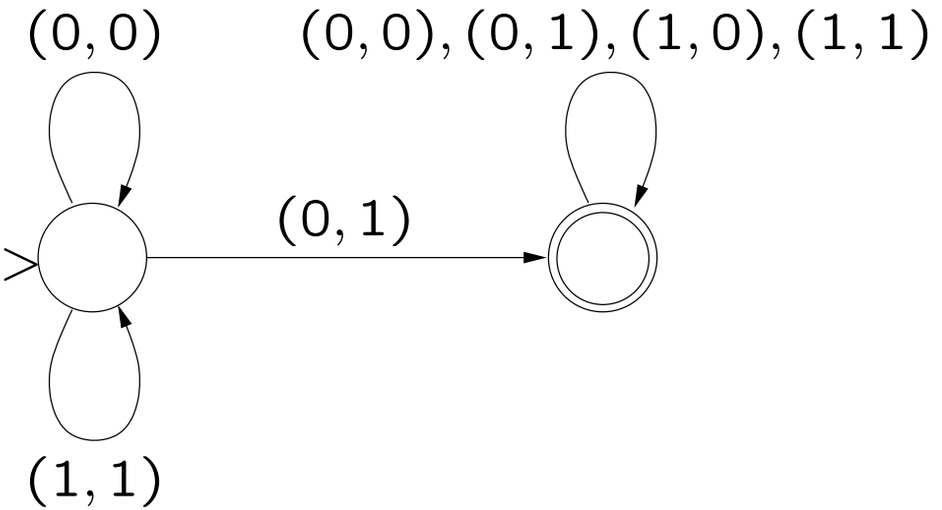
  – L'ensemble des encodages en binaire du vecteur $(5, 9)$ est $(0, 0)^*(0, 1)(1, 0)\ (0, 0)(1, 1)$.

Quels sont les ensembles de vecteurs de nombres définissables par des langages réguliers?

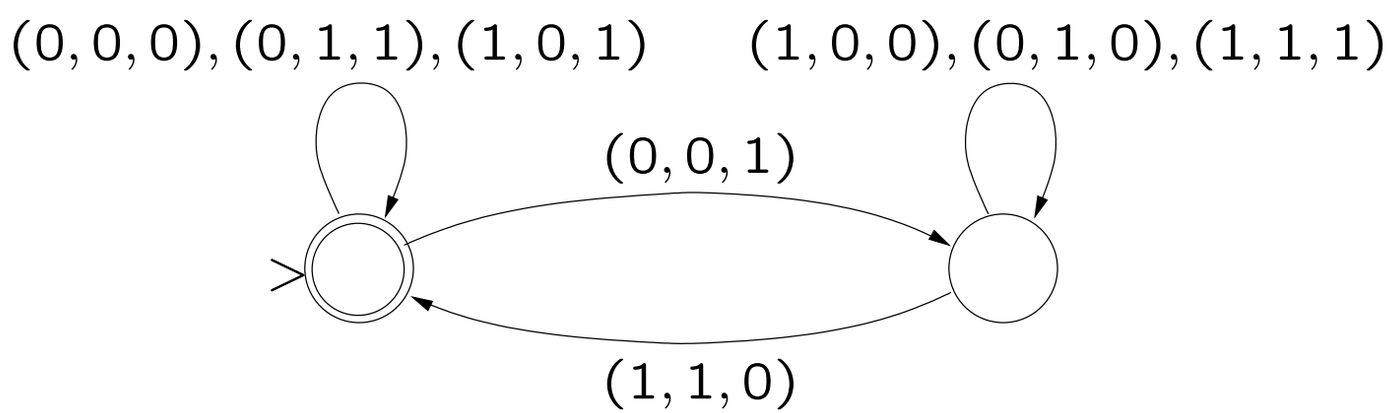
- L'ensemble des encodages en binaire des vecteurs (x, y) tels que $x = y$ est accepté par l'automate



- Vecteurs (x, y) tels que $x < y$



- Vecteurs à trois composantes (x, y, z) tels que $z = x + y$



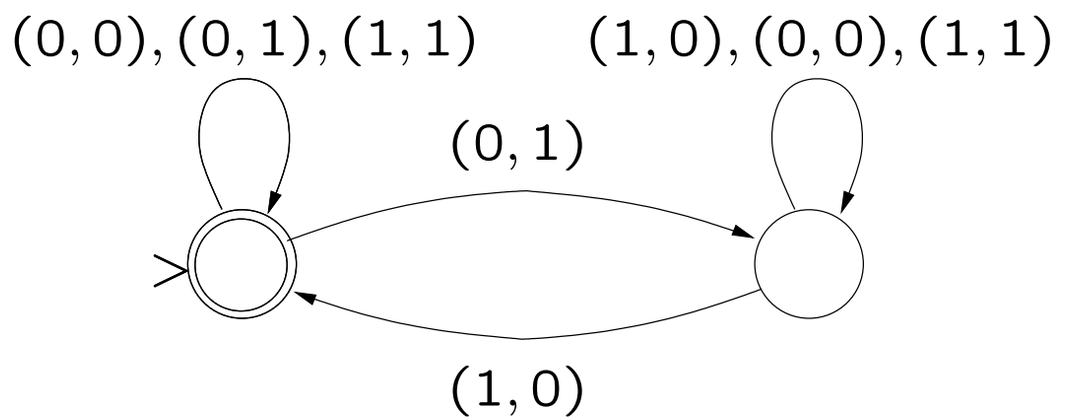
Ensembles de vecteurs de nombres définissables (suite)

- Intersection, union, complément des ensembles représentables (fermeture des langages réguliers).

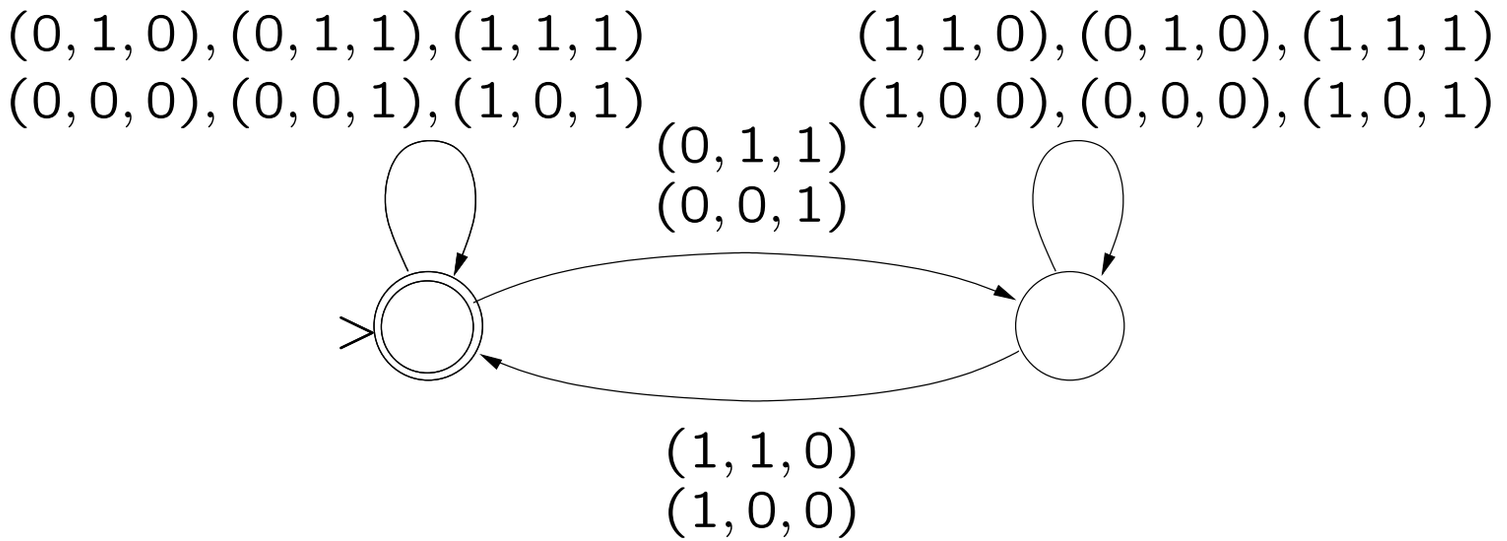
- Modifier le nombre de composantes: projection et opération inverse.

- **Remarque:** l'opération de projection ne préserve pas nécessairement le caractère déterministe des automates.

- **Exemple:** $\{(x, z) \mid \exists y\ x + y = z\}$ $(x \leq z)$.



- Ajouter une composante à l'automate précédent donne



- qui n'est pas équivalent à l'automate auquel la projection a été appliquée.

Ensembles de vecteurs représentables: conclusions

- Contraintes d'égalité ou inégalité linéaires

- **Exemple:** un automate pour $x + 2y = 5$ peut être obtenu en combinant les automates représentant les contraintes suivantes:

$$\begin{aligned} z_1 &= y \\ z_2 &= y + z_1 \\ z_3 &= x + z_2 \\ z_3 &= 5. \end{aligned}$$

- Il existe aussi une construction plus directe.

Ensembles de vecteurs représentables: conclusions (suite)

- Combinaisons booléennes de contraintes linéaires
- La projection permet de traiter la quantification existentielle ($\exists x$).
- Pour la quantification universelle, on utilise $\forall x f \equiv \neg\exists\neg f$
- **Exemple:** Il est possible de construire un automate acceptant les représentations des vecteurs (x, y) satisfaisant la contrainte arithmétique

$$\forall u \exists t[(2x + 3y + t - 4u = 5) \vee (x + 5y - 3t + 2u = 8)]$$

- Cela correspond à l'arithmétique de Presburger qui définit les ensembles représentables par automates, quelle que soit la base.

Chapitre 4
Automates à pile
et langages hors-contexte

Introduction

- Langage $a^n b^n$ n'est pas accepté par un automate fini.
- Par contre $L_k = \{a^n b^n \mid n \leq k\}$ est accepté.
- Mémoire finie, mémoire infinie, mémoire extensible.
- Automates à pile: mémoire LIFO.

4.1 Les automates à pile

- un ruban d'entrée et une tête de lecture,
- un ensemble d'états parmi lesquels on distingue un état initial et des états accepteurs,
- une relation de transition.
- une pile de capacité infinie.

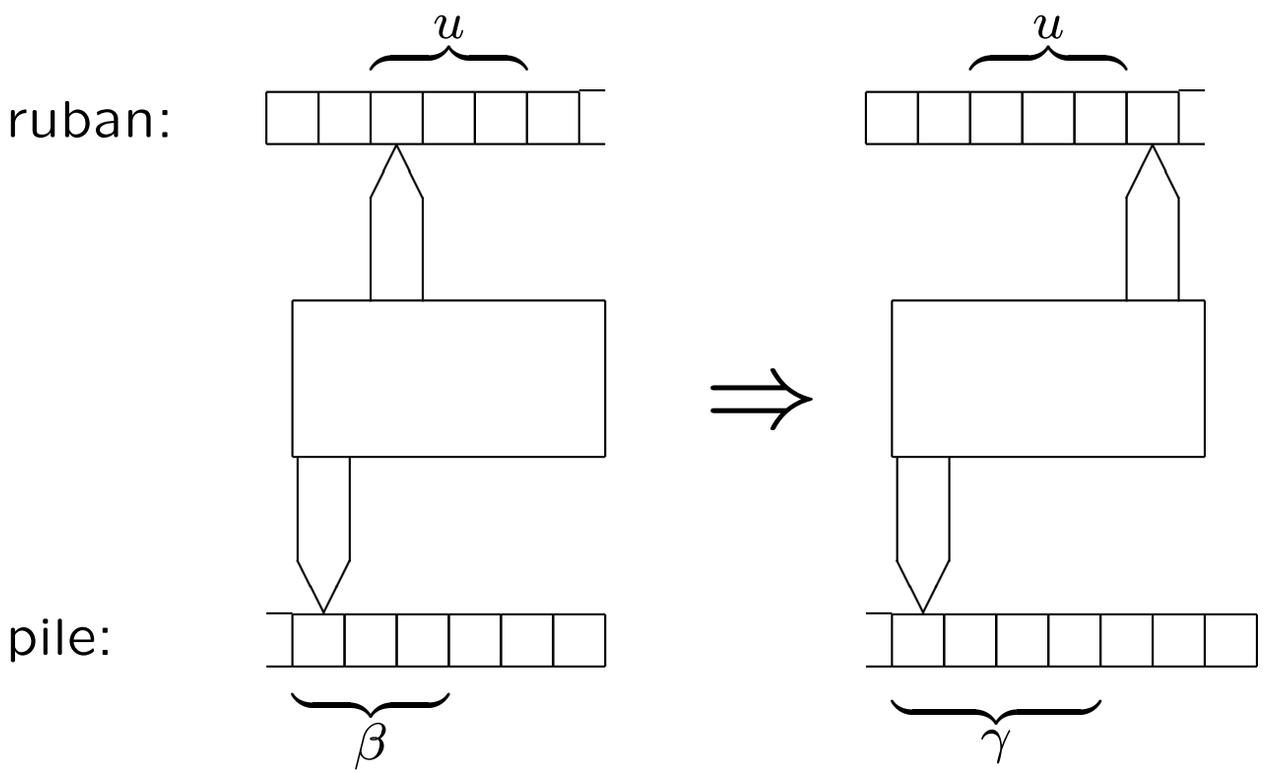
Formalisation

Septuplet $M = (Q, \Sigma, \Gamma, \Delta, Z, s, F)$, où

- Q est un ensemble fini d'états,
- Σ est un *alphabet d'entrée*,
- Γ est un *alphabet de pile*,
- $Z \in \Gamma$ est le *symbole initial de pile*,
- $s \in Q$ est l'état initial,
- $F \subseteq Q$ est l'ensemble des états accepteurs,
- $\Delta \subset ((Q \times \Sigma^* \times \Gamma^*) \times (Q \times \Gamma^*))$ est la relation de transition.

Transition

$$((p, u, \beta), (q, \gamma)) \in \Delta$$



Exécutions

La configuration (q', w', α') est *dérivable en une étape* de la configuration (q, w, α) par la machine M (notation $(q, w, \alpha) \vdash_M (q', w', \alpha')$) si

- $w = uw'$ (le mot w commence par le préfixe $u \in \Sigma^*$),
- $\alpha = \beta\delta$ (avant la transition le sommet de la pile lu de gauche à droite contient $\beta \in \Gamma^*$),
- $\alpha' = \gamma\delta$ (après la transition la partie β de la pile a été remplacée par γ, le premier symbole de γ est maintenant le sommet de la pile),
- $((q, u, \beta), (q', \gamma)) \in \Delta$.

Une configuration C' est *dérivable en plusieurs étapes* de la configuration C par la machine M (notation $C \vdash_M^* C'$) s'il existe $k \geq 0$ et des configurations intermédiaires $C_0, C_1, C_2, \ldots, C_k$ telles que

- $C = C_0$,
- $C' = C_k$,
- $C_i \vdash_M C_{i+1}$ pour $0 \leq i < k$.

Une *exécution d'un automate à pile* sur un mot w est une suite de configurations

$$(s, w, Z) \vdash (q_1, w_1, \alpha_1) \vdash \cdots \vdash (q_n, \varepsilon, \gamma)$$

où s est l'état initial, Z est le symbole initial de pile et ε représente le mot vide.

Un mot w est *accepté par un automate à pile* $M = (Q, \Sigma, \Gamma, \Delta, Z, s, F)$ si

$$(s, w, Z) \vdash_M^* (p, \varepsilon, \gamma), \text{avec } p \in F.$$

Exemples

$$\{a^n b^n \mid n \geq 0\}$$

- $Q = \{s, p, q\}$,
- $\Sigma = \{a, b\}$,
- $\Gamma = \{A\}$,
- $F = \{q\}$ et Δ contient les transitions

$$(s, a, \varepsilon) \rightarrow (s, A)$$
$$(s, \varepsilon, Z) \rightarrow (q, \varepsilon)$$
$$(s, b, A) \rightarrow (p, \varepsilon)$$
$$(p, b, A) \rightarrow (p, \varepsilon)$$
$$(p, \varepsilon, Z) \rightarrow (q, \varepsilon)$$

L'automate $M = (Q, \Sigma, \Gamma, \Delta, Z, s, F)$ décrit ci-dessous accepte le langage

$$\{ww^R\}$$

- $Q = \{s, p, q\}$,
- $\Sigma = \{a, b\}$,
- $\Gamma = \{A, B\}$,
- $F = \{q\}$ et Δ contient les transitions

$$(s, a, \varepsilon) \to (s, A)$$
$$(s, b, \varepsilon) \to (s, B)$$
$$(s, \varepsilon, \varepsilon) \to (p, \varepsilon)$$
$$(p, a, A) \to (p, \varepsilon)$$
$$(p, b, B) \to (p, \varepsilon)$$
$$(p, \varepsilon, Z) \to (q, \varepsilon)$$

Les langages hors-contexte

Définition:
Un langage est hors-contexte s'il existe une grammaire hors-contexte qui génère ce langage.

Exemples

Le langage $a^n b^n$, $n \geq 0$, est généré par la grammaire dont les productions sont

1. $S \rightarrow aSb$

2. $S \rightarrow \varepsilon$.

Le langage des mots de la forme ww^R est généré par la grammaire dont les productions sont

1. $S \to aSa$
2. $S \to bSb$
3. $S \to \varepsilon$.

Le langage généré par la grammaire dont les productions sont

1. $S \to \varepsilon$
2. $S \to aB$
3. $S \to bA$
4. $A \to aS$
5. $A \to bAA$
6. $B \to bS$
7. $B \to aBB$

est le langage de tous les mots comportant le même nombre de lettres a et de lettres b dans un ordre quelconque.

Relation avec les automates à pile

Théorème

Un langage est hors-contexte si et seulement si il est accepté par un automate à pile.

Propriétés des langages hors-contexte

Soit L_1 et L_2 deux langages hors-contexte.

- Le langage $L_1 \cup L_2$ est hors-contexte.
- Le langage $L_1 \cdot L_2$ est hors-contexte.
- L_1^* est hors-contexte.
- $L_1 \cap L_2$ et $\overline{L_1}$ ne sont pas forcément hors-contexte !
- Si L_R est régulier et si le langage L_2 est hors-contexte $L_R \cap L_2$ est hors-contexte.

Soit $M_R = (Q_R, \Sigma_R, \delta_R, s_R, F_R)$ un automate fini déterministe acceptant L_R et $M_2 = (Q_2, \Sigma_2, \Gamma_2, \Delta_2, Z_2, s_2, F_2)$ un automate à pile acceptant le langage L_2. Le langage $L_R \cap L_2$ est accepté par l'automate à pile $M = (Q, \Sigma, \Gamma, \Delta, Z, s, F)$ où

- $Q = Q_R \times Q_2$,
- $\Sigma = \Sigma_R \cup \Sigma_2$,
- $\Gamma = \Gamma_2$,
- $Z = Z_2$,
- $s = (s_R, s_2)$,
- $F = (F_R \times F_2)$,

- $(((q_R, q_2), u, \beta), ((p_R, p_2), \gamma)) \in \Delta$ si et seulement si

  $(q_R, u) \vdash^*_{M_R} (p_R, \varepsilon)$ (l'automate M_R peut passer de l'état q_R à l'état p_R en consommant le mot u, ce passage se faisant en une ou plusieurs étapes) et

  $((q_2, u, \beta), (p_2, \gamma)) \in \Delta_2$ (l'automate à pile peut passer de l'état q_2 à l'état p_2 en consommant le mot u et en remplaçant la partie β de la pile par γ).

4.3 Au-delà des langages hors-contexte

- Il existe des langages qui ne sont pas hors-contexte (argument de cardinalité).

- On désire montrer que certains langages particuliers ne sont pas hors-contexte.

- Pour cela, nous allons établir une forme de théorème de gonflement.

- Il est nécessaire d'abstraire la notion de dérivation pour établir un tel théorème.

Exemple

1. $S \to SS$
2. $S \to aSa$
3. $S \to bSb$
4. $S \to \varepsilon$

Génération de $aabaab$:

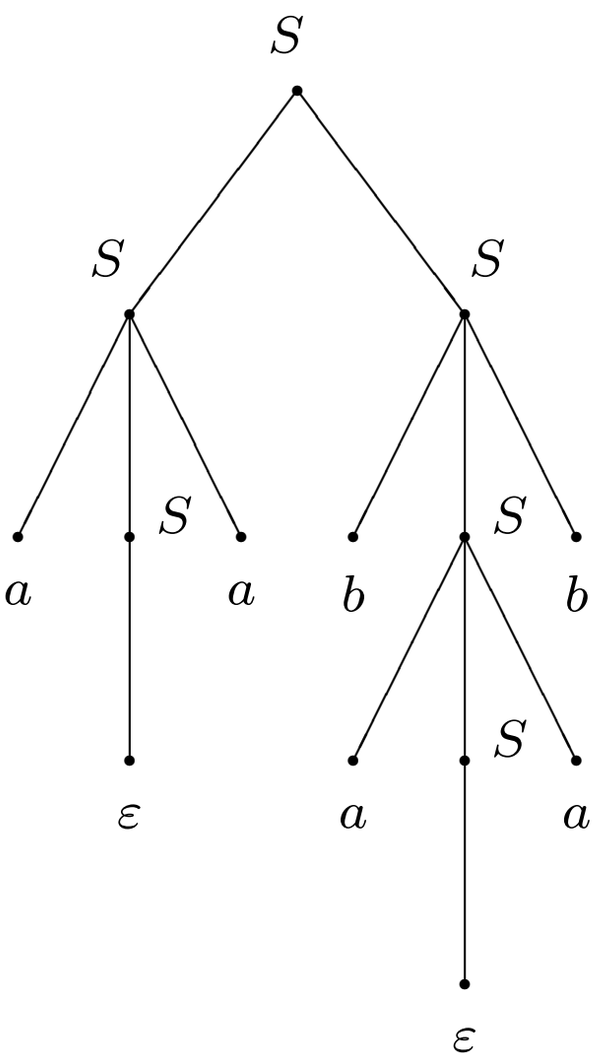
$$S \Rightarrow SS \Rightarrow aSaS \Rightarrow aaS$$
$$\Rightarrow aabSb \Rightarrow aabaSab \Rightarrow aabaab$$

$$S \Rightarrow SS \Rightarrow SbSb \Rightarrow SbaSab$$
$$\Rightarrow Sbaab \Rightarrow aSabaab \Rightarrow aabaab$$

et encore de 8 autres façons.

On désire une représentation des dérivations qui ne tienne pas compte de l'ordre d'application des productions.

La notion d'arbre d'analyse



Arbre d'analyse pour $aababaab$

Définition

Un arbre d'analyse pour une grammaire hors-contexte $G = (V, \Sigma, R, S)$ est un arbre dont chaque nœud est étiqueté par un élément de $V \cup \varepsilon$ et qui satisfait les conditions suivantes.

- La racine est étiquetée par le symbole de départ S.
- Chaque nœud intérieur est étiqueté par un non-terminal. Chaque feuille est étiquetée par un symbole terminal ou par ε.

- Pour tout nœud intérieur, si son étiquette est le non-terminal A et si ses descendants immédiats sont les nœuds $n_1, n_2, \ldots, n_k$ ayant respectivement pour étiquettes $X_1, X_2, \ldots, X_k$, alors

  $$A \to X_1 X_2 \ldots X_k$$

  doit être une production de G.

- Si un nœud est étiqueté par ε, alors ce nœud est le seul descendant immédiat de son prédécesseur (cette dernière règle évite simplement l'introduction d'instances inutiles de ε dans l'arbre d'analyse).

Mot généré

Le mot généré par un arbre d'analyse est celui obtenu par la concaténation des feuilles de l'arbre prises de gauche à droite.

Théorème
Etant donné une grammaire hors-contexte G, un mot w est généré par G ($S \overset{*}{\underset{G}{\Rightarrow}} w$) si et seulement si il existe un arbre d'analyse de la grammaire G qui génère w.

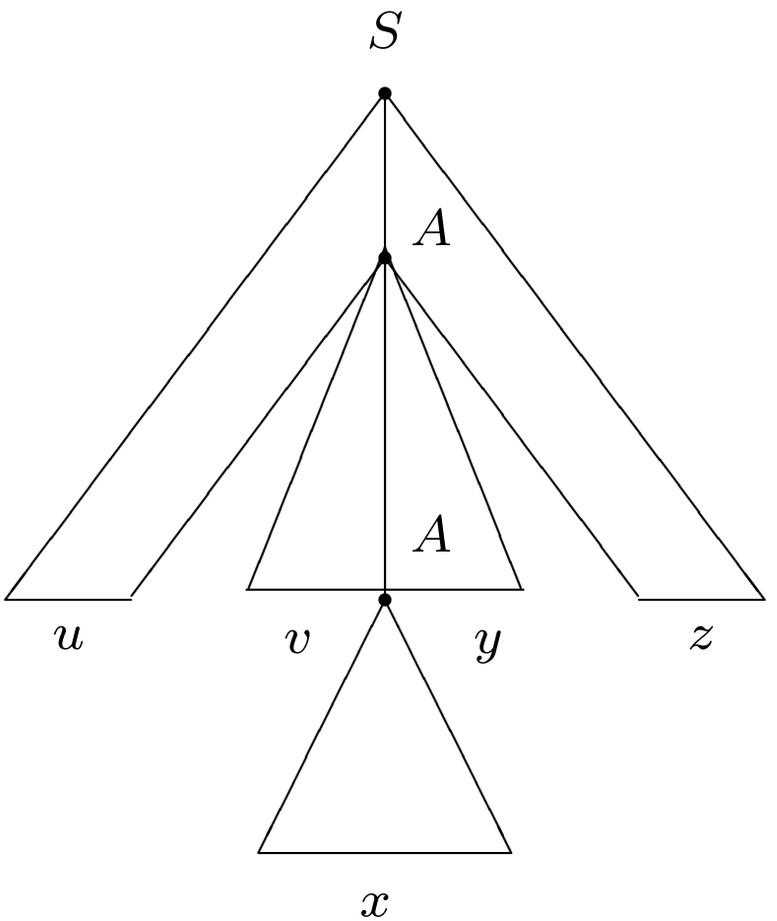
Le théorème du gonflement

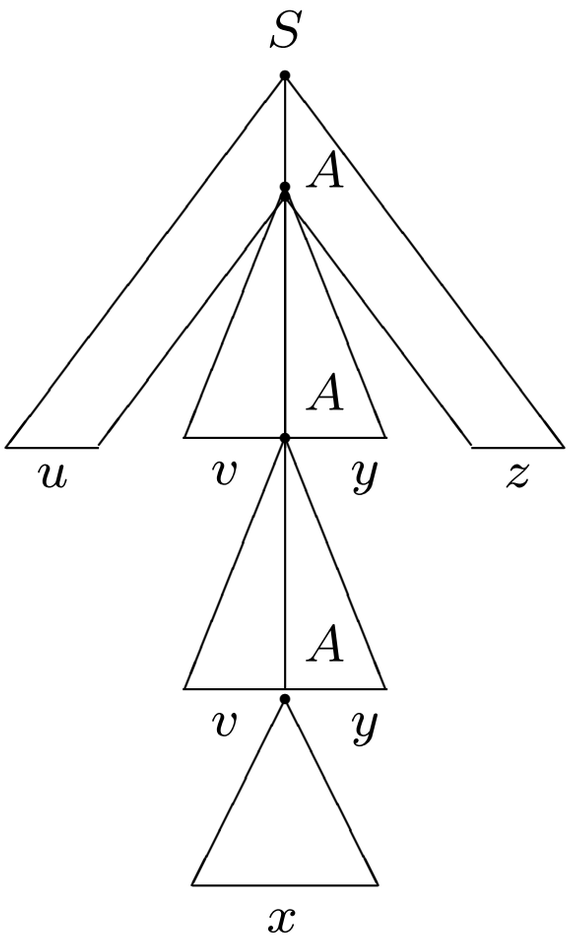
Théorème

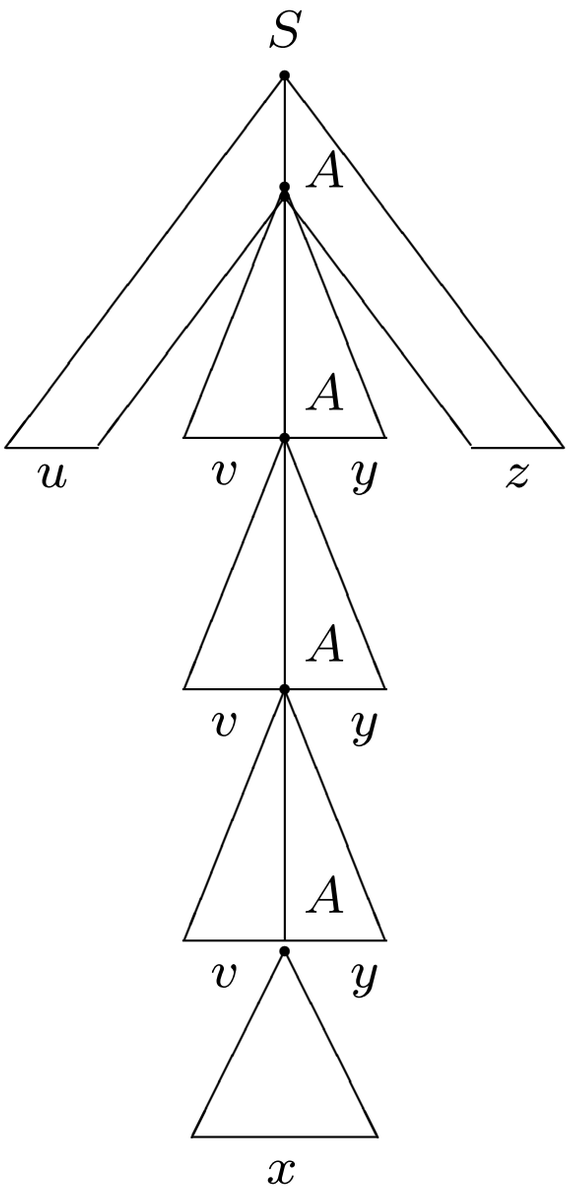
Soit un langage hors-contexte L. Alors, il existe une constante K telle que tout mot $w \in L$ satisfaisant $|w| \geq K$ peut être écrit $w = uvxyz$ avec v ou $y \neq \varepsilon$, $|vxy| \leq K$ et $uv^n xy^n z \in L$ pour tout $n > 0$.

Démonstration

Un arbre d'analyse pour G correspondant à un mot suffisamment long doit contenir un chemin sur lequel *le même non-terminal apparaît au moins deux fois*


S
A
A
u
v
y
z
x


S
A
A
u
v
y
z
A
v
y
x


S
A
A
u
v
y
z
A
v
y
A
v
y
x

Choix de K

- $p = max\{|\alpha|, A \rightarrow \alpha \in R\}$
- La longueur maximale d'un mot généré par un arbre de profondeur i est p^i.
- On choisit $K = p^{m+1}$ où $m = |\{V - \Sigma\}|$.
- Donc $|w| > p^m$ et l'arbre d'analyse contient des chemins de longueur $\geq m + 1$ contenant un même non terminal au moins deux fois.
- En remontant le plus long de ces chemins un non-terminal sera rencontré pour la 2ème fois après au plus $m + 1$ arcs. Donc on peut choisir vxy de longueur au plus $p^{m+1} = K$.
- Note: v et y pas tous deux vides pour tous les chemins de longueur supérieure à $m + 1$, car sinon le mot généré sera de longueur inférieure à p^{m+1}.

Applications du théorème du gonflement

$L = \{a^n b^n c^n\}$ n'est pas hors-contexte.

Démonstration
Il n'existe pas de décomposition de $a^n b^n c^n$ en 5 parties u, v, x, y et z (v ou y non vide) telles que, pour tout $j > 0$, $uv^j xy^j z \in L$ et donc que le théorème du gonflement est contredit.

- v et y sont constitués de la répétition d'une même lettre. Impossible

- v et y sont constitués de lettres différentes. Impossible.

1. Il existe deux langages hors-contexte L_1 et L_2 tels que $L_1 \cap L_2$ n'est pas hors-contexte :

   - $L_1 = \{a^n b^n c^m\}$ est hors-contexte,
   - $L_2 = \{a^m b^n c^n\}$ est hors-contexte mais
   - $L_1 \cap L_2 = \{a^n b^n c^n\}$ n'est pas hors-contexte !

2. Le complément d'un langage hors-contexte n'est pas toujours hors-contexte. En effet, l'union de deux langages hors-contexte est hors-contexte. Donc si le complément était hors-contexte, l'intersection le serait aussi:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}.$$

Algorithmes pour langages hors-contexte

Soit L un langage hors-contexte (défini par une grammaire hors-contexte ou par un automate à pile).

1. Etant donné un mot w, il existe un algorithme qui détermine si $w \in L$.

2. Il existe un algorithme qui détermine si $L = \emptyset$.

3. Il n'existe pas d'algorithme qui détermine si $L = \Sigma^*$.

4. Si L' est aussi un langage hors-contexte, il n'existe pas d'algorithme qui détermine si $L \cap L' = \emptyset$.

Théorème
Etant donné une grammaire hors-contexte G, il existe un algorithme pour déterminer si un mot donné w appartient à $L(G)$.

Démonstration

- Automate à pile ? Non car non déterministe et transitions sur le mot vide.

- Idée: borner la longueur des exécutions. Sera fait dans le cadre des grammaires (borne sur la longueur des dérivations).

Hypothèse: dérivations bornées

Pour déterminer si $w \in L(G)$:

1. On calcule la borne k sur le nombre d'étapes nécessaires à la dérivation d'un mot de la longueur de w.
2. On construit systématiquement toutes les dérivations de longueur inférieure ou égale à k. Il y en a un nombre fini.
3. Si une de ces dérivations génère le mot w, alors le mot est généré par la grammaire G. Si ce n'est pas le cas, ce mot ne peut pas être généré par la grammaire G et n'appartient donc pas à $L(G)$.

Grammaires à dérivations bornées

Problème:

$$A \to B$$
$$B \to A$$

Solution: Grammaire satisfaisant

1. $A \to \sigma$ avec σ terminal, ou
2. $A \to v$ avec $|v| \geq 2$.
3. Exception: $S \to \varepsilon$

Borne: $2 \times |w| - 1$

Obtenir une grammaire à dérivations bornées

1. Eliminer les productions $A \to \varepsilon$.

Si $A \to \varepsilon$ et $B \to vAu$ on ajoute $B \to vu$. On élimine $A \to \varepsilon$.

Si on élimine $S \to \varepsilon$, on introduit un nouveau symbole de départ S' et on ajoute la production $S' \to \varepsilon$ et $S' \to \alpha$ pour chaque production de la forme $S \to \alpha$.

2. Eliminer les productions $A \to B$.

Pour chaque paire de non-terminaux A et B on détermine si $A \stackrel{*}{\Rightarrow} B$.

Si c'est le cas et pour chaque production de la forme $B \to u$ ($u \notin V - \Sigma$), on ajoute la production $A \to u$.

On élimine toutes les productions de la forme $A \to B$.

Théorème
Etant donné une grammaire hors-contexte G, il existe un algorithme pour déterminer si $L(G) = \emptyset$.

- Idée: chercher un arbre d'analyse pour G.
- On peut construire les arbres d'analyse par ordre de profondeur croissante.
- La profondeur des arbres peut être limitée à $|V - \Sigma|$.

Les automates à pile déterministes

Deux transitions $((p_1, u_1, \beta_1), (q_1, \gamma_1))$ et $((p_2, u_2, \beta_2), (q_2, \gamma_2))$ sont compatibles si

1. $p_1 = p_2$,

2. u_1 et u_2 compatibles (c'est-à-dire u_1 est un préfixe de u_2 ou u_2 est un préfixe de u_1),

3. β_1 et β_2 compatibles.

Un automate à pile est déterministe si pour toute paire de transitions compatibles, ces transitions sont identiques.

Les langages hors-contexte déterministes

Soit L un langage défini sur un alphabet Σ, le langage L est un langage hors-contexte déterministe si et seulement si il est accepté par un automate à pile déterministe.

- Tous les langages hors-contextes ne sont pas hors-contexte déterministes.
- $L_1 = \{wcw^R \mid w \in \{a, b\}^*\}$ est déterministe hors-contexte.
- $L_2 = \{ww^R \mid w \in \{a, b\}^*\}$ est hors-contexte, mais pas déterministe hors-contexte.

Propriétés des langages hors-contexte déterministes

Si L_1 et L_2 sont des langages hors-contexte déterministes,

- $\Sigma^* - L_1$ est aussi hors-contexte déterministe.
- Il existe des langages hors-contexte qui ne sont pas déterministes hors-contexte.
- Les langages $L_1 \cup L_2$ et $L_1 \cap L_2$ ne sont pas forcément déterministes hors-contexte.

Applications

- Description et analyse syntaxique des langages de programmation.
- Restriction à des langages hors-contexte déterministes.
- Familles particulières de grammaires: LR.

Chapitre 5
Les machines de Turing

5.1 Introduction

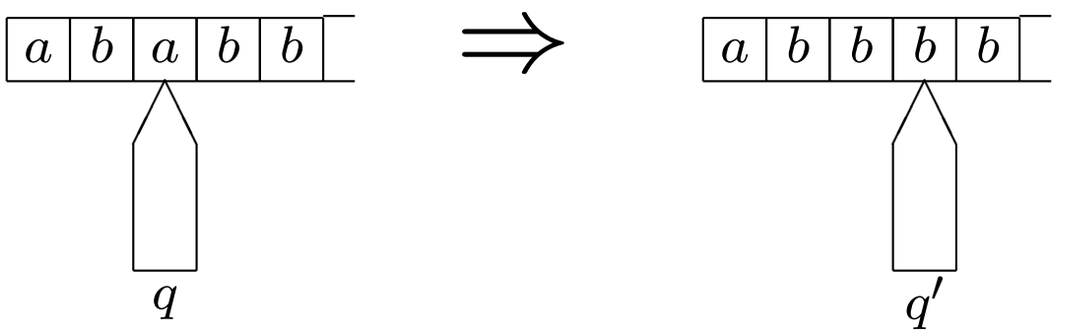
- Langage $a^n b^n c^n$ n'est pas accepté par un automate à pile.
- Machines à mémoire infinie dont l'accès n'est pas LIFO.
- Modélisation de la notion de procédure effective.
- Justification : extensions n'apportent rien; autres formalisations.

5.2 Définition

- Mémoire infinie sous forme de ruban divisé en cases; alphabet de ruban.
- Tête de lecture.
- Ensemble fini d'états. Etats accepteurs.
- Fonction de transition qui pour chaque état de la machine et symbole se trouvant sous la tête de lecture précise
  - l'état suivant,
  - un caractère qui sera écrit sur le ruban,
  - un sens de déplacement de la tête de lecture.

Exécution

- Initialement, mot d'entrée au début du ruban, symbole blanc partout ailleurs, tête de lecture sur la première case.
- A chaque étape, la machine
  - lit le symbole se trouvant sous sa tête de lecture,
  - remplace ce symbole suivant la fonction de transition,
  - déplace sa tête de lecture d'une case vers la gauche ou vers la droite suivant la fonction de transition,
  - change d'état comme indiqué par la fonction de transition.
- Mot accepté si état accepteur atteint.


a b a b b
q
a b b b b
q′

Formalisation

Septuplet $M = (Q, \Gamma, \Sigma, \delta, s, B, F)$, où :

- Q est un ensemble fini d'états,
- Γ est l'alphabet de ruban,
- $\Sigma \subseteq \Gamma$ est l'alphabet d'entrée,
- $s \in Q$ est l'état initial,
- $F \subseteq Q$ est l'ensemble des états accepteurs,
- $B \in \Gamma - \Sigma$ est le "symbole blanc" (#),
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ est la fonction de transition.

Configuration

Information nécessaire:

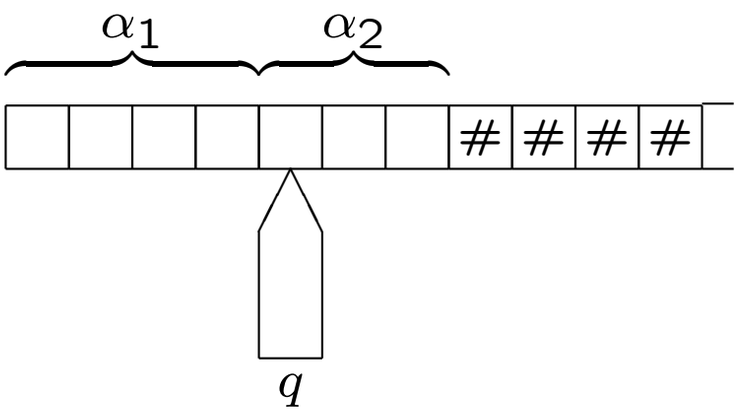
1. l'état,
2. le contenu du ruban,
3. la position de la tête de lecture.

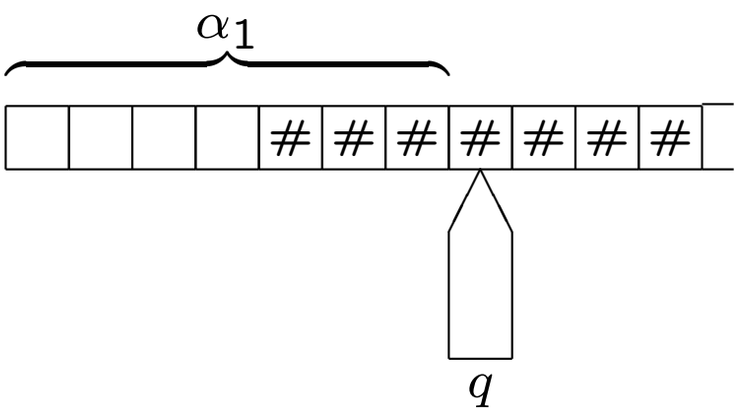
Représentation : triplet contenant

1. l'état de la machine,
2. le mot sur le ruban avant la tête de lecture,
3. le mot sur le ruban après la tête.

Formellement, élément de $Q \times \Gamma^* \times (\varepsilon \cup \Gamma^*(\Gamma - \{B\}))$.

Configurations (q, α_1, α_2) et $(q, \alpha_1, \varepsilon)$.





Dérivation

Configuration (q, α_1, α_2) écrite sous la forme $(q, \alpha_1, b\alpha_2')$ avec $b = \#$ si $\alpha_2 = \varepsilon$.

- Si $\delta(q, b) = (q', b', R)$ nous avons

$$(q, \alpha_1, b\alpha_2') \vdash_M (q', \alpha_1 b', \alpha_2').$$

- Si $\delta(q, b) = (q', b', L)$ et si $\alpha_1 \neq \varepsilon$ et est donc de la forme $\alpha_1' a$ nous avons

$$(q, \alpha_1' a, b\alpha_2') \vdash_M (q', \alpha_1', ab'\alpha_2').$$

Dérivation

Une configuration C' est dérivable en plusieurs étapes de la configuration C par la machine M ($C \vdash_M^* C'$) s'il existe $k \geq 0$ et des configurations intermédiaires $C_0, C_1, C_2, \ldots, C_k$ telles que

- $C = C_0$,
- $C' = C_k$,
- $C_i \vdash_M C_{i+1}$ pour $0 \leq i < k$.

Le langage $L(M)$ accepté par une machine de Turing est l'ensemble des mots w tels que

$$(s, \varepsilon, w) \vdash_M^* (p, \alpha_1, \alpha_2), \text{ avec } p \in F.$$

Exemple

Machine de Turing $M = (Q, \Gamma, \Sigma, \delta, s, B, F)$ avec

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
- $\Gamma = \{a, b, X, Y, \#\}$,
- $\Sigma = \{a, b\}$,
- $s = q_0$,
- $B = \#$,
- δ donné par

	a	b	X	Y	$\#$
q_0	(q_1, X, R)	$-$	$-$	(q_3, Y, R)	$-$
q_1	(q_1, a, R)	(q_2, Y, L)	$-$	(q_1, Y, R)	$-$
q_2	(q_2, a, L)	$-$	(q_0, X, R)	(q_2, Y, L)	$-$
q_3	$-$	$-$	$-$	(q_3, Y, R)	$(q_4, \#, R)$
q_4	$-$	$-$	$-$	$-$	$-$

M accepte $a^n b^n$. Par exemple son exécution sur $aaabbb$ est

$(q_0, \varepsilon, aaabbb)$	$\vdots$
$(q_1, X, aabbb)$	$(q_1, XXXYY, b)$
$(q_1, Xa, abbb)$	$(q_2, XXXY, YY)$
(q_1, Xaa, bbb)	(q_2, XXX, YYY)
$(q_2, Xa, aYbb)$	$(q_2, XX, XYYY)$
$(q_2, X, aaYbb)$	(q_0, XXX, YYY)
$(q_2, \varepsilon, XaaYbb)$	$(q_3, XXXY, YY)$
$(q_0, X, aaYbb)$	$(q_3, XXXYY, Y)$
$(q_1, XX, aYbb)$	$(q_3, XXXYYY, \varepsilon)$
	$(q_4, XXXYYY\#, \varepsilon)$

Langage accepté
Langage décidé

Machine de Turing = procédure effective ? Pas toujours. Les situations possibles sont les suivantes.

1. La suite des configurations contient un état accepteur.
2. La suite de configurations se termine parce que soit
   - la fonction de transition n'est pas définie, soit
   - elle impose un déplacement à gauche au début du ruban.
3. La suite de configurations ne passe jamais par un état final et est infinie.

Dans les cas 1 et 2 on a une procédure effective, dans le cas 3 pas.

L'*exécution* d'une machine de Turing sur un mot w est la suite de configurations

$$(s, \varepsilon, w) \vdash_M C_1 \vdash_M C_2 \vdash_M \cdots \vdash_M C_k \vdash_M \ldots$$

maximale, c'est-à-dire telle que soit

- elle est infinie,
- elle se termine dans une configuration dont l'état est accepteur, ou
- elle se termine par une configuration à partir de laquelle aucune configuration n'est dérivable.

Langage décidé : Un langage L est décidé par une machine de Turing M si

- M accepte L,
- M n'a pas d'exécution infinie.

Langage décidé

Automates déterministes !

- Automates finis déterministes : les langages acceptés sont les mêmes que les langages décidés.
- Automates finis non déterministes : pas de sens.
- Automates à pile non-déterministes : pas de sens, mais langages hors-contexte décidés par une machine de Turing.
- Automate à pile déterministe : langage décidé sauf si exécution infinie (boucle sur des transitions ε).

Autres définitions des machines de Turing

1. Un seul *état d'arrêt* et fonction de transition est partout définie. Dans l'état d'arrêt, résultat sur le ruban : "accepte" (1) ou "n'accepte pas" (0).

2. Deux états d'arrêt : q_Y et q_N, et fonction de transition est partout définie.

Langages récursifs et récursivement énumérables

Un langage est *récursif* s'il est décidé par une machine de Turing.

Un langage est *récursivement énumérable* s'il est accepté par une machine de Turing.

Thèse de Turing-Church

Les langages reconnus par une procédure effective sont ceux décidés par une machine de Turing.

Justification.

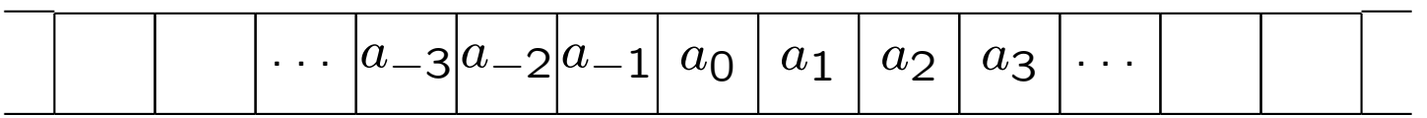
1. Si décidé par machine de Turing, alors calculable : immédiat.

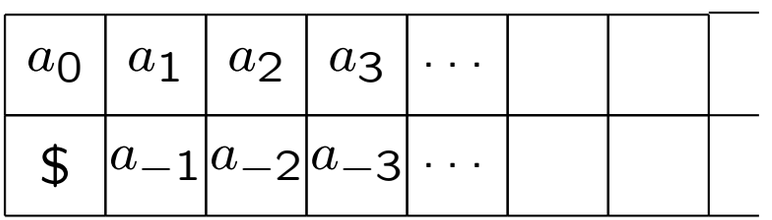
2. Si calculable alors décidé par machine de Turing :

   - Extensions des machines de Turing et autres machines.
   - Autres modélisations.

Extensions des machines de Turing

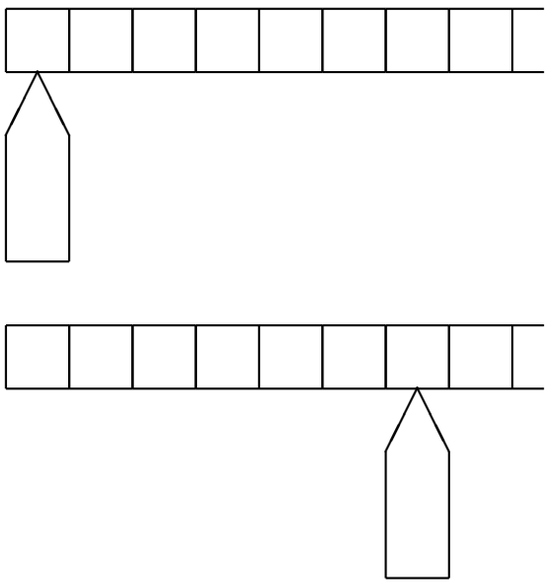
Ruban infini dans les deux sens





Rubans multiples

Plusieurs rubans et plusieurs têtes :

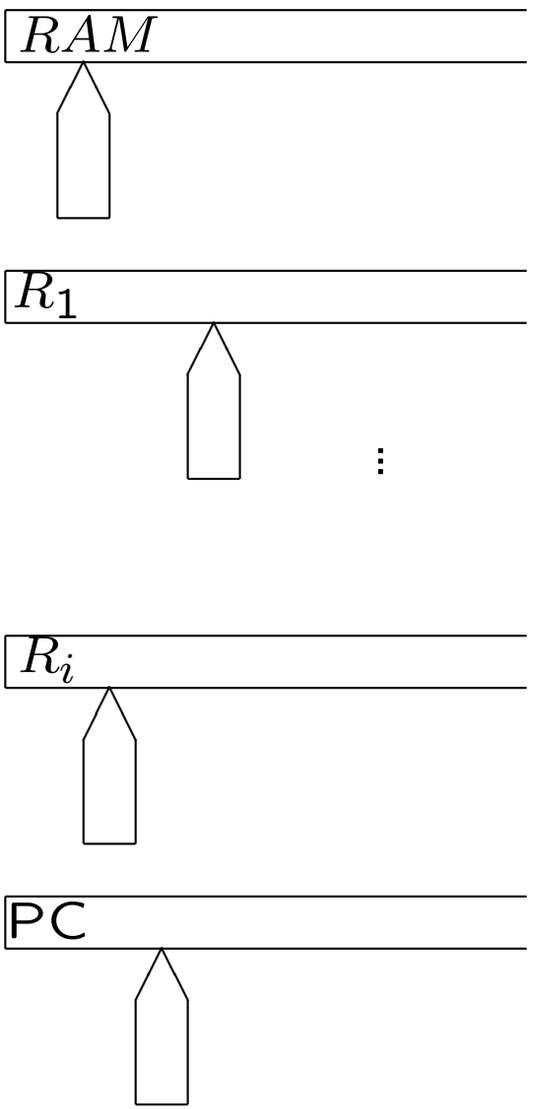


Simulation (2 rubans) : alphabet = quadruplet.

- Deux éléments représentent le contenu des rubans.
- Deux éléments représentent la position des têtes.

Machines à mémoire à accès direct

Un ruban pour la mémoire et un par registre.



Simulation :

#	0	$*$	v_0	#	1	$*$	v_1	#	...	#	a	d	d	i	$*$	v_i	#	

Machines de Turing non déterministes

Relation de transition :

$$\Delta : (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$$

L'exécution n'est plus unique.

Elimination du non-déterminisme

Théorème

Tout langage accepté par une machine de Turing non déterministe est aussi accepté par une machine de Turing déterministe.

Preuve

Simuler les exécutions par ordre de longueur croissante.

$$r = \max_{q \in Q, a \in \Gamma} |\{((q,a),(q',x,X)) \in \Delta\}|.$$

Machine à trois rubans :

1. Le premier ruban contient le mot d'entrée et n'est pas modifié.
2. Le deuxième ruban servira à contenir des séquences de nombres inférieurs à r.
3. Le troisième ruban sert à la machine déterministe pour simuler le comportement de la machine non déterministe.

La machine déterministe procède alors comme suit.

1. Sur le deuxième ruban, elle génère toutes les séquences finies de nombres inférieurs à r. Ces séquences sont générées par ordre de longueur croissante.

2. Pour chacune de ces séquences, elle simule la machine non déterministe en utilisant les choix représentés par la séquence.

3. Elle s'arrête et accepte dès que la simulation d'une exécution de la machine non déterministe atteint un état accepteur.

Machines de Turing universelles

Machine de Turing qui simule n'importe quelle machine de Turing.

- Machine de Turing M.
- Données pour M : M' et un mot w.
- M simule l'exécution de M' sur w.

Fonctions calculables par une machine de Turing

Une machine de Turing calcule une fonction $f : \Sigma^* \to \Sigma^*$ si, pour tout mot d'entrée w, elle s'arrête toujours dans une configuration où $f(w)$ se trouve sur le ruban.

Les fonctions calculables par une procédure effective sont les fonctions calculables par une machine de Turing

Chapitre 6
Les fonctions récursives

6.1 Introduction

- Autre formalisation de la notion de procédure effective : fonctions calculables sur les naturels.

- Fonctions calculables ?
  - Fonctions de base.
  - Composition de fonctions.
  - Mécanisme de récursion.

6.2 Fonctions primitives récursives

Fonctions dans l'ensemble $\{N^k \rightarrow N \mid k \geq 0\}$.

1. Fonctions primitives récursives de base.

   1. $\mathbf{0}()$

   2. $\pi_i^k(n_1, \ldots, n_k)$

   3. $\sigma(n)$

2. Composition de fonctions.

- Soit g fonction à ℓ arguments,
- $h_1, \ldots, h_\ell$ des fonctions à k arguments.
- $f(\overline{n}) = g(h_1(\overline{n}), \ldots, h_\ell(\overline{n}))$ est la composition de g et des fonctions h_i.

3. Récursion primitive.

- Soit g une fonction à k arguments et h une fonction à $k+2$ arguments.

- $$\begin{aligned} f(\overline{n}, 0) &= g(\overline{n}) \\ f(\overline{n}, m+1) &= h(\overline{n}, m, f(\overline{n}, m)) \end{aligned}$$

  est la fonction définie à partir de g et h par récursion primitive.

- Remarque: f est calculable si g et h sont calculables.

Définition

Les *fonctions primitives récursives* sont :

- les fonctions primitives récursives de base ;
- toutes les fonctions obtenues à partir des fonctions primitives récursives de base par un nombre quelconque d'applications de la composition et de la récursion primitive.

Exemples

Fonctions constantes :

$$\mathbf{j}() \;=\; \overbrace{\sigma(\sigma(\ldots\sigma(}^{j}\mathbf{0}()))))$$

Fonction d'addition :

$$\begin{array}{lcl} plus(n_1, 0) & = & \pi_1^1(n_1) \\ plus(n_1, n_2 + 1) & = & \sigma(\pi_3^3(n_1, n_2, plus(n_1, n_2))) \end{array}$$

Notation simplifiée :

$$\begin{array}{lcl} plus(n_1, 0) & = & n_1 \\ plus(n_1, n_2 + 1) & = & \sigma(plus(n_1, n_2)) \end{array}$$

Evaluation de $plus(4, 7)$:

$$\begin{aligned} plus(7,4) &= plus(7, 3+1) \\ &= \sigma(plus(7,3)) \\ &= \sigma(\sigma(plus(7,2))) \\ &= \sigma(\sigma(\sigma(plus(7,1)))) \\ &= \sigma(\sigma(\sigma(\sigma(plus(7,0))))) \\ &= \sigma(\sigma(\sigma(\sigma(7)))) \\ &= 11 \end{aligned}$$

Fonction produit :

$$\begin{aligned} n \times 0 &= 0 \\ n \times (m+1) &= n + (n \times m) \end{aligned}$$

Fonction puissance :

$$\begin{aligned} n^0 &= 1 \\ n^{m+1} &= n \times n^m \end{aligned}$$

Double puissance :

$$\begin{aligned} n \uparrow\uparrow 0 &= 1 \\ n \uparrow\uparrow m+1 &= n^{n \uparrow\uparrow m} \end{aligned}$$

$$n \uparrow\uparrow m = n^{n^{\cdot^{\cdot^{\cdot^{n}}}}\}m}$$

Triple puissance :

$$\begin{array}{lcl} n \uparrow\uparrow\uparrow 0 & = & 1 \\ n \uparrow\uparrow\uparrow m+1 & = & n \uparrow\uparrow (n \uparrow\uparrow\uparrow m) \end{array}$$

k-puissance :

$$\begin{array}{lcl} n \uparrow^k 0 & = & 1 \\ n \uparrow^k m+1 & = & n \uparrow^{k-1} (n \uparrow^k m). \end{array}$$

Si k est un argument :

$$f(k+1, n, m+1) = f(k, n, f(k+1, n, m)).$$

Fonction d'Ackermann :

$$\begin{array}{lcl} Ack(0, m) & = & m+1 \\ Ack(k+1, 0) & = & Ack(k, 1) \\ Ack(k+1, m+1) & = & Ack(k, Ack(k+1, m)) \end{array}$$

Fonction factorielle :

$$\begin{array}{lcl} 0! & = & 1 \\ (n+1)! & = & (n+1).n! \end{array}$$

Fonction prédécesseur :

$$\begin{array}{lcl} pred(0) & = & 0 \\ pred(m+1) & = & m \end{array}$$

Fonction différence :

$$\begin{array}{lcl} n \dot{-} 0 & = & n \\ n \dot{-} (m+1) & = & pred(n \dot{-} m) \end{array}$$

Fonction signe :

$$\begin{array}{lcl} sg(0) & = & 0 \\ sg(m+1) & = & 1 \end{array}$$

Produit borné :

$$f(\overline{n}, m) = \prod_{i=0}^{m} g(\overline{n}, i)$$

$$\begin{array}{lcl} f(\overline{n}, 0) & = & g(\overline{n}, 0) \\ f(\overline{n}, m+1) & = & f(\overline{n}, m) \times g(\overline{n}, m+1) \end{array}$$

6.3 Prédicats primitifs récursifs

Un prédicat P à k arguments est un sous-ensemble de N^k (les éléments de N^k pour lesquels P est vrai).

La *fonction caractéristique* d'un prédicat $P \subseteq N^k$ est la fonction $f : N^k \to \{0, 1\}$ telle que

$$f(\overline{n}) = \begin{cases} 0 & \text{si } \overline{n} \notin P \\ 1 & \text{si } \overline{n} \in P \end{cases}$$

Un prédicat est *primitif récursif* si sa fonction caractéristique est primitive récursive.

Exemples

Prédicat zéro :

$$\begin{aligned} zerop(0) &= 1 \\ zerop(n+1) &= 0 \end{aligned}$$

Prédicat $<$:

$$petit(n,m) = sg(m \dot{-} n)$$

Prédicats booléens :

$$\begin{aligned} et(g_1(\overline{n}), g_2(\overline{n})) &= g_1(\overline{n}) \times g_2(\overline{n}) \\ ou(g_1(\overline{n}), g_2(\overline{n})) &= sg(g_1(\overline{n}) + g_2(\overline{n})) \\ non(g_1(\overline{n})) &= 1 \dot{-} g_1(\overline{n}) \end{aligned}$$

Prédicat $=$:

$$egal(n,m) = 1 \dot{-} (sg(m \dot{-} n) + sg(n \dot{-} m))$$

Quantification bornée :

$$\forall i \leq m \ p(\overline{n}, i)$$

est vrai si $p(\overline{n}, i)$ est vrai pour tout $i \leq m$.

$$\exists i \leq m \ p(\overline{n}, i)$$

est vrai si $p(\overline{n}, i)$ est vrai pour au moins un $i \leq m$.

$\forall i \leq m p(\overline{n}, i)$:

$$\prod_{i=0}^{m} p(\overline{n}, i)$$

$\exists i \leq m p(\overline{n}, i)$:

$$1 \dot{-} \prod_{i=0}^{m} (1 \dot{-} p(\overline{n}, i)).$$

Définition par cas :

$$f(\overline{n}) = \begin{cases} g_1(\overline{n}) & \text{si } p_1(\overline{n}) \\ \vdots & \\ g_\ell(\overline{n}) & \text{si } p_\ell(\overline{n}) \end{cases}$$

$$f(\overline{n}) = g_1(\overline{n}) \times p_1(\overline{n}) + \ldots + g_\ell(\overline{n}) \times p_\ell(\overline{n}).$$

Minimisation bornée :

$$\mu i \leq m \ q(\overline{n}, i) =$$
$$\begin{cases} \text{le plus petit } i \leq m \text{ tel que } q(\overline{n}, i) = 1, \\ 0 \text{ s'il n'existe pas de tel } i \end{cases}$$

$$\begin{aligned} \mu i \leq 0 \ q(\overline{n}, i) &= 0 \\ \mu i \leq m+1 \ q(\overline{n}, i) &= \end{aligned}$$

$$\begin{cases} 0 & \text{si } \neg\exists i \leq m+1 \ q(\overline{n}, i) \\ \mu i \leq m \ q(\overline{n}, i) & \text{si } \exists i \leq m \ q(\overline{n}, i) \\ m+1 & \text{si } q(\overline{n}, m+1) \\ & \text{et } \neg\exists i \leq m \ q(\overline{n}, i) \end{cases}$$

6.4 Au-delà des fonctions primitives récursives

Théorème

Il existe des fonctions calculables qui ne sont pas primitives récursives.

A	0	1	2	$\ldots$	j	$\ldots$
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$\ldots$	$f_0(j)$	$\ldots$
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$\ldots$	$f_1(j)$	$\ldots$
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$\ldots$	$f_2(j)$	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
f_i	$f_i(0)$	$f_i(1)$	$f_i(2)$	$\ldots$	$f_i(j)$	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\ddots$

$$g(n) = f_n(n) + 1 = A[n, n] + 1.$$

n'est pas primitive récursive, mais calculable.

6.4 Les fonctions μ-récursives

Minimisation non bornée :

$$\mu i \ q(\overline{n}, i) = \begin{cases} \text{le plus petit } i \text{ tel que } q(\overline{n}, i) = 1 \\ 0 \text{ si un tel } i \text{ n'existe pas} \end{cases}$$

Un prédicat $q(\overline{n}, i)$ est dit *sûr* (*safe*) si

$$\forall \overline{n} \ \exists i \ q(\overline{n}, i) = 1.$$

Les fonctions et prédicats μ-récursifs sont ceux obtenus à partir des fonctions primitives récursives de base par :

- composition, récursion primitive et
- minimisation non bornée de prédicats sûrs.

Fonctions μ-récursives et fonctions calculables

Nombres et chaînes de caractères :

Lemme
Il existe une représentation effective des nombres naturels par des chaînes de caractères.

Lemme
Il existe une représentation effective des chaînes de caractères par les naturels.

Alphabet Σ de taille k. Représentation de chaque symbole de Σ par un entier entre 0 et $k-1$. Représentation de $w = w_0 \dots w_l$:

$$gd(w) = \sum_{i=0}^{l} k^{l-i} gd(w_i)$$

Exemple : $\Sigma = \{abcdefghij\}$.

$$\begin{array}{rcl} gd(a) & = & 0 \\ gd(b) & = & 1 \\ & \vdots & \\ gd(i) & = & 8 \\ gd(j) & = & 9 \end{array}$$

$$gd(aabaafgj) = 00100569.$$

Encodage non univoque :

$$gd(aaabaafgj) = 000100569 =$$
$$00100569 = gd(aabaafgj)$$

Utiliser un alphabet de taille $k+1$ en ne rien encoder par 0.

$$gd(w) = \sum_{i=0}^{l} (k+1)^{l-i} gd(w_i).$$

Des fonctions μ-récursives aux machines de Turing

Théorème

Toute fonction μ-récursive est calculable par une machine de Turing.

1. les fonctions primitives récursives de bases sont calculables par machine de Turing ;

2. la composition, récursion primitive et minimisation non bornée sûre de fonctions calculables par machines de Turing sont calculables par machine de Turing.

Des machines de Turing aux fonctions μ-récursives

Théorème
Toute fonction calculable par machine de Turing est μ-récursive.

Soit une machine de Turing M. On montre qu'il existe f μ-récursive telle que

$$f_M(w) = gd^{-1}(f(gd(w))).$$

Prédicats utiles :

1. $init(x)$ configuration initiale de M.

2. $config\_suivante(x)$

3.

$$config(x,n)\begin{cases} config(x,0) = x \\ config(x,n+1) = \\ config\_suivante(config(x,n)) \end{cases}$$

4. $stop(x) = \begin{cases} 1 & \text{si } x \text{ final} \\ 0 & \text{sinon} \end{cases}$

5. $sortie(x)$

On a alors :

$$f(x) = sortie(config(init(x), nb\_de\_pas(x)))$$

où

$$nb\_de\_pas(x) = \mu i \ \ stop(config(init(x), i)).$$

Fonctions partielles

Une fonction partielle $f : \Sigma^* \to \Sigma^*$ est calculée par une machine de Turing M si,

- pour tout mot d'entrée w pour lequel f est défini, M s'arrête dans une configuration où $f(w)$ se trouve sur le ruban,

- pour tout mot d'entrée w pour lequel f n'est pas défini, M ne s'arrête pas ou s'arrête en signalant, par une valeur conventionnelle écrite sur le ruban, que la fonction n'est pas définie.

Une fonction partielle $f : N \to N$ est μ-récursive si elle peut être définie à partir des fonctions primitives récursives de base par

- composition,
- récursion primitive,
- minimisation non bornée.

La minimisation non bornée peut être appliquée à un prédicat non sûr. La fonction $\mu i \; p(\overline{n}, i)$ n'est pas définie lorsqu'il n'existe pas de i tel que $p(\overline{n}, i) = 1$.

Théorème
Une fonction partielle est μ-récursive si et seulement si elle est calculable par machine de Turing.

Chapitre 7
La non-calculabilité

7.1 Introduction

- Indécidabilité de problèmes concrets.
- Premier problème indécidable obtenu par diagonalisation.
- Autres problèmes obtenus par la technique de la réduction.
- Propriétés des langages acceptés par les machines de Turing.

7.2 Démontrer l'indécidabilité
Les classes de décidabilité

Correspondance entre problème et langage de l'encodage des instances positives.

Definition
La classe de décidabilité R est l'ensemble des langages décidables par une machine de Turing.

La classe R est la classe des langages (problèmes)

- décidés par machine de Turing,
- récursifs , décidables, calculables,
- solubles algorithmiquement.

Definition
La classe de décidabilité RE est l'ensemble des langages acceptés par une machine de Turing.

La classe RE est la classe des langages (problèmes)

- acceptés par machine de Turing,
- partiellement récursifs , partiellement décidables, partiellement calculables,
- partiellement solubles algorithmiquement,
- récursivement énumérables.

Lemme
La classe R est contenue dans la classe RE ($R \subseteq RE$)

Un premier langage indécidable

A	w_0	w_1	w_2	$\ldots$	w_j	$\ldots$
M_0	O	N	N	$\ldots$	O	$\ldots$
M_1	N	N	O	$\ldots$	O	$\ldots$
M_2	O	O	N	$\ldots$	N	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
M_i	N	N	O	$\ldots$	N	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\ddots$

- $A[M_i, w_j] =$ O (oui) si la machine de Turing M_i accepte la mot w_j ;

- $A[M_i, w_j] =$ N (non) si la machine de Turing M_i n'accepte pas le mot w_j (boucle ou rejette le mot).

$$L_0 = \{w | w = w_i \wedge A[M_i, w_i] = \mathsf{N}\}.$$

n'est pas dans la classe RE.

Un deuxième langage indécidable

Lemme
Le complément d'un langage de la classe R est un langage de la classe R.

Lemme
Si un langage L et son complément $\overline{L}$ sont tous deux dans RE, alors à la fois L et $\overline{L}$ sont dans R.

Trois cas possibles :

1. L et $\overline{L} \in$ R,

2. $L \notin$ RE et $\overline{L} \notin$ RE,

3. $L \notin$ RE et $\overline{L} \in$ RE $\cap \overline{\text{R}}$.

Lemme

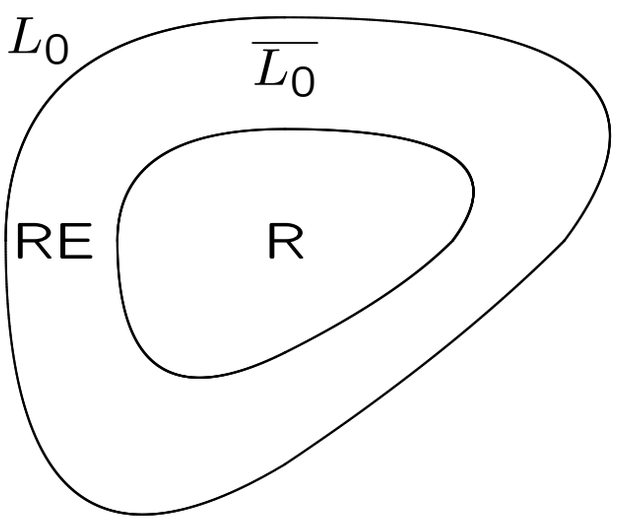
Le langage

$$\overline{L_0} = \{w | w = w_i \wedge M_i \text{ accepte } w_i\}$$

est dans la classe RE.

Théorème

Le langage $\overline{L_0}$ est indécidable (n'appartient pas à R) mais appartient à RE.



La technique de la réduction

1. On démontre que s'il existe un algorithme qui décide le langage L_2, alors il existe aussi un algorithme qui décide le langage L_1. Cela se fait en donnant un algorithme (formellement une machine de Turing s'arrêtant toujours) qui décide le langage L_1 en se servant comme d'un sous-programme d'un algorithme décidant L_2. Ce type d'algorithme est appelé une *réduction* de L_1 à L_2. En effet, il réduit la décidabilité de L_1 à celle de L_2.

2. On conclut que L_2 n'est pas décidable ($L_2 \notin \mathrm{R}$) car la réduction de L_1 vers L_2 montre que si L_2 était décidable L_1 le serait aussi, ce qui contredit l'hypothèse que L_1 est un langage indécidable.

Le *langage universel* LU

$$\text{LU} = \{< M, w > | \; M \text{ accepte } w\}$$

est indécidable.

Réduction à partir du langage $\overline{L_0}$:

1. Déterminer l'indice i tel que $w = w_i$.

2. Déterminer la machine de Turing M_i.

3. Appliquer la procédure de décision pour LU au mot $< M_i, w_i >$, si le résultat est positif, w est accepté, sinon il est rejeté.

Note : $\overline{\text{LU}} \notin \text{RE}$

Des problèmes indécidables

Le problème de l'arrêt

$$\mathsf{H} = \{< M, w > |\ M \text{ s'arrête sur } w\}$$

est indécidable. Réduction à partir de LU.

1. Appliquer l'algorithme décidant H à $< M, w >$.
2. Si l'algorithme décidant H donne la réponse "non" (*i.e.* la machine M ne s'arrête pas), répondre "non" (dans ce cas, on a effectivement $< M, w > \notin$ LU).
3. Si l'algorithme décidant H donne la réponse "oui", simuler l'exécution de M sur w et donner la réponse obtenue (dans ce cas, l'exécution de M sur w est finie et l'on obtient donc toujours une réponse).

Le problème de déterminer si un programme écrit dans un langage de programmation usuel (par exemple le PASCAL) s'arrête pour des valeurs fixées de ses données est indécidable. Réduction à partir du problème de l'arrêt pour les machines de Turing.

1. Construire un programme PASCAL P qui, étant donné une machine de Turing M et un mot w, simule le comportement de M sur w.

2. Décider si le programme P s'arrête pour les données $< M, w >$ et transmettre la réponse obtenue.

Le problème de déterminer si une machine de Turing s'arrête lorsque son mot d'entrée est le mot vide (*problème de l'arrêt sur mot vide*) est indécidable. La réduction se fait à partir du problème de l'arrêt.

1. Pour une instance $< M, w >$ du problème de l'arrêt, on construit une machine de Turing M' qui a le comportement suivant :

   - elle écrit le mot w sur son ruban d'entrée ;
   - elle se comporte ensuite comme M.

2. On résout le problème de l'arrêt sur mot vide pour M' et on transmet la réponse obtenue.

Le problème de déterminer si une machine de Turing s'arrête pour au moins un mot d'entrée (*problème de l'arrêt existentiel*) est indécidable. La réduction se fait à partir du problème de l'arrêt sur mot vide.

1. Pour une instance M du problème de l'arrêt sur mot vide, on construit une machine de Turing M' qui a le comportement suivant :

   - elle efface le contenu de son ruban d'entrée ;
   - elle se comporte ensuite comme M.

2. On résout le problème de l'arrêt existentiel pour M' et on transmet la réponse obtenue.

Le problème de déterminer si une machine de Turing s'arrête pour tout mot d'entrée (*problème de l'arrêt universel* est indécidable. La réduction se fait à partir du problème de l'arrêt sur ruban vide et est identique à celle de l'arrêt existentiel. La seule différence est que l'on résout le problème de l'arrêt universel pour M' plutôt que celui de l'arrêt existentiel.

Le problème de déterminer si le langage accepté par une machine de Turing est vide (*langage accepté vide*) est indécidable. Réduction à partir de $\overline{\text{LU}}$.

1. Pour une instance $< M, w >$ de $\overline{\text{LU}}$, on construit une machine de Turing M' qui :

   - simule l'exécution de M sur w sans tenir compte de son propre mot d'entrée ;
   - si M accepte w, elle accepte son mot d'entrée quel qu'il soit ;
   - si M n'accepte pas w (le rejette ou a une exécution infinie), elle n'accepte aucun mot.

2. On résout le problème du langage accepté vide pour M' et on transmet la réponse obtenue.

Cette réduction est correcte vu que

- $L(M') = \emptyset$ exactement quand M n'accepte pas w à savoir quand $< M, w > \in \overline{\mathsf{LU}}$;

- $L(M') = \Sigma^* \neq \emptyset$ exactement quand M accepte w à savoir quand $< M, w > \notin \overline{\mathsf{LU}}$.

Déterminer si le langage accepté par une machine de Turing est récursif (*langage accepté récursif*) est indécidable. Réduction à partir de $\overline{\text{LU}}$.

1. Pour une instance $< M, w >$ de $\overline{\text{LU}}$, on construit une machine de Turing M' qui
   - simule l'exécution de M sur w sans tenir compte de son propre mot d'entrée x ;
   - si M accepte w, elle se comporte sur son mot d'entrée x comme une machine de Turing universelle ;
   - si M n'accepte pas w, elle n'accepte aucun mot.
2. On résout le problème du langage accepté récursif pour M' et on transmet la réponse obtenue.

Cette réduction est correcte vu que

- $L(M') = \emptyset$ et est récursif exactement quand M n'accepte pas w à savoir quand $< M, w > \in \overline{\mathsf{LU}}$;

- $L(M') = \mathsf{LU}$ et n'est pas récursif exactement quand M accepte w à savoir quand $< M, w > \notin \overline{\mathsf{LU}}$.

Déterminer si le langage accepté par une machine de Turing est indécidable (*langage accepté indécidable*) est indécidable. Réduction à partir de $\overline{\mathsf{LU}}$.

1. Pour une instance $< M, w >$ de $\overline{\mathsf{LU}}$, on construit une machine de Turing M' qui

   - simule l'exécution de M sur w sans tenir compte de son propre mot d'entrée x ;
   - simultanément (c'est-à-dire en entrelaçant les exécutions), la machine de Turing M' simule la machine de Turing universelle sur son propre mot d'entrée x ;
   - dès qu'une des deux exécutions accepte (à savoir si M accepte w ou si le mot d'entrée de $x \in \mathsf{LU}$), M' accepte ;

2. si aucune des deux exécutions n'acceptent (à savoir si M n'accepte pas w ou si le mot d'entrée de $x \notin$ LU), M' n'accepte pas.

3. On résout le problème du langage accepté indécidable pour M' et on transmet la réponse obtenue.

Cette réduction est correcte vu que

- $L(M') = \mathsf{LU}$ et est indécidable exactement quand M n'accepte pas w à savoir quand $< M, w > \in \overline{\mathsf{LU}}$;

- $L(M') = \Sigma^*$ et est décidable exactement quand M accepte w à savoir quand $< M, w > \notin \overline{\mathsf{LU}}$.

Dans les réductions précédentes, le langage accepté par la machine M' est soit LU, soit $\emptyset$ ou Σ^*. Ces démonstrations peuvent donc aussi être utilisées pour établir que le problème de déterminer si le langage accepté par une machine de Turing est régulier (ou non régulier) est indécidable. En effet, $\emptyset$ et Σ^* sont réguliers alors que LU ne l'est pas.

7.4 Les Propriétés des langages récursivement énumérables

Les langages récursivement énumérables sont :

- les langages calculés par une machine de Turing,
- les langages générés par une grammaire,
- les langages énumérés par une procédure effective (ce qui explique l'appellation "récursivement énumérable").

Les langages calculés par une machine de Turing

Définition

Soit une machine de Turing M. Si M s'arrête sur un mot d'entrée u, dénotons par $f_M(u)$ le mot calculé par M pour u. Le langage calculé par M est alors l'ensemble de mots

$$\{w \mid \exists u \text{ tel que } M \text{ s'arrête pour } u \text{ et } w = f_M(u)\}.$$

Théorème

Un langage est calculé par une machine de Turing si et seulement si il est récursivement énumérable (accepté par une machine de Turing).

Soit un langage L accepté par une machine de Turing M. La machine de Turing M' suivante calcule ce langage.

1. La machine M' mémorise d'abord son mot d'entrée (on peut supposer qu'elle dispose d'un deuxième ruban).

2. Ensuite, elle se comporte exactement comme M.

3. Si M accepte, M' recopie le mot d'entrée mémorisé sur son ruban.

4. Si M n'accepte pas, M' poursuit indéfiniment son exécution.

Soit un langage L calculé par une machine de Turing M. La machine de Turing non déterministe M' suivante accepte ce langage.

1. La machine M' mémorise d'abord son mot d'entrée w.

2. Ensuite, elle génère de façon non déterministe un mot u.

3. La machine M' simule alors le comportement de M sur u.

4. Si M s'arrête sur u, alors M' compare w à $f_M(u)$ et accepte w si $w = f_M(u)$.

5. Si M ne s'arrête pas sur u, M' n'accepte pas w.

Les langages générés par une grammaire

Théorème
Un langage est généré par une grammaire si et seulement si il est récursivement énumérable.

Soit $G = (V, \Sigma, R, S)$, la machine de Turing M ci-dessous accepte le langage généré par G.

1. La machine M commence par mémoriser son mot d'entrée, supposons qu'elle dispose d'un deuxième ruban à cet effet.
2. Ensuite, elle efface son ruban et y place le symbole de départ S de la grammaire.

3. Le cycle suivant est alors répété :

   (a) de façon non déterministe, la machine choisit une règle de R et une chaîne contenue sur le ruban ;

   (b) si la chaîne sélectionnée sur le ruban est identique au membre de gauche de la règle, elle est remplacée par le membre de droite de la règle ;

   (c) le contenu du ruban est comparé au mot d'entrée mémorisé, en cas d'égalité, la machine accepte ; sinon elle poursuit son exécution.

Soit une machine de Turing $M = (Q, \Gamma, \Sigma, \delta, s, B, F)$. On construit une grammaire

$$G_0 = (V_{G_0}, \Sigma_{G_0}, R_{G_0}, S_{G_0})$$

telle que $S_{G_0} \stackrel{*}{\Rightarrow} w$ avec $w \in (Q \cup \Gamma)^*$ si et seulement si w décrit une configuration (q, α_1, α_2) écrite $\alpha_1 q \alpha_2$ de M.

La grammaire G_0 est définie par

- $V_{G_0} = Q \cup \Gamma \cup \{S_{G_0}, A_1, A_2\}$,
- $\Sigma_{G_0} = \Sigma$,
- R_{G_0} est l'ensemble de règles décrit ci-dessous.

1. Configuration initiale de M :

$$\begin{array}{lcll} S_{G_0} & \rightarrow & sA_1 & \\ A_1 & \rightarrow & aA_1 & \forall a \in \Sigma \\ A_1 & \rightarrow & A_2 & \\ A_2 & \rightarrow & BA_2 & \\ A_2 & \rightarrow & \varepsilon. & \end{array}$$

2. Transitions. Pour tout $p, q \in Q$ et $X, Y \in \Gamma$ tels que

$$\delta(q, X) = (p, Y, R)$$

nous incluons la règle

$$qX \rightarrow Yp.$$

Similairement, pour tout $p, q \in Q$ et $X, Y, Z \in \Gamma$ tels que

$$\delta(q, X) = (p, Y, L)$$

nous incluons la règle

$$ZqX \rightarrow pZY.$$

Problème: mot d'entrée perdu. Solution: simuler une machine à deux rubans.

$G_1 = (V_{G_1}, \Sigma_{G_1}, R_{G_1}, S_{G_1})$ où

- $V_{G_1} = \Sigma \cup Q \cup ((\Sigma \cup \{e\}) \times \Gamma) \cup \{S_{G_1}, A_1, A_2\}$ (nous représenterons un élément de $((\Sigma \cup \{e\}) \times \Gamma)$ par une paire $[a, X]$),

- $\Sigma_{G_1} = \Sigma$,

- R_{G_1} est l'ensemble de règles décrit ci-dessous.

1. Configuration initiale de M :

$$\begin{array}{lcll} S_{G_1} & \rightarrow & sA_1 & \\ A_1 & \rightarrow & [a,a]A_1 & \forall a \in \Sigma \\ A_1 & \rightarrow & A_2 & \\ A_2 & \rightarrow & [e,B]A_2 & \\ A_2 & \rightarrow & \varepsilon. & \end{array}$$

2. Transitions. Pour tout $p, q \in Q$, $X, Y \in \Gamma$ et $a \in \Sigma \cup \{e\}$ tels que

$$\delta(q, X) = (p, Y, R)$$

nous incluons la règle

$$q[a, X] \rightarrow [a, Y]p.$$

Similairement, pour tout $p, q \in Q$, $X, Y, Z \in \Gamma$ et $a, b \in \Sigma \cup \{e\}$ tels que

$$\delta(q, X) = (p, Y, L)$$

nous incluons la règle

$$[b, Z]q[a, X] \rightarrow p[b, Z][a, Y].$$

3. Pour tout $q \in F$, $X \in \Gamma$ et $a \in \Sigma \cup \{e\}$, nous incluons les règles

$$\begin{aligned} q[a, X] &\rightarrow qaq \\ [a, X]q &\rightarrow qaq \end{aligned}$$

si $a \neq e$ et

$$\begin{aligned} q[a, X] &\rightarrow q \\ [a, X]q &\rightarrow q \end{aligned}$$

si $a = e$. Ces règles permettent de placer une copie de q devant chaque non-terminal $[a, X]$ et d'en extraire la première composante. Finalement, nous ajoutons

$$q \rightarrow \varepsilon$$

qui permet d'effacer les copies de l'état q.

Les langages énumérés par une procédure effective

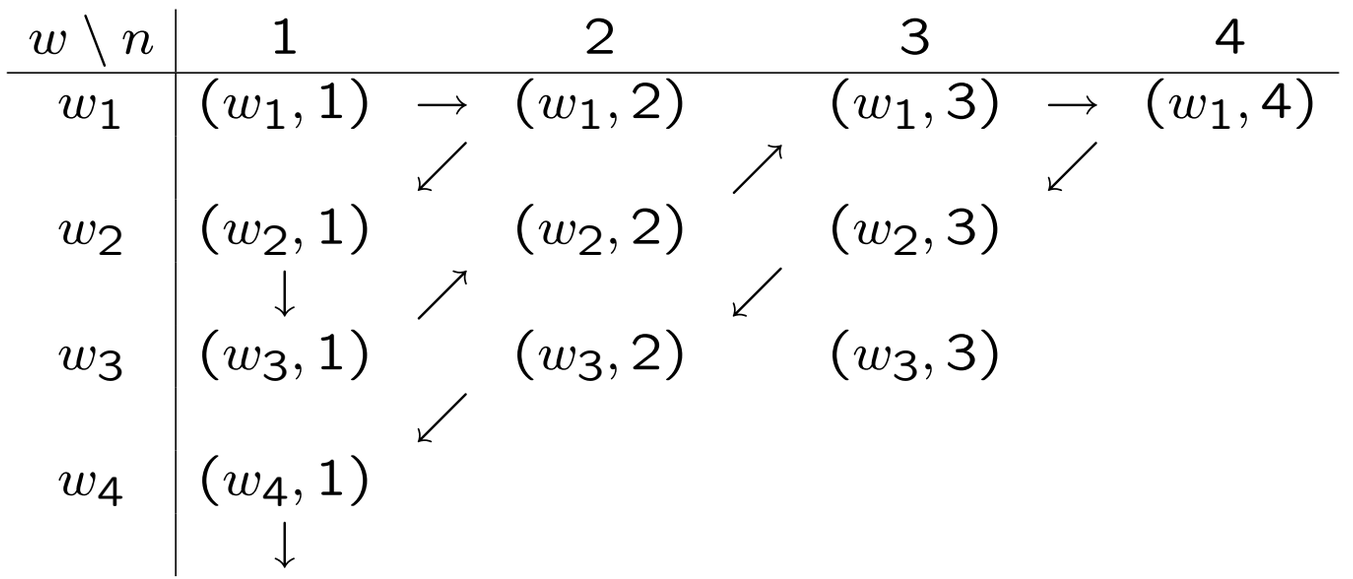
Machine de Turing qui énumère les mots acceptés par M.

- génèrer tous les mots par ordre lexicographique et de longueur croissante,

- simuler M sur chaque nouveau mot généré et conserver celui-ci uniquement s'il est accepté par M.

Incorrect: la machine de Turing peut avoir des exécutions infinies.

Solution: autre ordre d'énumération.



- On considère les paires (w, n) dans l'ordre de leur énumération.
- Pour chacune de ces paires, on simule l'exécution de M sur w mais on limite l'exécution à n étapes. On produit le mot w si cette exécution accepte w.
- On passe ensuite à la paire (w, n) suivante.

7.5 D'autres problèmes indécidables

Le problème de déterminer si un mot w appartient au langage généré par une grammaire G est indécidable.

Réduction à partir du problème LU. Soit une instance $< M, w >$ du problème LU, elle peut être résolue comme suit :

1. on construit la grammaire G générant le langage accepté par M

2. on détermine si $w \in L(G)$ et on transmet la réponse obtenue.

Le problème de déterminer si deux grammaires G_1 et G_2 génèrent le même langage est indécidable.

Réduction à partir du problème de l'appartenance au langage d'une grammaire. Une instance $< w, G >$ de ce problème est résolue comme suit.

1. Soit $G = (V, \Sigma, R, S)$. On construit les grammaires $G_1 = G$ et $G_2 = (V, \Sigma, R', S')$, avec
$$R' = R \cup \{S' \to S, S' \to w\}.$$
2. On vérifie si $L(G_1) = L(G_2)$ et on transmet le résultat obtenu.

On a bien $L(G_2) = L(G_1) \cup \{w\}$ et donc $L(G_2) = L(G_1)$ si et seulement si $w \in L(G)$.

Le problème de la *validité dans la calcul des prédicats* est indécidable.

Le problème de *l'universalité d'un langage hors-contexte*, à savoir le problème de déterminer si pour une grammaire hors-contexte G on a $L(G) = \Sigma^*$ est indécidable.

Le problème de *l'intersection vide de langages hors-contexte* est indécidable. Il s'agit de déterminer pour deux grammaires hors-contexte G_1 et G_2 si $L(G_1) \cap L(G_2) = \emptyset$.

Le *dixième problème de Hilbert* est indécidable. Ce problème consiste à déterminer si l'équation

$$p(x_1, \ldots, x_n) = 0$$

où $p(x_1, \ldots, x_n)$ est un polynôme à coefficients entiers a une solution dans le domaine des entiers.

Fonctions non calculables

Une fonction totale

$$f : \Sigma_1^* \to \Sigma_2^*$$

est calculable si et seulement si les questions suivantes sont décidables.

1. Etant donné $n \in N$ et $w \in \Sigma_1^*$, est-ce que $|f(w)| > n$?

2. Etant donné $k \in N$, $w \in \Sigma_1^*$ et $a \in \Sigma_2$, est-ce que $f(w)_k = a$? (la k^{e} lettre de $f(w)$ est-elle a?).

La situation est semblable dans le cas d'une fonction partielle. Une fonction

$$f : \Sigma_1^* \to \Sigma_2^*$$

est une fonction partielle calculable si et seulement si les conditions suivantes sont satisfaites.

1. Il est partiellement décidable de déterminer si pour un w donné $f(w)$ est défini.

2. Pour $n \in N$ et $w \in \Sigma_1^*$ tels que $f(w)$ est défini, il est décidable de déterminer si $|f(w)| > n$.

3. Pour $k \in N$, $a \in \Sigma_2$ et $w \in \Sigma_1^*$ tels que $f(w)$ est défini, il est décidable de déterminer si $f(w)_k = a$.

Chapitre 8
La complexité

8.1 Introduction

- Problème soluble – problème soluble efficacement.
- Mesure de la complexité : fonction de complexité.
- Complexité polynomiale.
- Problèmes NP-complets.

8.2 Mesurer la complexité

- Abstraction par rapport à la machine utilisée.
- Abstraction par rapport aux données (taille uniquement).
- Notation O.
- Critère d'efficacité : polynomial.

8.3 Les problèmes polynomiaux

- Influence de l'encodage.
- Exemple du graphe.
- Encodage raisonnable :
  - pas de bourrage,
  - décodage polynomial,
  - pas de numérotation unaire.

Complexité et machines de Turing

Complexité en temps d'une machine de Turing s'arrêtant toujours :

$$T_M(n) = \max \{m \mid \exists x \in \Sigma^*, |x| = n \text{ et l'exécution de } M \text{ sur } x \text{ comporte } m \text{ étapes}\}.$$

Une machine de Turing est polynomiale s'il existe un polynôme $p(n)$ tel que

$$T_M(n) \leq p(n)$$

pour tout $n \geq 0$.

La classe P est la classe des langages décidés par une machine de Turing polynomiale.

8.4 Les transformations polynomiales

- Diagonalisation inefficace pour montrer que des problèmes ne sont pas dans P.

- Autre approche : comparaison de problèmes.

Le voyageur de commerce (TS)

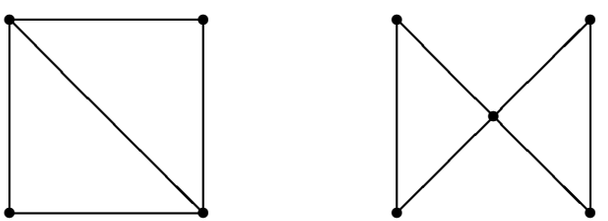
- Ensemble V de n villes.
- Distances $d(v_i, v_j)$.
- Constante b.
- Permutation des villes telle que :

$$\sum_{1 \leq i < n} d(v_{p_i}, v_{p_{i+1}}) + d(v_{p_n}, v_{p_1}) \leq b.$$

Le circuit hamiltonien (HC)

- Graphe $G = (V, E)$

- Existe-t-il un parcours fermé du graphe contenant chaque sommet une et une seule fois.



Définition des transformations polynomiales

But : établir un lien entre des problèmes tels que HC et TS (l'un appartient à P si et seulement si l'autre appartient à P.

Définition :
Soient un langage $L_1 \in \Sigma_1^*$ et un langage $L_2 \in \Sigma_2^*$. Une *transformation polynomiale* de L_1 vers L_2 (notation $L_1 \propto L_2$) est une fonction $f : \Sigma_1^* \to \Sigma_2^*$ qui satisfait les conditions suivantes :

1. elle est calculable en temps polynomial,

2. $f(x) \in L_2$ si et seulement si $x \in L_1$.

HC $\propto$ TS

- L'ensemble des villes est identique à l'ensemble des sommets du graphe, c'est-à-dire $C = V$.

- Les distances sont données par $d(v_i, v_j) = \begin{cases} 1 & \text{si } (v_i, v_j) \in E \\ 2 & \text{si } (v_i, v_j) \notin E \end{cases}$.

- La constante b est égale au nombre de villes, c'est-à-dire $b = |V|$.

Propriétés de $\propto$

Si $L_1 \propto L_2$, alors

- si $L_2 \in \mathsf{P}$ alors $L_1 \in \mathsf{P}$,
- si $L_1 \notin \mathsf{P}$ alors $L_2 \notin \mathsf{P}$.

Si $L_1 \propto L_2$ et $L_2 \propto L_3$, alors

- $L_1 \propto L_3$.

Problèmes polynomialement équivalents

Définition
Deux langages L_1 et L_2 sont *polynomialement équivalents* notation $L_1 \equiv_P L_2$) si et seulement si $L_1 \propto L_2$ et $L_2 \propto L_1$.

- Classes d'équivalence polynomiale : soit tous les membres dans P, soit aucun.
- Possibilité de construire une classe d'équivalence de proche en proche.
- Définition plus abstraite de la classe de HC et TS.

La classe NP

- Caractériser les problèmes pour lesquels il est nécessaire d'examiner un grand nombre de cas, mais tels que la vérification de chaque cas est rapide.

- Donc, solution rapide si l'énumération des cas ne coûte rien.

- Modélisation : non-déterminisme.

Complexité des machines non déterministes

Le temps de calcul d'une machine de Turing sur un mot w est donné par

- la longueur de *la plus courte* exécution acceptant le mot si celui-ci est accepté,
- la valeur 1 si le mot n'est pas accepté.

La complexité en temps de M (non déterministe) est la fonction $T_M(n)$ définie par

$$T_M(n) = \max \{m \mid \exists x \in \Sigma^*, |x| = n \text{ et le temps de calcul de } M \text{ sur } x \text{ est } m\}.$$

Définition de NP

Définition

La classe NP (de *N*on déterministe *P*olynomial) est la classe des langages acceptés par une machine de Turing non déterministe polynomiale.

Exemple

HC et TS sont dans NP.

Théorème

Soit $L \in$ NP. Il existe une machine de Turing déterministe M et un polynôme $p(n)$ tel que M décide L et est de complexité en temps bornée par $2^{p(n)}$.

Machine M_{nd} de complexité polynomiale $q(n)$ qui accepte L. Simuler toutes les exécutions de M_{nd} de longueur inférieure à $q(n)$. Pour un mot w, la machine doit :

1. Déterminer la longueur n de w et calcule $q(n)$. polynôme.

2. Simuler chaque exécution de M_{nd} de longueur $q(n)$ (temps nécessaire $q'(n)$). Si r est le nombre maximum de choix possibles de M_{nd} à chaque étape de son exécution, il y a au plus $r^{q(n)}$ exécutions de longueur $q(n)$.

3. Si une des exécutions simulées accepte, M accepte. Sinon, M s'arrête et rejette le mot w.

Complexité : bornée par $r^{q(n)} \times q'(n)$ et donc par $2^{\log_2(r)(q(n)+q'(n))}$ qui est de la forme $2^{p(n)}$.

La structure de NP

Définition Une classe d'équivalence polynomiale C_1 est inférieure à une classe d'équivalence polynomiale C_2 (notation $C_1 \preceq C_2$) s'il existe une transformation polynomiale de tout langage de C_1 vers tout langage de C_2.

Plus petite classe dans NP : P

- La classe NP contient la classe P (P $\subseteq$ NP).
- La classe P est une classe d'équivalence polynomiale.
- Pour tout $L_1 \in$ P et pour tout $L_2 \in$ NP on a $L_1 \propto L_2$.

Plus grande classe dans NP : NPC

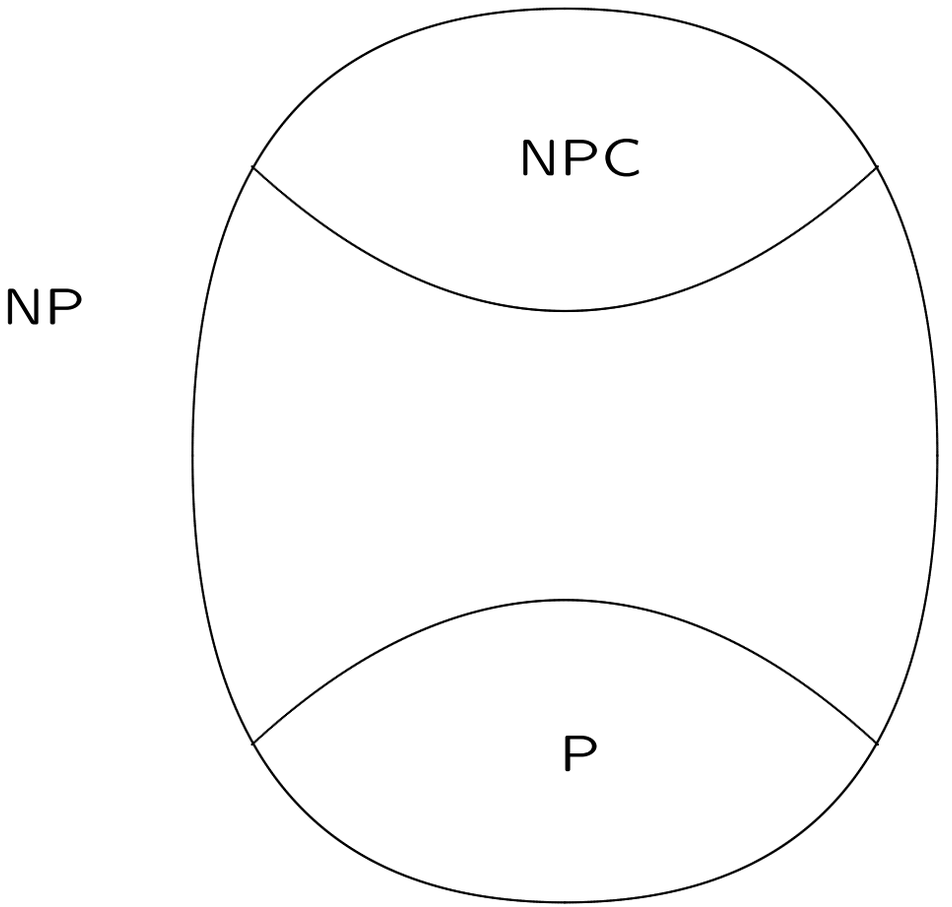
Un langage L est NP-complet si

1. $L \in$ NP,

2. pour tout langage $L' \in$ NP, $L' \propto L$.

Théorème
S'il existe un langage NP-complet L décidé par un algorithme polynomial, alors tous les langages de NP sont décidables en temps polynomial, c'est-à-dire P = NP.

Conclusion : Un problème NP-complet n'a pas de solution polynomiale si et seulement si P $\neq$ NP


NP
NPC
P

Démontrer la NP-complétude

Pour démontrer qu'un langage L est NP-complet, il faut établir

1. qu'il fait effectivement partie de la classe NP ($L \in \text{NP}$),

2. que pour tout langage $L' \in \text{NP}$, $L' \propto L$.

ou encore,

3. il existe $L' \in \text{NPC}$ tel que $L' \propto L$.

Notion de problème NP-dur.

Un premier problème NP-complet
Le calcul des propositions

Calcul booléen :

p	$\neg p$
0	1
1	0

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

p	q	$p \supset q$
0	0	1
0	1	1
1	0	0
1	1	1

- Expression booléenne : $(1 \wedge (0 \vee (\neg 1))) \supset 0$.

- Variables propositionnelles et calcul des propositions :
  $(p \wedge (q \vee (\neg r))) \supset s$.

- Fonction d'interprétation. Formule valide, formule satisfaisable.

- Forme normale conjonctive : conjonction de disjonction de littéraux.

Le théorème de Cook

Problème SAT : satisfaisabilité des formules du calcul des propositions en forme normale conjonctive.

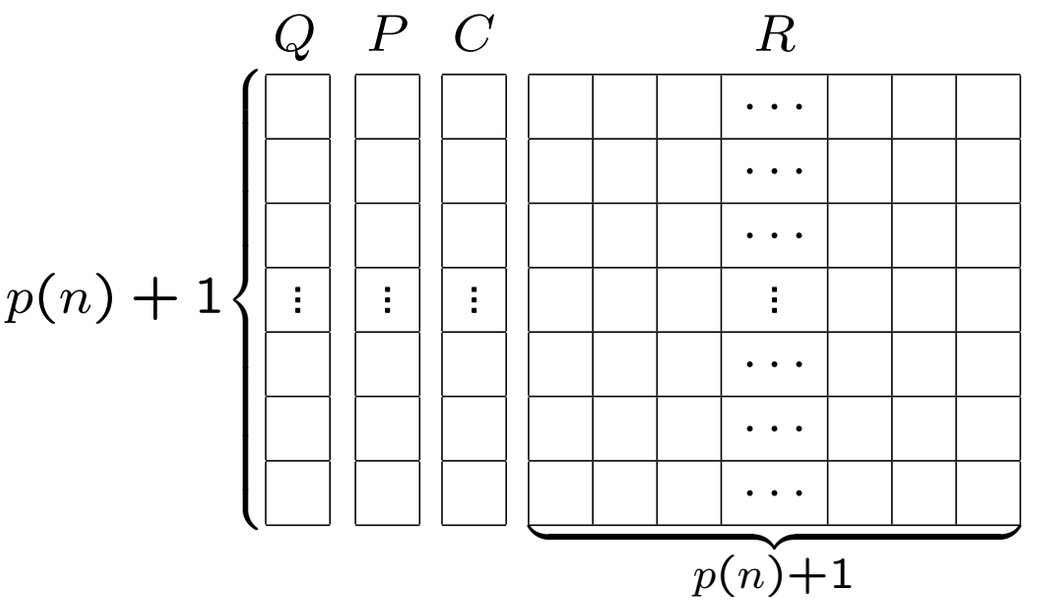
Théorème
Les problème SAT est NP-complet

Démonstration

1. SAT appartient à NP.
2. Il existe une transformation polynomiale de tout langage de NP vers L_{SAT}.
   - Transformation à 2 arguments : mot et langage.
   - Langages de NP caractérisés par une machine de Turing non déterministe polynomiale.

Mot w ($|w| = n$) et machine de Turing non déterministe polynomiale $M = (Q, \Gamma, \Sigma, \Delta, s, B, F)$ (borne $p(n)$).

Description d'une exécution de M (R : ruban; Q : état; P : position; C : choix.)



Représentation d'une exécution par des variables propositionnelles :

1. Une proposition $r_{ij\alpha}$ pour $0 \leq i, j \leq p(n)$ et $\alpha \in \Gamma$.

2. Une proposition $q_{i\kappa}$ pour $0 \leq i \leq p(n)$ et $\kappa \in Q$.

3. Une proposition p_{ij} pour $0 \leq i, j \leq p(n)$.

4. Une proposition c_{ik} pour $0 \leq i \leq p(n)$ et $1 \leq k \leq r$.

Formule satisfaite uniquement par une exécution de M qui accepte le mot w : conjonction des formules suivantes.

$$\bigwedge_{0 \leq i,j \leq p(n)} \left[\left(\bigvee_{\alpha \in \Gamma} r_{ij\alpha} \right) \wedge \bigwedge_{\alpha \neq \alpha' \in \Gamma} (\neg r_{ij\alpha} \vee \neg r_{ij\alpha'}) \right]$$

Une proposition par case. Longueur $O(p(n)^2)$.

$$\bigwedge_{0 \leq i \leq p(n)} \left[\left(\bigvee_{0 \leq j \leq p(n)} p_{ij} \right) \wedge \bigwedge_{0 \leq j \neq j' \leq p(n)} (\neg p_{ij} \vee \neg p_{ij'}) \right]$$

Une proposition par case. Longueur $O(p(n)^3)$.

$$\left[\bigwedge_{0\leq j\leq n-1} r_{0jw_{j+1}} \wedge \bigwedge_{n\leq j\leq p(n)} r_{0jB}\right] \wedge q_{0s} \wedge p_{00}$$

Etat initial. Longueur $O(p(n))$

$$\bigwedge_{\substack{0\leq i<p(n)\\ 0\leq j\leq p(n)\\ \alpha\in\Gamma}} [(r_{ij\alpha} \wedge \neg p_{ij}) \supset r_{(i+1)j\alpha}]$$

$$\bigwedge_{\substack{0\leq i<p(n)\\ 0\leq j\leq p(n)\\ \alpha\in\Gamma}} [\neg r_{ij\alpha} \vee pij \vee r_{(i+1)j\alpha}]$$

Transitions, ruban non modifié. Longueur $O(p(n)^2)$.

$$\bigwedge_{\substack{0\leq i<p(n)\\ 0\leq j\leq p(n)\\ \alpha\in\Gamma\\ 1\leq k\leq r}} \left[\begin{array}{l} ((q_{i\kappa}\wedge p_{ij}\wedge r_{ij\alpha}\wedge c_{ik})\supset q_{(i+1)\kappa'})\wedge \\ ((q_{i\kappa}\wedge p_{ij}\wedge r_{ij\alpha}\wedge c_{ik})\supset r_{(i+1)j\alpha'})\wedge \\ ((q_{i\kappa}\wedge p_{ij}\wedge r_{ij\alpha}\wedge c_{ik})\supset p_{(i+1)(j+d)}) \end{array}\right]$$

$$\bigwedge_{\substack{0\leq i<p(n)\\ 0\leq j\leq p(n)\\ \alpha\in\Gamma\\ 1\leq k\leq r}} \left[\begin{array}{l} (\neg q_{i\kappa}\vee\neg p_{ij}\vee\neg r_{ij\alpha}\vee\neg c_{ik}\vee q_{(i+1)\kappa'})\wedge \\ (\neg q_{i\kappa}\vee\neg p_{ij}\vee\neg r_{ij\alpha}\vee\neg c_{ik}\vee r_{(i+1)j\alpha'})\wedge \\ (\neg q_{i\kappa}\vee\neg p_{ij}\vee\neg r_{ij\alpha}\vee\neg c_{ik}\vee p_{(i+1)(j+d)}) \end{array}\right]$$

Transitions, partie modifiée. Longueur $O(p(n)^2)$.

$$\bigvee_{\substack{0 \leq i \leq p(n) \\ \kappa \in F}} [q_{i\kappa}]$$

Etat final atteint. Longueur $O(p(n))$.

- Longueur totale de la formule $O(p(n)^3)$.
- Formule constructible en temps polynomial.
- Donc, transformation polynomiale en fonction de $n = |w|$.
- Formule satisfaisable si et seulement si la machine M accepte.

D'autres problèmes NP-complets

3-SAT : satisfaisabilité en forme normale conjonctive avec 3 littéraux par clause.

SAT $\propto$ 3-SAT.

1. Une clause $(x_1 \vee x_2)$ comportant deux littéraux est remplacée par

$$(x_1 \vee x_2 \vee y) \wedge (x_1 \vee x_2 \vee \neg y)$$

2. Une clause (x_1) comportant un seul littéral est remplacée par

$$\begin{array}{ccccc} (x_1 \vee y_1 \vee y_2) & \wedge & (x_1 \vee y_1 \vee \neg y_2) & \wedge \\ (x_1 \vee \neg y_1 \vee y_2) & \wedge & (x_1 \vee \neg y_1 \vee \neg y_2) & \end{array}$$

3. Une clause

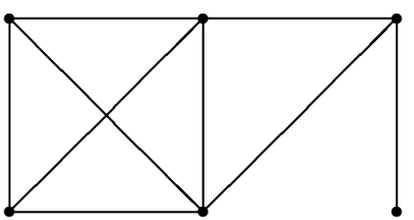
$$(x_1 \vee x_2 \vee \cdots \vee x_i \vee \cdots \vee x_{\ell-1} \vee x_\ell)$$

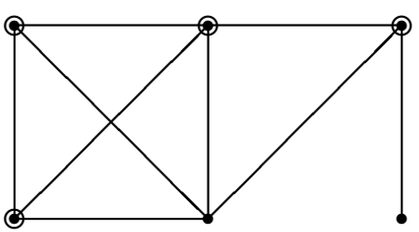
comportant $\ell \geq 4$ littéraux est remplacée par

$$\begin{aligned}(x_1 \vee x_2 \vee y_1) \;& \wedge (\neg y_1 \vee x_3 \vee y_2) \\ & \wedge (\neg y_2 \vee x_4 \vee y_3) \wedge \cdots \\ & \wedge (\neg y_{i-2} \vee x_i \vee y_{i-1}) \wedge \cdots \\ & \wedge (\neg y_{\ell-4} \vee x_{\ell-2} \vee y_{\ell-3}) \\ & \wedge (\neg y_{\ell-3} \vee x_{\ell-1} \vee x_\ell)\end{aligned}$$

Le problème de la *couverture de sommets* (VC) est NP-complet.

Etant donnés un graphe $G = (V, E)$ et un entier $j \leq |V|$, il faut déterminer s'il existe un sous-ensemble $V' \subseteq V$ tel que $|V'| \leq j$ et tel que pour tout arc $(u, v) \in E$, soit u, soit $v \in V'$.





3-SAT $\propto$ VC

Instance de 3-SAT :

$$E_1 \wedge \cdots \wedge E_i \wedge \cdots \wedge E_k$$

Chaque E_i est de la forme

$$x_{i1} \vee x_{i2} \vee x_{i3}$$

où x_{ij} est un littéral. L'ensemble des variables propositionnelles est

$$\mathcal{P} = \{p_1, \ldots, p_\ell\}.$$

L'instance de VC qui est construite est alors la suivante.

1. L'ensemble V des sommets du graphe contient

   (a) une paire de sommets étiquetés p_i et $\neg p_i$ pour chaque variable propositionnelle de $\mathcal{P}$,

   (b) un triplet de sommets étiquetés x_{i1}, x_{i2}, x_{i3} pour chaque clause E_i.

   Le nombre de sommets du graphe est donc égal à $2\ell + 3k$.

2. L'ensemble E des arcs du graphe contient

   (a) l'arc $(p_i, \neg p_i)$ pour chaque paire de sommets $p_i, \neg p_i$, $1 \leq i \leq \ell$,

   (b) les arcs (x_{i1}, x_{i2}), (x_{i2}, x_{i3}) et (x_{i3}, x_{i1}) pour chaque triplet de sommets x_{i1}, x_{i2}, x_{i3}, $1 \leq i \leq k$,

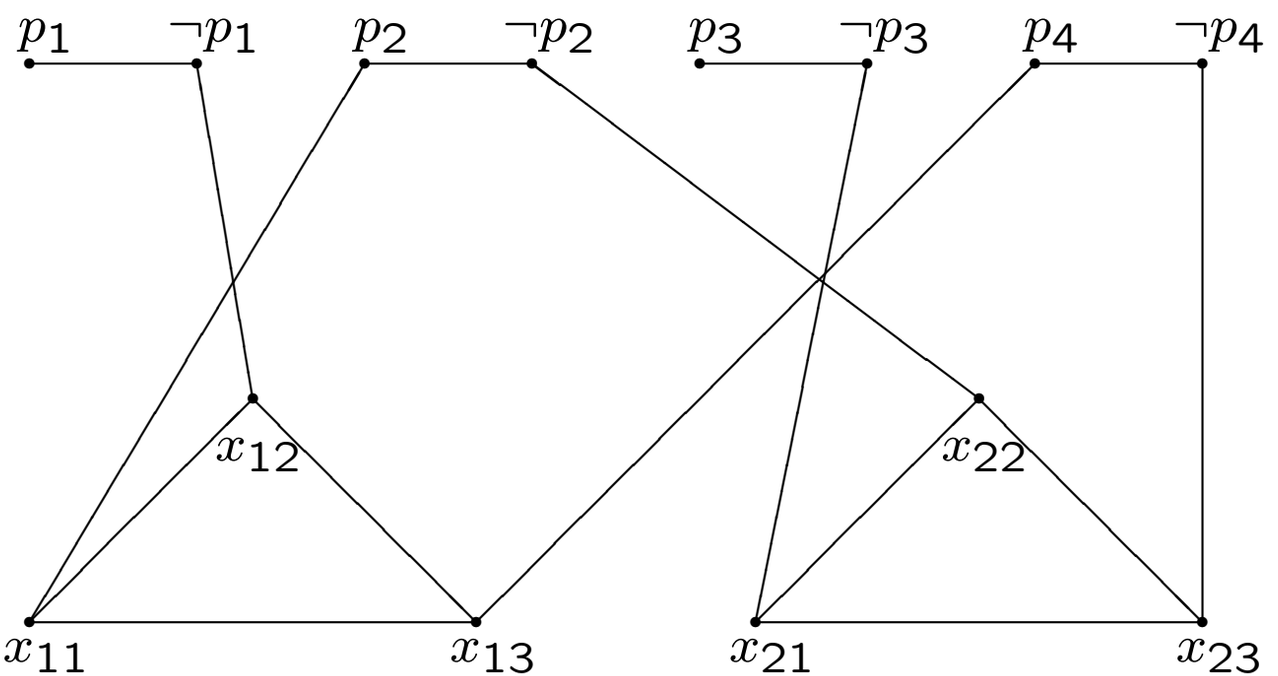
   (c) un arc entre chaque sommet x_{ij} et le sommet p ou $\neg p$ représentant le littéral correspondant.

   Le nombre d'arcs du graphe est donc de $\ell + 6k$.

3. La constante j est $\ell + 2k$.

Exemple

$$(p_2 \vee \neg p_1 \vee p_4) \wedge (\neg p_3 \vee \neg p_2 \vee \neg p_4)$$



Autres exemples

Les problèmes du circuit hamiltonien (HC) et du voyageur de commerce (TS) sont NP-complets.

Le problème du *nombre chromatique* d'un graphe est NP-complet. Ce problème consiste à déterminer pour un graphe G et une constante k s'il est possible de colorier les sommets du graphe en utilisant k couleurs distinctes de telle façon que deux sommets adjacents (reliés par un arc) aient des couleurs distinctes.

Le problème de la *programmation entière* est NP-complet. Les données de ce problème sont

1. un ensemble de m paires $(\overline{v_i}, d_i)$ où chaque $\overline{v_i}$ est un vecteur d'entiers de taille n et chaque d_i un entier,
2. un vecteur $\overline{d}$ de taille n,
3. une constante b.

La question à résoudre est de déterminer s'il existe un vecteur d'entiers $\overline{x}$ de taille n tel que $\overline{x} \cdot \overline{v_i} \leq d_i$ pour $1 \leq i \leq m$ et tel que $\overline{x} \cdot \overline{d} \geq b$.

Pour les rationnels, le problème est soluble en temps polynomial (programmation linéaire)

Le problème de l'équivalence d'automates finis non déterministes est NP-dur. Il n'y a même pas d'algorithme non déterministe polynomial pour résoudre ce problème. En fait, il est complet dans la classe PSPACE.

8.8 Interpréter la NP-complétude

- Cas le plus mauvais. Algorithmes efficaces “en moyenne” possibles.
- Méthodes heuristiques pour limiter l'examen d'un nombre exponentiel de cas.
- Solution approchée des problèmes d'optimisation.
- Les instances usuelles du problème peuvent satisfaire des contraintes qui les rendent polynomiales.

8.9 Autres classes de complexité

La classe co-NP est la classe des langages L dont le complément ($\Sigma^* - L$) est dans NP.

La classe EXPTIME est la classe des langages décidés par une machine de Turing déterministe dont la complexité en temps est bornée par une fonction exponentielle ($2^{p(n)}$ où $p(n)$ est un polynôme).

La classe PSPACE est la classe des langages décidés par une machine de Turing déterministe dont la complexité en espace (le nombre de cases du ruban utilisées) est bornée par un polynôme.

La classe NPSPACE est la classe des langages acceptés par une machine de Turing non déterministe dont la complexité en espace (le nombre de cases du ruban utilisées) est bornée par un polynôme.

$$\mathsf{P} \subseteq \begin{matrix}\mathsf{NP}\\ \mathsf{co\text{-}NP}\end{matrix} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME}.$$