École Jeunes Chercheurs
27–31 mars 2000 à Caen

———

Une introduction à la recherche de mot

Thierry Lecroq
LIFAR – ABISS
Université de Rouen
lecroq@dir.univ-rouen.fr

1 Définitions et notations *2*

2 Recherche de motif *3*

3 Recherche d'un mot fixé dans un texte *4*

4 Complexité du problème *5*

5 Simulation d'automate *6*
- 5.1 Algorithme naïf *6*
- 5.2 Algorithme de Morris et Pratt *6*
- 5.3 Algorithme de Knuth, Morris et Pratt *9*
- 5.4 Algorithme de Simon *10*
- 5.5 Algorithme de Colussi *12*

6 Algorithmes rapides en pratique *14*
- 6.1 Algorithme de Boyer-Moore *14*
- 6.2 Algorithme de Turbo-BM *16*
- 6.3 Algorithme d'Apostolico-Giancarlo *17*

Bibliographie *21*

1 Définitions et notations

On considère un ensemble A fini de lettres appelé ***alphabet***.
Un ***mot*** est une suite de lettres de A.
Un mot w de m lettres s'écrit $w = w[0 \mathinner{.\,.} m-1]$.
La ***longueur*** de w se note $|w|$.
On note ε le mot vide de longueur 0.
Pour un mot w de longueur m, une ***position*** i sur x est un entier compris entre 0 et $m-1$. La lettre à la position i est $w[i]$.
Un mot u est un ***préfixe*** d'un mot w si w peut s'écrire $w = uv$. Un mot v est un ***suffixe*** d'un mot w si w peut s'écrire $w = uv$. Un mot z est un ***facteur*** d'un mot w si w peut s'écrire $w = uzv$.
Un ***bord*** d'un mot w est un facteur qui est à la fois préfixe et suffixe. ***Le bord*** d'un mot est le plus long de ses bords (il peut être le mot vide).
Un entier p est une ***période*** d'un mot w si pour tout i tel que $0 \leq i < m-p$, $w[i] = w[i+p]$. ***La période*** d'un mot est la plus petite de ses périodes non nulles (elle peut être égale à $|w|$). Un mot est dit ***périodique*** si sa période est inférieure à la moitié de sa longueur.
Il y a une relation évidente entre les bords d'un mot et ses périodes.
Exemple avec `abaabaabaab`

bords	périodes	|bords|+périodes
`abaabaabaab`	0	11
`abaabaab`	3	11
`abaab`	6	11
`ab`	9	11
ε	11	11

2 Recherche de motif

Le problème de la recherche de motif consiste à localiser une ou plus généralement toutes les occurrences d'un motif dans un texte. Un motif, peut être un mot, un ensemble fini de mots ou un ensemble infini de mots exprimé sous la forme d'une expression rationnelle. Ce problème important apparaît dans beaucoup de domaines de l'informatique notamment dans les traitements de textes et en analyse lexicale. En biologie cela intervient dans l'analyse de l'ADN et des séquences de protéines. Pour un certain nombre de programmes les techniques utilisées en recherche de motif constituent un gros pourcentage du travail effectué. Améliorer ces techniques augmente considérablement le rendement de ces programmes.

On s'intéressera ici à la recherche d'un mot dans un texte. Les techniques utilisées dans ce type de problème servent généralement de bases aux autres types de recherche.

Le problème de la recherche d'un mot x de longueur m dans un texte y de longueur n possède deux variantes. Lorsque le texte y est connu à l'avance, un prétraitement est alors autorisé sur celui-ci, pour construire une structure d'index en temps et espace $O(n)$ et la localisation de toutes les occurrences de x dans y peut alors s'effectuer en temps et espace $O(m)$. Lorsque le mot x est connu à l'avance, un prétraitement est alors autorisé sur celui-ci, pour construire une structure de données en temps et espace $O(m)$ et la localisation de toutes les occurrences de x dans y peut alors s'effectuer en temps $O(n)$. On s'intéressera ici au second cas.

3 Recherche d'un mot fixé dans un texte

Les algorithmes de recherche d'un mot fixe dans un texte utilisent un mécanisme dit « de fenêtre glissante ». Une fenêtre de la même longueur que le mot x est utilisée pour accéder au texte y. L'extrémité gauche de cette fenêtre est d'abord alignée avec l'extrémité gauche du texte. Un travail spécifique appelé ***tentative*** permet de déterminer si le contenu de la fenêtre est identique à celui du mot. Ensuite l'algorithme ***décale*** la fenêtre vers la droite et le même processus est appliqué jusqu'à ce que l'extrémité droite de la fenêtre dépasse l'extrémité droite du texte. On associe à chaque tentative la position droite de la fenêtre : lors d'une tentative j la fenêtre est positionnée sur le facteur $y[j-m+1\,..\,j]$ du texte. Les algorithmes de recherche de mot diffèrent par les méthodes employées lors des tentatives et des décalages. Un « bon » algorithme de recherche de mot cherche à minimiser le travail effectué lors de chaque tentative et à maximiser la longueur des décalages.

On peut classer les algorithmes de recherche exacte d'un mot en plusieurs catégories :

- Les algorithmes qui simulent le fonctionnement d'un automate. Ils ont une bonne complexité théorique. Ils examinent généralement les lettres de la fenêtre de la gauche vers la droite lors de chaque tentative.
- Les algorithmes rapides en pratique. Ils examinent généralement les lettres de la fenêtre de la droite vers la gauche lors de chaque tentative.
- Les algorithmes optimaux dans le compromis espace/temps. Ils partitionnent les lettres de la fenêtre en deux ensembles et examinent les lettres d'un ensemble dans un sens et les lettres de l'autre dans l'autre sens.
- Les algorithmes qui simulent le comportement d'un automate en représentant l'état de la recherche par un vecteur binaire. Ces algorithmes sont très efficaces pour la recherche de mots courts car alors les mots machines peuvent être utilisés pour représenter l'état de la recherche.

4 Complexité du problème

Nous allons maintenant présenter quelques résultats théoriques.

Théorème 4.1 (Galil et Seiferas, 1983)
La recherche peut être faite en temps $O(n)$ et en espace $O(1)$.

Théorème 4.2 (Yao, 1979)
La recherche peut être faite en temps moyen optimal $O(\frac{\log m}{m} \times n)$.

Théorème 4.3 (Cole et Hariharan, 1997)
Le nombre maximal de comparaisons, entre des lettres du mot et des lettres du texte, effectuées durant la recherche est supérieur ou égal à $n + \frac{9}{4m}(n-m)$. La recherche peut être faite en au plus $n + \frac{8}{3(m+1)}(n-m)$ comparaisons de lettres.

5 Simulation d'automate

5.1 Algorithme naïf

L'algorithme naïf est caractérisé par le fait qu'il ne mémorise aucune information d'une tentative à l'autre et que la longueur des décalages qu'il effectue après chaque tentative est exactement égale à 1.

Un exemple d'exécution de l'algorithme naïf est donné figure 5.1. Cet exemple sera repris tout au long de ce chapitre.

La complexité temporelle de sa phase de recherche est quadratique : $O(m \times n)$ mais il ne nécessite aucun prétraitement et un espace supplémentaire (par rapport au mot et au texte) constant. Le nombre maximal de comparaisons est $(n - m + 1) \times m$ alors que le nombre moyen de comparaisons est de l'ordre de $2n$ (sur un alphabet à deux lettres avec des conditions d'équiprobabilité et d'indépendance des lettres).

5.2 Algorithme de Morris et Pratt

Une simple remarque suggère comment améliorer l'algorithme naïf : ce dernier oublie les lettres reconnues pendant la tentative courante pour calculer la longueur du décalage à effectuer. Au cours d'une tentative $j + m - 1$ on reconnaît un préfixe de longueur i de x tel que :

$$x[0 \mathinner{..} i-1] = y[j \mathinner{..} i+j-1] \text{ et } x[i] \neq y[i+j]$$

Lors de la tentative suivante on peut comparer directement la lettre $y[i+j]$ avec la lettre $x[b]$ tel que $x[0, b-1]$ est le plus long bord propre de $x[0, i-1]$. En effet si $x[0 \mathinner{..} b-1]$ est un bord de $x[0 \mathinner{..} i-1]$ cela signifie que $x[0 \mathinner{..} b-1] = y[i+j-b \mathinner{..} i+j-1]$. De plus si $x[0 \mathinner{..} b-1]$ est le plus long bord de $x[0 \mathinner{..} i-1]$ cela signifie que l'on ne manquera pas une occurrence de x dans y. Autrement dit on décale de la période du préfixe reconnu.

Pour bien visualiser comment il convient d'effectuer ce décalage, il suffit de superposer deux occurrences du préfixe $x[0 \mathinner{..} i-1]$ du mot x et

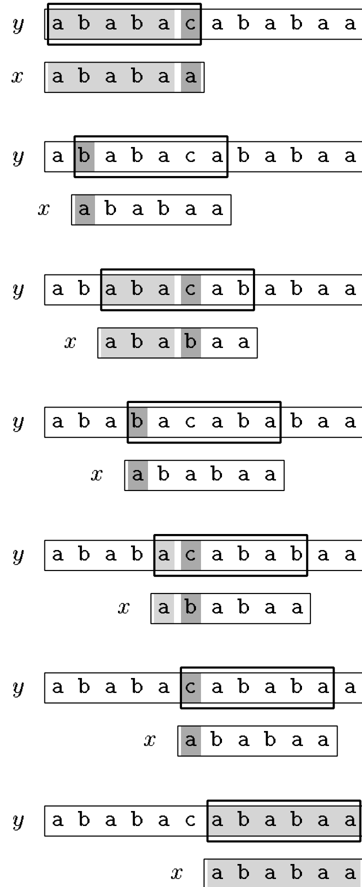


Figure 5.1 L'algorithme naïf effectue 21 comparaisons en 7 tentatives lors de la recherche de ababaa dans ababacababaa. Les lettres sur fond gris clair représente des comparaisons positives (entre lettres identiques) alors que les lettres sur fond gris foncé représente des comparaisons négatives.

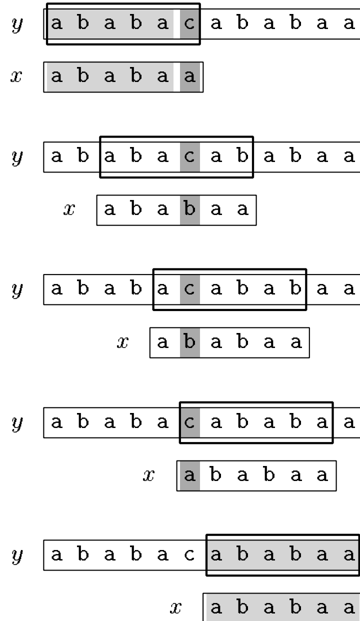


Figure 5.2 L'algorithme de Morris et Pratt effectue 15 comparaisons en 5 tentatives lors de la recherche de ababaa dans ababacababaa.

de faire coulisser l'une de ces occurrences vers la droite jusqu'à ce que les lettres des deux occurrences du préfixe se correspondent. La longueur du décalage est alors donné par le nombre de lettres qui dépassent.

Un exemple d'exécution de l'algorithme de Morris et Pratt est donné figure 5.2.

Pour calculer ces décalages il faut disposer pour chaque préfixe du mot x de la longueur de son plus long bord. Ces informations peuvent être calculées et mémorisées, dans une phase de prétraitement, en temps et espace $O(m)$ et ne dépendent que du mot x.

Théorème 5.1

L'algorithme de Morris et Pratt localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(n)$. Il effectue au plus $2n$ comparaisons entre des lettres du mot et des lettres du texte. Sa phase de prétraitement s'exécute en temps et espace $O(m)$. Il nécessite un espace supplémentaire $O(m)$.

Le ***délai*** est le nombre maximum de comparaisons pour chaque lettre du texte.

Théorème 5.2

Le délai lors de la recherche d'un mot de longueur m dans un texte à l'aide de l'algorithme de Morris et Pratt est au plus m.

5.3 Algorithme de Knuth, Morris et Pratt

Pour passer de l'algorithme naïf à l'algorithme de Morris et Pratt, il a suffit de tenir compte, au moment du décalage, des lettres reconnues lors de la tentative courante. On peut toutefois améliorer l'algorithme de Morris et Pratt en faisant la remarque suivante : toujours dans le cas où on a reconnu un préfixe du mot x de longueur i dans le texte y tel que :

$$x[0\,..\,i-1] = y[j\,..\,i+j-1] \text{ et } x[i] \neq y[i+j]$$

il ne sert à rien de poursuivre les comparaisons avec $x[b]$ et $y[i+j]$ (avec $x[0\,..\,b-1]$ le plus long bord propre de $x[0\,..\,i-1]$) dans le cas où $x[b] = x[i]$ car dans ce cas on sait déjà que $x[b] \neq y[i+j]$.

Il faut poursuivre les comparaisons avec les lettres $y[i+j]$ et $x[b']$ tel que $x[0\,..\,b'-1]$ soit le plus long bord propre de $x[0\,..\,i-1]$ tel que $x[b'] \neq x[j]$.

Cette fois pour imaginer comment effectuer le bon décalage on superpose deux occurrences du préfixe $x[0\,..\,i]$ et on fait coulisser l'une de deux occurrences jusqu'à ce que les lettres de $x[0\,..\,i-1]$ de l'occurrence immobile soient alignées avec des lettres identiques et que $x[i]$ soit alignée avec une lettre différente.

Un exemple d'exécution de l'algorithme de Knuth, Morris et Pratt est donné figure 5.3.

Pour calculer ces décalages il faut disposer pour chaque préfixe du mot x de la longueur de son plus long bord suivi par une lettre différente de celle qui suit le préfixe. Ces informations peuvent être calculées et mémorisées, dans une phase de prétraitement, en temps et espace $O(m)$ et ne dépendent que du mot x.

Théorème 5.3

L'algorithme de Knuth, Morris et Pratt localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(n)$. Il effectue au plus $2n$ comparaisons entre des lettres du mot et des lettres du texte. Sa phase de prétraitement s'exécute en temps et espace $O(m)$. Il nécessite un espace supplémentaire $O(m)$.

Théorème 5.4

Le délai lors de la recherche d'un mot de longueur m dans un texte à l'aide de l'algorithme de Knuth, Morris et Pratt est au plus $\log_\Phi(m+1)$ (où $\Phi = (1+\sqrt{5})/2$ est le nombre d'or).

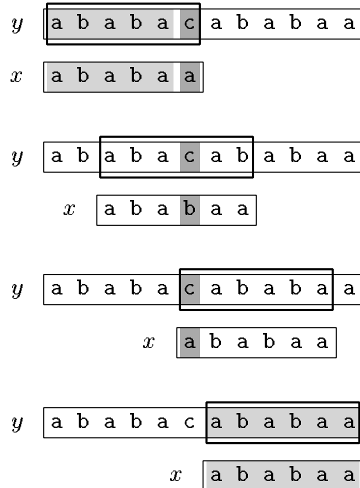


Figure 5.3 L'algorithme de Knuth, Morris et Pratt effectue 14 comparaisons en 4 tentatives lors de la recherche de ababaa dans ababacababaa.

5.4 Algorithme de Simon

L'algorithme de Knuth, Morris et Pratt peut s'interprêter en termes d'automates, en effet un texte y contient le mot x si $y \in A^*xA^*$. Pour trouver toutes les occurrences de x dans y il suffit de déterminer tous les préfixes de y qui appartiennent au langage rationnel A^*x. Il faut donc construire l'automate minimal déterministe reconnaissant A^*x pour obtenir un algorithme de recherche de mot efficace.

L'automate déterministe minimal $\mathcal{A}$ reconnaissant le langage A^*x est défini comme suit : $\mathcal{A} = (Q, q_0, T, \delta)$ où :

- Q est un ensemble de $m+1$ états correspondant aux $m+1$ préfixes du mot x (chaque état $q \in Q$ est donc identifié à un préfixe de x) ;
- $q_0 = \varepsilon$ est l'état initial de l'automate ;
- $T = \{x\}$ est l'ensemble formé par l'unique état terminal de l'automate ;
- δ est la fonction de transition de l'automate avec $\delta(p, a) = f_x(pa)$ avec $p \in Q$ et $a \in A$ où $f_x(u)$ est le plus long suffixe de u qui est un préfixe de x.

Un exemple d'automate est donné figure 5.4.

Le principal inconvénient de cet algorithme est que l'automate prend une place en mémoire en $O(\text{card}\, A \times (m+1))$ ce qui peut être préjudiciable pour de grands alphabets. Simon a remarqué que seules quelques transitions de l'automate sont significatives, ce sont les flèches « avant » qui font passer d'un préfixe de longueur i à un préfixe de longueur $i+1$

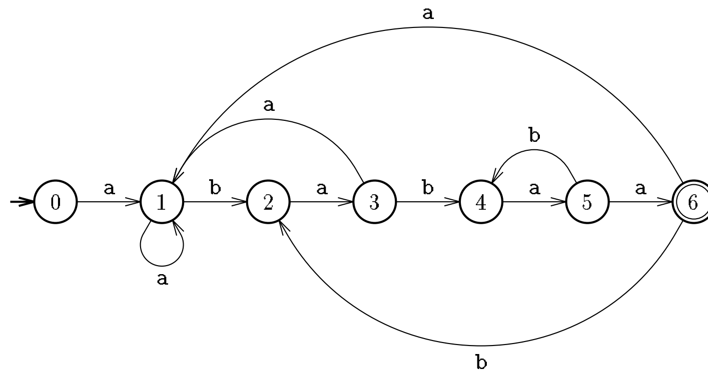


Figure 5.4 L'automate déterministe reconnaissant A^*ababaa sans les flèches menant à l'état initial.

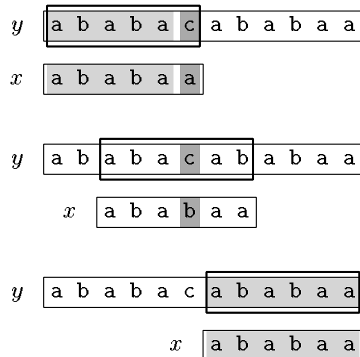


Figure 5.5 L'algorithme de Simon effectue 13 comparaisons en 3 tentatives lors de la recherche de ababaa dans ababacababaa.

et les flèches « arrière » qui font passer d'un préfixe de longueur i à un préfixe de longueur k avec $0 < k \leq i$. Les autres flèches menant à l'état initial de l'automate ne sont pas significatives car on peut les déduire. Simon propose donc de représenter l'automate $\mathcal{A}$ en associant à chaque état la liste des transitions significatives qui partent de cet état. Au total, il y a au plus $2m$ flèches significatives.

On associe donc à chaque état $p \in Q$ la liste des transitions (a, q) telles que $\delta(p, a) = q$ et $q \neq q_0$.

Un exemple d'exécution de l'algorithme de Simon est donné figure 5.5.

Théorème 5.5

L'algorithme de Simon localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(n)$. Il effectue au plus $2n$ comparaisons entre des lettres du mot et des lettres du texte. Sa

phase de prétraitement s'exécute en temps et espace $O(m)$. Il nécessite un espace supplémentaire $O(m)$.

Théorème 5.6
Le délai lors de la recherche d'un mot de longueur m dans un texte à l'aide de l'algorithme de Simon est au plus $\min\{1 + \log_2 m, \text{card}\, A\}$.

5.5 Algorithme de Colussi

Colussi a découvert qu'on pouvait dériver l'algorithme de Knuth, Morris et Pratt à partir de l'algorithme naïf en utilisant les preuves de programme de Hoare. Il ne s'est pas arrêté là et en utilisant toujours la même technique il a aboutit à un nouvel algorithme de recherche de mot qui effectue au plus $\frac{3}{2}n$ comparaisons dans le pire des cas.

L'idée principale de l'algorithme consiste à découper chaque tentative en deux phases en partitionnant les positions sur le mot en deux ensembles disjoints :

- la première phase consiste à examiner d'abord dans l'ordre croissant les lettres du texte alignées avec des lettres du mot pour lesquelles la fonction de décalage de l'algorithme de Knuth, Morris et Pratt est supérieure à 0 (on appelle ces positions dans le mot des positions privilégiées).
- la seconde consiste à examiner dans l'ordre décroissant les positions restantes.

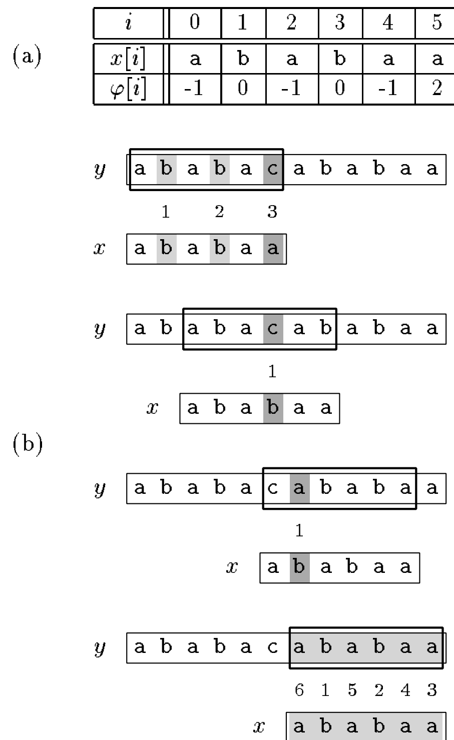
Cette stratégie présente deux avantages :

(i) si une inégalité intervient dans la première phase alors après avoir effectué le décalage correspondant, il est inutile de réexaminer les lettres du texte alignées avec des positions privilégiées déjà examinées lors de la tentative précédente.

(ii) si une inégalité intervient dans la seconde phase cela signifie qu'on a reconnu un facteur du texte qui est un suffixe du mot, après le décalage ce facteur du texte sera aligné avec un préfixe du mot et lui sera égal, il sera donc inutile de le réexaminer lors de la tentative suivante.

Un exemple d'exécution de l'algorithme de Colussi est donné figure 5.6.

Théorème 5.7
L'algorithme de Colussi localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(n)$. Il effectue au plus $\frac{3}{2}n$ comparaisons entre des lettres du mot et des lettres du texte. Sa phase de prétraitement s'exécute en temps et espace $O(m)$. Il nécessite un espace supplémentaire $O(m)$.



(a)

i	0	1	2	3	4	5
$x[i]$	a	b	a	b	a	a
$\varphi[i]$	-1	0	-1	0	-1	2

Figure 5.6 **(a)** Les valeurs de la fonction de décalage de Knuth, Morris et Pratt. On mémorise pour chaque position i sur le mot x la position de la lettre à comparer après une comparaison négative avec $x[i]$. **(b)** L'algorithme de Colussi effectue 11 comparaisons en 4 tentatives lors de la recherche de ababaa dans ababacababaa. Les chiffres indiquent l'ordre dans lequel les comparaisons sont effectuées lors de chaque tentative.

6 Algorithmes rapides en pratique

Les algorithmes de ce chapitre effectuent les comparaisons durant chaque tentative de la droite vers la gauche à l'intérieur de la fenêtre. Cette stratégie permet de ne pas avoir à examiner systématiquement toutes les lettres du texte et ainsi économiser des comparaisons.

6.1 Algorithme de Boyer-Moore

Pendant une tentative de l'algorithme de Boyer-Moore on compare les symboles du mot avec ceux de la fenêtre de la droite vers la gauche en commençant par la lettre la plus à droite du mot avec la lettre la plus à droite de la fenêtre. En cas de succès on se déplace d'une position vers la gauche à l'intérieur du mot et à l'intérieur de la fenêtre et on continue les comparaisons jusqu'à une inégalité ou jusqu'à l'extrémité gauche du mot et de la fenêtre. En cas d'inégalité ou de découverte d'une occurrence du mot, la fenêtre est décalée vers la droite grâce à une fonction de décalage préalablement calculée sur le mot.

La fonction de décalage consiste à déplacer la fenêtre vers la droite de manière à faire correspondre le suffixe déjà reconnu du mot dans le texte avec son occurrence la plus à droite dans $x[0\,..\,m-2]$. Si ce n'est pas possible on essaye d'aligner le plus long suffixe du mot déjà reconnu dans le texte avec un préfixe du mot.

Considérons le cas général pendant une tentative j de l'algorithme de Boyer-Moore, $x[0\,..\,m-1]$ est donc aligné avec $y[j-m+1\,..\,j]$. On commence par comparer $x[m-1]$ avec $y[j]$ et on progresse vers la gauche tant que les lettres comparées sont égales. Supposons que les lettres correspondent jusqu'à $x[k+1]$ et $y[j-m+k+2]$ et que $x[k] \neq y[j-m+k+1]$. Posons $u = y[j-m+k+2\,..\,j] = x[k+1\,..\,m-1]$, $a = x[k]$ et $b = y[i+k]$. Alors le décalage suffixe consiste à aligner le facteur u du texte avec le facteur u du mot le plus à droite dans $x[0\,..\,m-2]$. On peut distinguer trois cas suivant la restriction imposée sur la lettre c précédant ce facteur u dans $x[0\,..\,m-2]$:

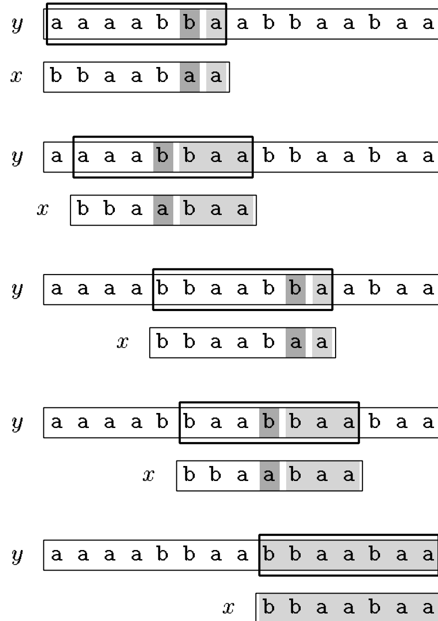


Figure 6.1 L'algorithme de Boyer-Moore effectue 19 comparaisons en 5 tentatives lors de la recherche de bbaabaa dans aaaabbaabbaabaa.

- Aucune restriction n'est imposée sur c, en particulier c peut être égale à a. On parle alors de décalage de ***faible suffixe***.
- On impose à c d'être différente de a. On parle alors de décalage de ***bon suffixe***.
- On impose à c d'être égale à b. On parle alors de décalage de ***meilleur suffixe***.

S'il n'existe pas de facteur cu dans le mot alors on considère le facteur v qui est le plus long bord de x de longueur inférieure à $|u|$. On aligne son occurrence gauche dans le mot avec son occurrence dans le facteur u de y.

Ces trois fonctions de décalages peuvent être calculées en temps et espace $O(m)$. Toutefois la quasi-totalité des présentations de l'algorithme de Boyer-Moore utilise le décalage de bon suffixe. C'est également ce décalage que nous retiendrons ici.

Un exemple d'exécution de l'algorithme de Boyer-Moore, utilisant le décalage de bon suffixe, est donné figure 6.1.

Théorème 6.1

L'algorithme de Boyer-Moore localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(m \times n)$. Il nécessite un espace supplémentaire $O(m)$. Sa phase de prétraitement

peut être effectuer en temps et espace $O(m)$.

Théorème 6.2 (Cole 94)
L'algorithme de Boyer-Moore localise toutes les occurrences d'un mot non-périodique de longueur m dans un texte de longueur n en temps $O(n)$. Il effectue, dans ce cas, au plus $3n - n/m$ comparaisons entre lettres du mot et lettres du texte.

En pratique la fonction de bon suffixe est utilisée conjointement avec une autre fonction de décalage définie pour chaque lettre de l'alphabet. Cette fonction, appelée fonction de dernière occurrence est définie pour une lettre $a \in A$ comme étant la distance entre l'occurrence la plus à droite de a dans $x[0\,..\,m-2]$ et l'extrémité droite de x. Cette fonction peut être calculée en temps et espace $O(m + \mathrm{card}\, A)$. Son utilisation ne modifie pas la complexité de la phase de recherche.

6.2 Algorithme de Turbo-BM

L'algorithme Turbo-BM est une variante de l'algorithme Boyer-Moore. Le trait principal de l'algorithme Turbo-BM est que lors d'une tentative il mémorise le plus long facteur du texte qui correspondait avec un suffixe du mot lors de la tentative précédente. Cela présente deux avantages :

- pouvoir éventuellement effectuer un saut par dessus ce facteur lors de la tentative courante ;
- pouvoir éventuellement effectuer un ***turbo-décalage*** après la tentative courante.

Nous allons maintenant expliquer ce qu'est un turbo-décalage. Soit v le plus long suffixe du mot qui correspond avec le texte pendant la tentative courante. Soit u le facteur mémorisé lors de la tentative précédente et qui est donc un facteur du texte et un suffixe du mot. Et soient a et b les lettres respectivement du mot et du texte qui provoquent l'inégalité lors de la tentative courante. Le mot x peut donc s'écrire $x'uzv$. Un turbo-décalage peut intervenir dans le cas où $|v| < |u|$. Puisque v est moins long que u, av est un suffixe de u. Donc a et b apparaissent à une distance $|zv|$ dans le texte, mais puisque le suffixe uzv du mot à une période de longueur $|zv|$, le décalage à effectuer doit être supérieur à $|u| - |v|$. Par un argument similaire et toujours dans le cas où $|v| < |u|$ on peut démontrer que la longueur du décalage doit être au moins aussi grande que $|v|$.

La figure 6.2 montre un exemple d'exécution de l'algorithme Turbo-BM.

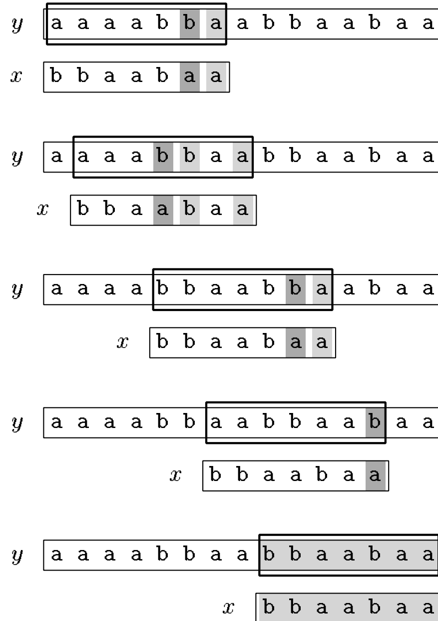


Figure 6.2 L'algorithme de Turbo-BM effectue 15 comparaisons en 5 tentatives lors de la recherche de bbaabaa dans aaaabbaabbaabaa.

Théorème 6.3
L'algorithme de Turbo-BM localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(n)$. Il effectue au plus $2n$ comparaisons entre lettres du mot et lettres du texte. Il nécessite un espace supplémentaire $O(m)$. Sa phase de prétraitement peut être effectuer en temps et espace $O(m)$.

L'algorithme Turbo-BM nécessite un espace supplémentaire constant par rapport à l'algorithme de Boyer-Moore.

6.3 Algorithme d'Apostolico-Giancarlo

L'algorithme d'Apostolico-Giancarlo est une variante de l'algorithme de Boyer-Moore qui mémorise, lors de la phase de recherche, pour chaque position droite de la fenêtre la longueur du plus long suffixe du mot qui se termine à cette position. Après chaque tentative à une position j' dans le texte y, la longueur du plus long suffixe de x reconnu à la position droite j', $|u|$, est mémorisée dans une table S ($S[j'] = |u|$). Ainsi, lorsque pendant la tentative courante à la position j sur le texte y on est amené à examiner une position j', $j' < j$, (on a $y[j'+1\,..\,j]$ est un suffixe de x),

pour laquelle la valeur $k = S[j']$ est définie, on sait que $y[j' - k + 1 \mathinner{..} j']$ est un suffixe de x. Soit $i = m - 1 - j + j'$. Il suffit alors de connaitre la longueur $s = \textit{suff}[i]$, du plus long suffixe de x se terminant à la position i dans x, pour conclure la tentative dans la majorité des situations.

Quatre cas peuvent se produire :

- Si $s \leq k$ et $s = i + 1$, une occurrence de x apparait à la position droite j dans le texte y. On a $S[j] = m$ et le décalage à appliquer est de longueur égale à la période de x.
- Si $s \leq i$ et $s < k$, on a $S[j] = m-1-i+s$ et la longueur du décalage à appliquer est donnée par la valeur du décalage pour la position $i - s$ sur le mot.
- Si $k < s$, on a $S[j] = m - 1 - i + k$ et la longueur du décalage à appliquer est donnée par la valeur du décalage pour la position $i-k$ sur le mot.
- Si $k = s$, on a $x[i-s+1 \mathinner{..} m-1] = y[j'-s+1 \mathinner{..} j]$. Il faut effectuer un saut et reprendre les comparaisons avec les lettres $x[i - s]$ et $y[j' - s]$.

Seul un cas sur quatre requiert des comparaisons supplémentaires.

La table *suff* est calculée en temps et espace $O(m)$ pendant le calcul de la fonction de décalage. Seules les valeurs de la table S pour les lettres du texte contenues dans la fenêtre sont nécessaires à chaque tentative. L'ensemble de ces valeurs peuvent donc être mémorisées en espace $O(m)$.

Un exemple d'exécution de l'algorithme d'Apostolico-Giancarlo est donné figure 6.3.

Théorème 6.4

L'algorithme d'Apostolico-Giancarlo localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(n)$. Il effectue au plus $\frac{3}{2}n$ comparaisons entre lettres du mot et lettres du texte. Il nécessite un espace supplémentaire $O(m)$. Sa phase de prétraitement peut être effectuer en temps et espace $O(m)$.

L'algorithme d'Apostolico-Giancarlo nécessite un espace supplémentaire linéaire en la longueur du mot par rapport à l'algorithme de Boyer-Moore.

Notes

Quelques ouvrages sont entièrement consacrés au traitement du texte ([2], [3], [4], [14], [17], [19], [25] et [29]).

Des ouvrages collectifs ([1], [12], [13] et [15]) contiennent des chapitres dédiés à cette problèmatique.

Quelques sites consacrés à l'algorithmique du texte se trouve sur la Toile de l'Internet. Ils contiennent des bibliographies régulièrement mises

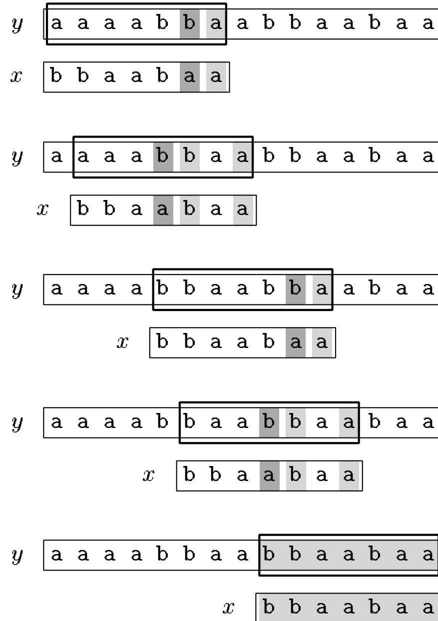


Figure 6.3 L'algorithme d'Apostolico-Giancarlo effectue 14 comparaisons en 5 tentatives lors de la recherche de `bbaabaa` dans `aaaabbaabbaabaa`.

à jour [23], des pointeurs sur les acteurs du domaine [24] et des descriptions et animations d'algorithmes [7]. Ce dernier site recense une trentaine d'algorithmes de recherche de mot et en propose une description synthétique, le code en C ainsi que la possibilité de visualiser l'exécution des algorithmes avec des données fournies par l'utilisateur.

Le théorème 4.1 est apparu dans [18]. Le théorème 4.2 a été publié dans [30]. Le théorème 4.3 est apparu dans [9].

L'algorithme de Morris et Pratt est apparu dans [26]. L'exemple utilisé dans le chapitre 5 est tiré de [20]. L'algorithme de Knuth, Morris et Pratt a été publié dans [22] L'algorithme de Simon est décrit dans [28], [20] et [21]. Le théorème 5.6 est dû à [20]. L'algorithme de Colussi a été publié dans [10].

L'algorithme de Boyer-Moore a été publié dans [6] et analysé par D. Knuth dans [22]. La première publication du calcul de la fonction de décalage de bon suffixe est due à W. Rytter [27]. Un calcul de la fonction de décalage du meilleur suffixe se trouve dans [20]. La preuve du théorème 6.2 est due à R. Cole [8]. L'algorithme Turbo-BM a été publié dans [11]. L'algorithme d'Apostolico-Giancarlo a été publié dans [5]. La présentation qui en est faite ici et le théorème 6.4 sont issus de [16].

Remerciements

Je remercie chaleureusement Maxime Crochemore de m'avoir permis d'utiliser le matériel de son cours « Text Searching and Processing » au « Master in Advanced Computing » du King's College London. Je remercie non moins chaleureusement Christophe Hancart de m'avoir permis d'utiliser le style LaTeX qu'il a développé pour [14].

Bibliographie

[1] A. V. Aho. Algorithms for finding patterns in strings. Édité par J. van Leeuwen. *Handbook of Theoretical Computer Science, Algorithms and Complexity*, vol. A, chap. 5, pp. 255–300. Elsevier, 1990.

[2] J.-I. Aoe, éditeur. *String pattern matching strategies.* IEEE Computer Society Press, 1994.

[3] A. Apostolico et Z. Galil, éditeurs. *Combinatorial algorithms on words*, vol. 12. Springer-Verlag, 1985.

[4] A. Apostolico et Z. Galil, éditeurs. *Pattern matching algorithms.* Oxford University Press, 1997.

[5] A. Apostolico et R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, 1986.

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

[7] C. Charras et T. Lecroq. *Exact String Matching Algorithms.* `http://www-igm.univ-mlv.fr/~lecroq/string/`

[8] R. Cole. Tight bounds on the complexity of the Boyer-Moore string matching algorithm. *SIAM J. Comput.*, 23(5):1075–1091, 1994.

[9] R. Cole et R. Hariharan. Tighter Upper Bounds on the Exact Complexity of String Matching. *SIAM J. Comput.*, 26(3):803–856, 1997.

[10] L. Colussi. Correctness and efficiency of the pattern matching algorithms. *Inf. Comput.*, 95(2):225–251, 1991.

[11] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski et W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

[12] M. Crochemore et C. Hancart. Automata for matching patterns. Édité par G. Rozenberg et A. Salomaa. *Handbook of Formal Languages, Volume 2 Linear Modeling: Background and Application*, chap. 9, pp. 399–462. Springer-Verlag, 1997.

[13] M. Crochemore et C. Hancart. Pattern Matching in Strings. Édité par M. J. Atallah. *Algorithms and Theory of Computation Handbook*, chap. 11, pp. 11.1–11.28. CRC Press, 1998.

[14] M. Crochemore, C. Hancart et T. Lecroq. *Algorithmique du texte.* En préparation, à paraître chez Vuibert.

[15] M. Crochemore et T. Lecroq. Pattern matching and text data compression algorithms. Édité par A. B. Tucker Jr. *The Computer Science and Engineering Handbook*, chap. 8, pp. 162–202. CRC Press, 1996.

[16] M. Crochemore et T. Lecroq. Tight bounds on the complexity of the Apostolico-Giancarlo algorithm. *Inform. Process. Lett.*, 63(4):195–203, 1997.

[17] M. Crochemore et W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[18] Z. Galil et J. Seiferas. Time-space optimal string matching. *J. Comput. Syst. Sci.*, 26(3):280–294, 1983.

[19] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[20] C. Hancart. *Analyse exacte et en moyenne d'algorithmes de recherche d'un motif dans un texte*. Thèse de doctorat de l'Université Paris 7. Rapport 93-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1993.

[21] C. Hancart. On Simon's string searching algorithm. *Inform. Process. Lett.*, 47(2):95–99, 1993.

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

[23] T. Lecroq, webmestre.
`http://www-igm.univ-mlv.fr/~lecroq/tq.bib`

[24] S. Lonardi, webmestre.
`http://www.dei.unipd.it/stelo/`
`http://www.cs.purdue.edu/homes/stelo/pattern.html`

[25] M. Lothaire, éditeur. *Combinatorics on words*. Cambridge University Press, seconde édition, 1997.

[26] J. H. Morris, Jr, et V. R. Pratt. *A linear pattern-matching algorithm*. Rapport 40, University of California, Berkeley, 1970.

[27] W. Rytter. A correct preprocessing algorithm for Boyer-Moore string searching. *SIAM J. Comput.*, 9(3):509–512, 1980.

[28] I. Simon. String matching algorithms and automata. Édité par R. Baeza-Yates et N. Ziviani. *Proceedings of the 1st South American Workshop on String Processing*, 1993, pp. 151–157.

[29] G. A. Stephen. *String searching algorithms*. World Scientific Press, 1994.

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