Chapitre 10

Motifs

Dans ce chapitre, nous considérons d'abord le problème de la recherche d'une ou de toutes les occurrences d'un mot x dans un texte t. Nous présentons l'algorithme naïf, l'algorithme de Morris et Pratt, et sa variante due à Knuth, Morris et Pratt, l'implémentation par automate fini et l'algorithme de Simon, et pour finir l'algorithme de Boyer et Moore, dans sa version de Horspool et dans la version complète. Ensuite, nous considérons la recherche d'une occurrence de plusieurs motifs, et décrivons l'algorithme de Aho et Corasick. Dans la dernière section, nous étudions la recherche d'occurrences de mots décrits par une expression rationnelle.

Introduction

La recherche de motifs dans un texte est un problème important qui apparaît dans de nombreux domaines scientifiques. En informatique, on le rencontre naturellement en traitement des données, dans l'édition de textes, en analyse syntaxique ou en recherche d'informations ; en particulier, tous les éditeurs de textes et de nombreux langages de programmation offrent des possibilités de recherche de motifs.

Dans sa forme la plus simple, le problème se ramène à localiser une occurrence d'un mot, le *motif*, dans une chaîne de caractères, le *texte*. Par exemple, le texte `rechercher` contient deux occurrences du motif `cher`. Même pour ce problème simple, il existe de nombreux algorithmes intéressants, plus ou moins sophistiqués, et qui sont plus efficaces que la méthode naïve qui vient immédiatement à l'esprit. Le problème devient plus complexe lorsque l'on autorise des ensembles de motifs ou des motifs représentés par des expressions rationnelles. Par exemple, dans plusieurs traitements de textes courants l'expression `ch.*r` dénote les mots qui commencent par `ch` et qui se terminent par `r`. Dans le texte `rechercher`, les mots `cher` et `chercher` sont des occurrences de mots décrits par cette expression.

Le résultat effectif de l'algorithme dépend de l'application considérée. Dans le cas où les données sont organisées en lignes par exemple, comme dans un dictionnaire ou un listage de programme, nous pouvons être intéressés par les lignes qui correspondent au motif. En compilation, on vise plutôt à partitionner la chaîne de caractères d'entrée en une suite de lexèmes, tels que commentaires, identificateurs, opérateurs, où la forme des lexèmes est déterminée par une expression rationnelle. Dans l'édition de textes, on cherche des occurrences de motifs en vue notamment de les remplacer par d'autres.

Dans ce chapitre, nous considérons trois problèmes : le plus important, et qui sera traité en détail, est la recherche d'une ou de toutes les occurrences d'un mot x dans un texte t. Nous présentons trois algorithmes pour ce problème. Ensuite, nous considérons la recherche d'une occurrence de plusieurs motifs, et enfin la recherche d'occurrences de mots décrits par une expression rationnelle. Dans tous les cas, c'est le motif recherché qui subit une analyse préalable, et jamais le texte. En effet, le motif est considéré comme fixe, et le texte est changeant et inconnu. Une situation duale, où le texte est fixe et le motif peut varier, se présente par exemple dans la recherche d'entrées dans un dictionnaire. Une recherche efficace demande alors d'organiser et de coder de manière convenable le dictionnaire. Ce problème ne sera pas considéré ici.

10.1 Recherche d'un motif

Le problème que nous abordons dans cette section est le suivant : étant donnés un *texte* $t \in A^*$ et un *motif* $x \in A^*$, où A est un alphabet, déterminer si x est un facteur de t, et le cas échéant trouver une occurrence de x dans t. L'efficacité d'un algorithme de recherche se mesure en nombre de comparaisons de caractères.

Posons $t = t_1 \cdots t_n$ et $x = x_1 \cdots x_m$, où les t_j et les x_i sont des lettres. Les algorithmes que nous présentons sont bâtis sur le schéma que voici : le motif x est comparé aux facteurs $t_{k+1} \cdots t_{k+m}$ de longueur m de t jusqu'à trouver une coïncidence, si elle existe. L'algorithme naïf fait cette comparaison pour toutes les positions $k = 0, \ldots, n-m$. Les améliorations que nous examinons ensuite ne font la comparaison que pour un sous-ensemble des positions, en tirant profit de la connaissance du texte t accumulée lors des comparaisons précédentes et d'un prétraitement du motif x. Le schéma général est donc :

```
RECHERCHE-D'UN-MOTIF(x, t) ;
  k := 0 ;
  tester l'égalité x = t_{k+1} ··· t_{k+m} ;
  s'il y a égalité, reporter k et arrêter
  sinon augmenter k et recommencer.
```

De façon imagée, la méthode consiste à faire «glisser» le motif x le long de la chaîne t, et à comparer x au bloc de t couvert (voir figure 1.1). Les divers algorithmes déterminent selon des critères différents la position suivante où le motif x a des chances de se trouver dans le texte t. L'efficacité d'une méthode spécifique dépend grandement du prétraitement appliqué au motif x. L'information recueillie sur le motif peut être employée pour former un analyseur, voire un automate fini, qui est ensuite appliqué au texte.

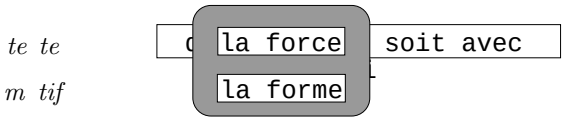

Figure 1.1: *Le motif glissant sur le texte.*

10.1.1 Un algorithme naïf

Posons $t = t_1 \cdots t_n$ et $x = x_1 \cdots x_m$. L'algorithme «naïf» consiste à comparer le motif x à chaque facteur de t de longueur m. Si une occurrence est rencontrée, on la signale; sinon, on recommence avec le facteur suivant de t. Le but est de calculer un entier k où commence une occurrence de x, c'est-à-dire tel que

$$x_1 \cdots x_m = t_{k+1} \cdots t_{k+m}$$

L'algorithme «naïf» suivant réalise cette recherche :

```
procédure RECHERCHE-NAÏVE(x, t);
  i := 1; j := 1;
  tantque i ≤ m et j ≤ n faire
    si t[j] = x[i] alors i := i + 1; j := j + 1
    sinon j := j − i + 2; i := 1 finsi
  fintantque;
  si i > m alors
    occurrence de x à la position j − m
  sinon
    pas d'occurrence
  finsi.
```

Dans le cas le plus défavorable, le nombre de comparaisons est $(n - m + 1)m = nm - m^2 + m$. Ce cas est réalisé par exemple pour $x = a^{m-1}b$, $t = a^{n-1}b$. Pour m petit devant n, ce nombre est de l'ordre de $nm = |x||t|$. Toutefois, le comportement moyen de l'algorithme naïf est plutôt bon, comme le montre l'observation suivante :

Proposition 1.1. *Si l'alphabet comporte au moins deux lettres, et dans l'hypothèse d'une distribution de probabilité uniforme et indépendante sur les lettres, le nombre moyen de comparaisons pour rechercher un motif dans un texte de longueur n par l'algorithme naïf est au plus $2n$.*

Bien entendu, les textes et les motifs n'ont, dans la pratique, aucune raison d'être équiprobables, ce qui limite quelque peu la portée de la proposition.

Preuve. Soit $q = |A|$. Considérons un motif fixé $x = x_1 \cdots x_m$. Nous montrons que le nombre moyen de comparaisons de caractères faites dans la comparaison de x à $t_{k+1} \cdots t_{k+m}$ est majoré par 2. Ce nombre est

$$\sum_{i=1}^{m} i c_i$$

où c_i est la probabilité pour que l'on fasse exactement i comparaisons. Or

$$\sum_{i=1}^{m} i c_i = \sum_{i=1}^{m} d_i$$

où $d_i = c_i + \cdots + c_m$ est la probabilité pour que l'on fasse au moins i comparaisons. Maintenant, on fait au moins i comparaisons lorsque

$$x_1 \cdots x_{i-1} = t_{k+1} \cdots t_{k+i-1}$$

de sorte que $d_i = 1/q^{i-1}$. Le nombre moyen de comparaisons est donc

$$1 + 1/q + \cdots + 1/q^{m-1} \leq 1 + \frac{1}{q-1} \leq 2$$

Il en résulte que le nombre moyen de comparaisons de l'algorithme naïf sur un texte t de longueur n est majoré par $2n$. ∎

10.1.2 L'algorithme de Morris et Pratt

La lenteur de l'algorithme naïf, dans le cas le plus défavorable, s'explique par le fait qu'en cas d'échec, il recommence à zéro la comparaison du motif x au facteur suivant de t, sans exploiter l'information contenue dans la réussite partielle de la tentative précédente. Si l'échec s'est produit à la i-ième lettre du motif x, on a (voir figure 1.2) pour un entier k

$$x_1 \cdots x_{i-1} = t_{k+1} \cdots t_{k+i-1} \quad \text{et} \quad x_i \neq t_{k+i}$$

Toute recherche ultérieure qui commence dans le facteur $t_{k+1} \cdots t_{k+i-1}$ peut tirer profit du fait que le mot $t_{k+1} \cdots t_{k+i-1}$ est un préfixe du motif x. Si une nouvelle recherche commence en position $\ell + 1$ (avec $k < \ell < k + i - 1$), elle compare

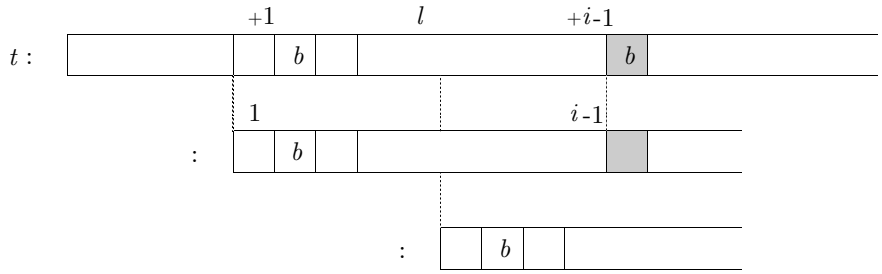

Figure 1.2: *Echec à la i-ième lettre du motif.*

$t_{\ell+1}t_{\ell+2}\cdots$ à $x_1x_2\cdots$. Or $t_{\ell+1}t_{\ell+2}\cdots$ est un suffixe de $t_{k+1}\cdots t_{k+i-1}$ qui lui est égal à $x_1\cdots x_{i-1}$. En d'autres termes, on compare $x_1x_2\cdots$ à un suffixe de $x_1\cdots x_{i-1}$, donc à un morceau du motif lui-même!

Ces comparaisons sont indépendantes du texte t, puisqu'elles ne concernent que des morceaux du motif. On peut donc les faire avant de considérer t, et on peut en attendre un gain de temps substantiel dans la mesure où ce prétraitement sur le motif ne sera fait qu'une seule fois et qu'il permettra d'éviter, lors de l'examen du texte, de répéter des comparaisons identiques à plusieurs endroits différents du texte.

Le *prétraitement* sur le motif x que nous allons réaliser permettra de reconnaître rapidement les seules configurations où la recherche d'une occurrence vaut la peine d'être continuée. Pour cela, il s'agit de déterminer les indices i où le mot $x_1\cdots x_{i-1}$ se termine par un préfixe de x. Introduisons une définition. Soit u un mot quelconque non vide; un *bord* de u est un mot distinct de u qui est à la fois préfixe et suffixe de u.

Exemple. Le mot $u = abacaba$ possède les trois bords ε, a, et aba. Le mot $u = abcabcab$ possède les bords ε, ab et $abcab$.

On note Bord(x) et on appelle *bord maximal* le bord le plus long d'un mot non vide x. Si x est de longueur m, on définit une fonction

$$\beta : \{0, 1, \ldots, m\} \to \{-1, \ldots, m-1\}$$

dépendant de x par $\beta(0) = -1$ et pour $i > 0$, par

$$\beta(i) = |\mathrm{Bord}(x_1 \cdots x_i)|$$

Bien entendu, $\beta(i) \leq i - 1$. Voici par exemple les bords maximaux et leurs longueurs, pour les préfixes du mot *abacabac* :

		a	b	a	c	a	b	a	c
Indice	0	1	2	3	4	5	6	7	8
Bord	$-$	ε	ε	a	ε	a	ab	aba	$abac$
β	-1	0	0	1	0	1	2	3	4

Revenons à l'amélioration de l'algorithme naïf. Si (voir figure 1.3)

$$x_1 \cdots x_{i-1} = t_{k+1} \cdots t_{k+i-1} \quad \text{et} \quad x_i \neq t_{k+i},$$

alors le mot $t_{k+p+1} \cdots t_{k+i-1}$ ne peut être début d'une occurrence de x que s'il est un bord de $x_1 \cdots x_{i-1}$. Il suffit donc, pour chercher une occurrence, de «décaler» x

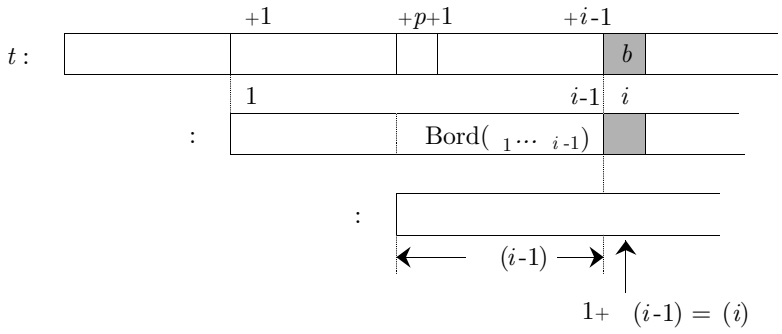

Figure 1.3: *Décalage d'une position.*

d'une longueur appropriée pour superposer $x_1 \cdots x_{i-1}$ avec son plus grand bord. Le décalage se réalise en poursuivant les comparaisons entre la lettre t_{k+i} et la lettre $x_{1+\beta(i-1)}$. Dans la pratique, on utilise à la place de β une fonction

$$s : \{1, \ldots, m\} \to \{0, \ldots, m\}$$

définie pour $i = 1, \ldots, m$ par

$$s(i) = 1 + \beta(i-1)$$

de sorte que le décalage consiste à remplacer i par $s(i)$. La fonction s est appelée la *fonction de suppléance* du motif x. Voici par exemple les fonctions β et s pour le mot *abacabac* :

		a	b	a	c	a	b	a	c
	0	1	2	3	4	5	6	7	8
β	-1	0	0	1	0	1	2	3	4
s		0	1	1	2	1	2	3	4

Avec la fonction de suppléance, on obtient l'algorithme ci-dessous, dû à Morris et Pratt. Dans cet algorithme, j est l'indice de la lettre courante du texte t, et i l'indice de la lettre courante du motif x. A chaque tour dans la boucle *tantque*, on a

$$x_1 \cdots x_{i-1} = t_{j-i+1} \cdots t_{j-1}$$

Si $t_j = x_i$, on progresse dans l'analyse, et sinon on décale x en remplaçant i par $s(i) = 1 + \beta(i-1)$.

```
procédure MORRIS-PRATT(x, t);
  i := 1; j := 1;
  tantque i ≤ m et j ≤ n faire
    si i ≥ 1 etalors t[j] ≠ x[i] alors i := s(i)
    sinon i := i + 1; j := j + 1 finsi
  fintantque;
  si i > m alors
    occurrence de x à la position j − m
  sinon
    pas d'occurrence de x dans t
  finsi.
```

Proposition 1.2. *L'algorithme de Morris et Pratt calcule une occurrence d'un motif x dans un texte t en au plus $2|t| - 1$ comparaisons de caractères, si l'on dispose de la fonction de suppléance sur x.*

Preuve. Posons $n = |t|$. Appelons test positif un test pour lequel $t[j] = x[i]$, et test négatif un test pour lequel $t[j] \neq x[i]$. Chaque test positif incrémente i et j, et chaque test négatif diminue i, éventuellement de plus d'une unité. Comme chaque test positif augmente j, il y a au plus n tests positifs. Il n'y a n tests positifs que si i ne s'annule pas.

Pour compter le nombre de tests négatifs, considérons l'entier $j - i$. Au début, $j - i = 0$. Un test positif ne change pas la valeur de $j - i$, un test négatif l'augmente strictement. Donc le nombre de tests négatifs est majoré par la valeur qu'a $j - i$ à la fin du calcul, donc par n ou par $n - 1$ selon que i s'annule ou non. ∎

Exemple. La table suivante donne, pour la recherche du motif $x = abacabac$ dans le texte $t = babacacabacaab$, la suite des valeurs que prend l'indice i dans l'algorithme :

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	b	a	b	a	c	a	c	a	b	a	c	a	a	b
i	1	1	2	3	4	5	6	1	2	3	4	5	6	2
	0						2						2	
							1						1	
							0							

La figure 1.4 montre les décalages successifs du motif. Les différences sont constatées aux positions sombres du texte. Le nombre total de comparaisons est 18. Lorsque $j = 7$ et $i = 6$, on a $t_j \neq x_i$, et on compare t_7 à la lettre qui suit le bord maximal de $x_1 \cdots x_5$, à savoir x_2. Le même schéma se répète pour $j = 13$, et en fait chaque fois qu'une lettre t_j est comparée à x_6. A chaque fois, on compare t_j à x_2, et cette comparaison est inutile parce que $x_6 = x_2$. Une version de l'algorithme, due à Knuth, Morris et Pratt et que nous présentons plus loin permet d'éliminer en partie ces comparaisons redondantes.

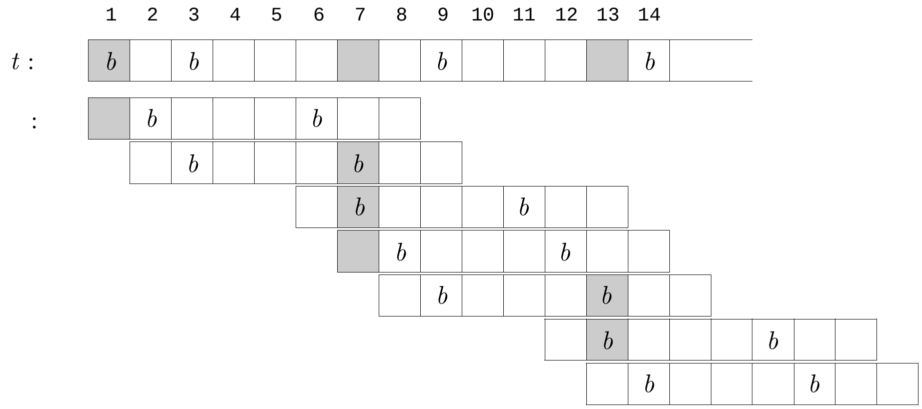

Figure 1.4: *Décalages successifs du motif.*

10.1.3 Bords

Pour compléter l'exposé de l'algorithme de Morris et Pratt, il faut indiquer comment calculer la fonction β ou, de manière équivalente, la fonction de suppléance. Cela demande une étude plus approfondie des bords d'un mot.

Proposition 1.3. *Soit x un mot non vide, et soit k le plus petit entier tel que* $\mathrm{Bord}^k(x) = \varepsilon$.

(1) *Les bords de x sont les mots* $\mathrm{Bord}(x), \mathrm{Bord}^2(x), \ldots, \mathrm{Bord}^k(x)$.

(2) *Soit a une lettre. Alors* $\mathrm{Bord}(xa)$ *est le plus long préfixe de x qui est dans l'ensemble* $\{\mathrm{Bord}(x)a, \mathrm{Bord}^2(x)a, \ldots, \mathrm{Bord}^k(x)a, \varepsilon\}$.

Preuve. (1) Un bord de $\mathrm{Bord}(x)$ est aussi un bord de x, donc les mots $\mathrm{Bord}(x)$, $\mathrm{Bord}^2(x)$, …, $\mathrm{Bord}^k(x)$ sont tous des bords de x. Réciproquement, soit z un bord de x. Ou bien $z = \mathrm{Bord}(x)$, ou alors z est un bord de $\mathrm{Bord}(x)$. Par récurrence, z est un des mots $\mathrm{Bord}(x)$, $\mathrm{Bord}^2(x)$, …, $\mathrm{Bord}^k(x)$.

(2) Soit z un bord de xa. Si $z \neq \varepsilon$, alors $z = z'a$, où z' est un bord de x. Donc, par (1), z est un des mots de l'ensemble

$$B = \{\mathrm{Bord}(x)a, \mathrm{Bord}^2(x)a, \ldots, \mathrm{Bord}^k(x)a, \varepsilon\}$$

Réciproquement, tout mot de B est suffixe de xa, donc est un bord de xa s'il est préfixe de xa. ∎

On déduit de la proposition la caractérisation suivante du plus long bord :

Corollaire 1.4. *Soit x un mot non vide et soit a une lettre. Alors*

$$\mathrm{Bord}(xa) = \begin{cases} \mathrm{Bord}(x)a & \text{si } \mathrm{Bord}(x)a \text{ est préfixe de } x, \\ \mathrm{Bord}(\mathrm{Bord}(x)a) & \text{sinon.} \end{cases}$$

Preuve. Si $\mathrm{Bord}(x)a$ est préfixe de x, alors $\mathrm{Bord}(xa) = \mathrm{Bord}(x)a$. Dans le cas contraire, posons $y = \mathrm{Bord}(x)$. Alors $\mathrm{Bord}(xa)$ est préfixe de y, et par le (2) de la proposition 1.3, $\mathrm{Bord}(xa)$ est le plus long préfixe de x, donc de y, dans l'ensemble $\{\mathrm{Bord}(y)a, \mathrm{Bord}^2(y)a, \ldots, \mathrm{Bord}^{k-1}(y)a, \varepsilon\}$. A nouveau par 1.3(2), cela signifie que $\mathrm{Bord}(xa) = \mathrm{Bord}(ya)$. ∎

Voici quelques exemples :

$x = ababb$	$\mathrm{Bord}(x) = \varepsilon$	$\mathrm{Bord}(xa) = \mathrm{Bord}(x)a$
$x = babbaa$	$\mathrm{Bord}(x) = \varepsilon$	$\mathrm{Bord}(xa) = \mathrm{Bord}(a) = \varepsilon$
$x = abaaab$	$\mathrm{Bord}(x) = ab$	$\mathrm{Bord}(xa) = \mathrm{Bord}(x)a = aba$
$x = abbaab$	$\mathrm{Bord}(x) = ab$	$\mathrm{Bord}(xa) = \mathrm{Bord}(aba) = a$

Un autre corollaire de la proposition 1.3 indique comment calculer efficacement la fonction β. On a en effet :

Corollaire 1.5. *Soit x un mot de longueur m. Pour $j = 0, \ldots, m-1$, on a*

$$\beta(1+j) = 1 + \beta^k(j)$$

où $k \geq 1$ est le plus petit entier tel que l'une des deux conditions suivantes est vérifiée

(*i*) $1 + \beta^k(j) = 0$;
(*ii*) $1 + \beta^k(j) \neq 0$ *et* $x_{1+\beta^k(j)} = x_{1+j}$. ∎

Ce corollaire se traduit en une procédure qui calcule la fonction β pour un mot x, sous la forme d'un tableau :

```
procédure BORDS-MAXIMAUX(x, β);
  β[0] := −1;
  pour j de 1 à m faire
    i := β[j − 1];
    tantque i ≥ 0 etalors x[j] ≠ x[i + 1] faire i := β[i] fintantque;
    β[j] := i + 1
  finpour.
```

Une procédure tout à fait semblable calcule la fonction de suppléance :

```
procédure SUPPLÉANCE(x, s);
  s[1] := 0;
  pour j de 1 à m − 1 faire
    i := s[j];
    tantque i > 0 etalors x[j] ≠ x[i] faire i := s[i] fintantque;
    s[j + 1] := i + 1
  finpour.
```

Une preuve analogue à celle de la proposition 1.2 montre que le calcul de la fonction β d'un motif de longueur m se fait en $2m-3$ comparaisons de caractères.

Corollaire 1.6. *La recherche d'une occurrence d'un motif x de longueur m dans un texte t de longueur n par l'algorithme de Morris et Pratt demande au plus $2(n+m)-4$ comparaisons de caractères.* ∎

10.1.4 L'algorithme de Knuth, Morris et Pratt

L'algorithme de Knuth, Morris et Pratt que nous présentons maintenant est une amélioration de l'algorithme précédent, basée sur l'élimination de situations qu'il est inutile d'examiner.

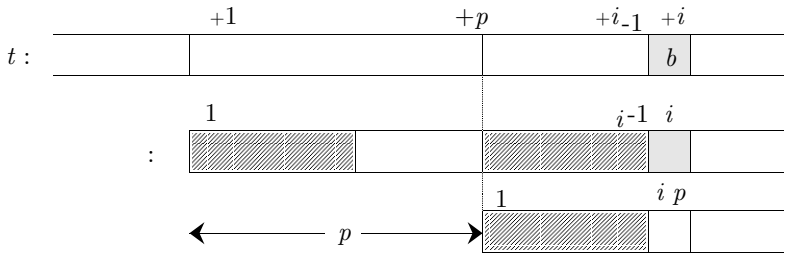

Figure 1.5: *Lorsque $b \neq a$, le décalage est inutile si $c = a$.*

Revenons à la configuration où une recherche du motif $x = x_1 \cdots x_m$ dans un texte $t = t_1 \cdots t_n$ a échoué parce que

$$x_1 \cdots x_{i-1} = t_{k+1} \cdots t_{k+i-1} \qquad \text{et} \qquad x_i \neq t_{k+i}$$

Le décalage proposé par l'algorithme de Morris et Pratt est déterminé par la longueur du bord $\mathrm{Bord}(x_1 \cdots x_{i-1})$; la nouvelle position du motif débute à la lettre d'indice $p+1$, avec $p = i - 1 - \beta(i-1)$ (voir figure 1.5). Ce décalage n'est utile que si l'on est assuré de pouvoir réellement progresser, c'est-à-dire si la lettre t_{k+i} est égale à la lettre de x qui va lui être comparée, à savoir la lettre x_{i-p} ; sinon, il faut chercher un autre bord. Or, cette condition ne peut pas être traduite en une condition sur le mot x seul, donc ne peut pas être incluse dans le prétraitement de x. Mais on peut la remplacer par une condition plus faible et exiger de ne pas se retrouver dans la même situation que précédemment, à savoir que

$$x_{i-p} \neq x_i.$$

C'est cette condition supplémentaire qui est testée dans l'algorithme de Knuth, Morris et Pratt. Pour la mettre en œuvre, on définit une fonction analogue à la fonction β de Morris et Pratt, et qui va tenir compte de cette condition. Auparavant, introduisons une définition.

Soit $x = x_1 \cdots x_m$. Deux préfixes u et v de x sont *disjoints* (dans x) si $x_{1+|u|} \neq x_{1+|v|}$ ou si l'un des deux mots est x tout entier. Le préfixe u est un *bord disjoint* du préfixe v si u est un bord de v, et si u et v sont des préfixes disjoints de x. Si v possède un bord disjoint dans x, on note $\mathrm{DBord}(v)$ le plus long bord disjoint de v, appelé *bord disjoint maximal* de v. Contrairement à la notion de bord, le concept de bord disjoint n'est défini que sur les préfixes d'un mot x.

Exemple. Soit $x = abcababcac$. Le préfixe $v = abcababca$ de x possède les trois bords ε, a, $abca$ qui sont tous les trois disjoints de v. Les préfixes ab et $abcabab$ de x ne sont pas disjoints car ils sont tous les deux suivis de la lettre c. Le préfixe $w = abcababc$ enfin a les deux bords ε et abc, dont aucun n'est disjoint de w; il n'a donc pas de bord maximal disjoint.

On définit la fonction

$$\gamma = \gamma_x : \{0, \ldots, m\} \to \{-1, \ldots, m-1\}$$

par $\gamma(0) = -1$ et, pour $j = 1, \ldots, m$,

$$\gamma(j) = \begin{cases} |\mathrm{DBord}(x_1 \cdots x_j)| & \text{si } x_1 \cdots x_j \text{ a un bord disjoint dans } x; \\ -1 & \text{sinon.} \end{cases}$$

Exemple. Voici les fonctions β et γ tabulées pour le motif $x = abcababcac$. Leur comparaison illustre le gain que l'on peut attendre de l'usage de γ à la place de β.

		a	b	c	a	b	a	b	c	a	c
j	0	1	2	3	4	5	6	7	8	9	10
$\beta(j)$	-1	0	0	0	1	2	1	2	3	4	0
$\gamma(j)$	-1	0	0	-1	0	2	0	0	-1	4	0

Soit v un préfixe du mot x. Les bords de v sont, par la proposition 1.3, les mots $\mathrm{Bord}(v), \mathrm{Bord}^2(v), \ldots, \mathrm{Bord}^k(v), \varepsilon$, où k est le plus grand entier tel que $\mathrm{Bord}^k(v) \neq \varepsilon$. Les longueurs des bords de v sont donc les nombres $\beta(j)$, $\beta^2(j)$, …, $\beta^k(j)$, 0, avec $j = |v|$. Il en résulte que si $\gamma(j) \neq -1$, alors $\gamma(j) = \beta^d(j)$, où d est le plus petit entier tel que $\mathrm{Bord}^d(v)$ est disjoint de v.

Proposition 1.7. *Soit $x = x_1 \cdots x_m$ un mot non vide; la fonction $\gamma = \gamma_x$ vérifie, pour $j \geq 1$*

$$\gamma(j) = \begin{cases} \beta(j) & \text{si } j = m \text{ ou } x_{1+j} \neq x_{1+\beta(j)}, \\ \gamma(\beta(j)) & \text{sinon.} \end{cases}$$

Preuve. Soit $j \geq 1$. Si $j = m$, alors $\gamma(m) = \beta(m)$. Supposons donc $j < m$, et soit $x_1 \cdots x_i$ le bord maximal de $x_1 \cdots x_j$. On a donc $i = \beta(j)$. Si $x_{1+i} \neq x_{1+j}$, alors $\gamma(j) = \beta(j)$. Si $x_{1+i} = x_{1+j}$, les bords disjoints de $x_1 \cdots x_i$ sont exactement les bords disjoints de $x_1 \cdots x_j$ car si $x_1 \cdots x_k$ est un tel bord, on a $x_{1+k} \neq x_{1+j}$ si et seulement si $x_{1+k} \neq x_{1+i}$, donc $\gamma(j) = \gamma(i)$. ∎

Cette proposition permet de calculer γ à partir de la fonction β. Réciproquement, β peut s'exprimer en fonction de γ, et la combinaison de ces deux relations permet d'évaluer γ sans calculer préalablement la fonction β.

Proposition 1.8. *Soit x un mot de longueur m. Pour $j = 0, \ldots, m-1$, on a*

$$\beta(1+j) = 1 + \gamma^k(\beta(j))$$

où $k \geq 0$ est le plus petit entier tel que l'une des deux conditions suivantes est vérifiée :

(i) $1 + \gamma^k(\beta(j)) = 0$;

(ii) $1 + \gamma^k(\beta(j)) \neq 0$ *et* $x_{1+\gamma^k(\beta(j))} = x_{1+j}$.

Preuve. Posons $i = \beta(j)$ et $y = x_1 \cdots x_i$, et soit $a = x_{1+i}$, et $b = x_{1+j}$. Si $a = b$, alors $\beta(1+j) = 1 + \beta(j)$, et la formule est vraie avec $k = 0$. Sinon, $\beta(1+j)$ est soit nul, soit l'un des nombres $1 + \beta^r(j)$. Or, parmi les mots y, $\mathrm{Bord}(y), \ldots$, il suffit d'examiner ceux qui sont suivis d'une lettre autre que b. Il suffit donc de chercher le bord maximal *disjoint* de y, dont la longueur est $\gamma(i)$. ∎

En vertu des propositions 1.7 et 1.8, on obtient la procédure que voici pour le calcul de γ. La boucle *tantque* calcule en fait $\beta(j)$:

```
procédure BORDS-DISJOINTS-MAXIMAUX(x, γ);
  γ[0] := −1; i := −1;
  pour j de 1 à m faire          {ici i = β(j − 1)}
    tantque i ≥ 0 etalors x[j] ≠ x[i + 1] faire i := γ[i] fintantque;
    i := i + 1;                  {ici i = β(j)}
    si x[1 + j] ≠ x[1 + i] alors γ[j] := i sinon γ[j] := γ[i]
  finpour.
```

Introduisons une *deuxième fonction de suppléance* r définie par

$$r(i) = 1 + \gamma(i-1)$$

On peut alors calculer r par

```
procédure DEUXIÈME-SUPPLÉANCE(x, r);
  r[1] := 0; i := 0;
  pour j de 1 à m − 1 faire
    tantque i > 0 etalors x[j] ≠ x[i] faire i := r[i] fintantque;
    i := i + 1;
    si x[1 + j] ≠ x[i] alors r[1 + j] := i sinon r[1 + j] := r[i]
  finpour.
```

Avec cette deuxième fonction de suppléance, l'algorithme de Knuth, Morris et Pratt s'écrit exactement comme l'algorithme de Morris et Pratt. La seule modification est le remplacement de s par r. Nous changeons très légèrement sa structure, en remplaçant le double test sur i et j par deux boucles *tantque* imbriquées, ceci pour pouvoir mettre en évidence la notion de délai entre l'examen de deux caractères :

```
procédure KNUTH-MORRIS-PRATT(x, t);
  i := 1; j := 1;
  tantque i ≤ m et j ≤ n faire
    tantque i > 0 etalors t[j] ≠ x[i] faire i := r[i] fintantque;
    i := i + 1; j := j + 1
  fintantque;
  si i > m alors
    occurrence de x à la position j − m
  sinon
    pas d'occurrence de x dans t
  finsi.
```

Revenons sur un exemple précédent, où l'on cherche le motif $x = abacabac$ dans le texte $t = babacacabacaab$. Les deux fonctions de suppléance s et r sont les suivantes :

	a	b	a	c	a	b	a	c
	1	2	3	4	5	6	7	8
s	0	1	1	2	1	2	3	4
r	0	1	0	2	0	1	0	2

La figure 1.6 montre les décalages successifs du motif. Les différences sont constatées aux positions sombres du texte. Le nombre total de comparaisons est 16.

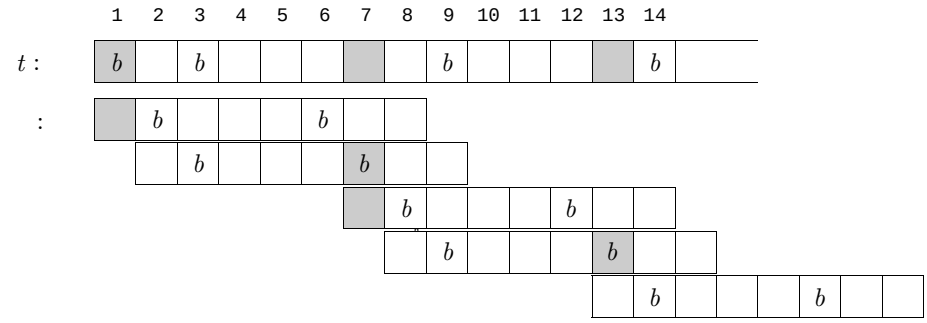

Figure 1.6: *Décalages successifs du motif.*

Le tableau donne, pour chaque lettre, la suite des valeurs que prend l'indice i dans l'algorithme :

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	b	a	b	a	c	a	c	a	b	a	c	a	a	b
i	1	1	2	3	4	5	6	1	2	3	4	5	6	2
	0						1						1	
							0							

Lorsque $j = 7$ et $i = 6$, on a $t_j \neq x_i$, et on compare cette fois-ci t_7 directement à la lettre x_1. La différence entre les deux fonctions de suppléance apparaît bien si on trace leur graphe (figure 1.7). Il est clair que l'algorithme de Knuth, Morris et

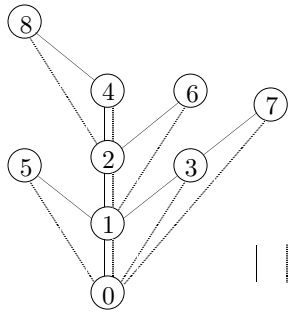

Figure 1.7: *Les graphes des deux fonctions de suppléance.*

Pratt est plus efficace que l'algorithme de Morris et Pratt, même si, dans le cas le plus défavorable, leur complexité est la même. Une différence de comportement notable entre ces deux algorithmes concerne le *délai*, c'est-à-dire le nombre maximum de comparaisons de caractères faites sur *un* caractère du texte à analyser. Dans l'algorithme ci-dessus, c'est le nombre de tours effectués dans la boucle *tant-que* interne. Le délai est le temps que l'on doit attendre avant de pouvoir passer au caractère suivant du texte t, et il mesure donc jusqu'à quel point l'algorithme est différent d'un algorithme *en temps réel*, c'est-à-dire est capable de traiter un symbole par unité de temps. Il a été prouvé que le délai de l'algorithme de Morris et Pratt peut atteindre la longueur m du motif, alors que pour Knuth, Morris et Pratt, il ne dépasse jamais $1 + \log_\phi m$, où $\phi = (1 + \sqrt{5})/2$. En ce sens aussi, l'algorithme de Knuth, Morris et Pratt, sans être plus difficile à programmer, est plus efficace.

10.1.5 L'automate des occurrences

Dans cette section, nous montrons comment l'algorithme de Knuth, Morris et Pratt s'interprète en termes d'automates finis. Soit A l'alphabet sur lequel sont écrits le texte et le motif. Soit $x \in A^*$ le motif dont on cherche à déterminer les

occurrences dans le texte t. Comme x a une occurrence dans t si et seulement si $t \in A^*xA^*$, déterminer les occurrences de x dans t équivaut à trouver les préfixes de t qui appartiennent au langage (rationnel) A^*x. Pour cela, il suffit de construire l'automate reconnaissant A^*x, et de lui faire lire le texte t. Or, un automate reconnaissant A^*x est vite construit. C'est l'automate

$$\mathcal{A} = (P, \varepsilon, x, \mathcal{F})$$

où P est l'ensemble des préfixes de x, l'état initial est le mot vide, l'unique état final est le motif x, et dont les flèches sont

$$\mathcal{F} = \{(\varepsilon, a, \varepsilon) \mid a \in A\} \cup \{(p, a, pa) \mid p, pa \in P, a \in A\}$$

Exemple. Pour $A = \{a, b, c\}$, et pour $x = abcababcac$, l'automate $\mathcal{A}$ est donné dans la figure 1.8, où les états sont représentés par leur longueur.

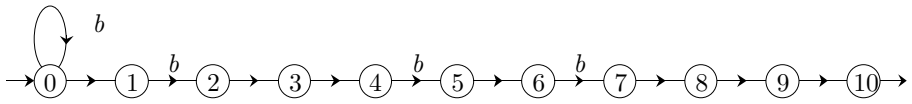

Figure 1.8: *Automate reconnaissant le langage $A^*abcababcac$.*

Dans la mesure où cet automate n'est pas déterministe, il n'est pas facilement exploitable. Nous allons voir que l'automate minimal déterministe reconnaissant A^*x n'est pas difficile à calculer et a autant d'états que l'automate ci-dessus, à savoir $1 + m$, où m est la longueur de x. Pour le caractériser, nous considérons la fonction f_x qui à tout mot u associe

$$f_x(u) = \text{ le plus long suffixe de } u \text{ qui est préfixe de } x$$

Par exemple, pour $x = abcababcac$, on a $f_x(abacababc) = abc$; si p est préfixe de x, on a évidemment $f_x(p) = p$.

Proposition 1.9. *Soit x un mot et soit P l'ensemble de ses préfixes. L'automate minimal reconnaissant A^*x est l'automate déterministe $\mathcal{A}(x) = (P, \varepsilon, x)$ dont la fonction de transition est définie par $p \cdot a = f_x(pa)$.*

Preuve. Nous allons vérifier que pour tout $u \in A^*$,

$$u^{-1}(A^*x) = f_x(u)^{-1}(A^*x)$$

Ceci prouve que les états de l'automate minimal s'identifient aux préfixes de x, et que la fonction de transition est bien celle indiquée. Soit donc $u \in A^*$, et soit u' tel que $u = u' f_x(u)$. On a

$$u^{-1}(A^*x) = f_x(u)^{-1}u'^{-1}(A^*x) \supset f_x(u)^{-1}(A^*x)$$

Pour prouver l'inclusion réciproque, soit $w \in u^{-1}(A^*x)$. Alors $uw \in A^*x$, et il existe donc un mot v tel que $uw = vx$. Si x est suffixe de w, alors $w \in A^*x$, donc $f_x(u)w \in A^*x$ et $w \in f_x(u)^{-1}(A^*x)$. Si en revanche w est suffixe de x, appelons z le mot tel que $x = zw$. Alors on a aussi $u = vz$, donc z est suffixe de u et préfixe de x. Par définition de $f_x(u)$, il existe y tel que $f_x(u) = yz$. Mais alors $f_x(u)w = yzw = yx$, montrant que $w \in f_x(u)^{-1}(A^*x)$. Ceci prouve l'inclusion et achève la démonstration. ∎

L'automate $\mathcal{A}(x)$ est l'*automate des occurrences*. On obtient immédiatement l'algorithme suivant de recherche de motifs, où le texte $t = t_1 \cdots t_n$ est de longueur n. Dans la mesure où l'automate des occurrences $\mathcal{A}(x)$ est disponible et rangé dans une table (de taille $O(|A|(|x|+1))$), le calcul de l'état suivant se fait en temps constant, et le temps d'exécution de l'algorithme est $O(n)$ pour un texte de longueur n.

```
procédure RECHERCHE-AUTOMATE(x, t);
  q :=état initial;
  pour j de 1 à n faire
    q := q · t[j];
    si q est état final alors j est une fin d'occurrence finsi
  finpour.
```

L'automate $\mathcal{A}(x)$ pour $x = abcababcac$ est donné dans la figure 1.9. Les états sont représentés par leurs longueurs. L'expression donnée ci-dessous fait le lien avec

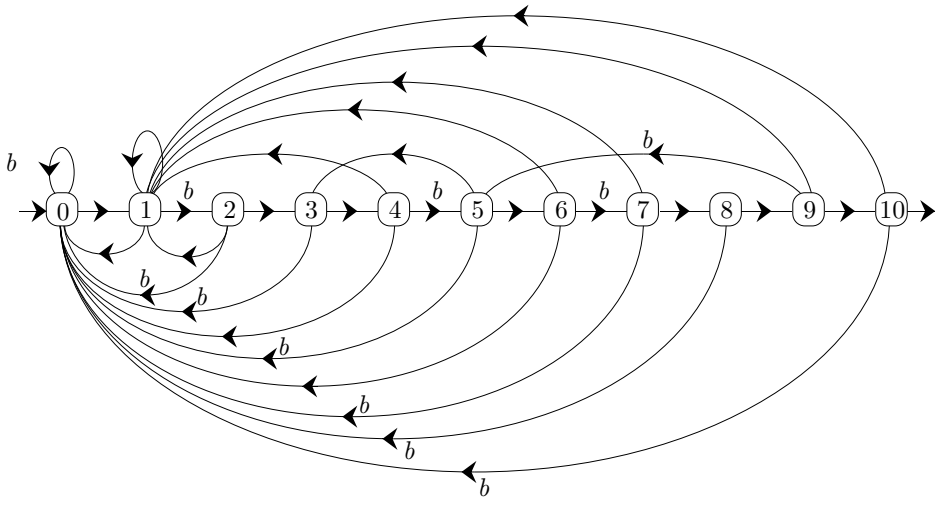

Figure 1.9: *Automate déterministe* $\mathcal{A}(abcababcac)$.

l'algorithme de Morris et Pratt.

Proposition 1.10. *La fonction de transition de l'automate des occurrences $\mathcal{A}(x)$ vérifie, pour tout préfixe p non vide de x et toute lettre a,*

$$p \cdot a = \begin{cases} pa & \textit{si } pa \textit{ est préfixe de } x\,; \\ \mathrm{Bord}(pa) & \textit{sinon.} \end{cases}$$

Preuve. Si pa est préfixe de x, alors $f_x(pa) = pa$; sinon, $f_x(pa)$ est le plus long suffixe de pa qui est préfixe de x, donc préfixe de pa, c'est-à-dire Bord(pa). ∎

Le calcul de la fonction de transition de l'automate des occurrences est en fait assez facile, et est réalisable en temps linéaire. Ceci résulte du corollaire suivant :

Corollaire 1.11. *La fonction de transition de l'automate des occurrences $\mathcal{A}(x)$ vérifie, pour tout préfixe p non vide de x et toute lettre a*

$$p \cdot a = \begin{cases} pa & \textit{si } pa \textit{ est préfixe de } x\,; \\ \mathrm{Bord}(p) \cdot a & \textit{sinon.} \end{cases}$$

Preuve. L'énoncé résulte immédiatement de la proposition si pa est préfixe de x. Si pa n'est pas préfixe de x, alors en vertu de 1.4

$$p \cdot a = \mathrm{Bord}(pa) = \begin{cases} \mathrm{Bord}(p)a & \text{si Bord}(p)a \text{ est préfixe de } p, \\ \mathrm{Bord}(\mathrm{Bord}(p)a) & \text{sinon,} \end{cases}$$

donc $\mathrm{Bord}(pa) = \mathrm{Bord}(p) \cdot a$. ∎

Corollaire 1.12. *La fonction de transition de l'automate des occurrences $\mathcal{A}(x)$ vérifie, pour tout préfixe p de x et toute lettre a*

$$p \cdot a = \begin{cases} pa & \textit{si } pa \textit{ est préfixe de } x\,; \\ \mathrm{DBord}(p) \cdot a & \textit{si } \mathrm{DBord}(p) \textit{ existe;} \\ \varepsilon & \textit{sinon.} \end{cases}$$

Preuve. Si $p \neq \varepsilon$ et pa n'est pas préfixe de x, alors $p.a = \mathrm{Bord}(pa)$. Si $\mathrm{Bord}(pa) \neq \varepsilon$, alors $\mathrm{Bord}(pa) = qa$, où q est le plus long bord de p tel que qa est préfixe de x. Comme pa n'est pas préfixe de x, le mot q est *un* bord disjoint de p; d'où la formule par récurrence. ∎

Exemple. Pour l'automate de la figure 1.9, les fonctions β et γ prennent les valeurs suivantes :

		a	b	c	a	b	a	b	c	a	c
	0	1	2	3	4	5	6	7	8	9	10
β	-1	0	0	0	1	2	1	2	3	4	0
γ	-1	0	0	-1	0	2	0	0	-1	4	0

On en déduit par exemple que $5 \cdot b = 2 \cdot b = 0$, et de façon similaire $9 \cdot a = 4 \cdot a (= 1 \cdot a) = 0 \cdot a = 1$, selon que l'on utilise la formule du premier ou du deuxième corollaire. Enfin, $3 \cdot b = 3 \cdot c = 0$.

La recherche d'un motif x avec un automate fini se fait en *temps réel*. La fonction de transition de l'automate se calcule, à partir de β ou de γ, en temps linéaire, c'est-à-dire en temps $O(|x| \cdot |A|)$, où A est l'alphabet de base. L'inconvénient majeur est l'encombrement de l'automate : sa taille, qui est également $O(|A|(|x|+1))$, devient prohibitive si A est par exemple l'alphabet des 256 caractères Ascii étendus.

10.1.6 L'algorithme de Simon

I. Simon a proposé un algorithme qui est un compromis entre l'algorithme de Knuth, Morris et Pratt, et l'implémentation par automate. Simon part de l'automate minimal $\mathcal{A}(x)$ associé à un motif x de longueur m, et remarque qu'il n'y a que peu de flèches «significatives» : il y a d'abord les flèches «avant», qui font passer d'un état i à l'état $i+1$, et les flèches «arrière», qui font passer d'un état i à un état $j < i$, avec $j \neq 0$. Les autres flèches mènent à l'état 0, et ne sont pas «significatives». Nous prouverons que les flèches significatives (nous dirons actives) sont en nombre au plus $2m$, et si l'on ne stocke que celles-ci, on est ramené à un algorithme en place linéaire, indépendamment de la taille de l'alphabet. En contrepartie, la recherche de la «bonne» flèche pour effectuer une transition ne se fait plus en temps constant (donc l'algorithme n'est pas en temps réel), mais le rangement des transitions peut être organisé de manière à rendre cette recherche toujours au moins aussi efficace que dans l'algorithme de Knuth, Morris et Pratt.

Soit donc $x \in A^*$ un motif de longueur m, et soit $\mathcal{A}(x) = (P, \varepsilon, x)$ l'automate minimal reconnaissant A^*x. On identifiera souvent un état de P, c'est-à-dire un préfixe p de x, à sa longueur. Une flèche (p, a, q) de l'automate est une flèche *active* si $q \neq \varepsilon$, elle est *passive* si $q = \varepsilon$. Une flèche active (p, a, q) est une flèche *avant* si $q = pa$, c'est une flèche *arrière* si $q \neq pa$ (et $q \neq \varepsilon$).

Exemple. La figure 1.10 donne l'automate de la figure 1.9, où l'on n'a conservé que les flèches actives. Il y a 9 flèches arrière.

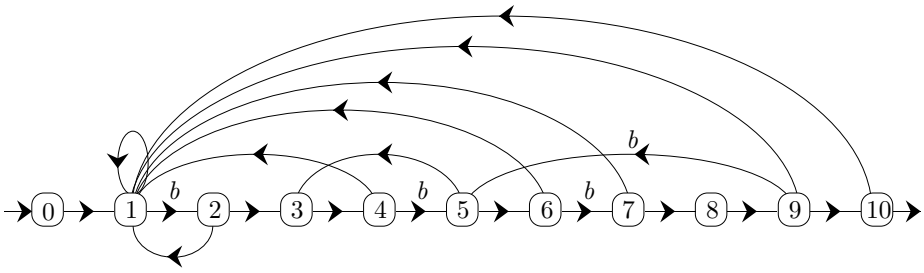

Figure 1.10: *Automate $\mathcal{A}(abcababcac)$, sans ses flèches passives.*

Le motif x étant de longueur m, l'automate $\mathcal{A}(x)$ a m flèches avant. Nous allons prouver qu'il a au plus m flèches arrière. Pour cela, nous avons besoin d'une définition qui a par ailleurs son intérêt propre. Soit $w = a_1 \cdots a_n$ un mot, avec $a_1, \ldots, a_n$ des lettres. Une *période* de w est un entier $p > 0$ tel que

$$a_i = a_{p+i} \quad \text{pour tout } i = 1, \ldots, n-p$$

Notons que $|w|$ est toujours une période de w. Par exemple, le mot $w = abcababca$, de longueur 9, a les périodes 9, 8, 5. On peut voir une période comme un «décalage» qui conserve la coïncidence des lettres qui se superposent. Plus précisément, on a le lemme suivant :

Lemme 1.13. *Soit w un mot de longueur n, et soit $p > 0$. Alors p est une période de w si et seulement si w possède un bord de longueur $n - p$.*

Preuve. Soit $w = a_1 \cdots a_n$. Alors p est une période si et seulement si

$$a_1 \cdots a_{n-p} = a_{1+p} \cdots a_n,$$

donc si et seulement si le préfixe de longueur $n - p$ de w est aussi suffixe de w. ∎

Notons pér(w) la plus petite période de w. Il résulte du lemme 1.13 que

$$\text{pér}(w) + \beta(w) = |w| \tag{1.1}$$

où $\beta(w)$ est la longueur du plus long bord de w. Revenons à l'automate $\mathcal{A}(x)$.

Proposition 1.14. *L'automate $\mathcal{A}(x)$ a au plus $|x|$ flèches arrière.*

Preuve. Nous allons d'abord prouver l'assertion que voici : si (p, a, q) et (p', a', q') sont deux flèches arrière distinctes, alors pér$(pa) \neq$ pér$(p'a')$.

Soient en effet (p, a, q) et (p', a', q') deux flèches arrière. Par la proposition 1.10, on a $q = \text{Bord}(pa)$, $q' = \text{Bord}(p'a')$, et $q \neq \varepsilon$, $q' \neq \varepsilon$. Supposons, en raisonnant par l'absurde, que pa et $p'a'$ ont même période, disons $t =$ pér(pa) et, pour fixer les idées, supposons $|p| \leq |p'|$. Si $k = |pa|$, on a $t < k$ par l'équation 1.1. Notons b la lettre d'indice k dans $p'a'$. Alors $a \neq b$. Ceci est clair si $|p'| > |p|$ parce que p' est préfixe de x, alors que pa ne l'est pas. C'est vrai aussi si $|p'| = |p|$ parce qu'alors $b = a' \neq a$. Comme t est une période de pa, les lettres d'indice k et $k - t$ de pa sont les mêmes. Comme p est préfixe de x, la lettre d'indice $k - t$ de pa est x_{k-t}, donc $x_{k-t} = a$. Or t est aussi une période de $p'a'$, et donc les lettres d'indice k et $k - t$ de $p'a'$ sont les mêmes. On a donc aussi $x_{k-t} = b$, ce qui est impossible puisque $a \neq b$. Ceci prouve l'assertion.

Il résulte de l'assertion que l'application qui à une flèche arrière $(p, a, \text{Bord}(pa))$ associe pér(pa) est injective. Or pér$(pa) = |pa| - \text{Bord}(pa) < |pa|$ d'après l'équation 1.1, donc $1 \leq$ pér$(pa) \leq |x|$, ce qui montre la proposition. ∎

L'*implémentation de Simon* de l'automate $\mathcal{A}(x)$ consiste à associer, à chaque état p, la liste des flèches avant et arrière issues de p, ordonnées par numéro décroissant d'état d'arrivée. Chaque flèche $u = (p, a, q)$, pour p fixé, est représentée par un couple $(\mathit{lettre}(u), \mathit{état}(u)) = (a, q)$. Avec cette structure de données, le calcul de l'état suivant se fait par la fonction :

```
fonction ETAT-SUIVANT-PAR-SIMON(p, a);
  u :=tête-liste(p);
  tantque u ≠ vide faire
    si lettre(u) = a alors retourner état(u)
    sinon u := suivant(u)
  fintantque;
  retourner(0).
```

et l'algorithme de recherche du motif x, de longueur m, dans un texte t de longueur n est, comme pour tout automate (voir la procédure RECHERCHE-AUTOMATE) :

```
procédure SIMON(x, t);
  p := 0;
  pour j de 1 à n faire
    p :=ETAT-SUIVANT-PAR-SIMON(p, t[j]);
    si p = m alors j est une fin d'occurrence
  finpour.
```

Exemple. L'implémentation de l'automate $\mathcal{A}(abcababcac)$ est représentée dans la figure 1.11 ; les préfixes sont représentés par leur longueur.

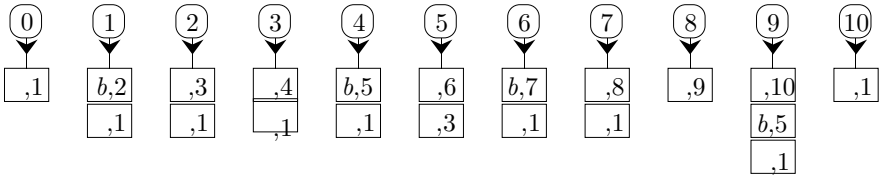

Figure 1.11: *Implémentation de l'automate* $\mathcal{A}(abcababcac)$.

Proposition 1.15. *L'algorithme de Simon calcule les occurrences d'un motif x dans un texte t en un nombre de comparaisons de caractères inférieur ou égal à celui de l'algorithme de Knuth, Morris et Pratt.*

Preuve. Soit p un état, et soit a une lettre. Le nombre de comparaisons faites par l'algorithme de Simon pour déterminer l'état suivant est égal au nombre de comparaisons faites dans la liste associée à l'état p. La première comparaison concerne la flèche avant, les suivantes des flèches arrière ; si toutes les comparaisons sont négatives, c'est une flèche passive qui est utilisée.

Soit $(p, b, p \cdot b)$ une flèche arrière. Alors par le corollaire 1.11, $p \cdot b = qb$, et q est un bord disjoint de p. En d'autres termes, si

$$(p, a_1, q_1a_1), \ldots, (p, a_k, q_ka_k)$$

est la suite ordonnée des flèches arrière, les états $q_1, \ldots, q_k$ sont des bords disjoints de p, et mutuellement disjoints. Ils figurent donc dans l'ensemble

$$\{\mathrm{DBord}(p), \mathrm{DBord}^2(p), \ldots\}$$

Comme les états sont rangés en ordre décroissant, ces bords disjoints sont calculés successivement dans l'algorithme de Knuth, Morris et Pratt, alors que l'algorithme de Simon n'en sélectionne qu'un sous-ensemble. ∎

Il reste à préciser comment réaliser, en temps linéaire, l'implémentation de Simon, c'est-à-dire le calcul de la suite ordonnée des flèches actives pour chaque état. Notons $F(p)$ la suite ordonnée des flèches actives issues de l'état p ; chaque flèche est représentée par le couple (lettre, état d'arrivée), comme dans la figure 1.11.

Soit $x = x_1 \cdots x_m$ un motif. Pour l'état initial ε de l'automate $\mathcal{A}(x)$, la suite $F(\varepsilon)$ des flèches actives est réduite à l'élément $(x_1, 1)$. Soit p un autre état, et soit $i = |p|$ sa longueur. La suite $F(p)$ contient tout d'abord la flèche $(x_{i+1}, i+1)$, sauf lorsque $i = m$. Pour déterminer les autres flèches, nous utilisons le corollaire 1.12.

Si $\mathrm{DBord}(p)$ existe, c'est-à-dire si $\gamma(i) \neq -1$, alors on a $p \cdot a = \mathrm{DBord}(p) \cdot a$ pour toute lettre $a \neq x_{i+1}$. Ainsi, la suite $F(p)$ se prolonge par la suite $F(\mathrm{DBord}(p))$ des flèches actives de $\mathrm{DBord}(p)$, purgée de la flèche dont la lettre est x_i, si cette flèche y figure.

Si en revanche $\mathrm{DBord}(p)$ n'existe pas, la suite $F(p)$ est réduite à son premier élément.

Il est clair que la copie d'une liste, avec purge éventuelle d'un élément, se fait en temps linéaire. L'implémentation de Simon se calcule donc en temps linéaire, puisque la fonction γ se calcule, elle aussi, en temps linéaire.

Exemple. La fonction γ_x pour $x = abcababcac$ a déjà été donnée plus haut. Ses valeurs sont :

		a	b	c	a	b	a	b	c	a	c
j	0	1	2	3	4	5	6	7	8	9	10
$\gamma(j)$	-1	0	0	-1	0	2	0	0	-1	4	0

La liste $F(8)$ est réduite à $(a, 9)$ parce que $\gamma(8) = -1$; la liste $F(9)$ est la concaténation de $(c, 10)$ et de $F(4)$; enfin, la liste $F(5)$ est composée de $(a, 6)$ et de la liste $F(2)$, purgée de son élément $(a, 1)$, donc réduite à $(c, 3)$.

10.2 L'algorithme de Boyer et Moore

L'algorithme de Boyer et Moore se rattache au schéma général des algorithmes de recherche que nous avons exposé au début de la section précédente : on compare le motif x à certains facteurs du texte t, en faisant glisser x de gauche à droite le long du texte t.

La différence principale entre l'algorithme de Boyer et Moore et les algorithmes précédents réside dans la manière de comparer le motif x aux facteurs du texte. Alors que l'algorithme naïf et l'algorithme de Knuth, Morris et Pratt comparent les lettres du motif x aux lettres du facteur de t de la gauche vers la droite, l'algorithme de Boyer et Moore les compare de la droite vers la gauche. Cette modification apparemment anodine est très rentable puisqu'en pratique, l'algorithme de Boyer et Moore est le plus rapide des algorithmes connus.

10.2.1 Algorithme de Horspool

Dans sa version la plus simple, l'algorithme (appelé algorithme de Boyer-Moore-Horspool) compare le motif $x = x_1 \cdots x_m$ à un facteur $t_{k+1} \cdots t_{k+m}$ du texte. Si une différence entre x et ce facteur est constatée, on décale le motif vers la droite de manière à faire coïncider la dernière lettre du facteur, c'est-à-dire la lettre t_{k+m}, avec son occurrence la plus à droite dans x.

Exemple. La figure 2.1 montre le fonctionnement de l'algorithme de Boyer-Moore-Horspool sur le texte $t = aabbbababacaabbaba \cdots$ et le motif $x = aababab$. Les différences sont constatées aux positions 4, 11, 18 du texte (cases sombres du

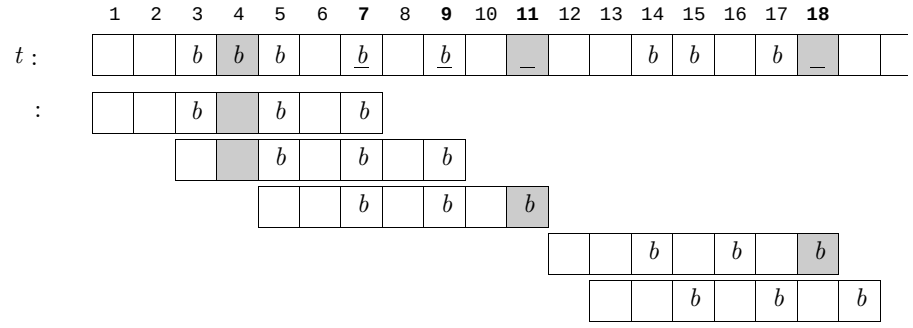

Figure 2.1: *Algorithme de Horspool.*

texte et de l'occurrence du motif). Les lettres qui déterminent le décalage sont soulignées (les indices correspondants sont gras). La première différence constatée conduit à décaler le motif de 2 vers la droite pour faire coïncider la lettre b à la cinquième position dans x avec la septième lettre du texte. La deuxième différence entraîne à nouveau un décalage de 2. La troisième différence constatée doit entraîner un décalage pour faire coïncider la lettre c avec une lettre du motif.

Comme il n'y a pas d'occurrence de la lettre c dans le motif, le motif est décalé de toute sa longueur. Le dernier décalage est d'une unité, parce qu'il y a une occurrence de a en avant-dernière position dans x.

Soit x un mot de longueur m. Pour toute lettre a de l'alphabet, soit $d(a)$ la distance entre la dernière occurrence de a dans x (la dernière place exceptée) et la dernière lettre de x. Plus précisément

$$d(a) = \begin{cases} |u| & \text{si } au \text{ est suffixe de } x,\ u \neq \varepsilon \text{ et } a \text{ ne figure pas dans } u, \\ |x| & \text{si } a \text{ ne figure pas dans } x. \end{cases}$$

La fonction d est la *fonction de dernière occurrence*. Par exemple, pour le mot $x = aababab$, la fonction de dernière occurrence vaut :

	a	b	c
d	1	2	7

L'algorithme esquissé plus haut est le suivant (x est un mot de longueur m, et t un texte de longueur n). La version que nous donnons calcule toutes les occurrences de x dans t :

```
Algorithme BOYER-MOORE-HORSPOOL(x, t);
  j := m;
  tantque j ≤ n faire
    i := m;
    tantque i > 0 etalors t_{j-m+i} = x_i faire i := i − 1 fintantque;
    si i = 0 alors
      j est une fin d'occurrence de x;
      j := j + 1
    sinon
      j := j + d[t_j]
    finsi
  fintantque.
```

Sur notre exemple, les valeurs successives que prend l'indice j dans la boucle externe sont 7, 9, 11, 18, 19. Le nombre de comparaisons de caractères n'est que 12.

Cet exemple illustre l'efficacité des algorithmes du type Boyer et Moore : en pratique, on constate que le nombre total de comparaisons est très souvent inférieur à la longueur du texte. En d'autres termes, ces algorithmes n'examinent même pas tous les caractères, par opposition à tous les algorithmes opérant de la gauche vers la droite qui, eux, examinent au moins une fois chaque caractère. En revanche, il n'est pas difficile de voir que l'algorithme est en temps $O(|x||t|)$ dans le cas le plus défavorable. Considérons le motif $x = ba^{m-1}$ dont on cherche une occurrence

dans a^n. La fonction de décalage n'apporte pas d'amélioration puisque $d(a) = 1$. On fait donc $n - m$ décalages, et chaque test demande (s'il est fait de droite à gauche) m comparaisons de caractères.

Le comportement en moyenne de l'algorithme a fait l'objet d'études expérimentales et est tout à fait excellent. On peut prouver, mais cela dépasse le cadre de ce texte, que l'algorithme est sous-linéaire en moyenne (voir les notes).

La comparaison entre x et $t_{j-m+1} \cdots t_j$ est faite de droite à gauche, mais on pourrait aussi bien la faire de gauche à droite; ce qui importe c'est le calcul du décalage en fonction de la dernière lettre du facteur.

Il reste à calculer la fonction de dernière occurrence d pour le motif x. Cela se fait très simplement comme suit :

```
procédure DERNIÈRE-OCCURRENCE (x, d);
   pour toute lettre a de A faire d[a] := m finpour;
   pour i de 1 à m − 1 faire d[x_i] := m − i finpour.
```

Bien entendu, ce calcul de d peut être inclus au début de la procédure pour l'algorithme de Boyer, Moore et Horspool.

10.2.2 Algorithme de Boyer et Moore

La fonction de dernière occurrence peut être employée différemment : alors que dans la version de Horspool, le décalage est déterminé par la dernière lettre du facteur examiné, on peut utiliser la lettre du facteur où la différence a été constatée. Ainsi, la superposition du motif $x = aababab$ au facteur $aabcbab$ conduit, dans Horspool, à un décalage de 2 à cause de la lettre b terminant le facteur, et ceci indépendamment du fait que c'est à l'occurrence de la lettre c dans le facteur que la différence a été constatée. Dans une variante de l'algorithme de Boyer et Moore (pour être historiquement exact, c'est Horspool qui a introduit une variante de l'algorithme de Boyer et Moore), c'est la lettre où la différence se manifeste qui détermine le décalage. Plus précisément, lorsqu'une coïncidence partielle du texte et du motif est constatée :

$$x_i \neq t_j, \quad x_{i+1} \cdots x_m = t_{j+1} \cdots t_{m+j-i}$$

on décale le motif pour faire coïncider la lettre t_j avec $x_{d(t_j)}$, c'est-à-dire avec la dernière occurrence de t_j dans x, puis on recommence les comparaisons sur le nouveau facteur du texte; toutefois, ce décalage n'est fait que si cela fait réellement avancer le motif, c'est-à-dire lorsque $d(t_j) > m - i$; sinon, on décale le motif de 1. En d'autres termes, la comparaison est reprise pour les nouvelles valeurs

$$j := j + \max(d(t_j), m - i + 1); \ i := m$$

Voici l'algorithme obtenu :

```
Algorithme BOYER-MOORE-SIMPLIFIÉ(x, t);
  j := m;
  tantque j ≤ n faire
    i := m;
    tantque i > 0 etalors t_j = x_i faire
      i := i − 1; j := j − 1 fintantque;
    si i = 0 alors fin d'une occurrence de x en j;
    j := j + max(d[t_j], m − i + 1);
  fintantque.
```

Exemple. Reprenons l'exemple précédent de la recherche du motif $x = aababab$ dans le texte $t = aabbbababacaabbaba\cdots$ avec l'algorithme de Boyer-Moore simplifié. La figure 2.2 montre les décalages successifs du motif. Les différences sont

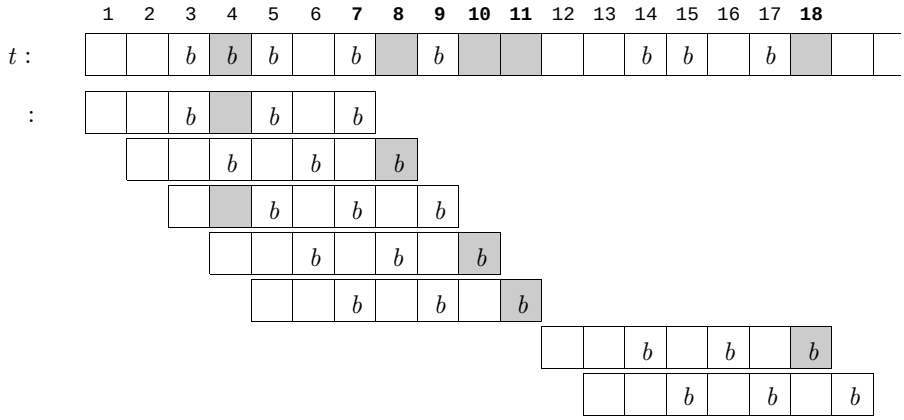

Figure 2.2: *Algorithme de Boyer-Moore simplifié.*

constatées aux positions sombres du texte. Les valeurs successives que prend l'indice j en début de la boucle *tantque* extérieure sont indiquées en gras. Le nombre total de comparaisons de caractères est 14.

La version complète de l'algorithme de Boyer et Moore fait intervenir, en plus de la fonction de dernière occurrence d, une deuxième fonction qui prend en compte le suffixe où la différence entre le motif et le texte a été constatée. La fonction de dernière occurrence est employée de la façon que nous venons d'indiquer. Avant la description formelle, donnons une description sommaire de la deuxième fonction de décalage. Supposons que la comparaison, de droite à gauche, du motif $x = x_1 \cdots x_m$ à un facteur du texte t ait mis en évidence une coïncidence avec un suffixe propre $u = x_{i+1} \cdots x_m$ de x, mais que la lettre x_i soit différente de la lettre correspondante t_j du texte. On a donc (voir figure 2.3) :

$$x_i \neq t_j, \quad x_{i+1} \cdots x_m = t_{j+1} \cdots t_{j+m-i}$$

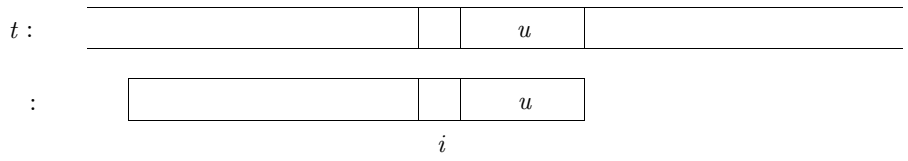

Figure 2.3: *Coïncidence partielle du motif et du texte.*

Un décalage qui tient compte de cette information va aligner x sur la dernière occurrence de u dans x qui est précédée d'une lettre différente de x_i (voir figure 2.4). Le décalage (noté $d_2(i)$ dans la figure) est la quantité dont l'indice j va

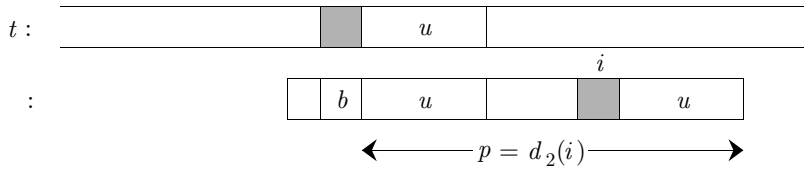

Figure 2.4: *Décalage : premier cas.*

être augmentée; il est égal à la longueur p du plus court suffixe v de x qui a u comme bord, et qui n'est pas précédé de la lettre x_i. Il se peut que x n'ait pas

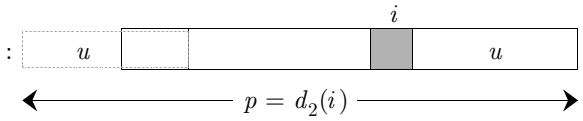

Figure 2.5: *Décalage : deuxième cas.*

d'occurrence du suffixe u précédée d'une lettre différente de x_i. Dans ce cas, on cherche le plus long suffixe de u qui est un préfixe de x; le décalage (encore noté $d_2(i)$) dépasse alors $|x|$ (voir figure 2.5). Cette situation se produit en particulier si $x = u$. Dans ce cas, $i = 0$ et $d_2(0) = |x| + |\mathrm{Bord}(x)|$.

Il est commode d'appeler, par analogie avec la notion de bord disjoint, *bord disjoint droit* d'un suffixe v de x un bord u de v tel que les suffixes u et v ne sont pas précédés de la même lettre dans x. Pour tout suffixe u de x on pose

$$V(u) = \{v \mid v \text{ est suffixe de } x, \ \ u \text{ est bord disjoint droit de } v\}$$

et

$$W(u) = \{w \mid x \text{ est suffixe de } w, \ \ u \text{ est bord de } w, \ |w| \leq |ux|\}$$

L'ensemble $V(u)$ contient les suffixes de x pour lesquels le premier type de décalage est possible, et $W(u)$ contient les mots intervenant dans le deuxième

cas; on peut clairement se limiter aux mots de longueur majorée par $|ux|$. La deuxième fonction de décalage, que nous appellerons la *fonction du bon suffixe* est traditionnellement notée d_2. Elle est définie, pour $i = 0, \ldots, m$, en posant $u = x_{i+1} \cdots x_m$ par

$$d_2(i) = \min_{v \in V(u) \cup W(u)} |v|$$

Notons que les mots de $W(u)$ sont de longueur supérieure à $|x|$; on les considère donc uniquement lorsque $V(u) = \emptyset$.

Exemple. Considérons le mot $x = aababab$. La fonction du bon suffixe est :

	0	1	2	3	4	5	6	7
x		a	a	b	a	b	a	b
$d_2(i)$	14	13	12	6	10	6	8	1

Pour $i = 5$, le suffixe $u = ab$ est bord des suffixes $abab$ et $ababab$. Seul le deuxième est disjoint de u, donc $V(u) = \{ababab\}$, et $d_2(5) = 6$. Pour $i = 4$, le suffixe est $u = bab$; on a $V(u) = \emptyset$ et $W(u) = \{ux\}$, donc $d_2(4) = 10$. Pour $i = 7$, le mot vide est bord disjoint de b, donc $d_2(1) = 1$.

L'algorithme de Boyer et Moore procède comme suit : Le motif $x = x_1 \cdots x_m$ est comparé aux facteurs de longueur m du texte t. Pour une position donnée, on compare les lettres de la droite vers la gauche. Lorsqu'une différence est constatée, c'est-à-dire lorsque $x_i \neq t_j$ pour des indices i et j, l'indice j est incrémenté, l'indice i remis à m, et la comparaison recommence sur les lettres t_j et x_i. Pour l'incrémentation de j, on peut choisir l'une ou l'autre des fonctions de décalage; en fait, on choisit celle qui fournit la plus grande valeur. D'où l'algorithme :

```
Algorithme BOYER-MOORE(x, t);
  j := m;
  tantque j ≤ n faire
    i := m;
    tantque i > 0 etalors t_j = x_i faire
      i := i − 1; j := j − 1 fintantque;
    si i = 0 alors
      j est une fin d'occurrence de x;
      j := j + d_2[i]
    sinon
      j := j + max(d[t_j], d_2[i])
    finsi
  fintantque.
```

Exemple. Reprenons le même exemple du texte $t = aabbbababacaabbaba \cdots$ et du motif $x = aababab$ avec l'algorithme de Boyer et Moore complet. La

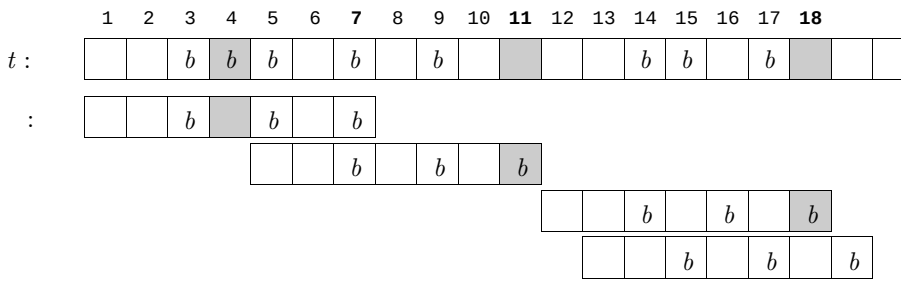

Figure 2.6: *Algorithme de Boyer-Moore complet.*

figure 2.6 montre les décalages successifs du motif. Les différences sont constatées aux positions sombres du texte. Les valeurs successives que prend l'indice j en début de la boucle *tantque* extérieure sont indiquées en gras. Le nombre total de comparaisons de caractères est 8.

Malgré son efficacité expérimentale, l'algorithme de Boyer et Moore peut prendre un temps en $O(|x||t|)$ lorsque l'on cherche *toutes* les occurrences d'un motif dans un texte (prendre $x = a^m$ et $t = a^n$). Une modification de l'algorithme a été proposée qui permet, au moyen d'un prétraitement plus compliqué, d'obtenir un temps linéaire dans tous les cas. R. Cole a prouvé récemment qu'avec cet algorithme, le nombre de comparaisons est toujours borné par $3|t|$. La preuve est compliquée et ne sera pas donnée ici.

10.2.3 Fonction du bon suffixe

Pour compléter l'exposé de l'algorithme de Boyer et Moore, nous allons calculer la fonction du bon préfixe d_2. Il s'avère que l'on peut se ramener à des concepts familiers en *retournant* le motif (c'est-à-dire en prenant son image miroir). Nous allons en fait calculer une *fonction du bon préfixe* du motif.

Soit $x = x_1 \cdots x_m$ un mot, et soit u un préfixe de x. Posons

$$V^{\sim}(u) = \{v \mid v \text{ est préfixe de } x \text{ et } u \text{ est bord disjoint de } v\}$$

Par ailleurs, soit (en conformité avec la notation de la section 10.2.5) $f_u(x)$ le plus long suffixe de x qui est préfixe de u et $w_u = g_u(x)u$, où $g_u(x)$ est défini par $g_u(x)f_u(x) = x$. On définit la fonction du bon préfixe δ pour $i = 0, \ldots, m-1$ et $u = x_1 \cdots x_i$ par

$$\delta(i) = \min_{v \in V^{\sim}(u) \cup \{w_u\}} |v|$$

Si d_2 est la fonction du bon suffixe de $x^{\sim} = x_m \cdots x_1$, alors par construction

$$d_2(i) = \delta(m-i) \qquad i = 1, \ldots, m-1$$

Pour déterminer la fonction du bon préfixe, il faut évaluer l'ensemble $V^{\sim}(u)$ et le mot w_u pour chaque préfixe u de x. Les mots w_u se calculent bien à l'aide de la

fonction $\beta = \beta_x$ qui donne, pour chaque préfixe $x_1 \cdots x_j$ de x, la longueur $\beta(j)$ du bord maximal de $x_1 \cdots x_j$. Nous avons déjà expliqué plus haut comment calculer cette fonction. Considérons alors les nombres

$$m,\ \beta(m),\ \beta^2(m), \ldots,\ \beta^r(m) = 0$$

et soit $u = x_1 \cdots x_i$ un préfixe de x. Si i est dans l'intervalle $[\beta^k(m), \beta^{k-1}(m)[$, on a $|f_u(x)| = \beta^k(m)$, puisque $f_u(x)$ est un bord de x. Il en résulte que

$$|w_u| = |g_u(x)| + |u| = m - \beta^k(m) + i$$

Ceci conduit aux instructions suivantes qui calculent les longueurs $|w_u|$ pour les préfixes de x :

```
j := m;
tantque j > 0 faire
   pour i de β[j] à j − 1 faire δ[i] := m − β[j] + i;
   j := β[j]
fintantque
```

Venons-en aux mots de $V^{\sim}(u)$. Posons $u = x_1 \cdots x_i$. Alors

$$V^{\sim}(u) = \{x_1 \cdots x_j \mid \beta(j) = i \text{ et } x_{j+1} \neq x_{i+1}\} \cup \Delta_i$$

où Δ_i est un terme correctif défini par

$$\Delta_i = \begin{cases} \{x\} & \text{si } \beta(m) = i; \\ \emptyset & \text{sinon.} \end{cases}$$

Pour évaluer correctement ces mots, il semble à première vue nécessaire de déterminer tous les bords disjoints de tous les préfixes de x, ce qui conduit à un algorithme qui n'est plus linéaire en temps. En fait, le morceau de programme suivant convient :

```
(1) pour j de 1 à m − 1 faire
(2)    i := β[j];
(3)    tantque i ≥ 0 etalors x_{j+1} ≠ x_{i+1} faire
(4)       δ[i] := min(δ[i], j);
(5)       i := β[i]
(6)    fintantque
(7) finpour
```

Expliquons pourquoi ce programme est correct. Si l'on remplace les lignes (3)−(6) par

```
(3′)   tantque i ≥ 0 faire
(4′)      si x_{j+1} ≠ x_{i+1} alors δ[i] := min(δ[i], j);
(5′)      i := β[i]
(6′)   fintantque
```

le programme calcule tous les bords disjoints, et choisit le plus petit. Le premier programme ne calcule qu'un sous-ensemble des bords disjoints; pour un indice j donné, il calcule uniquement la suite de bords disjoints *consécutifs* à partir du plus grand bord de $x_1 \dots x_j$. En d'autres termes, dès que l'on rencontre un bord, disons $x_1 \cdots x_k$, qui n'est pas disjoint de $x_1 \dots x_j$, le calcul des bords est interrompu. Ceci se justifie par la double observation suivante : un bord de $x_1 \cdots x_k$ est disjoint de $x_1 \cdots x_k$ si et seulement s'il est disjoint de $x_1 \cdots x_j$, et les bords disjoints de $x_1 \cdots x_k$ (donc les bords disjoints manquants de $x_1 \cdots x_j$) ont déjà été calculés lorsque l'indice de la boucle *pour* a pris la valeur k.

En composant les deux parties que nous venons de développer, on obtient le programme complet calculant la fonction du bon préfixe :

```
procédure Bon-Préfixe(x, δ);
  j := m;
  tantque j > 0 faire
    pour i de β[j] à j − 1 faire δ[i] := m − β[j] + i;
    j := β[j]
  fintantque;
  pour j de 1 à m − 1 faire
    i := β[j];
    tantque i ≥ 0 etalors x_{j+1} ≠ x_{i+1} faire
      δ[i] := min(δ[i], j); i := β[i]
    fintantque
  finpour.
```

Cette procédure calcule la fonction du bon préfixe en temps linéaire. La fonction du bon suffixe s'obtient en trois étapes; on prend l'image miroir du motif, on en calcule la fonction du bon préfixe, et on en déduit d_2 :

```
procédure Bon-Suffixe(x, d_2);
  pour i de 1 à m faire y[i] := x[m + 1 − i];
  Bords-maximaux(y, β);
  Bon-Préfixe(y, δ);
  d_2[0] := m + β[0];
  pour i de 1 à m faire d_2[i] := δ[m − i].
```

On peut ne pas passer par l'image miroir et on peut incorporer le calcul de β à l'intérieur de la procédure, mais le programme devient moins facile à comprendre.

10.3 L'algorithme de Aho et Corasick

Soit X un ensemble fini de mots. Nous considérons ici le problème de localisation des occurrences des mots de X dans un texte t.

On peut faire cette recherche séquentiellement, et chercher successivement les occurrences de chaque motif dans le texte, à l'aide d'un des algorithmes présentés précédemment. Cette démarche n'est pas efficace, puisqu'elle oblige à lire le texte t autant de fois qu'il y a de motifs. En revanche, un algorithme qui effectue la recherche parallèlement pour chaque motif semble prometteur. C'est essentiellement le fonctionnement de l'algorithme de Aho et Corasick que nous présentons ici. Il généralise l'algorithme de Knuth, Morris et Pratt au cas de plusieurs motifs à rechercher dans un texte. L'algorithme se présente comme une généralisation de l'automate des occurrences. On construit un automate déterministe reconnaissant l'ensemble A^*X, sans toutefois expliciter complètement sa fonction de transition, puis on utilise une fonction de suppléance similaire à celle décrite précédemment pour guider le cheminement dans l'automate.

Soit donc X un ensemble fini de mots sur un alphabet A, et soit P l'ensemble des préfixes des mots de X. On construit un automate

$$\mathcal{A} = (P, \varepsilon, P \cap A^*X)$$

reconnaissant A^*X, dont P est l'ensemble d'états, ε est l'état initial et $P \cap A^*X$ l'ensemble d'états terminaux. La fonction de transition est définie, pour $p \in P$ et $a \in A$, par

$$p \cdot a = f_X(pa)$$

où f_X est une extension de la fonction f_x utilisée dans l'algorithme de Knuth, Morris et Pratt. Elle est définie par

$$f_X(u) = \text{ le plus long suffixe de } u \text{ qui est dans } P.$$

En particulier, $p \cdot a = pa$ si $pa \in P$. La partie de l'automate formée seulement des flèches (p, a, pa), avec p et pa dans P, est appelée le *squelette* de l'automate.

Exemple. Considérons l'ensemble $X = \{aba, bab, acb, acbab, cbaba\}$ de 5 mots sur l'alphabet $A = \{a, b, c\}$. Le squelette de l'automate $\mathcal{A}$ est donné dans la figure 3.1. Les états sont numérotés de façon arbitraire. La façon d'obtenir que 4 est état terminal sera expliqué plus loin.

Pour poursuivre l'analogie avec l'automate des occurrences, définissons, pour tout mot u non vide, le mot $\mathrm{Bord}_X(u)$ comme le plus long suffixe propre de u qui est dans P. Voici la fonction Bord_X pour l'ensemble ci-dessus. On a remplacé les préfixes par leurs numéros.

i =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\mathrm{Bord}_X(i)$ =	−	0	13	14	15	8	0	13	14	1	2	3	15	0	6	7

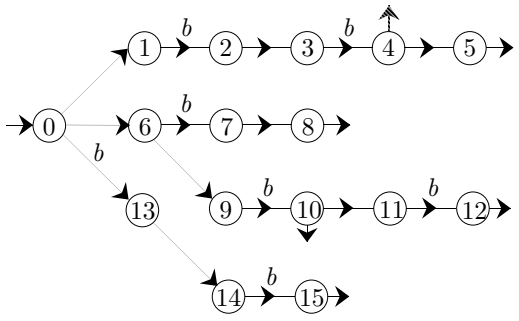

Figure 3.1: *Automate pour l'ensemble X.*

La fonction de transition de l'automate s'exprime à l'aide des bords. En effet :

$$p \cdot a = \begin{cases} pa & \text{si } pa \in P; \\ \text{Bord}_X(pa) & \text{sinon}; \end{cases}$$

et de plus,

$$\text{Bord}_X(pa) = \begin{cases} \text{Bord}_X(p) \cdot a & \text{si } p \neq \varepsilon; \\ \varepsilon & \text{sinon.} \end{cases}$$

Ces formules permettent d'évaluer la fonction de transition comme suit :

```
fonction TRANSITION(p, a);
   tantque pa ∉ P et p ≠ ε faire
      p := Bord_X(p)
   fintantque;
   si pa ∈ P alors retourner(pa) sinon retourner(ε).
```

Pour mettre en œuvre cette fonction, on doit résoudre deux problèmes : il faut connaître la fonction Bord_X, et il faut savoir tester rapidement si pa est dans P. Considérons d'abord la réalisation du test. On peut calculer une table g indicée par $P \times A$, avec

$$g[p, a] = \begin{cases} pa & \text{si } pa \in P; \\ \text{échec} & \text{sinon,} \end{cases}$$

mais cette façon de faire est irréaliste, car la table occupe trop de place dès que l'alphabet est grand (par exemple, si A est l'ensemble des caractères ASCII). Si, dans une application donnée, un tableau de cette taille ne crée pas de problème, alors mieux vaut remplir avec la fonction de transition complète de l'automate les cases qui contiendraient «échec», ce qui accélère ensuite la recherche.

De façon plus concrète, on range l'ensemble P des préfixes des motifs de X dans un arbre; tester si, pour un préfixe p et une lettre a, le mot pa est préfixe, revient à tester s'il y a un arc sortant de p et étiqueté par a. Ce test peut prendre un temps proportionnel à $|A|$ (ou $\log(|A|)$ si l'on programme plus soigneusement, en

rangeant les fils d'un sommet dans un arbre binaire équilibré). La place prise par l'arbre est proportionnelle à $\mathrm{Card}(P) \leq m$, où $m = \sum_{x \in X} |x|$ est la somme des longueurs des motifs. La recherche des occurrences des mots de X dans un texte $t = t_1 \dots t_n$ se fait par l'algorithme suivant, où l'on a incorporé l'évaluation des transitions :

```
procédure AHO-CORASICK(X, t)
  q := ε ;      {état initial}
  pour i de 1 à n faire
    tant que q t_i ∉ P et q ≠ ε faire q := Bord_X[q] fintantque;
    si q t_i ∈ P alors q := q t_i sinon q := ε finsi;
    si q est un état final alors afficher(q) finsi
  finpour.
```

Une table représentant la fonction Bord_X se calcule à l'aide d'un parcours en largeur du squelette de $\mathcal{A}$. Simultanément, on calcule les états terminaux autres que ceux correspondant aux mots de X. L'algorithme est le suivant :

```
procédure BORDS-DE-X(P, Bord_X)
  pour a ∈ A faire Bord_X[a] := ε finpour;
  pour p ∈ P \ ε et a ∈ A tels que pa ∈ P faire
    q := Bord_X[p];
    tantque qa ∉ P et q ≠ ε faire q := Bord_X[q] fintantque;
    si qa ∈ P alors Bord_X[pa] := qa sinon Bord_X[pa] := ε finsi;
    si qa est terminal alors ajouter pa aux états terminaux finsi;
  finpour.
```

Il n'est alors pas difficile de voir que l'algorithme de calcul des occurrences des mots de X dans t est en temps $O(n+m)$ et en place $O(m)$ (plus précisément en temps $O((n+m)|A|)$ ou $O((n+m)\log|A|)$ si l'on tient compte de la taille de l'alphabet).

10.4 Recherche d'expressions

Le problème que nous considérons dans cette section est le suivant : étant donnés une expression rationnelle e et un texte t, déterminer s'il existe un mot dans le langage $X = L(e)$ dénoté par e qui figure dans le texte t, et dans l'affirmative rapporter le mot et son occurrence. Ce problème apparaît couramment dans les éditeurs de textes, dès qu'ils ont des possibilités de recherche un peu sophistiquées.

Comme pour la recherche de motifs, la question revient à chercher le ou les préfixes du texte t qui appartiennent au langage A^*X ; la démarche est aussi la même : on construit d'abord, en « prétraitement », un automate reconnaissant A^*X, puis on cherche les préfixes de t reconnus par cet automate.

La construction d'un automate déterministe pour le langage A^*X n'est pas difficile, mais il peut avoir un nombre considérable d'états, même pour une expression courte : l'exemple donné dans le chapitre précédent montre que le nombre d'états peut être exponentiel en fonction de la taille de l'expression. Si l'on se contente d'un automate qui n'est pas nécessairement déterministe, c'est en revanche la reconnaissance d'un mot qui s'en trouve compliquée. Nous proposons un compromis entre ces deux possibilités : on construit d'abord un automate asynchrone particulier pour le langage dénoté par une expression rationnelle e, et dont le nombre d'états et de flèches est linéaire en fonction de la taille de e. On montre ensuite que la recherche des occurrences des mots du langage dans un texte t de longueur n peut se faire en temps $O(nm)$, où m est la taille de l'expression e.

10.4.1 Calcul efficace d'un automate

Nous étudions ici un procédé de calcul efficace d'un automate reconnaissant le langage dénoté par une expression rationnelle. Soit A un alphabet, et soit e une expression rationnelle sur A. La *taille* de e, notée $|e|$, est le nombre de symboles figurant dans e. Plus précisément, on a

$$|0| = |1| = |a| = 1 \quad \text{pour} \quad a \in A$$
$$|e + f| = |e \cdot f| = 1 + |e| + |f|, \quad |e^*| = 1 + |e|$$

Nous allons construire, pour toute expression e, un automate reconnaissant $L(e)$ qui a des propriétés particulières. Un automate asynchrone $\mathcal{A}$ est dit *normalisé* s'il vérifie les conditions suivantes :

(i) il existe un seul état initial, et un seul état final, et ces deux états sont distincts ;

(ii) aucune flèche ne pointe sur l'état initial, aucune flèche ne sort de l'état final ;

(iii) tout état est soit l'origine d'exactement une flèche étiquetée par une lettre, soit l'origine d'au plus deux flèches étiquetées par le mot vide ε.

Notons que le nombre de flèches d'un automate normalisé est au plus le double du nombre de ses états.

Proposition 4.1. *Pour toute expression rationnelle e de taille m, il existe un automate normalisé reconnaissant $L(e)$, et dont le nombre d'états est au plus $2m$.*

Preuve. Elle est constructive, et elle constitue en fait une autre démonstration de ce que tout langage rationnel est reconnaissable. On procède par récurrence

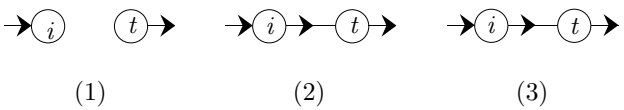

Figure 4.1: *Automates normalisés reconnaissant* (1) *l'ensemble vide,* (2) *le mot vide,* (3) *la lettre a.*

sur la taille de e. Pour $e = 0$, $e = 1$, et $e = a$, où $a \in A$, les automates de la figure 4.1 donnent des automates normalisés ayant 2 états. Si $e = e' + e''$, soient $\mathcal{A}' = (Q', i', t')$ et $\mathcal{A}'' = (Q'', i'', t'')$ deux automates normalisés reconnaissant des langages $X' = L(e')$ et $X'' = L(e'')$. On suppose Q' et Q'' disjoints.

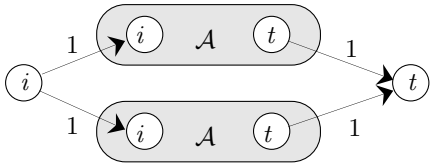

Figure 4.2: *Automate pour l'union.*

L'automate

$$\mathcal{A} = (Q' \cup Q'' \cup \{i, t\}, i, t)$$

où i et t sont deux nouveaux états distincts et dont les flèches sont, en plus de celles de $\mathcal{A}'$ et de A'', les quatre flèches $(i, 1, i')$, $(i, 1, i'')$, $(t', 1, t)$, $(t'', 1, t)$ reconnaît $X' \cup X'' = L(e)$ (voir figure 4.2). L'automate est normalisé, et $|Q| \leq 2|e|$. Si $e = e' \cdot e''$, considérons l'automate

$$\mathcal{A} = ((Q' \setminus t') \cup Q'', i', t'')$$

obtenu en «identifiant» t' et i'', c'est-à-dire en remplaçant toute flèche aboutissant en t' par la même flèche, mais aboutissant en i'' (voir figure 4.3).

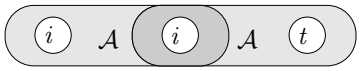

Figure 4.3: *Automate pour le produit.*

Cet automate reconnaît le langage $X'X''$; clairement, son nombre d'états est majoré par $2|e|$. Enfin, si $e = e'^*$, l'automate

$$\mathcal{A} = (Q' \cup \{i, t\}, i, t)$$

de la figure 4.4 qui, en plus des flèches de $\mathcal{A}'$, possède les quatre flèches $(i, 1, i')$, $(i, 1, t)$, $(t', 1, i')$, $(t', 1, t)$ reconnaît le langage $X'^* = L(e)$. Il est normalisé et a 2 états de plus que $\mathcal{A}'$. ∎

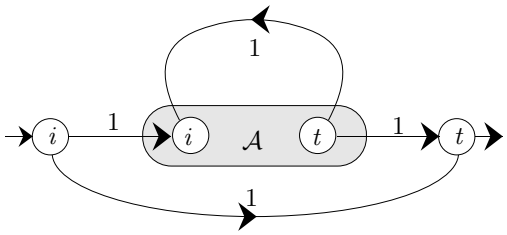

Figure 4.4: *Automate pour l'étoile.*

Exemple. Pour l'expression $(a+b)^*b(1+a)(1+a)^*$, la construction de la preuve produit l'automate de la figure 4.5, dans laquelle les flèches non marquées portent comme étiquette le mot vide. Observons qu'il existe de nombreux chemins dont

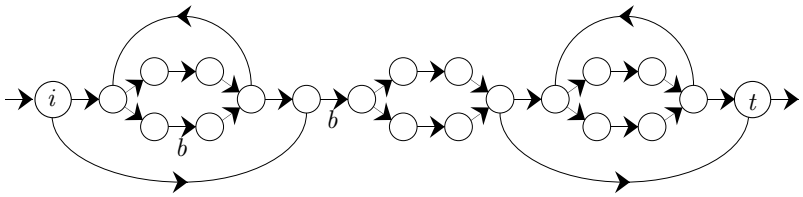

Figure 4.5: *Un automate pour l'expression* $(a+b)^*b(1+a)(1+a)^*$.

l'étiquette est vide, par exemple de l'état marqué d'une étoile à l'état terminal.

10.4.2 Recherche d'occurrences

Soit e une expression rationnelle; la recherche d'occurrences de mots dénotés par l'expression e dans un texte t se fait en deux étapes; dans un premier temps, on construit un automate normalisé $\mathcal{A}$ reconnaissant le langage $A^*L(e)$, à partir de l'expression $(a_1 + \cdots + a_h)^*e$, où $A = \{a_1, \ldots, a_h\}$. Cette étape constitue le prétraitement. Dans un deuxième temps, le texte t est lu lettre par lettre. Pour chaque préfixe de t, on calcule l'ensemble des états de $\mathcal{A}$ accessibles à partir de l'état initial par ce préfixe, et on affiche les préfixes pour lesquels l'état final est accessible. Cette deuxième étape est réalisable en temps $O(Nn)$ où N est le nombre d'états de $\mathcal{A}$ et n est la longueur de t; la raison en est la forme particulière de $\mathcal{A}$ qui permet de calculer chaque nouvel ensemble d'états accessibles en temps linéaire en N à partir du précédent. Comme $N = O(|e|)$, on obtient un algorithme en $O(nm)$, où $m = |e|$.

Le calcul d'un automate normalisé à partir d'une expression rationnelle peut se faire en temps et en place linéaires en fonction de la taille de l'expression. Une façon économique de représenter un automate normalisé est d'utiliser deux tables indicées par son ensemble d'états : la première contient, pour l'indice q, l'unique couple (a, q'), s'il existe, tel que (q, a, q') est une flèche de l'automate avec a une

lettre; la deuxième contient l'ensemble des états qui sont extrémités de flèches d'origine q et étiquetées par le mot vide; cet ensemble a au plus 2 éléments.

Nous en venons maintenant à la reconnaissance. Soit $\mathcal{A} = (Q, i, t)$ un automate normalisé. Comment vérifier si un mot w est reconnu par l'automate $\mathcal{A}$? Evidemment, on ne veut pas déterminiser l'automate, pour ne pas perdre le profit du nombre réduit d'états et de flèches. La méthode consiste à simuler la déterminisation de l'automate $\mathcal{A}$ en conservant, à chaque étape, l'ensemble des états accessibles de l'état initial par un chemin dont l'étiquette est la partie déjà lue du mot d'entrée. La seule difficulté est le calcul des états accessibles à partir d'un état par un chemin d'étiquette 1.

Soit P un ensemble d'états. Notons $\varepsilon(P)$ l'ensemble des états $q \in Q$ pour lesquels il existe $p \in P$ et un chemin $c : p \to q$ d'étiquette 1. Pour toute lettre a, notons $P \cdot a$ l'ensemble des états $q \in Q$ tels qu'il existe une flèche (p, a, q) dans $\mathcal{A}$, avec $p \in P$. Pour vérifier si un mot $w = a_1 \cdots a_n$ est reconnu par $\mathcal{A}$, on construit une suite $P_0, \ldots, P_n$ d'ensembles d'états par

$$\begin{aligned} P_0 &= \varepsilon(\{i\}); \\ P_k &= \varepsilon\left(P_{k-1} \cdot a_k\right), \qquad k = 1, \ldots, n. \end{aligned}$$

Il est immédiat que P_k est l'ensemble des états accessibles de i par un chemin d'étiquette $a_1 \cdots a_k$. Par conséquent, w est accepté par $\mathcal{A}$ si et seulement si l'état final t appartient à P_n. On en déduit donc l'algorithme suivant (où $w = a_1 \cdots a_n$, $\textsc{trans}(P, a) = P \cdot a$, et $\textsc{epsilon}(P) = \varepsilon(P)$) :

```
fonction RECONNAISSANCE(w : mot) : booléen;
  P :=EPSILON({i});
  pour k de 1 à n faire
    P :=EPSILON(TRANS(P, a_k))
  finpour;
  si (t ∈ P) alors retourner(vrai) sinon retourner(faux) finsi.
```

Pour la mise en œuvre de cet algorithme, nous représentons les ensembles d'états par deux structures de données, l'une qui permet l'adjonction et la suppression d'un élément en temps constant, comme une pile ou une file, et l'autre qui permet l'adjonction et le test d'appartenance en temps constant, comme un vecteur booléen. La première ou la deuxième structure est utilisée dans le calcul de la fonction de transition :

```
fonction TRANS(P : ensemble; a : lettre) : ensemble;
(1)  R := ∅;
(2)  pour q dans P faire
(3)     s'il existe q' tel que (q, a, q') est une flèche alors
(4)        R := R ∪ q
(5)     finsi
(6)  finpour;
(7)  retourner(R).
```

L'initialisation à la ligne (1) se fait en temps $O(N)$, si $\mathcal{A}$ a N états; le test à la ligne (3) est en temps constant, donc la boucle est en temps $O(N)$.

Le calcul de ε se fait bien entendu par un parcours de graphe, puisqu'il s'agit de calculer les états accessibles par des chemins composés uniquement de flèches étiquetées par le mot vide. On peut l'implémenter par exemple comme suit, en utilisant deux représentations d'un même ensemble, la première (notée T) permettant de tester en temps constant si cet ensemble est vide, et d'ajouter et de supprimer des éléments (pile ou file par exemple), la deuxième (notée R) permettant de tester l'appartenance en temps constant (un tableau booléen par exemple) :

```
fonction EPSILON(P : ensemble) : ensemble;
(1)  T := P; R := P;
(2)  tant que T ≠ ∅ faire
(3)     choisir q dans T;
(4)     pour chaque flèche (q, 1, q') d'origine q faire
(5)        si q' ∉ R alors R := R ∪ q'; T := T ∪ q' finsi
(6)     finpour
(7)  fintantque;
(8)  retourner(T).
```

La ligne (1) se fait en temps linéaire en N (nombre d'états de l'automate $\mathcal{A}$), la boucle (2) n'est pas exécutée plus de N fois, puisqu'à chaque tour on enlève un élément à T, et que l'on n'ajoute que des états non encore visités. Le nombre de flèches examinées dans la boucle (4) est au plus 2 (et ceci est essentiel pour la linéarité de l'algorithme), et les opérations de la ligne (5) se font en temps constant. Ceci montre que l'algorithme est en temps $O(N)$.

Pour le test d'occurrences, on procède comme pour la reconnaissance, sauf que l'on affiche toutes les positions où se termine un mot reconnu par l'automate. Voici l'algorithme (on a posé $t = t_1 \cdots t_n$) :

```
procédure OCCURRENCES(t : mot) : booléen;
  P :=EPSILON({i});
  pour k de 1 à n faire
    P :=EPSILON(TRANS(P, t_k));
    si l'état terminal est dans P alors afficher(k) finsi;
  finpour.
```

Notes

Parmi les différents algorithmes de recherche d'un motif qui ont été exposés, l'algorithme de Boyer et Moore est expérimentalement le plus rapide, et dans la variante de Horspool, il est simple à programmer. La fonction `egrep` de Unix utilise par exemple un algorithme de recherche d'expressions et, pour des chaînes de caractères, l'algorithme de Boyer-Moore dans la variante de Horspool. L'éditeur de texte Emacs fait amplement appel à la recherche d'expressions, par exemple dans le traitement du courrier. Les performances des divers algorithmes dans le cas le plus défavorable sont bien connues, pour l'efficacité en moyenne (sous l'hypothèse de distribution uniforme) certains des résultats sont très récents, et d'autres manquent encore. Voici une table de valeurs connues :

	pire cas	moyenne
Naïf	$\|x\|\,\|t\|$	$1 + 1/(q-1)$
Morris-Pratt	$2\|t\|$	$1 + 1/q - 2/q^2$
Knuth-Morris-Pratt	*id.*	—
Simon	*id.*	—
Horspool	$\|x\|\,\|t\|$	$2/(1+q)$
Boyer-Moore simple	*id.*	—
Boyer-Moore (variante)	$3\|t\|$	—

Les estimations en moyenne sont de M. Régnier et de Baeza-Yates, Régnier. Elles désignent le nombre moyen de comparaisons par caractère du texte, et valent lorsque la longueur du motif et la taille q de l'alphabet sont «grandes». L'estimation dans le cas le plus défavorable de l'algorithme de Boyer-Moore, due à R. Cole, s'applique à une variante de cet algorithme.

L'exposé de ce chapitre a profité des notes du cours de M. Crochemore à l'Université Paris 7. L'histoire des algorithmes de recherche de motifs jusqu'en 1977 est contée dans :

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

La version simplifiée de l'algorithme, due à Morris et Pratt, et qui date de 1970, n'a jamais été publiée. La référence la plus complète est :

A. V. Aho, Algorithms for finding patterns in strings, in : J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A*, North-Holland, 1990, 255–300.

Ce chapitre contient d'autres algorithmes, comme l'algorithme de Commentz-Walter qui est une version à la Boyer et Moore de la recherche d'un ensemble de motifs dans un texte. Les recherches continuent activement sur l'élaboration et l'évaluation d'algorithmes de recherche de motifs, en particulier d'algorithmes qui minimisent à la fois le nombre de comparaisons, le délai, et la place requise pour conserver le prétraitement. D'autres algorithmes pour une recherche en parallèle, ou pour une recherche avec symboles indifférents, ont aussi été proposés.

Exercices

10.1. Décrire comment il convient de modifier l'algorithme de Knuth, Morris et Pratt pour qu'il détecte *toutes* les occurrences d'un motif dans un texte.

10.2. Deux mots u et v sont *conjugués* s'il existe x, y tels que $u = xy$, $v = yx$. Donner un algorithme qui teste en temps $O(n)$ si deux mots de longueur n sont conjugués.

10.3. Les *mots de Fibonacci* sont définis par $f_0 = a$, $f_1 = ab$, et $f_{n+2} = f_{n+1}f_n$ pour $n \geq 0$. Montrer que $\mathrm{Bord}(f_{n+2}) = f_n$.

10.4. Donner des exemples de mots x pour lesquels l'automate $\mathcal{A}(x)$ a exactement $|x|$ flèches arrière.

10.5. Montrer que tout chemin d'étiquette x dans l'automate $\mathcal{A}(x)$ utilise au plus une flèche arrière.

10.6. Montrer que l'algorithme de Boyer et Moore fait de l'ordre de $3|t|$ opérations en cherchant une occurrence de $x = (ba^k)^2$ dans $t = a^{k+2}(ba^{k+2})^p$, quels que soient les entiers positifs k et p.

10.7. Montrer que l'automate construit dans l'algorithme de Aho et Corasick n'est en général pas l'automate minimal reconnaissant A^*X.

10.8. Soit A un alphabet, et soit $\bullet$ une lettre qui n'est pas dans A. Si $u = a_1 \cdots a_m$ et $v = b_1 \cdots b_m$ sont deux mots, avec $a_i, b_i \in A \cup \{\bullet\}$, on pose $u \sqsubseteq v$ si $a_i \neq \bullet$ implique $a_i = b_i$ pour $1 \leq i \leq m$.

Soit $x = x_1 \cdots x_m$ un mot de longueur m sur A. L'*automate de Boyer et Moore* de x, noté $\mathcal{A}(x)$, a pour états les mots y de longueur m sur $A \cup \{\bullet\}$ tels que $y \sqsubseteq x$ (accessibles à partir de l'état initial comme expliqué ci-dessous). L'état initial est $\bullet^m$, l'état final est x. La fonction de transition est définie comme suit. Soit $q = u\bullet v$, avec $v = x_{k+1} \cdots x_m \in A^*$, soit $a \in A$, et soit $p = uav = y_1 \cdots y_m$. Alors

$$q \cdot a = \begin{cases} p & \text{si } a = x_k\,; \\ y_{1+d} \cdots y_m \bullet^d & \text{sinon,} \end{cases}$$

où d est le plus petit entier tel que $y_{1+d} \cdots y_m \sqsubseteq x_1 \cdots x_{m-d}$. Par exemple, l'automate de Boyer et Moore de *aba* est donné dans la figure ci-dessous.

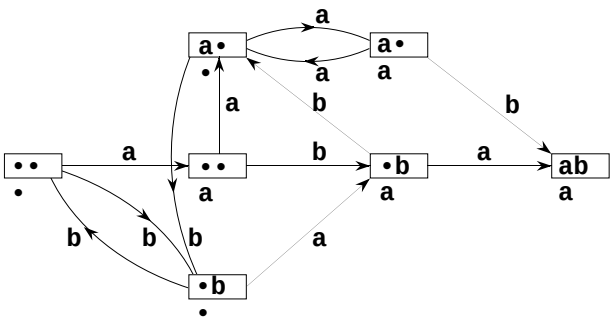

Figure 4.6: *L'automate de Boyer et Moore pour aba.*

a) Ecrire un programme qui prend en argument un mot x et qui calcule l'automate de Boyer et Moore.

b) Montrer que si toutes les lettres de x sont distinctes, l'automate a exactement $m(m+1)/2$ états.

c) Montrer que si $x = a^i b a^j$ avec $i + j + 1 = m$, alors $\mathcal{A}(x)$ a $\theta(m^3)$ états.