Complexité en état d'opération sur des langages rationnels

Edouard HADDAG

2 juin 2025

Résumé

Dans cet article, nous étudions deux transformations particulières de langages rationnels : le « trognon » et le « langage permuté ». Nous définissons formellement ces deux constructions et proposons pour chacune un algorithme permettant de calculer un automate non déterministe reconnaissant le langage transformé. Nous démontrons la correction de ces algorithmes et analysons leur complexité en nombre d'états dans le pire des cas.

Table des matières

1 Introduction

Cet article s'intéresse à la complexité en nombre d'états des automates associés à des langages rationnels soumis à des transformations spécifiques. Plus précisément, nous étudions deux opérations définies sur les langages rationnels : le « trognon » et le « langage permuté ».

La première transformation, appelée « trognon », consiste à retirer, dans chaque mot du langage de départ, un préfixe dont le miroir est également un suffixe. Le mot central ainsi isolé constitue le « trognon » du mot. Nous proposons un algorithme, que nous appelons algorithme de « grignotage », permettant, à partir d'un automate non déterministe reconnaissant un langage rationnel donné, de construire un automate reconnaissant le trognon de ce langage. Nous détaillons le principe de l'algorithme, formalisons sa construction, en prouvons la correction et analysons sa complexité dans le pire des cas.

La seconde transformation, appelée « langage permuté », consiste à permuter toutes les paires de lettres consécutives situées à des positions paires et impaires dans les mots du langage (positions $2k$ et $2k + 1$). Pour cette transformation également, nous construisons un algorithme, que nous nommons algorithme de « twistage », qui transforme un automate reconnaissant le langage de départ en un automate reconnaissant le langage permuté. Nous décrivons le fonctionnement de cet algorithme, en démontrons la validité et étudions sa complexité.

L'objectif principal de ce travail est d'évaluer l'impact de ces opérations sur la taille des automates résultants, en particulier dans le cadre non déterministe. Pour chaque transformation, nous établissons des bornes supérieures sur le nombre d'états des automates produits, ce qui permet d'estimer la complexité opérationnelle de ces constructions. En conclusion, nous proposons des pistes pour des travaux futurs visant à affiner ces bornes ou à déterminer leur optimalité.

2 Préliminaires

Dans cette section préliminaire, nous allons rappeler les notions fondamentales nécessaires à la compréhension du reste du document. Nous y définirons notamment les concepts d'alphabet, de mot, de langage, d'automate et de complexité en nombre d'états. Ces définitions seront accompagnées d'exemples et de remarques pour en faciliter l'assimilation

2.1 Les mots

★ **Définition 2.1** (Alphabet)
Un *alphabet* Σ est un ensemble fini non-vide de *symboles*.

★ **Définition 2.2** (Mot)
Un *mot* est une suite finie de symboles sur un alphabet Σ. Le mot composé de zéro symbole est appelé mot vide et est noté ε. De plus, l'ensemble des mots sera noté Σ^*

Exemple 2.1 :

$$\Sigma = \{a, b, c, d\},$$
$$w = abbcdda.$$

★ **Définition 2.3** (Longueur d'un mot)
On parlera de la *longueur d'un mot* w noté $|w|$ pour désigner le nombre de symboles qui le composent.

Exemple 2.2 :

$$w = abbcdda,$$
$$|w| = 7.$$

★ **Définition 2.4** (Concaténation de deux mots)
On notera la *concaténation* de deux mots $u = a_0 \cdots a_n$ et $v = b_0 \cdots b_m$ par $u \cdot v$. Qui est ainsi égal à $u \cdot v = a_0 \cdots a_n b_0 \cdots b_m$. On notera que :

- La concaténation est associative $(w \cdot u) \cdot v = w \cdot (u \cdot v)$.
- La concaténation admet un élément neutre $u \cdot \varepsilon = \varepsilon \cdot u = u$.

Exemple 2.3 :

$$u = aba,$$
$$v = cdcd,$$
$$u \cdot v = abacdcd.$$

★ **Définition 2.5** (Facteur, préfixe, suffixe d'un mot)
Un *facteur* u d'un mot w est une suite extraite de la suite de lettre qui composent le mot w :

$$u \text{ est un facteur de } w \iff \exists (v, x) \in (\Sigma^*)^2 \mid w = v \cdot u \cdot x.$$

- Par ailleurs, on parlera de *préfixe* quand $v = \varepsilon$.
- Enfin, on parlera de *suffixe* quand $x = \varepsilon$.

Par ailleurs, on remarquera que ε est : *préfixe*, *suffixe* et *facteur* de tout mot.

Exemple 2.4 :

Dans cet exemple, on a que u est un facteur de w, que v est un préfixe de w et enfin que y est un suffixe de w :

$$w = abbcdda,$$
$$u = bcd,$$
$$v = abb,$$
$$y = dda.$$

★ **Définition 2.6** (Le miroir d'un mot)
Le *miroir* d'un mot w noté $\overleftarrow{w}$. Qui est ainsi défini récursivement comme ceci :

$$\overleftarrow{\varepsilon} = \varepsilon,$$
$$\overleftarrow{u \cdot a} = a \cdot \overleftarrow{u},$$
$$\text{Avec } a \in \Sigma \text{ et } u \in \Sigma^*.$$

Exemple 2.5 :

$$w = abbcdda,$$
$$\overleftarrow{w} = addcbba.$$

2.2 Les langages

★ **Définition 2.7** (Langage)
Un *langage* L est un ensemble de mots sur un alphabet fini Σ. On appellera *langage vide* le langage ne comportant aucun mot et sera noté $\varnothing$.

Exemple 2.6 :

$$\Sigma = \{a, b, c, d\},$$
$$L_1 = \{a, aa, bc, da, \varepsilon\},$$
$$L_2 = \varnothing.$$

★ **Définition 2.8** (L'union de langages)
L'*union* de deux langages sera notée $L_1 \cup L_2$ et est définie comme ceci :

$$L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \vee w \in L_2\}.$$

Exemple 2.7 :

$$L_1 = \{\varepsilon, a, aa, bc, da\},$$
$$L_2 = \{d, aa, cd\},$$
$$L_1 \cup L_2 = \{\varepsilon, a, d, aa, cd, bc, da\}.$$

★ **Définition 2.9** (Concaténation de langages)
La *concaténation* de deux langages sera notée $L_1 \cdot L_2$ et est définie grâce à la concaténation des mots qui composent les langages :

$$L_1 \cdot L_2 = \{u \cdot v \mid u \in L_1, v \in L_2\}.$$

Exemple 2.8 :

$$L_1 = \{\varepsilon, a, aa\},$$
$$L_2 = \{d, cc\},$$
$$L_1 \cdot L_2 = \{d, ad, aad, cc, acc, aacc\}.$$

★ **Définition 2.10** (Copie n-ième d'un langage)
La *copie n-ième* d'un langage L notée L^n et est définie récursivement comme ceci :

$$L^0 = \{\varepsilon\},$$
$$L^n = L^{n-1} \cdot L.$$

Exemple 2.9 :

$$L = \{\varepsilon, a\},$$
$$L^3 = \{\varepsilon, a, aa, aaa\}.$$

★ **Définition 2.11** (L'étoile (de KLEENE) d'un langage)
L'*étoile* (de KLEENE) d'un langage noté L^*, sera définie comme ceci :

$$L^* = \bigcup_{i \geq 0} L^i.$$

Exemple 2.10 :

$$L = \{a\},$$
$$L^* = \{\varepsilon\} \cup \{a\} \cup \{aa\} \cup \cdots$$

! Remarque 2.1
Grâce à cette opération, on comprend dorénavant la notion Σ^* comme l'ensemble des mots de l'alphabet Σ, il pourra donc aussi être vu comme le langage contenant tous les mots.

★ **Définition 2.12** (Langages rationnel)
On note $Rat(\Sigma^*)$ le plus petit ensemble de langages sur Σ^* vérifiant les propriétés suivantes :

- $\varnothing \in Rat(\Sigma^*)$.
- $\{a\} \in Rat(\Sigma^*)$ avec $a \in \Sigma$.
- $Rat(\Sigma^*)$ est fermé pour les opérations d'union, de concaténation et pour l'étoile.

! Remarque 2.2
Pour simplifier l'écriture, nous écrirons + pour représenter l'union, et nous utiliserons également ab au lieu de $\{ab\}$.

Exemple 2.11 :

$$L = \varnothing + (a^* \cdot (b + \varepsilon)).$$

2.3 Les automates

Nous allons parler des automates sans ε-transition (des automates utilisent des ε-transitions, comme ceux de THOMPSON [3], utilisés par nos ordinateurs). Pour autant, les automates que nous verrons ne sont pas limités par le manque de ces transitions.

★ **Définition 2.13** (Automate)
Un *automate* est un objet mathématique *reconnaissant* un langage. On notera $M \in AFN(\Sigma, \eta)$ l'automate qui a pour *transition* des valeurs dans Σ, des valeurs d'*état* dans η.

On aura aussi, $AFN(\Sigma, \eta)$, l'ensemble des automates finis *non déterministes* de valeur de transition dans Σ et de valeur d'état dans η. Enfin, on écrira $L(M)$ pour désigner le langage qu'il reconnait.

Un automate est un tuple qu'on peut écrire de cette forme $M = (Q, I, F, \delta)$ avec :

$Q \subseteq \eta$	États de l'automate.
$I \subseteq Q$	États initiaux.
$F \subseteq Q$	États finaux.
$\delta : Q \times \Sigma \to \mathcal{P}(Q)$	Fonction de transition.

! **Remarque 2.3** (Extension de la fonction de transition)
On peut étendre la fonction de transition pour ce qu'elle prenne en entrée non plus un simple symbole, mais un mot. Sa signature devient alors $\delta : Q \times \Sigma^* \to \mathcal{P}(Q)$, et elle est définie comme suit :

$$\begin{aligned} \delta(q, \varepsilon) &= \{q\}, \\ \delta(q, a \cdot w) &= \bigcup_{p \in \delta(q,a)} \delta(p, w), \text{ avec } a \in \Sigma. \end{aligned}$$

★ **Définition 2.14** (Reconnaissance d'un mot par un automate)
Un mot est *reconnu* (ou *accepté*) s'il existe une suite de transition partant d'un état initial vers un état final :

$$w \text{ est accepté par } M \iff \left(\bigcup_{i \in I} \delta(i, w) \right) \cap F \neq \varnothing.$$

★ **Définition 2.15** (Langage d'un automate)
Le *langage d'un automate* est donc l'ensemble des mots qu'il reconnait :

$$L(M) = \{w \in \Sigma^* \mid w \text{ est accepté par } M\}.$$

Exemple 2.12 : Représentation d'un automate

Un automate peut se représenter à l'aide d'un graphe orienté et valué particulier. Par exemple, si on veut représenter $M = (\{q_1, q_2, q_3, q_4, q_5\}, \{q_1, q_2\}, \{q_2, q_3\}, \delta)$ avec $M \in$

$AFN(\Sigma, \eta)$, $\Sigma = \{a, b\}$, $\eta = \{q_1, q_2, q_3, q_4, q_5\}$ et δ défini comme ceci :

$$\begin{aligned}
&\delta(q_1, a) = \{q_2, q_4\}, && \delta(q_3, b) = \{q_4\},\\
&\delta(q_1, b) = \varnothing, && \delta(q_4, a) = \{q_5\},\\
&\delta(q_2, a) = \varnothing, && \delta(q_4, b) = \{q_3\},\\
&\delta(q_2, b) = \varnothing, && \delta(q_5, a) = \{q_4\},\\
&\delta(q_3, a) = \{q_3\}, && \delta(q_5, b) = \{q_5\}.
\end{aligned}$$

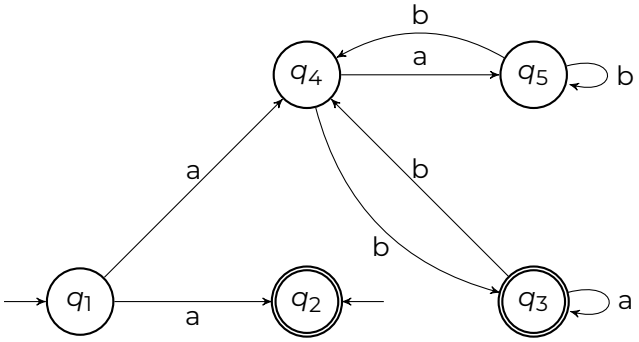

FIGURE 1 – Exemple de représentation graphique d'un automate.

On peut voir que les états initiaux ont une petite flèche qui pointe sur eux et que les états finaux ont un double contour. Et, que les transitions sont symbolisées par des flèches entre les états et que ces flèches sont labellisées.

★ **Définition 2.16** (Automate déterministe)
Un automate est dit *déterministe* quand tous ses états vont au maximum à un état par symbole et que l'automate ne possède qu'un seul état initial.

$$|I| = 1 \land \forall q \in Q, \forall a \in \Sigma, |\delta(q, a)| \leq 1 \iff M \in DFA(\Sigma, \eta)$$

avec $DFA(\Sigma, \eta)$ l'ensemble des automates déterministe.

Exemple 2.13 : Représentation d'un automate déterministe

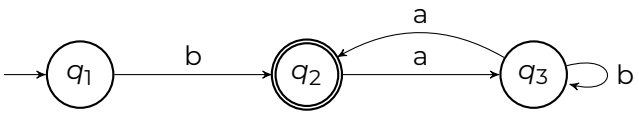

FIGURE 2 – Exemple de représentation graphique d'un automate déterministe.

★ **Définition 2.17** (L'ensemble des langages reconnaissable)
On note $Rec(\Sigma^*)$, l'ensemble des langages de Σ^* dit *reconnaissable* :

$$L \in Rec(\Sigma^*) \iff \exists M \in NFA(\Sigma, \eta) \land L = L(M).$$

➤ **Théorème 2.1** (Théorème de KLEENE)
Le mathématicien STEPHEN COLE KLEENE a démontré que :

$$Rec(\Sigma^*) = Rat(\Sigma^*).$$

On pourra alors passer de langage rationnel à automate et vice-versa.

! Remarque 2.4
La notation $|M|$ représentera le nombre d'états de l'automate M.

★ Définition 2.18 (Automate minimal)
Un automate est dit *minimal* lorsqu'il s'agit de l'automate ayant le plus petit nombre d'états possible qui reconnaît un langage donné :

$$M \text{ est minimal} \Longrightarrow \nexists N \in NFA(\Sigma, \eta) \wedge L(M) = L(N).$$

On parlera d'automate déterministe minimal quand :

$$M \text{ est déterministe minimal} \Longrightarrow \nexists N \in DFA(\Sigma, \eta) \wedge L(M) = L(N).$$

Exemple 2.14 : Représentation d'un automate minimal

Dans cet exemple, l'automate M_2 est bien plus petit que l'automate M_1, c'est même le plus petit automate reconnaissant leur langage (a^*)

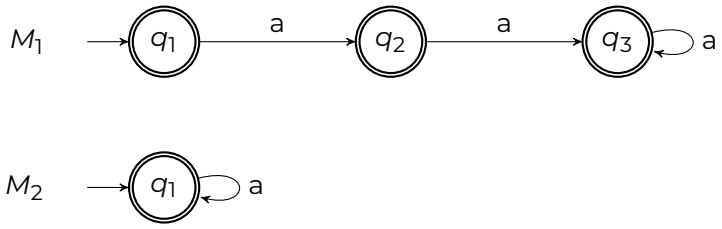

FIGURE 3 – Exemple de représentation graphique d'un automate minimal.

2.4 La complexité en état

★ Définition 2.19 (Complexité en état d'un langage (rationnel))
La *complexité en état* d'un langage est le nombre d'états de l'automate minimal reconnaissant ce langage :

$$\mathcal{C}(L) = |M| \wedge L(M) = L \text{ avec}$$
$$M \text{ est un automate minimal.}$$

Cette complexité se scinde en deux parties :

- La complexité non déterministe ($\mathcal{C}_{ndet}$) quand :

$$M \in NFA(\Sigma, \eta).$$

- La complexité déterministe ($\mathcal{C}_{det}$) lorsque :

$$M \in DFA(\Sigma, \eta).$$

Exemple 2.15 :

Si on reprend le langage de la figure 3, M_2 est l'automate minimal reconnaissant a^* et il n'a qu'un état, de ce fait ce langage à une complexité en état déterministe de 1. Et, donc aussi, une complexité non déterministe de 1, puisque tout automate déterministe est dans $NFA(\Sigma, \eta)$ et qu'on ne peut pas faire un automate à moins d'un état.

★ **Définition 2.20** (Complexité en état opérationnel)
La *complexité en état opérationnel* peut être vu comme la borne supérieure de la complexité en état du langage produit par l'opération. Autrement dit, f a une complexité de $g(n_1, \ldots, n_k)$ quand :

$$\forall (L_1, \ldots, L_k) \in \mathcal{P}(\Sigma^*) \quad \mathcal{C}(f(L_1, \ldots, L_k)) \leq g(\mathcal{C}(L_1), \ldots, \mathcal{C}(L_k)).$$

De même que la complexité en état, la complexité en état opérationnel se scinde en deux parties :

- La complexité non déterministe quand :

$$\mathcal{C}_{ndet}(f(L_1, \ldots, L_k)) \leq g(\mathcal{C}_{ndet}(L_1), \ldots, \mathcal{C}_{ndet}(L_k)).$$

- La complexité déterministe lorsque :

$$\mathcal{C}_{det}(f(L_1, \ldots, L_k)) \leq g(\mathcal{C}_{det}(L_1), \ldots, \mathcal{C}_{det}(L_k)).$$

! **Remarque 2.5**
La complexité en état déterministe sera toujours supérieure ou égale à la complexité en état non déterministe, dû au fait que les automates déterministes sont aussi des automates non déterministe, mais avec des contraintes en plus.

Exemple 2.16 :

Voici quelques complexités en état opérationnel connues et importantes :

Fonction	Complexité	Source
L'union non déterministe	$m + n$	Holzer et Kutrib [1]
L'union déterministe	mn	Maslov [2]
L'intersection non déterministe	mn	Holzer et Kutrib [1]
L'intersection déterministe	mn	Maslov [2]
La concaténation non déterministe	$m + n$	Holzer et Kutrib [1]
La concaténation déterministe	$m2^n - 2^{n-1}$	Maslov [2]
L'étoile (de Kleene) non déterministe	$n + 1$	Holzer et Kutrib [1]
L'étoile (de Kleene) déterministe	$\frac{3}{4}2^n$	Maslov [2]

Table 1 – Tableau de complexité en état opérationnel de plusieurs opérations.

! **Remarque 2.6** (Complexité de la déterminisation)
L'opération de *déterminisation*, c'est-à-dire l'opération de rendre un automate non déterministe quelconque en un automate déterministe à une complexité en état de 2^n.

Même si cette opération ne rentre pas réellement dans notre définition de la complexité en état, car prenant en entrée un automate non déterministe et renvoyant un automate déterministe.

3 Le trognon d'un langage

Dans cette partie, nous nous concentrerons sur un langage bien particulier, le « trognon ». Dans un premier temps, nous définirons formellement ce langage. Enfin, nous présenterons un algorithme de « grignotage » permettant de le calculer à partir d'un automate non déterministe, l'automate du trognon du langage de l'automate initial. Nous présenterons ensuite une preuve de cet algorithme ainsi que la complexité dans le pire des cas de celui-ci. Enfin, pour finir, nous conclurons cette partie en faisant une courte synthèse de ce qu'on aura vu et de la possible continuation.

3.1 Définition

★ **Définition 3.1** (Le langage trognon)
Le langage $Core(L)$ (aussi noté $Trognon(L)$) est construit à partir de L en retirant, pour chaque mot de L, un préfixe dont le miroir est également un suffixe ; ce suffixe est, lui aussi, retiré :

$$Core(w) = \{v \in \Sigma^* \mid \exists u \in \Sigma^* \wedge w = u \cdot v \cdot \overleftarrow{u}\},$$
$$Core(L) = \bigcup_{w \in L} Core(w).$$

Exemple 3.1 :

$$L = \{a, b, aa, aabcaa\},$$
$$Core(L) = \{\varepsilon, a, b, bc, abca, aabcaa\}.$$

Maintenant que nous savons ce qu'est le trognon d'un langage, nous allons présenter un algorithme permettant de construire un automate le reconnaissant, à partir d'un automate reconnaissant le langage initial.

3.2 Algorithme de grignotage

Notre algorithme prendra en entrée un automate non déterministe et renverra un automate non déterministe. Mais, avant de rentrer dans le dur de l'algorithme, nous regardons le principe de l'algorithme, puis nous verrons sa définition formelle.

3.2.1 Principe de l'algorithme

Le principe de cet algorithme est assez simple, si on prend l'automate suivant :

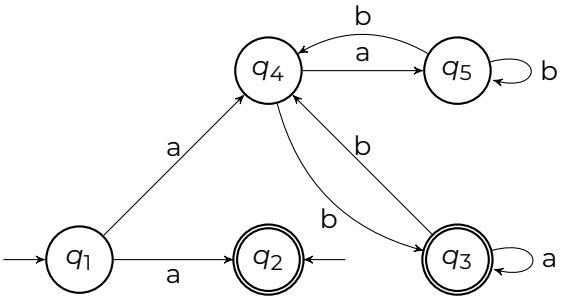

FIGURE 4 – Exemple d'un automate avant grignotage.

Nous devons alors le « grignoté » de tous les mots pour en faire l'automate reconnaissant le trognon du langage. Avant d'essayer de le grignoter de tous les mots, on va tenter de le grignoter d'un seul mot, *ab*. On va donc simplement « avancer » les états initiaux de la transition *ab* et « reculer » les états finaux de la transition *ab* :

$$I' = \bigcup_{i \in I} \delta(i, ab) = \{q_3\},$$
$$F' = \bigcup_{f \in F} \delta^{-1}(f, ab) = \{q_4\}.$$

avec $\forall p \in \delta^{-1}(q, w) \implies q \in \delta(p, w)$.

Et, le reste de l'automate ne changerait pas, ce qui nous donnerait à la fin :

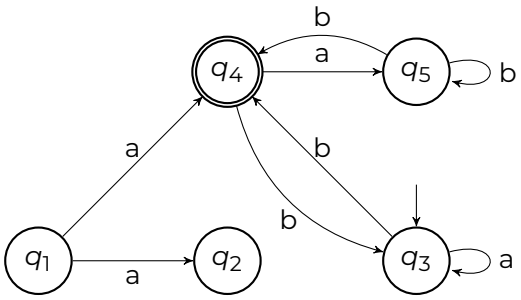

FIGURE 5 – Exemple d'un automate après grignotage de *ab*.

★ **Définition 3.2** (Grignotage d'un automate par un mot)
Soit $M = (Q, I, F, \delta)$, un automate, le *grignotage* de M par un mot w sera l'automate :

$$Nibbling(M, w) = (Q, I', F', \delta),$$
$$I' = \bigcup_{i \in I} \delta(i, w) \wedge F' = \bigcup_{f \in F} \delta^{-1}(f, w).$$

♦ **Lemme 3.1** (Premier lemme du grignotage)
Si un mot w est reconnu par $Nibbling(M, v)$, alors $v \cdot w \cdot \overleftarrow{w}$ est reconnu par M.

$$\forall M \in NFA(\Sigma, \eta) \wedge \forall v \in \Sigma^* \wedge N = Nibbling(M, v) \wedge w \in L(N) \iff v \cdot w \cdot \overleftarrow{v} \in L(M).$$

♦ **Lemme 3.2** (Second lemme du grignotage)
Soit un automate $M \in NFA(\Sigma, \eta)$, et un mot $w = u \cdot v \cdot \overleftarrow{u} \in L(M)$, si :

$$\exists (x, y, z) \in (\Sigma^*)^3 : Nibbling(M, y) = Nibbling(M, x) \wedge u = x \cdot z \implies (y \cdot z) \cdot v \cdot \overleftarrow{(y \cdot z)} \in L(M).$$

Il nous reste donc à présent à calculer le grignoté de tous les mots dits « pertinents », puis à effectuer l'union non déterministe des automates ainsi obtenus.

Cependant, une question importante demeure : quels sont les mots que l'on qualifie de pertinents ?

Une première intuition serait de considérer tous les fragments internes des mots du langage, autrement dit toutes les sous-parties situées entre le début et la fin, ce que l'on pourrait appeler, de manière imagée, *la chair* du mot (par opposition au trognon).

Mais, cette approche soulève un problème : le langage étant, en général, infini, il existe potentiellement une infinité de « chairs ». Une telle démarche, si elle n'est pas restreinte, est alors incalculable en pratique.

★ **Définition 3.3** (La chair d'un langage)
Le langage *Flesh*(*L*) (aussi noté *Chair*(*L*)) est construit à partir de *L* en gardant, pour chaque mot de *L*, un préfixe dont le miroir est également un suffixe :

$$Flesh(w) = \{u \in \Sigma^* \mid \exists v \in \Sigma^* \wedge w = u \cdot v \cdot \overleftarrow{u}\},$$
$$Flesh(L) = \bigcup_{w \in L} Flesh(w).$$

Exemple 3.2 :

$$L = \{a, b, aa, aabcaa\},$$
$$Flesh(L) = \{\varepsilon, a, b, aa\}.$$

Puisque l'automate reconnaissant le langage de départ est fini, notons *n* son nombre d'états. Si un mot *u* appartient à la chair d'un mot, et que sa longueur vérifie $|u| \geq n$, alors la lecture de *u* force la réapparition d'au moins un état (par le *principe des tiroirs*). Il en résulte l'existence d'un cycle dans l'automate de départ.

L'algorithme explore donc tous les mots $w \in \Sigma^*$ en construisant successivement les automates grignotés $N = Nibbling(M, w)$. Puisque le nombre d'automates à états fixes est fini, on aboutit forcément à une répétition : il existe un mot plus court *u* tel que

$$N = Nibbling(M, u) \wedge |u| < |w|.$$

À partir de ce moment, il n'est plus nécessaire « d'étendre » *w*, car tout mot préfixé par *w* aurait déjà été obtenu via un préfixe plus court. Cette condition garantit l'arrêt de l'algorithme et garantit que l'on ne construit qu'un nombre fini d'automates.

3.2.2 Formalisation de l'algorithme

Maintenant que nous avons l'idée de l'algorithme, nous verrons sa définition formelle :

```
1  nibbling(M):
2    N ← M
3    la ← {M}
4    lw ← queue_empty()
5    enqueue(lw, ε)
6    while lw ≠ ∅ do
7      for s in Σ do
8        w ← queue_dequeue(lw) · s
9        A ← Nibbling(M, w)
10       if A.I ≠ ∅ ∧ A.F ≠ ∅ ∧ A ∉ la then
11         N ← N ∪ A
12         la ← la ∪ {A}
13         enqueue(lw, w)
14   return N
```

! Remarque 3.1
On observe que l'algorithme effectue un parcours en largeur de l'arborescence des mots. Cela garantit que tous les mots de longueur strictement inférieure ont été explorés avant de commencer à « grignoter » ceux de longueur supérieure.

3.3 Preuve de notre algorithme

Pour prouver que notre algorithme est correct et construit bien l'automate du trognon d'un langage, nous allons d'abord montrer que tous mots reconnus par l'automate sont des mots appartenant au trognon du langage. Enfin, nous allons montrer que tous les mots du trognon d'un langage sont reconnu par l'automate.

3.3.1 Tous les mots reconnus sont des mots du trognon

■ Preuve 3.1
Soit $M \in NFA(\Sigma, \eta)$, $N = \texttt{nibbling}(M)$ et un mot w reconnu par l'automate N. S'il est reconnu par N, c'est qu'il est reconnu par au moins un automate $Nibbling(M, v)$ qui compose N. Or, par le lemme 3.1, alors $v \cdot w \cdot \overleftarrow{v}$ est reconnu par M, et donc w fait partie de $Core(L(M))$.
□

3.3.2 Tous les mots du trognon sont reconnus

■ Preuve 3.2
Pour prouver que tous les mots du trognon de $M \in NFA(\Sigma, \eta)$ sont bien reconnus par l'automate $N = \texttt{nibbling}(M)$, nous allons faire une preuve par récurrence sur la longueur de la chair d'un mot $w = u \cdot v \cdot \overleftarrow{u}$ du langage $L(M)$:

• **Initialisation :** $|u| = 0$
Si $|u| = 0 \iff u = \varepsilon \iff v = w$.

Ensuite, puisque nous faisons l'union de l'automate M dans notre algorithme :

$$w \in L(M) \iff v \in L(M) \iff v \in L(\texttt{nibbling}(M)).$$

• **Hérédité :** $|u| \geq 1$
- Si $Nibbling(M, u)$ fait partie des automates dont on a fait l'union dans notre algorithme, alors par le lemme 3.1 :

$$v \in L(Nibbling(M, u)) \iff v \in L(\texttt{nibbling}(M)).$$

- Sinon cela veut dire que :

$$\exists (y, x, z) \in \Sigma^* : Nibbling(M, y) = Nibbling(M, x) \wedge u = x \cdot z \wedge |y| < |x|.$$

On s'intéresse donc au mot $w' = (y \cdot z) \cdot v \cdot \overleftarrow{(y \cdot z)}$ qui est dans $L(M)$ par le lemme 3.2, or par hypothèse de récurrence ($|y \cdot z| < |u|$) $v \in L(\texttt{nibbling}(M))$.
□

3.4 Pire cas de notre algorithme

Dans cette section, nous établirons une borne supérieure sur le nombre d'états de l'automate que notre algorithme peut engendrer. Cette estimation, distincte de la complexité du trognon d'un langage, nous offre néanmoins un plafond pour la complexité en états.

Dans notre pire cas, notre algorithme fera l'union de tous les automates distincts qu'ont les mêmes états et la même fonction de transition. La seule chose qui peut changer c'est l'ensemble des états initiaux et l'ensemble des états finaux.

On sait que ces deux ensembles sont des ensembles finis non vide à maximum n élément, il y a donc $2^n - 1$ ensembles d'états initiaux et $2^n - 1$ ensembles d'états finaux. Ce qui nous donne $(2^n - 1)^2$ automate différents. Ensuite, puisqu'on fait l'union non déterministe de deux automates, l'automate résultant de notre algorithme aurait dans le pire des cas $n(2^n - 1)^2$ états avec n le nombre d'états de l'automate de départ.

3.5 Conclusion

Pour conclure, nous venons de créer un algorithme qui prendre n'importe quel automate en entrée et qui calcule l'automate du trognon du langage de cette automate. De plus, notre algorithme, créer un automate dans le pire des cas à $n(2^n - 1)^2$ états. Par conséquent, la complexité en états du grignotage d'un langage est d'au moins $n(2^n - 1)^2$.

En exécutant cet algorithme, nous pouvons remarquer que les automates produit ne sont pas minimaux, ce qui nous laisse penser que cette complexité en états n'est pas atteinte. Néanmoins, par manque de temps, aucune preuve de cette hypothèse sera fourni.

Il serait donc intéressant d'étudier la complexité en état du grignotage plus en détail dans un autre article.

4 Le langage permuté

Dans cette section, nous nous intéressons à un langage particulier, appelé « langage permuté ». Nous commencerons par en donner une définition formelle. Nous introduirons ensuite un algorithme de « twistage » permettant de construire un automate non déterministe reconnaissant ce langage, à partir d'un automate initial. Nous établirirons ensuite la validité de cet algorithme par une preuve, et analyserons sa complexité dans le pire des cas. Enfin, nous conclurons cette section par une brève synthèse des résultats obtenus et des perspectives de prolongement.

4.1 Définition

★ **Définition 4.1** (Le langage permuté)
Le langage $Twist(L)$ (aussi noté $Permute(L)$) est construit à partir de L en échangeant toutes les paires de lettres d'indice $2k$ et $2k+1$ pour $k \in \mathbb{N}$:

$$Twist(\varepsilon) = \varepsilon, \qquad Twist(a) = a, \qquad Twist(a \cdot b \cdot v) = b \cdot a \cdot Twist(v),$$
$$\text{avec } (a, b) \in \Sigma^2 \text{ et } v \in \Sigma^*,$$
$$Twist(L) = \bigcup_{w \in L} Twist(w).$$

Exemple 4.1 :

$$L = \{a, ab, aba, baba\},$$
$$Twist(L) = \{a, ba, baa, abab\}.$$

Maintenant que nous savons ce qu'est le langage permuté, nous allons présenter un algorithme permettant de construire un automate le reconnaissant, à partir d'un automate reconnaissant le langage initial.

4.2 Algorithme de twistage

Notre algorithme prendra en entrée un automate non déterministe et renverra un automate non déterministe. Mais, avant de rentrer dans le dur de l'algorithme, nous regardons le principe de l'algorithme, puis nous verrons sa définition formelle.

4.2.1 Principe de l'algorithme

Avant de twister un automate quelconque, on va s'intéresser à des cas particuliers pour essayer de trouver une logique commune.

Le premier cas intéressant sont les automates qui forment qu'une longue « branche », c'est-à-dire que chaque état n'a qu'une transition vers un autre état différent de lui-même et que l'automate ne possède qu'un seul état initial :

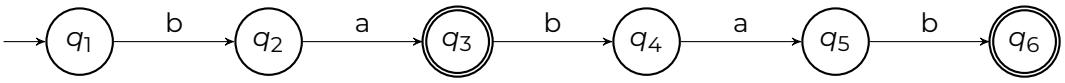

FIGURE 6 – Exemple d'un automate qui forme qu'une « branche ».

En effet, pour les automates de cette forme, il nous suffit d'échanger les paires de transitions d'indice $2k$ et $2k+1$ pour $k \in \mathbb{N}$, sauf pour la dernière transition si la branche est de longueur impaire :

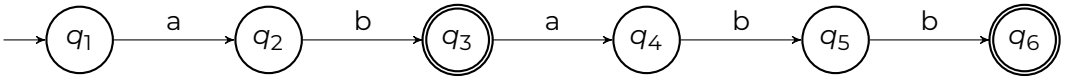

FIGURE 7 – Exemple de l'automate figure 6 après twistage.

De plus, les automates qui forment des arborescences dont les *sections interembranchements* sont de longueur paire et avec des longueurs de *sections terminales* quelconques, sont intéressants :

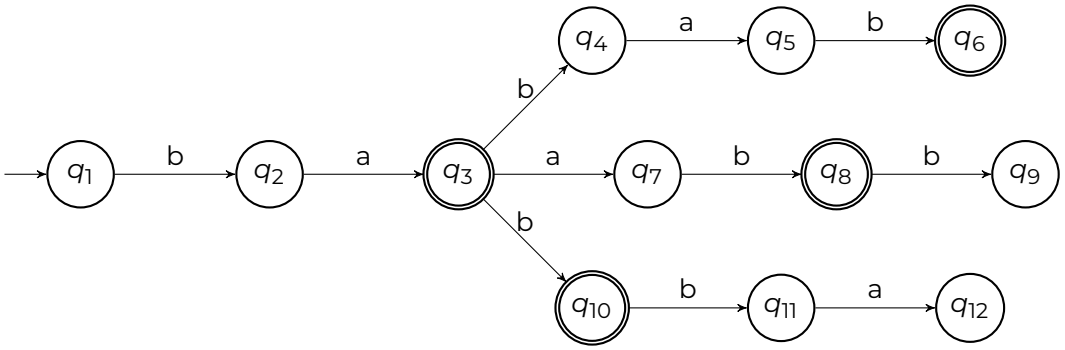

FIGURE 8 – Exemple d'un automate qui forme une arborescence dont les sections interembranchements sont de longueur paire.

On devra alors appliquer la même logique que l'on a utilisée pour branches, mais pour chaque section cette fois-ci :

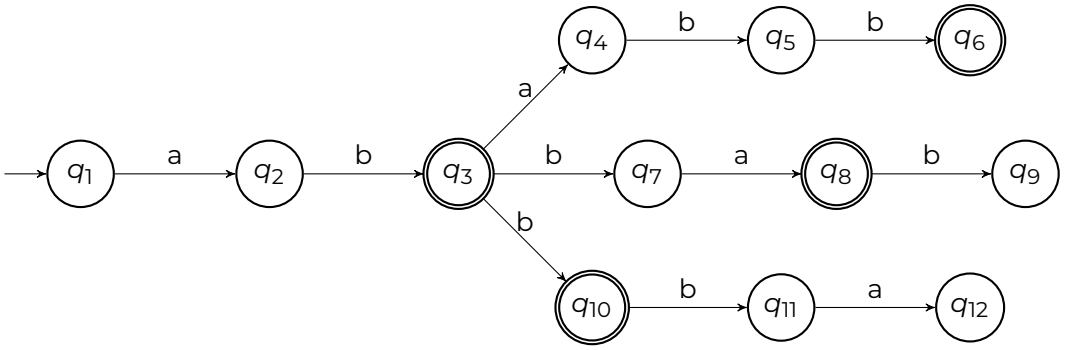

FIGURE 9 – Exemple de l'automate figure 8 après twistage.

Enfin, considérons les automates qui forment des arborescences dans lesquelles les sections interembranchements de longueur impaire ne se séparent que par un seul symbole. Dans ce cas, si la section suivante a une longueur paire, elle doit également se séparer par un seul symbole. En revanche, si cette section est de longueur impaire, alors aucune contrainte ne s'applique à elle :

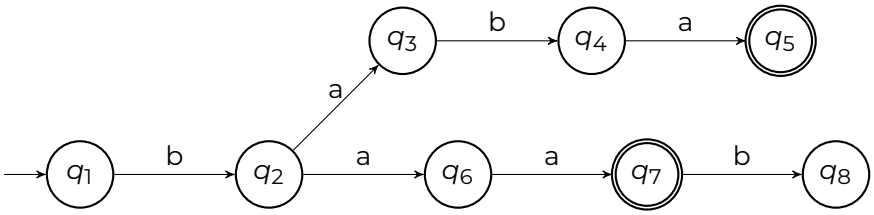

FIGURE 10 – Exemple d'un automate forme une arborescence avec des sections interembranchements contraintes.

On pourra ensuite appliquer la même logique qu'avant sans que l'automate reconnaisse des mots qu'il ne devrait pas :

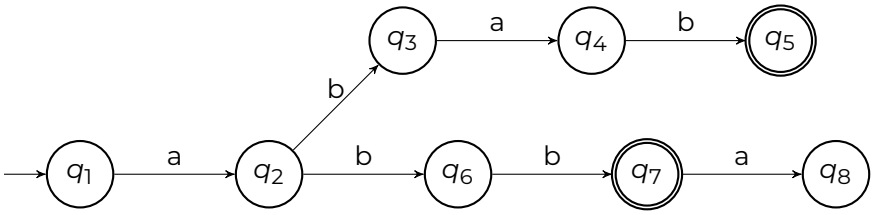

FIGURE 11 – Exemple de l'automate figure 10 après twistage.

Nous avons donc vu tous les automates sur lesquels il « suffit » d'échanger les paires de transitions pour que ça marche. Cependant, nous allons maintenant voir un automate avec lequel il faut avoir une étape de transition pour le transformer dans un cas qu'on vient d'observer.

Prenons maintenant un automate qui forme une arborescence, mais qui ne respecte pas les conditions que nous avons fixées précédemment :

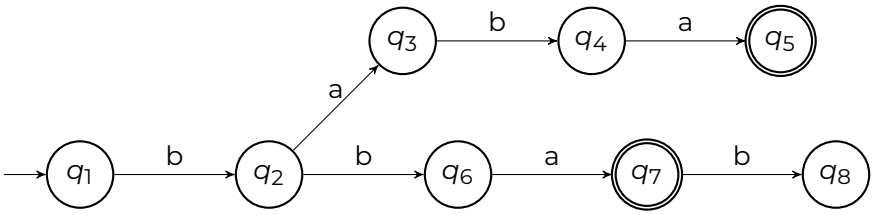

FIGURE 12 – Exemple d'un automate forme une arborescence avec des sections interembranchements sans contraintes.

Dans ce cas, puisque l'automate de la figure 12 ne respecte pas les contraintes fixées précédemment (notamment sur la parité des longueurs des sections interembranchements et leur séparation par un unique symbole), on ne peut pas simplement échanger les paires de transitions comme dans les exemples précédents.

Pour résoudre cela, transformons l'automate en dupliquant l'état d'embranchement (ici q_2) autant de fois qu'il y a de transitions sortantes avec des symboles différents. Dans cet exemple, q_2 possède deux transitions sortantes : l'une étiquetée par a et l'autre par b. On créera donc deux copies distinctes de q_2, chacune dédiée à un seul symbole de transition. Cela permet d'isoler les chemins et de leur appliquer ensuite les règles précédentes de transformation (comme l'échange des paires de transitions).

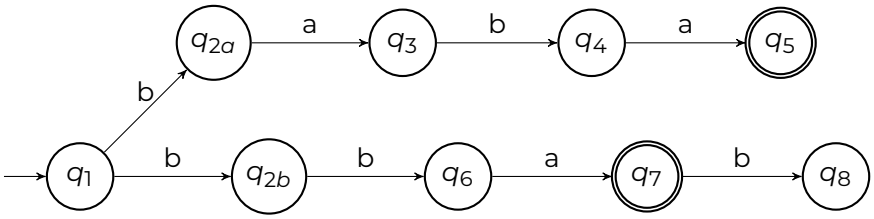

FIGURE 13 – Exemple de l'automate figure 12 après la duplication de q_2.

Il nous reste donc à appliquer les précédentes règles de transformation :

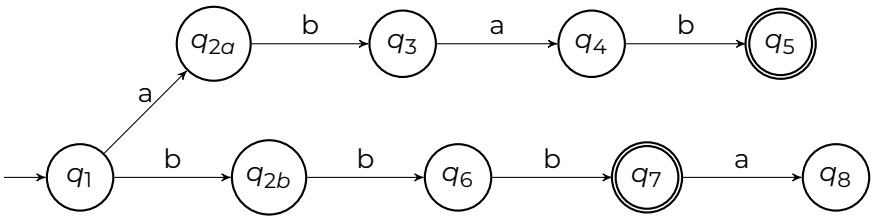

FIGURE 14 – Exemple de l'automate figure 13 après le twistage.

Maintenant que nous savons transformer tous les automates qui forment des arborescences, pour pouvoir transformer n'importe quel automate, nous allons devoir introduire des « cycles » (notions de théorie des graphes).

Lorsque des cycles sont présents, un même état peut être atteint par plusieurs chemins différents, parfois de longueurs impaires. Cela complique l'application des règles de transformation. Ainsi, chaque état intermédiaire devra, comme nous l'avons fait précédemment avec q_2, être dupliqué autant de fois que nécessaire. Cette duplication permet d'isoler les différents contextes d'accès à cet état, en fonction de la parité du chemin qui y mène, et donc garantir que les règles de transformation restent applicables même en présence de cycles.

Dans notre algorithme final, il ne sera pas nécessaire d'avoir une phase de prétraitement séparée pour transformer l'automate. En effet, l'algorithme intégrera directement la transformation (duplication des états) ainsi que le twistage.

4.2.2 Formalisation de l'algorithme

Maintenant que nous avons l'idée de l'algorithme, nous verrons sa définition formelle :

```
1 twister(M):
2   N ← (∅, ∅, ∅, ∅ → ∅)
3   for q in N.I do
4     N.Q ← N.Q∪{q}
5     N.I ← N.I∪{q}
6     N.F ← N.F∪(N.F∩{q})
7     twister_aux(M, q, N)
8   return N
```

```
twister_aux(M, q, N):
  for (p, a) in N.δ(q) do
    if p ∈ M.F then
      N.Q ← N.Q ∪ {p_a}
      N.F ← N.F ∪ {p_a}
      N.δ ← N.δ ∪ {(q, a, p_a)}
    for (r, b) in N.δ(p) do
      A ← N
      A.Q ← A.Q ∪ {p_b,r}
      A.F ← A.F ∪ (M.F ∩ {r})
      A.δ ← A.δ ∪ {(q, b, p_b),(p_b, a, r)}
      if r ∉ N.Q then
        A ← twister_aux(M, r, A)
      N ← A
  return N
```

! Remarque 4.1

Cet algorithme parcourt les paires de transitions sortantes d'un état donné, puis duplique l'état intermédiaire en fonction du symbole de transition utilisé. Il applique ensuite l'opération de twistage en inversant l'ordre des deux transitions, avant de poursuivre récursivement le traitement sur l'état d'arrivée de la paire de transitions.

4.3 Preuve de notre algorithme

Pour prouver que notre algorithme est correct et construit bien l'automate du langage permuté, nous allons d'abord montrer que tous mots reconnus par l'automate sont des mots appartenant au langage permuté. Enfin, nous allons montrer que tous les mots du langage permuté sont reconnus par l'automate.

4.3.1 Tous les mots reconnus sont des mots du langage permuté

■ Preuve 4.1

Soit $M \in NFA(\Sigma, \eta)$, N = `twister`(M) et un mot w reconnu par l'automate N. Pour prouver qu'il fait partie du langage permuté, nous allons faire une preuve par récurrence sur la longueur de ce mot :

- **Initialisation :** $|w| \leq 1$
  - Si $|w| = 0$, c'est qu'alors qu'un des états initiaux est finaux aussi, or dans l'algorithme, on duplique les états initiaux (en les laissant finaux s'ils le sont) donc cela veut dire que M reconnait lui aussi ε. Ainsi, $w \in Twist(L(M))$.

  - Si $|w| = 1$, c'est qu'alors après en une transition d'un état initial, il y a un état final. Or dans l'algorithme, s'il y a une transition qui amène vers un état final, alors cette transition est gardée et l'état reste final. Par conséquent, cela veut dire que l'automate M possède aussi cette transition et le permuté d'un mot composé d'un symbole est-ce même mot. Ainsi, $w \in Twist(L(M))$.

- **Hérédité :** $|w| \geq 2$

  On peut donc décomposer le mot w tel que $w = a \cdot b \cdot u$, avec $(a, b) \in \Sigma^2$ et $u \in \Sigma^*$.
  S'il est reconnu, cela veut dire qu'il existe un chemin partant d'un état initial i avec une

transition en a puis en b. Or dans notre algorithme, nous inversons les paires de transitions en partant des états initiaux, cela veut dire que dans l'automate M il existe une transition partant du même état initial i en b ensuite en a. Il reste donc à savoir si le mot u fait partie du langage permuté de l'automate M qu'aurait comme états initiaux $\delta(i, b \cdot a)$. Or dans notre algorithme, nous faisons un appel récursif pour calculer l'automate twister en partant de i, ainsi par hypothèse de récurrence $w \in Twist(L(M))$.

□

4.3.2 Tous les mots du langage permuté sont reconnus

■ **Preuve 4.2** (Esquisse de preuve)
Nous procédons également par récurrence sur la longueur des mots du langage de base $L(M)$. L'objectif est de montrer que pour tout mot $w \in L(M)$, sa permutation correspondante (par inversion de paires de transitions) est bien reconnue par l'automate $N = \texttt{twister}(M)$.

• **Initialisation :** $|w| \leq 1$
Alors w est soit le mot vide, soit un mot d'un seul symbole. Dans les deux cas, comme vu précédemment, l'algorithme conserve ces cas en dupliquant les états initiaux et finaux sans modification, ce qui garantit que $Twist(w) \in L(N)$.

• **Hérédité :** $|w| \geq 2$
Soit $w = b \cdot a \cdot u \in L(M)$, d'après la construction de l'algorithme, les transitions sont inversées par paire. Ainsi, si M reconnaît le préfixe $b \cdot a$, alors l'automate N reconnaît le préfixe $a \cdot b$. Par hypothèse de récurrence, la mot $Twist(u)$ est également reconnue par N, ce qui conclut que $Twist(w) \in L(N)$.

□

4.4 Pire cas de notre algorithme

Dans cette section, nous établirons une borne supérieure sur le nombre d'états de l'automate que notre algorithme peut engendrer. Cette estimation, distincte de la complexité du langage permuté, nous offre néanmoins un plafond pour la complexité en états.

Dans notre pire cas, notre algorithme dupliquera tous les états par au pire le nombre de symboles de l'alphabet plus un (puisqu'on a les états indicés et ceux non indicés). Par conséquent, dans le pire des cas, l'automate produit aura $n(|\Sigma| + 1)$ états, avec n le nombre d'états de l'automate de départ.

4.5 Conclusion

Pour conclure, nous venons de créer un algorithme qui prend n'importe quel automate en entrée et qui calcule l'automate du langage permuté de cet automate. De plus, notre algorithme crée un automate dans le pire des cas à $n(|\Sigma| + 1)$. états. Par conséquent, la complexité en états du twistage d'un langage est d'au moins $n(|\Sigma| + 1)$.

Malgré la petite complexité, on suppose que cette complexité n'est pas éteinte. Néanmoins, par manque de temps, aucune preuve de cette hypothèse ne sera pas fournie. Il serait donc intéressant d'étudier la complexité en état du twistage plus en détail dans un autre article.

5 Conclusion

Durant cet article, nous avons défini deux transformations particulières de langages réguliers : le trognon d'un langage et le langage permuté. Pour chacune d'elles, nous avons conçu un algorithme capable de construire, à partir d'un automate non déterministe initial, un nouvel automate reconnaissant le langage transformé.

Nous avons également fourni une preuve de correction pour chaque algorithme, montrant que les automates produits reconnaissent exactement les langages visés. Enfin, nous avons analysé la complexité en nombre d'états dans le pire des cas.

Des perspectives intéressantes s'ouvrent à la suite de ce travail. En particulier, affiner les bornes de complexité.

Références

[1] MARKUS HOLZER et MARTIN KUTRIB. « NONDETERMINISTIC DESCRIPTIONAL COMPLEXITY OF REGULAR LANGUAGES ». In : *International Journal of Foundations of Computer Science* 14.06 (2003), p. 1087-1102. DOI : 10.1142/S0129054103002199.

[2] A. N. MASLOV. « Estimates of the number of states of finite automata ». In : *Soviet Mathematics - Doklady* 11 (1970), p. 1373-1375.

[3] Ken THOMPSON. « Programming techniques : Regular expression search algorithm ». In : *Communications of the ACM* 11.6 (1968), p. 419-422.