Oracle des facteurs, Oracle des suffixes

Cyril Allauzen Maxime Crochemore Mathieu Raffinot

Institut Gaspard-Monge, Université de Marne-la-Vallée,
Cité Descartes, Champs-sur-Marne,
77454 Marne-la-Vallée Cedex 2, France.
{allauzen,mac,raffinot}@monge.univ-mlv.fr

6 mai 1999

Résumé

Nous définissons un nouvel automate sur un mot $p = p_1 p_2 \dots p_m$, suite de lettres prises dans un alphabet Σ, que nous appelons *oracle des facteurs*. Cet automate est acyclique, reconnait au moins les facteurs de p, a $m+1$ états, est linéaire en nombre de transitions. Nous donnons un algorithme de construction séquentielle de l'oracle des facteurs. Les liens étroits de cette structure avec l'automate des suffixes nous permettent d'introduire une deuxième structure, l'oracle des suffixes. Ces deux structures sont utilisées dans des algorithmes de recherche de mots, que nous conjecturons optimaux en moyenne, au vu des résultats expérimentaux. Ces algorithmes sont aussi efficaces que ceux qui existent déjà, tout en utilisant moins de place mémoire et en étant beaucoup plus simples à implémenter.

1 Introduction

Un *mot* p est une séquence finie $p = p_1 p_2 \dots p_m$ de lettres prises dans un alphabet Σ. Nous garderons la notation p tout au long de cette partie pour représenter le mot sur lequel on travaille.

Nous cherchons à construire un automate (a) acyclique (b) qui reconnaisse au moins les facteurs de p (c) qui ait le moins d'états possible (d) et qui soit linéaire en nombre de transitions. D'ores et déjà, on peut remarquer qu'un tel automate a nécessairement au moins $m+1$ états. L'automate des suffixes ou des facteurs [3, 6] satisfait (a)-(b)-(d) mais pas (c), et l'automate des sous-mots [1] satisfait (a)-(b)-(c) mais pas (d). Nous proposons dans cet article une construction d'un automate de $m+1$ états qui satisfait ces 4 critères, que l'on appelle *oracle des facteurs*.

En nous inspirant de la structure de l'automate des suffixes, nous introduisons une nouvelle structure, l'*oracle des suffixes*.

Nous utilisons ces deux nouvelles structures dans des algorithmes de recherche de mots. Ces algorithmes ont un très bon comportement moyen, que nous conjecturons optimal. Les deux principaux intérêts de nos nouveaux algorithmes de recherches de mots sont (1) leur simplicité d'implémentation pour un comportement optimal (2) l'économie mémoire réelle que permet de réaliser l'oracle des suffixes par rapport à l'automate des suffixes.

Nous avons besoin tout au long de cet article de différentes notions et définitions que nous donnons maintenant.

Un mot $x \in \Sigma^*$ est un facteur de p si p peut être écrit $p = uxv$, $u, v \in \Sigma^*$. On note $\mathrm{Fact}(p)$ l'ensemble des facteurs du mot p. Un facteur x de p est appelé un *préfixe* (resp. *suffixe*) de p si $p = xu$, $u \in \Sigma^*$ (resp. $p = ux$, $u \in \Sigma^*$). L'ensemble des préfixes est appelé $\mathrm{Pref}(p)$ et celui des suffixes $\mathrm{Suff}(p)$. On dit que x est facteur propre (resp. préfixe propre, suffixe propre) de p si x est facteur (resp. préfixe, suffixe) différent du mot vide et de p.

On note $\mathrm{pref}_p(i)$ le préfixe de longueur i de p pour $i \leq |p|$.

On définit pour $u \in \mathrm{Fact}(p)$, $\mathrm{poccur}(u,p) = \min\{|z|\ ,\ z = wu \text{ et } p = wuv\}$, la position de la première occurrence de u dans p.

Enfin, on définit pour $u \in \mathrm{Fact}(p)$ l'ensemble $\mathrm{endpos}_p(u) = \{i \mid p = wup_{i+1}\ldots p_m\}$. Si deux facteurs u et v de p sont tels que $\mathrm{endpos}_p(u) = \mathrm{endpos}_p(v)$, on note $u \sim_p v$. On peut vérifier que $\sim_p$ est une relation d'équivalence; c'est en fait l'équivalence syntaxique du language $\mathrm{Suff}_p(v)$.

2 Oracle des facteurs

2.1 Algorithme de construction

Construit_Oracle$(p = p_1p_2\ldots p_m)$
1. Pour i de 0 à m
2. Créer un état i
3. Pour i de 0 à $m-1$
4. Contruire un arc de i à $i+1$ d'étiquette p_{i+1}
5. Pour i de 0 à $m-1$
6. Soit u un mot minimal de l'état i
7. Pour tout $\sigma \in \Sigma, \sigma \neq p_{i+1}$
8. Si $u\sigma \in \mathrm{Fact}(p_{i-|u|+1}\ldots p_m)$
9. Construire un arc de i à $i + \mathrm{poccur}(u\sigma, p_{i-|u|+1}\ldots p_m)$ par σ

FIG. 1 – *Algorithme de construction de l'Oracle*

Définition 1 *On appelle* oracle des facteurs *d'un mot $p = p_1p_2\ldots p_m$ l'automate obtenu par l'algorithme Construit_Oracle (figure 1) sur le mot p, dont on considère tous les états terminaux. On le note Oracle(p).*

L'oracle des facteurs du mot $p = abbbaab$ est donné en exemple figure 2. Sur cet exemple, on peut remarquer que le mot aba est reconnu alors que ce n'est pas un facteur de p.

Remarque: toutes les transitions qui arrivent sur un état i de Oracle(p) sont étiquettée p_i.

Lemme 1 *Soit $u \in \Sigma^*$ un mot de longueur minimale parmi les mots reconnus dans l'état i de Oracle(p). Alors $u \in$ Fact(p) et $i =$ poccur(u,p).*

Preuve. Par récurrence sur le numéro de l'état i. C'est vrai pour l'état 0, ainsi que pour l'état 1. On suppose que c'est vrai pour tous les états $0 \leq j \leq i-1$. On se place sur l'état i, soit u

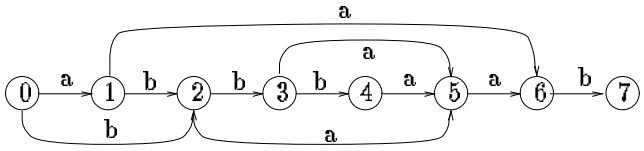


FIG. 2 – *Oracle des facteurs du mot abbbaab. Le mot aba est reconnu alors qu'il n'est pas facteur.*

un mot de longueur minimale parmi les mots reconnus dans l'état i. Considérons la dernière transition sur le chemin étiqueté par u menant de 0 à i.

(i) Si cette transition a été créé à la ligne 4, alors $u = zp_i$ avec $z \in \Sigma^*$. Comme u est un mot de longueur minimale de l'état i, z est un mot de longueur minimale de l'état $i-1$. D'après l'hypothèse de récurrence, $z \in \mathrm{Fact}(p)$ et $i - 1 = \mathrm{poccur}(z, p)$. Et ainsi $u = zp_i \in \mathrm{Fact}(p)$ et $i = \mathrm{poccur}(u, p)$.

(ii) Si cette transition a été créé à la ligne 9, alors elle mène de j à i par p_i avec $0 < j < i-1$. Ainsi $u = zp_i$ et z est un mot de longueur minimale de l'état j. D'après l'hypothèse de récurrence, $z \in \mathrm{Fact}(p)$ et $j = \mathrm{poccur}(z, p)$. Et d'après la construction de la transition à la ligne 9, $u = zp_i \in \mathrm{Fact}(p)$ et $i = \mathrm{poccur}(u, p)$.

□

Corollaire 1 *Soit $u \in \Sigma^*$ un mot de longueur minimale parmi les mots reconnus dans l'état i, alors u est unique.*

On note par la suite $\min(i)$ le mot de longueur minimale parmi les mots reconnus dans l'état i et on l'appelle *mot minimal* ou *minimum* de l'état i.

Corollaire 2 *Soient i et j deux états de Oracle(p) avec $j < i$. Soient $u = min(i)$ et $v = min(j)$, alors u ne peut pas être suffixe de v.*

Preuve. On suppose que u est suffixe de v. Dans ce cas, $\mathrm{poccur}(u, p) \leq \mathrm{poccur}(v, p)$ ce qui contredit (par le lemme 1) $j < i$. □

Lemme 2 *Soit i un état de Oracle(p) et $u = min(i)$. Alors u est suffixe de tout mot $c \in \Sigma^*$ étiquetant un chemin de l'état 0 à l'état i.*

Preuve. Par récurrence sur le numéro de l'état i. C'est vrai pour l'état 0, ainsi que pour l'état 1. On suppose que c'est vrai pour tous les états $0 \leq j \leq i-1$. On se place sur l'état i, soit $u = \min(i)$.

Soit c un chemin qui même à l'état i, $c = c_1 a$ et c_1 mène à un état $j < i$. Soit $v = \min(j)$. Alors

- $|va| \geq |u|$ car u le minimum de i et va mène à i.
- D'après la construction (figure 1 ligne 9), $i \in \mathrm{endpos}_p(va)$ car $v = \min(j)$ et il y a une transition de j vers i d'étiquette a.

– Le lemme 1 impose que $i \in \text{endpos}_p(u)$.

De ces trois faits on déduit que u est un suffixe de va. Comme $j < i$, par hypothèse de récurrence, v est suffixe de c_1, et u est suffixe de c. □

Lemme 3 *Soit $w \in Fact(p)$. w est reconnu par Oracle(p) en un état $j \leq poccur(w, p)$.*

Preuve. Par récurrence sur la longueur de $w = w_0 w_1 \dots w_f$. On note $i = \text{poccur}(w, p)$.

– Il existe une transition partant de l'état 0 par w_0 qui mène à un état $k_0 \leq i - f$.

– Supposons qu'il existe un chemin étiquetté par $w_0 \dots w_j$ qui mène à un état $k_j \leq i-f+j$. Soit $u = \min(k_j)$. D'après le lemme 2, $u = w_{j-|u|+1} \dots w_j$. Comme uw_{j+1} est un facteur de p et que $i - f + (j + 1) \in \text{endpos}_p(uw_{j+1})$, il existe une transition de k_j étiquettée par w_{j+1} qui arrive dans un état $k_{j+1} \leq i - f + (j + 1)$.

□

Remarque: Dans le lemme 3 précédent, j peut être strictement inférieur à $\text{poccur}(w, p)$. L'exemple donné figure 3 représente l'automate *Oracle(abbcabc)*, et l'état atteint après la lecture du mot *abc*, l'état 4, est strictement plus petit que $\text{poccur}(abc, p) = 7$.

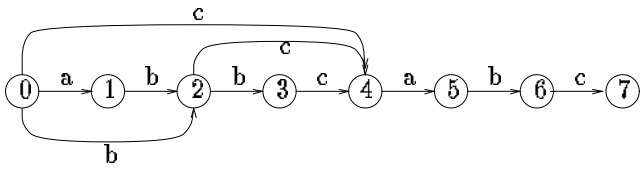


FIG. 3 – *Exemple de reconnaissance d'un facteur* (abc) *qui ne mène pas à la fin de la première occurrence.*

Corollaire 3 *Soit $w \in Fact(p)$. Tout mot $v \in Suff(w)$ est reconnu par Oracle(p) en un état $j \leq poccur(w)$.*

Lemme 4 *Soit i un état de Oracle(p) et $u = min(i)$. Alors tout chemin finissant par u arrive dans un état $j \geq i$.*

Preuve. Par récurrence sur le numéro de l'état i. C'est vrai pour l'état 0, ainsi que pour l'état 1. On suppose que c'est vrai pour tous les états $0 \leq j \leq i - 1$. On se place sur l'état i, soit u un mot minimal de la classe d'équivalence représentée par l' état i. On considère le chemin minimal d'étiquette u qui mène à i. On note j l'état précédant i sur ce chemin, et $v = \min(j)$. On a $u = va$ par construction de Oracle(p). Prenons un chemin $c = c_1 u$ qui arrive dans un état k. Supposons que $k < i$.

Par hypothèse de réccurrence, le chemin $c_1 v$ arrive dans un état $l \geq j$. Si $l = j$, alors $k = i$, ce qui contredit $k < i$. On a maintenant $j < l < k < i$. Soit $w = \min(l)$.

– Si $|w| > |v|$, alors v est un suffixe de w (lemme 2). Dans ce cas, $u = va$ est un suffixe de wa, si bien que $k \in \text{endpos}_p(u)$, ce qui contredit $k < i = \text{poccur}(u, p)$ (lemme 1).

– Si $|w| \leq |v|$, alors w est un suffixe de v (lemme 2). Ce qui contredit $j < l$ (corollaire 2).

On obtient une contradiction dans tous les cas, si bien que $k \geq i$ et l'hypothèse de récurrence est vérifiée pour i. □

Lemme 5 *Soit $w \in \Sigma^*$ un mot reconnu par Oracle(p) en i, alors tout suffixe de w est lui aussi reconnu en un état $j \leq i$.*

Preuve. Par récurrence sur $|w|$. C'est vrai si $|w| = 0$ ou $|w| = 1$. Supposons que c'est vrai pour tous les mots ζ tels que $|\zeta| < |w|$. On montre que c'est vrai pour w, reconnu en i.

Posons $w = \zeta a$, ζ est reconnu en $k < i$. Prenons un suffixe propre de w il s'écrit va où v est un suffixe propre de ζ. D'après l'hypothèse de récurrence, v est reconnu en $l \leq k$. Soient $\bar{\zeta} = min(k)$ et $\bar{v} = min(l)$. Le lemme 2 implique que $\bar{\zeta}$ est un suffixe de ζ et $\bar{v}$ un suffixe de v. Le corollaire 2 impose que $\bar{v}$ est un suffixe de $\bar{\zeta}$. Comme $i \in \text{endpos}_p(\bar{\zeta}a)$ (par construction de la transition par a), $i \in \text{endpos}_p(\bar{v}a)$. Il existe donc une transition de l par a arrivant dans un état $j \leq i$. Ainsi, le suffixe propre va est reconnu en j. □

Le nombre d'états de Oracle(p) avec $p = p_1 p_2 \ldots p_m$ est $m+1$. On s'intéresse à son nombre de transitions.

Lemme 6 *Le nombre $T_{Or}(p)$ de transitions de Oracle($p = p_1 p_2 \ldots p_m$) vérifie $m \leq T_{Or}(p) \leq 2m - 1$.*

Preuve. Il y a toujours m transitions de la forme $i \to i+1$ d'étiquette p_{i+1} créées à la ligne 4 de l'algorithme. Les mots de la forme a^m n'ayant que ces m transitions, m est le nombre minimal de transitions.

Considérons maintenant les transitions de la forme $i \to j$ avec $j > i+1$ créées à la ligne 9. On cherche à construire une fonction injective qui à chacune de ces transitions associe un suffixe propre de p. A chaque transition $i \to j$ avec $j > i+1$ d'étiquette σ, on associe le mot $\min(i)\sigma p_{j+1} \ldots p_m$. Par construction de Oracle(p), ce mot est un suffixe propre de p. On montre à contrario que cette fonction est injective. Supposons qu'il existe deux transitions distinctes $i_1 \to j_1$ et $i_2 \to j_2$ d'étiquettes respectives σ_1 et σ_2 telles que

$$\min(i_1)\sigma_1 p_{j_1+1} \ldots p_m = \min(i_2)\sigma_2 p_{j_2+1} \ldots p_m. \tag{1}$$

On suppose $i_1 \geq i_2$. On distingue trois cas.

(i) Si $j_1 = j_2$, on a alors $\sigma_1 = \sigma_2$ et l'égalité 1 entraine $\min(i_1) = \min(i_2)$ et $i_1 = i_2$.

(ii) Si $j_1 > j_2$, $\min(i_2)\sigma_2$ est un préfixe propre de $\min(i_1)$. Posons $\delta = |\min(i_1)| - |\min(i_2)\sigma_2|$, on a alors $j_2 = j_1 - \delta - 1$. Comme une occurrence de $\min(i_1)$ se termine en i_1 (d'après le lemme 1), une occurrence de $\min(i_2)\sigma_2$ finit en $i_1 - \delta < j_1 - \delta - 1 = j_2$. Ce qui contredit la construction de Oracle(p).

(iii) Si $j_1 < j_2$, $\min(i_1)$ est un préfixe propre de $\min(i_2)$, il existe donc une occurrence de $\min(i_1)$ qui se termine avant $i_2 \leq i_1$, ce qui est contraire au lemme 1.

Les trois autre cas, pour $i_1 \leq i_2$, sont obtenus de façon symétrique. La fonction est bien injective, et comme l'ensemble des suffixes propres de $p = p_1 p_2 \ldots p_m$ est de cardinal $m-1$, $T_{Or}(p) \leq m + m - 1 = 2m - 1$. Cette borne est atteinte pour les mots $a^{m-1}b$. □

2.2 Algorithme séquentiel

On présente dans cette section un algorithme séquentiel de construction de l'automate Oracle(p), c'est à dire permettant de construire l'automate en lisant les lettres de p une à une de gauche à droite.

On note $\text{repet}_p(i)$ le plus long suffixe de $\text{pref}_p(i)$ qui possède au moins deux occurrences distinctes dans $\text{pref}_p(i)$.

On introduit une fonction S_p sur les états de l'automate, dite fonction de suppléance, qui à chaque état $i > 0$ de Oracle(p) associe l'état j dans lequel se termine la lecture de $repet_p(i)$. On note arbitrairement $S_p(0) = -1$.
Remarques:

- $S_p(i)$ est définie pour tout état i de Oracle(p) (Corollaire 3).
- Pour tout état i de Oracle(p), $i > S_p(i)$ (lemme 3).

On note, $k_0 = m$, $k_i = S_p(k_{i-1})$ pour $i \geq 1$. La suite des k_i est finie, strictement decroissante, et se termine en l'état 0. On note

$$CS_p = \{k_0, k_1 \ldots, k_t = 0\}$$

le chemin suffixe de p dans Oracle(p).

Lemme 7 *Soient $k > 0$ un état de Oracle(p), $w_k = repet_p(k)$ tel que $s = S_p(k)$ soit strictement positif. On note $w_s = repet_p(s)$. Alors w_s est un suffixe de w_k.*

Preuve. Soit j l'état de Oracle(p) après la lecture de w_s. Soit $v = \min(s)$.

- $j < s$. Par le lemme 3, $j \leq \text{poccur}(w_s)$ qui par définition est strictement inférieur à s.
- $|w_s| < |v|$. Supposons le contraire. Alors il existe un chemin terminant par $v = \min(s)$ qui arrive en $j < s$, ce qui est contradictoire avec le lemme 4.

Par le lemme 1, $s = \text{poccur}(v, p)$. Comme w_s est un suffixe de $\text{pref}_s(p)$ de taille inférieure à $|v|$, w_s est un suffixe propre de v, et, comme v est lui-même un suffixe de w_k (lemme 2), w_s l'est aussi. □

Corollaire 4 *Soit $CS_p = \{k_0, k_1 \ldots, k_t = 0\}$ le chemin suffixe de p dans Oracle(p) et $w_i = repet_p(k_{i-1})$ pour $1 \leq i \leq t$, et $w_0 = p$. Alors, pour $0 < l \leq t$, w_l est un suffixe de tous les w_i, $0 \leq i < l \leq t$.*

On s'intéresse maintenant pour un mot $p = p_1 p_2 \ldots p_m$ et une lettre $\sigma \in \Sigma$ à la construction de Oracle($p\sigma$) à partir Oracle(p).

On note Oracle(p) + σ l'automate Oracle(p) auquel on a rajouté une transition par σ de l'etat m vers un état $m + 1$. On peut d'ores et déjà remarquer qu'une transition qui existe dans Oracle(p) + σ existe dans Oracle($p\sigma$), si bien que la différence entre les deux automates ne peut provenir que de transitions par σ vers l'état $m+1$ à rajouter dans Oracle(p) + σ pour obtenir Oracle($p\sigma$).

On recherche les états à partir desquels pouraient partir des transitions par σ vers l'état $m+1$.

Lemme 8 *Soit k un état de Oracle$(p)+\sigma$ tel qu'il existe une transition de k par σ vers l'état $m+1$ dans Oracle$(p\sigma)$. Alors k est un des états du chemin suffixe $CS = \{k_0 = m, k_1, \ldots, k_t = 0\}$ de p dans Oracle$(p)+\sigma$.*

Preuve. A contrario. Supposons qu'il existe un état $k_{i+1} < k < k_i$ pour lequel il existe une transition vers l'état $m+1$ par σ dans Oracle$(p\sigma)$. On note $w_j = repet_p(k_{j-1})$ pour $1 \leq j \leq t$, et $w_0 = p$. On a $m \in \text{endpos}_p(w_j)$. Soit $v = min(k)$. Comme il existe une transition par σ de k à $m+1$, $m \in \text{endpos}_p(v)$, et v est comparable aux facteurs w_j (corollaire 4). Le facteur v doit alors vérifier:

(i) $|v| < |w_i|$. Supposons au contraire que $|v| \geq |w_i|$. $|v| > |w_i|$, sinon il existe deux chemins d'étiquettes v menant à des états différents. Il existe un plus grand indice $0 \leq d < i$ tel que $|w_{d+1}| < |v| < |w_d|$. Le facteur v est suffixe de w_d. Comme $v = \min(k)$, d'après le lemme 1, il apparait en $k < k_d$, et comme il est aussi suffixe de w_d, il apparait au moins deux fois dans $\text{pref}_p(k_d)$. Dans ce cas, par définition des k_j, k_{d+1} ne peut pas être $S_p(d)$, et on a une contradiction. On a donc $|v| < |w_i|$ et v est un suffixe propre de w_i.

(ii) $|v| > |w_{i+1}|$. Supposons au contraire que $|v| \leq |w_{i+1}|$. $|v|$ est alors un suffixe propre de w_{i+1} et le chemin d'étiquette w_{i+1} menant à $k_{i+1} < k$ finit par $v = \min(k)$, ce qui est contradictoire avec le lemme 4.

(iii) $|v| < |\min(k_i)|$. Supposons au contraire que $|v| \geq |\min(k_i)|$. Le facteur $\min(k_i)$ est un suffixe de w_i (lemme 2) qui admet lui aussi v comme suffixe (par (i)). D'où $\min(k_i)$ est suffixe de $v = \min(k)$ et on obtient une contradiction avec le corollaire 2.

Comme v est suffixe de $\min(k_i)$ (par (iii)), $k_i \in \text{endpos}_p(v)$ (lemme 1). Or $k_i > k \in \text{endpos}_p(v)$, et v apparait au moins deux fois dans $\text{pref}_p(k_i)$. Comme (par (ii)) $|v| > |w_{i+1}|$, on obtient une contradiction avec le fait que $w_{i+1} = repet_p(k_i)$. □

Parmis les états du chemin suffixe de p, tous les états qui n'ont pas de transitions par σ dans Oracle$(p)+\sigma$ doivent en avoir une dans Oracle$(p\sigma)$. Plus formellement, le lemme suivant établit ce fait.

Lemme 9 *Soit $k_l < m$ un état du chemin suffixe $CS = \{k_0 = m, k_1, \ldots, k_t = 0\}$ de l'état m dans Oracle$(p = p_1p_2 \ldots p_m)+\sigma$, si k_l n'est pas suivi d'une transition par σ dans Oracle(p), alors il existe dans Oracle$(p\sigma)$ une transition par σ de k_l vers $m+1$.*

Preuve. Soit $v = \min(k_l)$, alors v est un suffixe de $w_l = repet_p(k_{l-1})$ (lemme 2). Comme w_l est un suffixe de p, $w_l\sigma$ est un suffixe de $p\sigma$, et $m+1 = \text{poccur}(w_l\sigma)$. Par contruction de Oracle$(p\sigma)$, il existe une transition par σ de k_l vers $m+1$. □

Lemme 10 *Soit $k_l < m$ un état du chemin suffixe $CS = \{k_0 = m, k_1, \ldots, k_t = 0\}$ de m dans Oracle$(p = p_1p_2 \ldots p_m)+\sigma$, si k_l a une transition par σ dans Oracle$(p)+\sigma$, alors tous les états k_i, $0 \leq i \leq l$ ont aussi une transition pas σ dans Oracle$(p)+\sigma$.*

Preuve. Soit $w_l = repet_p(k_{l-1})$. Tous les $w_i = repet_p(k_{i-1})$, $0 < i \leq l$ sont suffixes de w_l. Comme $w_l\sigma$ est reconnu par Oracle$(p) + \sigma$, par le lemme 5, tous ses suffixes le sont aussi. □

L'algorithme de construction *on-line* apparait petit à petit. Avec les trois lemmes 8, 9, 10, pour transformer Oracle$(p) + \sigma$ en Oracle$(p\sigma)$, il suffit de parcourir le chemin suffixe $CS = \{k_0 = m, k_1, \ldots, k_t = 0\}$ de l'état m, et tant que l'état k_l sur lequel on est n'a pas de transition sortante par σ, on en rajoute une de k_l à $m+1$ (lemme 9). S'il en a une, on s'arrête, car d'après le lemme 10, tous les k_j inférieurs auront déjà eux aussi une transition par σ.

Si on ne devait ajouter qu'une lettre pour construire l'automate, l'algorithme précédent suffirait. Seulement, dans l'idée de rajouter les lettres du mot p une à une, il est nécessaire d'être capable de mettre à jour la fonction de suppléance $S_{p\sigma}$ du nouvel automate Oracle$(p\sigma)$. Comme (par définition de S_p), la fonction de suppléance des états $0 \leq i \leq m$ ne change pas de Oracle(p) à Oracle$(p\sigma)$, la seule opération à effectuer est le calcul de $S_{p\sigma}(m+1)$ dans Oracle$(p\sigma)$. C'est l'objet du lemme 11 suivant.

Lemme 11 *S'il existe un état k_d plus grand élément de $CS_p = \{k_0 = m, k_1, \ldots, k_t = 0\}$ dans Oracle(p) tel qu'il existe une transition sortante de k_d vers un état s par σ dans Oracle(p), alors $S_{p\sigma}(m+1) = s$ dans Oracle$(p\sigma)$. Sinon $S_{p\sigma} = 0$.*

Preuve. Soit $w = \text{repet}_{p\sigma}(m+1)$ dans Oracle$(p\sigma)$. Supposons tout d'abord qu'il n'existe pas de tel état. Comme $0 \in CS_p$ dans Oracle(p), il n'existe pas de transition par σ partant de 0 dans Oracle(p), donc σ n'est pas une lettre de p, et $w = \epsilon$ et $S_{p\sigma} = 0$.

On suppose maintenant qu'il existe un tel état k_d. Alors w n'est pas le mot vide, on peut donc écrire $w = \alpha\sigma$. De plus $k_d < m$, car m est le dernier état de Oracle(p). On pose $w_j = \text{repet}_p(k_{j-1})$ pour $0 < j \leq t$ et $w_0 = p$. On prouve tout d'abord les deux points suivant:

(1) $|\alpha| < |w_{d-1}|$. Supposons au contraire que $|\alpha| \geq |w_{d-1}|$, alors w_{d-1} est un suffixe de α. Comme $\alpha\sigma$ est un facteur de p (il se répète à deux occurrences distinctes dans $p\sigma$), $\alpha\sigma$ est reconnu dans Oracle(p) (lemme 5) et $w_{d-1}\sigma$ aussi. On obtient une contradiction avec le fait que d est le plus petit indice tel que k_d admet une transition sortante par σ dans Oracle(p).

(2) Soit i l'état de reconnaissance de α dans Oracle(p). i est strictement inférieur à k_{d-1}, d'après le lemme 5 et le fait que i est suivi d'une transition par σ alors que k_d non.

On compare maintenant α et w_d. L'un est suffixe de l'autre. On obtient deux cas.

(i) Supposons que $|\alpha| \geq |w_d|$. $i \geq k_d$ car w_d est suffixe de α. On prouve à contrario que $k_d = i$. Supposons que $k_d < i$. Alors $|w_d| < |\min(i)|$, car sinon le chemin étiqueté par w_d finirait par $\min(i)$ et arriverait strictement avant i, ce qui est contradictoire avec le lemme 4. Or $\min(i)$ a aussi une occurrence en k_{d-1} car: $\min(i)$ et $\min(k_{d-1})$ sont comparables et, d'après le corollaire 2, $\min(k_{d-1})$ ne peut pas être suffixe de $\min(i)$ donc $\min(i)$ est suffixe de $\min(k_{d-1})$.

Si bien que $\min(i)$ est un suffixe de w_{d-1} qui apparaît à deux occurrences distinctes dans pref_{d-1} et qui est strictement plus long que $w_d = \text{repet}_p(k_{d-1})$. Contradiction, et $k_d = i$, si bien que $\alpha\sigma$ mène dans le même état que $w_d\sigma$, en s.

(ii) Supposons maintenant que $|\alpha| < |w_d|$, alors $i \leq k_d$ car α est suffixe de w_d. $|\alpha| \geq |\min(k_d)|$ car, comme il existe une transition de k_d par σ vers s, $\min(k_d)\sigma$ est à la fois un facteur

de p et un suffixe de $p\sigma$, et $\alpha\sigma$ est le plus grand de ce type de facteurs. $\min(k_d)$ est un suffixe de α et par le lemme 4 $i \geq k_d$. D'où $i = k_d$, et $\alpha\sigma$ mène dans le même état que $w_d\sigma$, en s.

le chemin $\alpha\sigma$ mène lui aussi en s, et donc $s = S_{p\sigma}(m+1)$. □

A l'aide de ces différents lemmes, on peut maintenant concevoir un algorithme *ajout_lettre* pour transformer Oracle(p) en Oracle($p\sigma$). Il est donné figure 4.

Fonction ajout_lettre(Oracle($p = p_1p_2\ldots p_m$), σ)
1. Création d'un nouvel état $m+1$
2. Création d'une transition m à $m+1$ d'étiquette σ
3. $k \leftarrow S_p(m)$
4. **Tant que** $k > -1$ et qu'il n'existe pas de transition sortante de k par σ **Faire**
5. Création d'une transition de k vers $m+1$ par σ
6. $k \leftarrow S_p(k)$
7. **Fin du Tant que**
8. **Si** $(k = -1)$ **Alors** $s \leftarrow 0$
9. **Sinon** $s \leftarrow$ arrivée de la transition de k par σ.
10. $S_{p\sigma}(m+1) \leftarrow s$
11. **Renvoyer** Oracle($p = p_1p_2\ldots p_m\sigma$)

FIG. 4 – *Ajout d'une lettre σ à Oracle($p = p_1p_2\ldots p_m$) pour obtenir Oracle($p\sigma$)*

Lemme 12 *L'algorithme* ajout-lettre *construit bien Oracle($p = p_1p_2\ldots p_m\sigma$) à partir de Oracle($p = p_1p_2\ldots p_m$) et met à jour la fonction de suppléance du nouvel état $m+1$ de Oracle($p\sigma$)*

Preuve. On parcours le chemin suffixe de p conformément au lemme 9. On s'arrête en accord avec le lemme 10 et on met à jour la suppléance de l'état $m+1$ en fonction du lemme 11. □

L'algorithme *on-line* complet de construction de Oracle($p = p_1p_2\ldots p_m$) consiste maintenant juste à ajouter les lettre p_i une à une de la gauche vers la droite, il est donné figure 5.

Oracle-on-line($p = p_1p_2\ldots p_m$)
1. Créer Oracle(ϵ) avec:
2. un seul état 0
3. $S_\epsilon(0) \leftarrow -1$
4. **Pour** $i \leftarrow 1$ à m **Faire**
5. Oracle($p = p_1p_2\ldots p_i$) $\leftarrow$ `ajout_lettre`(Oracle($p = p_1p_2\ldots p_{i-1}$),p_i)
6. **Fin du Pour**

FIG. 5 – *Algorithme* on-line *de construction de Oracle($p = p_1p_2\ldots p_m$).*

Théorème 1 *L'algorithme* Oracle-on-line*($p = p_1p_2\ldots p_m$) construit Oracle(p).*

Preuve. Par récurrence sur le mot p en utilisant le lemme 12. □

Théorème 2 *La complexité de* Oracle-on-line$(p = p_1p_2 \dots p_m)$ *est* $O(m)$ *en temps et en espace.*

*Preuve.*L'algorithme est en $O(m)$ en espace. En effet, toutes les transitions que crée l'algorithme sont des transitions de Oracle(p). On crée exactement $m+1$ états et à chacun on associe une fonction de suppléance qui peut être codée en espace constant. On est donc linéaire en espace.

L'algorithme est en $O(m)$ en temps. Comme on ne crée que des états et des transitions nécessaires à l'automate que que celui-ci est linéaire, le seul point à vérifier est que le nombre de retours arrière dus aux sauts de suppléance (lignes 4-6 sur la figure 4) est linéaire.

A chaque étape i de la construction (lignes 4-5 de la figure 5), c'est-à-dire à chaque ajout de la lettre p_i, le nombre de retours arrière dus aux sauts de suppléance est borné par la longueur $k_i = |\text{repet}_p(i-1)|$. On note r_i le nombre de sauts de suppléance pour terminer l'étape i. Lors du passage de l'étape i à l'étape $i+1$, on a $k_{i+1} \leq k_i - r_i + 2$, et $r_i \leq k_i - k_{i+1} + 2$. La somme $\sum_{i=1}^{n} r_i$ est donc bornée par $2n$, et l'algorithme est linéaire. □

Remarque Les constantes intervenant dans la majoration asymptotique de la complexité de la construction de l'oracle dépendent de l'implémentation et peuvent dans certains cas faire intervenir la taille de l'alphabet Σ. Si l'on choisit d'implémenter les transitions de telles façon qu'elles soient accessibles en $O(1)$ (représentation sous forme de tables), alors la complexité est $O(m)$ en temps et $O(|\Sigma|.m)$ en espace. Si les transitions sont accessible en $O(log|\Sigma|)$ (représentation sous forme d'arbres de recherche), la complexité est alors $O(log|\Sigma|.m)$ en temps et $O(m)$ en espace.

Exemple La construction *on-line* de Oracle(*abbbaab*) est donné figure 6.

3 Oracle des suffixes

Les liens entre l'automate des suffixes et notre oracle des facteurs amène à une extension directe de l'oracle: il est possible de marquer des états terminaux sur l'oracle des facteurs, comme sur l'automate des suffixes, pour reconnaitre des suffixes de p. Cette extension va permettre d'utiliser certaines propriétés des automates des suffixes. On appelle cette nouvelle structure *oracle des suffixes*, et on la note SOracle(p).

Définition 2 *Un état q de l'oracle des suffixes est* terminal *si et seulement s'il existe un chemin étiquetté par un suffixe de p qui mène de l'état initial à q.*

L'algorithme haut-niveau de construction de l'oracle des facteurs (voir 1) n'est pas facilement modifiable pour construire l'oracle des suffixes, car il ne permet pas de détecter d'éventuels états terminaux. Par contre, l'algorithme de construction séquentielle le permet, grâce à la fonction suffixe. C'est l'objet du lemme suivant. On rappelle que pour un Oracle($p = p_1p_2 \dots p_m$), $k_0 = m$, $k_i = S_p(k_{i-1})$ pour $i >= 1$, que la suite des k_i est finie, strictement décroissante, et se termine en l'état 0, et que l'on note $CS_p = \{k_0 = m, k_1 \dots, k_t = 0\}$ le chemin suffixe de p dans Oracle(p).

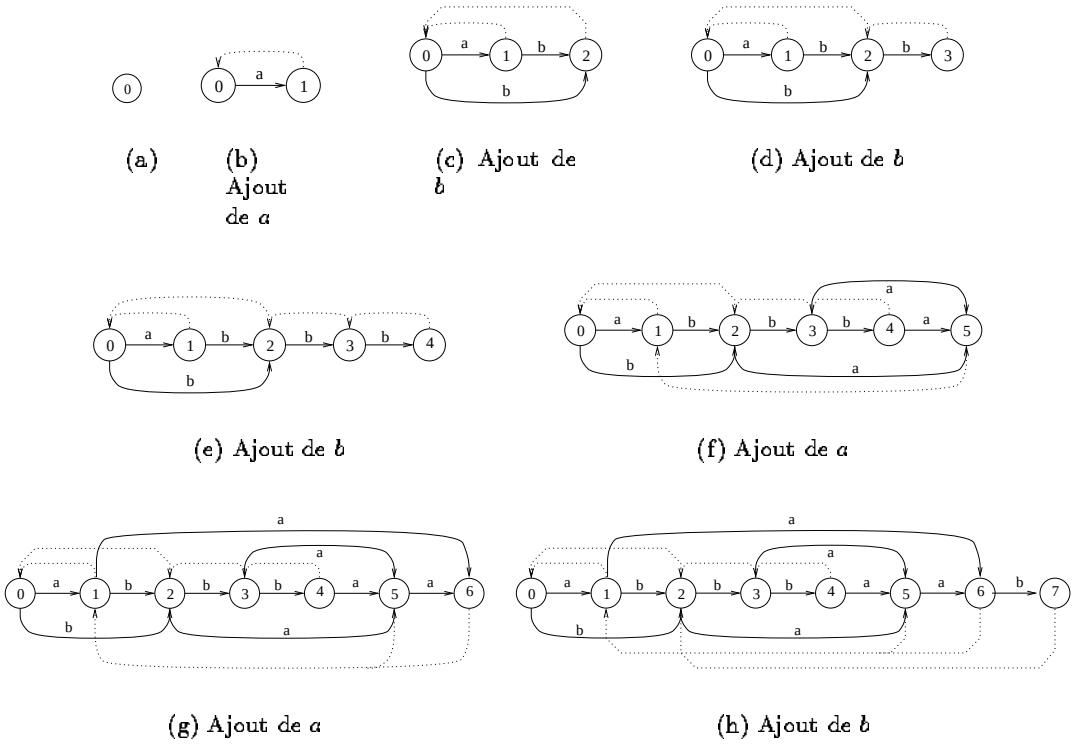


FIG. 6 – *Exemple de construction* on-line *de Oracle*(abbaba). *Les flèches en pointillés représentent la fonction de suppléance.*

Lemme 13 *Les états terminaux de SOracle*(p) *sont les états de Oracle*(p) *qui appartiennent au chemin suffixe* CS_p.

Preuve.

(i) Si $k \in CS_p$, alors k est un état terminal de SOracle(p). Le corollaire 4 indique que, si on note $w_i = repet_p(k_{i-1})$ pour $1 \leq i \leq t$, et $w_0 = p$, w_l est un suffixe de tous les w_i, $0 \leq i < l \leq t$, donc w_l est un suffixe du mot entier p. Comme, d'après la définition des k_i, w_i mène à k_i, $0 \leq i \leq t$, les états k_i sont atteignables par un suffixe de p, donc terminaux.

(ii) Si q est un état de Oracle(p) tel qu'il existe un chemin de l'état initial 0 à q qui a pour étiquette un suffixe s de p, alors $q \in CS_p$. Si $q = m$, $q = k_0$ et la propriété est vrai. Si $q = 0$, la propriété est vrai aussi. On suppose maintenant que $0 < q < m$. Comme s est un suffixe de p, il existe un i tel que s est un suffixe propre de w_{i-1} (qui mène à k_{i-1}) et tel que w_i (qui mène à k_i) est un suffixe propre de s. D'après le lemme 5 on a $k_i \leq q \leq k_{i-1}$.

Supposons que $k_i < q < k_{i-1}$ et montrons une contradiction. On s'intéresse à $v = \min(q)$.

On élimine le cas où $w_i = \epsilon$ et $k_i = 0$. En effet, si $k_i = 0$, alors le chemin de label s qui se termine dans un état $q \neq 0$ finit par $v = \min(q)$, qui n'est pas le mot vide. Comme s

est suffixe propre de w_{i-1}, v est lui aussi suffixe de w_{i-1} et $k_{i-1} \in \text{endpos}_p(v)$. Comme de plus $q < k_{i-1}$, v est suffixe de w_{i-1} et apparait une fois en q (d'après le lemme 1) strictement avant k_i, on est en contradiction avec le fait que $c = w_i = \text{repet}_p(w_{i-1})$ par définition de CS_p.

Revenons au cas où $|w_i| > 0$.

On a tout d'abord $|v| < |w_i|$. En effet, supposons à contrario que $|v| \geq |w_i|$. On ne peut avoir $|v| = |w_i|$, car dans ce cas $v = w_i$ (ils sont tous les deux suffixes de s) et $q = \text{poccur}(v, p)$. Comme on a supposé que $k_i < q$, on entre en contradiction avec le lemme 4. On suppose donc que $|v| > |w_i|$, mais dans ce cas, comme (1) $q \in \text{endpos}_p(v)$ (lemme 1) (2) $k_{i-1} \in \text{endpos}_p(v)$ (lemme 2: v est suffixe de s qui est suffixe de w_i) (3) $q < k_i$ par hypothèse, v est suffixe de w_{i-1}, apparait strictement avant k_{i-1}, et est plus grand que w_i, ce qui contredit la définiton de $w_i = \text{repet}_p(w_{i-1})$.

Comme $|v| < |w_i|$ et que w_i et v (lemme 2) sont suffixes de s, v est suffixe propre de w_i. Comme $v = \min(q)$, le lemme 4 contredit le fait que $k_i < q$.

□

Pour transformer l'oracle des facteur de p en oracle des suffixes, il faut donc parcourir le chemin suffixe du dernier état crée lors de la construction séquentielle de Oracle(p) en marquant chaque état de ce chemin comme terminal. Le pseudo-code de la construction de l'oracle des suffixes (en utilisant l'algorithme séquentiel de construction de l'oracle des facteurs) est donné à la figure 7.

Oracle-des-suffixes($p = p_1 p_2 \ldots p_m$)
1. Oracle-on-line(p)
2. $t \leftarrow m$
3. **Tant que** $S_p(t) \neq -1$ **Faire**:
4. marquer t comme état terminal
5. $t \leftarrow S_p(t)$
6. **Fin du Tant que**

FIG. 7 – *Algorithme de construction de l'oracle des suffixes SOracle($p = p_1 p_2 \ldots p_m$).*

Nous donnons pour exemple SOracle(*abbbaab*) à la figure 8.

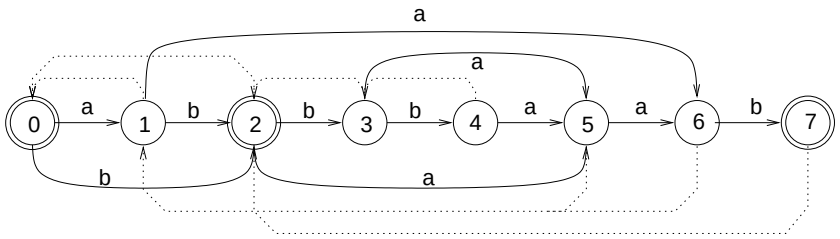


FIG. 8 – *Exemple d'oracle des suffixes. Les états entourés par un double cercle sont terminaux. La fonction de suppléance est représentée par les arcs en pointillés.*

Nous utilisons principalement la structure d'oracle des facteurs et non celle d'oracle des suffixes pour la raison suivante. La puissance de la structure d'oracle des facteurs repose sur sa simplicité de construction (qui devient légèrement plus compliquée pour l'oracle des suffixes), mais surtout sur le peu de mémoire qu'il est nécessaire d'utiliser en pratique pour l'implémenter. La base de cette économie de mémoire est la possibilité de ne pas coder les états de l'automate (en utilisant chaque position du mot comme état), et de ne coder que les arcs extérieurs. Ceci est plus dur à réaliser pour l'oracle des suffixes, car il faut un moyen de marquer si une position est terminale ou pas, ce qui, d'une part complique l'implémentation, mais aussi ralentit le test du terminal si on veut garder une implémentation fine.

Remarque Dans certains cas, l'oracle des suffixes peut être exactement l'automate des suffixes et peut ainsi reconnaître exactement les suffixes. Sur l'alphabet binaire $\Sigma = \{0, 1\}$, c'est notamment le cas pour les mots de Fibonacci. L'étude plus générale des mots binaires pour lesquels l'oracle des suffixes est exactement l'automate des suffixes est liée à l'étude des mots sturmiens. Le lecteur intéressé est invité à se référer à [2] pour une définition et pour les principales propriétés des mots sturmiens.

Les liens entre l'oracle des facteurs (et l'oracle des suffixes) et l'automate des suffixes amènent à penser qu'il doit être possible de remplacer l'automate des suffixes par un oracle dans certaines de ses applications. C'est le cas dans un champ d'applications assez vaste de l'automate des suffixes, la recherche de mots.

4 Recherche de mots

L'Oracle(p) peut être utilisé, au même titre que l'automate des suffixes, pour la recherche des occurrences d'un mot $p = p_1 p_2 \dots p_m$ dans un texte $T = t_1 t_2 \dots t_n$, tous les deux pris sur un alphabet Σ.

L'automate des suffixes est utilisé dans [8, 7] pour obtenir un algorithme optimal en moyenne (le BDM - *Backward Dawg matching*). Sa complexité moyenne est en $O(n \log_{|\Sigma|}(m)/m)$ sous un modèle bernouillien de probabilité où toutes les lettres sont équiprobables, ce qui vérifie la borne minimale établie par Yao [11] lorsque $n > 2m$.

L'algorithme BDM déplace une fenêtre de taille m sur le texte. Pour chaque nouvelle position de cette fenêtre, l'automate des suffixes de p^r (image miroir de p) est utilisé pour rechercher un facteur de p de droite à gauche de la fenêtre.

L'idée fondamentale du BDM est que si cette recherche arrière échoue sur une lettre σ après avoir lu un mot u, alors σu n'est pas un facteur de p et le déplacement de la fenêtre après σ est sûr. Cette idée est affinée ensuite dans le BDM en utilisant certaines propriétés de l'automate des suffixes.

Cependant, cette idée de base est suffisante pour obtenir un algorithme efficace de recherche de mot. Le plus étonnant est que la reconnaissance stricte des facteurs (que permet l'automate des suffixes et l'automate des facteurs) n'est pas nécessaire. Pour que l'algorithme fonctionne, il suffit de savoir que σu n'est pas un facteur de p. L'oracle peut donc être utilisé à la place de l'automate des suffixes, comme l'illustre la figure 9. Nous appelons cet algorithme Backward Oracle Matching (BOM). Nous conjecturons, d'après les résultats expérimentaux,

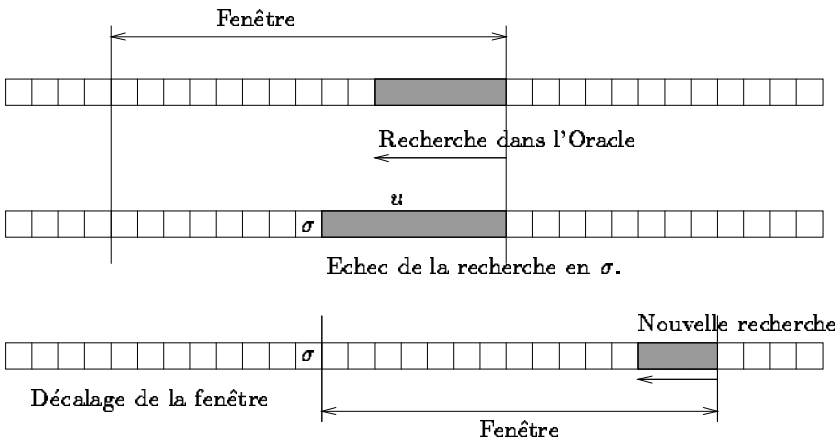


FIG. 9 – *Déplacement de la fenêtre de recherche après l'échec de la recherche par Oracle(p). Le mot σu n'est pas un facteur de p.*

que cet algorithme reste optimal en moyenne. Plus formellement, nous donnons le pseudo-code de l'algorithme BOM figure 4 et une preuve de sa validité au lemme 14.

Lemme 14 *L'algorithme BOM marque toutes les occurrences de p dans T et elles seules.*

Preuve.

- BOM ne marque que des occurrences valides, car le seul mot de longueur m reconnu par Oracle(p^r) est p^r lui-même.
- BOM trouve toutes les occurrences de p. En effet, supposons à contrario qu'il existe une occurrence de p telle qu'aucune fenêtre ne lui corresponde (si une telle fenêtre existait, l'occurrence serait reconnue). Comme le déplacement d'une fenêtre est au maximum de m, on a nécessairement la situation de la figure 11 (avec u qui peut être vide) et Fenêtre 1 et Fenêtre 2 consécutives dans l'algorithme. L'échec de la reconnaissance d'un facteur aurait donc dû se produire en σ, ce qui n'est pas possible car σu est un facteur de p.

□

La complexité de BOM dans le pire des cas est $O(nm)$. Cependant, en moyenne, nous faisons la conjecture suivante (d'après les résultats expérimentaux de la partie 4.3) :

Conjecture 1 *L'algorithme BOM, sous un modèle d'équiprobabilité et d'indépendance des lettres, a une complexité moyenne en $O(n \log_{|\Sigma|}(m)/m)$.*

4.1 Approche utilisant l'oracle des suffixes

Le fait d'utiliser l'oracle des suffixes au lieu de l'oracle des facteurs permet un raffinement de l'approche précédente. Ce raffinement vient directement de l'usage qui est fait de l'automate des suffixes dans le BDM. Lors de la phase de recherche arrière dans l'oracle des suffixes, si un état terminal (qui ne correspond pas au mot tout entier) est traversé, la position dans la fenêtre est sauvegardée dans une variable *last*. Ceci permet de donner un majorant du plus

```
BOM(p = p_1 p_2 ... p_m, T = t_1 t_2 ... t_n)
1.  Préparation
2.      Construction de l'Oracle de p^r
3.  Recherche
4.      pos ← 0
5.      Tant que (pos <= n − m) faire
6.          state ← état initial de Oracle(p^r)
7.          j ← m
8.          Tant que state existe faire
9.              state ← état image par T[pos + j] dans Oracle(p^r)
10.             j ← j − 1
11.         Fin du Tant que
12.         Si j = 0 faire
13.             marquer une occurrence à  pos + 1
14.             j ← 1
15.         Fin du Si
16.         pos ← pos + j
17.     Fin du Tant que
```

FIG. 10 – *Pseudo-code de l'algorithme* **BOM**.

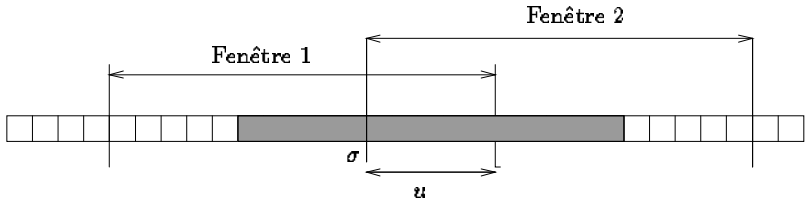


FIG. 11 – *Situation impossible de l'algorithme BOM lors de la phase de recherche (voir preuve du lemme 14).*

grand facteur lu qui est aussi un suffixe de p^r, donc un préfixe du mot p. En sauvegardant le dernier, on sauvegarde un majorant du plus grand, et on peut alors déplacer la fenêtre de recherche (que le mot soit ou ne soit pas dans le texte à la position courante de la recherche) jusqu'à *last*, et recommencer la recherche arrière. La figure 12 illustre cette amélioration.

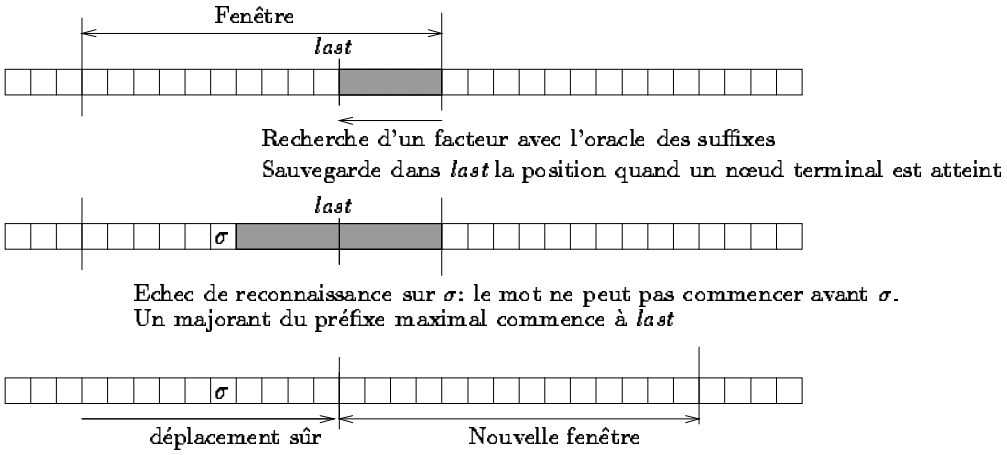


FIG. 12 – *Recherche avec l'oracle des suffixes. Le marquage des états terminaux permet une amélioration de la recherche qui est faite avec l'oracle des facteurs.*

On appelle ce nouvel algorithme **BSOM** (pour *Backward Suffix Oracle Matching*). Cet algorithme est toujours de complexité $O(nm)$ dans le pire des cas.

4.2 Un algorithme de recherche linéaire dans le pire des cas

Les deux algorithmes de recherche précédents, bien que très efficaces en pratique (ils sont même supposés optimaux en moyenne) sont de complexité $O(nm)$ dans le pire des cas.

Pour l'algorithme BDM utilisant un automate des suffixes, il existe plusieurs techniques différentes pour le transformer en un algorithme linéaire dans le pire des cas [8].

Une de ces techniques peut être utilisée pour rendre nos algorithmes linéaires dans le pire des cas, en introduisant l'algorithme de Knuth-Morris-Pratt (KMP) [9] pour effectuer une lecture avant de certains caractères du texte.

Pour expliquer l'utilisation combinée de l'algorithme KMP et de la recherche de mots dans l'oracle (des facteurs ou des suffixes), on se place à la position courante avant la recherche par oracle: on a déjà lu par l'algorithme KMP un préfixe v du mot comme préfixe de la fenêtre courante, et on commence alors la recherche par l'oracle de droite à gauche. La fin du mot v dans la fenêtre courante est appelée *position critique* et notée *Critpos*. La figure 13 schématise cette position courante.

On utilise l'algorithme de recherche par oracle de droite à gauche à partir du bord droit de la fenêtre. Deux cas se présentent, suivant que l' on a atteint la position critique ou non.

1. La position critique n'est pas atteinte. On a obtenu un échec de reconnaissance de facteur sur un caractère σ, comme dans le schéma général (figure 9). On décale alors la fenêtre sur la droite jusqu'à ce que le bord gauche ait passé le caractère σ. On relit ensuite les caractères par l'algorithme KMP (au moins jusqu'à la position critique), qui s'arrête lorsque le plus grand préfixe de p reconnu est assez petit, inférieur à αm, avec

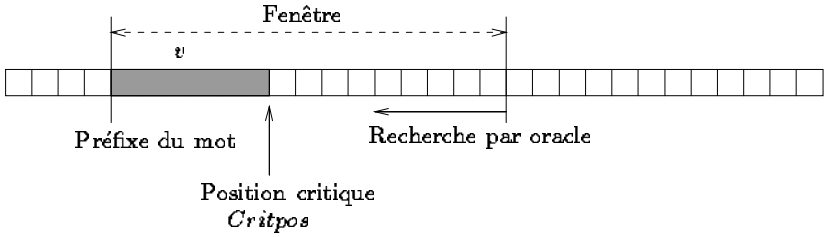


FIG. 13 – *Position courante de l'algorithme linéaire utilisant l'algorithme KMP combiné à la recherche de mots par l'oracle (des facteurs ou des suffixes).*

$0 < \alpha < 1$. La valeur de α est discutée lors des tests expérimentaux (voir la section 4.3). La figure 14 schématise cette étape.

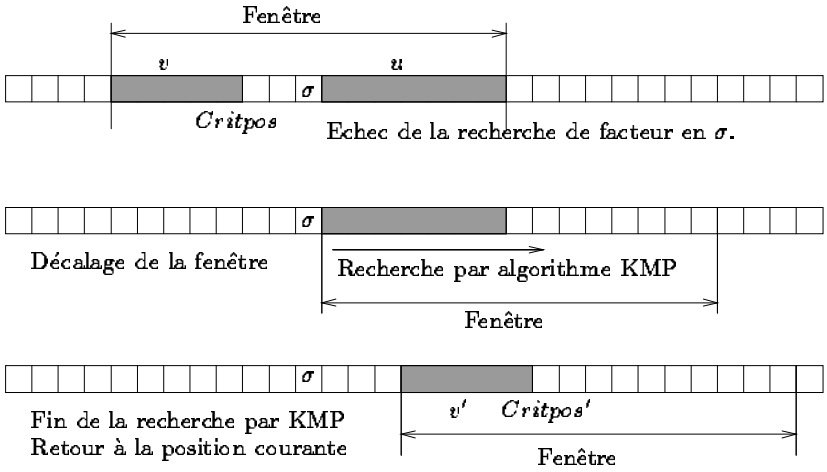


FIG. 14 – *Premier cas: la position critique n'est pas atteinte.*

2. La position critique est atteinte. On continue de lire les caractères par l'algorithme KMP à partir de l'état auquel on était arrivé avant de s'arrêter dans la position critique *Critpos*. On lit au moins tous les caractères jusqu'à la fin de la fenêtre et on s'arrête lorsque le préfixe reconnu est assez petit (comme dans le cas précédent). On se trouve de nouveau dans la position courante. La figure 15 schématise cette étape.

Cet algorithme peut être utilisé avec la recherche arrière de l'oracle des facteurs aussi bien que celle de l'oracle des suffixe (en gardant le dernier état terminal traversé). On appelle ces deux algorithmes **Turbo-BOM** et **Turbo-BSOM** . En ce qui concerne leurs complexités dans le pire des cas, on a le théorème suivant.

Théorème 3 *Les deux algorithmes Turbo-BOM et Turbo-BSOM sont*

(i) linéaires en nombre d'inspections de caractères du texte. Ce nombre d'inspections est inférieur à $2n$.

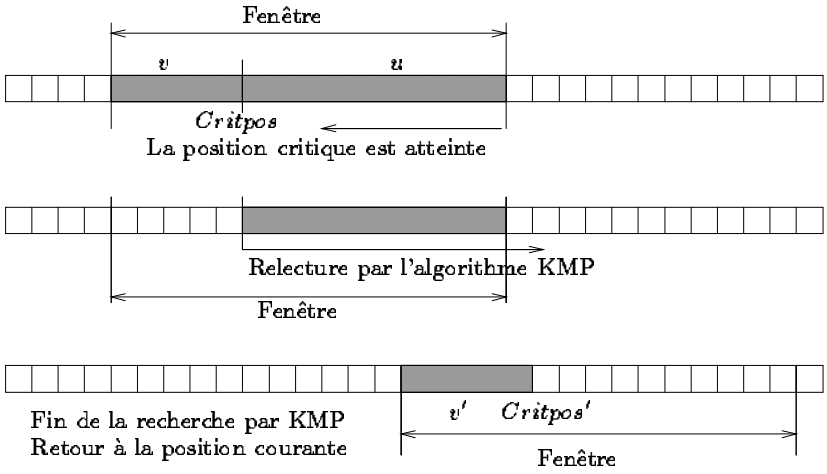


FIG. 15 – *Deuxième cas: la position critique est atteinte.*

(ii) linéaires en nombre de comparaisons de caractères. Ce nombre de comparaisons est inférieur à $2n$ *si les transitions de l'oracle sont accessibles en* $O(1)$*, et inférieur à* $2n + n \log \Sigma$ *si les transitions de l'oracle sont accessibles en* $\log \Sigma$.

Preuve.

(i) Chaque caractère du texte est lu au plus deux fois, une fois lors de la recherche arrière par l'oracle, une fois lors de la lecture par l'algorithme KMP, d'où le résultat.

(ii) Cette complexité provient directement du fait que le nombre de comparaisons totales de l'algorithmes KMP est inférieur ou égal à $2n$. Si les transitions de l'oracle sont accessibles en $O(1)$, la recherche arrière par l'oracle ne nécessite aucune comparaison (simplement des inspections au texte), et le nombre de comparaisons est borné par celui du KMP, c'est-à-dire $2n$. Si, par contre, les transitions de l'oracle sont accessibles en $\log \Sigma$ (par exemple sous forme d'arbre binaires de recherches), le nombre de comparaisons effectuées lors de la recherche arrière par l'oracle sont bornée par $n \log \Sigma$, et le nombre total par $2n + n \log \Sigma$.

□

4.3 Résultats expérimentaux

Nous présentons dans cette section des résultats expérimentaux en temps pour nos algorithmes de recherche de mots en les comparant aux plus rapides. Plus précisemment, nous comparons les algorithmes suivants.

- **Sunday**: l'algorithme de Sunday [10] est souvent considéré comme le plus rapide en pratique,
- **BM**: l'algorithme de Boyer-Moore [5],
- **BDM**: le Backward Dawg Matching classique avec un automate des suffixes [7],

- **Suff**: le Backward Dawg Matching avec un automate des suffixes, mais sans tester les états terminaux, ce qui revient à utiliser l'approche de base avec un automate des facteurs[1],
- **BOM**: le Backward Oracle Matching avec l'oracle des facteurs,
- **BSOM**: le Backward Oracle Matching avec l'oracle des suffixes (en testant les états terminaux),
- **Turbo-BOM**: l'algorithme linéaire utilisant le BOM et le KMP, avec $\alpha = 1/2$.

Nos expériences de recherches de mots sont faites sur des textes aléatoires de 10Mo, avec une précision de +/- 2% sûre à 95% (ce qui peut demander des milliers d'itérations), pour des alphabets de taille 2, 4, 16 et 32. La machine utilisée est un PC avec un pentium 350Mz équipé du système Linux 2.0.32 . Pour tous nos algorithmes, les transitions de l'automate sont sous forme de table, ce qui permet un branchement en $O(1)$, mais ce qui n'est pas réaliste (surtout pour l'automate des suffixes) dès que l'alphabet devient assez grand (par exemple pour un système de codage des caractères sur 16 bits). L'algorithme de Sunday devient inutilisable tel quel dès que l'alphabet est grand, car il utilise principalement une table de tous les caractères.

Les résultats expérimentaux de recherche de mots sur des textes sont toujours surprenants, car en fait les codes sont petits et le temps d'une opération de comparaison n'est pas beaucoup plus grand que d'incrémenter un indice. C'est la raison pour laquelle, par exemple, le Sunday (lorsqu'il est utilisable) est l'algorithme le plus rapide pour des petits mots. Les déplacements de la fenêtre sont très petits, mais le nombre d'opérations pour déplacer la fenêtre est lui-aussi très faible. C'est aussi la raison pour laquelle le BDM est plus lent que le Suff et le BSOM plus lent que le BOM, alors qu'en théorie les déplacements du BDM et du BSOM sont plus grands. Le temps utilisé pour tester si l'état est terminal est trop important.

Des quatre sous-figures de la figure 16 il ressort que le BOM est aussi rapide (sauf sur un alphabet binaire) que le Suff, qui utilise lui beaucoup plus de mémoire en étant de plus beaucoup plus compliqué.

Il est visiblement inutile (dans le cas de recherches sur des textes de caractères) de marquer et de tester des éventuels états terminaux sur l'automate des suffixes comme sur l'oracle des facteurs.

Le Turbo-BOM est plus lent que tous les autres, mais c'est le seul qui puisse être utilisé en temps réel, et dans ce cas ses performances sont loin d'être mauvaises. Il faut noter ici que nous avons fixé la valeur de α à $1/2$ de façon tout à fait arbitraire. Cependant, d'après les tests que nous avons effectués avec différentes valeurs de α, il s'avère que $\alpha = 1/2$ est le plus souvent la meilleure option, et que les variations pour les temps de recherche avec d'autres valeurs de α (si elles restent dans une fourchette de $(2\log_{|\Sigma|} m)/m$ et $(m - 2\log_{|\Sigma|} m)/m$) ne sont pas très significatives, et en tout cas pas assez significatives pour mériter à elles seules une étude détaillée.

1. L'automate des suffixes sans tenir compte des états terminaux (i.e en considérant tous les états comme terminaux) et l'automate des facteurs reconnaissent le même langage. La différence est que l'automate des facteurs est minimal, donc de taille inférieure ou égale au premier. Cependant, la différence de taille est en pratique peu importante, et en tout cas pas assez importante pour justifier l'implémentation d'un automate des facteurs, ce qui compliquerait et ralentirait la phase de préparation de l'algorithme de recherche de mots.

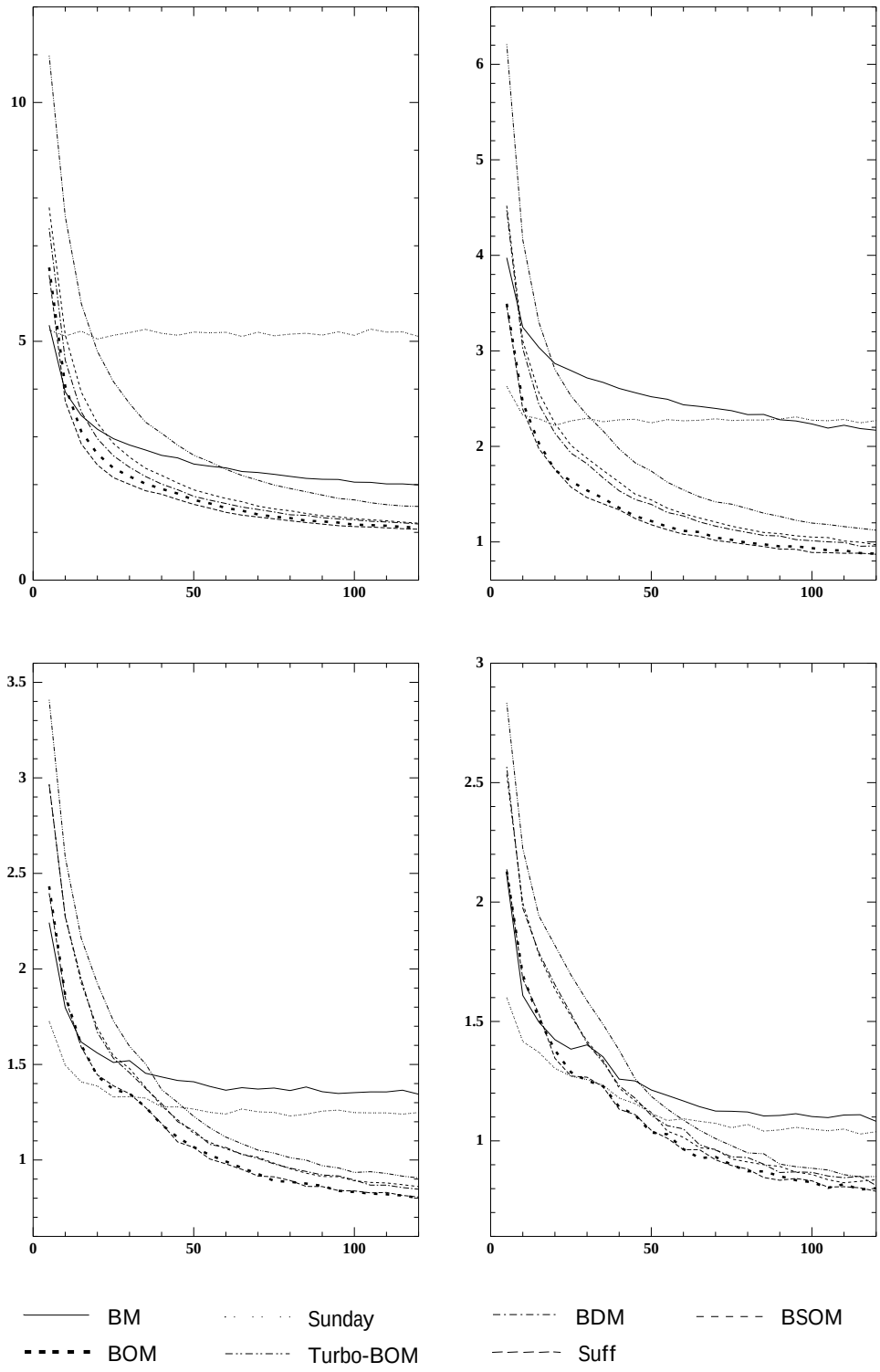


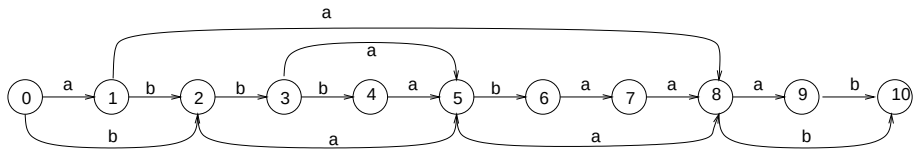
FIG. 16 – *Résultats expérimentaux en temps des différents algorithmes de recherche de mots sur des textes aléatoires de 10 Mo sur des alphabets de tailles 2, 4, 16 et 32. L'axe des abscisses représente la longueur des mots aléatoires recherchés, et l'axe des ordonnées représente le temps de recherche, en centièmes de secondes par Mo.*

5 Conclusions

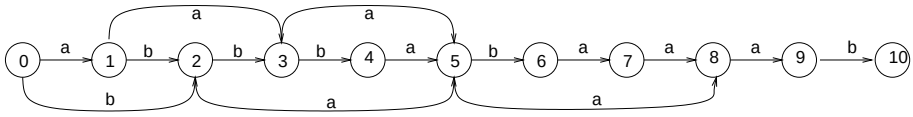
Les deux nouvelles structures que nous avons présentées, l'oracle des facteurs et l'oracle des suffixes, permettent d'obtenir des algorithmes de recherche de mots très rapides en pratique, aussi rapides que ceux qui existaient déjà, mais en utilisant un espace mémoire beaucoup plus faible tout en étant beaucoup plus simples à implémenter. Au vu des résulats expérimentaux, nous conjecturons que ces algorithmes sont optimaux en moyenne (sous un modèle d'indépendance et d'équiprobabitilté des lettres), mais cela reste à prouver.

En ce qui concerne la structure même d'oracle des facteurs, beaucoup de questions restent en suspens. Notamment, il serait utile d'avoir une caractérisation du langage reconnu.

Il serait aussi utile d'avoir une étude du nombre moyen d'arcs externes, pour avoir une idée de l'espace mémoire moyen que nécessite l'algorithme de recherche de mots.



(a) Oracle des facteurs



(b) Automate minimal

FIG. 17 – *Contre exemple à la minimalité en nombre d'arcs de l'oracle des facteurs pour l'ensemble des automates à $m+1$ états qui reconnaissent au moins l'ensemble des facteurs.*

Pour finir, l'oracle des facteurs n'est pas minimal en terme de nombre de transitions externes parmis les automates de $m+1$ états qui reconnaissent au moins les facteurs. Un contre-exemple est donné à la figure 17. Cet automate minimal pourrait sûrement être lui aussi utilisé pour faire de la recherche de mots, à condition qu'il soit constructible en temps linéaire. Cette construction reste un problème ouvert.

Références

[1] R. A. Baeza-Yates. Searching subsequences. *Theor. Comput. Sci.*, 78(2):363–376, 1991.

[2] J. Berstel. Recent results in sturmian words. In *Developments in Language Theory II*, pages 13–24. World Scientific, 1996.

[3] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40(1):31–55, 1985.

[4] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnel. Linear size finite automata for the set of all subwords of a word: an outline of results. *Bull. Eur. Assoc. Theor. Comput. Sci.*, 21:12–20, 1983.

[5] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[6] M. Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.

[7] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[8] A. Czumaj, M. Crochemore, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12:247–267, 1994.

[9] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

[10] D. Sunday. A very fast substring search algorithm. *CACM*, 33(8):132–142, August 1990.

[11] A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.