Langages formels, calculabilité et complexité

Eugène Assarin (Paris 7). Sur son site web : références, plan détaillé.

TD : Thomas Nowak

Notation : examen et projet personnel (exposé sur un article de recherche).

Support de cours : livre d'O.Cartan ; livre de P.Wolper *introduction à la calculabilité* ; Hopcroft Ullman (automates, langages) ; Papadimitriou (complexité) ; Garey Johnson (complexité) ; Perrin *mots infinis*.

Poly sur la calculabilité disponnible sur page web.

Plan du cours.

1. Automates finis et langages réguilers
2. Grammaires hors contexte (HC)
3. Calculabilité
4. Complexité

2013-09-26

I. Automates et langages

Déf. On appelle *alphabet* un ensemble fini, usuellement noté Σ, dont les éléments sont appellée *lettres*.

Ex. $\Sigma_1 = \{a, b\}$ ou $\Sigma_2 = \{a, (, +\}$

Déf. On appelle *mot* sur un alphabet Σ une séquence finie d'éléments de Σ.

Ex. Sur Σ_1, le mot $b\,a\,a\,b$, le mot vide noté ε (contenant 0 lettres), sur Σ_2 le mot $a + +(((a\,($.

Déf. On appelle *concaténation* de deux mots u et v le mot formé par les lettre de u suivies par les lettres de v.

Ex. $(b\,a\,a\,b) \cdot (b\,b) = b\,a\,a\,b\,b\,b$.

Combinatoire des mots : exemple : on a deux mots X et Y qui satisfont $XY = YX$, que dire sur leur structure ?

Notation. L'ensemble de tous les mots sur Σ est noté Σ^*.

Déf. Un *langage* sur Σ est un ensemble quelconque de mots (un sous-ensemble de Σ^*).

Parenthèse : induction structurelle

(voire Arnold Gessarian *maths pour l'informatique*)

L'induction structurelle permet de faire trois choses :

- comment définir un ensemble

- comment définir une fonction sur ceet ensemble
- comment démonterer des théorèmes

Déf. *Définition par induction structurelle.* On a un ensemble (appellé *univers*) noté U. Une définition par induction structurelle contient deux choses :

- Bases : $B \subset U$ l'ensemble des éléments de base, les axiomes
- Les (quelque) règles inductives : ensemble fini de fonctions de la forme $r_i : U^{k_i} \to U$, $i \in \{1, ..., n\}, k_i \in \mathbb{N}^*$

Cette définition décrit un sous-ensemble $C \subset U$ qui est le plus petit ensemble tel que :

- $C \supset B$
- C est clos par $r_1, ..., r_n$

Exemple. $U = \mathbb{N}$, $B = \{0\}$, $r : n \mapsto n + 4$, on définit ainsi l'ensemble $C = \{0, 4, 8, 12, ...\} = 4\,\mathbb{N}$.

Figure 1. Exemple d'arbre de dérivation

Exemple 2. $U = \mathbb{N}$, $B = \{0\}$, $r : n \mapsto n + 1$, on définit ainsi l'ensemble $C = \mathbb{N}$.

Exemple 3. On travaille sur les mots de $\{a, b\}^* = U$, $B = \{\varepsilon, a, b\}$, et on utilise les règles :

- $r_1 : w \mapsto a\,w\,a$
- $r_2 : w \mapsto b\,w\,b$

On définit ainsi l'ensemble C des palindromes sur $\{a, b\}$.

Exemple 4. $U = \Sigma^*$ avec $\Sigma = \{X, Y, Z, +, \times, (,)\}$, $B = \{X, Y, Z\}$, règles :

- $e \mapsto (e)$
- $e, f \mapsto e + f$
- $e, f \mapsto e \times f$

On peut donner : $X \in C$, $Y \in C$, $X + Y \in C$, $(X + Y) \in C$, $X \times (X + Y) \in C$. On définit donc l'ensemble des expressions algébriques (un peu limitées).

Remarque. C est l'ensemble des objets pouvant être obtenus à partir d'éléments de B en appliquant un nombre fini de fois les règles r_1, ..., r_n. Autrement dit, $x \in C$ ssi il possède un arbre de dérivation, c'est-à-dire un arbre dont il est la racine, dont les feuilles appartiennent à B et tels que chaque noeud interne s'obtienne par application d'une des règles sur l'ensemble de ses fils (cf schéma).

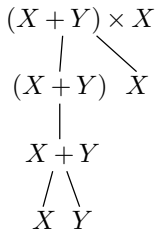


Figure 2. Exemple d'arbre de dérivation

Déf. Étant donné C défini par induction structurelle avec $B, r_1, ..., r_k$. Pour définir par *récurrence structurelle* une fonction $f : C \to X$ (X est un ensemble quelconque), il suffit :

- de donner $f(b)$ pour tout $b \in B$
- de dire comment passer de $f(b_1), ..., f(b_{k_i})$ à $f(r_i(b_1, ..., b_{k_i}))$ pour chaque règle r_i.

(peut se faire rigoureusement, il y a une propriété d'unicité derrière)

Exemple. (basé sur l'exemple 2 plus haut). On définit f par :

- $f(0) = 1$
- $f(n+1) = f(n) \cdot 2$

On a ainsi défini la fonction $f(n) = 2^n$.

Exemple. (basé sur l'exemple 3 plus haut). On définit f par :

- $f(\varepsilon) = 0$
- $f(a) = f(b) = 1$
- $f(a\,w\,a) = f(w) + 2$
- $f(b\,w\,b) = f(w) + 2$

La fonction f donne la longueur des mots du langage C.

Exemple. (basé sur l'exemple 4 plus haut). On définit $f : \Sigma^* \to \mathbb{N}$ par :

- $V(X) = 7$, $V(Y) = 3$, $V(Z) = 5$
- $V((E)) = V(E)$
- $V(E + F) = V(E) + V(F)$
- $V(E \times F) = V(E) \cdot V(F)$

Ce qui donne : $V(((X + Y))) = V((X + Y)) = V(X + Y) = V(X) + V(Y) = 7 + 3 = 10$. V correspond donc à la *valeur* d'une expression algébrique pour $X = 7, Y = 3, Z = 5$.

Mais attention il y a un problème : certaines expressions possèdent plusieurs arbres de dérivation qui donnent des valeurs différentes ! Par exemple, $V(X + Y \times Z) = V(X) + V(Y \times Z) = 22 = V(X + Y) \cdot V(Z) = 50$, ce qui est assez problématique...

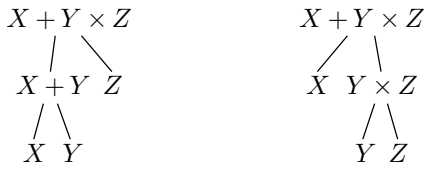


Figure 3. Exemple de dérivation ambigüe

Déf. Une définition par induction structurelle est *non-ambigüe* si tout élément de C possède exactement un seul arbre de dérivation.

Une fonction ne peut être définie par induction structurelle que si celle-ci est non-ambigüe.

On peut rendre l'exemple 4 non ambigü en utilisant les deux règles suivantes à la place :

- $r_+ : e, f \mapsto (e + f)$
- $r_\times : e, f \mapsto (e \times f)$

Les formules $X + Y$ et $X \times Y$ ne sont donc plus valables, seules $(X + Y)$ et $(X \times Y)$ le sont. On peut prouver la non-ambigüité de cette définition par comptage de parenthèses.

Théorème. *Preuve de théorème par induction structurelle* (généralisation de la preuve par récurrence). Soit P une propriété définie sur U et C défini par induction structurelle. Si :

- $\forall b \in B, P(b)$
- $\forall i, \forall c_1, ..., c_{k_i} \in C,\ P(c_1) \wedge \cdots \wedge P(c_{k_i}) \Rightarrow P(r_i(c_1, ..., c_{k_i}))$

Alors $\forall e \in C, P(e)$.

Cela se démontrerait par récurrence sur la taille de l'arbre de dérivation.

Exemple 1. Toute preuve par récurrence (sur $\mathbb{N}$).

Exemple 2. Théorème : dans une expression algébrique, on a : #occurences de variables = #opérations+1.

Avant la preuve, formalisons un peu : on peut utiliser des fonctions définie par induction structurelle à condition que la grammaire soit non-ambigue.

- $\#v(X) = \#v(Y) = \#v(Z) = 1$
- $\#v((e + f)) = \#v(e) + \#v(f)$
- $\#v((e \times f)) = \#v(e) + \#v(f)$
- $\#o$ est défini de la même manière.

Dans tous les cas, on peut faire la preuve directement :

Sur la base $\{X, Y, Z\}$, c'est ok.

Soit e, f deux formules vérifiant la propriété $\#v = \#o + 1$. On a bien :

- $\#v((e)) = \#v(e) = \#o(e) + 1 = \#o((e)) + 1$
- $\#v(e + f) = \#v(e) + \#v(f) = \#o(e) + 1 + \#o(f) + 1 = \#o(e + f) + 1$
- $\#v(e \times f) = \#v(e) + \#v(f) = \#o(e) + 1 + \#o(f) + 1 = \#o(e \times f) + 1$

La propriété est donc démontrée.

Automates finis et langages réguliers (rappels)

Déf. Un automate fini est un tuple $(Q, \Sigma, \Delta, I, F)$, où :

- Q est un ensemble fini, appellé ensemble des états
- Σ est un alphabet
- $\Delta \subset Q \times \Sigma \times Q$ est une relation de transition
- $I \subset Q$ et $F \subset Q$ les états initiaux et finals

Ex. (voir schéma) $A = (Q, \Sigma, \Delta, I, F)$ avec $Q = \{p, q\}$, $\Sigma = \{a, b\}$, $\Delta = \{(p, a, q), (q, a, p), (q, b, q)\}$, que l'on peut aussi noter $\Delta = \left\{ p \xrightarrow{a} q, q \xrightarrow{a} p, q \xrightarrow{b} q \right\}$, $I = \{p\}$, $F = \{p\}$.

Ce système peut représenter la dynamique d'un système informatique. Q correspond à l'ensemble des états du système (ce qui est dans la mémoire de la machine, l'état des portes, etc.) ; Σ correspond à l'ensemble des évènements, ie des entrées, possibles ; Δ correspond à comment les évènements changent l'état de la machine.

Ce système peut aussi être une représentation pour coder un langage.

Déf. On appelle *calcul* (ou run) de A sur $w = a_1\, a_2 \cdots a_n$ un chemin de la forme $q_0 \xrightarrow{a_1} q_1 \to \cdots \xrightarrow{a_n} q_n$ où $\forall i \in \{1, ..., n\}, \left(q_{i-1} \xrightarrow{a_i} q_i \right) \in \Delta$.

Déf. On dit qu'un mot w est *reconnu* (ou accepté ou généré) par A s'il existe un calcul (dit calcul *accepteur*) de w dont le premier état appartient à I et le dernier appartient à F.

Déf. On appelle *langage* de A et on note $\mathcal{L}(A)$ l'ensemble des mots reconnus par A.

Ex. Sur l'automate A de l'exemple précédent, $p \xrightarrow{a} q \xrightarrow{b} q \xrightarrow{b} q \xrightarrow{a} p$ est un calcul accepteur pour le mot $a\, b\, b\, a$, donc $a\, b\, b\, a \in \mathcal{L}(A)$. Il n'y a pas de calcul accepteur pour le mot $a\, b$ car le seul calcul pour ce mot est $p \xrightarrow{a} q \xrightarrow{b} q \notin F$, donc $a\, b \notin \mathcal{L}(A)$. L'ensemble des mots reconnus est l'ensemble des mots avec un nombre pair de a et avec un nombre quelconque de b tous les nombres impairs de a.

Typologie des automates

(voir schéma)

On a (2) un automate non déterministe qui reconnait tous les mots contenant $a\, a$. Par exemple, il y a plusieurs calculs partant de p pour le mot $a\, a$: $p \to q \to r \in F$ ou $p \to p \to q \notin F$.

Déf. Si $I = \{q_0\}$ et Δ est une fonction $Q \times \Sigma \to Q$, alors on dit que l'automate A est *déterministe complet*. Pour tout mot w, il existe un unique calcul partant de q_0. Si Δ est une fonction partielle, l'automate est *déterministe* mais pas complet.

déterministe $\Leftrightarrow \forall p, a, \exists ! \, q \,|\, \left(p \xrightarrow{a} q \right) \in \Delta$

Ex. L'automate du schéma (2) est non déterministe et non complet. L'automate du schéma (2) est déterministe et non complet.

Théorème. Si A est déterministe non complet avec n états, alors il existe B déterministe complet avec $n + 1$ états tel que $\mathcal{L}(A) = \mathcal{L}(B)$.

Il est très facile de compléter un automate déterministe non complet, il y a donc parfois confusion.

La façon standard de compléter est de rajouter un état « poubelle » qui n'est pas final, et sur lequel pointent toutes les flèches qui manquent.

Théorème. Si A est non-déterministe avec n états, alors il existe B déterministe avec au plus 2^n états tel que $\mathcal{L}(A) = \mathcal{L}(B)$.

Déf. On peut étendre des automates avec de ε-transitions, ils sont alors nécessairement non-déterministes.

Ex. Schéma (3), un calcul accepteur pour le mot $a\,a\,b\,a\,a$ est donc $p \xrightarrow{a} p \xrightarrow{a} p \xrightarrow{\varepsilon} q \xrightarrow{b} q \xrightarrow{\varepsilon} r \xrightarrow{a} r \xrightarrow{a} r$.

Théorème. Soit A un automate non-déterministe avec ε-transitions avec n états, alors il existe un automate B non-déterministe sans ε-transitions avec n états tel que $\mathcal{L}(A) = \mathcal{L}(B)$. (il faut en général rajouter beaucoup de transitions, ce qui rend l'automate moins compréhensible).

Déf. $L \subset \Sigma^*$ est dit *régulier* s'il existe un automate fini A tel que $\mathcal{L}(A) = L$. Cette définition ne dépend pas du type d'automate, elle est vrai pour des automates aussi bien déterministes que non-déterministes, avec ou sans ε-transitions, complets ou incomplets.

L'automate est une bonne structure de donnée pour repréesenter un langage.

Théorème. Propriétés des langages réguliers :

1. Ils sont clos par $^-$, $\cap$, $\cup$, $\cdot$, *, h, h^{-1} (morphisme, morphisme inverse), shuffle (symbole bizarre)

   Il existe des algorithmes pour construire les automates corrspondants.

2. Les questions suivantes sont décidables (il existe des algorithmes pour les résoudre) :

   $w \in L$? $L = \varnothing$? $L_1 = L_2$? $L_1 \subset L_2$? $L = \Sigma^*$?

   (pour les langages réguliers, représentation par des automates).

Explications pour le 1.

Soit $L = \mathcal{L}(A)$ et $M = \mathcal{L}(B)$ deux langages réguliers. Alors $L \cup M$ est régulier et on peut construire un automate C tel que $\mathcal{L}(C) = L \cup M$ (il suffit de faire l'union disjointe des automates A et B).

Pour le complémentaire, on déterminise et complète l'automate (si nécessaire) puis on transforme F en $\bar{F}$.

Pour l'intersection, c'est le complémentaire de l'union des complémentaires. Ou alors on peut construire un automate produit.

Pour la concaténation, on les met bout à bout en rajoutant des ε-transitions.

Pour l'étoile, on le fait boucler sur lui-même. (pour les puissances finies, cf concaténation). Remarque : $L^0 = \{\varepsilon\}, L^1 = L, L^k = \underbrace{L \cdot \cdots \cdot L}_{k \text{ fois}}, L^* = \bigcup_{k \geqslant 0} L^k$.

Un morphisme pour deux alphabets Σ, Γ est une application $\Sigma \to \Gamma$ qui est un morphisme pour la concaténation. $h(a_1 \cdots a_n) = h(a_1) \cdots h(a_n)$. On peut le prolonger sur des langages L : $h(L) = \{h(w), w \in L\}$. Exemple de morphisme $\Sigma = \{a, b, c\}, \Gamma = \{0, 1\}$, $h(a) = 1$, $h(b) = 0\,1$, $h(c) = 0\,0\,1$. Dans ce cas, on a par ex $h(a\,b\,c) = 0\,1\,1\,0\,0\,1$.

Pour construire l'automate qui reconnait le langage transformé par morphisme, on remplace les flèches pour chaque lettres par des flèches pour leurs images respectives par le morphisme.

Exemple de morphisme donné en schéma (4).

Morphisme inverse : $h^{-1}(M) = \{w \,|\, h(w) \in M\}$. Exercice : prouver la cloture par h^{-1}.

Shuffle : soient deux mots u et v. On les découpe en morceaux, puis on intercale des morceaux de u et de v (en les prenant dans l'ordre). Définition officielle : on prend une factorisation en sous-mots $u = u_1 \cdot \cdots \cdot u_k$ et $v = v_1 \cdot \cdots \cdot v_k$, et on construit $u_1 v_1 u_2 v_2 \cdots u_k v_k$. (le shuffle de L et M est l'ensemble des mots pouvant être obtenus de cette manière à partir de deux mots de L et M).

Exemple : si $L = \{a^{2n}\}$ et $M = \{b^{3n}\}$, alors L shuffle M est l'ensemble des mots ayant un nombre pair de a et un nombre multiple de 3 de b.

L'automate shuffle se construit avec un automate produit, en avançant à chaque lettre soit sur l'automate A soit sur l'automate B.

Explications pour le 2.

Soit $L_1 = \mathcal{L}(A)$ et $L_2 = \mathcal{L}(B)$.

Pour tester $w \in L$ c'est très simple avec un automate déterministe, un peu plus compliqué avec un non-déterministe.

Pour un sous-ensemble M d'états, on définit l'ensemble des sucesseurs par le mot w, par induction structurelle :

- $S(M, \varepsilon) = M$
- $S(M, w\,a) = \Delta(S(M, w), a) = \left\{ q \mid \exists p \in S(M, w), \left(p \xrightarrow{a} q\right) \in \Delta \right\}$

On a maintenant $w \in \mathcal{L}(A) \Leftrightarrow S(I, w) \cap F \neq \varnothing$, ce qui peut se programmer facilement.

Par exemple sur l'automate (2), comment montrer que $a\,a \in L$? On applique l'algorithme de calcul récursif : $S(I, \varepsilon) = I = \{p\}$, $S(I, a) = \{p, q\}$, $S(I, a\,a) = \{p, q, r\}$, $S(I, a\,a) \cap F = S(I, a\,a) \cap \{r\} = \{r\} \neq \varnothing$ donc le mot $a\,a$ est reconnu.

Pour tester $\mathcal{L}(A) \neq \varnothing$, il faut trouver un chemin partant d'un état de I et arrivant sur un état de F (voir cours d'algorithmique, algorithmique de graphes). C'est une question fondamentale car elle permet de faire ce qui suit.

Ensuite, $L_1 \subset L_2 \Leftrightarrow L_1 \cap \overline{L_2} = \varnothing$ que l'on sait construire puis tester. Il est ensuite facile de tester $L_1 = L_2$ (on teste les deux inclusions).

$L = \Sigma^* \Leftrightarrow \bar{L} = \varnothing$

2013-10-03

Rappel : expressions régulières sur Σ

Syntaxe : définie par induction.

1. Base : $\varepsilon, \varnothing$ et a pour chaque $a \in \Sigma$
2. Trois règles inductives :

   $f \mapsto f^*$

   $f, g \mapsto (f + g)$

   $f, g \mapsto (f \cdot g)$

Exemples. $(a + \varnothing)^{**} \cdot \varnothing$

Sémantique. Langage défini par une expression régulière :

1. $[\varepsilon] = \{\varnothing\}$
2. $[\varnothing] = \varnothing$
3. $[a] = \{a\}$
4. $[f^*] = [f]^*$

5. $[f+g] = [f] \cup [g]$

6. $[f \cdot g] = [f] \cdot [g]$

On peut avoir des expressions régulières un peu plus riches : par exemple, on peut avoir le complémentaire $\bar{f}$ ($[\bar{f}] = \overline{[f]}$).

Théorème de Kleene. L est un langage régulier (ie accepté par un automate) ssi il peut être définit par une expression régulière (ssi L défini par une expression régulière étendue).

Idée de preuve. La preuve est algorithmique.

1. Il est trivial de construire l'automate reconnaissant une expression régulière, par ce qui précède.

2. Construire une expression régulière à partir d'un automate est non trivial.

   Pour $q \in Q$, on note $L_q = \{\text{mots ayant un calcul } q \to \cdots \to f \in F\}$.

   On écrit maintenant un système d'équations sur les L_p. Par exemple sur l'automate (10), on a le système :

$$(1): \begin{cases} L_p = a\,L_p + a\,L_q \\ L_q = b\,L_p + \varepsilon \end{cases}$$

   Pour résoudre une équation du type : $X = L\,X + M$, alors si $L \not\ni \varepsilon$ il y a une unique solution, $X = L^*M$. Si $l \ni \varepsilon$, c'est la solution minimale. Ici :

$$\begin{aligned} (1) &\Leftrightarrow \begin{cases} L_p = a^*a\,L_q \\ L_q = b\,a^*\,a\,L_q + \varepsilon \end{cases} \\ &\Leftrightarrow \begin{cases} L_p = a\,a^*(ba^*a)^*\,\varepsilon \\ L_q = (b\,a^*a)^*\,\varepsilon \end{cases} \end{aligned}$$

   Le langage qui nous intéresse est $L_p = a\,a^*(ba^*a)^*\,\varepsilon$

Théorème de Myhill-Nérode et minimisation

Question. On a $L \subset \Sigma^*$. Est-il régulier ?

Déf. Quotient: soit L un langage et w un mot, $w\backslash L = \{u : w \cdot u \in L\}$.

Exemple. $L = (a\,b)^*$; $\varepsilon\backslash L = (a\,b)^*$; $a\backslash L = b\,(a\,b)^*$; $b\backslash L = \varnothing$; $a\,a\backslash L = \varnothing$; $a\,b\backslash L = (a\,b)^*$. On voit qu'ici on a trois quotients qui se répètent : $\varnothing, (a\,b)^*, b\,(a\,b)^*$

Exemple 2. $M = \{a^n\,b^n \ ; \ n \in \mathbb{N}\}$; $\varepsilon\backslash M = M$; $b\backslash M = \varnothing$; $a\backslash M = \{a^n b^{n+1} \ ; \ n \in \mathbb{N}\}$; $a^j\backslash M = \{a^k b^{k+j} ; k \in \mathbb{N}\}$. On voit qu'ici, on peut construire une infinité de quotiens différents.

Théorème de Myhill-Nérode. L est régulier ssi il existe un nombre fini de quotients différents de la forme $w\backslash L$. De plus, si L possède n quotients, alors tout automate déterministe complet pour L contient au moins n états, et il existe un automate déterministe avec n états reconaissant L.

Déf équivalente de Myhill-Nérode. Étant donné un langage L, $v \approx_L w \Leftrightarrow (\forall u,\ u\ v \in L \Leftrightarrow u\ w \in L) \Leftrightarrow v\backslash L = w\backslash L$. C'est une relation d'équivalence sur les mots, dont chaque classe correspond à un quotient (il y a une bijection).

Ex. $L = (a\ b)^*$; $b \approx a\ a$ $(w\backslash L = \varnothing)$; $\varepsilon \approx a\ b \approx a\ b\ a\ b$ $(w\backslash L = L)$; $a \approx a\ b\ a \approx a\ b\ a\ b\ a$ $(w\backslash L = b\,(a\,b)^*)$.

Reformulation du théorème. Le langage L est régulier ssi $\approx_L$ a un nombre fini de classes d'équivalence.

Prop. Soit A un automate déterministe, et L son langage. Si $i \xrightarrow{w} q$, alors $w\backslash L = L_q$.

Dém. $u \in w\backslash L \Leftrightarrow w\,u \in L \Leftrightarrow A$ accepte $w\,u \Leftrightarrow$ le calcule mène à $F \Leftrightarrow u \in L_q$

Lemme. Si A automate déterministe accepte L, alors son nombre d'états est supérieur ou égal à son nombre de quotients différents.

Dém. Soit $w_1\backslash L \neq w_2\backslash L \neq \cdots \neq w_n\backslash L$.

Les états q_i correspondant à $i \xrightarrow{w_i} q_i$ sont nécessairements distincts, donc il y a au minimum n états.

Corollaire. Si le nombre de quotients est infini, le langage est irrégulier.

Reste à faire : en ayant un langage à n classes d'équivalence (ie n quotients), comment construire un automate déterministe complet qui reconnaît L et avec n états ? Idée : les états de A seront les quotients, ie les classes d'équivalence.

Notation. $[w] = \{u : u \approx w\}$ = la classe d'équivalence de w.

Lemme. L'équivalence $\approx$ est une congruence :

1. Si $v \approx w$ alors $v \in L \Leftrightarrow w \in L$
2. Si $v \approx w$, alors $v\,a \approx w\,a, \forall a \in \Sigma$

Dém. Soit $v \approx w$, alors par définition $\forall u, v\,u \in L \Leftrightarrow w\,u \in L$.

1. Prenons $u = \varepsilon$, $v\,\varepsilon \in L \Leftrightarrow w\,\varepsilon \in L$, ie $v \in L \Leftrightarrow w \in L$;
2. Prenons $u = a\,y$. $\forall y$,$v\,a\,y \in L \Leftrightarrow w\,a\,y \in L$, ie $v\,a \approx w\,a$.

On peut donc construire l'automate de Myhill-Nérode :

- $Q = \{[w]\,; w \in \Sigma^*\}$, qui est fini par hypothèse
- $i = [\varepsilon]$ et $F = \{[w]\,; w \in L\}$
- $\delta([w], a) = [w\,a]$, qui ne dépend pas du choix du représentant dans la classe $[w]$ d'après le 2. du lemme précédent.

C'est bien un automate fini déterministe complet à n états, reste à prouver qu'il reconnait bien L. On voit en fait qu'on a pour tout $w \in \Sigma^*$, $[\varepsilon] \xrightarrow{w} [w]$ (lemme : $S([\varepsilon], w) = [w]$), donc w accepté par $A \Leftrightarrow S([\varepsilon], w) \in F \Leftrightarrow [w] \in F \Leftrightarrow w \in L$ puisque toutes les classes de F sont des classes d'éléments appartenant à L et elles y sont toutes. Donc A reconnait L. □

Application 1 : preuve de non régularité

Prop. $L = \{a^n b^n ; n \in \mathbb{N}\}$ n'est pas régulier.

Dém. Il a une infinité de quotients $a^i\backslash L = \{a^n b^{n+i}, n \in \mathbb{N}\}, i \in \mathbb{N}$, tous différents. □

Application 2 : bornes inférieures

Soit $L_n = \Sigma^*\,a\,\Sigma^a$.

Prop. L_n peut être reconnu par un automate non-déterministe à $n + 2$ états. Tout automate déterministe qui le reconnait a au moins 2^n états.

Corollaire. La déterminisation mène nécessairement àune explosion exponentielle.

Dém. Pour construire un automate non déterministe, voir (11).

Pour le déterminte, on compte les quotients. Soit $w \in \{a,b\}^n$ (ce qui fait 2^n possibilités pour w).

On s'intéresse à : $\begin{array}{l}\{a,b\}^n \to \mathcal{P}(\Sigma^*) \\ w \mapsto (w\backslash L) \cap b^*\end{array}$.

On voit que cette application est une injection :en effet, les possibilités d'obtenir un mot reconnu en rajoutant uniquement un certain nombre de b à la fin de w dépendent exactement de la configuration des a dans w : pour $k \in \{1,...,n\}$ $w \cdot b^k \in L \Leftrightarrow w_k = a$; faire un dessin pour comprendre.

$\begin{array}{l}\{a,b\}^n \to \mathcal{P}(\{1,...,n\}) \\ w \mapsto \{k \in \{1,...,n\} : w \cdot b^k \in L\}\end{array}$ est une bijection.

Cela signifie qu'il y a donc au minimum 2^n quotients de type $w\backslash L$.

Minimisation d'automate déterministe complet

Problème. On a un automate déterministe complet A qui reconnait L. Trouver le plus petit automate déterministe complet M qui reconaît le même langage.

Première observation. Le nombre d'états de M est le nombre de quotients du langage L.

« Algorithme ». On prend l'automate A.

1. On supprime les états inaccessibles depuis i, ce qui donne $A' = (Q, \Sigma, \delta, i, F)$;
2. On construit une équivalence sur Q l'ensemble des états de A', appellée congruence de Nérode : $p \approx q \Leftrightarrow L_p = L_q$;
3. On fusionne les états de Q qui sont équivalents par cette relation (on remplace les états par les classes d'équivalence sur les états) : $M = A'/\approx$.

Il faut justifier le passage de la deuxième à la troisième étape.

Lemme. La relation $\approx$ définie dans l'algorithme ci-dessus est une congruence (ie une relation d'équivalence qui se comporte bien vis-à-vis d'une certaine opération). C'est à dire, si $p \approx q$, alors :

1. $p \in F \Leftrightarrow q \in F$
2. $\delta(p,a) \approx \delta(q,a)$

Dém.

1. $p \in F \Leftrightarrow \varepsilon \in L_p = L_q \Leftrightarrow q \in F$
2. Si $p \xrightarrow{a} p'$ et $q \xrightarrow{a} q'$. $u \in L_{p'} \Leftrightarrow a\,u \in L_p = L_q \Leftrightarrow u \in L_{q'}$, donc $L_{p'} = L_{q'}$.

Détails de l'étape 3. On a $A' = (Q, \Sigma, \delta, i, F)$. Pour construire M :

1. $Q_M = \{[q] \,;\, q \in Q\}$
2. $i_M = [i]$
3. $F_M = \{[f] \,;\, f \in F\}$
4. $\delta_M([q], a) = [\delta(q,a)]$

Cela est correct grâce au fait que $\approx$ soit une congruence.

Propriétés.

1. $L(M) = L(A)$
2. $\#Q_M = \#\{w\backslash L; w \in \Sigma^*\} =$ nombres de classes de Myhill-Nérode
3. M est isomorphe à l'automate de Myhill-Nérode

Lemme. $S_M([i], w) = [S_A(i, w)]$

Dém. Faire un dessin :)

Si dans A : $i \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_m} q_m$, alors dans M : $[i] \xrightarrow{a_1} [q_1] \to \ldots \xrightarrow{a_m} [q_m]$

Dém.

1. $w \in L(A) \Leftrightarrow S_M([i], w) \in F_M \Leftrightarrow [S_A(i, w)] \in F_M \Leftrightarrow S_A(i, w) \in F \Leftrightarrow w \in L(A)$
2. Si $v \approx w$ et $i \xrightarrow{v} p$ et $i \xrightarrow{w} q$, alors $p \approx q$.

   Donc dans l'automate M, p et q sont un même état : v et w ammène sur le même état.

   De plus chaque état de M est atteignable depuis i.
3. En exo.

Thm de minimisation. Pour tout automat déterministe complet, il existe un automate $A' \subset A$ et une congruence $\approx$ sur A' tels que $M = A'/\approx$ a les propriétés $L(M) = L(A') = L(A)$ et Mest isomorphe à l'automate de Myhill-Nérode pour $L(A)$. M s'appelle *automate minimal* de A.

Exemple. Voir dessin (12). On a seulement deux langages quotients (plus le langage vide) :

$$\begin{cases} L_r = L_p = a^*b(b\,a^*\,b)^* \\ L_q = L_s = (b\,a^*\,b)^* \\ L_{\text{poubelle}} = \varnothing \end{cases}$$

Classes : $\{p, r\}$, $\{s, q\}$ et {poubelle}. On peut construire l'automate quotient (13).

On a déterminé ça intuitivement, sans algorithme satisfaisant !

Remarque. Quelque soit l'automate de départ, on tombe toujours sur le même automate minimal.

Il manque toujours un algorithme précis pour déterminer la congruence de Nérode !

Algorithme pour calculer $\approx$ sur Q.

Pour $k \in \mathbb{N}$, on définit $p \approx_k q \Leftrightarrow \left(\forall u \in \Sigma^{\leqslant k}, \left(p \xrightarrow{u} f \in F \Leftrightarrow q \xrightarrow{u} f \in F\right)\right) \Leftrightarrow (\forall u \in \Sigma^{\leqslant k}, (u \in L_p \Leftrightarrow u \in L_q))$.

On a $p \approx q \Leftrightarrow (\forall u \in \Sigma^*, (u \in L_p \Leftrightarrow u \in L_q)) \Leftrightarrow L_p = L_q \Leftrightarrow \forall k, p \approx_k q$.

On calcule d'abord $\approx_0$ qui est la relation : $p \approx_0 q \Leftrightarrow (\varepsilon \in L_p \Leftrightarrow \varepsilon \in L_q) \Leftrightarrow (p \in F \Leftrightarrow q \in F)$, les deux classes sont en fait F et $\bar{F}$.

Ensuite, on calcule successivement les $\approx_{k+1}$ à partir des $\approx_k$ en utilisant la relation :

$$p \approx_{k+1} q \Leftrightarrow \left(p \approx_k q \wedge \forall a \in \Sigma, \left(\left\{ \begin{array}{l} p \xrightarrow{a} p' \\ q \xrightarrow{a} q' \end{array} \right. \Rightarrow p' \approx_k q' \right) \right)$$

On s'arrête lorsque $\approx_{k+1}$ et $\approx_k$ deviennent la même relation d'équivalence (on raffine petit à petit notre relation d'équivalence).

En pratique. On utilise une matrice triangulaire adressée par tous les couples non ordonnés p, q.

1. Première itération : on coche les couples p, q tels que $p \in F$ et $q \notin F$ (on coche les cases non équivalentes). (penser à la poubelle quand on le fait à la main)

2. Puis on itère avec une seule règle : si pour p, q non coché, $\exists a : \left\{ \begin{array}{l} p \xrightarrow{a} p' \\ q \xrightarrow{a} q' \end{array} \right.$ tq $p \neq p'$ et p, p' est cochée, alors on coche p, q. On s'arrête lorsqu'on a parcouru une fois tout le tableau sans cocher de nouvelle case.

Maintenant, toute case p, q non cochée implique $p \approx q$.

Remarque. Il existe un algorithme plus efficace (variante en diviser pour régnér : algorighme de Hopcraft, voir livre de Cartan).

Remarque 2. Il existe un autre algorithme, très différent : l'algorithme de Brzozowski (voir en TD).

Rappel : lemme de pompage (étoile, itération)

Raison d'être : preuves de non-régularité.

Lemme. Soit L un langage régulier. Alors :

$$\exists k \in N : \forall w \in L, \left(|w| > k \Rightarrow \left(\exists x, y, z \in \Sigma^* : \left\{ \begin{array}{l} w = x\,y\,z \\ y \neq \varepsilon \\ |x\,y| < k \\ \forall n \in \mathbb{N}, w' = x\,y^n z \in L \end{array} \right. \right) \right)$$

Autrement dit, tout mot long de L peut être gonflé sans sortir du langage L.

Dém. Supposons L régulier. Soit A un automate déterminste qui le reconnait et k son nombre d'états. Soit $w \in L, |w| > k$. On regarde son calcul :

$$\underbrace{i \xrightarrow{w_1} q_1 \to \cdots \xrightarrow{w_n} f \in F}_{\text{plus de } k \text{ états}}$$

L'automate a k états, donc il y a forcément une répétition parmi les premiers $k+1$ états par lesquels on passe. On peut donc rajouter une boucle, et le préfixe $x\,y$ concerné est effectivement de longueur $\leqslant k$ (il sera $<k$ si on prend plutôt le nombre d'états plus un).

Prop. $L = \{a^n b^n \,;\, n \in \mathbb{N}\}$ n'est pas régulier.

Dém. Lemme de pompage : soit k le k donné par le lemme de pompage, on étudie $a^k b^k \in \mathbb{N}$, il existe r tel que $\forall p \in \mathbb{N},\ a^{k-r+pr} b^k \in L$, absurde.

2013-10-10

Langages réguliers, approche algébrique.

Monoides

Déf. Un *monoide* est un ensemble $(M, \cdot, e)$ vérifiant les axiomes :

- $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ (associativité)
- $e\,a = a\,e = a$ (élément neutre)

Exemples de monoides. $(\mathbb{N}, +, 0)$, ou $(\mathbb{N}, *, 1)$

Exemple. Soit A un ensemble et $\mathrm{Rel}(A) = \{\text{toutes les relations binaires sur } A\} = 2^{A\times A}$. On peut composer deux relations binaires en posant : $r_1 \cdot r_2 = \{x, y : \exists z \text{ tq } (x; z) \in r_1 \wedge (z, y) \in r_2\}$. L'associativité est facile à démontrer. L'élément neutre est la relation « égal » : $e = \{(a, a) ; a \in A\}$. On a donc un monoide.

Exercice. Tout monoide est isomorphe à un sous-monoide de $\mathrm{Rel}(A)$ pour un certain A.

Déf. Soit Σ un alphabet, $(\Sigma^*, \cdot, \varepsilon)$ s'appelle *monoide libre* engendré par Σ.

Morphismes de monoides

Déf. Soient deux monoides M_1 et M_2. Un morphisme de M_1 dans M_2 est une application $\varphi : M_1 \to M_2$ telle que $\varphi(a \cdot_1 b) = \varphi(a) \cdot_2 \varphi(b)$ et $\varphi(e_1) = e_2$.

Ex. $M_1 = (\mathbb{N}^{>0}, \times, 1)$ et $M_2 = (\mathbb{R}, +, 0)$. $\log_7 : M_1 \to M_2$ est un morphisme de monoides.

Prop. Soit M un monoide tel que $f : \Sigma \to M$ une fonction. Alors il existe un unique morphisme $\varphi : \Sigma^* \to M$ tel que $\forall a \in \Sigma, \varphi(a) = f(a)$ (propriété de *liberté* de Σ^*).

Ex. $\Sigma = \{a, b\}$. $\mathrm{cout}(a) = 3$, $\mathrm{cout}(b) = 7$. $\mathrm{cout}(b\,b\,a\,a\,b) = ?$. Par exemple, prenons le monoide d'arrivée $M = (\mathbb{N}, +, 0)$. Par morphisme, $\mathrm{cout}(b\,b\,a\,a\,b) = \mathrm{cout}(b) + \mathrm{cout}(b) + \mathrm{cout}(a) + \mathrm{cout}(a) + \mathrm{cout}(b) = 7 + 7 + 3 + 3 + 7 = 27$.

Contre-exemple. $M_1 = \{(0, 1), \times, 1\}$. $\mathrm{cout}(0) = 3$, $\mathrm{cout}(1) = 7$. $\mathrm{cout}(1) = 7 = \mathrm{cout}(1 \times 1) = \mathrm{cout}(1) \times \mathrm{cout}(1) = 49$, il y a un problème avec nos axiomes. On ne peut pas prolonger cette fonction en morphisme, car on n'a pas un monoide libre (il y a des identités non triviales, telles que $0 \times 0 = 0$).

$(\Sigma^*, \cdot, \varepsilon)$ est un monoide libre car il n'y a pas d'autres identités que l'associativité.

Preuve de la propriété. Soit $\varphi(a_1\, a_2 \cdots a_k) \overset{\text{déf}}{=} f(a_1)\ f(a_2) \cdots f(a_k)$ (1) pour tous $k \in \mathbb{N}$ et $a_i \in \Sigma, 1 \leqslant i \leqslant k$, et $\varphi(\varepsilon) = e$ (2). Il faut montrer que φ est un morphisme :

Soient u et v deux mots, $\varphi(u \cdot v) = \varphi(u_1 \cdots u_n \cdot v_1 \cdots v_m) = \varphi(u_1) \cdots \varphi(u_n)\ \varphi(v_1) \cdots \varphi(v_m) = \varphi(u)\,\varphi(v)$.

φ est unique car (1) et (2) sont nécessaires.

Déf. On dit qu'un langage $L \subset \Sigma^*$ est *reconnu* par un monoide M s'il existe $\varphi : \Sigma^* \to M$ un morphisme et $P \subset M$ (un sous ensemble quelconque) tel que $L = \varphi^{-1}(P)$.

Théorème. L est régulier ssi L est reconnu par un monoide fini.

Remarque. Tout langage est reconnu par un monoide infini. En effet, soit $M = \Sigma^*$, $\varphi = \mathrm{id}$ et $P = L$. Alors $L = \mathrm{id}^{-1}(L)$; par définition L est reconnu par Σ^*.

Preuve. Sens $\Leftarrow$: Soit $L = \{w : \varphi(w) \in P\}$ pour $P \subset M$. On construit un automate qui calcule $\varphi(w)$ en lisant w :

- $Q = M$
- $i = e$, $F = P$
- $\delta(m, a) = m \cdot \varphi(a)$

Calcul de $w = a_1 \cdots a_n$: $e \xrightarrow{a_1} \varphi(a_1) \xrightarrow{a_2} \varphi(a_1)\,\varphi(a_2) \xrightarrow{a_3} \cdots \xrightarrow{a_n} \varphi(a_1)\,\varphi(a_2) \cdots \varphi(a_n) = \varphi(w)$. $w \in L \Leftrightarrow \varphi(w) \in P \Leftrightarrow$ le calcul de w termine sur un élément de P, donc cet automate reconnait bien L, donc L est régulier.

Sens $\Rightarrow$: $L = L(A)$, $A = (Q, \Sigma, \Delta, I, F)$. On utilise le monoide $M = \mathrm{Rel}\ Q$ (ensemble des relations binaires sur Q), le morphisme $\varphi(w) = \left\{(p, q) : p \xrightarrow{w} q\right\}$ et l'ensemble final $P = \{r \subset Q \times Q : r \cap I \times F \neq \varnothing\}$. Alors $L = \varphi^{-1}(P)$. En effet :

1. φ est un morphisme :

$$\varphi(\varepsilon) = \left\{(p,q) : p \xrightarrow{\varepsilon} q\right\} = \{(p,p)\} = \mathrm{id}$$

$$\varphi(u\cdot v) = \left\{(p,q) : p \xrightarrow{u\cdot v} q\right\} = \left\{(p,q) : \exists r \,|\, p \xrightarrow{u} r \wedge r \xrightarrow{v} q\right\} = \varphi(u)\cdot\varphi(v)$$

2. $w \in L \Leftrightarrow W$ a un calcul accceptant $\Leftrightarrow I \ni i \xrightarrow{w} f \in F \Leftrightarrow \exists (i,\, f) \in I \times F \,|\, \varphi(w) \ni (i,\, f)$ $\Leftrightarrow \varphi(w) \in P$

Remarque 1. Monoide $\to$ automate : même taille. Automate $\to$ monoide : explosion, en effet $|M| = 2^{|Q|^2}$

Remarque 2. On peut prendre non pas Rel Q mais $\varphi(\Sigma^*) \subset \mathrm{Rel}\ Q$, qui est un monoide beaucoup plus petit et uqe l'on appelle *monoide de transition* de l'automate A.

Remarque 3. $\varphi(w)$ décrit ce que fait w sur Q.

Monoide syntaxiue

Problème. On a un langage L. Comment trouver le meilleur monoide qui le reconnait ?

Solution. Le monoide syntaxique ! (sans blague...)

Déf. On appelle *congruence syntaxique* sur Σ^* pour le langage L la relation binaire :

$$u \overset{\text{syn}}{\approx} v \Leftrightarrow \forall x, y \in \Sigma^*, (x\,u\,y \in L \Leftrightarrow x\,v\,y \in L)$$

Rappel. Congruence de Myhill-Nérode : $u \overset{\text{MN}}{\approx} v$ ssi $\forall y \in \Sigma^*,\ u\,y \in L \Leftrightarrow v\,y \in L$. C'est différent.

Notation. La classe d'équivalence de u est notée $[u]$.

Prop. $\approx$ (notation de $\overset{\text{syn}}{\approx}$) est une congruence :

1. $u \approx v \Rightarrow (u \in L \Leftrightarrow v \in L)$
2. $\left\{\begin{array}{l} u_1 \approx v_1 \\ u_2 \approx v_2 \end{array}\right. \Rightarrow u_1u_2 \approx v_1v_2$

Preuve.

(1) $u \in L \Leftrightarrow \varepsilon\,u\,\varepsilon \in L \overset{\text{déf de } u\approx v}{\Leftrightarrow} \varepsilon\,v\,\varepsilon \in L \Leftrightarrow v \in L$

(2) $x\,u_1\,u_2\,y \in L \overset{u_1\approx v_1}{\Leftrightarrow} x\,v_1\,u_2\,y \overset{u_2\approx v_2}{\Leftrightarrow} x\,v_1\,v_2\,y$

Déf. On note $M = \{[w]\,;\, w \in \Sigma^*\}$, et on l'appelle *monoide syntaxique* de L. $e = [\varepsilon]$ et $[u]\cdot[v] = [u\,v]$, ce qui est cohérent grâce au point 2 ci-dessus.

Prop. M reconnait L.

Dém. Soit $\varphi : \Sigma^* \to M, w \mapsto [w]$, et $P = \{[w]\,;\, w \in L\}$.

φ est un morphisme : $\varphi(\varepsilon) = [\varepsilon] = e$ et $\varphi(u\,v) = [u\,v] = [u]\,[v] = \varphi(u)\,\varphi(v)$.

$\varphi(u) \in P \Leftrightarrow [u] = [w], w \in L \Leftrightarrow u \in L$, donc ce monoide reconnait bien L.

Déf. $M \lhd K$ (M divise K) s'il existe N un sous monoide de K et $\varphi : N \to M$ un morphisme surjectif.

Remarque. $|M| \leqslant |K|$.

Théorème. Soit L un langage et M son monoide syntaxique. Alors un monoide K reconnait L si et seulement si $M \lhd K$.

Remarque. Même démarche que la minimisation d'automates.

Corollaire. L régulier ssi son monoide syntaxique est fini.

Corollaire. Pour tout Kqui reconnait L, $|M| \leqslant |K|$.

Démonstration du théorème. Sens $\Leftarrow$: à faire en exo.

Sens $\Rightarrow$: on a K qui reconnait L : il y a $\psi : \Sigma^* \to K$, $P \subset K$ tels que $L = \psi^{-1}(P)$.

On construit $N = \psi(\Sigma^*)$, $P' = P \cap N$. On a :

- Tout élément $x \in N$ s'écrit $x = \psi(w)$ pour un certain w.
- $w \in L \Leftrightarrow \psi(w) \in P \Leftrightarrow \psi(w) \in P'$.

On veut construire le morphisme $\alpha : N \to M, \psi(w) \mapsto [w]$, mais il faut justifier que cette définition est correcte :

- montrons que si $\psi(u) = \psi(w)$ alors $\alpha(\psi(u)) = \alpha(\psi(v))$, ie $[u] = [v]$:

  si $\psi(u) = \psi(w)$, $x\,u\,y \in L \Leftrightarrow \psi(x\,u\,y) \in P \Leftrightarrow \psi(x)\,\psi(u)\psi(y) \in P \Leftrightarrow \psi(x)\psi(w)\psi(y) \in P \Leftrightarrow \psi(x\,w\,y) \in P \Leftrightarrow x\,w\,y \in L$, donc $u \approx w$, donc $[u] = [w]$.
- montrons que α est un morphisme : ok, c'est assez clair.
- montrons que α est surjectif ($\alpha(N) = M$) :

  $M = \{[w]\,; w \in \Sigma^*\}$, chaque élément s'écrit $[w]$. $[w] = \alpha(\psi(w)) \in \operatorname{Im}\alpha$, donc il est surjectif.

Problème. Comment décrire/calculer le monoide syntaxique à partir de l'automate minimal ?

Rappel (Myhill-Nérode). $u \overset{\text{MN}}{\approx} v$ ssi $u\backslash L = v\backslash L$ ssi dans l'automate minimal, $q_0 \xrightarrow{u} q$ et $q_0 \xrightarrow{v} q$ (on arrive sur le même état en lisant u et v à partir de q_0).

Prop. $u \approx v$ si dans l'automate minimal, $\forall p \in Q$, $\exists q \in Q$ tq $p \xrightarrow{u} q$ et $p \xrightarrow{v} q$, ie $\forall p \in Q$, $S(p,u) = S(p,v)$.

Dém. Sens $\Leftarrow$: $x\,u\,y \in L \Leftrightarrow i \xrightarrow{x} p \xrightarrow{u} q \xrightarrow{y} f \Leftrightarrow i \xrightarrow{x} p \xrightarrow{v} q \xrightarrow{y} f \Leftrightarrow x\,v\,f \in L$ (« ça se voit »).

Sens $\Rightarrow$: on raisonne par contraposée. Dans l'automate, $\exists p \in Q$ tel que $p \xrightarrow{u} q_1$ et $p \xrightarrow{v} q_2$ avec $q_1 \neq q_2$. Puisque l'automate est minimal, $L_{q_1} \neq L_{q_2}$. Soit $w \in L_{q_1}$ tel que $w \notin L_{q_2}$ (ou symmétriquement, selon la composition de L_{q_1} et L_{q_2}). De plus, il existe un mot x tel que $q_0 \xrightarrow{x} p$. Donc $x\,u\,w \in L$ et $x\,v\,w \notin L$, donc $u \not\approx v$.

Prop. Le monoide syntaxique M de L est isomorphe au monoide de transition de l'automate minimal de L.

monoide syntaxique	monoide de transition
$w \mapsto [w]$	$w \mapsto \left\{(p,q) : p \xrightarrow{w} q\right\}$
$[\varepsilon]$	id

Tableau 1. Isomorphisme entre le monoide syntaxique et le monoide de transition.

Exercice. $L = (a\,b)^*$, calculer son monoide syntaxique.

	$q \in I, \in F$	r	p
a	r	p	p
b	p	q	p

Tableau 2. Tableau des transitions de l'automate minimal reconaissant L. (faire le schéma correspondant)

$w \in \Sigma^*$	$r \in \mathrm{Rel}\, Q$	nom pour r
ε	$p \to p, \boxed{q \to q}, \boxed{r \to r}$	e
a	$p \to p, \boxed{q \to r}, r \to p$	A
b	$p \to p, q \to p, \boxed{r \to q}$	B
$a\,a$	$p \to p, q \to p, r \to p$	0 (absorbant)
$a\,b$	$p \to p, \boxed{q \to q}, r \to p$	$A\,B$
$b\,a$	$p \to p, q \to p, \boxed{r \to r}$	$B\,A$
$b\,b$	$p \to p, q \to p, r \to p$	0
$a\,b\,a$	$p \to p, \boxed{q \to r}, r \to p$	A
$a\,b\,b$	$p \to p, q \to p, r \to p$	0
$b\,a\,a$	$p \to p, q \to p, r \to p$	0
$b\,a\,b$	$p \to p, q \to p, \boxed{r \to q}$	B

Tableau 3. Recherche de tous les éléments du monoide syntaxique. Les transitions encadrées sont celles, utiles, qui ne vont pas vers l'état « poubelle » p.

On découvre ici les deux identités non triviales : $A\,B\,A = A$ et $B\,A\,B = B$.

$\times$	e	0	A	B	$A\,B$	$B\,A$
e	e	0	A	B	$A\,B$	$B\,A$
0	0	0	0	0	0	0
A	A	0	0	$A\,B$	0	A
B	B	0	$B\,A$	0	B	0
$A\,B$	$A\,B$	0	A	0	$A\,B$	0
$B\,A$	$B\,A$	0	0	B	0	$B\,A$

Tableau 4. Table de multiplications du monoide syntaxique.

Utilité des monoides

1. Ça montre que les langages régulies forment une classe naturelle.

2. On peut démontrer des trucs par monoides.

   Ex. Soit $h : \Sigma^* \to \Gamma^*$ un morphisme. L est régulier (sur Γ^*) $\Rightarrow h^{-1}(L)$ est régulier.

   > **Démonstration.** L régulier $\Rightarrow L = \varphi^{-1}(P)$ pour un monoide fini K, un ensemble $P \subset K$ et un morphisme $\varphi : \Gamma^* \to K$. $h^{-1}(L) = h^{-1}(\varphi^{-1}(P)) = (\varphi \circ h)^{-1}(P)$, donc $h^{-1}(L)$ reconnaissable par le même K.

   Ex. On peut prouver les propriétés habituelles sur l'union, l'intersection, etc. de langages réguliers par monoides.

3. Sujet très riche : on prend un monoide B autre que Σ^*. On définit $L \subset B$ reconnu par K ssi $\exists \varphi : B \to K$ et $P \subset K$ tel que $L = \varphi^{-1}(P)$.

   Ex. Sur $\Sigma^* \times \Sigma^*$, on peut étudier ses langages reconaissables.

4. Soit C une classe de monoides. Quels sont les langages reconnus par les monoides de cette classe ?

   Ex. $C =$ classe des groupes.

Langages sans étoile (Star Free)

On s'intéresse ici à une sous-classe importante des langages réguliers.

Déf. Une expression régulière star-free est définie par les éléments de syntaxe :

- $\varnothing$
- ε
- $a, a \in \Sigma$
- $f \cdot g$ avec f et g des expressions star-ree
- $f + g$ avec f et g des expressions star-free
- $\bar{f}$ avec f une expression star-free

Déf. On dit que L est Star Free s'il est défini par une expression régulière star-free.

Exemples.

- $\{a\,b\,b, a\} = a\,b\,b + a$

  plus généralement, tout ensemble fini de mots est star-free
- $\Sigma^* = \bar{\varnothing}$
- Sur $\Sigma = \{a, b, c\}$, $(a+b)^*$ est l'ensemble des mots ne contenant pas c, c'est donc $\overline{\Sigma^* c\,\Sigma^*}$.
- Sur $\Sigma = \{a, b\}$, $(a\,b)^*$ est l'ensemble des mots ne contenant ni $a\,a$ ni $b\,b$, ne commencant pas par b et ne terminant pas par a. $(a\,b)^* = \overline{b\,\Sigma^* + \Sigma^* a + \Sigma^* a\,a\,\Sigma^* + \Sigma^* b\,b\,\Sigma^*}$.
- $(a\,a)^*$ est l'ensemble des mots de longueur pair. On verra qu'il n'est pas Star Free.

Un automate fini sait faire deux choses : reconnaitre des motifs, et compter modulo k. Pour les automates Star Free, on n'utilise que la première capacité.

Déf. Un monoide M est *apériodique* si $\forall x \in M$, $\exists k \in \mathbb{N}$ tel que $x^{k+1} = x^k$.

Pour une séquence : e, x, x^2, x^3, x^4, ... d'éléments de M, il y a plusieurs possibilités : on a une séquence infinie, on a un cycle, ou on a une stabilisation.

Prop. Variante : un monoide fini M est apériodique si $\exists k \in \mathbb{N}, \forall x \in M, x^{k+1} = x^k$.

Commentaire. Dans un monoide fini, prenons une séquence du type e, x, x^2, x^3, ... : dans une telle séquence, il y a forcément une répétition : $x^m = x^n$ pour $m < n$. Alors $x^{m+1} = x^{n+1}$, etc. Un tel élément s'appelle « poelle à frire » car si on trace une droite avec les x^i, on a une boucle à la fin qui revient en arrière (faire le dessin). Dans le cas d'un monoide apériodique, il n'y a que le manche de la poelle. (haha)

Théorème (Schutzenberger, 1965). Soit L un langage régulier. Alors L est star-free $\Leftrightarrow$ son monoide syntaxique est apériodique $\Leftrightarrow$ L est reconnu par un monoide apériodique.

Rmq. Le monoide syntaxique est apériodique ssi $\exists k \in \mathbb{N}$, $\forall u \in L$, $\forall x, y \in \Sigma^*$, $(x\,u^{k+1} y \in L \Leftrightarrow x\,u^k y \in L)$

Ex. Sur $(a\,a)^*$, $\varepsilon \in L, a \notin L, a^2 \in L, a^3 \notin L$, etc. Cette suite ne se stabilise pas (il y a un cycle de période 2 dans le monoide syntaxique), donc le monoide syntaxique n'est pas apériodique : le langage n'est donc pas star-free.

2013-10-17

Rappels.

Déf. Une expression sans étoile est définie par la syntaxe :

$$\mathrm{SF} = \varnothing \,|\, \varepsilon \,|\, a \,|\, \mathrm{SF} + \mathrm{SF} \,|\, \mathrm{SF} \cdot \mathrm{SF} \,|\, \overline{\mathrm{SF}}$$

$(a\,b)^*$ peut être décrit, $(a\,a)^*$ non.

Déf. L est star-free s'il peut être décrit par une expression star-free.

Prop. Un monoide fini M est apériodique si $\exists k \in \mathbb{N} \,|\, \forall x \in M, x^{k+1} = x^k$

Déf. L régulier est apériodique si son monoide syntaxique est apériodique.

Prop. L (régulier) est apériodique $\Leftrightarrow$ L est reconnu par un monoide fini apériodique.

Dém. $\Rightarrow$: Si L est apériodique, son monoide syntaxique est apériodique, et le reconnait.

$\Leftarrow$: Si L est reconnu par un monoide K apériodique, son monoide syntaxique M divise K, c'est-à-dire que M est image homomorphe d'un sous-monoide de K. M est donc apériodique.

Reformulation.

$$\begin{aligned} L \text{ régulier est apériodique } &\Leftrightarrow \exists k \in \mathbb{N} \,|\, \forall w \in \Sigma^*, [w^k] = [w^{k+1}] \\ &\Leftrightarrow \exists k \in \mathbb{N} \,|\, \forall w \in \Sigma^*, \forall x, y \in \Sigma^*, (x\,w^k\,y \in L \Leftrightarrow x\,w^{k+1}\,y \in L) \end{aligned}$$

Dans l'automate minimal de L, en partant de n'importe quel état p, en lisant le mot w successivement un certain nombre de fois on finit par boucler sur un seul état q, i.e. on n'a pas la configuration suivante : $p \xrightarrow{w} q \xrightarrow{w} p, p \neq q$ (ni avec une boucle de taille quelconque $\geqslant 2$). Si on a un cycle de type $q_0 \xrightarrow{w} q_1 \xrightarrow{w} \cdots \xrightarrow{w} q_{n-1} \xrightarrow{w} q$, on dit que l'automate compte modulo n.

Prop. L régulier est apériodique $\Leftrightarrow$ son automate minimal n'a pas de compteur.

Prop. L est apériodique $\Leftrightarrow$ $L = L(A)$ où A est un automate déterministe sans compteur.

Dém. en exo.

Ex. $(a\,b)^*$, dessiner l'automate, $p \xrightarrow{a} q \xrightarrow{a} r \xrightarrow{a} r \xrightarrow{a} \cdots$, $p \xrightarrow{ab} p \xrightarrow{ab} \ldots$

En revanche, tout automate reconaissant $(a\,a)^*$ contient une période.

Rappel. Théorème (Schutzenberger, 1965). Soit L un langage régulier. Alors L est apériodique $\Leftrightarrow$ L est star-free.

Dém. $\Rightarrow$: omise (se trouve facilement)

$\Leftarrow$: par induction structurelle sur l'expression star-free.

- $\varnothing$ apériodique : on prend $k = 0$ dans le langage, $x\,w^0\,y \in L \Leftrightarrow x\,w^1 y \in L$, c'est évident car L est vide
- ε apériodique : il faut prendre $k = 1$; $x\,w\,y \in L \Leftrightarrow x = y = w = \varepsilon \Leftrightarrow x\,w^2\,y \in L$
- a est apériodique : prenons $k = 2$, $x\,w^2\,y \in L \Leftrightarrow x\,w^2\,y = a \Leftrightarrow w = \varepsilon, x\,y = a \Leftrightarrow x\,w^3\,y \in L$.
- L apériodique $\Rightarrow \bar{L}$ apériodique : évident car $(x\,w^k y \in L \Leftrightarrow x\,w^{k+1} y \in L) \Leftrightarrow (x\,w^k y \in \bar{L} \Leftrightarrow x\,w^{k+1} y \in \bar{L})$
- L, M apériodiques avec $k_L, k_M \Rightarrow L \cup M$ apériodique : prenons $k_{L \cup M} = \max(k_L, k_M)$, c'est maintenant évident.
- L, M apériodiques avec k, $l \Rightarrow L \cdot M$ apériodique : prenons $n = k + m + 1$. Soit $x\,w^n y \in L \cdot M$, $x\,w^n\,y = l\,m$, avec $l \in L$ et $m \in M$.
  - Si $|l| \leqslant |x|$, on a $(l^{-1}x)\,w^{n+1}\,y \in M$ car $n \geqslant k$, et donc $x\,w^{n+1} y \in L \cdot M$

- De même si $|m| \leqslant |y|$

- Sinon, soit d'un côté soit de l'autre il y a $\geqslant k$ ou $\geqslant m$ fois le mot w, on gonfle donc ce côté là.

  $x\,w^n\,y = x\,w^a\,w\,w^b\,y$ avec $x\,w^a$ préfixe de l et $w^b\,y$ sffixe de m, on a soit $a \geqslant k$ soit $b \geqslant m$, si par exemple $a \geqslant k$ on a $x\,w^{a+1}((x\,w^a)^{-1}l) \in L$, donc $x\,w^{n+1}\,y \in L$.

Réciproquement, si $x\,w^{n+1}\,y \in L \cdot M$ on raisonne de même (il y a toujours un côté où on peut enlever un w ; on a bien besoin ici de $n = k+m+1$ et non simplement $n = k+m$).

Langages réguliers et logique

$w \in (\Sigma^* a\,a\,b\,\Sigma^* - b\,b\,b\,\Sigma^*) \Leftrightarrow$ il y a un $a\,a\,b$ quelque part et pas $b\,b\,b$ au début $\Leftrightarrow (\exists x : a(x) \wedge a(x+1) \wedge b(x+2)) \wedge \neg(b(0) \wedge b(1) \wedge b(2))$

$(a\,b)^* = \ \forall x, (a(x) \Rightarrow b(x+1)) \wedge (b(x) \Rightarrow (a(x+1) \vee x+1 = \text{fin}))$

On n'a utilisé que des formules de logique de premier ordre (FO).

$(a\,a)^* = \exists P : (P(0) \wedge \forall x, ((P(x) \Rightarrow \neg P(x+1) \wedge \text{fin}(x+1)) \wedge \neg P(x) \Rightarrow P(x+1))) \wedge \forall x, (x \neq \text{fin} \Rightarrow a(x))$

Cette dernière formule utilise un quantificateur de logique monadique du second ordre (MSO), dans laquelle on a droit d'utiliser un prédicat unaire avec des quantificateurs dessus.

Théorème.

- L est régulier $\Leftrightarrow$ il est définissable en logique $\text{MSO}(\Sigma, <)$.

- L est star-free $\Leftrightarrow$ il est définissable en logique $\text{FO}(\Sigma, <)$.

Déf. On fixe un alphabet Σ.

- Formules logiques FO : avec des variables (x, y, z, ...) (qui signifient des positions dans le mot) et la syntaxe :

$$\begin{array}{rl} \text{formules élémentaires} & x < y \,|\, x = y \,|\, a(x) \\ \text{si } f \text{ et } g \text{ sont des formules} & f \vee g \,|\, \neg f \\ \text{si } f \text{ est une formule} & \exists x : f \end{array}$$

  Sucre syntaxique pour les formules : on peut ajouter les constructions suivantes

$$\begin{array}{rcl} f \wedge g & \Leftrightarrow & \neg(\neg f \vee \neg g) \\ \forall x, f & \Leftrightarrow & \neg(\exists x : \neg f) \\ x = 0 & \Leftrightarrow & \neg(\exists y : y < x) \\ x = \text{fin} & \Leftrightarrow & \neg(\exists y : x < y) \\ y = x+1 & \Leftrightarrow & x < y \wedge \neg(\exists z : (x < z \wedge z < y)) \end{array}$$

- Formules MSO : on rajoute les variables pour prédicat qui sont $X, Y, Z, ...$ (et qui correspondent à des ensembles de positions) puis on rajoute les constructions suivantes :

$$\begin{array}{rl} \text{nouvelle formule de base} & X(x) \\ \text{si } f \text{ est une formule} & \exists X : f \end{array}$$

Nous n'avons pas encore défini la sémantique de nos formules.

Ex. $\exists x\colon (x < y \vee a(z))$ est une formule FO. $\exists Y\colon (Y(z) \vee b(x))$ est une formule MSO. Ces deux formules ne veulent rien dire.

Déf. Variables libres :

- $\mathrm{Lib}(x<y) = \{x, y\}$
- $\mathrm{Lib}(x=y) = \{x, y\}$
- $\mathrm{Lib}(a(x)) = \{x\}$
- $\mathrm{Lib}(\neg f) = \mathrm{Lib}(f)$
- $\mathrm{Lib}(f \vee g) = \mathrm{Lib}(f) \cup \mathrm{Lib}(g)$
- $\mathrm{Lib}(\exists x\colon f) = \mathrm{Lib}(f) \setminus \{x\}$
- $\mathrm{Lib}(X(x)) = \{X, x\}$
- $\mathrm{Lib}(\exists X\colon f) = \mathrm{Lib}(f) \setminus \{X\}$

Sémantique FO. Soit $w \in \Sigma^*$, on interprète les variables $x, y, z, ...$ sur $\{0, ..., n\}$, où $n = |w|$ (la position n correspond à la fin du mot).

Soit $w \in \Sigma^*$, soit f une formule avec variables libres $x_1, ..., x_k$. Comment définir si f est vraie? Cela dépend de w, qui est fixé, et des valeurs des variables libres. Soient $n_1, ..., n_k \in \{0, ..., n\}$, $[x_i] = n_i$: on a défini une valuation σ des variables libres. On pose pour sémantique :

$$\begin{aligned}
[x_i < x_j]_{w,\sigma} &\Leftrightarrow n_i < n_j \\
[x_i = x_j]_{w,\sigma} &\Leftrightarrow n_i = n_j \\
[a(x_i)]_{w,\sigma} &\Leftrightarrow w_{n_i} = a \\
[f \vee g]_{w,\sigma} &\Leftrightarrow [f]_{w,\sigma} \vee [g]_{w,\sigma} \\
[\neg f]_{w,\sigma} &\Leftrightarrow \neg [f]_{w,\sigma} \\
[\exists x\colon f]_{w,\sigma} &\Leftrightarrow \exists m \in \{0, ..., n\} : [f]_{w,\sigma[[x]=m]} = \mathrm{true}
\end{aligned}$$

On pose, si f a des variables libres :

$$\begin{aligned}
[\![f]\!] &= \{(w, n_1, ..., n_k) : [f]_{w,\sigma} = \mathrm{true}\} \\
&\subset \Sigma^* \times \mathbb{N}^k
\end{aligned}$$

Si f n'a pas de variables libres (on dit que f est *close*) :

$$\begin{aligned}
[\![f]\!] &= \{w \mid [f]_w\} \\
&\subset \Sigma^*
\end{aligned}$$

Donc une formule close définit un langage.

Exemples. $[\![a(x) \vee (\exists y\colon (y < x))]\!] = \{w, m = [x] : w_m = a \wedge m \neq 0\}$

$[\![\exists x\colon (a(x) \wedge \exists y\colon (y < x))]\!] = \Sigma^+ a \, \Sigma^*$

Sémantique MSO. Les variables $X, Y, ...$ prennent leurs valeurs dans $\mathcal{P}(\{0, ..., n\})$: $[X_i] = A_i$. Il reste à poser :

$$\begin{aligned}
[X_i(x_j)]_{w,\sigma} &\Leftrightarrow n_j \in A_i \\
[\exists X\colon f]_{w,\sigma} &\Leftrightarrow \exists A \subset \{0, ..., n\} \mid [f]_{w,\sigma([X]=A)} = \mathrm{true}
\end{aligned}$$

On a maintenant, pour une formule dont les variables libres sont $x_1, ..., x_k, X_1, ..., X_l$:

$$\begin{aligned}[\![f]\!] &= \{(w, n_1, ..., n_k, A_1, ..., A_l) \mid [f]_{w,\sigma} = \text{true}\} \\ &\subset \Sigma^* \times \mathbb{N}^k \times (\mathcal{P}(\mathbb{N}))^l\end{aligned}$$

Théorème.

1. L est régulier $\Leftrightarrow$ il existe f une formule close MSO telle que $L = [\![f]\!]$
2. L est régulier star-free $\Leftrightarrow$ il existe f une formule close FO telle que $L = [\![f]\!]$

Dém.

1. Sens $\Rightarrow$: on prend un automate fini et on décrit ses calculs en logique MSO.

   Soit $A = (\Sigma, Q, \Delta, I, F)$.

   $w \in L \Leftrightarrow$ il existe $q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \cdots \xrightarrow{w_{n-1}} q_n$ avec $q_0 \in I$ et $q_n \in F$ et $\forall x \in \{0, n-1\}$,$(q_x, w_x, q_{x+1}) \in \Delta$.

   Pour chaque état $q \in Q$, on introduit un prédicat X_q qui est vrai sur toutes les positions où l'automate passe (potentiellement, dans le cas nondéterministe) par l'état q.

   $Q = \{q_1, ..., q_n\}$

   $w \in L \Leftrightarrow f$ avec $f = \exists X_{q_1} : \exists X_{q_2} \cdots \exists X_{q_n} : (\text{init} \wedge \text{fin} \wedge \text{trans} \wedge \text{mutex})$, où

$$\begin{aligned}\text{init} &= \bigvee_{q \in \text{Init}} X_q(0) \\ \text{fin} &= \bigvee_{q \in F} X_q(\text{fin}) \\ \text{trans} &= \forall x, \left(x \neq \text{fin} \wedge \bigvee_{(p,a,q) \in \Delta} (X_p(x) \wedge a(x) \wedge (\exists y : y = x + 1 \wedge X_q(y))) \right) \\ \text{mutex} &= \forall x, \left(\left(\bigwedge_{p,q \in Q} (\neg X_q(x) \vee \neg X_p(x)) \right) \wedge \bigvee_{p \in Q} X_p(x) \right)\end{aligned}$$

   On vérifie que la formule ainsi construit définit bien le langage L.

   Sens $\Leftarrow$: soit f une formule, on va prouver que $[\![f]\!]$ est une langage régulier par induction structurelle. Mais $[\![f]\!] = \{(w, n_1, ..., n_k, A_1, ..., A_l) : f\}$ n'est pas un langage, on ne peut pas parler de régularité.

   On codera donc tout le tuple $(w, n_1, ..., n_k, A_1, ..., A_l)$ comme un mot : l'alphabet devient $(\Sigma \times \{0,1\}^{k+l}) \cup \{\varnothing\}$, où les termes du tuple alphabet correspondant à n_i dans une lettre valent 1 ssi n_i est la position où se trouve cette lettre, et les positions correspondant à A_i dans une lettre valent 1 ssi la position où se trouve cette lettre appartient à A_i. On obtient une matrice encodant $(w, n_1, ..., n_k, A_1, ..., A_l)$:

$a \in \Sigma$	w_0	w_1	w_2	w_3	w_4	...	w_{n-1}	$\varnothing$
n_1	0	0	0	1	0	...	0	
⋮								
n_k	0	0	1	0	0	...	0	
A_1	1	0	1	0	1	...	0	
⋮								
A_l	0	1	1	0	1	...	1	

Tableau 5. Les colonnes de ce nouveau mot sont des lettres du nouvel alphabet.

Soit f une formule, on peut maintenant tenter de prouver que $[\![f]\!]$ est régulier :

- $[\![x<y]\!] = \left\{ \begin{array}{|c|ccccc|} \hline w & \ldots & \ldots & \ldots & \ldots & \ldots \\ x & (0) & 1 & (0) & 0 & (0) \\ y & (0) & 0 & (0) & 1 & (0) \\ \hline \end{array} \right\} = \begin{array}{|c|}\hline \Sigma \\ 0 \\ 0 \\ \hline \end{array}^* \cdot \begin{array}{|c|}\hline \Sigma \\ 1 \\ 0 \\ \hline \end{array} \cdot \begin{array}{|c|}\hline \Sigma \\ 0 \\ 0 \\ \hline \end{array}^* \cdot \begin{array}{|c|}\hline \Sigma \\ 0 \\ 1 \\ \hline \end{array} \cdot \begin{array}{|c|}\hline \Sigma \\ 0 \\ 0 \\ \hline \end{array}^*$, il est bien régulier (et même star-free).

- $[\![a(x)]\!] = \left\{ \begin{array}{|c|ccc|} \hline w & \ldots & a & \ldots \\ x & (0) & 1 & (0) \\ \hline \end{array} \right\} = \begin{array}{|c|}\hline \Sigma \\ 0 \\ \hline \end{array}^* \cdot \begin{array}{|c|}\hline a \\ 0 \\ \hline \end{array} \cdot \begin{array}{|c|}\hline \Sigma \\ 0 \\ \hline \end{array}^*$, il est bien régulier

- $[\![X(x)]\!] = \left\{ \begin{array}{|c|ccc|} \hline w & \ldots & \ldots & \ldots \\ x & (0) & 1 & (0) \\ y & (0 \text{ ou } 1) & 1 & (0) \\ \hline \end{array} \right\} = \begin{array}{|c|}\hline \Sigma \\ 0 \\ \{0,1\} \\ \hline \end{array}^* \cdot \begin{array}{|c|}\hline \Sigma \\ 1 \\ 1 \\ \hline \end{array} \cdot \begin{array}{|c|}\hline \Sigma \\ 0 \\ \{0,1\} \\ \hline \end{array}^*$

- Si $\text{enc}[\![f]\!]$ et $\text{enc}[\![g]\!]$ sont réguliers, $\text{enc}[\![f \vee g]\!] = \text{enc}[\![f]\!] \cup \text{enc}[\![g]\!]$, il est donc régulier (il se peut que f et g aient un nombre inégal de variables libres, ce ne sont pas des mots sur le même alphabet, il faut donc insérer des pistes parfois inutilisées dans notre alphabet afin de pouvoir exprimer les deux formules, ce qui correspond à un morphisme inverse sur des alphabets, et préserve donc la régularité)

- $\text{enc}[\![\neg f]\!] = \overline{\text{enc}[\![f]\!]} \cap K$, où K est l'ensemble des mots corrects, c'est-à-dire n'ayant qu'un seul 1 sur les pistes de premier ordre, et qui ont toujours la lettre $\varnothing$ à la fin (c'est clairement un langage régulier).

- $\text{enc}[\![\exists x\colon f]\!] = h(\text{enc}[\![f]\!])$, où h est le morphisme qui efface la piste correspondant à la variable x (c'est pareil pour les variables du second ordre).

La preuve du premier point est ainsi terminée.

2. Second point : cf TD du 18/10 et 25/10.

Corollaire. MSO($<$) est décidable (sans Σ).

La logique FO est donc la logique du premier ordre avec la signature : $<$ et $=$ (prédicats binaire) et $a()$ (prédicat unaire pour chaque $a \in \Sigma$).

2013-10-24

1. Les notions de langages réguliers et langages star-free sont donc naturelles.

2. *Applications d'automates à la logique.*

   Corollaire. Étant donné une formule close MSO($<$) (on l'interprète sur $\{0, ..., n\}$). On peut dire si elle est valide (c'est-à-dire vraie pour tous les n).

   Ex. $f = \forall X, \exists x\colon \forall Y, \forall y, (Y(y) \wedge X(x) \Rightarrow y > x)$

   Algo. Construire l'automate pour $\neg f$ et tester si son langage est $\varnothing$.

   Déf. On dit que MSO($<$) interprétée sur $\{0, ..., n\}$ est décidable.

   Corrolaire. Chaque formule de MSO($\Sigma, <$) est équivalente à une formule existentielle de la forme :

   $$\exists X_1, \exists X_2, ..., \exists X_p, (\text{formule FO})$$

   Preuve. Soit f une formule close MSO($\Sigma, <$) quelconque. On la traduit en automate, puis en formule MSO existentielle.

3. *Vérification et model-checking.*

   On a un programme P (ou un circuit, protocole, ...). On a une spécification S (propriété).
   Question. Toutes les exécutions de P vérifient-elles la spécification S ?

   Méthode.

   a. Modéliser P par un automate

   b. Écrire S en logique

   c. On décide $L_P \subset L_S$ en utilisant des automates

ω-mots et ω-langages réguliers

On se donne un alphabet Σ.

Déf. Un ω-mot est une séquence infinie $v = a_1\, a_2\, a_3 \cdots$ avec $a_i \in \Sigma$. (formmellement $v : \mathbb{N} \to \Sigma$).

Exemples.

- $w_1 = a\, b\, a\, b\, a\, b\, a\, b\, a\, b \cdots$
- $w_2 = 3141592653589793238462643383 \cdots$

Notation. On note l'ensemble de tous les ω-mots : Σ^ω.

Déf. On appelle ω-langage tout sous-ensemble de Σ^ω.

Exemples.

- $L_1 = \{w_1\}$, $L_2 = \{w_2\}$
- $L_3 = \{\text{mots comportant un nombre fini de } a\}$
- $L_4 = \overline{L_3} = \{\text{mots comportant un nombre infini de } a\}$
- $L_{2.5} = \{\text{mots sans } a\, b\, b\, a\}$

Il y a plusieurs définitions d'automates sur les mots infinis. On ne s'intéresse ici qu'aux automates de Büchi.

Déf. Un automate de Büchi est un tuple $A = (Q, \Sigma, \Delta, I, F)$ où

- Q est un ensemble fini d'états
- Σ est un alphabet
- $\Delta \subset Q \times \Sigma \times Q$
- $I, F \subset Q$

Cet automate est défini exactement comme un automate fini ; simplement, il sera utilisé différemment.

Déf. Un calcul sur un ω-mot $w = a_0 a_1 \cdots$ est une séquence infinie $q_0 \xrightarrow{a_9} q_1 \xrightarrow{a_1} q_2 \to \ldots$ tel que $\forall i$, $(q_i, a_i, q_{i+1}) \in \Delta$.

Déf. Un calcul sur un ω-mot est accepteur si $q_0 \in I$ et s'il existe $q_f \in F$ visité une inifité de fois dans le calcul.

Notation. $w \in L^\omega(A)$ s'il existe un calcul accepteur de A sur w.

Déf. L un ω-langage est ω-réguliser ssi $\exists A : L = L^\omega(A)$.

Théorème de « Kleene »

Expressions ω-régulières.

$$\mathrm{RE} = \varepsilon \mid \varnothing \mid A \mid \mathrm{RE} + \mathrm{RE} \mid \mathrm{RE} \cdot \mathrm{RE} \mid \mathrm{RE}^*$$

$$\mathrm{ORE} = \mathrm{ORE} + \mathrm{ORE} \mid \mathrm{RE} \cdot \mathrm{ORE} \mid (\mathrm{RE})^\omega$$

Exemple. $(a+b)^* \cdot a \cdot (a\,b)^\omega$

Sémantique. Soient f, g deux formules de ORE :

- $[\![f+g]\!] = [\![f]\!] \cup [\![g]\!]$
- $[\![f \cdot g]\!] = \{u \cdot v ; u \in [\![f]\!], v \in [\![g]\!]\}$. Il est clair que l'on peut concaténer un mot fini au début d'un ω-mot tout en restant dans Σ^ω.
- $[\![f^\omega]\!] = \{u_1 \cdot u_2 \cdot u_3 \cdots \text{ concaténation infinie tq } \forall i \in \mathbb{N}^*, u_i \in [\![f]\!] \setminus \{\varepsilon\}\}$

Théorème. L est ω-régulier $\Leftrightarrow$ L est définissable par une ORE.

Dém. Si $L = L^\omega(A)$:

On note $L_{p,q} = \left\{v \in \Sigma^* : p \xrightarrow{w} q \text{ dans } A\right\}$. $L_{p,q}$ est clairement régulier.

$$L = \bigcup_{i \in I} \bigcup_{f \in F} L_{i,f} \cdot (L_{f,f})^\omega \tag{1}$$

Démonstration de (1). Soit $w \in L^\omega(A)$, son calcul accepteur commence dans $i \in I$ et visite une infinité de fois un certain $f \in F$.

$i \xrightarrow{\in L_{i,f}} f \xrightarrow{\in L_{f,f}} f \xrightarrow{\in L_{f,f}} f \to \ldots$

On a bien $w \in L_{i,f} \cdot (L_{f,f})^\omega$

Réciproquement, soit $\omega \in L_{i,f} \cdot (L_{f,f})^\omega$ pour un certain i, f. C'est symmétrique, on a donc $w \in L^\omega(A)$.

On traduit par théorème de Kleene fini tous les $L_{i,f}$ et $L_{f,f}$ en RE, on obtient ainsi une ORE pour L.

Réciproquement, si $L = [\![f]\!]$, f ORE :

Comment trouver un automate A tel que $L^\omega(A) = L = [\![f]\!]$? Induction structurelle sur f, laissé en exercice.

Propriétés

1. Les langages ω-réguliers sont clos par : $\cup, \cap, {}^-, h(\cdot), h^{-1}(\cdot)$ (précautions à prendre : il faut rester parmi les ω-langages)
2. Les problèmes suivants sont décidables pour $L = L^\omega(A)$, $M = L^\omega(B)$:
   - $L = \varnothing$
   - $L = \Sigma^\omega$

- $L = M$
- $L \subset M$

Preuve.

- $\cup$: facile, on met les deux automates côte-à-côte
- $\cap$: On fait $A \times B$; ω accepté si son calcul visite infiniment souvent $F_A \times Q_B$ et infiniment souvent $Q_A \times F_B$. Cette construction peut se faire en utilisant $Q_A \times Q_B \times \{0,1\}$ comme ensemble d'états. Il faut que une infinité de fois on visite un état de F_B en ayant visité juste avant (en terme d'états finaux pour A ou B) un état de F_A.

  On construit l'automate $A \times B \times \{0,1\}$:

  $Q = Q_A \times Q_B \times \{0,1\}$

  $I = I_A \times I_B \times \{0\}$, $F = F_A \times Q_B \times \{0\} \cup Q_A \times F_B \times \{1\}$

  $\Delta = \left\{ \begin{pmatrix} p_A \\ p_B \\ 0 \end{pmatrix} \xrightarrow{a} \begin{pmatrix} q_A \\ q_B \\ 0 \end{pmatrix} \text{ si } \begin{matrix} p_A \xrightarrow{a} q_A \\ p_B \xrightarrow{a} q_B \\ p_A \notin F_A \end{matrix} \right\} \cup \left\{ \begin{pmatrix} p_A \\ p_B \\ 0 \end{pmatrix} \xrightarrow{a} \begin{pmatrix} q_A \\ q_B \\ 1 \end{pmatrix} \text{ si } \begin{matrix} p_A \xrightarrow{a} q_A \\ p_B \xrightarrow{a} q_B \\ p_A \in F_A \end{matrix} \right\} \cup S$ où S contient la partie symmétrique de ce qui a été écrit.

  Exo. w accepté par A et par $B \Leftrightarrow w$ accepté pour $A \times B \times \{0,1\}$
- $\bar{}$: $L = L^\omega(A)$, est-ce que $\bar{L} = L^\omega(B)$?

  Théorème de Büchi. L ω-régulier $\Rightarrow \bar{L}$ ω-régulier.

  La construction de Büchi est assez horrible, il existe une construction plus simple faite par Safra. C'est lourd et difficile. Non traîté. (des papiers en parlent).

Parenthèse. Automate de Büchi déterministe.

Déf. C'est exactement ce à quoi on s'attend : c'est un automate de Büchi avec $\left(p \xrightarrow{a} q_1 \wedge p \xrightarrow{a} q_2\right) \Rightarrow q_1 = q_2$.

Prop. Il existe L ω-régulier mais qui n'est reconnu par aucun automate déterministe de Büchi.

Preuve. $L = (a+b)^* b^\omega$. On prouve qu'il n'y a pas d'automate déterministe pour L en utilisant le lemme de pompage :

Soit A un tel automate, $b^\omega \in L$, donc b^ω est reconnu par A. Regardons l'unique calcul correspondant : $\exists n_1, \exists f_1 \in F : i \xrightarrow{b^{n_1}} f_1 \to \cdots$. Considérons $b^{n_1} a\, b^\omega$, ce mot est aussi dans L donc il est reconnu par A. $i \xrightarrow{b^{n_1}} f_1 \xrightarrow{a} q \xrightarrow{b^{n_2}} f_2 \to \cdots$. Considérons maintenant $b^{n_1} a\, b^{n_2} a\, b^\omega \in L$.

À la fin de cette construction, on a une séquence $n_1, n_2, ..., n_k$ et $f_1, f_2, ..., f_k$, pour tous k. Le mot construit $u = b^{n_1} a\, b^{n_2} a\, b^{n_3} a \cdots \notin L$ car il y a une inifinité de a. Pourtant le calcul sur ce mot est $i \xrightarrow{b^{n_1}} f_1 \xrightarrow{ab^{n_2}} f_2 \to \cdots$. On visite une infinité de fois F, or F est fini donc il existe un état $f \in F$ qui est visité une infinité de fois, donc u est accepté par A.

On en conclut qu'il n'existe pas d'automate déterministe de Büchi pour reconnaître L.

La déterminisation d'un automate de Büchi est donc impossible.

En TD, on verra un critère de déterminisabilité.

Retour à la preuve des propriétés.

- Tester $L^\omega = \varnothing$: on cherche $i \in I$, $f \in F$ tel que $L_{i,f} \neq \varnothing$, $L_{f,f} \neq \varnothing$ et $L_{f,f} \neq \{\varepsilon\}$.
- $L = \Sigma^\omega$: il faut passer au complémentaire...

- $L \subset M \Leftrightarrow L \cap \bar{M} = \varnothing$

Toutes ces propriétés sont décidables, parfois difficilement.

Théorème. L est ω-régulier $\Leftrightarrow$ L est définissable en MSO$(\Sigma, <)$. La seule différence est qu'on interprète ici les fomules avec des variables de position sur $\mathbb{N}$, et les propositionnelles sur $\mathcal{P}(\mathbb{N})$.

Preuve. Comme pour les langages fini. La seule difficulté est dans l'expression de la condition d'acceptation de Büchi :

$$\bigvee_{f \in F} \forall x, \exists y, (y > x \wedge X_f(y))$$

(un certain $f \in F$ est visité une infinité de fois).

Corollaire (Büchi). MSO$(\Sigma, <)$ interprété sur $\mathbb{N}$ est décidable.

II. Grammaires

Exemple de grammaire.

Phrase = Sujet Verbe Objet

Sujet = Nom | Pronom

Objet = Nom

Verbe = « voit » | « mange » | « déteste » | « aime »

Pronom = « il » | « elle »

Nom = « Marie » | « Pierre » | « un steak » | « un chat »

Exemple.

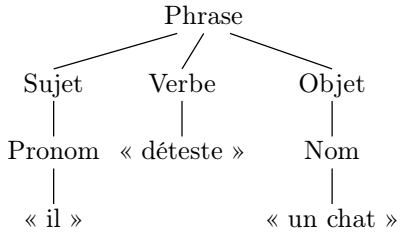


Figure 4. Arbre de dérivation pour la phrase « Il déteste un chat ».

Dérivation. Phrase → Sujet Verbe Objet → Nom Verbe Objet → Nom « voit » Objet → Nom « voit » Nom → « un steak voit » Nom → « un steak voit un steak ».

Ici, on va plutôt s'intéresser à des grammaires formelles.

Définition des grammaires

Déf (Chomski). Une grammaire est un tuple $\Gamma = (V, T, P, S)$, où :

1. V est un alphabet fini de variables (ou non terminaux)
2. T est un alphabet fini de terminaux
3. P est un ensemble de productions, chacune de la forme $\alpha \to \beta$, avec $\alpha, \beta \in (V \cup T)^*$

4. $S \in V$ est un symbole pour la variable initiale (axiome)

Notations. Les variables sont notées A, B, C, ... ; les terminaux sont notés a, b, c, ... ; les productions sont notées $\alpha \to \beta$; les mots de $(V \cup T)^*$ seront notés $\alpha, \beta, \gamma, ...$; les mots de T^* seront notés $u, v, ...$

Exemple. Sur l'exemple du haut,

$$\begin{aligned} V &= \{\text{Sujet}, \text{Phrase}, \text{Objet}, \text{Verbe}, \text{Pronom}, \text{Nom}\} \\ T &= \{\text{« il »}, \text{« elle »}, \text{« un chat »}, \text{« déteste »}, \text{« voit »}, ...\} \\ S &= \text{Phrase} \\ P &= \begin{cases} \text{Phrase} \to \text{Sujet Verbe Objet} \\ \text{Sujet} \to \text{Nom} \\ \text{Sujet} \to \text{Pronom} \\ \text{Objet} \to \text{Nom} \\ ... \end{cases} \end{aligned}$$

Définition de $\Rightarrow$. Soient $\alpha, \beta, \gamma, \delta \in (T \cup V)^*$ des mots quelconques. Si $\alpha \to \beta \in P$, alors $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$.

Déf. Une *dérivation* est une suite $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ de réécritures.

Déf. α est engendré par la grammaire Γ si il existe une dérivation $S \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow a$, ce que l'on notera $S \overset{*}{\Longrightarrow} \alpha$.

Déf. Le langage *engendré* par Γ est l'ensemble $\left\{ w \in T^* : S \overset{*}{\Longrightarrow} w \right\}$, noté $L(\Gamma)$.

Exemple de grammaire.

$V = \{S\}$, $T = \{a, b\}$, $S = S$,

$$P = \begin{cases} S \to a\,S\,a \\ S \to b\,S\,b \\ S \to a \\ S \to b \\ S \to \varepsilon \end{cases}$$

Cette grammaire Γ_1 engendre les palindromes.

Par exemple : $S \to a\,S\,a \to a\,a\,S\,a\,a \to a\,a\,b\,S\,b\,a\,a \to a\,a\,b\,a\,b\,a\,a$.

Exemple.

$\Gamma_2 = \{S \to a\,S\,b \,|\, \varepsilon\}$

$L(\Gamma_2) = \{a^n b^n\}$

$$\Gamma_3 = \begin{cases} S \to a\,S\,b \,|\, \varepsilon \\ a\,b \to \text{ba} \\ b\,a \to a\,b \end{cases}$$

$L(\Gamma_3) = \{w : \#a(w) = \#b(w)\}$.

Hiérarchie de Chomski

- Type 0 : toutes les grammaires. Elles produisent la classe des langages récursivement énumérables (voir plus tard), et on ne les étudie pas en fait à partir des grammaires.

- Type 1 : grammaires avec des productions de la forme

$$\alpha\,A\,\beta \to \alpha\,\gamma\,\beta, \gamma \neq \varepsilon$$

(les terminaux ne peuvent pas évoluer et les non-terminaux évoluent un par un)

Ce sont les grammaires dites *sous contexte* (context-sensitive). On ne s'y intéresse pas énormément.

- Type 2 : grammaires *hors-contexte* (context-free) : toutes les productions de la forme

$$A \to \gamma$$

Classe très importante ; on va s'y intéresser dans les deux cours suivants.

- Type 3 : grammaires *linéaires* : toutes les productions linéaires à gauche, ie de la forme

$$A \to u\,B \text{ ou } A \to u$$

(ou toutes les productions linéaires à droite : $A \to B\,u$ et $A \to u$; attention : pas de mélange).

Il n'y a pas grand chose à dire sur cette classe : elle produit les langages réguliers !

Déf. On dit que L est un langage de type k si $L = L(\Gamma)$ où Γ est une grammaire de type k.

Exemple de type 3.

$$\begin{aligned} S &\to a\,Y \\ Y &\to b\,S \,|\, a\,Y \\ Y &\to \varepsilon \end{aligned}$$

On constate que cette grammaire est représentée par un automate fini...

Grammaires linéaires

Théorème. L est définissable par une grammaire linéaire $\Leftrightarrow$ L est un langage régulier.

Preuve.

Grammaire vers automate :

Soit $L = L(\Gamma)$, avec $\Gamma = (V, T, P, S)$, P linéaire à gauche.

On construit l'automate généralisé (on met des mots sur les flèches) $A = (Q, \Sigma, \Delta, I, F)$, où :

- $Q = V \cup \{f\}$
- $\Sigma = T$
- $I = \{S\}$
- $F = \{f\}$
- $\Delta = \left\{A \xrightarrow{u} B ; A \to u\,B \in P\right\} \cup \left\{A \xrightarrow{u} f ; A \to u \in P\right\}$

On construit Δ à partir de P.

Si $w \in L(\Gamma)$, alors $S \to u_1\,A_1 \to u_1\,u_2\,A_2 \to \cdots \to u_1\,u_2 \cdots A_k \to u_1\,u_2 \cdots u_{k+1} = w$. Par construction de l'automate, dans lequel on a donc $S \xrightarrow{u_1} A_1 \xrightarrow{u_2} \cdots \xrightarrow{u_{k+1}} f$, il est clair que $w \in L(A)$.

Pareil pour l'autre sens... on a bien $L(\Gamma) = L(A)$. Cet automate généralisé peut être traduit vers un automate normal très facilement en rajoutant tout plein d'états au milieu.

2013-10-31

Automate vers grammaire : soit $L = L(A)$ avec $A = (Q, \Sigma, \Delta, i, F)$ un automate ayant un seul état initial. On construit la grammaire Γ :

- $V = Q$
- $T = \Sigma$
- $S = i$
- $P = \{P \to a\,Q ; p \xrightarrow{a} q \in \Delta\} \cup \{P \to \varepsilon ; p \in F\}$

On a $L(\Gamma) = L(A)$ car pour chaque calcul accepteur $i \xrightarrow{a_1} q_1 \to \cdots \to q_n \in F$ correspond une dérivation $S \to a_1 Q_1 \to \cdots \to a_1 \cdots a_n Q_n \to a_1 \cdots a_n$, et vice-versa.

Exercice. Prouver la même chose pour les langages linéaires à droite.

Exemple. Cf (1) sur feuille annexe.

Remarque. Si on mélange linéaire à gauche et linéaire à droite, alors on n'obtient plus un langage régulier. En effet, soit la grammaire :

$$\begin{gathered} S \to a\,T \\ T \to S\,b \\ S \to \varepsilon \end{gathered}$$

Cette grammaire engendre le langage $\{a^n b^n ; n \in \mathbb{N}\}$ qui n'est pas régulier.

Grammaires et langages hors contexte (algébriques)

Déf. Dans une grammaire *hors contexte*, toutes les productions ont la forme :

$$A \in V \;\to\; \alpha \in (T \cup V)^*$$

Déf. Les langages hors contexte sont les langages $L(G)$ pour G grammaire hors contexte.

Exemple 1. Tout langage régulier est hors-contexte. En effet, les grammaires linéaires sont hors contexte.

Exemple 2. Le langage $\{a^n b^n ; n \in \mathbb{N}\}$ est hors-contexte, et peut être engendré par :

$$S \;\to\; a\,S\,b \,|\, \varepsilon$$

Exemple 3. Dans les langages de programmation, on définit une grande partie de la syntaxe par des langages hors-contexte par BNF (la forme de Backus-Naur). Exemple de règles en BNF :

$$\begin{aligned} S &:= a\,T \,|\, \varepsilon \\ T &:= T\,b \\ \text{Expr} &:= \text{Var} \,|\, \text{Const} \,|\, (\text{Expr}) \,|\, \text{Expr} + \text{Expr} \,|\, ... \end{aligned}$$

Exemple 4. Langage de Dyct :

$$S \;\to\; (S) \,|\, S\,S \,|\, \varepsilon$$

Ce langage donne l'ensemble des parenthésages corrects. On peut aussi faire :

$$S \;\to\; (S) \,|\, [S] \,|\, \{S\} \,|\, S\,S \,|\, \varepsilon$$

On reconnaît alors par exemple l'expression (()[][{()}]).

Remarque. En compilation on utilise la technologie hors-contexte.

Dérivations et arbres

Soit G une grammaire hors contexte.

$w \in L(G) \Leftrightarrow w$ a une dérivation $\Leftrightarrow w$ a une arbre de dérivation (appellé également arbre syntaxique)

Exemple. On a la grammaire :

$$\begin{aligned} S &\to S\,S \,|\, R \\ R &\to a\,R\,a \,|\, T \\ T &\to b\,T\,b \,|\, \varepsilon \end{aligned}$$

On a le mot $a\,b\,b\,a\,a\,a\,b\,b\,a\,a \in L$, dont l'arbre de dérivation est donné en (2).

Déf. *Arbre syntaxique :* La racine est S. Chaque noeud interne est étiqueté par un non-terminal ; chaque noeud interne et ses fils correspond à une production ; chaque feuille est étiquetée par un terminal ou par ε. Les feuilles prises de gauche à droite produisent le mot w.

En descendant, cet arbre explique la dérivation ammenant à un mot. En montant, on analyse un mot pour retrouver sa structure syntaxique.

Remarque. Un arbre syntaxique correspond à plusieurs dérivations (très similaires).

Exemple. $S \to S\,S \to R\,S \to a\,R\,a\,S \to a\,R\,a\,R \to a\,R\,a\,a\,R\,a \to a\,T\,a\,a\,R\,a \to a\,T\,a\,a\,a\,R\,a\,a \to \cdots$, ou bien : $S \to S\,S \to S\,R \to R\,R \to R\,a\,R\,a \to a\,R\,a\,a\,R\,a \to \cdots$.

L'arbre syntaxique est une forme canonique de dérivation.

Définition. *Dérivation à gauche :* on se permet seulement de dériver à chaque étape le non-terminal le plus à gauche. Cela donne une seconde forme canonique de dérivation. On a de même la *dérivation à droite*.

Prop. Pour une grammaire hors-contexte G,

$$\begin{aligned} w \in L(G) &\Leftrightarrow w \text{ a une dérivation} \\ &\Leftrightarrow w \text{ a un arbre syntaxique (arbre de dérivation)} \\ &\Leftrightarrow w \text{ a une dérivation à gauche} \\ &\Leftrightarrow w \text{ a une dérivation à droite} \end{aligned}$$

Déf. G est non-ambigüe si $\forall w \in L(G)$, w a un seul arbre de dérivation (ou une seule dérivation à gauche, etc.)

Exemple 1. Exemple de grammaire non ambigüe pour générer $\{a^n b^n \,;\, n \in \mathbb{N}\}$:

$$S \to a\,S\,b \,|\, \varepsilon$$

Exemple 2. Exemple de grammaire ambigüe pour la même chose :

$$\begin{aligned} S &\to a\,S\,b \,|\, \varepsilon \\ S &\to T \\ T &\to a\,T\,b \,|\, \varepsilon \end{aligned}$$

Le mot $a\,a\,b\,b$ peut être généré de plusieurs façons :

$$\begin{gathered} S \to a\,S\,b \to a\,a\,S\,b\,b \to a\,a\,b\,b \\ S \to a\,S\,b \to a\,T\,b \to a\,a\,T\,b\,b \to a\,a\,b\,b \end{gathered}$$

Pour $L = \{a^n b^n\}$, on a une grammaire ambigüe et une grammaire non-ambigüe.

Exemple 3.

$$S \ \to \ x \,|\, y \,|\, z \,|\, S + S \,|\, S \times S$$

Le langage généré est l'ensemble des expressions non parenthésées avec $+$ et $\times$ sur x, y, z. Prenons par exemple l'expression $x + y \times z$, arbre donné en (3).

Définition. L est intrinséquement ambigü si toute grammaire G engendrant L est ambigüe.

Conjecture. Le langage de l'exemple 3 est intrinséquement ambigü.

Exemple. $L = \{a^n\, b^n\, c^m ; n, m \in \mathbb{N}\} \cup \{a^m\, b^n\, c^n ; n, m \in \mathbb{N}\}$. Il peut être défini par la grammaire :

$$\begin{aligned} S &\to RT \,|\, UV \\ R &\to a\, R\, b \,|\, \varepsilon \\ T &\to cT \,|\, \varepsilon \\ U &\to aU \,|\, \varepsilon \\ V &\to bVc \,|\, \varepsilon \end{aligned}$$

Le mot $a^n\, b^n\, c^n$ a deux dérivations. Ce langage est intrinséquement ambigü.

Analyse syntaxique

Problème. On a une grammaire G et un mot w ; est-ce que $w \in L(G)$? Si oui, donner son arbre syntaxique. (c'est le problème d'appartenance à un langage, ou d'analyse syntaxique).

On trouvera un algorithme efficace :

1. On met la grammaire en forme normale de Chomsky
2. On applique la programmation dynamique

Déf. Une grammaire en CNF (forme normale de Chomsky) a des productions du type :

$$\begin{aligned} A &\to BC \\ A &\to a \end{aligned}$$

Et éventuellement, une règle en plus pour gérer ε :

$$S \ \to \ \varepsilon$$

Dans ce cas, S n'aparait pas à droite des productions.

Théorème. Pour toute grammaire hors-contexte G, il existe une grammaire en CNF G' telle que $L(G) = L(G')$.

Preuve. On va transformer G en G' par plusieurs nettoyages :

- TERM : on se débarasse des terminaux à droite, ie des règles de la forme :
  $A \to \gamma$, avec $|\gamma| \geqslant 2$ et γ contient des terminaux
- BIN : on se débarasse de $A \to \gamma$ avec $|\gamma| \geqslant 3$
- DEL : on se débarasse de $A \to \varepsilon$
- UNIT : on se débarasse de $A \to B$

Étape TERM.

Pour chaque terminal a, on introduit X_a un non-terminal qui le représente. On rajoute les productions :

$$X_a \to a$$

et dans toutes les règles $A \to \gamma$, avec $|\gamma| \geqslant 2$, on remplace les a par des X_a.

Exemple.

Dans G	Dans G'
$A \to a\,B\,b$	$X_a \to a$ $X_b \to b$ $A \to X_a\,B\,X_b$

Tableau 6.

Étape BIN.

Pour chaque règle de la forme $A \to B_1 B_2 \cdots B_n$, on la remplace par :

$$\begin{aligned} A &\to B_1 A_1 \\ A_1 &\to B_2 A_2 \\ &\vdots \\ A_{n-2} &\to B_{n-2} A_n \\ A_{n-1} &\to B_{n-1} B_n \end{aligned}$$

(on a rajouté $n-1$ non-terminaux intermédiaires).

Étape DEL.

Déf. On dit que A est mortelle si $A \overset{*}{\Longrightarrow} \varepsilon$.

Si S est mortel, on ajoute un nouvel état initial S_0, et on rajoute

$$\begin{aligned} S_0 &\to S \\ S_0 &\to \varepsilon \end{aligned}$$

1. Calculer l'ensemble M des lettres mortelles :

   Soit M_k l'ensemble des lettres qui meurent en k étapes. On les calcules avec :

   $$\begin{aligned} M_1 &= \{A : A \to \varepsilon \in P\} \\ M_{k+1} &= M_k \cup \{A : A \to B\,C \text{ avec } B, C \in M_k\} \end{aligned}$$

   On itère jusqu'au point fixe, qui est maintenant M.

2. On se débarasse de $A \to \varepsilon$, mais pour tout $B \in M$, on effectue les remplacements :

Dans G	Dans G'
$A \to B\,C$	$A \to B\,C$ $A \to C$
$A \to C\,B$	$A \to C\,B$ $A \to C$
$A \to B\,B$	$A \to B\,B \mid B$

Tableau 7.

Exo. Compléter la preuve d'équivalence.

Étape UNIT.

Préparation. On considère les règles unitaires $A \to B$ comme un graphe orienté. On déduit un ordre $A \geqslant B$ si $A \overset{*}{\Longrightarrow} B$. Grâce aux nettoyages déjà fait, la seule possibilité de faire $A \overset{*}{\Longrightarrow} B$ est par une séquence de dérivations selon des règles unitaires.

Nettoyage. On efface les règles unitaires $A \to B$. On remplace :

Dans G	Dans G'
$A \to BC$	$A_0 \to BC$ pour $A_0 \geqslant A$
$A \to a$	$A_0 \to a$ pour $A_0 \geqslant A$

Tableau 8.

Exo. Raisonnement d'équivalence à compléter.

On a fini notre transformation de grammaire context-free vers CNF.

Algorithme CYK (Cocke, Younger, Kasama). Algorithme pour tester $w \in L(G)$, où $w \in T^*$ et G est une grammaire en CNF.

On va calculer la matrice d'ensembles suivante, par récurrence sur $j - i$.

$$P[i,j] \;=\; \left\{ A : A \overset{*}{\Longrightarrow} w_i\, w_{i+1} \cdots w_j \right\}$$

Pour $j - i = 0$, ie $i = j$, on a la formule :

$$P[i,i] \;=\; \{A : A \to w_i\}$$

Puis la récurrence est la suivante :

$$P[i,j] \;=\; \left\{ A : \exists i \leqslant k < j : \begin{array}{r} B \in P[i,k] \\ \wedge\; C \in P[k+1,j] \\ \wedge\; A \to BC \text{ est une production} \end{array} \right\} (*)$$

Algorithme 1

Base. Pour tout i, $P[i,i] \leftarrow \{A : A \to w_i\}$
Récurrence.
  Pour Δ de 1 à $n-1$
    Pour tout i
      $P[i, i+\Delta] = (*)$
Si $s \in P[1,n]$
  Renvoyer OUI
Renvoyer NON.

Complexité. $O(n^3)$ en la longueur du mot, et une dépendance linéaire en la taille de la grammaire.

Lemme de pompage

Théorème. Soit L un langage hors-contexte. Alors il existe $n \in \mathbb{N}$ tel que $\forall w \in L$, si $|w| > n$ alors il existe une factorisation $w = u\,x\,v\,y\,z$ telle que $\forall k, u\,x^k\,v\,y^k\,z \in L$, avec $|x\,y| > 0$ et $|x\,v\,y| \leqslant n$.

Preuve. Soit $L = L(G)$, avec G en CNF. Soit $m = |V|$ et $n = 2^m$.

Soit $w \in L$ et $|w| > n$. On considère son arbre de dérivation : c'est un arbre binaire et il a $|w|$ feuilles.

Si toutes les branches étaient plus courtes ou égales à m en hauteur, il y aurait $\leqslant n = 2^m$ feuilles. Mais ici, on a strictement plus de feuilles, donc il existe une branche de hauteur $> m$.

Il existe donc le long de cette branche un non-terminal qui se répète. On peut donc pomper, en duplicant le sous-arbre issu de la dernière instance avant la plus basse à la place du sous-arbre issu de l'instance la plus basse, et ce répétitivement (il faut garantir que le sous-arbre pompé vérifie $|x\,v\,y| \leqslant n$).

En terme de dérivations : $S \overset{*}{\Longrightarrow} u\,A\,z \overset{*}{\Longrightarrow} u\,x\,A\,y\,z \overset{*}{\Longrightarrow} u\,x^2\,A\,y^2\,z \overset{*}{\Longrightarrow} u\,x^k A\,y^k z$.

Exo. Vérifier que $|x\,v\,y| \leqslant n$ sous la condition donnée.

Prop. $L = \{a^m b^m c^m \,;\, m \in \mathbb{N}\}$ n'est pas un langage hors contexte.

Preuve. Supposons que L est hors-contexte. Par lemme de pompage, il existe n tel que ... plein de choses ...

Prenons $w = a^n\,b^n\,c^n$, $w \in L$. Par lemme de pompage, il existe une factorisation gonflable :

$$a^n\,b^n\,c^n = u\,x\,v\,y\,z, |x\,v\,y| \leqslant n$$

Donc $x\,v\,y$ ne contient soit aucun c, soit aucun a. Par lemme de pompage, $w' = u\,x^2\,v\,y^2\,z \in L$, or il y a trop soit de c soit de a soit de b soit d'autre chose dans ce nouveau mot, donc c'est un fail lamentable et la propriété est démontrée.

Propriétés de cloture

Théorème. Les langages hors contexte...

1. sont clos par $\cup, \cdot, ^*, h, h^{-1}$

   ne sont pas clos par $\cap, \,^-$

2. soit $L = L(G_1)$ et $M = L(G_2)$. Alors :

   « $L = \varnothing$? » est décidable, « $w \in L$? » est décidable

   « $L \cap M = \varnothing$? » est indécidable (voir dans trois semaines).

Déf. On appelle substitution une application $\sigma : \Sigma \to \mathcal{P}(\Gamma^*)$.

Exemple.

$$\begin{aligned} a &\to \{00\} \\ b &\to 1^* \end{aligned}$$

Déf. On pose mainteant $\sigma(w) = \sigma(w_1)\,\sigma(w_2)\cdots\sigma(w_n)$, et on pose :

$$\sigma(L) = \bigcup_{w \in L} \sigma(w)$$

Exemple. Avec la substitution définie ci-dessus, $\sigma((a\,b)^*) = (001^*)^*$

Déf. Une substitution *hors contexte* est une substitution telle que tous les $\sigma(a)$ pour $a \in \Sigma$ sont des langages hors-contexte.

Lemme. Soit L un langage hors-contexte, soit σ une substitution hors-contexte. Alors $\sigma(L)$ est un langage hors-contexte.

Preuve. $L = L(V, \Sigma, P, S)$; $\Sigma = \{a_1, ..., a_k\}$.

$\forall i \in \{1, ..., k\}, \sigma(a_i) = L(V_i, \Gamma, P_i, S_i)$, où tous les V_i sont disjoints deux à deux et avec V.

On construit la grammaire G' où l'on réunit les V, V_i ainsi que les P, P_i, où l'on remplace les productions de la forme $A \to a_i$ par $A \to S_i$ ($S' = S$).

Preuve du théorème.

- Si L_1, L_2 sont hors-contexte...

  Soit $M = \{1, 2\}$ qui est un langage hors contexte, $\sigma : 1 \mapsto L_1, 2 \mapsto L_2$. $L_1 \cup L_2 = \sigma(M)$ qui est hors-contexte par substitution.

  Soit $M = \{1 \cdot 2\}$, qui est un langage hors contexte, alors $\sigma(M) = L_1 \cdot L_2$ est aussi hors contexte.

  Soit $M = 1^*$, alors $\sigma(M) = L_1^*$ qui est donc hors-contexte.

  Soit M hors contexte, $\sigma = h$ un morphisme (c'est un cas particulier de substitution hors-contexte, puisqu'un singleton est hors-contexte : $\sigma(a) = \{h(a)\}$). On a alors $\sigma(M) = h(M)$, qui est donc hors-contexte par le lemme.

- Soit $L_1 = \{a^n b^n c^m\}$ et $L_2 = \{a^m b^n c^n\}$, alors $L_1 \cap L_2 = \{a^n b^n c^n\}$, ce dernier n'étant pas hors-contexte alors que les deux premiers le sont.

  Si cloture par union et complément, on aurait cloture par intersection. On n'adonc pas cloture par complémet.

`2013-11-07`

Question. Pour une grammaire hors contexte : la question « $L = \varnothing$ » est décidable : donner un algorithme.

Déf. Un non-terminal A est productif si $\exists w \in T^* : A \overset{*}{\Longrightarrow} w$.

Algorithme. On pose $P_1 = \{A : A \to w \text{ avec } w \in T^*\}$, puis on calcule

$$P_{k+1} = P_k \cup \{A : A \to \gamma \text{ avec } \gamma \in (P_k \cup T)^*\}$$

L'ensemble des non-terminaux productifs est le point fixe P_∞ de cette suite. Le langage $L(G)$ est non-vide ssi $S \in P_\infty$.

Reconnaissance d'une grammaire

On a une grammaire G, on veut trouver un « automate » qui reconnait $L(G)$.

Exemple 1. $\{a^n\, b^n\}$ n'est pas un langage régulier. C'est un langage hors-contexte. On peut utiliser un automate du type suivant (1) :

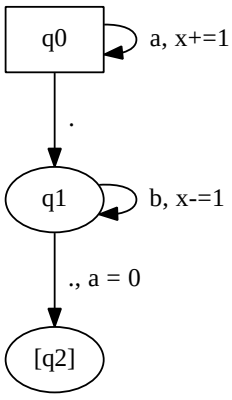


Figure 5.

Exemple 2. {palindromes sur $\{a, b\}$}, se reconaissent avec une pile (cf (2)).

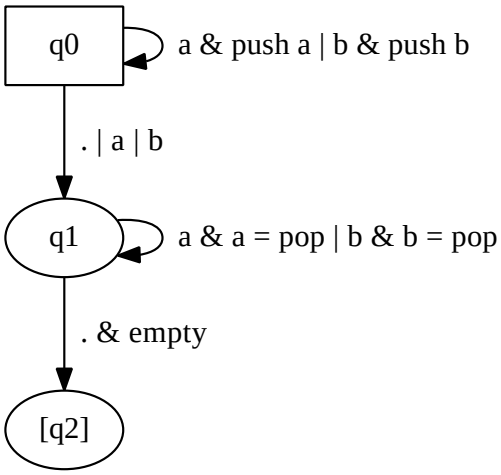


Figure 6.

Définition : automate à pile (pushdown automaton).

$$A = (Q, \Sigma, \Gamma, I, \Delta, Z_0, F)$$

- Q : états (ensemble fini)
- Σ : alphabet (fini)
- Γ : alphabet de pile (fini)
- $I \subset Q$: états initiaux
- Δ : ensemble fini des transitions (voir ci-dessous)
- $Z_0 \in \Gamma$: symboles de pile vide
- $F \subset Q$: états finals

Les transitions sont de la forme :

$$q, X \xrightarrow{a \text{ ou } \varepsilon} p, u \text{ avec } x \in \Gamma, u \in \Gamma^*$$

Cette transition peut être empruntée si on est à l'état q et que le symbole du haut de la pile est X. Si on la suit, on dépile X puis on empile tout le mot u, puis on va sur l'état p :

$$\left(\begin{array}{|c} q \\ \hline X v \end{array} \right) \xrightarrow{a} \left(\begin{array}{|c} p \\ \hline u, v \end{array} \right)$$

Une configuration de l'automate est un couple $(q, u) \in Q \times \Gamma^*$ (un état et une pile).

U calcul sur un mot $w \in \Sigma^*$ est une chaîne de transitions, ettiquetées par w.

Un calcul accepteur I sur I commence dans $\left(\begin{array}{|c} i \\ \hline Z_0 \end{array} \right)$ avec $i \in I$, et termine dans $\left(\begin{array}{|c} f \\ \hline u \end{array} \right)$ avec $f \in F$ et u quelconque (calcul accepteur par état final).

Un calcul accepteur II commence dans $\left(\begin{array}{|c|}\hline i \\ \hline Z_0 \\ \hline \end{array}\right)$ avec $i \in I$, et termine dans $\left(\begin{array}{|c|}\hline q \\ \hline \varepsilon \\ \hline \end{array}\right)$ avec q un état quelconque (calcul accepteur par pile vide).

Automate reconaissant les palindromes. (3)

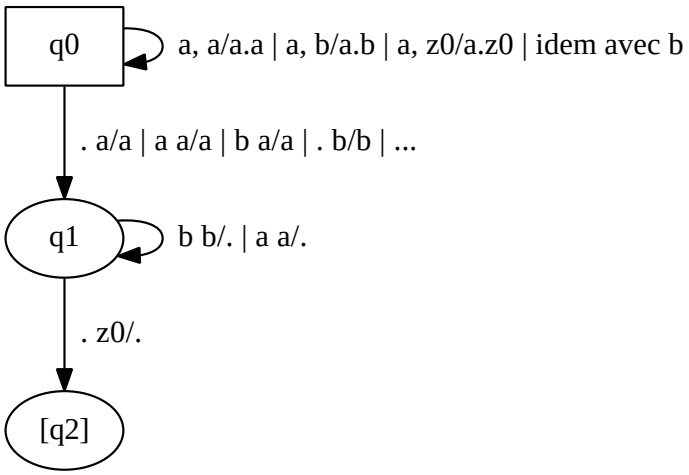


Figure 7.

Notation. $L(A) = \{w$ acceptés par état final$\}$ et $N(A) = \{w$ acceptés par pile vide$\}$

Théorème. L'acceptation par pile vide et par état final donne la même classe de langages, ie :

1. $\forall A, \exists A' : L(A) = N(A')$
2. $\forall A, \exists A' : N(A) = L(A')$

Preuve. On a A qui reconnait $L(A)$ par état final. On rajoute à l'automate un état poubelle, tous les états finals y vont, l'état boucle sur lui-même en poppant toutes les lettres. Mais il ne faut pas que la pile se vide avant, lors de l'exécution d'un mot à ne pas reconnaitre. On rajoute donc une lettre qui n'apparaît pas dans A pour la pile, que l'on empile au début avant d'aller dans chaque état initial (il y a donc besoin d'un nouvel état initial). Cette lettre ne sera jamais enlevée dans l'automate A, elle ne peut être dépilée que par l'état final rajouté. On a donc créé notre automate A'.

Maintenant, on a B qui reconnait $N(B)$. C'est à peu près aussi simple : on rajoute une transition Z_0/Z_0 sur tous les états vers un nouvel état qui sera le seul état final.

Théorème important. L est hors contexte $\Leftrightarrow$ L est reconaissable par un automate à pile.

Preuve sens grammaire $\rightarrow$ automate à pile.

On prend $L = L(G)$, avec G sous forme normale de Chomsky (CNF) (on suppose $\varepsilon \notin L$, il est facile à rajouter), et on construit l'automate A avec (4) :

- $Q = \{q\}$: il y a un seul état
- $\Sigma = T$ et $\Gamma = V$
- $Z_0 = S$

- Pour chaque production de type $A \to a$, on crée dans l'automate la transition

$$q, A \xrightarrow{a} q, \varepsilon$$

(on dépile A en lisant a)

- Pour chaque production de type $A \to BC$, on crée dans l'automate la transition

$$q, A \xrightarrow{\varepsilon} q, BC$$

(on dépile A que l'on remplace par BC, sans rien lire).

On veut montrer que $L(G) = N(A)$. Idée : A fait les dérivations gauche de G.

Remarque : une dérivation gauche est du type $S \to u_1 W_1 \to u_2 W_2 \to \cdots \to u_n W_n$, avec $u_i \in T^*$ et $W_i \in V^*$. Les mots u_i correspondent à ce qui est lu à chaque étape et W_i au contenu de la pile à cette étape.

Lemme. $A \overset{k}{\Longrightarrow} u\,W$ dans $G \Leftrightarrow \left(\begin{array}{|c|}\hline q \\ \hline A \\ \hline \end{array}\right) \xrightarrow{u} \left(\begin{array}{|c|}\hline q \\ \hline W \\ \hline \end{array}\right)$ dans A.

Le résultat suit du lemme :

$$\begin{array}{rcl} w \in L(G) & \Leftrightarrow & S \overset{*}{\Longrightarrow} w \\ & \Leftrightarrow & \left(\begin{array}{|c|}\hline q \\ \hline S \\ \hline \end{array}\right) \xrightarrow{w} \left(\begin{array}{|c|}\hline q \\ \hline \varepsilon \\ \hline \end{array}\right) \\ & \Leftrightarrow & w \in N(A) \end{array}$$

Démonstration du lemme. $\Rightarrow$ récurrence sur la longueur de w. $\Leftarrow$ récurrence sur la longueur du calcul. Preuve complète omise (en exo).

Exemple.

$$\begin{array}{rcl} S & \to & A\,A \\ A & \to & S\,B \\ B & \to & b \\ A & \to & a \end{array}$$

On crée les transitions :

$$\begin{array}{lcl} q & \xrightarrow{\text{pop } S, \text{ push } AA} & q \\ q & \xrightarrow{\text{pop } A, \text{ push } SB} & q \\ q & \xrightarrow{a, \text{ pop } A} & q \\ q & \xrightarrow{b, \text{ pop } B} & q \end{array}$$

Considérons par exemple la dérivation :

$$S \to A\,A \to S\,B\,A \to A\,A\,B\,A \to a\,A\,B\,A \to a\,a\,B\,A \to a\,a\,b\,A \to a\,a\,b\,a$$

Cette dérivation correspond au calcul suivant :

$$\begin{array}{|c|}\hline q \\ \hline S \\ \hline \end{array} \xrightarrow{\varepsilon} \begin{array}{|c|}\hline q \\ \hline A\,A \\ \hline \end{array} \xrightarrow{\varepsilon} \begin{array}{|c|}\hline q \\ \hline S\,B \\ \hline \end{array} \xrightarrow{\varepsilon} \begin{array}{|c|}\hline q \\ \hline A\,A\,B\,A \\ \hline \end{array} \xrightarrow{a} \begin{array}{|c|}\hline q \\ \hline A\,B\,A \\ \hline \end{array} \xrightarrow{a} \begin{array}{|c|}\hline q \\ \hline B\,A \\ \hline \end{array} \xrightarrow{b} \begin{array}{|c|}\hline q \\ \hline A \\ \hline \end{array} \xrightarrow{a} \begin{array}{|c|}\hline q \\ \hline \varepsilon \\ \hline \end{array}$$

Preuve du sens automate à pile → grammaire.

Les non-terminaux seront les triplets :

$$V = \{S\} \cup \{[p, X, q]\,;\, p, q \in Q, X \in \Gamma\}$$

On cherche les productions telles qu'on pourra vérifier le lemme suivant :

Lemme. $[p, X, q] \overset{*}{\Longrightarrow} w$ ssi $\left(\begin{array}{|c|}\hline p \\ \hline X \\ \hline \end{array}\right) \overset{w}{\longrightarrow} \left(\begin{array}{|c|}\hline q \\ \hline \varepsilon \\ \hline \end{array}\right)$.

($[p, X, q] \overset{*}{\Longrightarrow}$ mots qui permettent de passer de p à q en dépilant un X).

Dans notre grammaire, on met les productions :

$$S \;\rightarrow\; [i, Z_0, q] \text{ pour tout } i \in I$$

Alors :

$$\begin{array}{rcl} w \in L(G) & \Leftrightarrow & S \overset{*}{\Longrightarrow} w \\ & \Leftrightarrow & S \rightarrow [i, Z_0, q] \overset{*}{\Longrightarrow} w \\ & \Leftrightarrow & \left(\begin{array}{|c|}\hline i \\ \hline Z_0 \\ \hline \end{array}\right) \overset{w}{\longrightarrow} \left(\begin{array}{|c|}\hline q \\ \hline \varepsilon \\ \hline \end{array}\right) \\ & \Leftrightarrow & w \in N(A) \end{array}$$

Pour chaque transition $p, X \overset{a}{\longrightarrow} r_0, Y_1 Y_2 \cdots Y_k$ de Δ, on définit les productions :

$$[p, X, q] \;\rightarrow\; a\,[r_0, Y_1, r_1][r_1, Y_2, r_2] \cdots [r_{k-1}, Y_k, q]$$

pour tous les $r_1, ..., r_{k-1}, q \in Q$.

Preuve ⇒ du lemme.

Faisons une récurrence sur la taille de la dérivation (cette taille est toujours $\geqslant 1$).

- **Dérivation de taille 1.** La seule dérivation de taille 1 a la forme :

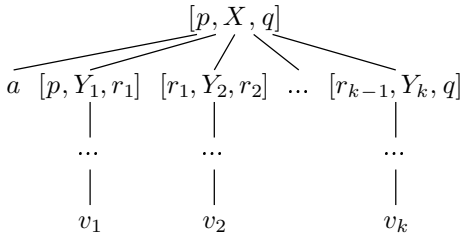
  $$[p, X, q] \rightarrow a$$

  où

  $$p, X \overset{a}{\longrightarrow} q \;\in\; \Delta$$

  Donc $\left(\begin{array}{|c|}\hline p \\ \hline X \\ \hline \end{array}\right) \overset{a}{\longrightarrow} \left(\begin{array}{|c|}\hline q \\ \hline \varepsilon \\ \hline \end{array}\right)$ est possible dans l'automate A.

- **Dérivation de taille >1.** Cet arbre a forcément la forme :



  On a $w = a\, v_1\, v_2 \cdots v_k$. Par construction de G,

  $$\left(\begin{array}{|c|}\hline p \\ \hline X \\ \hline \end{array}\right) \overset{a}{\longrightarrow} r_0, Y_1 Y_2 \cdots Y_k$$

Par hypothèse de récurrence :

$$\begin{array}{rcl}
\left(\begin{array}{c} r_0 \\ \hline Y_1 \end{array} \right) & \xrightarrow{v_1} & r_1 \\
\left(\begin{array}{c} r_1 \\ \hline Y_2 \end{array} \right) & \xrightarrow{v_2} & r_2 \\
\vdots & & \\
\left(\begin{array}{c} r_{k-1} \\ \hline Y_k \end{array} \right) & \xrightarrow{v_k} & q
\end{array}$$

On assemble le tout, ce qui est possible car les transitions sont indifférentes à ce qui se passe en profondeur de la pile :

$$\left(\begin{array}{c} p \\ \hline X \end{array} \right) \xrightarrow{a} \left(\begin{array}{c} r_0 \\ \hline Y_1\, Y_2 \cdots Y_k \end{array} \right) \xrightarrow{v_1} \left(\begin{array}{c} r_1 \\ \hline Y_2 \cdots Y_k \end{array} \right) \xrightarrow{v_2} \cdots \xrightarrow{v_k} \left(\begin{array}{c} q \\ \hline \varepsilon \end{array} \right)$$

- Ce résultat peut effectivement s'écrire :

$$\left(\begin{array}{c} p \\ \hline X \end{array} \right) \xrightarrow{w} \left(\begin{array}{c} q \\ \hline \varepsilon \end{array} \right)$$

□

Preuve ⇐ du lemme. On a un calcul. On considère sa première transition :

$$\left(\begin{array}{c} p \\ \hline X \end{array} \right) \xrightarrow{a} \left(\begin{array}{c} r_0 \\ \hline Y_1\, Y_2 \cdots Y_k \end{array} \right) \xrightarrow{v_1} \left(\begin{array}{c} r_1 \\ \hline Y_2 \cdots Y_k \end{array} \right) \to \cdots \to \left(\begin{array}{c} q \\ \hline \varepsilon \end{array} \right)$$

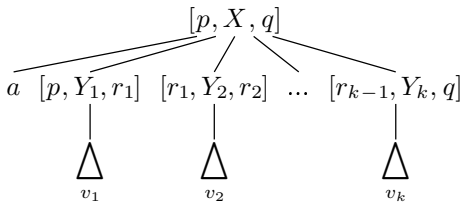
On s'intéresse au moment dans ce calcul (sur le schéma : après avoir lu v_1) où la longueur de la pile devient $k-1$ pour la première fois. À ce moment la pile contiendra donc $Y_2 \cdots Y_k$.

Il y aura ensuite un moment où pour la première fois la pile sera $Y_3 \cdots Y_k$, on sera sur l'état r_2, etc.

On fait maintenant une induction sur la longueur du calcul : par hypothèse d'induction :

$$\begin{array}{rcl}
[r_0, Y_1, r_1] & \overset{*}{\Longrightarrow} & v_1 \\
[r_1, Y_2, r_2] & \overset{*}{\Longrightarrow} & v_2 \\
\vdots & & \\
[r_{k-1}, Y_k, q] & \overset{*}{\Longrightarrow} & v_k
\end{array}$$

On assemble ces dérivations, on obtient $[p, X, q] \to a\,[r_0, Y_1, r_1] \cdots [r_{k-1}, Y_k, q]$ et l'arbre :



Donc $[p, X, q] \to a\, v_1\, v_2 \cdots v_k = w$. □

On a démontré que les grammaires hors contexte ont le même pouvoir expressif que les automates à piles. Utilités :

1. Les langages hors contexte sont clos par $\cup, ^*, \cdot, h$ (ce qu'on a déjà fait grâce aux substitutions)

2. **Prop.** Soit L hors contexte et M régulier, alors $L \cap M$ est hors-contexte.

   Preuve. On utilise un automate produit : $L = L(A)$ avec automate à pile et $M = L(B)$ avec B automate fini. Alors $A \times B$ est un automate à pile reconnaissant $L \cap M$.

Sous-classe importante : automates à pile déterministes

On ne peut avoir qu'une transition du type :

$$\left(\begin{array}{c} p \\ \hline X \end{array} \right) \xrightarrow{a} \ \ldots$$

On interdit également les ε-transitions.

Ces automates sont très utiles pour la compilation.

III. Calculabilité

Sur la page du cours : vieux poly de cours de maîtrise (42 pages).

On cherche à distinguer les problèmes décidable, indécidabe, semi-décidable.

Problèmes. Est-ce qu'il existe un algorithme qui décide :

1. Pour $x \in \mathbb{N}^*$, est-ce que le programme suivant termine ?

   Algorithme 2

   ```
   CAL(x) :
       tant que x > 1
           si x est pair
               x ← x/2
           sinon
               x ← 3x + 1
   ```

   Premier algorithme : **print** YES. On ne sait pas s'il est correct.

   Second semi-algorithme : calculer les valeurs, lorsque le programme s'arrête **print** YES. Cet algorithme est correct, mais on ne sait pas s'il termine toujours. Propriété de cet algo : si la réponse est OUI, il le trouvera. Si la réponse est NON, il ne s'arrêtera pas.

   Troisième algorithme : inconnu...

   Ce problème est semi-décidable, et on ne sait pas s'il est décidable.

2. On a une fonction $f : x \mapsto f(x)$ écrite en C. S'arrête-t-elle pour un x donné ?

   Semi-algorithme : calculer et attendre.

   Il n'existe pas d'algorithme pour ce problème.

   C'est un problème semi-décidable et indécidable.

3. On a une fonction $f : x \mapsto f(x)$ en C. Est-ce que $\forall x, f(x) = x!$?

   Pas d'algorithme, pas de semi-algorithme.

   C'est un problème indécidable non semi-décidable.

4. Tous les problèmes vus en cours d'algorithmique

5. On se donne une formule propositionnelle. Est-elle satisfiable ?

   Algorithme : faire une table de vérité. S'il n'y a que des 0, la réponse est NO, sinon c'est YES.

6. On se donne une formule close de $\mathrm{FO}(+, \times, 0, 1, =)$ (arithmétique). Est-elle vraie sur $\mathbb{N}$?

   Par exemple : « $\forall x, \exists y : (y \times y = x)$ » (cette proposition est fausse, pas besoin d'algorithme).

   Ce problème est indécidable et non semi-décidable.

7. Pareil mais sans $\times$. Ce problème est décidable (Presburger).

8. (algèbre) On a une équation polynomiale avec plusieurs variables et à coefficients entiers. On étudie l'équation :

$$P(\bar{X}) = 0$$

   a. A-t-elle une solution dans $\mathbb{C}$?

      Décidable et facile (pour une seule variable, c'est un théorème)

   b. A-t-elle une solution dans $\mathbb{R}$?

      Décidable, très difficile (exemple : peut-on avoir 13 boules de rayon 1 qui touchent une boule centrale de rayon 1 ? C'est une équation à 39 variables... théoriquement décidable mais prend un temps insupportable)

   c. A-t-elle une solution dans $\mathbb{Z}$?

      Semi-décidable (on essaye tout) et indécidable (à une seule variable, c'est décidable, mais pas à plusieurs).

Définition informelle. Un *problème* est donné par $B \in \mathcal{U}$ ($\mathcal{U}$ est l'univers). On pose la question : étant donné $x \in \mathcal{U}$, est-ce que $x \in B$?

Un problème est *décidable* s'il existe un algorithme A qui termine toujours et tel que $\forall x \in B$, $A(x)$ imprime OUI et $\forall x \in \mathcal{U} \setminus B$, $A(x)$ imprime NON.

Un algorithme est *indécidable* s'il n'est pas décidable.

Un algorithme est *semi-décidable* s'il existe un algorithme A tel que $\forall x \in B$, alors $A(x)$ termine et affiche OUI, et $\forall x \in \mathcal{U} \setminus B$, $A(x)$ imprime NON ou ne termine pas.

Mais il manque une définition formelle d'un algorithme !

Remarque. Pour démontrer que B est décidable ou semi-décidable, on fournit un algo qui remplit les critères ci-dessus (pas besoin de théorie). Pour démontrer que B est indécidable, il faut de la théorie car on a besoin de raisonner sur tous les algorithmes imaginables.

2013-11-14

III.1. Définitions : fonctions calculables

III.1.a. Fonctions récursives primitives

S'il y a un algo qui répond à « $x \in B$? », on dit que le problème est décidable.

Dans un premier temps on s'intéresse à des problèmes sur l'univers $\mathcal{U} = \mathbb{N}^k$.

Déf. On appelle *fonction caractéristique* de B la fonction $\chi_B : \mathbb{N}^k \to \mathbb{N}, x \mapsto \begin{cases} 1 & \text{si } x \in B \\ 0 & \text{sinon} \end{cases}$.

On dira plus tard : B est décidable $\Leftrightarrow$ χ_B est calculable. On s'intéresse donc plutôt aux problèmes des fonctions calculables.

Principe. On définira des langages de programmation et des modèles de calcul. Toute fonction que l'on peut programmer est calculable.

Déf. (Rosza Peter). On appelle classe des *fonctions récursives primitives*, et on note RP, la classe de fonctions $\mathbb{N}^k \to \mathbb{N}$ définie par induction comme suit :

- Fonctions de base :

  $0 : \mathbb{N}^0 \to \mathbb{N}$, $0() = 0$

  $\sigma : \mathbb{N} \to \mathbb{N}$, $\sigma(x) = x+1$

  $\pi_i^k : \mathbb{N}^k \to \mathbb{N}$, $\pi_i^k(x_1, ..., x_k) = x_i$

- Constructeur Comp :

  Si $f, g_1, ..., g_k \in \mathrm{RP}$ ayant les bonnes arités, alors $\mathrm{Comp}(f, g_1, ..., g_k) \in \mathrm{RP}$, avec :

$$\mathrm{Comp}(f, g_1, ..., g_k)(\bar{x}) = f(g_1(\bar{x}), ..., g_k(\bar{x}))$$

- Constructeur RecPri :

  Si $f, g \in \mathrm{RP}$ ayant les bonnes arités, alors $\mathrm{RecPri}(f, g) = h \in \mathrm{RP}$, avec :

$$\begin{aligned} h(\bar{x}, \bar{0}) &= f(\bar{x}) \\ h(\bar{x}, y+1) &= g(\bar{x}, y, h(\bar{x}, y)) \end{aligned}$$

Déf 1. On appelle *langage de programmation fonctionnelle* le langage engendré par les fonctions de base $0, \sigma, \pi$ et les deux constructeurs Comp et RecPri. Un programme est un terme construit dans ce langage ; il définit une fonction récursive primitive.

Exemple. $\mathrm{Plus}(x, 0) = x$ et $\mathrm{Plus}(x, y+1) = \sigma(\mathrm{Plus}(x, y))$.

Formellement, en écrivant $\mathrm{Plus}(x, 0) = f(x)$ et $\mathrm{Plus}(x, y+1) = g(x, y, \mathrm{Plus}(x, y))$, on trouve $f = \pi_1^1$ et $g(a, b, c) = \sigma(\pi_1^3(a, b, c))$ d'où $g = \mathrm{Comp}(\sigma, \pi_1^3)$, ce qui donne le programme :

$$\mathrm{Plus} = \mathrm{RecPri}(\pi_1^1, \mathrm{Comp}(\sigma, \pi_1^3))$$

On continue :

$$\begin{aligned} x \times y & \quad \begin{cases} x \times 0 &= 0 \\ x \times (y+1) &= x \times y + x \end{cases} \\ x^y = x \uparrow y & \quad \begin{cases} x \uparrow 0 &= 1 \\ x \uparrow (u+1) &= x \times (x \uparrow y) \end{cases} \\ x \uparrow\uparrow y = x^{x^{\cdot^{\cdot^{\cdot^x}}}} & \quad \begin{cases} x \uparrow\uparrow 0 &= 1 \\ x \uparrow\uparrow (y+1) &= x \uparrow (x \uparrow\uparrow y) \end{cases} \end{aligned}$$

Les fonctions suivantes sont donc RP : $+, \times, \uparrow, \uparrow\uparrow$, etc. On peut aussi programmer $\mathrm{sg}, \dot{-}$, définies par :

$$\begin{aligned} \mathrm{sg}(x) &= \begin{cases} 0 & \text{si } x = 0 \\ 1 & \text{si } x > 0 \end{cases} \\ x \dot{-} y &= \begin{cases} x - y & \text{si } x \geqslant y \\ 0 & \text{sinon} \end{cases} \end{aligned}$$

Propriété. Si $f(\bar{x}, y)$ est RP, alors $g(x, n) = \sum_{y=0}^{n} f(\bar{x}, y)$ et $h(x, n) = \prod_{y=0}^{n} f(\bar{x}, y)$ sont des fonctions RP (laissé en exercice).

Exemple. $n! = \prod_{y=1}^{n} y$ est donc RP.

Déf. Un prédicat P sur $\mathbb{N}^k$ ($P \subset \mathbb{N}^k$) est RP si sa fonction caractéristique χ_P est RP.

Exemple. Pair(x) définie par $\chi_{\text{Pair}}(0) = 1$ et $\chi_{\text{Pair}}(x+1) = 1 \dot{-} \chi_{\text{Pair}}(x)$.

Exemple. $x > y$ définie par $\chi_{>}(x, y) = \text{sg}(x \dot{-} y)$

Propriété. Cloture sur les prédicats. Si P, Q sont des prédicats dans RP, alors :

- $\neg P, P \wedge Q, P \vee Q$ sont dans RP
- $R(x, n) \overset{\text{déf}}{=} \forall y \leqslant n, P(x, y)$ est RP

  $R(x, n) \overset{\text{déf}}{=} \exists y \leqslant n, P(x, y)$ est RP

L'ensemble des prédicats RP estclos par opérations booléennes et quantification bornée.

Preuve. $\chi_{\neg P}(x) = 1 \dot{-} \chi_P(x)$

$\chi_{P \wedge Q}(x) = \chi_P(x) \times \chi_Q(x)$

$\chi_{P \vee Q}(x)$ se contruit avec $\wedge$ et $\neg$

$\chi_{\forall y \leqslant n, P}(x) = \prod_{Y=0}^{n} \chi_P(x, y)$

$\chi_{\exists y \leqslant n, P}(x)$ se construit à partir du précédent et de $\neg$: « $\exists^{\leqslant} = \neg \forall^{\leqslant} \neg$ »

Prop. Si $f_1, ..., f_n$ sont RP, si $P_1, ..., P_n$ sont des prédicats RP tels que $P_i \wedge P_j = \text{false}$ et $\bigvee P_i = \text{true}$, alors :

$$h(x) \;=\; \begin{cases} f_1(x) & \text{si } P_1(x) \\ f_2(x) & \text{si } P_2(x) \\ \vdots & \\ f_n(x) & \text{si } P_n(x) \end{cases}$$

est un prédicat RP.

Preuve. $h(x) = f_1(x)\, \chi_{P_1}(x) + f_2(x)\, \chi_{P_2}(x) + \cdots + f_n(x)\, \chi_{P_n}(x)$

Minimisation bornée. Si P est un prédicat RP, alors la fonction $f(x, n) = \mu^{\leqslant n} y \colon P(x, y)$ est RP, où l'on a défini : $\mu(x, n)$ est le plus petit $y \leqslant n$ tel que $P(x, y)$, ou 0 s'il n'y en a pas.

Preuve. $f(x, 0) = 0$ et $f(x, n+1) = \begin{cases} f(x,n) & \text{si } f(x,n) > 0 \\ n+1 & \text{si } P(x, n+1) \\ 0 & \text{sinon} \end{cases}$

Exemple. La division et la racine carrée entière s'écrivent :

$$\begin{aligned} x \operatorname{div} y &= (\mu m \leqslant x \colon m\, y > x) \dot{-} 1 \\ \lceil \sqrt{x} \rceil &= \mu y \leqslant x \colon y^2 \geqslant x \end{aligned}$$

Les prédicats divise et premier s'écrivent :

$$\begin{aligned} x \,|\, y &= \exists z \leqslant y \colon x\, z = y \\ \text{premier}\, x &= x > 1 \wedge (\neg \exists y < x \colon (y > 1 \wedge y \,|\, x)) \end{aligned}$$

$P(n) = n^{\text{ème}}$ nombre premier est RP ($P(0) = 2$, $P(1) = 3$, etc.) :

$$\begin{aligned} P(0) &= 2 \\ P(n+1) &= \mu y \leqslant (n+2)^2 \colon (\text{premier}\, y \wedge y > P(n)) \end{aligned}$$

$\mathrm{ex}_2(x)$ l'exposant de 2 dans la factorisation de x en nombres premiers est RP, de même que $\mathrm{ex}_3(x)$, de même que tous les $\mathrm{ex}(i,x)$ exposant de $P(i)$ dans la décomposition de x.

Remarque. On peut compiler le langage RP en fragments de C par exemple.

Compilateur. On fait une traduction par induction structurelle :

1. Fonction $0()$: `return 0;`
2. Fonction $\sigma(x)$: `return x+1;`
3. Fonction $\pi_i^k(x_1,...,x_k)$: `return x[i];`
4. Fonction $\mathrm{Comp}(f,g)(x)$:

   ```
   y = g(x);
   return f(y);
   ```

5. Fonction $\mathrm{RecPri}(f,g)(x,y)$:

   ```
   z = f(x);
   for (j = 0; j < y; j++) z = f(x, j, z);
   return z;
   ```

Ces fonctions sont définies avec des boucles `for` simples et sans récursion.

Question. Est-ce que RP est l'ensemble des fonctions calculables ? Problème, la notion de fonction calculable est informelle.

On comprend bien avec notre compilateur RP $\subset$ calculable. Mais on n'a pas l'inclusion inverse ! La classe RP n'est pas suffisante.

Exemple 1. On a $x\uparrow^0 y = x \times y$, et on a défini les flèches $x\uparrow^k y$ par récurrence :

$$\begin{aligned} x\uparrow^{k+1} 0 &= 1 \\ x\uparrow^{k+1}(y+1) &= x\uparrow^k (x\uparrow^{k+1} y) \end{aligned}$$

Ce qui donne : $x\uparrow^0 y = x\,y$, $x\uparrow^1 y = x^y$, $x\uparrow^2 y = x^{x^{\cdot^{\cdot^x}}}$, $x\uparrow^3 y = \cdots$ Bien sûr, $\forall k$ la flèche $\uparrow^k$ est RP. Soit :

$$\mathrm{Ack}(x,y,k) = x\uparrow^k y$$

1. Ack est calculable, on vient de l'écrire (on a un programme intuitif facile à traduire en C par exemple, ce programme s'arrête).
2. Ack $\notin$ RP, elle croît trop vite. Idée de preuve : montrer par induction structurelle que $\forall f \in \mathrm{RP}, \exists k : f \leqslant \uparrow^k$. Or pour tout k fixe, $\mathrm{Ack} \geqslant \uparrow^k$, donc elle n'est pas RP.

Exemple 2. On peut énumerer toutes les fonctions RP de $\mathbb{N}$ à $\mathbb{N}$, ie il existe une liste exhaustive f_1, $f_2, f_3, ..., f_n, ...$ de toute ces fonctions tel qu'il existe un interpréteur Int qui cacule $\mathrm{Int}(k,x) = f_k(x)$.

Fonctionnement de Int sur (k,x) :

1. Prendre la chaine ASCII numéro k, que l'on appelle s
2. Si s n'est pas un terme récursif primitif définissant une fonction unaire, `return 0;`
3. Si s est un terme correct, on calcule la fonction RP correspondante en x.

On peut le programmer en C par exemple, donc c'est calculable.

Soit $h(x) = f_x(x) + 1$. Alors :

a) $h(x) = \mathrm{Int}(x, x) + 1$, donc elle est calculable par un programme C

b) $h(0) = f_0(0) + 1$, donc $h \neq f_0$

$\forall k \in \mathbb{N},\ h(k) = f_k(k) + 1$, donc $h \neq f_k$

Donc $h \notin f_0, f_1, \ldots$! C'est-à-dire $h \notin \mathrm{RP}$.

(c'est l'argument de la diagonale)

c) Supposons que $h \in \mathrm{RP}$. Soit k son numéro, $h = f_k$, en particulier $h(k) = f_k(k)$ mais d'autre part $h(k) = f_k(k) + 1$, donc $1 = 0$, c'est impossible.

Conclusion. Il existe une fonction calculable h non RP. Si on étend la classe RP par de nouvelles fonctions ou constructions, ce problème restera.

III.1.b. Fonctions récursives partielles

Solution. On va considérer des fonctions partielles $f : \mathbb{N}^k \cdots \to \mathbb{N}$, c'est-à-dire des fonctions d'un sous-ensemble de $\mathbb{N}^k$ vers n. Si $x \notin \mathrm{dom}\, f$, on écrit $f(x)\uparrow$ (lire $f(x)$ diverge). Si $x \in \mathrm{dom}\, f$, on écrit $f(x)\downarrow$ (lire $f(x)$ converge).

Exemple.

$$\begin{aligned} f(x,y) &= \begin{cases} x/y & \text{si } y \text{ divise } x \\ \uparrow & \text{sinon} \end{cases} \\ g(x,y) &= f(x,y) \times 0 + 3 \\ &= \begin{cases} 3 & \text{si } y \text{ divise } x \\ \uparrow & \text{sinon} \end{cases} \end{aligned}$$

Remarque. Ça colle bien avec les langages de programmation. Par exemple pour une fonction en C, il y a des instructions `return` qui renvoient des résultats sur certaines entrées, mais pas forcément sur toutes.

Déf. Une *fonction totale* est une fonction telle que $\forall x,\ f(x)\downarrow$, ie $\mathrm{dom}\, f = \mathbb{N}^k$.

Minimisation non bornée (définition provisoire). Soit $f(x,\ y)$ une fonction totale. Alors $\mu y(f(x,y) = 1)$ est la fonction $g(x)$ qui vaut le plus petit entier y tel que $f(x,y) = 1$ si un tel entier existe, et diverge sinon.

Définition. Fonctions récursives partielles. Ce sont toutes les fonctions que l'on obtient avec $0, \sigma, \pi_i^k$ et les constructions Comp, RecPri et μ.

Exemple. $\mu z : y\, z = x$ définit la fonction de division partielle f donnée ci-dessus.

Exemple. $\mu z : z^2 = x$ définit la fonction racine carrée partielle. $\sqrt{4} = 2$ et $\sqrt{2}\uparrow$.

Exemple. Nombres parfaits.

$$\begin{aligned} \mathrm{par}(0) &= 6 \\ \mathrm{par}(n+1) &= \mu z : (z > \mathrm{par}(n) \wedge \mathrm{parfait}(z)) \end{aligned}$$

On ne sait pas si la fonction par est totale.

Thèse de Church. f est calculable par un algorithme $\Leftrightarrow$ f est récursive partielle.

Attention. Il faut bien redéfinir les opérateurs Comp et RecPri sur des fonctions partielles :

$$\begin{aligned}\mathrm{Comp}(f,g)(x) &= \begin{cases} f(g(x)) & \text{si } g(x)\downarrow \text{ et } f(g(x))\downarrow \\ \uparrow & \text{sinon} \end{cases} \\ \mathrm{RecPri}(f,g) &= \text{reprendre la définition précédente (cf programme en C), en précisant quelles fonctions doivent converger.}\end{aligned}$$

Traduction de l'opérateur μ en C. $\mu y\colon f(x,y)=0$ s'écrit :

```
y = 0;
while (f(x, y) != 0) y++;
return y;
```

Définition finale des fonctions récursives partielles. On corrige la définition de μ. $g(x) = \mu y\,(f(x,y)=1)$ se définit par :

$$g(x) = \begin{cases} \text{le plus petit } y \text{ tel que } f(x,y)=1 \text{ et } \forall y \text{ plus petit, } f(x,y)\downarrow \\ \uparrow \text{ sinon} \end{cases}$$

Remarques.

a) Il est très facile de se tromper en appliquant μ ou RecPri à des fonctions non-totales.

b) Heureusement, c'est inutile – voir forme normale, plus tard.

Remarque. Preuve d'1/2 théorème de Church. Si f est une fonction récursive partiele, alors il existe une foncion C qui calcule $f(x)$ et boucle dans le cas $f(x)\uparrow$. Cette démonstration est facile par induction structurelle, on « compile » les fonctions récursives partielles en code C.

On aimerait montrer qu'une fonction « calculable » est récursive partielle. On aura souvent des théorèmes du style : une fonction définie dans tel modèle de calcul est une fonction récursive partielle.

III.2. Machines de Turing : une autre définition des fonctions calculables.

Une machine de Turing est définie par un automate plus une mémoire (traditionnellement un ruban).

Déf. $M=(Q,\Sigma,q_0,\Delta)$, où Δ correspond au programme. Δ est un ensemble fini d'instructions de la forme :

$$\begin{aligned} p,a &\to q,b,\mathrm{depl} \\ \mathrm{depl} &\in \{G,D,-\}\end{aligned}$$

On dessinera parfois :

$$p \xrightarrow{a/b,G} q$$

Une telle transition correspond à : si on est dans l'état p et que la case du ruban qui est sous la tête de lecture contient un a, on remplace ce a par un b, on passe dans l'état q et on se déplace d'une case vers la gauche.

Déf. M est déterministe si pour p,a donnés, il existe dans Δ zéro ou une instruction $p,a\to\cdots$.

Pour les deux ou trois cours suivants, on ne s'intéresse qu'aux machines de Turing déterministes.

Déf.

- Une configuration se note w_G, q, w_D (la tête de lecture est sur la première lettre de w_D).
- Une transition est de la forme :

$$w_G\, q\, w_D \to w'_G\, q'\, w'_D$$

  et est conforme à une instruction.

- Un calcul est une chaîne (finie ou infinie) de transitions.

Déf. On se donne une machine de Turing et $k \in \mathbb{N}$. On définit son calcul sur $x_1, ..., x_k \in \mathbb{N}^k$ comme étant le calcul qui commence dans $(\varepsilon, q_0, 0\,1^{x_1}\,0\,1^{x_2}\,0 \cdots 0\,1^{x_k})$, qui termine dans une configuration où la machine n'a pas de transition, et dont le résultat est le nombre de 1 sur le ruban à la fin du calcul. On note $f^k_M(x_1, ..., x_k)$ le résultat du calcul de M sur $x_1, ..., x_k$ ou $\uparrow$ si le calcul est infini (la machine ne s'arrête pas).

Théorème. f est une fonction récursive partielle $\Leftrightarrow$ f est calculable par une machine de Turing.

Exemple. On a la machine M avec une seule instruction : $q_0, 0 \to q_1, 1, G$. La fonction calculée par cette machine est $f^1_M = \sigma$. Avec deux arguments, la machine calcule $f^2_M(x, y) = 1 + x + y$.

2013-11-21

Preuve.

Sens $\Rightarrow$: sens facile.

On prouve par induction structurelle sur f récursive partielle que f est *bien calculable* par une machine de turing.

Déf. On dit que M *calcule bien* f si :

1. M calcule f
2. M s'arrête sur l'état q_1 seulement
3. M ne va jamais à gauche de sa position initiale
4. À la fin du calcul sur $(x_1, ..., x_k)$ on a sur le ruban un mot de $0^*1^{f(x_1,\cdots,x_k)}0^*$, avec la tête de lecture positionnée sur le dernier 0 avant les 1.

Compilation des fonctions de classe RP vers machines de Turing qui calculent *bien* :

1. Fonction 0 : une seule transition,

$$q_0 \xrightarrow{0/0,-} q_1$$

2. Fonction σ : aller à droite jusqu'au zéro après le premier groupe de 1, remplaçer le 0 par un 1, revenir à gauche, s'arêter sur le premier zéro rencontré.
3. Fonction $\pi_{i,k}$: facile aussi
4. Fonction Comp dans le cas scalaire (f et g n'ont qu'un seul argument) :

   (rappel : $\mathrm{Comp}(g, f)(x) = g(f(x))$

On a une machine F qui calcule bien f : $F = (Q_F, \Sigma, q_{0,F}, \Delta_F)$

On a une machine G quil calcule bien g : $G = (Q_G, q_{0,G}, \Delta_G)$

On suppose que $Q_F \cap Q_G = \varnothing$ (il suffit de renommer les états).

On crée la machine $M = \left(Q_F \cup Q_G, \Sigma, q_{0,F}, \Delta_F \cup \Delta_G \cup \left\{ q_{1,F} \xrightarrow{0/0,-} q_{0,G} \right\} \right)$

On vérifie que cette machine calcule bien $g(f(x))$ (faire des jolis dessins)

5. Fonction RecPri(f, g) :

   On dispose de deux machines F et G qui calculent respectivement $f(x)$ et $g(x, i, v)$.

   On cherche une machine pour $h(x, y)$ définie par $h(x, 0) = f(x)$ et $h(x, i+1) = g(x, i, h(x, i))$, ce que l'on peut écrire par $h(x, y) = g(x, i-1, g(x, i-2, (x, i-3(\cdots g(x, 0, f(x))))))$

   Pseudocode pour la machine M calculant RecPri$(f, g)(x, y)$:

   Sur le ruban en entrée : x, y

   $y_1 = y$ (copier y sur le ruban)

   $i = 0$

   $x_1 = x$

   $h = f(x_1)$ (propriété : $h = h(x, 0)$)

   répéter tant que $y_1 > 0$ (invariant : $y_1 + i = y, h = h(x, i)$)

   $x_1 = x$

   $i_1 = i$

   $h_1 = h$

   decr y, incr i

   $i_1 = i$

   $h = g(x_1, i_1, h_1)$

   Effacer tout sauf h

   À la fin, on a $y_1 = 0$, $i = y$ par invariant, et donc $h = h(x, y)$. On a la valeur cherchée.

6. Calcul de μ : un peu plus facile, laissé en exercice.

On a donc la compilation de fonctions récursives partielles vers des machines de Turing.

Sens $\Leftarrow$: sens difficile. On se donne un machine de turing $M = (Q, \Sigma, q_0, \Delta)$, on veut la coder par une fonction récursive partielle.

Méthode : il faut coder les mots par des nombres : ce processus s'appelle arithmétisation ou Gödelisation (démarche inventée par Gödel).

On doit nécessairement faire un nombre important de choix sur la manière de Gödeliser, ceux-ci seront faits de manière arbitraire.

On définit les numéros de Gödel de x, noté $\langle x \rangle$ avec $\langle x \rangle \in \mathbb{N}$, pour divers types d'objets x :

1. Pour un n-uplet d'entiers $(x_1, ..., x_n)$, on pose :

$$\langle (x_1, ..., x_n) \rangle \;=\; 2^n \prod_{i=1}^{n} p_{i+1}^{x_i}$$

où p_i représente le i-ème nombre premier.

Ex. $\langle(10,0,4)\rangle = 2^3\,3^{10}\,5^0\,7^4 = 1134213192$

Prop. Ce codage est une application $\mathbb{N}^* \to \mathbb{N}$ injective.

Prop. Toutes les opérations utiles sont récursives primitives :

- $\text{est\_un\_no}(m) = \begin{cases} 1 & \text{si } \exists (x_1,\dots,x_n) : m = (\langle x_1,\dots,x_n\rangle) \\ 0 & \text{sinon} \end{cases}$

  Ex. $100 = \langle(0,2)\rangle$ mais $1100 = 2^2\,5^2\,11$ n'est pas un numéro de Gödel.

- $\text{long}(m) = \begin{cases} m & \text{si } m = \langle(x_1,\dots,x_m)\rangle \\ 0 & \text{sinon} \end{cases}$

- $\text{el}(m,i) = \begin{cases} x_i & \text{si } m = \langle(x_1,\dots,x_m)\rangle \text{ avec } m \geqslant i \\ 0 & \text{sinon} \end{cases}$

- replace(m,i,x) : remplacer x_i par x, ou 0 si erreur de taille

- etc... (remplacement d'un élément, ...)

2. On fixe un alphabet $\Sigma = \{a_1,\dots,a_n\}$,

   a. On pose $\langle a_i\rangle = i$

   b. Soit $w = w_1 \cdots w_k$, alors $\langle w\rangle = 2^k \prod_{i=1}^{k} p_{i+1}^{\langle w_i\rangle} = \langle(\langle w_1\rangle, \langle w_2\rangle, \dots, \langle w_k\rangle)\rangle$

   Exemple : si $a = a_1$ et $b = a_2$, alors $\langle b\,a\,b\,a\rangle = \langle(2,1,2,1)\rangle = 2^4 3^2\,5\,7^2\,11 = 388080$

3. Machine de Turing $M = (Q, \Sigma, q_0, \Delta)$:

   On numérote $Q = \{q_0, q_1, \dots, q_n\}$ et $\Sigma = \{a_0 = 0, a_1 = 1, a_2, \dots, a_n\}$

   a. Pour un état, $\langle q_i\rangle = i$

   b. Pour une configuration $w_G\,q\,w_D$ (tête de lecture sur la premierre lettre de w_D),

   $$\langle w_G\,q\,w_D\rangle \;=\; \langle(\langle w_G\rangle, \langle q\rangle, \langle w_D\rangle)\rangle$$

   c. Pour une instruction $p \xrightarrow{a/b,\text{depl}} q$, posons déjà $\langle\leftarrow\rangle = 0, \langle\rightarrow\rangle = 1, \langle -\rangle = 2$. Puis :

   $$\left\langle p \xrightarrow{a/b,\text{depl}} q \right\rangle \;=\; \langle(\langle p\rangle, \langle a\rangle, \langle b\rangle, \langle\text{depl}\rangle, \langle q\rangle)\rangle$$

   d. Soit $\Delta = \{i_1, i_2, \dots, i_r\}$. On pose maintenant le numéro de la machine M :

   $$\langle M\rangle \;=\; \langle(\langle i_1\rangle, \langle i_2\rangle, \dots, \langle i_r\rangle)\rangle$$

On a maintenant une machine M de numéro $m = \langle M\rangle$. On cherche à exprimer la fonction $f_M : \mathbb{N} \to \mathbb{N}$ en langage récursif partiel.

1. Soit $\text{init}(x) = \langle$configuration initiale de la machine : q_0 sur 0, suivit de $1^x\rangle$

   Alors init $\in$ RP

2. On prend la configuration de numéro c, on fait un pas de calcul de la machine de Turing numéro m :

On obtient une configuration dont le numéro est notée suivant(m, c).

$$\begin{aligned} \text{conf} &\xrightarrow{M} \text{conf}' \\ \langle \text{conf} \rangle &= c \\ \text{suivant}(\langle M \rangle, c) &= \langle \text{conf}' \rangle \text{ ou 0 si erreur} \end{aligned}$$

Alors suivant $\in$ RP

3. conf(m, x, t) : numéro de Gödel de la configuration de la machine m sur entrée x après t pas de calcul, 0 si problème.

   conf $\in$ RP :

$$\begin{aligned} \text{conf}(m, x, 0) &= \text{init}(x) \\ \text{conf}(m, x, t+1) &= \text{suivant}(\text{conf}(m, x, t)) \end{aligned}$$

4. $\text{stop}(m, c) = \begin{cases} 1 & \text{si } m \text{ est bloquée sur la configuration de numéro } c \\ 0 & \text{sinon} \end{cases}$

   stop $\in$ RP

5. temps(m, x) : temps de calcul de la machine de numéro m sur X, *lorsque celle-ci s'arrête.*

   $\text{temps}(m, x) = \mu.t : \text{stop}(m, \text{conf}(m, x, t))$

   temps est donc récursive *partielle* (première utilisation du quantificateur μ !)

6. sortie(c) : nombre de 1 sur le ruban dans la configuration de numéro c, 0 si erreur (par exemple si c n'est pas le code d'une configuration)

   sortie $\in$ RP

7. $\mathcal{U}(m, x)$: résultat de calcul de la machine de numéro m sur l'entrée x.

$$\mathcal{U}(m, x) = \text{sortie}(\text{conf}(m, x, \text{temps}(m, x)))$$

   $\mathcal{U}$ est récursive partielle.

On conclut : $f_M(x) = \mathcal{U}(\langle M \rangle, x)$, est donc récursive partielle. □

On a donc démontré. f MT-calculable $\Leftrightarrow$ f récursive partielle

Remarque. Dans le même style on peut faire f récursive partielle $\Leftrightarrow$ f programmable en C#.

Thèse de Church-Turing. f « calculable par un algo » $\Leftrightarrow$ f récursive partielle $\Leftrightarrow$ f MT-calculable.

Corollaire. Si f est récursive partielle, elle peut être exprimée sous la forme suivante, appellée *forme normale* :

$$f(x) = g(x, \mu.y : P(x, y))$$

où g est une fonction RP et P est un prédicat RP.

Cela signifie qu'un seul quantificateur μ suffit.

Preuve. On passe par une machine de Turing : on a la formule

$$f(x) = \mathcal{U}(m, x) = \text{sortie}(\text{conf}(m, x, \text{temps}(m, x)))$$

Cette écriture ne contient qu'un quantificateur μ. □

Théorème : fonction universelle. Il existe une fonction récursive partielle (ie. Turing-calculable) $\mathcal{U}(m,x)$ telle que pour chaque machine de Turing M,

$$\mathcal{U}(\langle M\rangle, x) = f_M(x)$$

$\mathcal{U}$ sait donc faire les calculs de chaque machine de Turing. On a donc une machine de Turing universelle $M_{\mathcal{U}}$ qui calcule $\mathcal{U}$, ou une fonction récursive partiellequi sait calculer toutes les fonctions récursives partielles.

(on a en fait des fonctions $\mathcal{U}_k : \mathbb{N}^{k+1} \to \mathbb{N}$, selon l'arité des fonctions calculées).

Théorème d'énumération. On peut énumérer toutes les fonctions récursives partielles $\varphi_0, \varphi_1, ..., \varphi_k, ...$, et la fonction $i, x \mapsto \varphi_i(x)$ est une fonction récursive partielle.

Preuve. Soit $\varphi_i(x) = \mathcal{U}(i, x)$, autrement dit $\varphi_{\langle M\rangle}(x) = f_M(x)$ quand M est une machine de Turing de numéro $\langle M\rangle$, et $\varphi_m(x)$ diverge si m est invalide. □

(remarque : problèe d'arité...)

Théorème s-m-n.

On a une fonction calculable $\varphi^{(2)}_m(x,y) = f(x,y)$.

On considère x comme paramètre : $y \mapsto f(0, y) = \varphi^{(1)}_{c_0}(y)$, $y \mapsto f(1, y) = \varphi^{(1)}_{c_1}(y)$, ..., $y \mapsto f(k, y) = \varphi^{(1)}_{c_k}(y)$, où c_k est la machine m où on a fixé le premier argument égal à k : $c_k = s(m,k)$ pour une certaine fonction $s \in \mathrm{RP}$.

Théorème s-m-n. Il existe une fonction récursive primitive s telle que $\varphi_m(x, y) = \varphi_{s(m,x)}(y)$. $s(m,x)$ prend le programme m ayant deux argument et fixele premier argument à x.

Idée de preuve. On a m un numéro d'une machine de Turing M et x une valeur de paramètre.

On construit une nouvelle machine M' (de numéro $s(m, x)$) d'argument y, qui passe de la configuration $q_{0,M'}\,0\,1^y$ à laconfiguration $q_{0,M}\,0\,1^x\,0\,1^y$, puis fait le calcul de M.

$M' =$ instructions pour écrire $x \cup$ toutes les instructions de M.

Application de s-m-n. Trouver h telle que $\varphi_{h(x,y)} = \mathrm{Comp}(\varphi_x, \varphi_y)$: comment composer deux fonctions récursives partielles, du point de vue de leurs numéros ?

$f(x,y,z) \overset{\text{def}}{=} \varphi_x(\varphi_y(z)) = \mathcal{U}(x, \mathcal{U}(y,z))$, donc f est récursive partielle. Soit b son numéro.

On a donc $\varphi_b(x,y,z) = \mathrm{Comp}(\varphi_x, \varphi_y)(z)$. Par s-m-n, $\varphi_{s(b,x,y)}(z) = \mathrm{Comp}(\varphi_x, \varphi_y)(z)$.

III.3. Ensembles (ou prédicats) décidables et indécidables.

Déf. $P \subset \mathbb{N}^k$ est décidable si $\chi_P(\bar{x}) = \begin{cases} 1 & \text{si } x \in P \\ 0 & \text{sinon} \end{cases}$ est récursive totale, c'est-à-dire récursive partielle et totale.

Propriété. Les ensembles décidables sont clos par $\cap, \cup, {}^-$.

Dém. $\chi_{A\cap B} = \chi_A\,\chi_B$, $\chi_{A\cup B} = \chi_A \cup \chi_B$, $\chi_{\bar{A}} = 1 - \chi_A$

Déf. Un ensemble/prédicat intéressant : $\mathrm{arret} = \{(x,y) : \varphi_x(y)\downarrow\}$

$\mathrm{arret}(x,y) \Leftrightarrow \varphi_x(y)\downarrow \Leftrightarrow$ la machine de numéro x s'arrête sur l'entrée y.

Déf. $K(x) \Leftrightarrow \mathrm{arret}(x,x)$

Théorème. K est indécidable.

Dém. Supposons K décidable : il existe une fonction récursive totale :

$$g(x) = \begin{cases} 1 & \text{si } \varphi_x(x)\downarrow \\ 0 & \text{si } \varphi_x(x)\uparrow \end{cases}$$

On construit une fonction paradoxale par une méthode « diagonale » :

$$d(x) = \begin{cases} 0 & \text{si } \varphi_x(x)\uparrow \\ \uparrow & \text{si } \varphi_x(x)\downarrow \end{cases} = v(g(x)) \text{ avec } v(0) = 0 \text{ et } v(1)\uparrow$$

On peut choisir v récursive partielle, par ex $v(z) = \mu.y\colon (0 = z)$. d est donc récursive partielle.

On a donc $d(x)\downarrow \Leftrightarrow \varphi_x(x)\uparrow$.

$\forall k, d(k)\downarrow \Leftrightarrow \varphi_k(k)\uparrow$, donc $d \neq \varphi_k$. d n'est donc pas dans la liste de toutes les fonctions récursives partielles, il y a contradiction.

Autre contradiction (paradoxe du barbier) : d est récursive partielle, soit b son numéro. $d = \varphi_b$. $\forall x, \varphi_b(x)\downarrow \Leftrightarrow \varphi_x(x)\uparrow$. Prenons $x = b$, on a $\varphi_b(b)\downarrow \Leftrightarrow \varphi_b(b)\uparrow$, contradiction.

Donc K est indécidable. □

$\chi_K(x) = \chi_{\text{arret}}(x, x)$. Si arrêt était décidable, χ_{arret} serait récursive, donc χ_K aussi, donc K serait décidable, contradiction. Donc arrêt est non décidable.

`2013-11-28`

Planning : on finit les cours en décembre ; le 9 janvier : 8 exposés ; le 17 janvier : 14 exposés ; le 24 janvier : examen.

Rappel. On ne peut pas énumérer les fonctions totales « calculables ». En effet, si on a une telle liste $f_0, f_1, ..., f_n, ...$, alors la fonction $d(x) = f_x(x) + 1$ ne peut pas être dans cette liste mais dois y être car elle est totale, absurde.

Objectif. Prouver l'indécidabilité de problèmes.

Définition. Soient A et B deux prédicats sur $\mathbb{N}^k$ et $\mathbb{N}^l$ (ou sous-ensembles de $\mathbb{N}^k$ et $\mathbb{N}^l$). On dit que $A \leqslant_m B$ (A se réduit à B) si il existe h récursive totale telle que $\forall x, A(x) \Leftrightarrow B(h(x))$.

($\leqslant_m$: many-to-one ; il existe aussi la réduction $\leqslant_1$ one-to-one)

Rmq. Il manque une définition de $h\colon \mathbb{N}^k \to \mathbb{N}^l$ récursive totale. Exo : à définir.

Propriétés.

1. $\leqslant_m$ est réflxive et transitive.

2. Si A est décidable, $B \neq 0$ et $\bar{B} \neq \varnothing$, alors $A \leqslant_m B$.

   Preuve. En exo.

3. Si $A \leqslant_m B$ et B est décidable, alors A est décidable.

   Preuve. $\chi_A(x) = \chi_B(h(x))$ est une fonction récursive totale □

   Corollaire. (utilisé en calculabilité) Si $A \leqslant_m B$ et A indécidable, alors B est indécidable. Donc pour prouver que A est indécidable, il suffit de prouver que $K \leqslant_m A$ (K est indécidable, comme le problème d'arrêt).

Exemple. $P(x) \Leftrightarrow \varphi_x(12) = 13$, on va prouver que P est indécidable en montrant la réduction $K \leqslant_m P$:

On transforme prog x en prog $h(x)$: on veut $\varphi_{h(x)}(y) = \begin{cases} 13 & \text{si } \varphi_x(x)\downarrow \\ \uparrow & \text{si } \varphi_x(x)\uparrow \end{cases}$. Si on a un tel h alors $x \in K \Rightarrow \varphi_x(x)\downarrow \Rightarrow \forall y,\ \varphi_{h(x)}(y) = 13 \Rightarrow \varphi_{h(x)}(12) = 13 \Rightarrow h(x) \in P$, et dans l'autre sens : $x \notin K \Rightarrow \varphi_x(x)\uparrow \Rightarrow \forall y, \varphi_{h(x)}(y)\uparrow \Rightarrow \varphi_{h(x)}(12) \neq 13 \Rightarrow h(x) \notin P$.

Informellement, construction de $h(x)$: $\varphi_{h(x)}(y)$: on calcule $\varphi_x(x)$ et quand ça s'arrête on renvoie 13. Formellement :

$$\varphi_{h(x)}(y) = \varphi_{s(a,x)}(y) = \varphi_a(x,y) = f(x,y) \overset{\text{def}}{=} \begin{cases} 13 & \text{si } \varphi_x(x)\downarrow \\ \uparrow & \text{si } \varphi_x(x)\uparrow \end{cases}$$

On sait que $f(x,y) = \mathcal{U}(x,y) \cdot 0 + 13$, donc f est récursive partielle. Soit a son numéro : on applique le s-m-n et on pose $h(x) = s(a,x)$.

Déf. Soit C un ensemble de fonctions partielles de $\mathbb{N} \to \mathbb{N}$. On dit que C est non-trivial si :

1. $\exists f$ récursive partielle telle que $f \in C$
2. $\exists f$ récursive partielle telle que $f \notin C$

Théorème de Rice. Soit C non trivial. On définit $P_C = \{x : \varphi_x \in C\}$. Alors P_C est indécidable.

« Toutes les propriétés des fonctions récursives partielles ou des machines de Turing sont indécidables. »

Preuve. Soit $\perp$ la fonction définie par $\forall x, \perp(x)\uparrow$ (elle est récursive partielle).

1. Cas n°1 : $\perp \notin C$.

   Soit f une fonction récursive partielle telle que $f \in C$. On va réduire $K \leqslant_m P_C$.

   Pour cela, on construit h telle que $\varphi_{h(x)}(y) = \begin{cases} f(y) & \text{si } \varphi_x(x)\downarrow \\ \uparrow & \text{si } \varphi_x(x)\uparrow \end{cases}$

   Alors on aura $x \in K \Rightarrow \varphi_{h(x)} = f \in C \Rightarrow h(x) \in P_C$ et $x \notin K \Rightarrow \varphi_{h(x)} = \perp \notin C \Rightarrow h(x) \notin P_C$, donc $K \leqslant_m P_C$.

   Pour construire $h(x)$ RP, on utilise s-m-n.

2. Cas n°2 : $\perp \in C$.

   Soit f une fonction récursive partielle telle que $f \notin C$. Alors avec la même réduction, on montre $K \leqslant_m \overline{P_C}$, donc $\overline{P_C}$ est indécidable, donc P_C est indécidable. □

Remarque. Les problèmes suivants sur les MT peuvent être dédidables :

1. Est-ce que la machine de numéro x a un nombre premier d'instructions ?
2. Est-ce que $\varphi_x(10) = 12$ et le calcul prend moins de $10^{10^{10}}$ étapes ?

Mais les propriétés suivantes ne sont pas décidables :

1. $\forall y, \varphi_x(y) = y!$
2. φ_x est récursive totale.

Remarque : semi-décidabilité. Pour $K(x)$, arret(x,y), $\varphi_x(12) = 13$, si la réponse est oui alors on le saura à un moment.

Semi-décidabilité

Déf. $P \subset \mathbb{N}^l$ est semi-décidable s'il existe f récursive partielle telle que $P = \text{dom } f$, c'est-à-dire $x \in P \Leftrightarrow f(x)\downarrow$.

Variante. Soit $\psi_P(x) = \begin{cases} 1 & \text{si } x \in P \\ \uparrow & \text{si } x \notin P \end{cases}$, que l'on appelle fonction semi-caractéristique de P. P est semi-décidable $\Leftrightarrow$ ψ_P est récursive partielle.

Preuve. $\Leftarrow$: Si ψ_P est récursive partielle, on a $P = \operatorname{dom} \psi_P$, donc P semi-décidable

$\Rightarrow$: Soit P semi-décidable. Alors il existe f RP telle que $x \in P \Leftrightarrow f(x)\downarrow$. On peut maintenant poser $\psi_p = f(x) \cdot 0 + 1$. □

Déf. $P \subset \mathbb{N}$ est récursivement énumérable (RE) si $P = \varnothing$ ou $\exists h$ récursive totale telle que $P = \operatorname{im} h$ ($P = \{h(0), h(1), ...\}$).

Théorème. P est RE $\Leftrightarrow$ P est semi-décidable.

Preuve. Sens $\Rightarrow$: Soit P RE non vide, alors $P = \{h(0), h(1), ...\}$

$x \in P \Leftrightarrow \exists n : h(n) = X \Leftrightarrow \mu.n : h(n) = x\downarrow \Leftrightarrow f(x)\downarrow$, avec $f(x) \overset{\text{def}}{=} \mu.n : h(n) = x$

Donc $P = \operatorname{dom} f$, et f est récursive partielle. Donc P est semi-décidable.

Sens $\Leftarrow$: Soit P semi-décidable et $P \neq \varnothing$.

Forme normale. P est semi-décidable $\Rightarrow$ il existe une fonction R récursive primitive telle que $(x \in P \Leftrightarrow \exists t : R(x, t))$.

Preuve. P semi-décidable $\Rightarrow$ $P = \operatorname{dom} f$. Par forme normale de f, $f(x) = h(x, \mu.t : R(x, t))$ avec h et R récursifs primitifs. On a maintenant $x \in P \Leftrightarrow f(x)\downarrow \Leftrightarrow \exists t : R(x, t)$

$x \in P \Leftrightarrow \exists t : R(x, t)$ avec R un prédicat RP. Soit $a \in P$ un élément quelconque. On définit h :

$$h(2^x 3^t) = \begin{cases} x & \text{si } R(x, t) \\ a & \text{sinon} \end{cases}$$

$$\text{ou, pour être totale : } h(z) = \begin{cases} \text{ex}_2(z) & \text{si } R(\text{ex}_2(z), \text{ex}_3(z)) \\ a & \text{sinon} \end{cases}$$

(les fonctions ex_p renvoient l'exposant de p dans la décomposition en facteurs premiers, par exemple $\text{ex}_2(n) = \mu.i \leqslant n : (2\uparrow(n+1)$ ne divise pas $n)$).

On prouve que $P = \operatorname{Im} h$:

$P(x) \Rightarrow \exists t : R(x, t) \Rightarrow h(2^x 3^t) = x \Rightarrow x \in \operatorname{Im} h$.

Réciproquement, $x \in \operatorname{Im}(h) \Rightarrow \exists z : x = h(z)$. Soit pour z on a la première ligne : $x = \text{ex}_2(z) \Rightarrow R(\text{ex}_2(z), \text{ex}_3(z)) \Rightarrow R(x, \text{ex}_3(z)) \Rightarrow \exists t : R(x, t) \Rightarrow P(x)$. Soit on a la deuxième ligne : $x = a$, donc $P(x)$. □

Théorème de Post. P décidable $\Leftrightarrow$ P semi-décidable et $\bar{P}$ semi-décidable.

Preuve.

Sens $\Rightarrow$:

P décidable $\Rightarrow$ $\chi_P(x) = \begin{cases} 1 & \text{si } x \in P \\ 0 & \text{si } x \notin P \end{cases}$ est récursive totale $\Rightarrow$ $\psi_P(x) = \begin{cases} 1 & \text{si } x \in P \\ \uparrow & \text{sinon} \end{cases} = v(\chi_P(x)) \Rightarrow \psi_P$ est récursive-partielle $\Rightarrow$ P est semi-décidable.

P décidable $\Rightarrow$ P semi-décidable, et P décidable $\Rightarrow$ $\bar{P}$ décidable $\Rightarrow$ $\bar{P}$ semi-décidable

Sens $\Leftarrow$: P et $\bar{P}$ semi-décidables, donc P et $\bar{P}$ sont récursivement énumérables(le cas où P ou $\bar{P}$ est vide est trivial). Il existe donc des fonctions h et g récursives totales telles que $P = \operatorname{Im} h$ et $\bar{P} = \operatorname{Im} g$.

Pour décider $x \in P$, on génère P et $\bar{P}$ et on attend que x apparaisse dans l'un des deux :

$$t(x) = \mu.t : (h(t) = x \vee g(t) = x)$$

Cette fonction est récursive totale.

On pose maintenant $\chi_P(x) = \begin{cases} 1 & \text{si } h(t(x)) = x \\ 0 & \text{sinon} \end{cases}$.

Si $x \in P$ alors $\exists t : h(t) = x$ et $\neg\exists t : g(t) = x$, μ trouvera le plus petit t tel que $h(t(x)) = x$, ok.

Si $x \notin P$, alors $\neg\exists t : h(t) = x$ et $\exists t : g(t) = x$, μ trouera le plus petit t tel que $g(t(x)) = x$, ok car $h(t(x)) \neq x$. □

Corollaire. K est semi-décidable, $\bar{K}$ n'est pas semi-décidable, arret est semi-décidable, $\overline{\text{arret}}$ n'est pas semi-décidable.

Notation. $K \in \text{SD}, \bar{K} \notin \text{SD}, \text{arret} \in \text{SD}, \overline{\text{arret}} \notin \text{SD}$.

Preuve. $x \in K \Leftrightarrow \varphi_x(x)\downarrow \Leftrightarrow \mathcal{U}(x,x)\downarrow$, $K = \text{dom}(x \mapsto \mathcal{U}(x,x))$ donc $K \in \text{SD}$.

Si $\bar{K} \in \text{SD}$, alors K serait décidable, ce qui n'est pas le cas, donc $\bar{K} \notin \text{SD}$. □

Exemple. Soit $\text{Tot} = \{x : \varphi_x \text{ est totale}\} = \{x : \forall y, \varphi_x(y)\downarrow\}$. On a : $\text{Tot} \notin \text{SD}$, $\overline{\text{Tot}} \notin \text{SD}$.

Preuve. $K \leqslant_m \text{Tot}$ (comme preuve du théorème de Rice), donc $\bar{K} \leqslant_m \overline{\text{Tot}}$ (à démontrer : $A \leqslant_m B \Rightarrow \bar{A} \leqslant_m \bar{B}$, laissé en exo).

On en conclut que $\overline{\text{Tot}} \notin \text{SD}$ (à démontrer : $(A \leqslant_m B \wedge B \in \text{SD}) \Rightarrow A \in \text{SD}$, laissé en exo).

Supposons que Tot $\in$ RE (récursivement énumérable). Alors Tot = Im h pour une fonction h récursive partielle. Posons $d(x) = \varphi_{h(x)}(x) + 1$, d est récursive totale, donc $d = \varphi_b$ pour $b \in \text{Tot}$, donc $\exists a : b = h(a)$, donc $\exists a : d = \varphi_{h(a)}$. $\forall x, \varphi_{h(a)}(x) = \varphi_{h(x)}(x) + 1$, soit $x = a$ on obtient $\varphi_{h(a)}(a) = \varphi_{h(a)}(a) + 1$, puis $0 = 1$ c'est absurde (ça marche car tout est total.) □

Clôture

Clôture 1. Soient P et Q semi-décidables, alors $P \cap Q$ et $P \cup Q$ sont récursivement énumérables.

Clôture 2. Si $P(x,y)$ est semi-décidable, alors $S(y) = \exists x : p(x,y)$ est semi-décidable.

Preuve.

- Pour $P \cap Q$: soit $P = \text{dom } f$ et $Q = \text{dom } g$. $x \in P \cap Q$, $x \in P \cap Q \Leftrightarrow f(x)\downarrow \wedge g(x)\downarrow \Leftrightarrow (f+g)(x)\downarrow$ par exempe, donc $P \cap Q = \text{dom}\,(f+g)$
- Pour $P \cup Q$: soit $P = \text{Im } f$ et $Q = \text{Im } g$, f et g sont récursives totales. On peut énumérer $P \cup Q$: $P \cup Q = \text{Im } h$, où $h(2\,m) = f(m)$ et $h(2\,m+1) = g(m)$.
- $S(y) \Leftrightarrow \exists x : P(x,y) \overset{\text{forme normale}}{\Leftrightarrow} \exists x : \exists t : R(t,x,y)$, avec R recursif primitif.

  $S(y) \Leftrightarrow \mu.z : R(\text{ex}_2(z), \text{ex}_3(z), y) \Leftrightarrow \mu.z : R(\text{ex}_2(z), \text{ex}_3(z), y)\downarrow \Leftrightarrow f(y)\downarrow \Leftrightarrow y \in \text{dom } f$. □

Exo. Les propriétés suivantes sont récursivement énumérables :

1. $\varphi_x(12) = 13$
2. Existent 3 y premiers tels que $\varphi_x(y)\downarrow$
3. etc.

Théorème (Matiassevitch). $P(x)$ est RE $\Leftrightarrow$ pour un polynome φ on a $(P(\bar{x}) \Leftrightarrow \exists \bar{y} : \varphi(\bar{x}, \bar{y}) = 0)$

Corollaire. Soit ψ un polynome aux coefficients entiers. La propriété $\text{Dioph}(\psi) \Leftrightarrow \exists \bar{x} : \psi(\bar{x}) = 0$ est indécidable.

Preuve. Soit $A = \{x : \varphi_x \neq \bot\}$

A est indécidable (Rice).

A est récursivement énumérable : $A(x) \Leftrightarrow \exists y : \text{arret}(x, y)$.

Par le théorème précédent, $A(\bar{x}) \Leftrightarrow \exists y : \varphi(\bar{x}, y) = 0 \Leftrightarrow \text{Dioph}(\varphi(\bar{x}, \cdot))$

$A \leqslant_m \text{Dioph}$, or A est indécidable, donc Dioph est indécidable. □

Indécidabilité des divers problèmes informatiques

Grammaires de type 0

Grammaires de type 0, ie avec des règles $\alpha \to \beta$, avec α, $\beta \in (V \cup T)^*$ quelconques. Problème MotGr0 : est-ce que $S \overset{*}{\Longrightarrow} w$ dans G ? On va montrer que arret $\leqslant_m$ MotGr0.

Idée. On veut décider arret(x, y). Pour la machine de Turng x, on fera une grammaire qui la simule.

$$\begin{aligned} \text{une config de MT} &\Leftrightarrow \text{un mot} \\ \text{une instruction de MT} &\Leftrightarrow \text{une/plusieurs règles de } G \\ \text{un calcul de MT} &\Leftrightarrow \text{une dérivation dans } G \\ \text{arret}(x, y) &\Leftrightarrow \text{mot dans } G \end{aligned}$$

2013-12-05

Notre problème. Étant donné une grammaire de type zéro G, est-ce que $S \overset{*}{\Longrightarrow} w$?

On va simuler une machine de Turing dans une grammaire (on utilise deux symboles particuliers $\triangleright$ et $\triangleleft$ pour le début et la fin du mot).

	MT M	**Grammaire G**
sémantique	w_g, q, w_d	$\triangleright w_g\, q\, w_d \triangleleft$
fonctionnement	$p \xrightarrow{a/b,-} q$	$p\, a \to q\, b$
	$p \xrightarrow{a/b,\leftarrow} q$	$\forall x \in \Sigma, x\, p\, a \to q\, x\, b$
	$p \xrightarrow{a/b,\rightarrow} q$	$p\, a \to b\, q$
		$\triangleright \to \triangleright 0$
		$\triangleleft \to 0 \triangleleft$
initialisation	$q_0, 0\, 1^y$	$S \to \triangleright q_0\, 0\, 1^y \triangleleft$
fin	$\forall q, a$ d'arrêt	$q\, a \to Z$
		$\forall x \in \Sigma, x\, Z \to Z$
		$\forall x \in \Sigma, x\, Z \to Z$

Tableau 9. Correspondance MT-Grammaire

1. On représente une configuration de MT par une « configuration » de G.
2. On traduit chaque instruction de MT par une ou pluseurs règles de G qui font la même chose.
3. Quel est le lien entre M et G ?

   Lemme.

   $$w_g, p, w_d \xrightarrow[\text{MT}]{*} v_g, q, v_d \;\Leftrightarrow\; \triangleright w_g\, p\, w_d \triangleleft \overset{*}{\underset{G}{\Longrightarrow}} \triangleright v_g\, q\, v_d \triangleleft$$

4. On réduit le problème d'arrêt en codant la configuration initiale et la terminaison.

Lemme. $\text{Arret}(x,y) \Leftrightarrow S \overset{*}{\Longrightarrow} \triangleright\, Z \,\triangleleft$ pour la grammaire correspondant à lamachine de Turing de numéro x et sur l'entrée y.

On a donc montré $\text{Arret} \leqslant_m \text{MotGr0}$, d'où :

Théorème. Le problème de savoir si un mot appartient au langage engendré par une grammaire de type 0 est indécidable.

Théorème (en exo). Un langage est engendré par une grammaire de type $0 \Leftrightarrow$ il est récursivement énumérable.

Machines à deux piles

Prop. L'arrêt est indécidable pour les machines à deux piles.

Preuve. On simule une MT par une machine à deux piles (la première contient w_d, la seconde w_g, rien d'extraordinaire, fait en TD). □

Intersection de grammaires hors-contexte

Théorème. Étant donné deux grammaires hors-contexte, il est indécidable de savoir si $L(G_1) \cap L(G_2) = \varnothing$.

On va pour le montrer réduire le problème d'arrêt à deux compteurs vers le problème d'intersection non vide de deux langages HC.

1. On code une configuration de machine à deux compteurs par un mot.

$$\begin{aligned} \text{config}\, C = q, m, n \;\; &\mapsto \;\; \vec{c} = \triangleright 1^m\, q\, 2^n \triangleleft \\ &\qquad\;\; \vec{c}^{\,-1} = \triangleleft 2^n\, q\, 1^m \triangleright \end{aligned}$$

2. On code une instruction de machine à deux compteurs : pour chaque instruction i, on veut deux langages hors contexte $L_i^{\rightarrow\leftarrow}$ et $L_i^{\leftarrow\rightarrow}$ tel que :

$$c_1 \xrightarrow{i} c_2 \;\; \Leftrightarrow \;\; \begin{cases} \vec{c}_1\, \vec{c}_2{}^{-1} \in L_i^{\rightarrow\leftarrow} \\ \vec{c}_1{}^{-1}\, \vec{c}_2 \in L_i^{\leftarrow\rightarrow} \end{cases}$$

   Les instructions de machines à deux compteurs sont parmi :

$$\begin{aligned} p \xrightarrow{\text{incr } m} q \quad & L = \{ \triangleright 1^m\, p\, 2^n \triangleleft \triangleleft 2^n\, q\, 1^{m+1} \triangleright \} \in \text{HC} \\ p \xrightarrow{\text{decr } m} q \quad & \text{etc.} \\ p \xrightarrow{\text{incr } n} q \quad & \text{etc.} \\ p \xrightarrow{\text{decr } n} q & \\ p \to (m = 0\,?\, q : r) & \end{aligned}$$

   (Exercice : finir la traduction)

3. On pose maintenant :

$$\begin{aligned} L^{\rightarrow\leftarrow} &= \bigcup_{i \in \Delta} L_i^{\rightarrow\leftarrow} \\ L^{\leftarrow\rightarrow} &= \bigcup_{i \in \Delta} L_i^{\leftarrow\rightarrow} \end{aligned}$$

4. Soit $\gamma = c_1 \to c_2 \to c_3 \to \cdots \to c_n$ une séquence de configurations de la machine à deux compteurs. On pose $w(\gamma) = \vec{c}_1\, \vec{c}_2{}^{-1}\, \vec{c}_3\, \vec{c}_4{}^{-1} \cdots \vec{c_n}^{\,(-1)^{n+1}}$

Lemme. Soit $w(\gamma)$ de longueur impaire. Alors

$$\gamma \text{ est un calcul} \Leftrightarrow w(\gamma) \in \vec{c_1}\,(L^{\leftarrow\rightarrow})^* \cap (L^{\rightarrow\leftarrow})^*\,\vec{c_n}$$

$$\begin{cases} \gamma \text{ est un calcul} \\ \gamma \text{ commence en } (q_0, m, n) \\ \gamma \text{ s'arrête} \end{cases} \Leftrightarrow w(\gamma) \in (\triangleright 1^m\, q_0\, 2^n \triangleleft)(L^{\leftarrow\rightarrow})^* \cap (L^{\rightarrow\leftarrow})^* \overrightarrow{\text{stop}}$$

$$\text{où } \overrightarrow{\text{stop}} = \bigcup_{q \text{ état d'arrêt}} \triangleright 1^*\, q\, 2^* \triangleleft$$

Donc $\text{Arret}_{2\,\text{compteurs}}(\text{prog}, m, n)$ s'arrête (en nombre pair d'opérations) $\Leftrightarrow L_1 \cap L_2 \neq \varnothing$ pour les deux langages L_1 et L_2 définis ci-dessus.

Exercice. Terminer la réduction pour tenir compte de la longueur paire.

$$\text{Arret}_{2\text{ compteurs}}(\text{prog}, m, n) \;\leqslant_m\; L_1 \cap L_2 \neq \varnothing$$

Autres preuves d'indécidabilité en Janvier ! (lors des exposés)

« Problème de codage »

On a un problème : « $B \subset \mathcal{U}$, étant donné $x \in \mathcal{U}$, $x \in B$? »

On a pris $\mathcal{U} = \mathbb{N}^k$, et on a défini puis étudié les problèmes décidables et semi-décidables.

Pour un autre $\mathcal{U}$, on peut tout coder sur $\mathbb{N}$ (ou $\mathbb{N}^k$) (cf Gödelisation).

Exemple. Soit $\mathcal{U} = \Sigma^*$, on dit qu'un langage $L \subset \Sigma^*$ est décidable (resp. semi-décidable, indécidable) si $\{\langle w \rangle\,; w \in L\}$ est décidable (resp. semi-décidable, indécidable).

Cette définition ne dépend pas du codage (choix de $\langle w \rangle$, raisonnable SVP).

Soit $\mathcal{U} = \Sigma^*$ (encore...). On peut définir L décidable (ou semi-décidable) directement par MT.

Déf. Soit M une MT, $q_0, q_Y, q_N \in Q$. On dit que M décide L si :

$$\begin{aligned} \forall w \in L \quad & q_0, \triangleright w \triangleleft \xrightarrow{*} u, q_Y, v \\ \forall w \notin L \quad & q_0, \triangleright w \triangleleft \xrightarrow{*} u, q_N, v \end{aligned}$$

Théorème. Les deux notions de langage décidable sont équivalentes.

Idée de preuve. Blah (comprendre : on transcrit le mot en numéro de Gödel puis on exécute l'autre machine, et dans l'autre sens ça marche aussi.) □

IV. Complexité

On a un problème : « $B \subset \mathcal{U}$, étant donné $x \in \mathcal{U}$, $x \in B$? ». On suppose qu'il est décidable.

Question. Combien de temps/de mémoire faut-il pour le décider ? (en fonction de la taille de x)

Les réponses dépendent du codage de x et du modèle de calcul.

(cf. Complexity Zoo)

On va coder $x \in \mathcal{U}$ par un mot en Σ^* et les nombres en système binaire.

Deux questions.

1. *Borne sup.* Pour P un problème donné, montrer qu'il y a un algorithme qui prend en temps ou en espace un $O(f(n))$. (question facile)

2. *Borne inf.* Pour P un problème donné, montrer qu'il n'y a pas d'algorithme en $O(f(n))$. (question plus difficile)

Déf. Soit M une machine, $x \in \Sigma^*$ une entrée. $t_M(x)$ est le nombre de pas de calcul de M sur x. $t_M(n) = \max\{t_M(x)\,;\, x \in \Sigma^n\}$.

Exemple. Palindromes sur $\{a, b, c\}$.

Prop 1. Toute machine de Turing (avec un ruban) qui décide le problème du palindrome a un temps de calcul $\Omega(n^2)$. (facile à faire en $O(n^2)$).

Preuve borne inf. Soit M une machine à un ruban qui décide le problème du palindrome.

1. Posons $u(w) = \underbrace{w}_{\text{taille } n/4}\ \underbrace{c\,c\,c\cdots c\,c}_{\text{taille } n/2}\ \underbrace{w^{-1}}_{\text{taille } n/4}$, avec $w \in \{a, b\}^*$ et w^{-1} le mot miroir de w.

   Idée. M doit déplacer $n/4$ bits d'information à la distance $n/2$, ce qui nécessite $O(n^2)$ pas.

2. Séquence de croisement $c_i(w)$: on prend le calcul de M sur $u(w)$. On note tous les états de M à la case du i-ème c dans le mot quand elle passe de la position i (du i-ème c) à la position $i+1$ (au $i+1$-ème c) ou de la position $i+1$ à la position i. Cette séquence d'états est notée $c_i(w)$. (faire un dessin pour comprendre) (explications peu claires)

3. **Lemme principal.** Si $w \neq v$ (w et v de même longueur $n/4$), alors pour tous i, j, on a $c_i(w) \neq c_j(v)$.

   Preuve du lemme. La machine M accepte ce mot, le calcul est accepteur. Supposons qu'on ait $c_i(w) = c_j(v)$, c'est-à-dire que la machine passe par la même succession d'états à la position i dans un mot et j dans l'autre. Alors on voit en examinant le calcul de la machine sur le mot $w\,c^i\,c^{n/2-j}\,v$ est une combinaison des deux calculs sur $u(w)$ et $u(v)$, et donc la machine acceptera le mot $w\,c^i\,c^{n/2-j}\,v$. (faire des dessins de machines pour comprendre) Or ce mot n'est pas un palindrome, c'est donc impossible. □

4. Pour chaque $w \in \{a, b\}^{n/4}$, soit $c(w) = c_i(w)$ tel que $|c_i(w)|$ soit minimal. Soit $\Lambda_n = \max\left\{|c(w)|\,;\, w \in \{a, b\}^{n/4}\right\}$. $c(w)$ est une séquence d'états de M de longueur $\leqslant \Lambda_n$, et par le lemme précédent, des mots w différents impliquent des $c(w)$ différents

   Soit B le nombre d'états de M. On a :

$$\begin{aligned} \text{nombre de mots } w &\leqslant \text{nombre de } c(w) \text{ distincts possibles} \\ 2^{n/4} &\leqslant B^{\Lambda_n+1} \\ \frac{n}{4}\log 2 &\leqslant (\Lambda_n+1)\log B \\ \alpha\, n &\leqslant \Lambda_n \end{aligned}$$

5. Soit w tel que $|c(w)| = \Lambda_n \geqslant \alpha\, n$. Alors sur ce mot, $\forall i, |c_i(w)| \geqslant \alpha\, n$.

   Lemme. $t_M(w) \geqslant \sum_{i=1}^{n/2} |c_i(w)|$. (en effet, chaque élément d'un $c_i(w)$ correspond à un pas de calcul).

   On en conclut donc : $t_M(w) \geqslant \sum_{i=1}^{n/2} |c_i(w)| \geqslant \frac{n}{2}\,\alpha\, n = \Omega(n^2)$. Or w a la longueur n, donc $t_M(n) \geqslant t_M(w)$. □

Prop 2. On peut décider le problème du palindrome en $O(n)$ (borne sup) avec une MT avec deux rubans.

Preuve. Il faut copier le mot d'un ruban à l'autre, en rembobiner un, puis avancer sur un ruban en reculant sur l'autre ; ces deux opérations prennent un temps linéaire, d'où une complexité $3\,n = O(n)$. □

Déf. $L \subset \text{Time}(f(n))$ s'il existe une machine de Turing M à plusieurs rubans qui décide L et telle que $t_M(n) = O(f(n))$.

Exemples. Palindromes $\in \text{Time}(n)$.

2013-12-12

On a vu que Palindrome se fait en $O(n)$ sur deux rubans et en $\Omega(n^2)$ sur un seul.

Prop. Si on peut décider L en $f(n)$ sur k rubans, on peut le faire en $O(f(n)^2)$ sur 1 ruban.

> **Preuve.** On a une machine à k rubans, on veut la simuler sur un seul. Pour cela, on définit un nouvel alphabet $\Sigma' = \mathcal{P}(\{1, ..., k\}) \times \Sigma^k$, qui permet d'encoder les k rubans sur un seul (la première piste enregistre la position des têtes des k rubans, les pistes suivantes le contenu des différents rubans).
>
> Un pas sur k rubans correspond à $f(n)$ pas sur la machine à un ruban (on fait un balayage $\rightarrow$ pour lire puis $\leftarrow$ pour écrire). $\square$

Remarque 1. Un calcul en temps t est confiné à $2\,t$ cases du ruban.

Remarque 2. Typiquement, $t_M(n) = \Omega(n)$.

On utilisera dorénavant les machines de Turing à k rubans comme référence.

Prop (théorème d'accélération). S'il existe M_1 qui décide L en $f(n)$, alors $\forall \varepsilon > 0, \exists M_2$ qui décide L en $\varepsilon\, f(n) + O(n)$.

> **Preuve.** On a une machine M_1 sur Σ. On fabrique une machine M_2 sur Σ^k, où l'on regroupe les symboles du mot par paquets de taille k. (ex : $a\,b\,c\,a\,a\,a\,b\,b\,b \rightarrow (a\,b\,c)(a\,a\,a)(b\,b\,b)$). Un pas de la machine M_2 décrit exactement k pas de la machine M_1. Soit on reste sur le même bloc de taille k, soit on se déplace d'au plus un bloc. On va donc k fois plus vite. $\square$

Déf. $\text{Time}(f(n)) = \{L\colon L$ peut être décidé par une MT déterministe à plusieurs rubans avec $t_M(n) = O(f(n))\}$

Classes importantes.

$$P = \bigcup_{k \in \mathbb{N}} \text{Time}\,(n^k) \quad \text{(temps déterministe polynomial ; pb dit « tractable »)}$$
$$\text{ExpTime} = \bigcup_{k \in \mathbb{N}} \text{Time}\left(2^{n^k}\right)$$

Remarque. Ces classes ne dépendent pas du choix du mode de calcul raisonnable (un ruban, dix rubans, RAM, etc.)

Question. Est-ce que $P \neq \text{ExpTime}$? Oui.

Théorème de hiérarchie. Si $g = o(f)$ et f est sympathique, alors $\text{Time}(g) \subsetneq \text{Time}(f)$. (voir TD)

Pour des milliers de problèmes L utiles, on est sûr que $L \in \text{ExpTime} \setminus P$, mais on ne sait pas le prouver.

On cherche une méthode pour s'assurer que $L \notin P$, mais pas forcément pour le démontrer.

Deux notions. 1) réduction polynomiale et 2) machines de Turing et classes de complexité non-déterministes.

Réduction polynomiale

Déf. $L \leqslant M$ si $\exists h$ calculable en temps polynomial telle que $\forall w$, $w \in L \Leftrightarrow h(w) \in M$

Prop. Si $L \leqslant M$ et $M \in P$, alors $L \in P$.

Corollaire. Si $L \leqslant M$ et $L \notin P$ alors $M \notin P$.

MT non-déterministe

Déf. C'est la définition usuelle d'une MT, sauf que l'on peut avoir plusieurs instructions pour le même (q, a) de départ.

Déf. Soit L un langage, M une MT non-déterministe, avec un état d'arrêt q_Y. On dit que M reconnait L si :

- pour tout $w \in L$, il existe un calcul de M qui s'arrête en q_Y ;
- pour tout mot $w \notin L$, il n'existe pas de tel calcul ;
- pour tout mot w, tout calcul de M s'arrête.

Exemple. $L =$ nombre composé (ie non premier) en base 10. On peut construire une MT non-déterministe qui le décide en $O(n^2)$ (où n est le nombre de chiffres qui compose le nombre à tester) :

1. On écrit deux nombres quelconques de $\leqslant n$ chiffres (différents de 1 et de 0) sur les rubans 2 et 3 (le ruban 1 contient le nombre dont on veut tester la non-primalité).
2. Partie déterministe : on vérifie si par hasard on aurait pas $x\, y = w$ (on calcule $x\, y$ sur le ruban 4 puis on compare au ruban 1). Si $x\, y = w$ on s'arrête sur l'état q_Y, sinon on s'arrête sur q_N.

Déf. Soit M une machine non-déterministe. On pose :

$$\begin{aligned} t_M(w) &= \max\{\text{nombre de pas de } C, \text{ avec } C \text{ calcul sur } W\} \\ t_M(n) &= \max\{t_M(w)\,;\, w \in \Sigma^n\} \end{aligned}$$

Déf.

$$\mathrm{NTime}(f(n)) = \{L : L \text{ est reconnu par } M \text{ une MT non-dét et avec } t_M(n) = O(f(n))\}$$

Classes intéressantes.

$$\begin{aligned} \mathrm{NP} &= \bigcup_{k\in\mathbb{N}} \mathrm{NTime}\,(n^k) \\ \mathrm{NExpTime} &= \bigcup_{k\in\mathbb{N}} \mathrm{NTime}\left(2^{n^k}\right) \end{aligned}$$

Prop.

$$\mathrm{Time}\,(f(n)) \overset{(1)}{\subseteq} \mathrm{NTime}\,(f(n)) \overset{(2)}{\subseteq} \mathrm{Time}\left(2^{f(n)}\right)$$

Dém.

1. Une machine déterministe est aussi une machine non déterministe : la machine déterministe reconnaissant L reconnait aussi L en tant que machine non déterministe.
2. Si M non-déterministe reconnait L en $O(f(n))$, sur w donné on parcourt l'arbre de tous les calculs de M. Sur chaque configuration, il y a un certain nombre de choix à faire ; l'arbre de décision est de profondeur $O(f(n))$, avec une arité maximale de $|\Delta|$ (approximation très grossière). La taille de l'arbre est donc $|\Delta|^{f(n)} = 2^{k[f(n)]}$. On parcourt cet arbre en DFS.

On sait donc.

$$P \subset \mathrm{NP} \subset \mathrm{ExpTime} \subset \mathrm{NExpTime}$$

Conjecture.

$$P \neq \mathrm{NP}$$

Déf. Soit C une classe de complexité. On dit que :

- L est C-easy si $L \in C$.
- L est C-hard si $\forall M \in C,\ M \leqslant L$
- L est C-complete si L est C-easy et C-hard.

Fait. Beaucoup de problèmes algorithmiques utiles sont NP-complets (on va le démontrer).

Prop. $P = \mathrm{NP} \Leftrightarrow \exists L \in P \cap \mathrm{NPComplet}$

Preuve. Si $P = \mathrm{NP}$, alors $\forall L \in P$, L est NP-complet (tout problème de P se réduit à tout autre problème de P).

Si $\exists L \in P \cap \mathrm{NPComplet}$, alors $\forall M \in \mathrm{NP}$ on a $M \leqslant L$ (car L est NP-Complet), donc $M \in P$, donc on a $\mathrm{NP} \subseteq P$. $\square$

Problème. On s'intéresse au problème SAT : on a une formule de logique propositionnelle f, est-elle satisfiable ?

Exemple. $(X \vee Y \vee Z) \wedge (\bar{X} \vee \bar{Y} \vee \bar{Z})$ est satisfiable en prenant $X = \mathrm{true}, Y = \mathrm{true}, Z = \mathrm{false}$.

Remarque. Il y a une science avancée pour écrire des logiciels résolvant SAT (SAT-solvers), et on a maintenant des logiciels très efficaces... mais pas polynomiaux.

Théorème (Cook-Levin). SAT est NP-complet.

Remarque. Méthode à essayer pour un langage $L \in \mathrm{NP}$: on réduit $L \leqslant \mathrm{SAT}$ ($w \in L \Leftrightarrow h(w)$ est une formule satisfiable), puis on utilise un SAT-solver efficace.

Preuve du théorème.

1. $\mathrm{SAT} \in \mathrm{NP}$: en effet, il est trivial pour une machine non-déterministe de résoudre SAT. D'abord, on devine la valuation à utiliser, puis ensuite on testela formule sur cette valuation (si elle est vrai on s'arrête sur q_Y, sinon sur un autre état).

2. SAT est NP-hard : preuve par table de calcul.

   Soit $L \in \mathrm{NP}$, il est reconnu par $\mathcal{M}$ non-déterministe avec $t_M(n) \leqslant c\, n^k$.

   $w \in L \Leftrightarrow \mathcal{M}$ a un calcul accepteur sur w. On décrit ce calcul par une table :

$t=0$	$0\,0 \cdots 0$	$w_1 \cdots w_n$	$\cdots$	q_0 k_0
$t=1$	$\cdots$			q_1 k_1
	$\vdots$			
$t=M$	$\cdots$			q_M k_M

Tableau 10. Les mots du ruban tiennent tous dans un espace $2\,M$; w est positionné en M

   $t_M(n) \leqslant c\ n^k$, machine à un ruban ; on prend $M = c\ n^k$. On rajoute une nouvelle transition :

$$\forall a, q_Y, a \xrightarrow{\text{ne bouge pas}} q_Y, a$$

Si la machine accepte, elle sera en état q_Y à la dernière ligne.

$$\begin{aligned} w \in L &\Leftrightarrow \mathcal{M} \text{ a un calcul accepteur sur } w \\ &\Leftrightarrow \text{il existe une table de calcul qui décrit un calcul accepteur :} \\ &\quad \text{sa taille est } M = c\,|w|^k ; \\ &\quad \text{ligne 0 : le ruban contient } 0^M w\, 0^*,\ q = q_0,\ \text{pos} = M\ ; \\ &\quad \text{ligne } M : q = q_Y\ ; \\ &\quad \forall k,\ \text{on passe de la ligne } k \text{ à la ligne } k+1 \text{ par une transition de } \mathcal{M} \end{aligned}$$

On code ce tableau avec une formule booléenne, avec beaucoup de variables :

$$\begin{aligned} \text{lettre}_{i,j,a} &= \begin{cases} \text{true} & \text{si la case } i \text{ du ruban au temps } j \text{ contient } a \\ \text{false} & \text{sinon} \end{cases} \\ &\quad (\text{il y a } M^2\,|\Sigma| = O(n^{2k}) \text{ telles variables, ok car polynomial}) \\ \text{etat}_{j,q} &= \begin{cases} \text{true} & \text{si la machine est dans l'état } q \text{ au temps } j \\ \text{false} & \text{sinon} \end{cases} \\ \text{pos}_{i,j} &= \begin{cases} \text{true} & \text{si au temps } j \text{ la tête se trouve en position } i \\ \text{false} & \text{sinon} \end{cases} \\ &\quad (\text{il y a } M^2 = O(n^{2k}) \text{ telles variables, ok}) \end{aligned}$$

On écrit une formule booléenne f qui indique que le tableau ainsi décrit représente un calcul accepteur sur le mot w.

$$\begin{aligned} f &= f_{\text{mutex}} \wedge f_{\text{init}} \wedge f_{\text{fin}} \wedge f_{\text{trans}} \\ f_{\text{mutex}} &= \text{tout est correct (une seule lettre par case, un et un seul état/pos par ligne)} \\ f_{\text{init}} &= \text{la ligne 0 contient ce qu'il faut (voir ci-dessus)} \\ f_{\text{fin}} &= \text{la ligne finale est sur } q_Y \\ f_{\text{trans}} &= \text{le passage d'une ligne à la suivante représente une transition de } \mathcal{M} \end{aligned}$$

Il est clair que toutes ces propositions peuvent être écrites en formules booléennes, et il se trouve que leurs tailles sont toujours polynomiales... Par exemple :

$$\begin{aligned} \text{1 et 1 seule pos par ligne} &= \left(\bigwedge_j \bigvee_i \text{pos}(i,j) \right) \wedge \left(\bigwedge_j \bigwedge_{i_1 \neq i_2} \neg(\text{pos}(i_1,j) \wedge \text{pos}(i_2,j)) \right) \\ \text{init} &= \text{lettre}(M,0,w_1) \wedge \text{lettre}(M+1,0,w_2) \wedge \cdots \\ \text{fin} &= \text{etat}(M, q_Y) \end{aligned}$$

Voilà. La formule f est de taille $O(n^{5k})$ et convient. □

(exo : essayer d'écrire f_{trans})

Prop/Méthode. Si L est NP-complet et $L \leqslant M$, avec $M \in \text{NP}$, alors M est NP-complet.

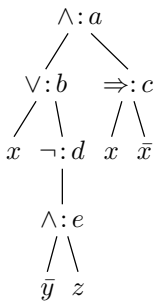
Problème. 3-SAT : étant donné une formule propositionnelle en 3CNF, est-elle satisfiable ? (3CNF : forme normale conjonctive où chaque clause contient 3 lettres ; par exemple : $(x \vee y \vee z) \wedge (\bar{x} \vee \bar{y} \vee u) \wedge (\bar{u} \vee x \vee z)$)

Théorème. 3-SAT est NP-complet.

Idée de preuve.

1. 3-SAT $\in$ NP
2. SAT $\leqslant$ 3-SAT : pour chaque formule booléenne f on construit une autre formule f_3 en 3CNF.

Construction. On a une formule f, on dessine son arbre syntaxique :



On donne des noms aux sous-arbres (a, b, c, d, e), et on peut écrire des formules :

$$\begin{aligned} a &\Leftrightarrow b \wedge c \\ b &\Leftrightarrow x \wedge d \\ c &\Leftrightarrow x \Rightarrow \bar{x} \\ d &\Leftrightarrow \neg e \\ e &\Leftrightarrow \bar{y} \wedge z \end{aligned}$$

On a donc :

$$f \Leftrightarrow (a \Leftrightarrow b \wedge c) \wedge (b \Leftrightarrow x \wedge d) \wedge (c \Leftrightarrow (x \Rightarrow \bar{x})) \wedge (d \Leftrightarrow \neg e) \wedge (e \Leftrightarrow \bar{y} \wedge z) \wedge a$$

On demande que la racine soit vraie ; on transcrit chaque petite formule en 3CNF (assez facile).

Rappel (à quoi bon ?). L est NP-complet $\Rightarrow$ sans doute, L est difficile.

Formellement. L NP-complet et $P \neq \mathrm{NP} \Rightarrow L \notin P$.

`2013-12-19 (dernier cours !)`

On a nos deux classes : $\mathrm{Time}(f(n))$ et $\mathrm{NTime}(f(n))$ correspondant respectivement à des machines déterministes et non déterministes (à k rubans).

On a ensuite défini $P = \bigcup_{k \in \mathbb{N}} \mathrm{Time}(n^k)$, $\mathrm{ExpTime} = \bigcup_{\kappa \in \mathbb{N}} \mathrm{Time}\left(2^{n^k}\right)$, $\mathrm{NP} = \bigcup_{k \in \mathbb{N}} \mathrm{NTime}(n^k)$ et $\mathrm{NExpTime} = \bigcup_{k \in \mathbb{N}} \mathrm{NTime}\left(2^{n^k}\right)$.

On sait (cf TD de la semaine dernière) que $P \subsetneq \mathrm{ExpTime}$ et $\mathrm{NP} \subsetneq \mathrm{NExpTime}$.

Problème : $P = \mathrm{NP}$? Conjecture $P \neq \mathrm{NP}$.

On a définit la réduction polynomiale $\leqslant$ ou $\preccurlyeq$.

On a définit : L est NP-complet si $L \in \mathrm{NP}$ (NP-facile) et $\forall M \in \mathrm{NP}$, $M \preccurlyeq L$ (NP-difficile).

On a vu que SAT et 3SAT est NP-complet.

Exemple de problème NP-complet. Problème STABLE : étant donné un graphe non-orienté $G = (V, E)$ et $K \in \mathbb{N}$. Existe-t-il un sous ensemble $M \subset V$ stable tel que $|M| = k$?

Déf. M est stable si $\forall u \neq v \in M$, $(u, v) \notin E$, ie. aucun deux sommets ne sont connectés.

Preuve de l'exemple.

1. Montrons que $\mathrm{STABLE} \in \mathrm{NP}$:

   On choisit de manière non-déterministe $M \subset V$

On vérifie déterministement (temps polynomiale) que M est stable et $|M| = k$.

2. Montrons que STABLE est NP-difficile : on va réduire 3SAT $\preccurlyeq$ stable :

   On a une formule φ sous forme 3-CNF. On cherche à construire un graphe.

   $$\varphi = (x \vee y \vee \bar{z}\,) \wedge (\bar{x} \vee u \vee \bar{y}\,) \wedge (\bar{u} \vee z \vee x)$$

   Pour chacune des k clauses, on crée un triangle du type par exemple $\{x-y, y-\bar{z}\,, x-\bar{z}\,\}$, et on construit le graphe G contenant tous ces triangles plus les arrêtes traduisant la non collision entre les x et les $\bar{x}$. On cherche un stable M de k sommets dans ce graphe. Cf exemple sur papier (112523).

   Si un tel stable existe, chaque sommet de M correspond à une clause vérifier. Il suffit de reconstruire la valuation correspondantes.

   (on prouve que φ est satisfiable $\Leftrightarrow$ G possède un stable de k sommets) $\square$

Remarque. Dans la vie de chercheur, on remarque très rapidement quand on a un problème L qu'il est dans NP. Si on ne trouve pas d'algo P, on essaye de réduire 3SAT $\leqslant L$ ou STABLE $\leqslant L$ ou 3Match $\leqslant L$ ou SubSetSum $\leqslant L$, etc.

Remarque. On a introduit nos classes pour les langages (problèmes de décision). Or STABLE serait plus naturellement vu comme un problème de décision : étant donné $G = (V, E)$, quelle est la taille maximale d'un stable (problème MST).

Prop. On sait résoudre MST en temps polynomial $\Leftrightarrow$ on sait résoudre STABLE en temps polynomial.

Preuve. $\Rightarrow$ trivial, $\Leftarrow$ on cherche le meilleur k par recherche dichotomique (par exemple)

Définition de la classe NP par « vérificateur »

Théorème. $L \in$ NP $\Leftrightarrow$ il existe une relation $R(x, y)$ sur deux mots telle que :

1. $x \in L \Leftrightarrow \exists y \colon R(x, y)$
2. Une MT peut décider si $R(x, y)$ en temps $|x|^k$.

Ce théorème fournit une définition alternative de NP, qui est tout-à-fait intéressante puisqu'il n'y a pas besoin de s'intéresser aux machines déterministes.

Preuve.

- Sens $\Leftarrow$: trouvons une MT B non-déterministe polynomiale pour décider si $x \in L$ (A est la MT déterministe qui reconnait R)

  1. deviner y (de longueur $\leqslant c\,|x|^k$)
  2. tester si A accepte (x, y)

  B accepte $x \Rightarrow \exists y$ tel que A accepte $(x, y) \Rightarrow \exists y \colon R(x, y) \Rightarrow x \in L$

  $x \in L \Rightarrow \exists y \colon R(x, y) \Rightarrow \exists y \colon A$ accepte $(x, y) \Rightarrow \exists y \in A$ accepte (x, y) en $c\,|x|^k \Rightarrow \exists y$ tel que A accepte (x, y') en $c\,|x|^k$ où y' est les $c\,|x|^k$ premières lettres de $y \Rightarrow \exists y'$ court tel que A accepte $(x, y') \Rightarrow B$ accepte x

- Sens $\Rightarrow$: $L \in$ NP donc L est reconnu par une MT non-déterministe B en temps $c\,|n|^k$.

$x \in L \Leftrightarrow B$ a un calcul accepteur sur x de longueur $\leqslant c\,|n|^k \Leftrightarrow \exists (i_1, ..., i_m)$ une séquence d'instructions de B ($m \leqslant c\,|n|^k$) telle que en appliquant $i_1, ..., i_m$ à x on obtient un calcul accepteur $\Leftrightarrow \exists y\colon R(x, y)$, on peut facilement tester R en temps polynomial. □

Remarques.

1. Toutes les classes de lengages déterministes Time($f(n)$), P, ExpTime, etc. sont closes par complémentaire

2. Ce n'est pas sûr pour les classes non-déterministes.

Déf. On définit coNP par $L \in \text{coNP} \Leftrightarrow \bar{L} \in \text{NP}$.

Conjecture. $\text{coNP} \neq \text{NP}$.

Exemple. Contr = {formules logiques contradictoires} et Taut = {tautologies} sont deux problèmes coNP-complets.

Complexité en espace

Première tentative de définition. Soit M une MT (déterministe ou non-déterministe), avec ruban d'entrée en *lecture seule*. On pose $S_M(x) = \{$nombre de cases du ruban visitées par M sur l'entrée $x\}$, et $S_M(n) = \max_{x \in \Sigma^n} S_M(x)$.

On pose ensuite. $\text{Space}(f(n)) = \{L\colon$ décidé par une MT déterministe avec $S_M(n) = O(f(n))\}$, $\text{NSpace}(f(n)) = \{L\colon$ décidé par une MT non-déterministe avec $S_M(n) = O(f(n))\}$

Remarque. On a souvent avec cette définition $S_M(n) \geqslant n$.

Classes de complexité utile.

$$\begin{aligned} L &= \text{Space}(\log n) \\ \text{NL} &= \text{NSpace}(\log n) \\ \text{PSpace} &= \bigcup_{k \in \mathbb{N}} \text{Space}(n^k) \\ \text{NPSpace} &= \bigcup_{k \in \mathbb{N}} \text{NSpace}(n^k) \end{aligned}$$

On a des inclusions évidentes :

$$L \subset \text{NL} \subset \text{PSpace} \subset \text{NPSpace}$$

Prop. $\text{Time}(f(n)) \subset \text{Space}(f(n)) \subset \bigcup_{c \in \mathbb{N}} \text{Time}\big(2^{cf(n)}\big)$.

Preuve.

- Si M décide L en temps $O(f(n))$ elle utilise moins de $f(n)$ cases, d'où première inclusion

- Si M décide L en espace $O(f(n))$, alors M peut avoir sur x $|\Sigma|^{O(f(n))}$ configurations différentes. Une configuration de MT prenant un espace S est codée par (G_1, D_1, G_2, $D_2, ..., G_k, D_k, i, q$) (i est la position sur le mot d'entrée x), on a donc $|G_1| + |D_1| + \cdots + |G_k| + |D_k| \leqslant S$, $i \leqslant |x|$. On a $O(|\Sigma|^S S^{2k} |x||Q|) = |x| 2^{O(S)}$configurations possibles. □

Prop. $\text{NTime}(f(n)) \subset \text{NSpace}(f(n)) \subset ?$

Preuve. Première inculsion : pareil que précédemment. □

Objectif. Décrire les classes NSpace.

Parcours de graphe

Parenthèse. Atteignabilité dans les graphes : REACH. Étant donné G graphe orienté et $s, t \in V$ deux sommets, existe-t-il un chemin s vers t ? Ce problème est dans P, il est facile (cf BFS ou DFS).

Propriété. $\text{REACH} \in \text{NSpace}(\log n) = \text{NL}$

Preuve. On fait un DFS non déterministe où on s'arrête après n étapes (le compteur tient sur $O(\log n)$ cases ; enregistrer le sommet courant se fait aussi sur $O(\log n)$ cases).

Thérorème (Savitch). $\text{REACH} \in \text{Space}((\log n)^2)$

Preuve. On définit $R(x, y, m) \Leftrightarrow \exists$ chemin de x vers y de longueur $\leqslant 2^m$. Alors $\text{Reach} \Leftrightarrow R(s, t, \log n)$. Algorithme pour R en diviser-pour-régner :

Algorithme 3

fonction $R(x, y, m)$
  si $m = 0$ **return** $x = y \vee (x, y) \in E$
  sinon, **pour tout** $z \in V$
    si $R(x, z, m-1) \wedge R(z, y, m-1)$ **return** false
  return false

Preuve de correction. $R(x, y, m) \Leftrightarrow \exists$ un chemin $x \to \cdots \to y$ de longueur $\leqslant 2^m \Leftrightarrow \exists z$ tel que $x \to \cdots \to z$ et $z \to \cdots \to y$ sont deux chemins de longueur $\leqslant 2^{-m-1} \Leftrightarrow \exists z : R(x, z, m-1) \wedge R(z, y, m-1)$. □

On implémente cet algorithme récursif avec une pile d'appels qui prend $O((\log n)^2)$ et un ruban de travail qui prend probablement $O(\log n)$.

Le calcul de $R(s, t, \log n)$ utilisera $O(\log n \log n)$ cases, cqfd. □

Inclusions sympathiques

Théorème (Savitch ?). $\text{NSpace}(f(n)) \subset \text{Space}(f(n)^2)$ pour tout f « raisonnable ».

Preuve. On suppose $f(n) = \Omega(\log n)$ et $f(n)$ constructible en espace (cf. TD n°11).

Soit L reconnu par une machine A non-déterministe d'espace $c\,f(n)$, et qui à la fin efface tout et se met en état q_Y. Soit x une entrée. On construit un graphe $G = (V, E)$ de configurations, où :

- $V = \{\text{toutes les configurations de } A \text{ de taille } \leqslant c\,f(|x|) \text{ sur l'entrée } x\}$
- $E = \{(u, v) \text{ qui correspondent aux transitions de } A\}$

Remarque 1. $x \in L \Leftrightarrow \exists$ un chemin de s vers t dans G, où s est la configuration initiale et t est une configuration acceptrice $\Leftrightarrow \text{REACH}(G, s, t)$

Remarque 2. $|G| = 2^{k f(n) + \log n}$.

On applique l'algorithme de Savitch donné précédemment pour décider REACH(G, s, t) en espace déterministe $O((\log |G|)^2) = O((k\,f(n) + \log n)^2) = O(f(n)^2)$.

Important. Il ne faut pas écrire le graphe tout entier ! On calcule a la volée le fait de savoir $(a, b) \in E$.

□

Corollaire. $\text{NPSpace} = \text{PSpace}$.

Corollaire. $\text{NSpace}(f(n)) \subset \text{Space}(f(n)^2) \subset \text{Time}\left(2^{c f(n)^2 + \log n}\right)$

Propriété.

$$L \subset \text{NL} \subset P \subset \text{NP} \subset \text{PSpace} = \text{NPspace} \subset \text{ExpTime}$$

PSpace-complétude

Problème. QSAT = {formules booléennes quantifiées satisfiables}, où les formules sont de la forme :

$$\forall x_1, \exists y_1 \colon \forall x_2, \exists y_2 ... (\psi(\bar{x}\,, \bar{y}\,, \bar{z}\,))$$

où les $\bar{x}\,, \bar{y}\,, \bar{z}$ sont des vecteurs de variables booléennes.

Exemple. $\forall x, \exists y \colon (x \vee y \vee \bar{z}\,)$

Remarqe. Toute formule booléenne quantifiée est équivalente à une formule non quantifiée beaucoup plus longue :

$$\begin{aligned} \forall x, f(x,z) &\Leftrightarrow f(0,z) \wedge f(1,z) \\ \exists y \colon f(y,z) &\Leftrightarrow f(0,z) \vee f(1,z) \end{aligned}$$

Dernier théorème. QSAT est PSpace-complet.

Preuve.

1. QSAT $\in$ PSpace : on trouve un algorithme non-déterministe qui résoud QSAT en espace polynomial :

   a. deviner les valeurs de $\bar{z}$; les mettre dans φ

   b. évaluer φ en espace polynomial

   Algorithme 4

   fonction eval(φ)
   si pas de quantificateurs : **return** eval(ψ)
   si $\varphi = \forall x, \Theta$: **return** eval($\Theta|_{x=0}$) $\wedge$ eval($\Theta|_{x=1}$)
   si $\varphi = \exists x \colon \Theta$: **return** eval($\Theta|_{x=0}$) $\vee$ eval($\Theta|_{x=1}$)

   La profondeur de la pile d'appels correspond au nombre de quantificateurs. Cet algorithme est réalisable en temps polynomial.

2. PSpace-complétude de QSAT : difficile... $\forall L \in$ PSpace, on a $L \leqslant$ QSAT. Fait en TD.

```
Informations pratiques :

Sur le site : 3 examens, deux corrigés.

Soutenances : 20 min + 10 min, français par défaut mais anglais toléré ;

     slides obligatoires, tex+beamer conseillé, français ou anglais,

     slides à envoyer par mail avant l'exposé ;

     présence souhaitable.
```