Minimisation d'un automate

Lucas Willems

30 novembre 2016

Table des matières

Introduction

Le problème consiste à trouver, pour un langage L donné, l'automate minimal qui reconnait ce langage i.e. l'automate avec le minimum d'état qui le reconnait.

Pour ce faire, tous les algorithmes présentés vont utiliser la congruence de Nerode définie comme suit :

Définition 1 (Congruence de Nerode)**.** *Soit $\mathcal{A} = (Q, \Sigma, i, F, \delta)$ déterministe. La congruence de Nerode est définie pour tous états q et q' par :*

$$q \sim q' \iff \forall w \in \Sigma^* \ \ (q \cdot w \in F \Leftrightarrow q' \cdot w \in F)$$

L'utilité de cette congruence est donnée par la proposition suivante :

Proposition 1. *Soit $\mathcal{A}$ un automate déterministe acceptant L. L'automate minimal $\mathcal{A}_\mathcal{L}$ est égal à $\mathcal{A}/\sim$ où $\sim$ est la congruence de Nerode de $\mathcal{A}$.*

Ainsi, cette proposition nous permet de ramener le problème du calcul de l'automate minimal reconnaissant un langage L au problème du calcul de la congruence de Nerode d'un automate reconnaissant ce langage.

1 L'algorithme itératif

Le premier algorithme permettant de calculer la congruence de Nerode d'un automate est l'algorithme itératif. Il repose sur la définition et la proposition qui suivent. L'idée est de calculer, pour tout $n \in \mathsf{N}$, la congruence de Nerode pour les mots de longueur au plus n, qui pour n assez grand, correspond à la congruence de Nerode.

Définition 2. *Soit $\mathcal{A} = (Q, \Sigma, i, F, \delta)$ déterministe. On définit les relations d'équivalences $(\sim_i)_{i \in \mathsf{N}}$ sur Q par :*

$$q \sim_0 q' \iff (q \in F \leftrightarrow q' \in F)$$
$$q \sim_{i+1} q' \iff q \sim_i q' \ \text{ et } \forall a \in \Sigma \ \ q \cdot a \sim_i q' \cdot a$$

Proposition 2. *Soit $\mathcal{A} = (Q, \Sigma, i, F, \delta)$ déterministe. Soit $(\sim_i)_{i\in\mathsf{N}}$ la suite des relations d'équivalence. Alors :*

1. *Si il existe $k \in \mathsf{N}$ tel que $\sim_k = \sim_{k+1}$, alors $\forall n \geq k$, $\sim_k = \sim_n$.*
2. *$\forall i \geq |Q|$, $\sim_i \; = \; \sim$*

La proposition précédente nous dit qu'il suffit, dans le pire des cas, de calculer les $|Q|$ premières relations d'équivalences de la suite pour obtenir la congruence de Nerode. Mais comment les calculer ?

L'idée de l'algorithme pour calculer la (i+1)-ème relation d'équivalence en connaissant la i-ème est la suivante :

1. Pour chaque lettre de l'alphabet $a \in \Sigma$ et chaque état $q \in Q$, noter le numéro de la classe d'équivalence, pour la i-ème relation, de $q \cdot a$ noté $\overline{q \cdot a}_i$.
2. Associer à chacun des états q le uplet formé de $\overline{q}_i$ et de $\overline{q \cdot a}_i$ pour tout $a \in \Sigma$.
3. Former la (i+1)-ème relation d'équivalence en regroupant les états qui ont le même uplet.

Algorithme 1 (Itératif)**.**
Description de l'algorithme à travers un exemple où l'automate est défini avec $Q = \{q_1, ..., q_8\}$ et $\Sigma = \{a, b\}$.

Etats	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8
$\overline{q}_i$	*1*	*1*	*1*	*3*	*2*	*3*	*4*	*2*
$\overline{q \cdot a}_i$	*2*	*2*	*3*	*2*	*2*	*2*	*4*	*2*
$\overline{q \cdot b}_i$	*3*	*3*	*3*	*1*	*3*	*1*	*4*	*2*
$\overline{q}_{i+1}$	*1,2,3*	*1,2,3*	*1,3,3*	*3,2,1*	*2,2,3*	*3,2,1*	*4,4,4*	*2,2,2*

D'après la description précédente de l'algorithme, il suffit de calculer, dans le pire des cas, $|Q|$ relations d'équivalences et le calcul de chacune consiste à remplir un tableau de taille $|Q| \times \Sigma$. On en déduit alors la complexité qui suit.

Complexité 1. *La complexité est en $O(|\Sigma||Q|^2)$.*

2 L'algorithme d'Hopcroft

Le second algorithme permettant de calculer la congruence de Nerode est l'algorithme d'Hopcroft. Comme nous allons le montrer, la complexité est en $O(|\Sigma||Q| \log_2(|Q|))$.

2.1 Définitions & lemmes

Pour comprendre cet algorithme qui est plus compliqué, nous avons besoin d'introduire la définition de stabilité et de coupure, qui correspond à étudier le comportement des états obtenus à partir d'une partie $B \subset Q$ en lisant une lettre $a \in \Sigma$ par rapport à une autre partie $C \subset Q$.

Définition 3 (Stabilité et coupure)**.** *Soit $\mathcal{A} = (Q, \Sigma, i, F, \delta)$ déterministe. Soit $a \in \Sigma$ et $B, C \subset Q$. On pose $B_1 = \{q \in B \mid q \cdot a \in C\}$ et $B_2 = \{q \in B \mid q \cdot a \notin C\}$. On définit :*

- *Si $B_1 = \emptyset$ ou $B_2 = \emptyset$, alors B est **stable** pour (C, a).*
- *Sinon, B est **coupé** par (C, a) en B_1 et B_2.*

A partir de ces définitions, le lemme suivant peut être démontrer.

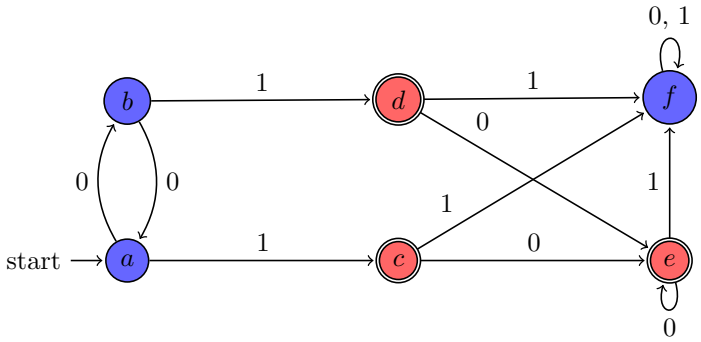
Lemme 1. *Soit $B, C \subset Q$, $C = C_1 \uplus C_2$ et $a \in \Sigma$.*

1. *B stable pour (C_1, a) et $(C_2, a) \Rightarrow$ B stable pour (C, a)*
2. *B stable pour $(C, a) \Rightarrow$ (B stable pour $(C_1, a) \Leftrightarrow$ B stable pour (C_2, a))*

Grâce à ce lemme, nous comprenons que la connaissance de la stabilité d'une partie pour certains couples (C, a) induit une stabilité pour d'autres couples. C'est sur ce principe qu'est basé l'algorithme d'Hopcroft.

2.2 Exemple

Essayons de comprendre comment utiliser ce lemme, et par conséquent, comment fonctionne l'algorithme à travers un exemple. Pour ce faire, prenons l'automate suivant :



où les états rouges sont les états finaux et les états bleus les états non-finaux.

Comme nous voulons calculer la congruence de Nerode et que nous savons que les états finaux ● ne sont pas en relation avec les états non-finaux ● , nous pouvons commencer par prendre {●,●} comme congruence que nous allons noter P et qui correspond à $\sim_0$ définie plus tôt.

Nous voulons alors savoir si cette congruence est celle de Nerode ce qui revient, en utilisant la définition que nous avons introduite, à tester si tout $B \in P$ est stable par tout (C, a) où $C \in P$ et $a \in \Sigma$.

Ainsi, l'algorithme commence en initialisant deux variables :

- $P = \{$ ● , ● $\}$ qui correspond à la congruence.
- $S = \{($ ● $, 0), ($ ● $, 1), ($ ● $, 0), ($ ● $, 1)\}$ qui correspond à la liste des couples dont on doit tester la stabilité sur tous les éléments de la partition.

Toutefois, de manière naïve, tous les couples ont été mis dans S ce qui n'est pas nécessaire d'après le lemme démontré ci-dessus. En effet, toutes les parties

$B \in P$ étant stables par (Q, a) pour tout $a \in \Sigma$, tester la stabilité par $(●, 0)$ équivaut à tester la stabilité par $(●, 0)$. Ainsi, nous avons le choix entre tester la stabilité par la partie ● ou par la partie ● et nous choisissons de tester la stabilité par la partie qui a le moins d'éléments. Dans notre cas, on peut prendre indifféremment ● ou ● .

De manière générale, à chaque fois que nous avons le choix entre deux parties pour tester la stabilité, nous choisissons la partie ayant le moins d'éléments. C'est ce choix qui permet d'obtenir la complexité voulue.

Pour résumer, l'algorithme commence en initialisant ces deux variables :

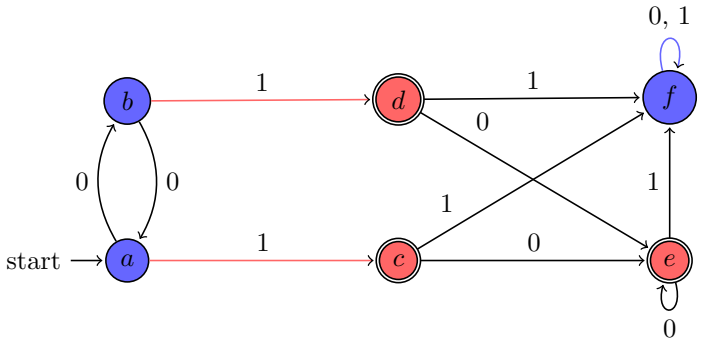
— $P = \{ ● , ● \}$

— $S = \{(● , 0), (● , 1), \cancel{(● , 0)}, \cancel{(● , 1)}\}$

Ensuite, nous prenons un élément quelconque de S et nous le supprimons. Par exemple, prenons $(●, 1)$ ce qui donne $S = \{(● , 0)\}$ et regardons si chacunes des parties $B \in P$ sont stables :

— La partie $B =$ ● est coupée en {f} et {a, b} (voir automate ci-dessous).

— La partie $B =$ ● est stable.



La coupure de ● nous dit que les états de {f} et {a, b} ne peuvent être dans la même classe d'équivalence dans la congruence de Nerode. Par conséquent, la partition doit être raffinée en $P = \{ ● , ● , ● \}$ avec ● = ● ⊎ ●.

Il nous faut alors vérifier que les nouvelles parties ● et ● de P laissent stables les autres parties. Naïvement, on pourrait ajouter $(●, 0)$, $(●, 0)$, $(●, 1)$ et $(●, 1)$ à S. Cependant, le lemme nous permet de ne pas avoir besoin de tester tous ces couples :

— Comme $(●, 1)$ laisse maintenant stable les parties de la nouvelle partition P (raison pour laquelle $(●, 1)$ a été enlevé de S), tester si les parties sont stables par $(●, 1)$ équivaut à tester si elles sont stables par $(●, 1)$ d'après le lemme. Nous pouvons donc en ajouter qu'une des deux et choisissons d'ajouter la partie avec le moins d'éléments i.e. $(●, 1)$.

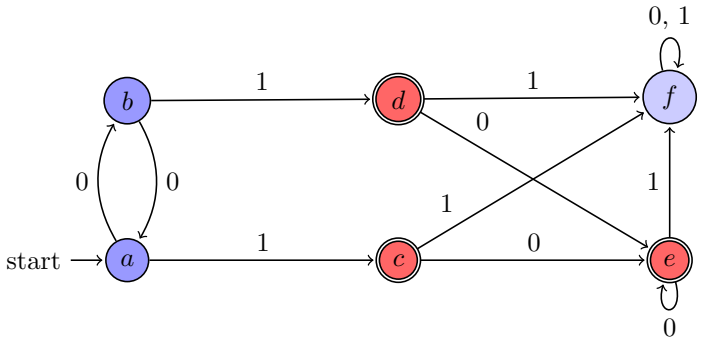
— Si on ajoute $(●, 0)$ et $(●, 0)$ à S, d'après le lemme, on peut enlever $(●, 0)$ de S.

Ainsi, les deux variables valent :

— $P = \{ ● , ● , ● \}$ avec ● = ● ⊎ ●

— $S = \{\cancel{(● , 0)}, (● , 0), (● , 0), (● , 1), \cancel{(● , 1)}\}$

ce qui donne si on colorise l'automate :



L'algorithme continue tant que $S \neq \emptyset$. On prend un élément de S et on le supprime et on répéte sur cet élément ce que l'on vient de faire. Dans notre exemple, tous les éléments restant dans S laissent stable la partition actuelle. Ainsi, celle-ci n'est plus modifiée. Finalement, la congruence de Nerode pour cet automate est { ● , ● , ● } i.e. $\{\{a, b\}, \{c, d, e\}, \{f\}\}$.

2.3 Algorithme

D'après la description détaillée de l'algorithme sur l'exemple précédent, celui-ci s'écrit naturellement comme suit :

Algorithme 2 (Hopcroft)**.**

Data: $\mathcal{A} = (Q, \Sigma, i, F, \delta)$ *déterministe*
$P = \{F,\ Q \backslash F\}$
$S = \{(\min(F, Q \backslash F), a) \,|\, a \in \Sigma\}$
while $S \neq \emptyset$ **do**
  Prendre et supprimer un élément (C, a) *de* S
  for $B \in P$ *coupé en* B_1 *et* B_2 *par* (C, a) **do**
    Remplacer B *par* B_1 *et* B_2 *dans* P
    for $b \in \Sigma$ **do**
      if $(B, b) \in S$ **then**
        Remplacer (B, b) *par* (B_1, b) *et* (B_2, b) *dans* S
      else
        Ajouter $(\min(B_1, B_2), b)$ *dans* S
      end
    end
  end
end

Nous allons essayer maintenant de prouver la terminaison, la correction et la complexité de cet algorithme.

2.4 Terminaison

Pour montrer la terminaison de l'algorithme, on peut remarquer qu'à chaque passage dans la boucle *while* :

— Soit la partition est raffinée
— Soit S perd un élément

Comme la partition ne peut être raffinée qu'un nombre fini de fois, S va finir par être vide. Ainsi, le programme termine.

Dans la suite de cette partie, nous allons utiliser les notations suivantes :

Définition 4. *Soit $\mathcal{A} = (Q, \Sigma, i, F, \delta)$ déterministe. Soit $a \in \Sigma$. Soit P une partition de Q. Pour $q \in Q$:*
— *$\overline{q}$ est l'unique $B \in P$ tel que $q \in B$*
— *$L_q = \{w \in \Sigma^* \mid q \cdot w \in F\}$*
— *$a^{-1}q = \{q' \in Q \mid q' \cdot a = q\}$*

2.5 Correction

Pour montrer la correction de l'algorithme, on peut remarquer qu'à chaque passage dans la boucle *while*, l'invariant suivant est respecté :

1. Tout (B, a) de $(P \times \Sigma) \backslash S$ laisse stable tous les éléments de P
2. $\forall (q, q') \in Q^2, L_q = L_{q'} \Rightarrow \overline{q} = \overline{q'}$

En sortant de la boucle *while*, on a alors :
— Tout (B, a) de $(P \times \Sigma)$ laisse stable tous les éléments de P
— $\forall (q, q') \in Q^2, L_q = L_{q'} \Rightarrow \overline{q} = \overline{q'}$

ce qui donne :

$$\forall (q, q') \in Q^2, \overline{q} = \overline{q'} \Leftrightarrow L_q = L_{q'}$$

Cette dernière équivalence nous dit que notre partition correspond à la congruence de Nerode. L'algorithme est bien correct.

2.6 Complexité

Pour montrer la complexité, il nous faut expliciter la complexité des opérations de base de notre algorithme. Dans notre cas, en ayant fait au préalable un parcours en profondeur de notre automate et stocké dans un tableau les ensembles $a^{-1}q$ pour chaque $q \in Q$ et $a \in \Sigma$, nous pouvons, pour un séparateur (C, a) :

1. Parcourir les éléments de $a^{-1}q$ en $O(|a^{-1}q|)$.
2. Parcourir les éléments de $a^{-1}C$ en $O(|a^{-1}C|)$.
3. Obtenir $\overline{q}$ et $|\overline{q}|$ en temps constant pour $q \in Q$.

Dans ces conditions, en comptant le nombre d'éléments de $a^{-1}C$ par élément de la partition, on peut obtenir la liste des éléments de P coupés par (C, a) en $O(|a^{-1}C|)$. Donc, la complexité des instructions dans la boucle *while* est en $O(|a^{-1}C|)$.

Par conséquent, la complexité de l'algorithme est :

$$\sum_{(C,a) \text{ traitée}} O(|a^{-1}C|)$$

Or, pour $a \in \Sigma$ et $q \in Q$, si on note $C_1, ..., C_k$ toutes les parties contenant q traitées par l'algorithme, classées par ordre de traitement, alors, par construction, $\forall i \in [\![1, k-1]\!], |C_{i+1}| \leq \frac{|C_i|}{2}$ donc $k \leq \log_2(|Q|)$.

Ainsi, un partitionnement de l'ensemble des séparateurs traités en fonction des états et des lettres donne :

$$\begin{aligned}\sum_{(C,a) \text{ traitée}} O(|a^{-1}C|) &= \sum_{\substack{q \in Q \\ a \in \Sigma}} O(|\{(C,a) \,|\, (C,a) \text{ traitée et } q \cdot a \in C\}|) \\ &= \sum_{\substack{q \in Q \\ a \in \Sigma}} O(\log_2(|Q|)) = O(|\Sigma||Q| \log_2(|Q|))\end{aligned}$$

Pour cet algorithme, nous obtenons alors le résultat suivant :

Complexité 2. *La complexité est en* $O(|\Sigma||Q| \log_2(|Q|))$.

3 Lien avec l'algorithme de Brzozowski

Le troisième algorithme permettant de calculer la congruence de Nerode est l'algorithme de Brzozowski, qui n'a, a priori, rien à voir avec l'algorithme d'Hopcroft. Cependant, une implémentation judicieuse de cet algorithme montre que les deux algorithmes se ressemblent fortement.

3.1 Algorithme naïf

L'algorithme repose sur le lemme suivant :

Lemme 2. *Soit* $\mathcal{A} = (Q, \Sigma, i, F, \delta)$ *déterministe reconnaissant* L. *Alors,* $D(R(A))$ *est l'automate minimal reconnaissant* L^r, *où* D *donne le déterminisé,* R *le miroir et* L^r *le langage miroir de* L.

Grâce à ce lemme, la minimisation d'un automate correspond à un enchaînement de retournement et déterminisation d'automate.

Algorithme 3 (Brzozowski naïf)**.**
Data: $\mathcal{A} = (Q, \Sigma, i, F, \delta)$ *déterministe*
$\mathcal{A}' = D(R(D(R(A))))$

De cette manière, on peut montrer la complexité suivante :

Complexité 3. *La complexité est en* $O(|\Sigma| \exp(|Q|))$.

3.2 Algorithme amélioré

Avec l'implémentation précédente, la complexité est exponentielle. Toutefois, cette implémentation est nettement améliorable ! Un algorithme polynomial est présenté dans l'article DFA minimization : from Brzozowski to Hopcroft écrit

par Pedro Garcia, Damian Lopez et Manuel Vazquez de Parga. Essayons de le comprendre intuitivement.

Prenons $\mathcal{A} = (Q, \Sigma, i, F, \delta)$ déterministe. L'idée est la suivante :

1. Considérer que l'automate minimal a 2 états : $P = \{F, Q\backslash F\}$. Initialiser une liste d'état à tester : $S = \{(F, a) \,|\, a \in \Sigma\}$.
2. Pour $(F, a) \in S$, regarder si les états de l'automate $D(R(A))$ obtenus en lisant une seule lettre a depuis l'état initial I se trouvent dans les états déjà présents dans l'automate minimal (i.e. regarder si (F, a) coupe F et $Q\backslash F$). Ainsi, 2 cas :
   - Soit les états s'y trouvent, alors l'automate est bien minimal.
   - Soit ils ne s'y trouvent pas tous, alors l'automate n'est pas encore minimal. Du coup, on raffine les états et on ajoute dans S les états qui ne se trouvaient pas dans l'automate.
3. Recommencer 2 en prenant un autre élément de S.

Cette description nous donne alors l'algorithme suivant :

Algorithme 4 (Brzozowski amélioré)**.**

Data: $\mathcal{A} = (Q, \Sigma, i, F, \delta)$ *déterministe*
$P = \{F,\ Q\backslash F\}$
$S = \{(\min(F, Q\backslash F), a) \,|\, a \in \Sigma\}$
while $S \neq \emptyset$ **do**
  Prendre et supprimer un élément (C, a) *de* S
  $P' = P$
  for $B \in P$ *coupé en* B_1 *et* B_2 *par* (C, a) **do**
    Remplacer B *par* B_1 *et* B_2 *dans* P
  end
  if $P \neq P'$ **then**
    for $b \in \Sigma$ **do**
      Ajouter $(a^{-1}C, b)$ *dans* S
    end
  end
end

Nous pouvons alors facilement voir que les implémentations de l'algorithme d'Hopcroft et de Brzozowski se ressemblent fortement ! Toutefois, la complexité de ce dernier algorithme est moins bonne :

Complexité 4. *La complexité est en* $O(|\Sigma||Q|^2)$.

Conclusion

A l'heure actuelle, différents algorithmes existent pour calculer l'automate minimal reconnaissant un langage L donné. Nous en avons vu trois : l'algorithme itératif, l'algorithme d'Hopcroft et deux implémentations de l'algortihme de Brzozowsky. Des trois, l'algortihme d'Hopcroft est celui avec la meilleure

complexité, en $O(|\Sigma||Q|\log_2(|Q|))$. Il est à l'heure actuelle l'algorithme le plus perfomant.