Automates, Langages et Logique 5. Minimisation

L2, Université d'Orléans — S1 2025/2026

Nicolas Ollinger

Rappels

Un automate fini déterministe $\mathcal{A} = (Q, A, \delta, q_0, F)$ reconnait le langage $L(\mathcal{A}) \in \operatorname{Rec} A^*$ des mots étiquettant des chemins acceptants de l'automate.

Un langage est **reconnaissable** s'il est reconnu par un AFD. Dans ce cas, il est reconnu par un AFD **complet** avec au plus deux états puits.

Deux automates sont **équivalents** s'ils reconnaissent le même langage.

Automate fini déterministe (AFD)

Définition Un automate fini déterministe (noté AFD dans la suite) $\mathcal A$ est un quintuplet (Q,A,δ,q_0,F) avec

- *Q* l'ensemble fini des **états**;
- *A* l'alphabet d'entrée;
- $\delta: Q \times A \to Q \cup \{\bot\}$ la fonction partielle de transition;
- q₀ ∈ Q | 'état initial;
- $F \subseteq Q$ l'ensemble des **états acceptants** de l'automate.

Un AFD est **complet** si sa fonction de transition est **totale**, c'est-à-dire définie pour toute paire état/lettre.

Fonction de transition étendue

La **fonction de transition** est étendue des **lettres** aux **mots** par récurrence.

La **fonction de transition étendue** applique successivement la fonction de transition aux différentes lettres qui composent le mot.

Pour tout état $q \in Q$, lettre $x \in A$ et mot $u \in A^*$, on pose :

$$\begin{split} \delta^*(q,\varepsilon) &= q \\ \delta^*(q,u\cdot x) &= \begin{cases} \delta(\delta^*(q,u),x) & \text{si } \delta^*(q,u) \neq \bot \\ \bot & \text{sinon} \end{cases} \end{split}$$

Intuitivement, $\delta^*(q, u)$ est l'état atteint en partant de l'état q et en lisant le mot u dans l'automate.

Chemin dans un automate

La fonction de transition δ décrit les transitions $q \stackrel{a}{\to} \delta(q,a)$ possibles dans l'automate.

Définition Un chemin dans un automate (Q, A, δ, q_0, F) est une suite finie de transitions successives $p_0 \xrightarrow{x_0} p_1 \xrightarrow{x_1} \cdots \xrightarrow{x_{n-1}} p_n$ où p_0 est l'état de départ, p_n est l'état d'arrivée et $\delta(p_i, x_i) = p_{i+1}$ pour tout i < n. Le mot $x_0 \cdots x_{n-1}$ est l'étiquette du chemin.

En utilisant la fonction de transition généralisée on s'autorise une notation plus concise $p_0 \xrightarrow{x_0 x_1 \cdots x_{n-1}} p_n = \delta^*(p_0, x_0 x_1 \cdots x_{n-1})$.

Langage reconnu

Définition Un chemin est acceptant pour un AFD lorsque l'état de départ est l'état initial et l'état d'arrivée un état acceptant.

$$q_0 \xrightarrow{u} q = \delta^*(q_0, u) \in F$$

Définition Un mot u est **accepté** par un automate \mathcal{A} s'il est l'étiquette d'un **chemin acceptant** de \mathcal{A} .

Définition Le langage reconnu par un automate $\mathcal A$ est l'ensemble $L(\mathcal A)$ des mots acceptés par $\mathcal A$.

Quotient

Définition Le quotient gauche d'un langage $L \subseteq A^*$ par un mot $u \in A^*$ est le langage $u^{-1}L = \{v \in A^* | uv \in L\}$.

Définition Le quotient droit d'un langage $L \subseteq A^*$ par un mot $u \in A^*$ est le langage $Lu^{-1} = \{v \in A^* | vu \in L\}$.

Ces notations sont étendues aux langages par

$$K^{-1}L = \{ v \in A^* | \exists u \in K \quad uv \in L \}$$

$$LK^{-1} = \{ v \in A^* | \exists u \in K \quad vu \in L \}$$

Quotients et automates finis

Soit $\mathcal A$ un **AFD complet** (Q,A,δ,q_0,F) reconnaissant un langage $L\in\operatorname{Rec} A^*$. Pour tout état $q\in Q$, notons L_q le langage associé à l'état q, c'est-à-dire

$$L_q = \{ v \in A^* | \delta^*(q, v) \in F \}$$

Lemme Pour tout $u \in A^*$ et $q = \delta^*(q_0, u)$, on a $u^{-1}L = L_q$.

Démonstration

$$\begin{split} u^{-1}L &= \{v \in A^* | uv \in L\} \\ &= \{v \in A^* | \delta^*(q_0, uv) \in F\} \\ &= \{v \in A^* | \delta^*(\delta^*(q_0, u), v) \in F\} \\ &= \{v \in A^* | \delta^*(q, v) \in F\} \\ &= L_q \end{split}$$

Caractérisation par les quotients

Proposition Un langage est **reconnaissable** si et seulement si il a un nombre fini de quotients gauche.

Démonstration

 \Rightarrow L'ensemble des quotients est contenu dans l'ensemble fini des langages L_q des états d'un automate fini qui le reconnait.

Caractérisation par les quotients

Proposition Un langage est **reconnaissable** si et seulement si il a un **nombre fini de quotients gauche**.

Démonstration

 \Rightarrow L'ensemble des quotients est contenu dans l'ensemble fini des langages L_q des états d'un automate fini qui le reconnait.

← Construisons un automate à partir de l'ensemble des quotients!

Automate minimal

Définition Soit L un langage sur un alphabet A. L'automate minimal \mathcal{A}_L de L est l'automate (Q, A, δ, q_0, F) où :

$$Q = \{u^{-1}L | u \in A^*\}$$

$$q_0 = \{L\}$$

$$F = \{u^{-1}L | u \in L\}$$

$$\delta(u^{-1}L, x) = (ux)^{-1}L$$

 $\forall u \in A^*, x \in A$

Automate minimal

Définition Soit L un langage sur un alphabet A. L'automate minimal \mathcal{A}_L de L est l'automate (Q, A, δ, q_0, F) où :

$$\begin{split} Q &= \left\{ u^{-1} L \middle| u \in A^* \right\} \\ q_0 &= \left\{ L \right\} \\ F &= \left\{ u^{-1} L \middle| u \in L \right\} \\ \delta(u^{-1} L, x) &= (ux)^{-1} L & \forall u \in A^*, x \in A \quad . \end{split}$$

Lemme L'automate minimal \mathcal{A}_L reconnaît le langage L.

Minimalité

Soit (Q, A, δ, q_0, F) un AFD reconnaissant $L \subseteq A^*$.

Lemme Soient $u,v\in A^*$. Si $u^{-1}L\neq v^{-1}L$ alors $\delta^*(q_0,u)\neq \delta^*(q_0,v)$.

Minimalité

Soit (Q, A, δ, q_0, F) un AFD reconnaissant $L \subseteq A^*$.

Lemme Soient $u, v \in A^*$. Si $u^{-1}L \neq v^{-1}L$ alors $\delta^*(q_0, u) \neq \delta^*(q_0, v)$.

Lemme Soient $p,q\in Q$ et $x\in A$. Si $\delta(p,x)=q$ alors $L_q=x^{-1}L_p$.

Minimalité

Soit (Q, A, δ, q_0, F) un AFD reconnaissant $L \subseteq A^*$.

Lemme Soient $u,v\in A^*$. Si $u^{-1}L\neq v^{-1}L$ alors $\delta^*(q_0,u)\neq \delta^*(q_0,v)$.

Lemme Soient $p,q\in Q$ et $x\in A$. Si $\delta(p,x)=q$ alors $L_q=x^{-1}L_p$.

Proposition L'automate minimal \mathcal{A}_L d'un langage reconnaissable L est l'unique (à renommage des états) automate fini déterministe complet de taille minimale reconnaissant L.

Congruence et automate quotient

Soit \mathcal{A} un **AFD complet** (Q, A, δ, q_0, F) et $L = L(\mathcal{A})$.

Définition Une congruence sur \mathcal{A} est une relation d'équivalence \sim sur l'ensemble Q qui vérifie pour tous $p, q \in Q$ et $x \in A$:

- (i) $p \sim q \Longrightarrow (p \in F \Leftrightarrow q \in F)$;
- (ii) $p \sim q \Longrightarrow \delta(p, x) \sim \delta(q, x)$.

Congruence et automate quotient

Soit \mathcal{A} un **AFD complet** (Q, A, δ, q_0, F) et $L = L(\mathcal{A})$.

Définition Une congruence sur \mathcal{A} est une relation d'équivalence \sim sur l'ensemble Q qui vérifie pour tous $p,q\in Q$ et $x\in A$:

- (i) $p \sim q \Longrightarrow (p \in F \Leftrightarrow q \in F)$;
- (ii) $p \sim q \Longrightarrow \delta(p, x) \sim \delta(q, x)$.

On note [q] la classe d'équivalence de q pour \sim .

Définition L'automate quotient \mathcal{A}/\sim est l'AFD complet $(Q/\sim, A, \delta', [q_0], [F])$ où $\delta'([q], x) = [\delta(q, x)]$.

Congruence et automate quotient

Soit \mathcal{A} un **AFD complet** (Q, A, δ, q_0, F) et $L = L(\mathcal{A})$.

Définition Une congruence sur \mathcal{A} est une relation d'équivalence \sim sur l'ensemble Q qui vérifie pour tous $p,q\in Q$ et $x\in A$:

- (i) $p \sim q \Longrightarrow (p \in F \Leftrightarrow q \in F)$;
- (ii) $p \sim q \Longrightarrow \delta(p, x) \sim \delta(q, x)$.

On note [q] la classe d'équivalence de q pour \sim .

Définition L'automate quotient \mathcal{A}/\sim est l'AFD complet $(Q/\sim, A, \delta', [q_0], [F])$ où $\delta'([q], x) = [\delta(q, x)]$.

Lemme L'automate quotient A/\sim reconnaît le langage L.

Soit \mathcal{A} un **AFD complet** (Q, A, δ, q_0, F) et $L = L(\mathcal{A})$.

Définition La congruence de Nerode de \mathcal{A} est définie pour tous $p,q\in Q$ par $p\sim q\equiv_{\mathsf{def}}\forall w\in A^*\quad \delta^*(p,w)\in F\Leftrightarrow \delta^*(q,w)\in F$

Soit \mathcal{A} un **AFD complet** (Q, A, δ, q_0, F) et $L = L(\mathcal{A})$.

Définition La congruence de Nerode de \mathcal{A} est définie pour tous $p, q \in Q$ par $p \sim q \equiv_{\mathsf{def}} \forall w \in A^* \quad \delta^*(p, w) \in F \Leftrightarrow \delta^*(q, w) \in F$

Proposition L'automate minimal \mathcal{A}_L est égal à l'automate quotient \mathcal{A}/\sim où \sim est la congruence de Nerode.

Soit \mathcal{A} un **AFD complet** (Q, A, δ, q_0, F) et $L = L(\mathcal{A})$.

Définition La congruence de Nerode de \mathcal{A} est définie pour tous $p, q \in Q$ par $p \sim q \equiv_{\mathsf{def}} \forall w \in A^* \quad \delta^*(p, w) \in F \Leftrightarrow \delta^*(q, w) \in F$

Proposition L'automate minimal \mathcal{A}_L est égal à l'automate quotient \mathcal{A}/\sim où \sim est la congruence de Nerode.

On peut donc construire l'automate minimal en fusionnant des états équivalents!

Soit \mathcal{A} un **AFD complet** (Q, A, δ, q_0, F) et $L = L(\mathcal{A})$.

Définition La congruence de Nerode de \mathcal{A} est définie pour tous $p,q\in Q$ par $p\sim q\equiv_{\mathsf{def}} \forall w\in A^* \quad \delta^*(p,w)\in F \Leftrightarrow \delta^*(q,w)\in F$

Proposition L'automate minimal \mathcal{A}_L est égal à l'automate quotient \mathcal{A}/\sim où \sim est la congruence de Nerode.

On peut donc construire l'automate minimal en fusionnant des états équivalents!

Mais en pratique...comment faire?

Méthode itérative de Moore

Soit \mathcal{A} un **AFD complet** (Q, A, δ, q_0, F) et $L = L(\mathcal{A})$.

La suite \sim_k de relations d'équivalences sur Q est définie récursivement par :

$$p \sim_0 q \equiv_{\mathsf{def}} (p \in F \Leftrightarrow q \in F)$$
$$p \sim_{i+1} q \equiv_{\mathsf{def}} p \sim_i q \land \forall x \in A \quad \delta(p, x) \sim_i \delta(q, x)$$

Méthode itérative de Moore

Soit \mathcal{A} un **AFD complet** (Q, A, δ, q_0, F) et $L = L(\mathcal{A})$.

La suite \sim_k de relations d'équivalences sur Q est définie récursivement par :

$$p \sim_0 q \equiv_{\mathsf{def}} (p \in F \Leftrightarrow q \in F)$$
$$p \sim_{i+1} q \equiv_{\mathsf{def}} p \sim_i q \land \forall x \in A \quad \delta(p, x) \sim_i \delta(q, x)$$

Proposition La suite (\sim_i) atteint un point fixe \sim_k pour un k < |Q|. Ce point fixe est égal à la congruence de Nerode.

Idée Définir $L_p^{(k)} = L_p \cap A^{\leqslant k}$. Montrer que $L_p^{(n-2)}$ caractérise L_p pour L.

Les classes d'équivalence d'une relation d'équivalence sur un ensemble Xforment une partition de X.

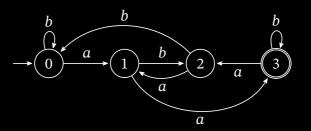
Une partition \mathcal{P} d'un ensemble X est un ensemble de parties non vides, $\begin{array}{c|c} \textbf{disjointes} \ \text{de} \ X \ \text{qui} \ \textbf{couvrent} \ X. \\ & | \ Y = X \qquad Y \neq \emptyset \qquad Y \cap Y' = \emptyset \qquad \forall Y, Y' \in \mathcal{P} \end{array}$

$$\bigcup_{Y \in \mathcal{P}} Y = X' \qquad Y \neq \emptyset \qquad Y \cap Y' = \emptyset \qquad \forall Y, Y' \in \mathcal{F}$$

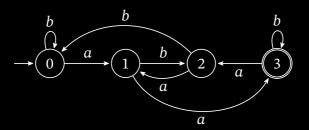
Les **éléments** d'une partition sont appelés des **classes** ou des **blocs**.

L'ensemble des partitions d'un ensemble forme un treillis complet pour la relation plus fin/plus grossier: une partition \mathcal{P} est plus fine qu'une partition \mathcal{P}' si les classes de \mathcal{P}' sont des unions de classes de \mathcal{P} .

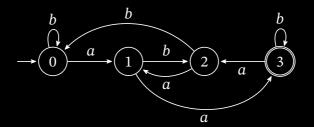
La **congruence de Nerode** est la congruence la plus grossière.



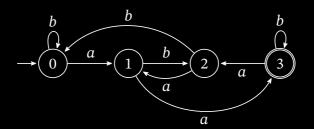
δ					
q	а	b			
$\rightarrow 0$	1	0			
1	3	2			
2	1	0			
3 *	2	3			



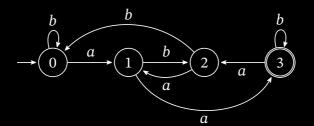
δ			\sim_0		
q	а	b	[q]		
$\rightarrow 0$	1	0	A		
1	3	2	A		
2	1	0	A		
3 *	2	3	B		



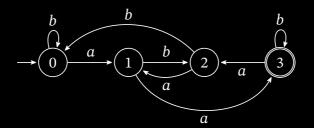
δ			\sim_0	δ/	\sim_0		
q	а	b	[q]	а	b		
$\rightarrow 0$	1	0	\overline{A}	\overline{A}	\overline{A}		
1	3	2	A	B	\boldsymbol{A}		
2	1	0	A	\boldsymbol{A}	\boldsymbol{A}		
3 *	2	3	B	\boldsymbol{A}	B		



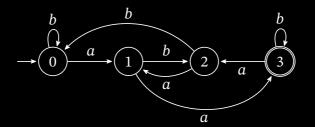
δ			\sim_0	δ/	\sim_0	~1	
q	а	b	[q]	а	b	[q]	
$\rightarrow 0$	1	0	A	\overline{A}	\overline{A}	A	
1	3	2	A	B	\boldsymbol{A}	B	
2	1	0	A	\boldsymbol{A}	\boldsymbol{A}	A	
3 *	2	3	B	\overline{A}	\overline{B}	C	



δ			\sim_0	δ	\sim_0	~ ₁	$\delta/$	\sim_1	
q	а	b	[q]	а	b	[q]	а	b	
→ 0	1	0	\overline{A}	\overline{A}	\overline{A}	A	В	\overline{A}	
1	3	2	A	B	\boldsymbol{A}	B	C	\boldsymbol{A}	
2	1	0	A	\boldsymbol{A}	\boldsymbol{A}	A	B	\boldsymbol{A}	
3 *	2	3	B	A	В	C	\boldsymbol{A}	C	

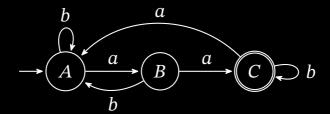


δ			\sim_0	δ/	\sim_0	~ ₁	δ/	\sim_1	~2
q	а	b	[q]	а	b	[q]	а	b	[q]
	1	0	A	\overline{A}	\overline{A}	A	В	\overline{A}	A
1	3	2	A	B	\boldsymbol{A}	B	C	\boldsymbol{A}	B
2	1	0	A	\boldsymbol{A}	\boldsymbol{A}	A	B	\boldsymbol{A}	A
3 *	2	3	B	\overline{A}	B	C	\overline{A}	\overline{C}	\overline{C}



δ			\sim_0	δ/	\sim_0	~1	δ/	\sim_1	~2
q	а	b	[q]	а	b	[q]	а	b	[q]
$\rightarrow 0$	1	0	A	\overline{A}	\overline{A}	A	В	\overline{A}	\boldsymbol{A}
1	3	2	A	B	\boldsymbol{A}	B	C	\boldsymbol{A}	B
2	1	0	A	\boldsymbol{A}	\boldsymbol{A}	A	B	\boldsymbol{A}	\boldsymbol{A}
3 *	2	3	B	\overline{A}	\overline{B}	C	\overline{A}	\overline{C}	C

δ			\sim_0	δ/	~_0	~ ₁	δ/	′ ∼ ₁	~2
q	а	b	[q]	а	b	[q]	а	b	[q]
$\rightarrow 0$	1	0	A	\boldsymbol{A}	\boldsymbol{A}	A	В	\boldsymbol{A}	A
1	3	2	A	B	\boldsymbol{A}	B	C	\boldsymbol{A}	B
2	1	0	A	\boldsymbol{A}	\boldsymbol{A}	A	B	\boldsymbol{A}	A
3 *	2	3	B	\boldsymbol{A}	B	C	\boldsymbol{A}	C	C



1: function MINIMIZE-MOORE(
$$\{0, ..., n-1\}, A, \delta, q_0, F$$
)
2: # Cas dégénérés

return $(\{0\}, A, (0, x) \mapsto 0, 0, \{0\})$

if
$$F = \emptyset$$
 then

4:

5:

6:

7:

8:

Initialisation

$$\mathbf{return}\ (\{0\}, A, (0, x) \mapsto 0, 0, \emptyset)$$

$$\emptyset$$
 then

$$\emptyset$$
 then

if $F = \{0, ..., n - 1\}$ then

for all $p \in 0 \dots n-1$ do

 $\pi'(p) = \begin{cases} 0 & \text{si } p \notin F \\ 1 & \text{si } p \in F \end{cases}$

10: # Calculitératif
11:
$$\pi = \bot$$

12: while $\pi \neq \pi'$ do
13: $\pi = \pi'$
14: # Calcule π' tel que $\pi'(p) = \pi'(q)$ si et seulement si $\pi(p) = \pi(q)$
 $\pi(\delta(p, x)) = \pi(\delta(q, x))$ pour tout $x \in A$
15: $\pi' = \text{REFINE-MOORE}(\pi, n, A, \delta)$
16: return $(\{0, ..., n-1\}, A, \delta, q_0, F) / \pi$

```
17: function Refine-Moore(\pi, n, \{a_1, ..., a_k\}, \delta)
        for all p \in 0 \dots n-1 do
18:
             V(p) = (\pi(p), \pi(\delta(p, a_1)), ..., \pi(\delta(p, a_k)))
19:
        # Calcule une permutation qui trie selon V
20:
         \sigma = RADIX-SORT(V)
21:
         i = 0
22:
         \pi'(\sigma(0)) = i
23:
        for all p \in 1 \dots n-1 do
24:
             if V(p) \neq V(p-1) then
25:
             i = i + 1
26:
             \pi(\sigma(p)) = i
27:
        return \pi'
```

28:

Analyse du temps d'exécution

Chaque étape de **raffinement** de l'algorithme de Moore s'exécute en temps $\Theta(kn)$ où k=|A| et n=|Q|.

Le nombre d'étapes de raffinement est au plus n-2, une valeur atteinte dans les pires cas. La **complexité au pire** de l'algorithme de minimisation de Moore est donc en $O(kn^2)$.

Analyse du temps d'exécution

Chaque étape de **raffinement** de l'algorithme de Moore s'exécute en temps $\Theta(kn)$ où k=|A| et n=|Q|.

Le nombre d'étapes de raffinement est au plus n-2, une valeur atteinte dans les pires cas. La **complexité au pire** de l'algorithme de minimisation de Moore est donc en $O(kn^2)$.

Peut-on espérer mieux?

Avancer à petits pas

Soit (Q, A, δ, q_0, F) un automate fini déterministe émondé et complété.

Définition Soient $B, C \subseteq Q$ et $x \in A$. On observe les transitions $\delta^*(B, x)$ qui aboutissent en C:

- (1) La partie B est stable pour (C, x) si $\delta^*(B, x) \subseteq C$ ou si $\delta^*(B, x) \cap C = \emptyset$.
- (2) Sinon la paire (C, x) coupe la partie B en deux parties $B_1 = \{q \in B | \delta(q, x) \in C\}$ et $B_2 = \{q \in B | \delta(q, x) \notin C\}$.

Une partition de Q est **stable** pour (C, x) si chacune de ses classe le sont.

Avancer à petits pas

Soit (Q, A, δ, q_0, F) un automate fini déterministe émondé et complété.

Définition Soient $B, C \subseteq Q$ et $x \in A$. On observe les transitions $\delta^*(B, x)$ qui aboutissent en C:

- (1) La partie B est stable pour (C, x) si $\delta^*(B, x) \subseteq C$ ou si $\delta^*(B, x) \cap C = \emptyset$.
- (2) Sinon la paire (C, x) coupe la partie B en deux parties $B_1 = \{q \in B | \delta(q, x) \in C\}$ et $B_2 = \{q \in B | \delta(q, x) \notin C\}$.

Une partition de Q est **stable** pour (C, x) si chacune de ses classe le sont.

Proposition Une partition de Q est une **congruence** si et seulement si elle est **compatible** avec F et **stable** pour chaque (Y, x) où Y est une de ses parties et x parcourt A.

1: **function** MINIMIZE-MOORE-BIS
$$(Q, A, \delta, q_0, F)$$

2: $\mathcal{P} = \{F, Q \setminus F\}$

$$S = \emptyset$$

4:

5:

6:

7:

8:

9:

10:

$$rall x \in A do$$

$$x \in A$$
 do

ajouter
$$(F, :$$

outer
$$(F, I)$$

ajouter
$$(F, \lambda)$$

ajouter
$$(F, x)$$
 et $(Q \setminus F, x)$ à S

ajouter
$$(F, x)$$

(C, x) = Pop(S)

for all $B \in \mathcal{P}$ do

 $C' = \{ q \in Q | \delta(q, x) \in C \}$

all
$$x \in A$$
 up

an
$$x \in A$$
 us

$$\mathsf{rall}\ x \in A$$

while $S \neq \emptyset$ do

for all
$$x \in A$$
 do

or all
$$x \in A$$

$$\mathbf{all} \ x \in A \ \mathbf{do}$$

$$x \in A do$$

$$x \in A do$$

$$all \ x \in A \ c$$

$$all x \in A$$

$$\mathbf{all} \ x \in A \mathbf{d}$$

$$c \in A do$$

$$\in A$$
 do

$$A$$
 do







Refine-Moore-Bis(C', B, \mathcal{P}, S, A)

```
11: function Refine-Moore-Bis(C', B, \mathcal{P}, S, A)
```

$$B_1 = B \cap C'$$

$$B_2 = B \setminus B_1$$

else

Est-ce que ça coupe?

$$D \perp \alpha$$

if
$$B_1 \neq \emptyset \land B_2 \neq \emptyset$$
 then

$$2 \neq \emptyset$$
 then $2 \neq \emptyset$

$$B$$
 par B_1 e

$$B$$
 par B_1 e

remplacer
$$B$$
 par B_1 et B_2 dans $\mathcal P$

for all
$$x \in A$$
 do

if
$$(B, x) \in S$$
 then

remplacer
$$(B, x)$$
 par (B_1, x) et (B_2, x) dans S

e ajouter
$$(B_1, x)$$
 et (B_2, x) à S

12:

13:

14:

15:

17:

18:

19:

20:

21:

Algorithme d'Hopcroft

Lemme Soit $x \in A$. Si $B \subseteq Q$ et $C = C_1 \cup C_2$ avec $C_1 \cap C_2 = \emptyset$ alors:

- (1) Si B est **stable** pour (C_1, x) et (C_2, x) alors B est **stable** pour (C, x);
- (2) Si B est stable pour (C_1, x) et (C, x) alors B est stable pour (C_2, x) .

Ce lemme est la clé de l'algorithme d'Hopcroft qui permet de diminuer la quantité de travail à faire pour le raffinement et d'atteindre une complexité au pire en $O(kn\log n)$.

1: function MINIMIZE-HOPCROFT
$$(Q, A, \delta, q_0, F)$$

2: $\mathcal{P} = \{F, Q \setminus F\}$

$$S =$$

for all
$$x \in A$$
 do

(C, x) = Pop(S)

for all $B \in \mathcal{P}$ do

 $C' = \{ q \in Q | \delta(q, x) \in C \}$

Refine-Hopcroft(C', B, \mathcal{P}, S, A)

air
$$x \in H$$
 do

ajouter
$$(\min(F, Q \setminus F), x)$$
 à S

while $S \neq \emptyset$ do

an
$$x \in A$$
 do

rall
$$x$$

$$\mathbf{all} \ x \in A \ \mathbf{do}$$

rall
$$x$$

$$\mathbf{all} \ x \in$$

$$\mathbf{all} \ x \in$$

$$\mathbf{all} x$$

$$S = V$$

$$S =$$

$$S =$$



2:

4:

5:

6:

7:

8:

9:

10:









```
11: function Refine-Hopcroft (C', B, \mathcal{P}, S, A)
        B_1 = B \cap C'
        B_2 = B \setminus B_1
```

$$\neq \emptyset \land B_{\circ} \neq \emptyset$$
 then

if
$$B_1 \neq \emptyset \land B_2 \neq \emptyset$$
 then
remplacer B par B_1 et B_2 dans \mathcal{P}

13:

14:

15:

16:

17:

18:

19:

20:

21:

for all
$$x \in A$$
 do

else

if
$$(B, x) \in S$$
 then

remplacer
$$(B, x)$$
 par (B_1, x) et (B_2, x) dans S

ajouter
$$(\min(B_1, B_2), x)$$
 à S

Mise en œuvre

Attention! L'algorithme d'Hopcroft possède une bonne complexité... seulement si les structures de données utilisées ont elles aussi la bonne complexité!

En pratique, cet algorithme est subtil à mettre en œuvre. On trouve d'ailleurs plusieurs articles dans la littérature qui expliquent comment faire cette mise en œuvre.

Bonne nouvelle! En 2024, on peut faire plus simple et efficace.

La recherche en marche

L'informatique est une discipline scientifique jeune et active!

Les connaissances continuent de progresser, y compris en algorithmique, y compris sur de vieux problèmes comme celui de la minimisation (les travaux d'Hopcroft datent de 1971).

On ne connaît toujours pas la complexité du problème de minimisation.

ON THE AVERAGE COMPLEXITY OF MOORE'S STATE MINIMIZATION ALGORITHM

FRÉDÉRIQUE BASSINO 1 AND JULIEN DAVID 2 AND CYRIL NICAUD 2

¹ LIPN UMR 7030, Université Paris 13 - CNRS, 99, avenue Jean-Baptiste Clément, 93430 Villetaneuse. France.

E-mail address: Frederique.Bassino@lipn.univ-paris13.fr

² Institut Gaspard Monge, Université Paris Est, 77454 Marne-la-Vallée Cedex 2, France E-mail address: Julien.David@univ-paris-est.fr,Cyril.Nicaud@univ-paris-est.fr

ABSTRACT. We prove that, for any arbitrary finite alphabet and for the uniform distribution over deterministic and accessible automata with n states, the average complexity of Moore's state minimization algorithm is in $\mathcal{O}(n\log n)$. Moreover this bound is tight in the case of unary automata.

The Average Complexity of Moore's State Minimization Algorithm is $\mathcal{O}(n \log \log n)$ *

Julien David

Institut Gaspard Monge, Université Paris Est 77454 Marne-la-Vallée Cedex 2, France.

Abstract. We prove that the average complexity, for the uniform distribution on complete deterministic automata, of Moore's state minimization algorithm is $\mathcal{O}(n \log \log n)$, where n is the number of states in the input automata.

Minimizing incomplete automata

MARIE-PIERRE BÉAL¹ and MAXIME CROCHEMORE²¹

Université Paris-Est, Institut Gaspard-Monge
 Marne-la-Vallée Cedex 2, France, beal@univ-mlv.fr
 King's College London, Strand, London WC2R 2LS, UK
 maxime.crochemore@kcl.ac.uk

Abstract. We develop a $O(m \log n)$ -time and O(k+n+m)-space algorithm for minimizing incomplete deterministic automata, where n is the number of states, m the number of edges, and k the size of the alphabet. Minimization reduces to the partial functional coarsest partition problem. Our algorithm is a slight variant of Hopcroft's algorithm for minimizing deterministic complete automata.

Keywords: algorithms, automata, minimization, partitioning.



Contents lists available at SciVerse ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl



Fast brief practical DFA minimization

Antti Valmari

Tampere University of Technology, Department of Software Systems, PO Box 553, FI-33101 Tampere, Finland

ARTICLE INFO

Article history: Received 27 May 2011 Received in revised form 2 December 2011 Accepted 5 December 2011 Available online 7 December 2011 Communicated by I. Torán

Keywords:
Algorithms
Formal languages
Data structures
Deterministic finite automata
Partition refinement

$A\ B\ S\ T\ R\ A\ C\ T$

Minimization of deterministic finite automata has traditionally required complicated programs and correctness proofs, and taken $O(nk\log n)$ time, where n is the number of states and k the size of the alphabet. Here a short, memory-efficient program is presented that runs in $O(n + m\log m)$, or even in $O(n + m\log n)$, time, where m is the number of transitions. The program is complete with input, output, and the removal of irrelevant parts of the automaton. Its invariant-style correctness proof is relatively short.

 $\ensuremath{\text{@}}$ 2011 Elsevier B.V. All rights reserved.

Algorithme de Valmari

Idée Partitionner les états et les transitions

L'algorithme de Valmari fonctionne pour des automates finis déterministes partiels.

Sa complexité est $O(n + m \log n)$ où n est le nombre d'états et m le nombre de transitions (donc $m \le kn$ où k est le nombre de lettres).

Pour distinguer les deux partitions, on utilise le vocabulaire de **blocs** et de **cordes** pour les classes respectives des partitions des états et des transitions.

À l'initialisation les blocs sont partitionnés selon F et les cordes selon les étiquettes des transitions.

```
1: function MINIMIZE-VALMARI(Q, T, F)
```

2:
$$\mathcal{P} = \text{Initial-Block-Partition}(Q, F)$$

3: $\mathcal{T} = \text{Initial-Cord-Partition}(T)$

while
$$\mathcal{F}$$
 contient une corde non traitée C do

marquer
$$C$$
 comme traitée
SPLIT-BLOCKS (\mathcal{P}, C)

while
$$\mathcal{P}$$
 contient un bloc non traité B do marquer B comme traité

9: SPLIT-CORDS
$$(\mathcal{T}, T, B)$$

return
$$(Q, T, F)/\mathcal{P}$$

6:

7:

8:

11: **function** SPLIT-BLOCKS(
$$\mathcal{P}, C$$
)

CLEAR-MARKS
$$(\mathcal{P})$$

13:

14:

15:

17:

18:

21:

for all
$$p \stackrel{a}{\rightarrow} q \in C$$
 do
$$MARK(\mathcal{P}, p)$$

$$igsquare$$
 Mark (\mathcal{P},p)
Split-with-Marks (\mathcal{P})

16: **function** SPLIT-CORDS(\mathcal{T}, T, B)

CLEAR-MARKS
$$(\mathcal{F})$$
for all $q \in B$ do

for all
$$p \xrightarrow{a} q \in T$$
 do

MARK $(\mathcal{T}, p \xrightarrow{a} q)$

SPLIT-WITH-MARKS (\mathcal{T})

22: **function** SPLIT-WITH-MARKS
$$(\mathcal{P})$$

for all
$$Y \in \mathcal{P}$$
 do

24:

25:

26:

27:

28:

 $Y_1 = Marked-Elements(\mathcal{P})$

if $Y_1 \neq \emptyset \land Y_2 \neq \emptyset$ then

 $Y_2 = \text{Unmarked-Elements}(\mathcal{P})$

remplacer Y par $\max(Y_1, Y_2)$ dans \mathcal{P}

aiouter min (Y_1, Y_2) , non traité, à \mathcal{P}

Démonstration

Lemme Si p et q sont dans deux blocs différents alors $L_p \neq L_q$.

Lemme Si $p \xrightarrow{x} q$ et $p' \xrightarrow{y} q'$ sont dans deux cordes différentes alors ou bien $x \neq y$ ou bien $L_q \neq L_q'$.

Lemme Soient p et q deux états d'un même bloc. Si p et q ont une transition pour une même lettre x dans deux cordes distinctes C et C' alors C ou C' n'est pas traitée. Si p a une transition pour une lettre x mais pas q alors la transition est dans une corde non traitée.

Lemme Soient $p \xrightarrow{x} q$ et $p' \xrightarrow{y} q'$ deux transitions d'une **même corde**. Si p et p' sont dans des **blocs distincts** B et B' alors B ou B' n'est **pas traité**.

Structure de données pour les partitions

La clé pour obtenir la bonne complexité avec l'algorithme de Valmari est une structure de donnée pour les **partitions raffinables**. Valmari propose une solution reposant uniquement sur l'utilisation de tableaux.

La mise en œuvre est très facile!

La complexité est aussi bonne que Hopcroft!

Les constantes sont meilleures!

Algorithmes à connaître

- (1) émondage et complétion
- (2) déterminisation (automate des parties)
- (3) élimination des ε -transitions
- (4) algorithme de Brzozowski et McCluskey (AFN → regexp)
- (5) algorithme de Glushkov (regexp \rightarrow AFN)
- (6) algorithme de minimisation de Moore