Cours de l'option informatique

Lycée Louis-le-Grand

Année 2000–2001

Automates

sommaire

— notion d'automate, leur intérêt et leurs usages ;

— calculs d'un automate et langage reconnu ;

— déterminisme, comment s'en dispenser, et comment s'en assurer ;

— langages reconnaissables ;

— leurs propriétés de stabilité ;

— langages rationnels ;

— le théorème de Kleene et le lemme de pompage ;

— expressions régulières ;

— le problème de la minimisation

Alphabet et mots

Un *alphabet* est un ensemble **fini** non vide $\mathcal{A}$ dont les éléments sont appelés des *caractères*.

Un *mot* sur $\mathcal{A}$ est soit le *mot vide*, noté ε, soit un mot de *taille* p :
$m = m_1 m_2 \dots m_p$ où chaque m_i est un caractère.

On note la *longueur* (on dit aussi la taille) d'un mot ainsi :

$$0 = |\varepsilon|, \qquad p = |m_1 \dots m_p|.$$

La *concaténation* de deux mots est notée par un point.

Si $|m| = p$ et $|m'| = q$, $|m.m'| = p + q$ et le i-ème caractère de $m.m'$ est

$$\begin{cases} m_i, & \text{si } i \leqslant p\,; \\ m'_{i-p}, & \text{si } p+1 \leqslant i \leqslant p+q. \end{cases}$$

Notations

L'ensemble des mots de longueur p se note $\mathcal{A}^p$.

L'ensemble de tous les mots est $\mathcal{A}^\star = \{\varepsilon\} \cup \bigcup_{p \in \mathbb{N}^\star} \mathcal{A}^p$.

$(\mathcal{A}^\star, .)$ est un monoïde (c'est le monoïde libre sur $\mathcal{A}$), dont l'élément neutre est le mot ε.

La loi est évidemment non commutative mais associative.

Le mot `abbaaabcc` peut être naturellement noté $\texttt{ab}^2\texttt{a}^3\texttt{bc}^2$.

Le *miroir* d'un mot m est noté $\overline{m}$. Par exemple :

$$\overline{\texttt{miroir}} = \texttt{riorim}.$$

Langages

(L'alphabet $\mathcal{A}$ est supposé choisi une fois pour toutes.)

Un *langage* L est simplement un ensemble (fini ou non) de mots : l'ensemble de tous les langages est donc $\mathcal{P}(\mathcal{A}^\star)$.

On dispose donc des opérateurs ensemblistes habituels $\cup$, $\cap$, Δ, $\setminus$ et des relations d'inclusion entre langages.

On pourra de la même façon parler du complément d'un langage L : il s'agit de $\mathcal{A}^\star \setminus L$.

Opérations sur les langages

En outre si L et L' sont deux langages, leur *concaténé* est $L.L' = \{m.m', m \in L, m' \in L'\}$. $L.L$ est noté L^2, et ainsi de suite.

Ne pas confondre L^2 et $\{m.m, m \in L\}$.

L'*étoile* d'un langage L est le langage $L^\star = \{\varepsilon\} \cup \bigcup_{p \in \mathbb{N}^\star} L^p$.

On peut définir bien d'autres opérations, par exemple $\sqrt{L} = \{m \in \mathcal{A}^\star, m.m \in L\}$. (A-t-on $\sqrt{L}^2 = L$? et $\sqrt{L^2} = L$?)

Ou encore le mélange (*shuffle*) de deux langages L et M : pour écrire un mot de $L \sharp M$, on choisit un mot a de L et un mot b de M, puis on mélange les lettres, en conservant toutefois l'ordre relatif dans chacun des mots a et b.

Par exemple : `abbbabcc` est dans $\{\texttt{a}, \texttt{b}\}^\star \sharp \{\texttt{b}, \texttt{c}\}^\star$, mais pas `abbbbacc`.

Automates finis déterministes

Un *afd* sur l'alphabet $\mathcal{A}$ est un quadruplet $\alpha = (Q, q_0, F, \delta)$ où :

— Q est un ensemble fini, ses éléments sont les *états* de l'automate ;

— $q_0 \in Q$ est l'*état initial* ;

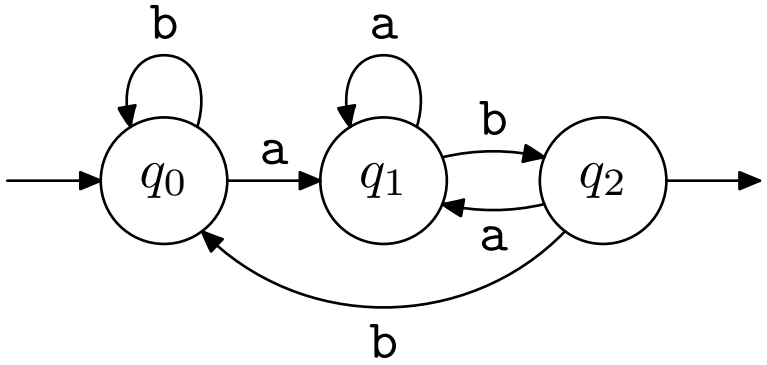
— $F \subset Q$ est l'ensemble des *états finals* ;

— δ est une fonction (pas une application) de $Q \times \mathcal{A}$ dans Q, appelée *fonction de transition* de l'automate.

Si $\delta(q, c) = q'$ on dit que l'automate passe de l'état q à l'état q' à la lecture du caractère c. On note également : $q.c = q'$.

Exemple $\alpha = (\{q_0, q_1, q_2\}, q_0, \{q_2\}, \delta)$ avec :

q	q_0		q_1		q_2	
c	a	b	a	b	a	b
$\delta(q, c)$	q_1	q_0	q_1	q_2	q_1	q_0



Calculs d'un automate

Soit $\alpha = (Q, q_0, F, \delta)$ un afd.

On généralise la notation $q.c = \delta(q, c)$ en posant

$$\begin{aligned} \forall q \in Q, \qquad q.\varepsilon &= q \\ \forall q \in Q, \forall c \in \mathcal{A}, \forall u \in \mathcal{A}^\star, \qquad q.(u.c) &= (q.u).c \end{aligned}$$

ce qui fait agir $\mathcal{A}^\star$ sur Q.

Ces notations ne sont pas forcément partout définies. On parle alors de *blocage* de l'afd sur une certaine transition.

Certains auteurs utilisent la notation $\delta^*(q, u)$ au lieu de $q.u$.

Programmation

Un typage possible des afd

On se contente de numéroter les états.

```
type afd = { initial : int ; finals : int list ;
             transitions : (char * int) list vect } ;;
exception Blocage ;;
```

Les transitions sont gérées par une liste associative.

Calculs de l'afd

```
let calcul alpha q u =
    let n = string_length u
    in
    let rec avance q i =
        if i = n then q
        else avance (assoc u.[i] alpha.transitions.(q)) (i + 1)
    in
    try avance q 0
    with Not_found -> raise Blocage ;;
```

Reconnaissance

```
let reconnaît alpha u =
    mem (calcul alpha alpha.initial u) alpha.finals ;;
```

Rappel (fonctions de la bibliothèque Caml)

```
let rec mem x = function
    | [] -> false
    | t :: q -> t = x || mem t q ;;

let rec assoc x = function
    | [] -> raise Not_found
    | (a,b) :: _ when a = x -> b
    | _ :: q -> assoc x q ;;
```

Langage reconnu

Le langage *reconnu* par l'afd $\alpha = (Q, q_0, F, \delta)$ est défini par :

$$L(\alpha) = \{u \in \mathcal{A}^\star, q_0.u \in F\}.$$

Autrement dit, u est un mot reconnu s'il fait passer l'automate de son état initial à un état final.

Un langage est dit *reconnaissable* s'il existe un afd dont il est le langage.

Accessibilité

Un état q est dit *accessible* s'il existe un mot u tel que $q_0.u = q$. L'automate est dit accessible si tous ses états sont accessibles.

Un état q est dit *co-accessible* s'il existe un mot u et un état final q_f tel que $q.u = q_f \in F$. L'automate est dit co-accessible si tous ses états sont co-accessibles.

Certains nomment *utiles* les états à la fois accessibles et co-accessibles. *Émonder* un automate, c'est supprimer tous les états inutiles.

Théorème 1

Pour tout afd α tel que $L(\alpha) \neq \emptyset$, il existe un afd α' émondé qui reconnaît le même langage.

Élimination des états non utiles

✧ Soit $\alpha = (Q, q_0, F, \delta)$ un afd tel que $L(\alpha) \neq \emptyset$. Notons U l'ensemble de ses états utiles : U n'est pas vide, car il existe au moins un mot $u = u_1 \dots u_p$ dans $L(\alpha)$ qui, depuis l'état q_0 fait passer α dans des états successifs $q_i = q_0.(u_1 \dots u_i)$. Comme $u \in L(\alpha)$, c'est que $q_p \in F$, et alors tous les q_i sont des états utiles. (Remarque : cela reste vrai si $L(\alpha)$ est réduit à $\{\varepsilon\}$.) En particulier : $q_0 \in U$ et $F \cap U \neq \emptyset$.

Soit alors $\alpha' = (U, q_0, F \cap U, \delta')$ où $\delta'(q, c) = \delta(q, c)$ si $q \in U$ et $\delta(q, c) \in U$, et n'est pas défini dans le cas contraire.

L'inclusion $L(\alpha') \subset L(\alpha)$ est à peu près évidente : tout calcul de α' est en effet un calcul de α.

Réciproquement, on a vu qu'un calcul **réussi** de α ne passe que par des états utiles, donc est encore un calcul (réussi) de α'. ✦

Exercice de programmation

Écrire les fonctions suivantes :

```
accessibles : afd -> int list
co_accessibles : afd -> int list
```

et tenter d'évaluer leurs complexités.

```
let rec subtract l m = match l with
    | [] -> []
    | t :: q -> if mem t m then subtract q m
                else t :: (subtract q m) ;;

let accessibles alpha =
    let rec progresse trouvés = function
        | [] -> trouvés
        | t :: q -> let d = map snd alpha.transitions.(t)
                    in
                    let d' = subtract d trouvés
                    in
                    progresse (d' @ trouvés) (d' @ q)
    in
    progresse [ alpha.initial ] [ alpha.initial ] ;;
```

Complétude d'un afd

Un afd est dit *complet* si sa fonction de transition δ est définie partout : il n'est jamais l'objet d'un blocage.

Théorème 2

Pour tout afd α, il existe un afd complet α' qui reconnaît le même langage.

On pourra donc *gratuitement* supposer que l'afd considéré est sans blocage.

Complétion d'un afd

✧ Soit donc $\alpha = (Q, q_0, F, \delta)$ un afd non complet.

On adjoint à Q un nouvel état q_ω, dit état-puits, obtenant un nouvel ensemble d'états $Q' = Q \cup \{q_\omega\}$. On pose alors $\alpha' = (Q', q_0, F, \delta')$ avec :

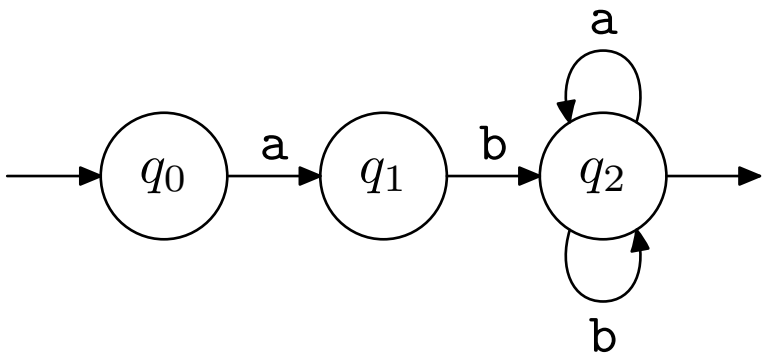
$$\begin{aligned} \forall q \in Q, \forall c \in \mathcal{A}, \delta'(q,c) &= \begin{cases} \delta(q,c), & \text{quand elle est définie ;} \\ q_\omega, & \text{sinon ;} \end{cases} \\ \forall c \in \mathcal{A}, \delta'(q_\omega, c) &= q_\omega . \end{aligned}$$

Ainsi a-t-on défini δ' sur tout $Q' \times \mathcal{A}$, et α' est sans blocage.

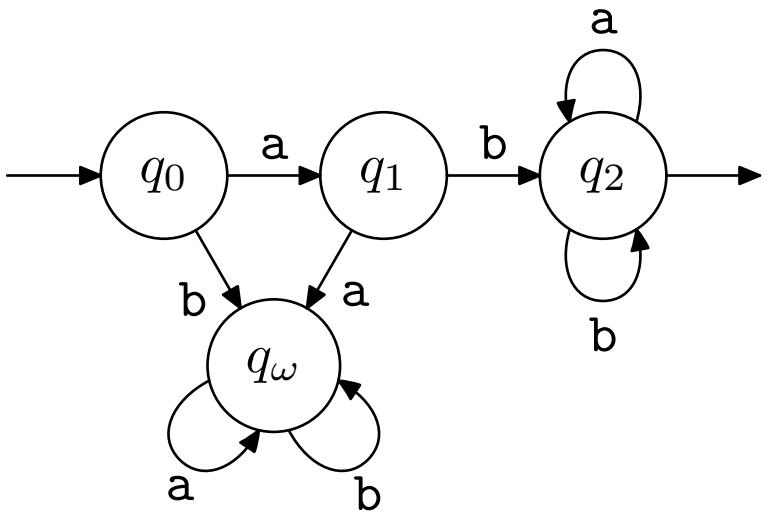
En outre, tout calcul de α est aussi calcul de α' et donc $L(\alpha) \subset L(\alpha')$.

Les seules transitions qui sortent de q_ω y retournent, et cet état n'est pas final, donc les calculs réussis de α' doivent éviter l'état-puits : ce sont donc aussi des calculs (réussis) de α, et $L(\alpha') = L(\alpha)$. ✦

Exemple Cet afd se bloque sur la chaîne aa (entre autres. . .) :



Après complétion, il devient sans blocage :



Automate fini non déterministe

Un *afnd* sur l'alphabet $\mathcal{A}$ est un quadruplet $\alpha = (Q, I, F, \delta)$ où

— Q est l'ensemble fini des *états* ;

— I est une partie finie de Q, constituée des *états initiaux* ;

— F est une partie finie de Q, constituée des *états finaux* ;

— δ est une **application** de $Q \times \mathcal{A}$ dans $\mathcal{P}(Q)$, appelée *fonction de transition*.

On aura noté : qu'il peut y avoir plusieurs états initiaux ; que les transitions sont définies pour tout couple (q, c), mais leur font correspondre des ensembles d'états (éventuellement vides, ce qui correspond aux blocages).

Calculs d'un afnd

On généralise la définition de δ à $\delta^\star : Q \times \mathcal{A}^\star \to \mathcal{P}(Q)$:

$$\begin{aligned} \forall q \in Q, \qquad q.\varepsilon &= \{q\} \\ \forall q \in Q, \forall c \in \mathcal{A}, \forall u \in \mathcal{A}^\star, \qquad q.(u.c) &= \bigcup_{q' \in q.u} q'.c \end{aligned}$$

avec la notation alternative $q.u = \delta^\star(q, u)$.

La programmation des calculs d'un afnd est bien plus coûteuse que pour un afd : on perd la linéarité en la taille de la chaîne de caractères.

Typage d'un afnd

```
type afnd = { initials : int list ; finals : int list ;
              transitions : (char * int list) list vect } ;;
```

On trouvera la liste des états auxquels on peut aboutir à partir d'un état q à la lecture d'un caractère c par
`assoc c alpha.transitions.(q)` ; si l'on part d'une liste l d'états on utilisera la fonction `union` de la bibliothèque Caml et on évaluera

```
it_list (fun a q -> union a (assoc c alpha.transitions.(q))) [] l
```

On en déduit :

```
let calcul alpha départ u =
    let n = string_length u
    in
    let rec avance ql i =
        if i = n then ql
        else
        avance
          (it_list (fun a q -> union a (assoc c alpha.transitions.(q)))
                   [] ql)
          (i + 1)
    in
    try avance départ 0
    with Not_found -> raise Blocage ;;

let reconnaît alpha u =
    let arrivée = calcul alpha alpha.initials u
    in
    it_list (fun b q -> b || mem q alpha.finals) false arrivée ;;
```

Langage d'un afnd

Le *langage reconnu* par un afnd $\alpha = (Q, I, F, \delta)$ est l'ensemble des mots qui donnent lieu à un *calcul réussi*, c'est-à-dire faisant passer l'automate d'un état initial à un état final.

Autrement dit : $L(\alpha) = \{u \in \mathcal{A}^\star, \exists q_0 \in I, q_0.u \cap F \neq \emptyset\}$.

Les notions d'accessibilité (pour les états) et de complétude (pour l'automate) sont analogues à celles qu'on a décrites pour les afd.

Ici encore, l'ajout d'un état-puits permet de supposer qu'un afnd est sans blocage.

Afnd avec ε-transitions

Permettre les *ε-transitions* (on dit aussi *transitions spontanées* ou *instantanées*), c'est autoriser l'automate, quand il est dans un état q, à passer dans un nouvel état q' sans même lire un caractère.

Il s'agit d'une généralisation des afnd : on définit les "nouveaux" afnd comme des **quintuplets** $\alpha = (Q, I, F, \delta, \varphi)$ où $\varphi(q)$ est — pour tout état q — l'ensemble des états auxquels on peut aboutir à partir de q sans rien lire.

φ est donc une application de Q dans $\mathcal{P}(Q)$.

Clôtures

On appelle *clôture instantanée* (ou par ε-transitions) d'un ensemble X d'états la plus petite partie $\kappa(X)$ de Q qui contient X et qui reste stable par φ : $\forall q \in \kappa(X), \varphi(q) \subset \kappa(X)$.

Notons $\varphi(X) = \{\varphi(x), x \in X\}$, puis $\varphi^2(X) = \varphi(\varphi(X))$, etc.

On a alors $\kappa(X) = X \cup \bigcup_{p \in \mathbb{N}^\star} \varphi^p(X)$: la suite des $\varphi^k(X)$ est nécessairement stationnaire (pour l'inclusion) puisque Q est fini, sa limite est $\kappa(X)$.

La programmation de la clôture est un exercice intéressant...

Calcul des clôtures On représente φ par `phi : int list vect`.

```
let clôture phi ql =
    let rec progresse trouvés = function
        | [] -> trouvés
        | t :: q -> let d = subtract phi.(t) trouvés
                    in
                    progresse (d @ trouvés) (d @ q)
    in
    progresse ql ql ;;
```

On aura remarqué l'analogie avec le calcul des transitions : c'est ici un parcours de graphe.

Mais, au fait, *en largeur d'abord* ou *en profondeur d'abord* ?

(Il s'en faut de peu...)

Calculs d'un afnd avec transitions instantanées

On généralise encore en posant :

$$\forall q \in Q, \qquad q \bullet \varepsilon = \kappa(\{q\})$$

$$\forall q \in Q, \forall c \in \mathcal{A}, \forall u \in \mathcal{A}^\star, \qquad q \bullet (u.c) = \kappa\left(\bigcup_{q' \in q \bullet u} q'.c\right).$$

En particulier pour un caractère c :

$$q \bullet c = q \bullet (\varepsilon.c) = \kappa\left(\bigcup_{q' \in \kappa(\{q\})} q'.c\right).$$

Déterminisme ou non ?

Avantage des automates déterministes : leur efficacité en termes de calcul.

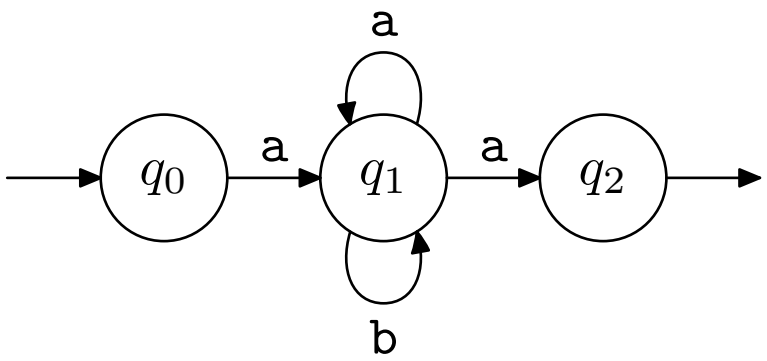
Avantage des automates non déterministes : leur expressivité.

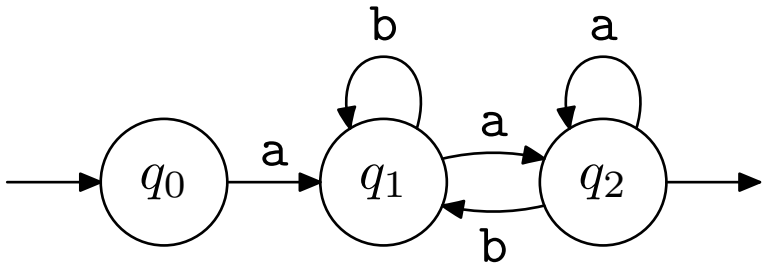
Il est bien évident que tout langage reconnu par un afd est également reconnu par un afnd (qu'il ne serait pas difficile de décrire !)... mais l'inverse est-il également vrai ?

C'est le problème de la *déterminisation* d'un afnd.

Nous allons voir qu'on dispose d'une méthode générale de déterminisation, mais qu'elle peut être très coûteuse : un coût exponentiel !

Expressivité Ces deux automates reconnaissent le même langage : les mots qui commencent et finissent par un a.





Déterminisation

Théorème 3

Soit α un automate non déterministe. Il existe un automate déterministe β qui reconnaît le même langage.

Ce théorème permet d'utiliser indifféremment un afd ou un afnd dans nos preuves : il n'y a pas eu d'enrichissement de la famille des langages reconnaissables qui serait lié au non déterminisme.

La preuve est constructive : c'est *l'algorithme des parties*.

✧ On considère $\alpha = (Q, I, F, \delta, \varphi)$ et la fonction de clôture κ associée.

Voici la construction de l'afd $\beta = (\mathcal{P}(Q), \kappa(I), F', \delta')$:

— l'état initial est $\kappa(I)$, qui est bien élément de $\mathcal{P}(Q)$;

— $F' = \{X \subset Q, X \cap F \neq \emptyset\}$ est constitué des parties possédant au moins un état final ;

— pour tout ensemble X d'états de Q et tout caractère c,

$$\delta'(X, c) = \kappa\left(\bigcup_{q \in X} \delta(q, c)\right).$$

✦

Remarque : seuls les états de β qui sont des clôtures sont accessibles. En pratique, on ne construit effectivement que des états accessibles.

Complexité de la déterminisation

Si l'afnd de départ possède n états, notre algorithme conduit à un algorithme possédant 2^n états (même si en réalité, on ne conserve que les états accessibles). Le coût de la déterminisation est donc au moins exponentiel.

Mais rien n'interdit *a priori* à un algorithme plus astucieux d'être plus efficace.

Eh bien, si :

Théorème 4

Soit n un entier naturel non nul. Il existe un afnd α possédant n états tel que tout afd β vérifiant $L(\alpha) = L(\beta)$ doit posséder au moins 2^{n-1} états.

✧ Soit l'afnd sans ε-transition $\alpha = (\{q_1, \ldots, q_n\}, \{q_1\}, \{q_n\}, \delta)$ défini par : $\delta(q_1, \mathtt{a}) = \{q_1, q_2\}$, $\delta(q_1, \mathtt{b}) = \{q_1\}$; $\delta(q_n, \mathtt{a}) = \delta(q_n, \mathtt{b}) = \emptyset$ et, pour $2 \leqslant i \leqslant n-1$: $\delta(q_i, \mathtt{a}) = \delta(q_i, \mathtt{b}) = \{q_{i+1}\}$.

α reconnaît les mots d'au moins $n-1$ caractères dont le $(n-1)$-ème en partant de la fin est un $\mathtt{a}$.

Soit un afd $\beta = (Q', q'_0, F', \delta')$ qui reconnaisse ce même langage : pour tout mot u, le mot $u.\mathtt{a}^n$ est reconnu, donc $\exists q'_u = \delta'(q'_0, u) \in Q'$.

Montrons que $u \mapsto q'_u$ est une injection de l'ensemble des mots de taille $n-1$ sur $\{\mathtt{a}, \mathtt{b}\}$ dans Q' : on aura bien montré que $|Q'| \geqslant 2^{n-1}$.

Si u et v sont deux mots de longueur $n-1$ **distincts**, on peut les écrire $u = u'.\mathtt{a}.w$ et $v = v'.\mathtt{b}.w$ (ou le contraire).

$U = u.\mathtt{a}^{n-2-|w|} \in L(\alpha)$ mais $V = v.\mathtt{a}^{n-2-|w|} \notin L(\alpha)$. Si on avait $q'_u = q'_v$ alors $q'_0.U = q'_0.V$ qui devrait être final **et** non final. ✦

Récapitulation

On dira de deux automates qu'ils sont *équivalents* s'ils reconnaissent le même langage.

On a vu que tout automate non déterministe (avec ou sans ε-transitions) est équivalent à

— un automate déterministe ;

— un automate déterministe complet (sans blocage) ;

— un automate déterministe sans état inutile.

Notons $\mathcal{R}$ l'ensemble des langages reconnaissables.

Automate produit de deux automates

Soit $\alpha = (Q, q_0, F, \delta)$ et $\alpha' = (Q', q'_0, F', \delta')$ deux automates déterministes. Leur produit est l'automate

$$\alpha \times \alpha' = (Q \times Q', (q_0, q'_0), F \times F', \Delta),$$

où la fonction de transition Δ est définie — quand c'est possible — par : $\forall (q, q') \in Q \times Q', \forall c \in \mathcal{A}, \Delta((q, q'), c) = (\delta(q, c), \delta'(q', c))$.

On dispose d'une construction analogue pour les automates non déterministes, qu'ils soient avec ou sans transitions spontanées.

Stabilités de $\mathcal{R}$

Théorème 5

$\alpha \times \alpha'$ reconnaît l'intersection des langages $L(\alpha)$ et $L(\alpha')$.

Corollaire 1 $\mathcal{R}$ est stable par intersection.

Remplaçant dans $\alpha \times \alpha'$ l'ensemble des états finals par $F \times Q' \bigcup Q \times F$, on obtient un nouvel afd $\alpha \otimes \alpha'$.

Théorème 6

$\alpha \otimes \alpha'$ reconnaît la réunion des langages $L(\alpha)$ et $L(\alpha')$.

Corollaire 2 $\mathcal{R}$ est stable par réunion.

Supposons (gratuitement) α et α' complets, et choisissons comme ensemble d'états finals du produit $F \times (Q' \setminus F')$. On obtient un nouvel afd β.

Théorème 7
β reconnaît la différence ensembliste des langages $L(\alpha)$ et $L(\alpha')$.

Corollaire 3 $\mathcal{R}$ est stable par différence ensembliste.

Corollaire 4 $\mathcal{R}$ est stable par différence symétrique.

On en déduit le résultat suivant, pas si évident *a priori* :

Corollaire 5 On sait décider si deux automates reconnaissent le même langage.

Parce que le langage reconnu par leur différence est vide si et seulement si aucun état n'est utile.

Dessinons maintenant deux automates α et α' et ajoutons une transition instantanée de chaque état final de α vers chaque état initial de α'. On obtient un nouvel automate (non déterministe) qui reconnaît $L(\alpha).L(\alpha')$.

Théorème 8

$\mathcal{R}$ est stable par concaténation.

Il faut prendre plus de soin pour la stabilité par l'étoile. Soit α un automate. On ajoute un nouvel état q_α qui sera le nouvel et unique état initial et un nouvel état q_ω qui sera le nouvel et unique état final. Il suffit de placer des ε-transitions depuis q_α vers q_ω et vers chaque état initial de l'automate de départ d'une part et de chaque état final vers q_ω d'autre part, pour obtenir l'automate souhaité.

Théorème 9

$\mathcal{R}$ est stable par étoile.

Or il est évident que

Lemme 1 $\emptyset$, $\mathcal{A}^\star$ et, pour tout caractère c, $\{c\}$, sont tous des langages reconnaissables.

Corollaire 6 Tout langage fini est reconnaissable.

Corollaire 7 Le complémentaire d'un langage reconnaissable est reconnaissable.

D'ailleurs il suffit, dans un automate **complet**, d'échanger états finals et non finals pour obtenir l'automate qui reconnaît le complémentaire.

De même il suffit qu'on retourne les flèches d'un afd, qu'on nomme initiaux les anciens états finals, et final l'ancien état initial, pour obtenir un automate (non déterministe maintenant) qui reconnaît le miroir du langage reconnu par l'afd de départ.

Théorème 10

$\mathcal{R}$ est stable par miroir.

Exercice Montrer que $\mathcal{R}$ est stable par *shuffle* et par racine carrée, où $\sqrt{L} = \{u \in \mathcal{A}^*, u.u \in L\}$.

✧ [shuffle] Si $\alpha = (Q, q_0, F, \delta)$ (*resp.* $\beta = (Q', q'_0, F', \delta')$) est un afd qui reconnaît L (*resp.* M), on construit un afnd $\gamma = (Q \times Q', \{(q_0, q'_0)\}, F \times F', \Delta)$ en posant $\Delta((q, q'), c) = \{(\delta(q, c), q'), (q, \delta'(q', c))\}$ dont on vérifie facilement qu'il reconnaît $L \sharp M$. ✦

✧ [$\sqrt{L}$] Notons $Q = \{q_0, q_1, \ldots, q_n\}$ les $n + 1$ états d'un afd $\alpha = (Q, q_0, F, \delta)$ qui reconnaît L. $\sqrt{L}$ est alors reconnu par l'afd $\beta = (Q^{n+1}, (q_0, q_1, \ldots, q_n), F', \Delta)$ où on a posé $\Delta((p_0, \ldots, p_n), c) = (\delta(p_0, c), \ldots, \delta(p_n, c))$ et où les états finaux sont de la forme : $(p_0, \ldots, p_n)$ avec $p_i \in F$ dès que $p_0 = q_i$. En effet un mot u conduira β dans un tel état final si $q_0.u = p_0 = q_i$ et si $q_i.u = p_i \in F$, donc $q_0.(uu) \in F$. ✦

Syntaxe des expressions régulières

On définit de façon analogue aux expressions algébriques l'ensemble $\mathcal{E}$ des expressions régulières sur l'alphabet $\mathcal{A}$.

constantes tout caractère de $\mathcal{A}$, ε et $\emptyset$ sont des expressions régulières ;

variables on utilisera à volonté un ensemble dénombrable de variables ;

concaténation si m et m' sont deux expressions régulières, $m.m'$ aussi (il s'agit d'un opérateur d'arité 2) ;

étoile si m est une expression régulière, $m\star$ aussi (il s'agit d'un opérateur d'arité 1) ;

choix si m et m' sont deux expressions régulières, $m \mid m'$ aussi (il s'agit d'un opérateur d'arité 2).

Remarques

Pour éviter un parenthésage trop lourd, on définit un ordre de priorité sur les opérateurs : $\star$ a priorité sur . qui a priorité sur $|$

Ainsi $\mathtt{ab}\star|\mathtt{a}$ représente $(\mathtt{a}.(\mathtt{b}\star))|\mathtt{a}$.

On pourrait voir $\mathcal{E}$ comme un langage sur l'alphabet $\mathcal{A} \cup \{\emptyset, \varepsilon, |, \star, ., (,)\}$. Il ne s'agit pas d'un langage reconnaissable.

La *taille* d'une expression régulière est le nombre de symboles qui la composent.

Sémantique des expressions régulières

Pour définir une sémantique sur $\mathcal{E}$ nous définissons (de façon inductive) une interprétation : c'est-à-dire une application μ qui à toute expression régulière associe un langage sur l'alphabet $\mathcal{A}$.

Les variables sont d'abord interprétées dans le contexte courant : on les remplace par les expressions régulières qu'elles représentent.

Reste à définir notre interprétation des constantes et des opérateurs : $\mu(\emptyset) = \emptyset$, $\mu(\varepsilon) = \{\varepsilon\}$, et pour tout caractère c, $\mu(c) = \{c\}$;

si x et y sont deux expressions régulières, $\mu(x.y) = \mu(x).\mu(y)$, $\mu(x|y) = \mu(x) \cup \mu(y)$, $\mu(x\star) = \mu(x)^\star$.

Par exemple : $\mu(\texttt{ab}(\texttt{a}|\texttt{b}) \star \texttt{ab})$ est l'ensemble des mots d'au moins quatre caractères sur $\{\texttt{a}, \texttt{b}\}$ qui commencent et se terminent par $\texttt{ab}$.

L'égalité des expressions régulières est bien sûr notée $=$.

Deux expressions régulières x et y qui *représentent* le même langage (c'est-à-dire que $\mu(x) = \mu(y)$) sont dites *équivalentes*, ce qu'on note $x \equiv y$.

Le problème de décider de l'équivalence de deux expressions régulières est difficile. On en dira davantage plus tard.

Dans notre sémantique, on remarque que $\emptyset$ est neutre pour $|$, absorbant pour la concaténation, et ambigü pour l'étoile. En pratique, la seule expression régulière contenant $\emptyset$ qui sera utilisée est $\emptyset$ elle-même.

Typage des expressions régulières

Caml est tout à fait adapté au typage des expressions régulières :

```
type regexp =
   | Vide
   | Mot of string
   | Concat of regexp list
   | Choix of regexp list
   | Étoile of regexp ;;
```

Pour éviter trop de parenthèses, on a choisi d'utiliser le type `string` plutôt qu'une concaténation de `char`. De la même façon, on profite de l'associativité du choix et de la concaténation pour permettre de les appliquer à plus de deux arguments.

Langages rationnels

Définition 1 (Langage rationnel) L'ensemble Rat des langages rationnels sur l'alphabet $\mathcal{A}$ est la plus petite partie de $L(\mathcal{A})$ qui contienne le langage vide et les singletons, et qui soit stable pour l'union, la concaténation et l'étoile.

Sont donc en particulier langages rationnels : $\emptyset$, $\mathcal{A}^\star$, tout langage fini...

Les Américains disent *regular expression* et *regular language*. Il arrivera que vous entendiez en français *expression rationnelle*, voire même *langage régulier*... tout cela n'a guère d'importance !

Langages rationnels et expressions régulières

Théorème 11

Un langage est rationnel si et seulement si il existe une expression régulière qui le représente.

✧ De par la définition même des langages rationnels, il est clair que tous les langages associés aux expressions régulières sont bien rationnels.

Pour montrer la réciproque, on remarque tout d'abord que vide et tout singleton sont représentés par des expressions régulières. L'ensemble des langages associés aux expressions régulières étant clairement stable par union, concaténation et étoile, on en déduit qu'on a bien décrit tous les langages rationnels. ✦

Langages rationnels et langages reconnaissables

Le théorème de Kleene répond à la question que tous se posent :

Théorème 12 (Kleene)
Les langages reconnaissables sont les langages rationnels. Et réciproquement, d'ailleurs.

Toutes les stabilités qu'on a pu découvrir pour $\mathcal{R}$ sont donc encore vraies pour Rat.

Les automates fournissent de fait un moyen de démontrer ce qui ne serait pas du tout évident sans eux... par exemple l'intersection de deux langages rationnels.

Des expressions régulières aux automates

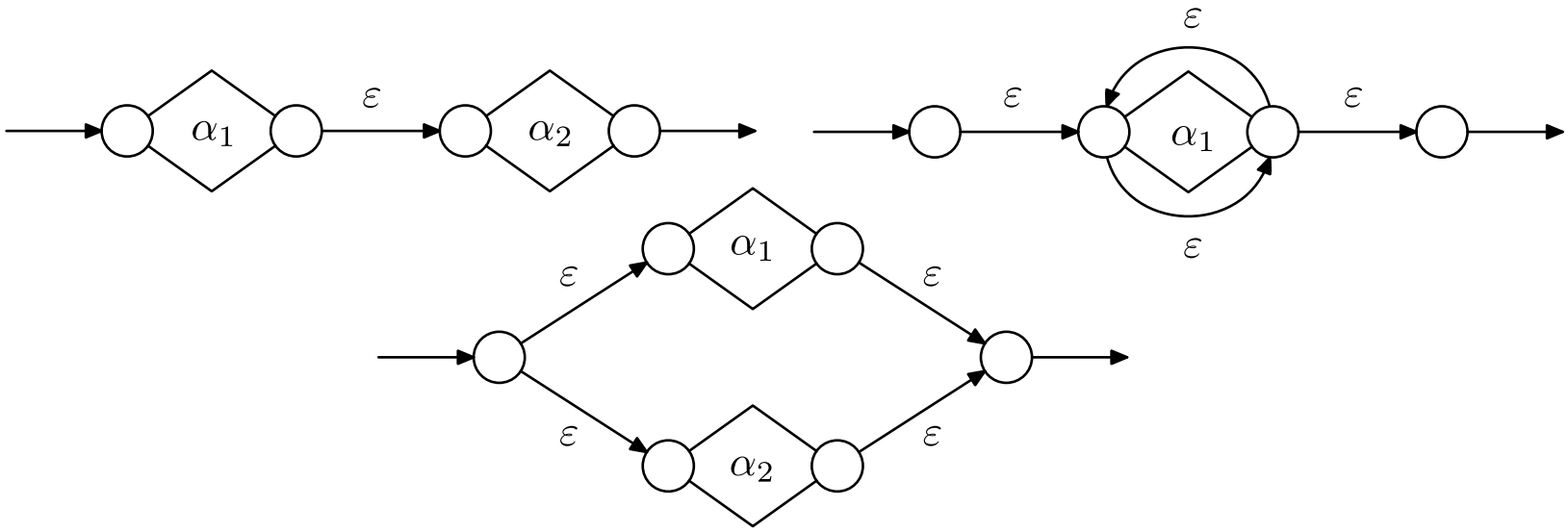
C'est le seul sens de l'équivalence dont la démonstration figure au programme officiel.

Nous verrons deux algorithmes qui construisent un automate reconnaissant le langage décrit par une expression régulière :

— le premier construit par une induction structurelle des automates de Thompson : c'est-à-dire des afnd ne comportant qu'un état initial où n'aboutit aucune transition et un seul état final d'où ne part aucune transition ;

— le second construit un automate de Gloushkov à partir d'une expression régulière qu'on aura au préalable débarassée de toute occurrence de ε.

Automates de Thompson On dessine sans difficulté des automates pour les langages $\emptyset$, $\{\varepsilon\}$ et les singletons. Puis, supposant construits des automates de Thompson α_1 et α_2 pour les expressions régulières e_1 et e_2, on construit les automates de Thompson pour $e_1.e_2$, $e_1\star$ et $e_1|e_2$ de la façon suivante :



Élimination des ε Soit une expression régulière (sans $\emptyset$) : on cherche une expression régulière équivalente mais où ne figure pas ε.

On applique les règles simples suivantes : $\varepsilon.e \equiv e.\varepsilon \equiv e$, $(\varepsilon|e)\star \equiv e\star$, $(\varepsilon|e).e' \equiv e'|ee'$ et $e'.(\varepsilon|e) \equiv e'|e'e$.

Finalement, on aboutit ou bien à une expression régulière sans aucun ε, ou bien à une expression du type $\varepsilon|e$ où e ne contient pas de ε.

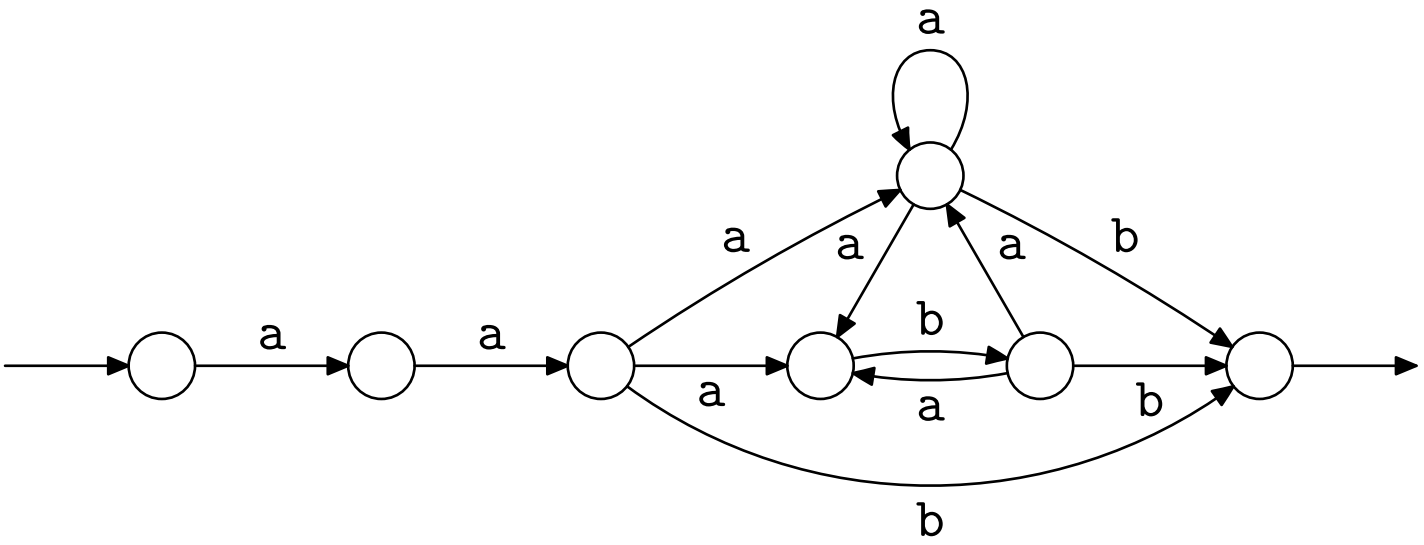
Dans ce dernier cas, une simple transformation de l'automate de Gloushkov associé à e convient : il suffit de décider que l'état initial est aussi final.

Algorithme de Gloushkov Soit donc une expression régulière sans ε, par exemple $\mathtt{aa}(\mathtt{a}|\mathtt{ab})^\star\mathtt{b}$. On indice chaque lettre : $\mathtt{a}_1\mathtt{a}_2(\mathtt{a}_3|\mathtt{a}_4\mathtt{b}_5)^\star\mathtt{b}_6$ et on crée les états correspondants (ici $q_1, \ldots, q_6$) auxquels on ajoute un état final q_0. Sont finals les états qui correspondent à une lettre pouvant terminer un mot (ici : q_6 seul).

On crée une transition de q_i vers q_j, étiquettée par la lettre c_j d'indice j, dès que le facteur c_ic_j apparaît dans un mot et de q_0 vers q_j (étiquettée par c_j) si un mot peut commencer par c_j.

Dans notre exemple : transitions étiquettées par $\mathtt{a}$: de q_0 vers q_1 ; de q_1 vers q_2 ; de q_2, q_3 et $q5$ vers q_3 et vers q_4 ; transitions étiquettées par $\mathtt{b}$: de q_4 vers q_5 ; de q_2, q_3 et q_5 vers q_6.

Exemple d'automate de Gloushkov pour aa(a|ab)⋆b.



Des automates aux expressions régulières

Ceci ne figure pas au programme officiel.

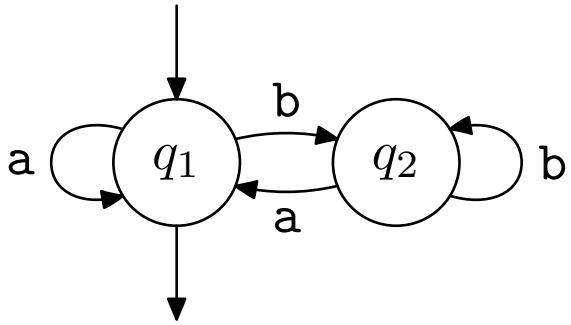
Nous présentons trois algorithmes pour écrire une expression régulière qui représente le langage reconnu par un automate donné :

— l'algorithme par élimination des états ;

— l'algorithme de McNaughton et Yamada ;

— l'algorithme par résolution d'équations.

L'automate suivant servira de support :



Algorithme par élimination des états

L'idée est d'étiqueter les transitions par des expressions régulières : cela permet de supprimer successivement tous les états, jusqu'à n'en avoir plus qu'un final et initial ou deux : un initial et un final. Quand on supprime l'état p, on liste l'ensemble X_p (*resp.* Y_p) des états pour lesquels il existe une transition arrivant dans p (*resp.* issue de p). Pour chaque couple $(x, y) \in X_p \times Y_p$, on étiquette la transition de x vers y par $e.f \star .g \mid h$ où e (*resp.* f, g, h) est l'étiquette de la transition de x vers p (*resp.* de p vers p, de p vers y, de x vers y).

Ici : on étiquette la transition de l'état 1 sur lui-même par $\mathtt{a}|\mathtt{bb} \star \mathtt{a}$, et finalement une expression régulière représentant $L(\alpha)$ s'écrit $e = (\mathtt{a}|\mathtt{bb} \star \mathtt{a})\star$.

Algorithme de McNaughton et Yamada

On numérote de 1 à n les états de l'automate α, et on note $L_{p,q}^{(k)}$ l'ensemble des mots qui font passer l'automate de l'état p à l'état q en ne passant que par des états de numéros inférieurs ou égaux à k. On note $e_{p,q}^{(k)}$ une expression régulière qui représente cet ensemble. Alors $L(\alpha) = \bigcup_{p\in I, q\in F} L_{p,q}^{(n)}$ plus éventuellement $\{\varepsilon\}$ si un état est à la fois initial et final, et on obtient l'expression régulière cherchée en évaluant le choix (*i.e.* $|$) des $e_{p,q}^{(n)}$ pour p initial et q final.

La récurrence s'enclenche car : $e_{p,q}^{(k+1)} = e_{p,q}^{(k)} \mid e_{p,k+1}^{(k)}.(e_{k+1,k+1}^{(k)}\star).e_{k+1,q}^{(k)}$.

Dans notre exemple : on écrit matriciellement $E^{(0)} = \begin{pmatrix} \mathtt{a} & \mathtt{b} \\ \mathtt{b} & \mathtt{a} \end{pmatrix}$,
$E^{(1)} = \begin{pmatrix} \mathtt{a}|\mathtt{aa}\star\mathtt{a} & \mathtt{b}|\mathtt{aa}\star\mathtt{b} \\ \mathtt{a}|\mathtt{aa}\star\mathtt{a} & \mathtt{b}|\mathtt{aa}\star\mathtt{b} \end{pmatrix} = \begin{pmatrix} \mathtt{aa}\star & \mathtt{a}\star\mathtt{b} \\ \mathtt{aa}\star & \mathtt{a}\star\mathtt{b} \end{pmatrix}$ puis $E^{(2)}$ se simplifie en
$\begin{pmatrix} (\mathtt{a}\star\mathtt{b})\star\mathtt{aa}\star & \mathtt{a}\star\mathtt{b}(\mathtt{a}\star\mathtt{b})\star \\ (\mathtt{a}\star\mathtt{b})\star\mathtt{aa}\star & \mathtt{a}\star\mathtt{b}(\mathtt{a}\star\mathtt{b})\star \end{pmatrix}$ et on trouve l'expression régulière finale : $L(\alpha)$ est représenté par l'expression $e = \varepsilon \mid (\mathtt{a}\star\mathtt{b})\star\mathtt{aa}\star$.

Algorithme par résolution d'équations

Pour tout état p de l'automate α, notons L_p l'ensemble des mots qui font passer α de l'état p à un état final. Bien sûr : $L(\alpha) = \bigcup_{p \in I} L_p$.

En outre, si $A_{p,q}$ est l'ensemble des étiquettes des transitions de p vers q, on dispose de $L_p = \begin{cases} \bigcup_{q \in Q} A_{p,q}.L_q, & \text{si } p \text{ n'est pas final ;} \\ \{\varepsilon\} \cup \bigcup_{q \in Q} A_{p,q}.L_q, & \text{si } p \text{ est final.} \end{cases}$

Dans notre exemple on a le système suivant d'équations :

$$\begin{aligned} L_1 &= \mathtt{a}L_1 | \mathtt{b}L_2 | \varepsilon \\ L_2 &= \mathtt{a}L_1 | \mathtt{b}L_2 \end{aligned}$$

On résout un tel système grâce au *lemme d'Arden* : on trouve d'abord $L_2 = \mathtt{b}^\star \mathtt{a}L_1$, que l'on reporte en obtenant $L_1 = \mathtt{b}^\star \mathtt{a}L_1 | \varepsilon$. Une dernière application du lemme d'Arden fournit l'expression régulière $e = (\mathtt{b}^\star \mathtt{a})^\star$.

Comme souvent dans ce genre de calculs, on confond allègrement expressions régulières et langages correspondants, afin d'alléger au maximum les notations.

Lemme d'Arden

Théorème 13

Soient K et L deux langages sur le même alphabet $\mathcal{A}$. $K^\star.L$ est solution de l'équation $X = K.X \cup L$. Si $\varepsilon \notin K$, il s'agit de la seule solution.

✧ On vérifie sans difficulté que $K^\star.L$ est solution.

Réciproquement, si X est solution, $L \subset X$ donc par une récurrence simple $\forall n, K^n.L \subset X$ et donc $K^\star.L \subset X$.

S'il existait un mot dans $X \setminus K^\star.L$, considérons en un, m, le plus court possible. $m \notin L$ donc $m \in K.X$ et s'écrit ainsi $m = k.x$ avec $|k| \geqslant 1$ si on suppose $\varepsilon \notin K$. On a trouvé ainsi $x \in X$ plus court que m : cela impose $x \in K^\star.L$, d'où à son tour $m = k.x \in K^\star.L$, ce qui est la contradiction attendue. ✦

Équivalence des expressions régulières

Rappelons qu'on a trouvé trois expressions régulières distinctes pour représenter le langage de l'automate-exemple :

$$(\mathtt{a}|\mathtt{bb}\star\mathtt{a})\star, \qquad \varepsilon|(\mathtt{a}\star\mathtt{b})\star\mathtt{aa}\star, \qquad (\mathtt{b}\star\mathtt{a})\star\,.$$

Ces trois expressions sont donc naturellement équivalentes, ce qu'on peut prouver à la main (exercice...). Mais on aimerait disposer d'un moyen sûr de décider de cette équivalence.

C'est une des applications de ce qui suit : à chaque expression régulière, on associe un afd par une des méthodes déjà vues. S'il existait une forme *canonique* pour les automates, on aurait la solution.

Résiduels d'un langage

On appelle *résiduel* d'un langage L (on dit aussi *quotient à gauche*) tout langage de la forme $u^{-1}.L = \{v \in \mathcal{A}, u.v \in L\}$.

On a bien sûr $(u.v)^{-1}.L = v^{-1}.(u^{-1}.L)$.

L'ensemble des résiduels du langage L s'écrit

$$\mathrm{res}(L) = \{u^{-1}.L, u \in \mathcal{A}^\star\} \in \mathcal{P}(\mathcal{P}(\mathcal{A}^\star)).$$

Exemple Soit L le langage des mots sur $\mathcal{A} = \{\mathtt{a}, \mathtt{b}\}$ se terminant par $\mathtt{aab}$. L possède 4 résiduels :
$L = \varepsilon^{-1}.L = \mathtt{b}^{-1}.L = \mathtt{b}^{-1}.(L \cup \{\mathtt{ab}\}) = \mathtt{b}^{-1}.(L \cup \{\varepsilon\})$,
$L \cup \{\mathtt{ab}\} = \mathtt{a}^{-1}.L = \mathtt{a}^{-1}.(L \cup \{\varepsilon\})$,
$L \cup \{\mathtt{ab}, \mathtt{b}\} = \mathtt{a}^{-1}.(L \cup \{\mathtt{ab}\}) = \mathtt{a}^{-1}.(L \cup \{\mathtt{ab}, \mathtt{b}\})$,
$L \cup \{\varepsilon\} = \mathtt{b}^{-1}.(L \cup \{\mathtt{ab}, \mathtt{b}\})$.

Taille minimale d'un automate

Soit $\alpha = (Q, q_0, F, \delta)$ un afd complet et accessible. Pour chaque état q, on note $L_q = \{u \in \mathcal{A}^\star, q.u \in F\}$.

Théorème 14
$\mathrm{res}(L(\alpha)) = \{L_q, q \in Q\}$.

Corollaire 8 Tout afd complet accessible qui reconnaît un langage L possède au moins $|\mathrm{res}(L)|$ états.

Corollaire 9 Un langage rationnel admet un nombre fini de résiduels.

✧ Soit $q \in Q$: α est accessible et donc $\exists w, q_0.w = q$. Alors $L_q = \{u, q.u \in F\} = \{u, q_0.(w.u) \in F\} = \{u, w.u \in L(\alpha)\} = w^{-1}.L(\alpha) \in \mathrm{res}(L(\alpha))$.

Inversement, soit u un mot, et, comme α est complet, soit $q = q_0.u$: on a de même $L_q = u^{-1}.L(\alpha)$. ✦

Un automate minimal

Soit L un langage ayant un nombre fini de résiduels.

Définition 2 (Automate des résiduels) Il s'agit de l'afd complet et accessible $\alpha = (\mathrm{res}(L), L, F, \delta)$ où F est l'ensemble des résiduels de L contenant ε, et où $\delta(u^{-1}.L, c) = c^{-1}.(u^{-1}.L) = (u.c)^{-1}.L$.

Théorème 15
L'automate des résiduels du langage L reconnaît le langage L. Il est de taille minimale parmi les afd complets et accessibles reconnaissant L.

Corollaire 10 Un langage est reconnaissable si et seulement s'il possède un nombre fini de résiduels.

✧ Par récurrence sur $|w|$, on montre que pour tout mot w, et tout résiduel $R = u^{-1}.L$, on a $\delta^\star(R, w) = w^{-1}.R = (u.w)^{-1}.L$.

Alors $u \in L(\alpha) \Leftrightarrow \delta^\star(L, u) \in F \Leftrightarrow u^{-1}.L \in F \Leftrightarrow \varepsilon \in u^{-1}.L \Leftrightarrow u \in L$. ✦

Minimisation

On a vu l'existence d'un afd complet accessible minimal reconnaissant un langage L reconnaissable : l'automate des résiduels, qui fournit donc en quelque sorte un *automate canonique*.

Mais si on me donne un afd, il n'est pas évident de trouver le langage qu'il reconnaît et de calculer ses résiduels.

On aimerait un algorithme permettant de minimiser l'automate sans passer par la description du langage reconnu.

C'est l'objet de l'étude qui suit : afin de minimiser le nombre d'états, il convient de faire en sorte qu'à deux états distincts $q \neq q'$ correspondent deux résiduels distincts : $L_q \neq L_{q'}$.

Équivalence de Nérode

Soit $\alpha = (Q, q_0, F, \delta)$ un afd complet et accessible. On dit de deux états q et q' qu'ils sont *équivalents (au sens de Nérode)*, et on note $q \sim q'$, si $L_q = \{u, q.u \in F\} = \{u, q'.u \in F\} = L_{q'}$. On notera $[q]$ la classe d'un état q, et, pour toute partie X de Q, $[X] = \{[q], q \in X\}$.

L'*automate quotient* $\alpha/\sim$ est l'afd complet accessible $([Q], [q_0], [F], \delta_\sim)$ où $\delta_\sim$ est définie par $\delta_\sim([q], c) = [\delta(q, c)]$ (vérifier que cela a du sens !).

En pratique, cela signifie simplement qu'on peut confondre en un seul état toute une classe d'équivalence.

Théorème 16

L'automate quotient de Nérode d'un afd accessible et complet est isomorphe à l'automate des résiduels du langage qu'il reconnaît.

Calcul de la relation d'équivalence de Nérode

Soit α un afd complet et accessible. On va décider pour chaque paire d'états s'ils sont ou non distinguables : c'est-à-dire si on peut s'assurer qu'ils ne sont pas dans la même classe d'équivalence. Nous proposons ici (sans démonstration) l'algorithme de Moore.

1. toute paire formée d'un état final et d'un état non final est marquée distinguable ;
2. pour toute paire distinguable (q, q') et tout caractère c, on marque la paire $\{\delta(q, c), \delta(q', c)\}$ comme étant distinguable ;
3. on itère l'étape 2 jusqu'à ce que la situation n'évolue plus.

Langages non reconnaissables

La propriété suivante, appelée *Lemme de pompage* ou *Lemme de l'étoile*, permet d'établir que de nombreux langages ne sont pas reconnaissables :

Lemme 2 (lemme de pompage) Soit L un langage reconnaissable. Il existe un entier $n > 0$ tel que tout facteur v d'un mot $u.v.w$ de L vérifiant $|v| \geqslant n$ se décompose sous la forme $v = r.s.t$ avec $|s| \geqslant 1$ et $\forall k \in \mathbb{N}, u.r.s^k.t.w \in L$.

Application Le langage $L = \{\mathtt{a}^p.\mathtt{b}^p, p \in \mathbb{N}\}$ n'est pas reconnaissable.

En effet, sinon, il existerait un entier n comme ci-dessus, et posant $u = \varepsilon$, $v = \mathtt{a}^n$, $w = \mathtt{b}^n$, il existerait $p \geqslant 1$ tel que $\forall k, \mathtt{a}^{n+kp}.\mathtt{b}^n \in L$.

Démonstration du lemme de pompage

✧ Soit en effet $\alpha = (Q, i, F, \delta)$ un afd qui reconnaît L, et $n = |Q|$. Écrivons $v = v_1 v_2 \dots v_p$ avec $p = |v|$. Soit $q_0 = i.u$, et, pour $1 \leqslant j \leqslant p$, posons $q_j = i.(u.v_1 \dots v_j) = q_0.(v_1 \dots v_j)$. Comme $p \geqslant n = |Q|$, il existe deux indices $0 \leqslant \ell < m \leqslant p$ tels que $q_\ell = q_m = q$. Il suffit pour conclure de poser $r = v_1 \dots v_\ell$, $s = v_{\ell+1} \dots v_m$ et $t = v_m \dots v_p$. On a alors

$$\begin{aligned} i.(u.r.s^k.t.w) &= ((i.(u.r)).s^k).(t.w) = (q_\ell.s^k).(t.w) \\ &= q.(t.w) = q_m.(t.w) = i.(u.v.w) \in F. \end{aligned}$$

✦