# Automate des suffixes Application à la recherche d'un motif dans un texte

Laurent Chéno
Lycée Louis-le-Grand, Paris

Colloque Luminy 2001

# 1 Automate des suffixes

Pour cette section, la référence de choix est :
C. Allauzen, M. Raffinot, Algorithme simple et optimal de recherche d'un mot dans un texte, IGM 99-14, Institut Gaspard Monge, 1999
`http://www-igm.univ-mlv.fr/LabInfo/rapportsInternes/99-14.ps.gz`

## 1.1 Un exemple

Dessinons l'arbre des suffixes correspondant à la chaîne $p = baabbaa$ (voir ci-après).

**Remarque 1** On a marqué (avec \$) la fin de la chaîne, pour que les suffixes se retrouvent tous aux feuilles.

On colorie ensuite de la même façon les racines des sous-arbres identiques.
Ceci permet d'obtenir un automate qui *résume* l'information de l'arbre.

**Remarque 2** Cette fois, on n'a plus besoin de marqueur de fin de chaîne. Les suffixes correspondent aux états finals de l'automate.

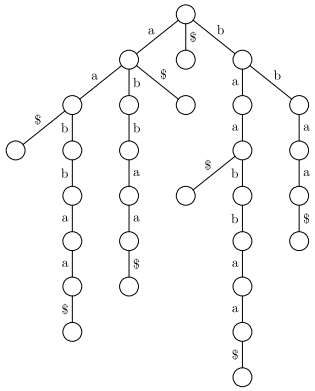

FIG. 1: un arbre des suffixes

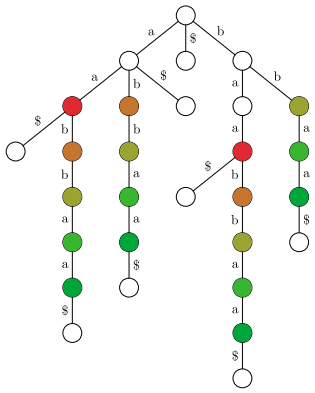

FIG. 2: l'arbre des suffixes colorié

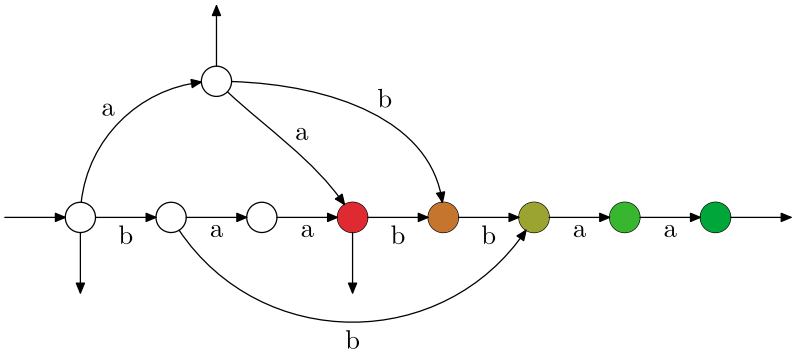


FIG. 3: l'automate des suffixes colorié

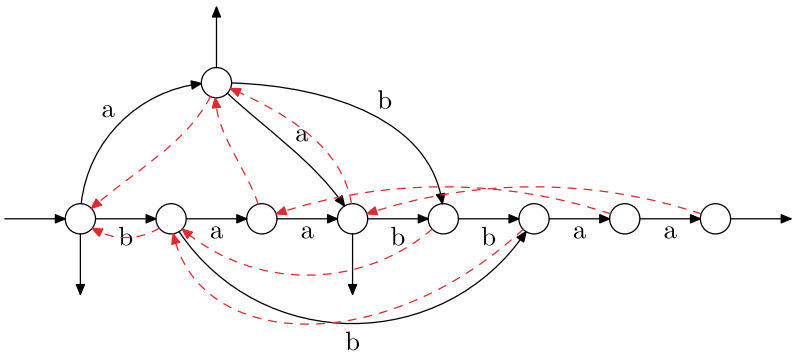


FIG. 4: l'automate des suffixes et la fonction de suppléance

## 1.2 Notations et définition de l'automate

L'alphabet (fini) de travail est noté $\Sigma$. Le mot vide est noté $\varepsilon$.
La longueur (ou taille) d'un mot $u$ est notée $|u| \in \mathbb{N}$.
L'intervalle d'entiers $\{i \in \mathbb{Z}, p \leqslant i \leqslant q\}$ est ici noté $[p..q]$.
Le mot à rechercher, $p = p_1 \dots p_m$, de longueur $|p| = m$, est fixé.
On note $\mathrm{Fact}(p)$ (*resp.* $\mathrm{Suff}(p)$) l'ensemble des facteurs (*resp.* des suffixes) de $p$.
$p^R$ dénote le *miroir* de $p$ : $p^R = p_m \dots p_1$.
La notation $u \preccurlyeq v$ signifie : $u$ *est suffixe de* $v$. La relation $\preccurlyeq$ est un ordre partiel sur $\mathrm{Fact}(p)$.
Si $u$ est un facteur de $p$, on note

$$\mathrm{endpos}\, u = \{|xu|, p = xuy\} \in \mathcal{P}([|u|..|p|]).$$

On définit, sur $\mathrm{Fact}(p)$, une relation d'équivalence en posant :

$$u \equiv v \iff \mathrm{endpos}\, u = \mathrm{endpos}\, v.$$

Dans la suite, nous noterons $\overline{u}$ la classe d'un facteur $u$ de $p$.

**Remarque 3** On peut généraliser la définition ci-dessus, en décrétant que si $u \notin \mathrm{Fact}(p)$ alors $\mathrm{endpos}\, u = \emptyset$.

**Remarque 4** La classe de $\varepsilon$ est réduite à $\{\varepsilon\}$.

**Remarque 5** Si $u \equiv v$ alors l'un des deux mots ($u$ et $v$) est suffixe de l'autre.

**Remarque 6** On en déduit que toute classe est totalement ordonnée par la relation $\preccurlyeq$.

Dans la suite, nous noterons $\rho(q)$ le plus long mot appartenant à une classe $q$.
Nous disposons donc, pour toute classe $q$, de : $u \in q \Longrightarrow u \preccurlyeq \rho(q)$.

Attention, la réciproque est fausse ! On a vu par exemple que la classe de $p$ ne contient en général pas tous les suffixes de $p$. (Ce n'est le cas que si la dernière lettre de $p$ ne figure pas auparavant dans $p$.)

**Lemme 1** Soient $u$ et $v$ deux facteurs de $p$.
Si $u$ est suffixe de $v$, alors on dispose de : $\mathrm{endpos}\, v \subset \mathrm{endpos}\, u$.

Autrement dit : $u \preccurlyeq v \Longrightarrow \mathrm{endpos}\, v \subset \mathrm{endpos}\, u$.

Si $v = wu$ et $i \in \mathrm{endpos}\, v$ : $\exists x, y,\ p = xvy = xwuy$ avec $i = |xv| = |xwu|$.

**Lemme 2** Soient $u$ et $v$ deux facteurs de $p$.
Si $u \equiv v$ alors, pour toute lettre $\alpha \in \Sigma$, on a : $u\alpha \equiv v\alpha$.

Si par exemple $u$ est suffixe de $v$, alors $u\alpha$ est suffixe de $v\alpha$ et donc $\mathrm{endpos}(v\alpha) \subset \mathrm{endpos}(u\alpha)$.
Si $i \in \mathrm{endpos}(u\alpha)$, alors $i - 1 \in \mathrm{endpos}\, u = \mathrm{endpos}\, v$ donc on peut écrire $p = xu\alpha y = x'vy'$ avec $|xu| = |x'v| = i - 1$ et donc $y' = \alpha y$. Mais alors $p = x'v\alpha y$ avec $|x'v\alpha| = i \in \mathrm{endpos}(v\alpha)$.

**Corollaire 1** Si $u$ et $v$ sont deux facteurs de $p$ alors

$$u \equiv v \Longrightarrow \forall w \in \Sigma^\star, uw \equiv vw.$$

**Définition 1 (Automate des suffixes)** L'automate des suffixes du mot $p$ est défini de la façon suivante :

- son ensemble d'états est $Q = \{\overline{u}, u \in \operatorname{Fact} p\}$ ;
- l'état initial est $q_0 = \overline{\varepsilon} = \{\varepsilon\}$ ;
- l'ensemble des états finals est $F = \{\overline{u}, u \in \operatorname{Suff} p\}$ ;
- la fonction de transition est

$$\delta : (\overline{u}, \alpha) \mapsto \begin{cases} \overline{u\alpha}, & \text{si } u\alpha \in \operatorname{Fact} p\,; \\ \text{indéfini}, & \text{sinon.} \end{cases}$$

C'est bien sûr le lemme 2 qui donne du sens à la définition de $\delta$.
Nous noterons $\delta^*(\overline{u}, w) = \overline{uw}$, et pour alléger les notations, si $q \in Q$ et $w \in \Sigma^\star$, $q.w$ désignera $\delta^*(q, w)$.

**Théorème 1** Soient $u$ et $v$ deux facteurs de $p$.
Alors : $u \equiv v \iff u^{-1}.\operatorname{Suff}(p) = v^{-1}.\operatorname{Suff}(p)$.

Les classes sont donc les résiduels de $\operatorname{Suff}(p)$ !

Il suffit d'observer pour conclure qu'on dispose des équivalences suivantes :

$$\begin{aligned} y \in u^{-1}.\operatorname{Suff}(p) &\iff uy \in \operatorname{Suff}(p) \\ &\iff \exists x,\ p = xuy \\ &\iff \begin{cases} |p| - |y| \in \operatorname{endpos} u \\ y \in \operatorname{Suff}(p) \end{cases} \end{aligned}$$

On en déduit l'important

**Théorème 2 (Minimalité de l'automate des suffixes)** L'automate des suffixes de $p$

- est déterministe, accessible et coaccessible ;
- est en général non complet ;
- reconnaît les suffixes de $p$ ;
- est minimal parmi les automates qui reconnaissent les suffixes de $p$.

Remarquons que pour rendre l'automate complet (ce que nous ne ferons pas), il suffirait d'ajouter la classe $\emptyset$ des mots non facteurs de $p$.
L'automate est co-accessible car si $u \in \operatorname{Fact} p$, $p = xuy$ et donc $\overline{u}.y = \overline{uy} \in F$, puisque $uy$ est un suffixe de $p$.

## 1.3 Fonction de suppléance

D'aucuns parlent plutôt de *liens suffixes*.

**Définition 2 (Fonction de suppléance)** La fonction de suppléance $s$ de l'automate des suffixes de $p$ associe à tout facteur $v$ de $p$ distinct de $\varepsilon$ le plus grand suffixe $u$ de $v$ tel que $u \not\equiv v$.

**Remarque 7** Si $v \in \mathrm{Fact}(p) \setminus \{\varepsilon\}$, $\overline{v} \neq \overline{\varepsilon} = \{\varepsilon\}$, donc $s(v)$ existe bien.

**Remarque 8** On a pour tout facteur non vide $v$ :

$$s(v) \prec v \quad \text{et} \quad \operatorname{endpos} v \subsetneq \operatorname{endpos} s(v).$$

On peut dessiner les liens suffixes sur l'automate des suffixes :

**Théorème 3** Soient $u$ et $v$ deux facteurs non vides de $p$. Alors :

$$u \equiv v \Longrightarrow s(u) = s(v).$$

Il s'agit bien d'une égalité, non d'une équivalence !

Par exemple, $u \preccurlyeq v$.
On ne peut avoir $u \preccurlyeq s(v)$ car sinon

$$\operatorname{endpos} u = \operatorname{endpos} v \subsetneq \operatorname{endpos} s(v) \subset \operatorname{endpos} u.$$

Donc $s(v) \preccurlyeq u \preccurlyeq v$. Ainsi, $s(v)$ est un suffixe de $u$, $\overline{s(v)} \neq \overline{v} = \overline{u}$, et si $w$ est un suffixe de $u$ (donc de $v$) plus grand que $s(v)$, alors $w \equiv v \equiv u$.
C'est bien dire que $s(v) = s(u)$.

Le théorème 3 permet de définir une application $S : Q \setminus \{q_0\} \to Q$ par $S(q) = \overline{s(u)}$ pour n'importe quel représentant $u$ de $q$.
Mais en fait on a mieux : $s(u)$ n'est pas n'importe quel élément de $S(q)$, c'est précisément $\rho(S(q))$, c'est-à-dire le plus grand élément de $S(q)$.

**Lemme 3** Pour tout facteur $u$ non vide de $p$, on a : $s(u) = \rho(S(\overline{u}))$.

Soit $v = \rho(\overline{u})$ et $w = s(u) = s(v)$. Soit $w'$ un autre élément de $S(\overline{u})$ : $w \equiv w'$.
On a $\operatorname{endpos} u = \operatorname{endpos} v \subsetneq \operatorname{endpos} w = \operatorname{endpos} w'$.
Montrons que $|w'| \leqslant |w|$ : sinon, $w$ serait suffixe de $w'$, mais $w'$ et $v$ sont comparables par $\preccurlyeq$ et le lemme 1 interdit que $v \preccurlyeq w'$. Ainsi $w'$ serait suffixe de $v$, $|w'| > |w| = |s(v)|$ et $w' \not\equiv v$, ce qui contredirait la définition de $s$.

**Lemme 4** Soit $q$ un état de l'automate des suffixes, distinct de $q_0$, et $u \in q$. Alors $s(u)$ (et donc également tout mot de $S(q)$) est suffixe de tout mot reconnu en $q$.

Si $v \in q$, c'est que $u \equiv v$ et on sait qu'alors $s(u) = s(v)$, qui par définition est suffixe de $v$.

**Définition 3 (Chemin de suppléance)** Le chemin de suppléance d'un état $q$ de l'automate des suffixes est la suite (finie) des itérées de $S$ sur $q$ : $CS(q) = \{q, S(q), S(S(q)), \ldots, q_0\}$.

On a montré que $q_0 = \{\varepsilon\} \prec \cdots \prec S(S(q)) \prec S(q) \prec q$, où on note, pour deux classes $p$ et $q$, $p \preccurlyeq q$ (*resp.* $p \prec q$) si tout mot de $p$ est suffixe (*resp.* suffixe propre) de tout mot de $q$.
De même a-t-on $\operatorname{endpos} q \subsetneq \operatorname{endpos} S(q) \subsetneq \cdots \subsetneq \operatorname{endpos} \varepsilon = [0..|p|]$, où on note (naturellement), pour toute classe $q$, $\operatorname{endpos} q = \operatorname{endpos} u$ pour tout mot $u \in q$.

**Remarque 9** Le chemin de suppléance de $\overline{p}$ est l'ensemble des états terminaux de l'automate, comme l'indique la figure suivante.

**Remarque 10** L'arbre de suppléance est aussi l'arbre d'inclusion des $\mathrm{endpos}\, q$, $q \in Q$.

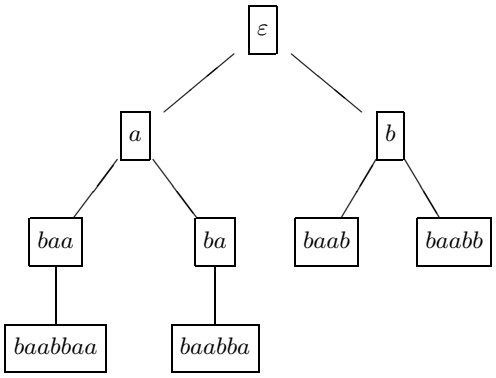


FIG. 5: l'arbre de suppléance (et, en rouge, les états finals)

Ce théorème est la clé de l'algorithme FDM.

**Théorème 4** Soit $\alpha \in \Sigma$, fixé. Soit $q$ un état de l'automate des suffixes, tel qu'il n'y ait pas de transition depuis $q$ à la lecture de $\alpha$.
Soit $p$ l'éventuel premier état rencontré dans la lecture de $CS(q)$ (lu dans l'ordre, de $q$ à $q_0$) tel que $\delta(p, \alpha)$ existe.
Alors, notant $v = \rho(p)$, $v$ est le plus grand suffixe des mots reconnus en $q$ tel que $v\alpha$ soit un facteur de $p$.

Supposons l'existence d'un tel état $p$. On sait que $v$ est suffixe de tout mot reconnu en $q$.
Dire que $\delta(p, \alpha)$ existe, c'est dire exactement que $v\alpha$ est un facteur de $p$.
Montrons qu'on aboutirait à une contradiction si l'on supposait l'existence d'un suffixe de tous les mots reconnus en $q$ tel que $w\alpha \in \operatorname{Fact} p$ et $|w| > |v|$.
En effet, si $r = \overline{w}$, $\delta(r, \alpha)$ existe puisque $w\alpha \in \operatorname{Fact} p$.
Montrons que $r \in CS(q)$ : sinon, il existerait un état $q' \in CS(q)$ tel que $r$ serait la classe d'un plus grand suffixe $w$ des mots de $q'$, avec $r \neq S(q')$, ce qui est contraire à la définition de la fonction de suppléance.
Mais alors, par définition de $p$, $r$ doit être rencontré **après** $p$, ce qui contredit $|w| > |v|$.

Enfin, nous définissons les *arcs solides* de l'automate et les *longueurs* de ses états.

**Définition 4 (Longueur des états)** Pour tout état $q$ de l'automate des suffixes, on appelle longueur de $q$ et on note $\operatorname{long}(q) = |\rho(q)|$ la taille du plus grand facteur de $p$ reconnu en $q$.

**Définition 5 (Arcs solides)** Une transition $q' = \delta(q, \alpha)$ de l'automate des suffixes est dite solide si $\operatorname{long}(q') = 1 + \operatorname{long}(q)$.

La figure suivante montre l'automate associé au mot $p = baabbaa$, avec en tireté rouge les arcs de la fonction de suppléance, les longueurs des états sont écrites dans chaque état, les arcs solides sont dessinés en trait gras.

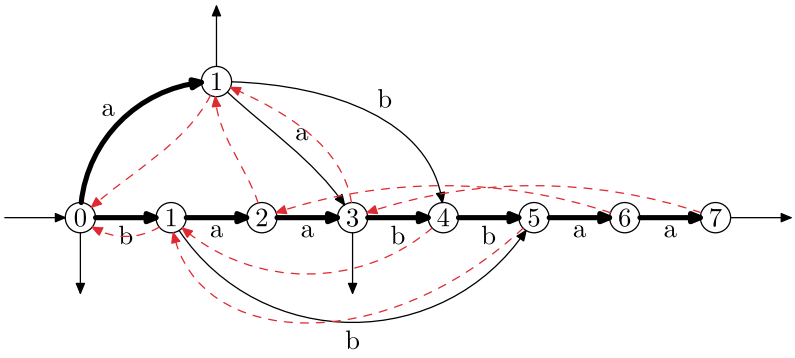


FIG. 6: arcs solides de l'automate des suffixes

Le tableau ci-dessous récapitule les données relatives à l'automate des suffixes du mot $p = baabbaa$.

| $v \in q$ | $\rho(q)$ | long $q$ | endpos $q$ | $S(q)$ | $v^{-1}.\operatorname{Suff}(p)$ |
|---|---|---|---|---|---|
| $\varepsilon$ | $\varepsilon$ | 0 | 01234567 | – | Suff$(p)$ |
| $a$ | $a$ | 1 | 2367 | $\varepsilon$ | $\varepsilon, a, bbaa, abbaa$ |
| $b$ | $b$ | 1 | 145 | $\varepsilon$ | $aa, baa, aabbaa$ |
| $aa, baa$ | $baa$ | 3 | 37 | $a$ | $\varepsilon, bbaa$ |
| $ab, aab, baab$ | $baab$ | 4 | 4 | $b$ | $baa$ |
| $bb, abb, aabb, baabb$ | $baabb$ | 5 | 5 | $b$ | $aa$ |
| $bba, abba, aabba, baabba$ | $baabba$ | 6 | 6 | $ba$ | $a$ |
| $bbaa, abbaa, aabbaa, baabbaa$ | $baabbaa$ | 7 | 7 | $baa$ | $\varepsilon$ |

## 2 Construction de l'automate

Pour cette section, la référence de choix est :
M. Crochemore, W. Rytter, Text algorithms, Oxford University Press, 1994
qui est hélas épuisé (et était de toutes façons absolument hors de prix !).

---

Nous décrivons ici l'algorithme proposé par Crochemore et Rytter pour construire, en temps linéaire, l'automate des suffixes d'un mot $p = p_1 \dots p_m$, qu'on lit de gauche à droite.
On va successivement construire les automates des suffixes des mots $p_1$, $p_1p_2$, $p_1p_2p_3$, etc. On prendra soin de distinguer les arcs solides des autres, et on tiendra à jour une table de calcul de la fonction long qu'on a définie précédemment.
Durant le déroulement de l'algorithme, nous construirons également les liens suffixes. Pour décider des états finals, il suffira de remonter les liens suffixes depuis l'état $\overline{p}$, comme on l'a déjà dit.
Nous prendrons à titre d'illustration l'exemple de notre chaîne préférée : $p = baabbaa$, pour laquelle les figures suivantes montrent les automates calculés successivement.

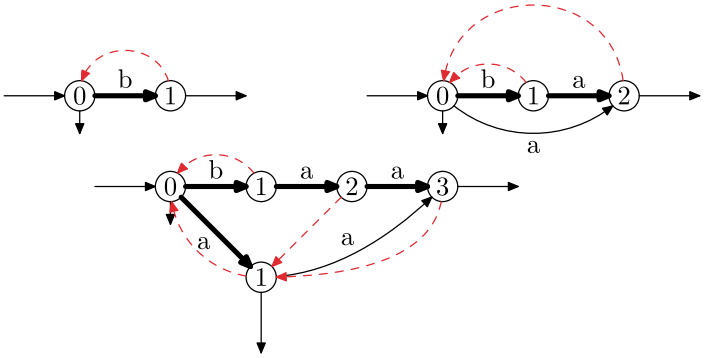


FIG. 7: les automates des suffixes de $b$, $ba$, $baa$

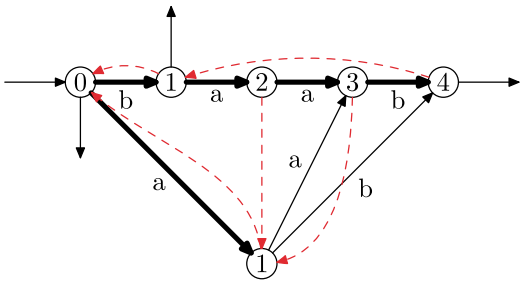


FIG. 8: l'automate des suffixes de $baab$

Supposons que nous soyons à l'étape $i$ : nous avons déjà construit l'automate des suffixes du mot $p_1p_2 \dots p_{i-1}$ et nous lisons la lettre $p_i$.
On crée un nouvel état $q$, et on ajoute une transition étiquetée par $p_i$ partant de chaque état de $CS(p_1 \dots p_{i-1})$ vers $q$. Bien sûr, on pose aussi : $\text{long}(q) = i$.

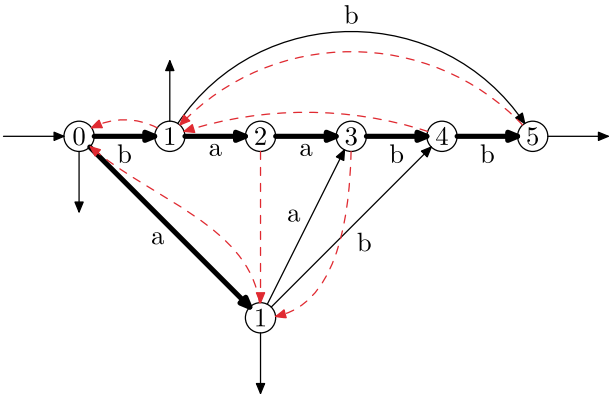


FIG. 9: l'automate des suffixes de *baabb*

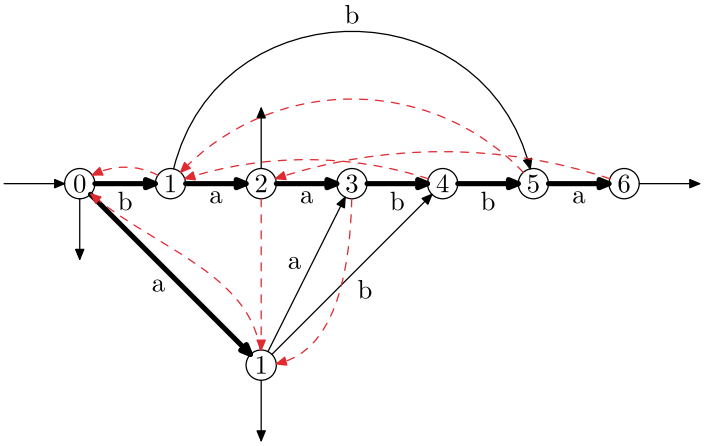


FIG. 10: l'automate des suffixes de *baabba*

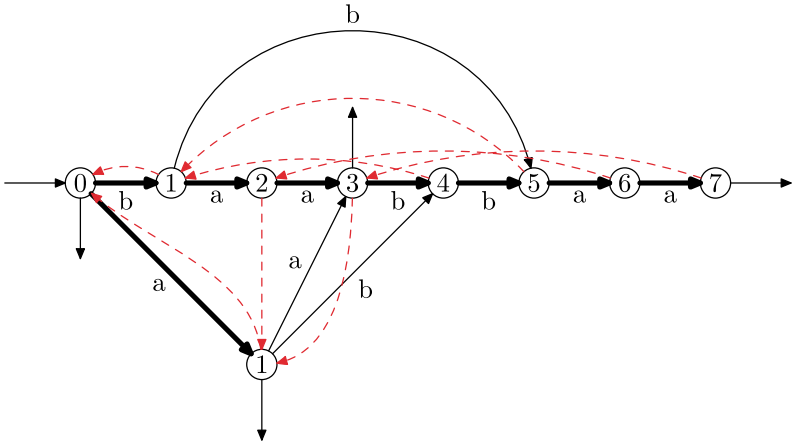


FIG. 11: l'automate des suffixes de *baabbaa*

On arrête s'il existe déjà une transition depuis $r \in CS(p_1 \dots p_{i-1})$, étiquetée par $p_i$, vers un état $r'$, et on distingue alors le cas où cet arc est solide ou pas.

**Cas d'un arc solide** Alors $\rho(r')$ est suffixe de $p_1 \dots p_i$, et il suffit de poser $s(q) = r'$ (on n'ajoute alors évidemment pas la transition de $r$ à $q$).
C'est ce qui arrive ici quand on passe de $baa$ à $baab$ : on commence par ajouter des arcs de $\overline{baa}$ puis $\overline{a}$ vers le nouvel état ; ensuite, comme il existe déjà un arc (solide) de $\overline{\varepsilon}$ vers $\overline{b}$, c'est donc le lien suffixe du nouvel état créé.

**Cas d'un arc non solide** C'est plus compliqué !
Il existe donc un arc non solide de $r \in CS(p_1 \dots p_{i-1})$ à $r'$ étiqueté par $p_i$. On duplique cet état $r'$ en un nouvel état $t$, qui sera le lien suffixe de $q$. Les arcs sortant de $t$ sont recopiés des arcs sortant de $r'$ ; $t$ hérite du lien suffixe qu'avait $r'$ ; enfin, tous les états suivants dans le chemin suffixe verront leur éventuel arc sortant étiqueté par $p_i$ redirigé vers $t$.
C'est ce qui arrive ici quand on passe de $ba$ à $baa$ : il existe déjà un arc (non solide) de $\overline{\varepsilon}$ vers $\overline{ba}$, on crée donc un nouvel état qui sera le nouveau $\delta(\overline{\varepsilon}, a)$ et le lien suffixe de $\overline{baa}$ .

**Algorithme** On peut écrire formellement :

Initialiser l'automate de racine $r_0$, avec $\text{long}(r_0) = 0$ et $s(r_0)$ indéfini
$dernier \leftarrow r_0$
**Pour** $i \in \{1, \ldots, |p|\}$ **faire**
  $r \leftarrow dernier$
  Créer un nouvel état $q$
  $\text{long}(q) \leftarrow \text{long}(dernier) + 1$
  **Tant que** $(r \neq r_0) \wedge (\delta(r, p_i)$ indéfini$)$ **faire**
    $\delta(r, p_i) \leftarrow q$
    $r \leftarrow s(r)$
  **Fait**
  **Si** $\delta(r, p_i)$ indéfini **alors**
    $\delta(r_0, p_i) \leftarrow q$
    $s(q) \leftarrow r_0$
  **sinon**
    $r' \leftarrow \delta(r, p_i)$
    **Si** $(1 + \log r = \log r')$ **alors** $s(q) \leftarrow r'$
    **sinon**
      $t \leftarrow Duplique(r, r', p_i)$
      $s(q) \leftarrow t$
  $dernier \leftarrow q$
**Fait**

La fonction de duplication $Duplique(p, q, a)$ s'écrit ainsi :

Créer un nouvel état $r$
**Pour tout arc** issu de $q$ étiqueté par $b$ **faire**
  $\delta(r, b) \leftarrow \delta(q, b)$
**Fait**
$\log r \leftarrow 1 + \log p$
$s(r) \leftarrow s(q)$
$s(q) \leftarrow r$
**Faire**
  $\delta(p, a) \leftarrow r$
  $p \leftarrow s(p)$
**Tant que** $(p \neq$ indéfini$) \wedge (\delta(p, a) = q)$

On peut montrer et nous admettrons le

**Théorème 5 (Coût de la construction de l'automate des suffixes)** L'algorithme précédent permet de construire l'automate des suffixes d'un mot $p$ en temps $\mathcal{O}|p|$ et en mémoire $\mathcal{O}(|p| \times |\Sigma|)$.

**Remarque 11** On peut observer que l'algorithme décrit ici créera un maximum de $2|p| + 1$ états, puisqu'à chaque étape on crée soit un soit deux nouveaux états.

**Exercice**
Dessiner l'automate des suffixes de *aabbab*, puis celui de *aabbabb*.

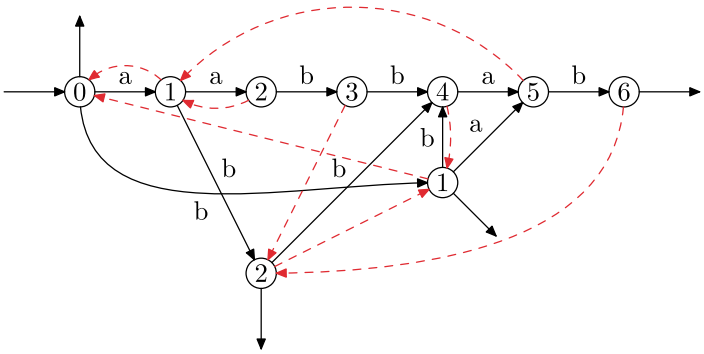


FIG. 12: l'automate des suffixes de *aabbab*

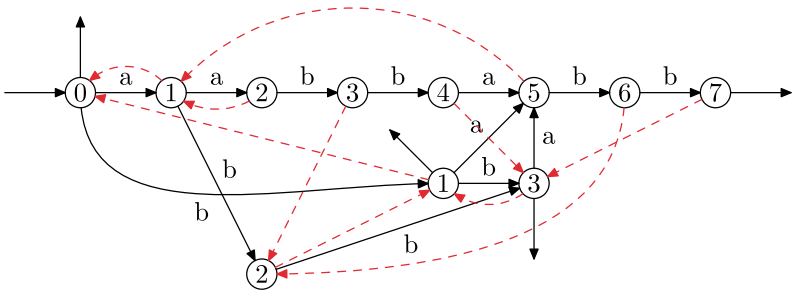


FIG. 13: l'automate des suffixes de *aabbabb*

# 3 FDM : *Forward DAWG Matching*

Pour cette section, la référence de choix est encore :
C. Allauzen, M. Raffinot, Algorithme simple et optimal de recherche d'un mot dans un texte, IGM 99-14, Institut Gaspard Monge, 1999
`http://www-igm.univ-mlv.fr/LabInfo/rapportsInternes/99-14.ps.gz`
Les auteurs présentent en outre un algorithme qui est lui optimal, fondé sur l'utilisation simultanée de deux algorithmes FDM, et qu'ils ont donc baptisé DFDM *(Double Forward DAWG Matching)*.
Nous laissons sa découverte au lecteur intéressé...

---

On cherche un mot $p = p_1 \dots p_m$ dans un texte $t = t_1 \dots t_n$ (avec, en général, $n \gg m$), tous les deux sur le même alphabet $\Sigma$.
L'idée est la suivante : pour chaque position *pos* du texte, on calcule la longueur du plus grand facteur de $p$ qui soit suffixe de $t_1 \dots t_{pos}$.
On commence par calculer l'automate des suffixes du motif $p$. Au début, on se trouve dans l'état initial, $\ell$ vaut 0, on n'a encore lu aucun caractère de $t$.
À un instant donné, on a lu $t_1 \dots t_{pos}$, on est dans un état $q$, et le suffixe de longueur $\ell$ de $t_1..t_{pos}$ est un facteur de $p$.

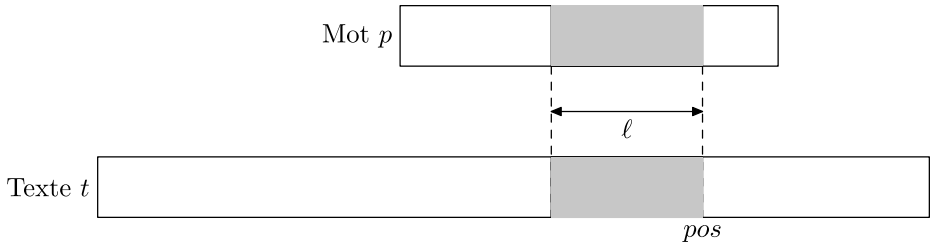


FIG. 14: la situation courante de l'algorithme FDM

On lit alors le caractère $t_{pos+1}$ :

- s'il existe une transition de $q$ par $t_{pos+1}$ vers un état $q'$, l'état courant devient $q'$, et $\ell$ est incrémenté d'une unité ;
- sinon, on remonte le chemin de suppléance de $q$ jusqu'à trouver un état qui ait une transition sortante par $t_{pos+1}$. De deux choses l'une :
  1. ou bien il existe un tel état $r$ qui admet une transition par $t_{pos+1}$ vers un état $r'$ , et l'état courant devient $r'$, la longueur courante devient $\text{long}(r) + 1$ ;
  2. ou bien il n'y en a pas, et on repart de l'état initial de l'automate avec une longueur nulle.

On vérifie :

**Théorème 6 (Validité du FDM)** À chaque étape de l'algorithme, $\ell$ est la longueur du plus long suffixe de $t_1 \dots t_{pos}$ qui est aussi facteur de $p$, et ce facteur est reconnu par l'automate des suffixes en l'état $q$.

et on en déduit bien sûr le

**Corollaire 2** $p$ apparaît dans $t$ en position $pos$ si et seulement si la longueur courante $\ell$ est égale à $m = |p|$.

**Remarque 12** La complexité du FDM est, en moyenne comme dans le pire des cas, $\mathcal{O}(|p| + |t|)$.

On peut écrire formellement l'algorithme FDM de la façon suivante :

Calculer l'automate des suffixes $\mathcal{AS}$ de $p$
$\ell \leftarrow 0$ ; $q \leftarrow q_0$, état initial de $\mathcal{AS}$
$pos \leftarrow 0$
**Tant que** $(\ell < |p|) \wedge (pos < |t|)$ **faire**
    $pos \leftarrow pos + 1$ ; $c \leftarrow t[pos]$
    **Si** $\delta(q, c)$ est défini **alors**
        $q \leftarrow \delta(q, c)$ ; $\ell \leftarrow \ell + 1$
    **sinon**
        **Faire** $q \leftarrow s(q)$
        **tant que** ($q$ défini) $\wedge$ ($\delta(q, c)$ non défini)
        **Si** $q$ indéfini **alors**
            $q \leftarrow q_0$ ; $\ell \leftarrow 0$
        **sinon**
            $\ell \leftarrow \text{long}(q) + 1$ ; $q \leftarrow \delta(q, c)$
**Fait**
**Si** $\ell = pos$ **alors Réussite**$(pos)$
**sinon Échec**

# 4 BDM : *Backward DAWG Matching*

Pour cette section, la référence de choix est encore :
M. Crochemore, W. Rytter, Text algorithms, Oxford University Press, 1994
mais je trouve plus lisible et plus claire la présentation (dans le chapitre 2) de la thèse de Mathieu Raffinot qu'on trouve en outre facilement sur Internet :
`http://genome.genetique.uvsq.fr/~raffinot/ftp/These.ps.gz`

---

On cherche un mot $p = p_1 \dots p_m$ dans un texte $t = t_1 \dots t_n$ (avec, en général, $n \gg m$), tous les deux sur le même alphabet $\Sigma$.
L'idée est la suivante : on déplace une fenêtre de largeur $m = |p|$ le long du texte, à l'intérieur de laquelle on procède à la lecture du texte, mais cette fois *à rebours*, c'est-à-dire de droite à gauche.

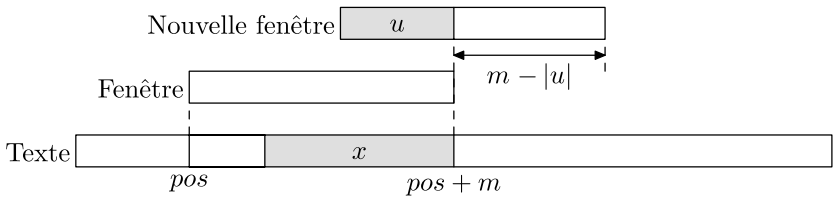


FIG. 15: le décalage effectué par l'algorithme BDM

On commence donc par calculer l'automate des suffixes du **miroir** du motif : $p^R = p_m \dots p_1$.
La recherche dans la fenêtre positionnée en $i$ permet de trouver le plus long facteur $x$ de $p$ qui est aussi suffixe de $t_{1..i+m}$.
Deux cas peuvent se produire alors :

- si $|x| = m$, on a trouvé une occurrence de $p$ dans $t$ ;
- sinon, c'est qu'on bloque sur un caractère $c$ : $cx$ n'est pas un facteur de $p$ ; dans ce cas, on peut recommencer la recherche après avoir décalé notre fenêtre de la position *pos* à la position $pos+m-|x|$.

En réalité, à peu de frais, on peut améliorer la situation dans le second cas, en augmentant la valeur de décalage.
En effet, on pourrait décaler la fenêtre de recherche de $m - |u|$, où $u$ est le plus long préfixe de $p$ (ou $u^R$ le plus long suffixe de $p^R$) qui est aussi suffixe de $x$ : comme $|u| < |x|$, le décalage est meilleur car

$$m - |u| > m - |x|.$$

Mais comment trouver ce mot $u$ ?
Il suffit, quand on utilise l'automate des suffixes de $p^R$, de se rappeler les moments où on passe par un état final de l'automate, puisque ce sont justement eux les suffixes de $p^R$ !
En sauvegardant le dernier passage, on se souvient du plus long suffixe de $p^R$, donc du plus long préfixe de $p$, qui est aussi suffixe de $x$, c'est-à-dire de $u$.

On peut écrire formellement l'algorithme BDM de la façon suivante :

Calculer l'automate des suffixes $\mathcal{AS}$ de $p^R$
$pos \leftarrow 0$
**Tant que** $(pos \leqslant n - m)$ **faire**
    $j \leftarrow m$
    $q \leftarrow q_0$, état initial de $\mathcal{AS}$
    **Tant que** ($q$ défini) $\wedge$ $(j > 0)$ **faire**
        **Si** $(j > 1) \wedge (q$ est terminal) **faire** $dernier \leftarrow j$
        $q \leftarrow \delta(q, t[pos + j])$
        $j \leftarrow j - 1$
    **Fait**
    **Si** $(j = 0)$ **alors Réussite**$(pos + 1)$
    $pos \leftarrow pos + dernier$
**Fait**
**Échec**

On démontre le

**Théorème 7 (Performance du BDM)** Le coût moyen du BDM est

$$\mathcal{O}(|p| + |t|\frac{\log_{|\Sigma|}|p|}{|p|}),$$

ce qui est optimal.

**Remarque 13** En revanche, dans le cas le pire, la complexité devient quadratique !

**Remarque 14** C'est Yao, en 1979, qui a trouvé le premier cette expression du coût minimum. (A.C. Yao, *The complexity of pattern matching for a random string*, SIAM J. Comput. 8(3))

# 5 Et pour prolonger l'étude...

Il existe, on l'a déjà signalé, de nombreuses améliorations possibles pour chacun des deux algorithmes FDM et BDM.
On peut aussi très facilement généraliser les algorithmes décrits au cas où l'on recherche toutes les occurrences d'un motif et non seulement la première.
Il est également possible de généraliser au cas de la recherche d'un ensemble de motifs.
Enfin, une piste prometteuse est à suivre : plutôt qu'utiliser l'automate des suffixes, auquel on peut reprocher la place mémoire qu'il utilise, on peut préférer un *oracle des suffixes* ou un *oracle des facteurs*.
À suivre !...

# 6 Programmes en Caml

**Programme 1** Typage des automates

```
let rec copie_liste = function
    | [] -> []
    | t :: q -> t :: (copie_liste q) ;;

type état == int ;;
type table_des_transitions == (char * état) list vect ;;
type table_des_liens_suffixes == état vect ;;

type automate = { mutable finals : état list ;
                  mutable nb_états : int ;
                  mutable transitions : table_des_transitions ;
                  mutable suppléance : table_des_liens_suffixes ;
                  mutable longueurs : int vect ;
                } ;;

let delta dawg q c =
    assoc c dawg.transitions.(q) ;;

let get_delta = delta ;;

let pas_de_transition dawg q c =
    not (mem_assoc c dawg.transitions.(q)) ;;

let set_delta dawg q c q' =
    dawg.transitions.(q) <- (c,q') :: dawg.transitions.(q) ;;

let get_s dawg = function
    | 0 -> raise Not_found
    | q -> dawg.suppléance.(q) ;;

let set_s dawg q q' =
    dawg.suppléance.(q) <- q' ;;

let get_len dawg q =
    dawg.longueurs.(q) ;;

let set_len dawg q l =
    dawg.longueurs.(q) <- l ;;
```

**Programme 2** Construction de l'automate des suffixes

```
29 let duplique dawg p q a =
30     let r = dawg.nb_états
31     in
32     let rec répète p =
33         set_delta dawg p a r ;
34         try
35             let p = get_s dawg p
36             in
37             if delta dawg p a = q then répète p
38         with Not_found -> ()
39     in
40     dawg.nb_états <- dawg.nb_états + 1 ;
41     dawg.transitions.(r) <- copie_liste dawg.transitions.(q) ;
42     dawg.longueurs.(r) <- dawg.longueurs.(p) + 1 ;
43     dawg.suppléance.(r) <- dawg.suppléance.(q) ;
44     dawg.suppléance.(q) <- r ;
45     répète p ;
46     r ;;

47 let rec calcule_finals dawg p =
48     dawg.finals <- p :: dawg.finals ;
49     try calcule_finals dawg (get_s dawg p)
50     with Not_found -> () ;;

51 let automate_des_suffixes p =
52     let n = (string_length p + 1) * 2
53     in
54     let dawg =
55         { finals = [] ;
56           nb_états = 1 ;
57           transitions = make_vect n [] ;
58           suppléance = make_vect n 0 ;
59           longueurs = make_vect n 0 }
60     in
61     let dernier = ref 0
62     in
63     for i = 0 to string_length p - 1 do
64         let r = ref !dernier
65         and q = dawg.nb_états
66         in
67         dawg.nb_états <- dawg.nb_états + 1 ;
68         dawg.longueurs.(q) <- dawg.longueurs.(!dernier) + 1 ;
69         while (0 <> !r) && (pas_de_transition dawg !r p.[i]) do
70             set_delta dawg !r p.[i] q ;
71             r := get_s dawg !r
72         done ;
73         begin
74             try let r'= delta dawg !r p.[i] in
75                     if dawg.longueurs.(!r) + 1 = dawg.longueurs.(r') then
76                         dawg.suppléance.(q) <- r'
77                     else
78                         dawg.suppléance.(q) <- duplique dawg !r r' p.[i]
79             with Not_found -> set_delta dawg 0 p.[i] q ;
80                               dawg.suppléance.(q) <- 0
81         end ;
82         dernier := q
83     done ;
84     calcule_finals dawg !dernier ;
85     dawg ;;
```

**Programme 3** Algorithme FDM

```
86  exception Réussite of int ;;

87  let fdm t p =
88      let dawg = automate_des_suffixes p
89      and m = string_length p
90      and l = ref 0
91      and q = ref 0
92      in
93      for pos = 0 to string_length t - 1 do
94          let c = t.[pos]
95          in
96          begin
97              if pas_de_transition dawg !q c then
98              begin
99                  try while pas_de_transition dawg !q c do
100                         q := get_s dawg !q
101                     done ;
102                     l := dawg.longueurs.(!q) + 1 ;
103                     q := delta dawg !q c
104                 with Not_found -> ( q := 0 ; l := 0 )
105             end
106             else
107             begin
108                 q := delta dawg !q c ;
109                 incr l
110             end
111         end ;
112         if !l = m then raise (Réussite(pos - m + 1))
113     done ;
114     false ;;
```

**Programme 4** Algorithme BDM

```
115 let reverse_string s =
116     let n = string_length s
117     in
118     let r = create_string n
119     in
120     for i = 0 to n - 1 do r.[i] <- s.[n-i-1] done ;
121     r ;;

122 let est_final dawg q =
123     mem q dawg.finals ;;

124 let bdm t p =
125     let dawg = automate_des_suffixes (reverse_string p)
126     and m = string_length p
127     and n = string_length t
128     in
129     let pos = ref 0
130     and dernier = ref m
131     and q = ref 0
132     in
133     while !pos <= n - m do
134         q := 0 ;
135         try    for j = m - 1 downto 0 do
136                 if est_final dawg !q then dernier := j + 1 ;
137                 q := delta dawg !q t.[!pos + j]
138             done ;
139             raise (Réussite(!pos))
140         with Not_found -> pos := !pos + !dernier
141     done ;
142     false ;;
```