Logique avancée

1 - Rappels sur les automates finis et les langages réguliers

Master Info Nice Sophia Antipolis
E. Lozes

15. avril 2018

Contenu du cours

- Automates
  - sur les mots finis
  - sur les mots infinis
  - sur les arbres finis
- Jeux
  - d'accessibilité
  - de Büchi
  - de parité
- Logique
  - du premier ordre
  - monadique du second ordre
  - arithmétique de Presburger

Exercices de programmation

- environ toutes les 3 séances un exercice de programmation
- environ 2 heures de travail entre deux séances
- Langage de programmation
  - un code pour aider à débuter l'exercice sera fourni en Python
  - d'autres langages de programmation sont autorisés
- Mona (`http://www.brics.dk/mona/`)

Contenu de la séance

1. déterminisme et non déterminisme
2. propriétés de clôture, déterminisation, minimisation
3. problèmes de décision sur les automates
4. expressions régulières et théorème de Kleene

Déterminisme
et
non déterminisme

Mots finis

On se donne $\Sigma = \{a, b, \ldots\}$ un alphabet, i.e. un ensemble fini de lettres ou symboles.

Un mot $w = a_1 \ldots a_n$ est une suite finie de symboles.
La longueur du mot est $|w| = n$.
On note $w.v$ la concaténation de deux mots et ϵ le mot vide (de longueur 0).

On note Σ^* l'ensemble de tous les mots sur l'alphabet Σ.

Homomorphisme

Un homomorphisme $h : \Sigma^* \to \Sigma^*$ est une fonction qui associe à un mot w un autre mot $h(w)$ obtenu en remplaçant chaque lettre a de w par le mot $h(a)$

exemple : si $h(a) = ba$, $h(b) = ab$, $h(c) = a$, alors $h(abca) = ba\,ab\,a\,ba$.

Pour définir un homomorphisme h, il suffit donc de définir $h(x)$ pour chaque lettre $x \in \Sigma$.

Langage

Un langage est un ensemble de mots.
Par exemple, $\emptyset$, $\{\epsilon\}$, $\{abba\}$, Σ, ... sont des langages.

On peut définir de nouveaux langages en les combinant par des opérations ensemblistes, telles que l'union $(L_1 \cup L_2)$, l'intersection $(L_1 \cap L_2)$, ou la différence ensembliste $(L_1 \setminus L_2)$.

On peut aussi concaténer des langages : $L_1.L_2$ est le langage des mots qui sont obtenus par concaténation d'un mot de L_1 avec un mot de L_2.

$$L_1.L_2 = \{w_1.w_2 : w_1 \in L_1, w_2 \in L_2\}$$

Attention : $L.L$ contient plus de mots que $\{w.w : w \in L\}$

Langage (2)

On peut aussi construire un nouveau langage avec l'étoile de Kleene :

$$L^* = \bigcup_{n \geq 0} L^n = \{w_1 \dots w_n : n \geq 0, w_1 \in L, \dots, w_n \in L\}$$

L'image de L par l'homomorphisme h est l'ensemble des mots obtenus après remplacement : $h(L) = \{h(w) : w \in L\}$

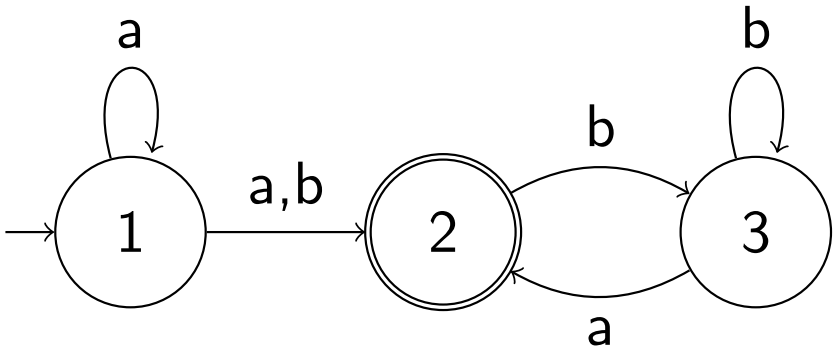
L'image inverse de L par l'homomorphisme h est l'ensemble des mots qui après remplacement se trouveraient dans L :
$h^{-1}(L) = \{w : h(w) \in L\}$

Automates finis non-déterministes

Un automate fini non-déterministe (AFN) est défini par le tuple $(Q, \Sigma, q_I, \delta, F)$ où

- Q est un ensemble fini d'états
- Σ est un alphabet
- $q_I \in Q$ est l'état initial
- $\delta : Q \times \Sigma \to 2^Q$ est la fonction de transition
- $F \subseteq Q$ est l'ensemble des états acceptants

On représente un automate par un graphe



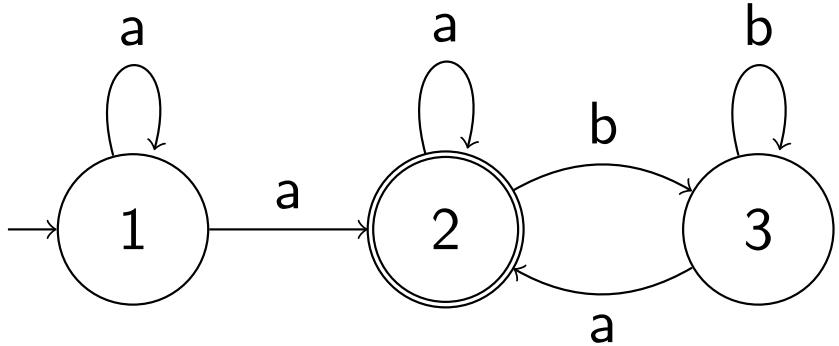
δ	a	b
1	$\{1, 2\}$	$\{2\}$
2	$\emptyset$	$\{3\}$
3	$\{2\}$	$\{3\}$

Langage d'un automate

- une exécution du mot $w = a_1 \dots a_n$ est une suite $q_I = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n$
- l'exécution est acceptante si $q_n \in F$
- le langage de $\mathcal{A}$ est l'ensemble des mots dont au moins une exécution est acceptante

On notera $\delta^*(q, w)$ l'ensemble des états où peut terminer une exécution de w qui débute en q. $L(A) = \{w : \delta^*(q_I, w) \in F\}$.

Le langage de l'automate ci-dessous est l'ensemble des mots qui commencent et terminent par la lettre a.



Propriétés de clôture

L est dit *reconnaissable* s'il existe un AFN $\mathcal{A}$ tel que $L = L(\mathcal{A})$
Théorème Si L, L_1, L_2 sont reconnaissables, alors

1. l'union $L_1 \cup L_2$
2. l'intersection $L_1 \cap L_2$
3. le complément $\Sigma^* \setminus L$
4. la concaténation $L_1.L_2$
5. l'étoile de Kleene L^*

sont reconnaissables.

Preuve : 1,2,4,5 correspondent à des opérations
≪ compositionnelles ≫ sur les automates. La complémentation fait appel à la déterminisation.

Automates finis déterministes

Un automate fini déterministe (AFD) est défini par un tuple $(Q, \Sigma, q_I, \delta, F)$, où désormais

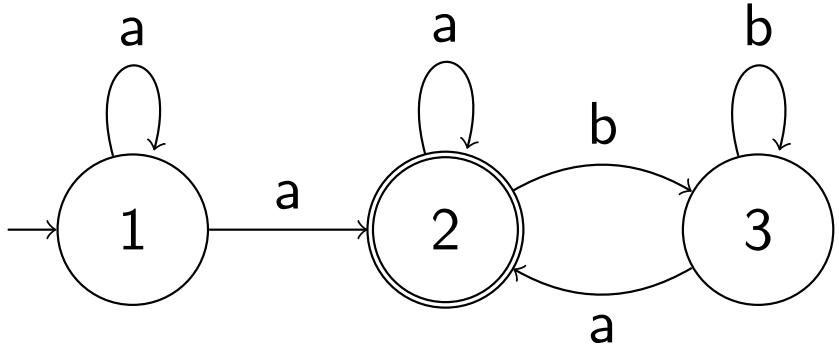
- $\delta : Q \times \Sigma \to Q$ (et non $Q \times \Sigma \to 2^Q$)
- $L(\mathcal{A}) = \{w : \underline{\text{l'exécution}}$ de w est acceptante$\}$

Théorème : un langage est reconnaissable (par un AFN) ssi il est reconnaissable par un AFD

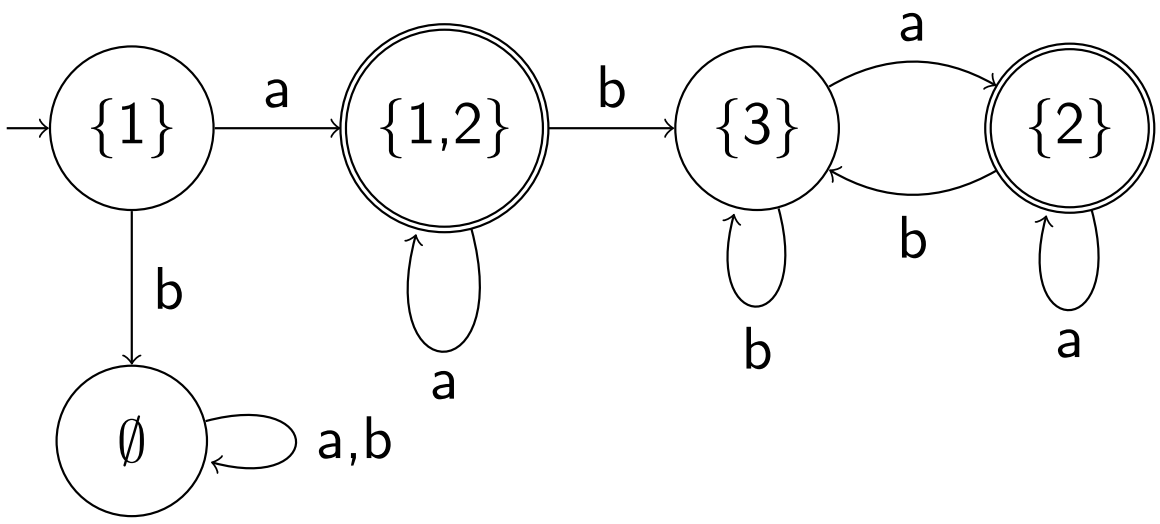
Preuve : à partir d'un AFN $(Q, \Sigma, q_I, \delta, F)$ on construit l'AFD $(Q', \Sigma, q'_I, \delta', F')$ où :

- $Q' = 2^Q$
- $q'_I = \{q_I\}$
- $\delta'(\{q_1, \ldots, q_n\}) = \delta'(q_1) \cup \cdots \cup \delta'(q_n)$
- $\{q_1, \ldots, q_n\} \in F'$ si $\{q_1, \ldots, q_n\} \cap F \neq \emptyset$

Exemple



admet pour déterminisation



Preuve de la clôture par complément

Étant donné un AFN $\mathcal{A} = (Q, \delta, q_I, F)$, on construit un AFD $\mathcal{A}'$ tel que

$$L(\mathcal{A}') = \Sigma^* \setminus L(\mathcal{A})$$

de la façon suivante :

- $Q' = 2^Q$
- $q'_I = \{q_I\}$
- $\delta'(\{q_1, \ldots, q_n\}, a) = \delta(q_1, a) \cup \cdots \cup \delta(q_n, a)$
- $F' = \{S \subseteq Q : S \cap F = \emptyset\}$

Clôture par homomorphisme

Théorème : Soit h un homomorphisme et $\mathcal{A}$ un AFN. Il existe un AFN $\mathcal{A}'$, tel que $L(\mathcal{A}') = h(L(A))$

Preuve (informelle) :
L'automate $\mathcal{A}'$ dispose d'un buffer pouvant contenir un mot de longueur au plus k, où $k = \max_{a \in \Sigma} |h(a)|$. Chaque nouvelle lettre lue est ajoutée en fin de buffer. L'automate dispose aussi d'un pointeur sur un état de $\mathcal{A}$. Si le début du buffer correspond à $h(a)$, alors l'automate peut avancer le pointeur en prenant une a-transition dans $\mathcal{A}$, et en même temps retirer $h(a)$ du début du buffer. Si le buffer devient trop gros, l'automate passe dans un état puits.

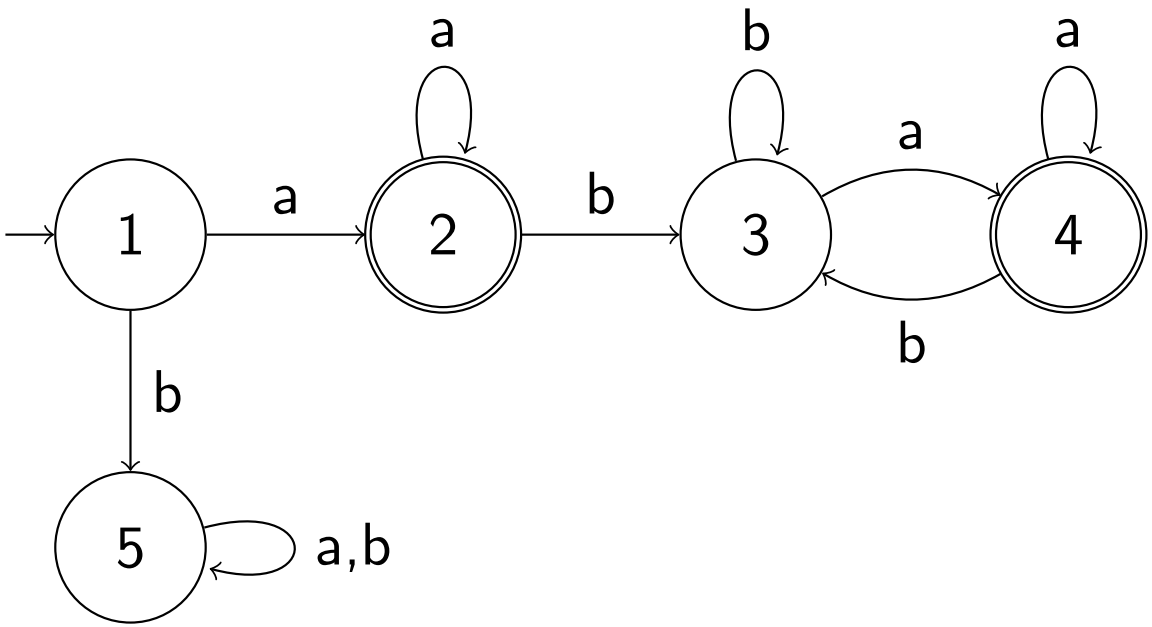
En TD : formaliser la construction de $\mathcal{A}'$

L'équivalence de Nerode

Deux états q, q' d'un AFD sont équivalents au sens de Nérode si

$$\text{pour tout } w \in \Sigma^*, \quad \delta^*(q, w) \in F \quad \text{ssi} \quad \delta^*(q', w) \in F.$$

exemple :



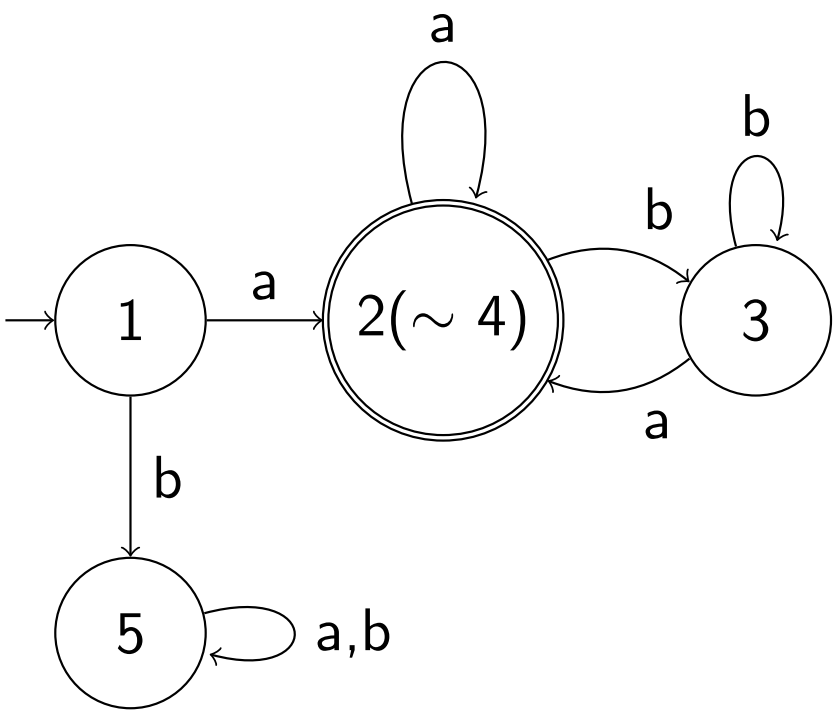
les états 2 et 4 sont équivalents, et ce sont les seuls

Minimisation

On note $\mathcal{A}/\sim$ l'automate quotient obtenu en fusionant les états équivalents au sens de Nérode.

Théorème :$\mathcal{A}/\sim$ est un AFD, et c'est le plus petit AFD qui reconnait $L(\mathcal{A})$ en comptant le nombre d'états.

exemple : la minimisation de l'automate précédent donne



Questions algorithmiques

Problèmes de décision

Un problème de décision se présente comme une question à laquelle il faut répondre oui ou non.

exemple : cette liste est-elle triée ? cet automate est-il minimal ? etc

Un problème de décision dépend d'une donnée (la liste, l'automate) qui a une certaine taille n (qui dépend de la façon dont la donnée est représentée)

Un problème admet en général plusieurs algorithmes dont l'efficacité varie. Pour les comparer, d'un point de vue théorique, on évalue leur complexité en temps ou en espace dans le pire cas ; cette complexité est en général proportionnelle à une certaine fonction $f(n)$. On parle d'algorithmes en $\mathcal{O}(n)$, en $\mathcal{O}(n^2)$, en $\mathcal{O}(2^n)$, etc.

Classes de complexité

La complexité d'un problème peut se penser comme la complexité du meilleur algorithme permettant de le résoudre.

On parle de problème résoluble en temps polynomial (PTIME), en temps polynomial par un algorithme non-déterministe (NP), en espace polynomial (PSPACE), en temps exponentiel (EXPTIME), etc.

Dans un cours de calculabilité et complexité on formalise ces notions et on établit les inclusions suivantes :

$$\textbf{PTIME} \subseteq \textbf{NP} \subseteq \textbf{PSPACE} \subseteq \textbf{EXPTIME}$$

On sait que certaines de ces inclusions sont strictes ($\textbf{PTIME} \subsetneq \textbf{EXPTIME}$), mais on ne sait pas lesquelles...

Problème de l'appartenance [1]

Le problème de l'appartenance se formule ainsi.

- Étant donné
  - un mot w, et
  - un automate fini non-déterministe
- a-t-on $w \in L(\mathcal{A})$?

Théorème : le problème de l'appartenance est dans **PTIME**.

1. aussi appelé problème du mot

Un algorithme

On fait une recherche en profondeur ; en pseudo-code Python :

```
def explore(q, w)
    if w is ϵ : return q in F
    for q2 in delta(q, w[0]) :
        if explore(q2, w[1:]) :
            return True
    return False

def member(w) : return explore(qinit, w)
```

Cet algorithme est en temps exponentiel (appels récursifs redondants), mais si l'on mémoïse la fonction `explore` on obtient un algorithme en temps polynomial : chaque combinaison (q, w) prend alors un temps constant

Le problème du vide

Étant donné un AFN $\mathcal{A}$, a-t-on $L(\mathcal{A}) = \emptyset$?

Théorème : Le problème du vide est dans **PTIME**

Preuve : on effectue une exploration (en largeur ou en profondeur) du graphe de l'automate en partant de l'état initial (en $\mathcal{O}(|\mathcal{A}|^2)$)

Le problème de l'universalité

Étant donné un AFN $\mathcal{A}$, a-t-on $L(\mathcal{A}) = \Sigma^*$?

Il faut tester qu'une infinité de mots sont dans $L(\mathcal{A})$: rien ne dit que ce soit un problème *décidable* !

Théorème : Le problème de l'universalité est dans **PSPACE**

Pour un AFD à n états, le problème de l'universalité peut être décidé en temps $\mathcal{O}(n^2)$ en se ramenant au problème du vide pour l'automate du complément.

Pour un AFN à n états : $L(\mathcal{A}) \neq \Sigma^*$ ssi il existe un mot w de longueur $|w| \leq 2^n$ tel que $w \notin L(\mathcal{A})$. L'algorithme consiste alors à vérifier tous les mots de longueur au plus 2^n
(on peut le faire en espace polynomial !).

Théorème de Kleene

Expression régulière

Les expressions régulières sont définies par récurrence

- $\emptyset$, ϵ, et a ($a \in \Sigma$) sont des expressions régulières
- si e, e_1, e_2 sont des expressions régulières alors e^*, $e_1.e_2$ et $e_1 + e_2$ sont des expressions régulières

Le langage $L(e)$ décrit par l'expression régulière est défini comme

- $L(\emptyset) = \emptyset$, $L(\epsilon) = \{\epsilon\}$, $L(a) = \{a\}$
- $L(e^*) = L(e)^*$, $L(e_1.e_2) = L(e_1).L(e_2)$, $L(e_1 + e_2) = L(e_1) \cup L(e_2)$

exemple : le langage des mots qui commencent et terminent par un a est décrit par $a + a.(a + b)^*.a$

Le théorème de Kleene

Théorème : Les langages reconnaissables sont exactement les langages réguliers (décrits par des expressions régulières)

On peut passer d'un automate à son expression régulière de diverses manières, la plus simple conceptuellement étant de simplifier le système d'équations associé à l'automate en appliquant le lemme d'Arden (transparent suivant).

De même on peut passer d'une expression régulière à un automate de diverses manières ; citons notamment :

- l'algorithme de Thompson [2] : par récurrence sur l'expression régulière (cf preuve de propriétés de clôture)
- les algorithmes de Glushkov, de Brzozowski, ou Antimirov : l'automate peut être construit à la volée

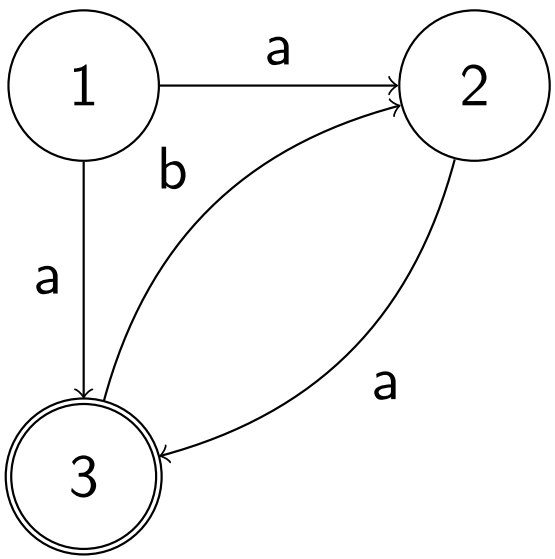
2. Ken Thompson : un des pères d'UNIX !

Lemme d'Arden et élimination

Lemme d'Arden

Satz : Soient $U, V, L \subseteq \Sigma^*$ des langages tels que $\epsilon \notin U$ et $L = U.L \cup V$. Alors $L = U^* V$.

Elimination



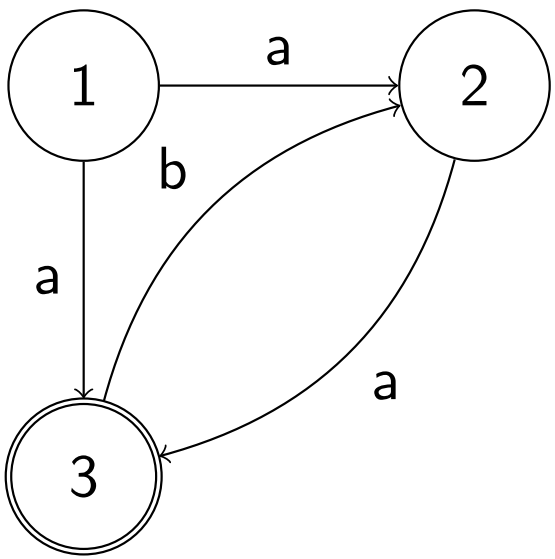
$$
\begin{aligned}
X_1 &= aX_2 + aX_3 \\
X_2 &= aX_3 \\
X_3 &= (ba)^*
\end{aligned}
$$

Lemme d'Arden et élimination

Lemme d'Arden

Satz : Soient $U, V, L \subseteq \Sigma^*$ des langages tels que $\epsilon \notin U$ et $L = U.L \cup V$. Alors $L = U^* V$.

Elimination



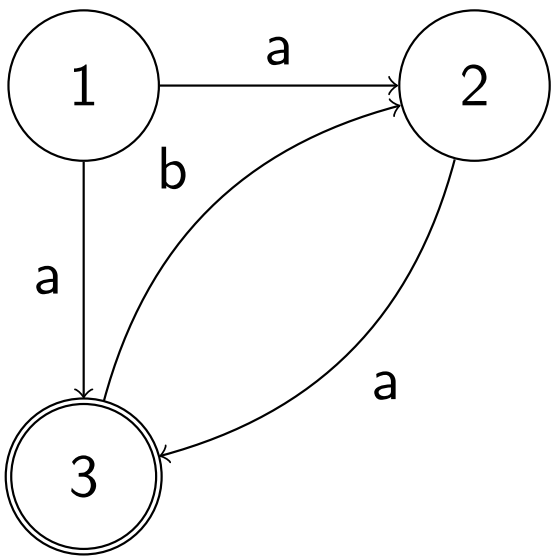
$$
\begin{aligned}
X_1 &= aX_2 + aX_3 \\
X_2 &= aX_3 \\
X_3 &= bX_2 + \epsilon
\end{aligned}
$$

Lemme d'Arden et élimination

Lemme d'Arden

Satz : Soient $U, V, L \subseteq \Sigma^*$ des langages tels que $\epsilon \notin U$ et $L = U.L \cup V$. Alors $L = U^* V$.

Elimination



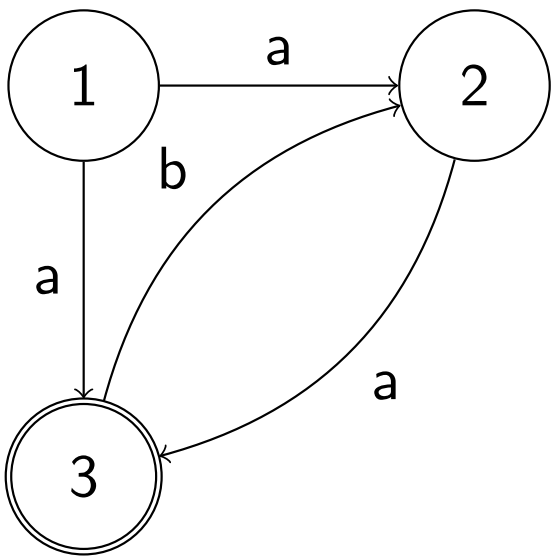
$$
\begin{aligned}
X_1 &= aX_2 + aX_3 \\
X_2 &= aX_3 \\
X_3 &= baX_3 + \epsilon
\end{aligned}
$$

Lemme d'Arden et élimination

Lemme d'Arden

Satz : Soient $U, V, L \subseteq \Sigma^*$ des langages tels que $\epsilon \notin U$ et $L = U.L \cup V$. Alors $L = U^* V$.

Elimination



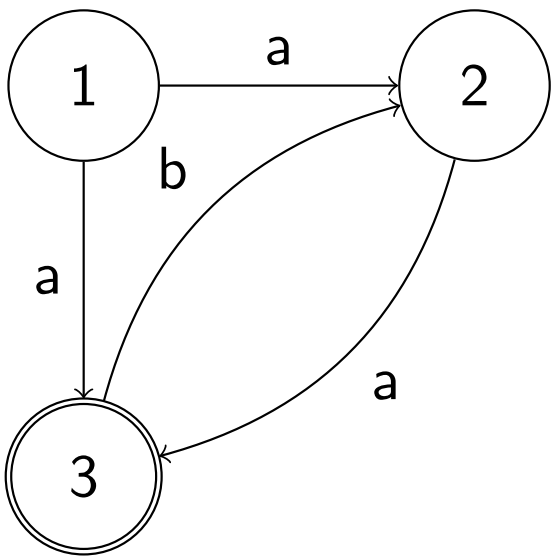
$$
\begin{aligned}
X_1 &= aX_2 + aX_3 \\
X_2 &= aX_3 \\
X_3 &= (ba)^*
\end{aligned}
$$

Lemme d'Arden et élimination

Lemme d'Arden

Satz : Soient $U, V, L \subseteq \Sigma^*$ des langages tels que $\epsilon \notin U$ et $L = U.L \cup V$. Alors $L = U^* V$.

Elimination



$$
\begin{aligned}
X_1 &= aa(ba)^* + a(ba)^* \\
X_2 &= a(ba)^* \\
X_3 &= (ba)^*
\end{aligned}
$$

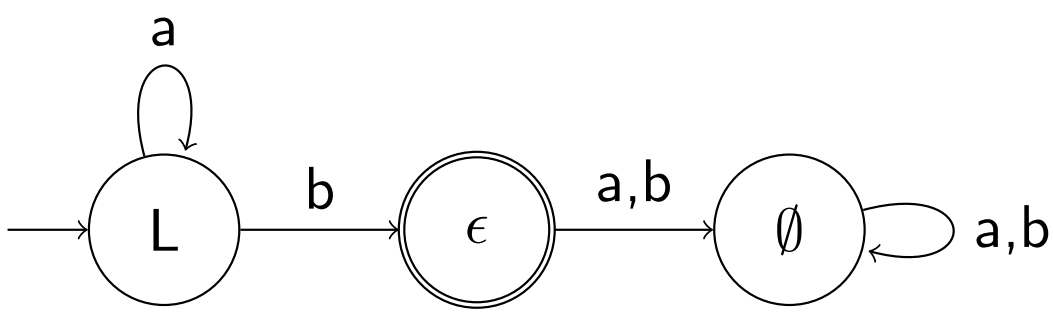
Notion de résidu

Le résidu du langage L par rapport au mot $u \in \Sigma^*$ est le langage

$$u^{-1}L = \{w \in \Sigma^* : uw \in L\}$$

Théorème : Un langage L est reconnaissable ssi il a un nombre fini de résidus. L'automate minimal qui reconnait L est celui dont les états sont les résidus de L avec $\delta(u^{-1}L,\ a) = (ua)^{-1}L$.

exemple : $L = a^*b$ a trois résidus distincts : $a^{-1}L = L$, $b^{-1}L = \{\epsilon\}$, et $(ba)^{-1}L = (bb)^{-1}L = \emptyset$



Les dérivées de Brzozowsi

La dérivée $D_u(e)$ de e par rapport à u est une expression régulière qui reconnait $u^{-1}L(e)$. Elle est *calculable* par récurrence.

- $D_{a_1 a_2 \ldots a_k}(e) = D_{a_k}(D_{a_{k-1}}(\ldots D_{a_1}(e)\ldots))$
- $D_a(\emptyset) = \emptyset$
- $D_a(\epsilon) = \emptyset$
- $D_a(a) = \epsilon$
- $D_a(b) = \emptyset$
- $D_a(e_1 + e_2) = D_a(e_1) + D_a(e_2)$
- $D_a(e_1.e_2) = D_a(e_1).e_2 + \downarrow_\epsilon(e_1).D_a(e_2)$
- $D_a(e^*) = D_a(e)e^*$

où $\downarrow_\epsilon (e) = \epsilon$ si $\epsilon \in L(e)$, et $\downarrow_\epsilon (e) = \emptyset$ sinon.

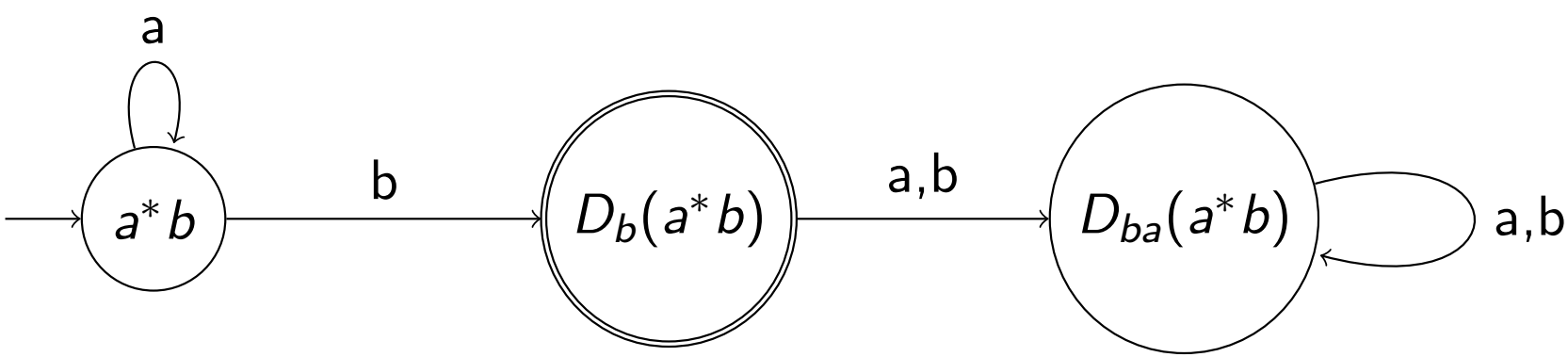
exemple :

$$\begin{aligned} D_a\big((a+b)^*\big) &= D_a(a+b).(a+b)^* \\ &= \big(D_a(a) + D_a(b)\big).(a+b)^* \\ &= (\epsilon + \emptyset).(a+b)^* \end{aligned}$$

De la regexp à l'automate (algorithme de Brzozowski)

On se donne e et on veut construire $\mathcal{A}$ tel que $L(\mathcal{A}) = L(e)$.

On calcule « toutes » les dérivées $D_u(e)$: ce sont les états de l'automate, on met une a transition entre $D_u(e)$ et $D_{ua}(e)$, et les états acceptants sont ceux pour lesquels $\downarrow_\epsilon (L_u(a)) = \{\epsilon\}$



Équivalence entre expressions régulières

Deux dérivées différentes peuvent correspondre à un même résidu, et il y a a priori une infinité de dérivées. Pour obtenir un automate fini, il faut donc savoir identifier si des expressions régulières définissent le même résidu, au moins de manière approchée (de manière exacte prendrait trop de temps). La méthode de Brzozowksi consiste à identifier les dérivées à l'aides de règles algébriques simples.

$$e_1 + (e_2 + e_3) \sim (e_1 + e_2) + e_3 \qquad e_1 + e_2 \sim e_2 + e_1 \qquad e_1 + e_1 \sim e_1$$

$$e + \emptyset \sim \emptyset + e \sim e \qquad e.\emptyset \sim \emptyset.e \sim \emptyset \qquad e.\epsilon \sim \epsilon.e \sim e$$

exemple : modulo $\sim$, on a une seule dérivée pour $(a+b)^*$

$$D_a\big((a+b)^*\big) = (\epsilon + \emptyset).(a+b)^* \sim (a+b)^*$$
$$D_b\big((a+b)^*\big) = (\emptyset + \epsilon).(a+b)^* \sim (a+b)^*$$

grep und perl

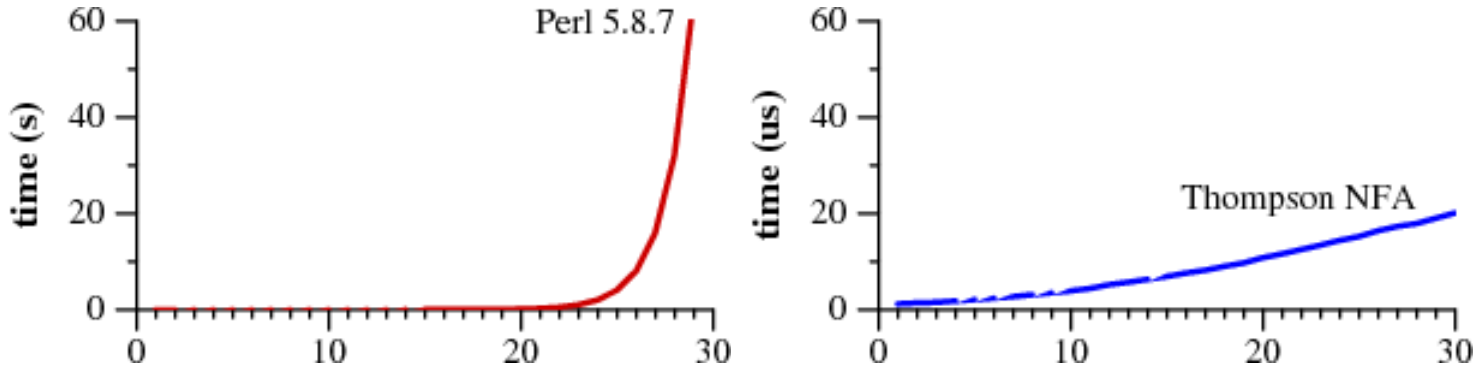
Grep et Perl utilisent deux algorithmes différents pour les gérer les expressions régulières, avec des performances différentes.

test : le mot

$$\underbrace{aa\ldots a}_{\text{n fois}}$$

est-il reconnu par

$$\underbrace{(a+\epsilon)(a+\epsilon)\ldots(a+\epsilon)}_{\text{n fois}}\underbrace{aa\ldots a}_{\text{n fois}}$$



Quelle : Russ Cox `http://swtch.com/~rsc`

Exercice de programmation

Écrire une fonction `match(w,e)` qui renvoie vrai si le mot w est reconnu par l'expression régulière e.

On prendra $\Sigma = \{a, \ldots, z\}$ et on codera w et e par des chaînes de caractères. Par simplicité, e sera noté en notation polonaise inversée.

exemples :
`match('abaa', 'ab+*')` $\rightarrow$ vrai car $abaa \in L\big((a+b)^*\big)$
`match('abaa', 'ab.*')` $\rightarrow$ faux, car $abaa \notin L\big((ab)^*\big)$