ALGORITHMIQUE DU TEXTE

Maxime Crochemore, Christophe Hancart, Thierry Lecroq

vuibert

Algorithmique du texte

Maxime Crochemore
Christophe Hancart
Thierry Lecroq

Algorithmique du texte

Vuibert Informatique

Algorithmique du texte – Maxime Crochemore, Christophe Hancart et Thierry Lecroq

Conception de la couverture : Jean Widmer

Les programmes et exemples figurant dans cet ouvrage ont pour but d'illustrer les sujets traités. Il n'est donné aucune garantie quant à leur utilisation dans le cadre d'une activité professionnelle ou commerciale.

Contact : `informatique@vuibert.fr`
Web : `www.vuibert.fr`

© Vuibert, Paris, 2001
ISBN 2-7117-8628-5

Toute représentation ou reproduction intégrale ou partielle, faite sans le consentement de l'auteur, etc.

Table des matières

Préface

Cet ouvrage présente un large panorama des méthodes algorithmiques utilisées pour traiter le texte. À ce titre il s'agit d'un livre d'algorithmique, mais dont l'objet est focalisé sur la manipulation du texte par ordinateur. L'idée de cette publication résulte du constat qu'il n'existe aucune autre référence en français abordant le sujet avec cette ampleur (et les rares livres en anglais consacrés au sujet sont essentiellement des monographies de recherche). Ce constat est d'autant plus surprenant que les problèmes du domaine sont connus depuis le développement des systèmes d'exploitation évolués et que le recours à des solutions efficaces devient crucial depuis l'utilisation massive de l'informatique et de la bureautique dans de nombreux secteurs de la société.

Sous une forme écrite ou vocale, le texte est le seul véhicule fiable des concepts abstraits. Il reste donc le support privilégié des systèmes d'informations, en dépit d'efforts importants vers l'utilisation d'autres médias (interfaces graphiques, systèmes de réalité virtuelle, films de synthèse, etc.). Cet aspect est encore renforcé par les bases de connaissances littéraires, juridiques, commerciales ou autres qui se développent sur l'Internet grâce en particulier aux services de la Toile.

Le contenu de l'ouvrage porte sur les éléments formels et les bases techniques utilisés dans les domaines de la recherche documentaire, de l'indexation automatique pour les moteurs de recherche, et des logiciels systèmes, ce qui inclut l'édition, le traitement et la compression de textes. Les méthodes qui sont décrites s'appliquent au traitement de la langue naturelle, au traitement et à l'analyse des séquences génétiques, à l'analyse de séquences musicales, aux questions de sécurité liées aux flux de données, et à la gestion des bases de données textuelles, pour ne citer que quelques applications immédiates.

Les sujets retenus comprennent la localisation de motifs textuels, l'indexation de données textuelles, la comparaison de textes par alignement et la recherche de régularités locales. En plus de leur intérêt pratique, ces sujets possèdent des aspects théoriques et combinatoires qui fournissent d'étonnants exemples de solutions algorithmiques.

Le but de l'ouvrage est avant tout pédagogique. Il s'adresse en premier lieu aux étudiants des classes préparatoires aux grandes écoles, aux étudiants des seconds et troisièmes cycles universitaires d'informatique, ainsi qu'aux élèves-ingénieurs en informatique. Mais il peut aussi être utilisé par des ingénieurs de production de logiciels.

Nous remercions vivement les chercheurs qui ont pris le temps de relire et de commenter les ébauches préalables de ce livre. Il s'agit de Saïd Abdeddaïm, Marie-Pierre Béal, Christian Charras, Sabine Mercier, Laurent Mouchard, Johann Pelfrêne, Bruno Petazzoni, Mathieu Raffinot, Giuseppina Rindone, Marie-France Sagot. Les erreurs qui restent sont les nôtres. Le lecteur avisé notera l'utilisation des règles de l'orthographe recommandées par le Conseil supérieur de la langue française (*Journal officiel* du 6 décembre 1990) qui nous fait par exemple écrire « apparaitre » et « ambigüité » sous ces formes.

Enfin, des éléments complémentaires au contenu de l'ouvrage sont accessibles sur le site `http://chl.univ-mlv.fr/` ou depuis les pages des auteurs.

Maxime Crochemore
Christophe Hancart
Thierry Lecroq
Marne-la-Vallée et Rouen,
juin 2001

1 Outils

Ce chapitre présente le cadre algorithmique et combinatoire dans lequel sont développés les chapitres suivants. Il précise pour commencer les notions et notations utilisées pour travailler sur les mots, les langages et les automates. La suite est principalement consacrée à l'introduction des structures de données retenues pour la réalisation d'automates et à la présentation de résultats combinatoires. Cette organisation s'appuie sur la constatation que les algorithmes efficaces de traitement du texte reposent sur l'un ou l'autre de ces aspects.

La section 1.2 fournit quelques propriétés combinatoires des mots qui reviennent dans bon nombre de preuves de validité d'algorithmes ou d'évaluations de leurs performances. Il s'agit principalement de résultats sur les périodicités qui apparaissent dans certains mots.

Le formalisme de description des algorithmes est présenté dans la section 1.3 qui est surtout centrée sur le type d'algorithmes décrits dans l'ouvrage et qui introduit quelques objets standard concernant la gestion de files et d'automates.

La section 1.4 détaille plusieurs procédés pour implanter des automates en mémoire, procédés qui contribuent notamment aux résultats des chapitres 2, 5 et 6.

Les premiers algorithmes de localisation de mots sont présentés dans la section 1.5. Les techniques de fenêtre glissante, d'automate de recherche et d'utilisation de mots machine qui y sont décrites sont reprises et améliorées dans les chapitres 2, 3 et 8, en particulier.

La section 1.6 est le joyau algorithmique du chapitre. Elle présente deux méthodes algorithmiques fondamentales utilisées pour le traitement du texte. Elles servent à calculer la table des bords et la table des préfixes d'un mot qui constituent deux tables essentielles car elles condensent une partie des propriétés combinatoires du mot. Leur utilisation ou adaptation est considérée dans les chapitres 2 et 3, mais revient ponctuellement dans d'autres chapitres.

Enfin, on notera que l'intuition s'appuie quelquefois sur des figures dont le style est introduit dans le chapitre et conservé ensuite.

1.1 Mots et automates

On introduit dans cette section les notations sur les mots, les langages et les automates.

Alphabet et mots

Un ***alphabet*** est un ensemble fini non vide dont les éléments sont appelés des ***lettres***. Un ***mot*** sur un alphabet A est une suite finie d'éléments de A. La suite de zéro lettre est appelée le ***mot vide*** et notée ε. Pour simplifier, les délimiteurs et les séparateurs utilisés habituellement dans les notations des suites sont supprimés et l'on écrit un mot comme la simple juxtaposition des lettres qui le composent. Ainsi, ε, `a`, `b` et `baba` sont-ils des mots sur tout alphabet qui contient les deux lettres `a` et `b`. L'ensemble de tous les mots sur l'alphabet A est noté A^*, et l'ensemble de tous les mots sur l'alphabet A excepté le mot vide ε est noté A^+.

La ***longueur*** d'un mot x est définie comme la longueur de la suite associée au mot x et est notée $|x|$. On note $x[i]$, pour $i = 0, 1, \ldots, |x|-1$, la lettre à l'indice i de x en convenant de commencer la numérotation des indices à partir de 0. Lorsque $x \neq \varepsilon$, on dit plus spécifiquement de chaque indice $i = 0, 1, \ldots, |x|-1$ qu'il est une ***position*** sur x. Il s'ensuit que la j-ième lettre de x est la lettre à la position $j-1$ sur x et que :

$$x = x[0]x[1]\ldots x[|x|-1] \ .$$

D'où aussi une définition élémentaire de l'identité de deux mots quelconques x et y :

$x = y$

si et seulement si

$|x| = |y|$ et $x[i] = y[i]$ pour $i = 0, 1, \ldots, |x|-1$.

L'ensemble des lettres sur lequel est formé le mot x est noté $alph(x)$. Par exemple, si $x =$ `abaaab`, on a $|x| = 6$ et $alph(x) = \{\mathtt{a}, \mathtt{b}\}$.

Le ***produit*** – on dit aussi la ***concaténation*** – de deux mots x et y est le mot composé des lettres de x puis de celles de y dans cet ordre. On le note xy ou encore $x \cdot y$ pour faire apparaitre une décomposition du mot résultant. L'élément neutre pour le produit est ε. Pour tout mot x et tout naturel n, on définit la n-ième ***puissance*** du mot x, notée x^n, par $x^0 = \varepsilon$ et $x^k = x^{k-1}x$ pour $k = 1, 2, \ldots, n$. Sont notés respectivement zy^{-1} et $x^{-1}z$ les mots x et y lorsque $z = xy$. Le ***renversé*** – ou ***image miroir*** – du mot x est le mot $x^\sim$ défini par :

$$x^\sim = x[|x|-1]x[|x|-2]\ldots x[0] \ .$$

Un mot x est un ***facteur*** d'un mot y s'il existe deux mots u et v tels que $y = uxv$. Lorsque $u = \varepsilon$, x est un ***préfixe*** de y ; et lorsque

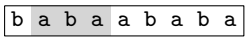


Figure 1.1 Une occurrence du mot `aba` dans le mot `babaababa` à la position (gauche) 1.

$v = \varepsilon$, x est un ***suffixe*** de y. Le mot x est un ***sous-mot*** de y s'il existe $|x| + 1$ mots $w_0, w_1, \ldots, w_{|x|}$ tels que $y = w_0x[0]w_1x[1]\ldots x[|x|-1]w_{|x|}$; de manière moins formelle, x est un mot obtenu de y en lui supprimant $|y| - |x|$ lettres. Un facteur ou un sous-mot x d'un mot y est qualifié de ***propre*** si $x \neq y$. On note respectivement $x \preceq_{fact} y$, $x \prec_{fact} y$, $x \preceq_{préf} y$, $x \prec_{préf} y$, $x \preceq_{suff} y$, $x \prec_{suff} y$, $x \preceq_{smot} y$ et $x \prec_{smot} y$ lorsque x est un facteur, un facteur propre, un préfixe, un préfixe propre, un suffixe, un suffixe propre, un sous-mot et un sous-mot propre de y. On peut vérifier que $\preceq_{fact}$, $\preceq_{préf}$, $\preceq_{suff}$ y et $\preceq_{smot}$ sont des relations d'ordre sur A^*.

L'***ordre lexicographique***, noté $\leq$, est un ordre sur les mots induit par un ordre sur les lettres noté de la même façon. Il est défini comme suit. Pour $x, y \in A^*$, $x \leq y$ si et seulement si, soit $x \preceq_{préf} y$, soit x et y se décomposent sous la forme $x = uav$ et $y = ubw$ avec $u, v, w \in A^*$, $a, b \in A$ et $a < b$. Ainsi, `ababb` < `abba` < `abbaab` en supposant `a` < `b`.

Étant donnés un mot non vide x et un mot y, on dit qu'il y a une ***occurrence*** de x dans y, ou, plus simplement, que x ***apparait*** dans y, lorsque x est un facteur de y. Toute occurrence, ou toute apparition, de x peut être caractérisée par une position sur y. Ainsi dit-on qu'une occurrence de x ***débute*** à la ***position (gauche)*** i sur y lorsque $y[i\,..\,i + |x| - 1] = x$ (voir figure 1.1). Il est quelquefois plus agréable de considérer la ***position droite*** $i + |x| - 1$ en laquelle cette même occurrence ***se termine***. Par exemple, les positions gauches et droites auxquelles apparait le mot $x =$ `aba` dans le mot $y =$ `babaababa` sont :

i	0	1	2	3	4	5	6	7	8
$y[i]$	`b`	`a`	`b`	`a`	`a`	`b`	`a`	`b`	`a`
positions gauches		1			4		6		
positions droites				3			6		8

La ***position de la première occurrence*** $pos(x)$ de x dans y est la position minimale en laquelle débute l'occurrence de x dans yA^*. Avec les notations sur les langages rappelées ci-après, on a :

$$pos(x) = \min\{|u| : \{ux\}A^* \cap \{y\}A^* \neq \emptyset\} \ .$$

Les notations avec des crochets pour les lettres des mots sont étendues aux facteurs. On définit le facteur $x[i\,..\,j]$ du mot x par :

$$x[i\,..\,j] = x[i]x[i+1]\ldots x[j]$$

pour tous entiers i et j satisfaisant $0 \leq i \leq |x|$, $-1 \leq j \leq |x| - 1$ et $i \leq j + 1$. Lorsque $i = j + 1$, le mot $x[i\,..\,j]$ est le mot vide.

Langages

Tout sous-ensemble de A^* est un ***langage*** sur l'alphabet A. Le produit défini sur les mots est étendu aux langages en posant :

$$XY = X \cdot Y = \{xy : (x, y) \in X \times Y\}$$

pour tous langages X et Y. On étend de même la notion de puissance en posant $X^0 = \{\varepsilon\}$ et $X^k = X^{k-1}X$ pour $k \geq 1$. L'***étoile*** de X est le langage :

$$X^* = \bigcup_{n \geq 0} X^n \ .$$

Est noté X^+ le langage défini par :

$$X^+ = \bigcup_{n \geq 1} X^n \ .$$

Remarquons que ces deux dernières notations sont compatibles avec les notations A^* et A^+. À la fois par abus et pour ne pas surcharger les notations, un langage réduit à un seul mot peut être nommé par le mot lui-même si cela n'entraine aucune confusion. Par exemple, l'expression abusive $A^*\mathtt{abaaab}$ désigne le langage des mots qui ont comme suffixe le mot `abaaab`, sous l'hypothèse $\{\mathtt{a}, \mathtt{b}\} \subseteq A$.

La notion de longueur est étendue aux langages en posant :

$$|X| = \sum_{x \in X} |x| \ .$$

De la même façon, on définit $alph(X)$ par :

$$alph(X) = \bigcup_{x \in X} alph(x)$$

et $X^\sim$ par :

$$X^\sim = \{x^\sim : x \in X\} \ .$$

Les ensembles des facteurs, des préfixes, des suffixes et des sous-mots des mots d'un langage X sont des langages particuliers souvent considérés dans la suite de l'ouvrage ; ils sont notés respectivement $Fact(X)$, $Préf(X)$, $Suff(X)$ et $SMot(X)$.

Le ***contexte droit*** d'un mot y relativement à un langage X est le langage :

$$y^{-1}X = \{y^{-1}x : x \in X\} \ .$$

La relation d'équivalence définie par l'identité des contextes droits est notée $\equiv_X$, ou simplement [1] $\equiv$. Ainsi :

$y \equiv z$ si et seulement si $y^{-1}X = z^{-1}X$

1. Comme dans toute la suite de l'ouvrage, les notations ne sont indicées par l'objet auquel elles réfèrent qu'en cas d'ambigüité éventuelle.

pour $y, z \in A^*$. Par exemple, lorsque $A = \{\mathtt{a}, \mathtt{b}\}$ et $X = A^*\{\mathtt{aba}\}$, la relation $\equiv$ admet quatre classes d'équivalence, à savoir $\{\varepsilon, \mathtt{b}\} \cup A^*\{\mathtt{bb}\}$, $\{\mathtt{a}\} \cup A^*\{\mathtt{aa}, \mathtt{bba}\}$, $A^*\{\mathtt{ab}\}$ et $A^*\{\mathtt{aba}\}$. Pour tout langage X, la relation $\equiv$ est une relation d'équivalence qui est compatible avec la concaténation. C'est la ***congruence syntaxique droite*** associée à X.

Expressions et langages rationnels

Les ***expressions rationnelles*** sur un alphabet A et les langages qu'elles décrivent, les ***langages rationnels***, sont définis récursivement comme suit :

- `0` et `1` sont des expressions rationnelles qui décrivent respectivement $\emptyset$ (l'ensemble vide) et $\{\varepsilon\}$;
- pour toute lettre $a \in A$, $\boldsymbol{a}$ est une expression rationnelle qui décrit le singleton $\{a\}$;
- si $\boldsymbol{x}$ et $\boldsymbol{y}$ sont des expressions rationnelles décrivant respectivement les langages rationnels X et Y, alors $(\boldsymbol{x}+\boldsymbol{y})$, $(\boldsymbol{x}\cdot\boldsymbol{y})$ et $(\boldsymbol{x})^*$ sont des expressions rationnelles qui décrivent respectivement les langages rationnels $X \cup Y$, $X \cdot Y$ et X^*.

L'ordre de priorité des opérations sur les expressions rationnelles est `*`, $\cdot$, puis `+`. D'où d'éventuelles simplifications d'écriture qui permettent d'omettre le symbole $\cdot$ et certaines paires de parenthèses. Le langage décrit par une expression rationnelle $\boldsymbol{x}$ est noté $\mathit{Lang}(\boldsymbol{x})$.

Automates

Un ***automate*** M sur l'alphabet A est composé d'un ensemble fini Q d'***états***, d'un état[2] ***initial*** q_0, d'un ensemble $T \subseteq Q$ d'états ***terminaux*** et d'un ensemble $F \subseteq Q \times A \times Q$ de ***flèches*** – ou ***transitions***. On note l'automate M par le quadruplet :

$$(Q, q_0, T, F) \ .$$

On dit d'une flèche (p, a, q) qu'elle sort de l'état p et qu'elle entre dans l'état q ; l'état p est la ***source*** de la flèche, la lettre a son ***étiquette***, l'état q sa ***cible***. Le nombre de flèches sortant d'un état donné est appelé le ***degré (sortant)*** de l'état. Le ***degré entrant*** d'un état est défini de façon duale. Par analogie avec les graphes, l'état q est un ***successeur*** par la lettre a de l'état p lorsque $(p, a, q) \in F$; dans le même cas, on dit du couple (a, q) qu'il est un ***successeur étiqueté*** de l'état p.

2. La définition standard des automates considère un ensemble d'états initiaux au lieu d'un seul état initial comme c'est le cas dans tout l'ouvrage. On laisse le soin au lecteur de se convaincre qu'il est possible de faire correspondre à tout automate défini de façon standard un automate à un seul état initial qui reconnait le même langage.

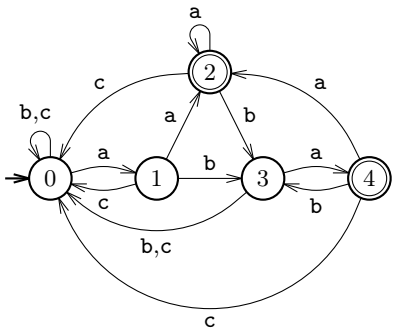


Figure 1.2 Représentation d'un automate sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Les états de l'automate sont numérotés de 0 à 4, son état initial est 0 et ses états terminaux sont 2 et 4. L'automate possède $3 \times 5 = 15$ flèches. Le langage qu'il reconnait est celui décrit par l'expression rationnelle `(a+b+c)*(aa+aba)`, c'est-à-dire l'ensemble des mots sur l'alphabet des trois lettres `a`, `b` et `c` se terminant par `aa` ou `aba`.

Un ***chemin*** de longueur n dans l'automate $M = (Q, q_0, T, F)$ est une suite de n flèches consécutives :

$$\langle (p_0, a_0, p'_0), (p_1, a_1, p'_1), \ldots, (p_{n-1}, a_{n-1}, p'_{n-1}) \rangle \ ,$$

c'est-à-dire telles que :

$$p'_k = p_{k+1}$$

pour $k = 0, 1, \ldots, n-2$. L'***étiquette*** du chemin est le mot $a_0 a_1 \ldots a_{n-1}$, son ***origine*** l'état p_0, sa ***fin*** l'état p'_{n-1}. Par convention, il existe pour chaque état p un chemin de longueur nulle d'origine et de fin p ; l'étiquette d'un tel chemin est ε, le mot vide. Un chemin dans l'automate M est ***réussi*** si son origine est l'état initial q_0 et si sa fin est dans T. Un mot est ***reconnu*** – ou ***accepté*** – par l'automate s'il est l'étiquette d'un chemin réussi. Le langage formé des mots reconnus par l'automate M est noté $Lang(M)$.

Souvent, bien plus que sa donnée formelle, un diagramme plan permet d'appréhender le fonctionnement d'un automate. On représente les états par des cercles et les flèches par des arcs dirigés de la source vers la cible et étiquetés par la lettre correspondante. Lorsque plusieurs flèches ont même source et même cible, on confond les arcs et l'on étiquette l'arc résultant par une énumération des lettres. L'état initial est distingué par une courte flèche entrante et les états terminaux sont doublement cerclés. Un exemple est montré figure 1.2.

Un état p d'un automate $M = (Q, q_0, T, F)$ est ***accessible*** s'il existe un chemin dans M d'origine q_0 et de fin p. Un état p est ***coaccessible*** s'il existe un chemin dans M d'origine p et de fin dans T.

Un automate $M = (Q, q_0, T, F)$ est ***déterministe*** si pour tout couple $(p, a) \in Q \times A$ il existe au plus un état $q \in Q$ tel que $(p, a, q) \in F$. Dans

un tel cas, il est naturel de considérer la ***fonction de transition*** :

$\delta: Q \times A \to Q$

de l'automate définie pour toute flèche $(p, a, q) \in F$ par :

$\delta(p, a) = q$

et non définie ailleurs. La fonction δ s'étend facilement aux mots. Il suffit de considérer son prolongement $\bar{\delta}: Q \times A^* \to Q$ défini récursivement par $\bar{\delta}(p, \varepsilon) = p$ et $\bar{\delta}(p, wa) = \delta(\bar{\delta}(p, w), a)$ pour $p \in Q$, $w \in A^*$ et $a \in A$. Il s'ensuit que le mot w est reconnu par l'automate M si $\bar{\delta}(q_0, w) \in T$. Généralement, la fonction δ et son prolongement $\bar{\delta}$ sont notés de la même manière.

L'automate $M = (Q, q_0, T, F)$ est ***complet*** lorsque pour tout couple $(p, a) \in Q \times A$ il existe au moins un état $q \in Q$ tel que $(p, a, q) \in F$.

Proposition 1.1
Pour tout automate, il existe un automate déterministe et complet qui reconnait le même langage. ∎

Rendre complet un automate n'est pas difficile : il suffit d'ajouter à l'automate un état ***rebut***, puis de le rendre cible de toutes les transitions non définies. Il est un peu plus délicat en revanche de ***déterminiser*** un automate, c'est-à-dire obtenir d'un automate $M = (Q, q_0, T, F)$ un automate déterministe reconnaissant le même langage. On peut utiliser la méthode dite de ***construction par sous-ensembles*** : soit M' l'automate dont les états sont les parties de Q, l'état initial le singleton $\{q_0\}$, les états terminaux les parties de Q dont l'intersection avec T est non vide, et les flèches les triplets (U, a, V) où V est l'ensemble des successeurs par la lettre a des états p appartenant à U ; alors M' est un automate déterministe qui reconnait le même langage que M. Dans la pratique, on ne construit pas l'automate M' en entier, mais seulement sa partie accessible en commençant à partir de l'état initial $\{q_0\}$.

Un langage X est ***reconnaissable*** s'il existe un automate M tel que $X = \mathit{Lang}(M)$. Suit l'énoncé d'un théorème fondamental de la théorie des automates qui établit le lien entre langages reconnaissables et langages rationnels sur un alphabet donné.

Théorème 1.2 (théorème de Kleene)
Un langage est reconnaissable si et seulement si il est rationnel. ∎

Si X est un langage reconnaissable, l'***automate minimal*** de X, noté $\mathcal{M}(X)$, est déterminé par la congruence syntaxique droite associée à X. C'est l'automate dont l'ensemble des états est $\{w^{-1}X : w \in A^*\}$, l'état initial X, l'ensemble des états terminaux $\{w^{-1}X : w \in X\}$, et l'ensemble des flèches $\{(w^{-1}X, a, (wa)^{-1}X) : (w, a) \in A^* \times A\}$.

Proposition 1.3
L'automate minimal $\mathcal{M}(X)$ d'un langage X est l'automate ayant le moins d'états parmi les automates déterministes et complets qui reconnaissent le langage X. L'automate $\mathcal{M}(X)$ est l'image homomorphe de tout automate reconnaissant X. ∎

On dit souvent d'un automate qu'il est minimal alors qu'il n'est pas complet. En fait, et par abus, cet automate est bien minimal si l'on prend soin de lui rajouter un état rebut.

Chacun des états d'un automate, ou parfois même chacune de ses flèches, peut être agrémenté d'une ***sortie***. Il s'agit d'une valeur ou d'un ensemble de valeurs associé à l'état ou à la flèche en question.

1.2 Un peu de combinatoire

On considère la notion de périodicité sur les mots pour laquelle on donne les propriétés de base. On commence par présenter deux familles de mots qui ont des propriétés combinatoires intéressantes au regard des questions de périodicités et de répétitions examinées dans plusieurs chapitres.

Quelques mots en particulier

Les ***nombres de Fibonacci*** sont définis par la récurrence :

$$\begin{aligned} F_0 &= 0\ , \\ F_1 &= 1\ , \\ F_n &= F_{n-1} + F_{n-2} \qquad \text{pour } n \geq 2\ . \end{aligned}$$

Ces nombres fameux vérifient des propriétés toutes plus remarquables les unes que les autres. Parmi celles-ci, on en signale juste deux :

- pour tout naturel $n \geq 2$, $\mathrm{pgcd}(F_n, F_{n-1}) = 1$;
- pour tout naturel n, F_n est l'entier le plus proche de $\Phi^n/\sqrt{5}$, où $\Phi = \frac{1}{2}(1+\sqrt{5}) = 1{,}61803\ldots$ est le ***nombre d'or***.

Les ***mots de Fibonacci*** sont définis sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}\}$ par la récurrence suivante :

$$\begin{aligned} f_0 &= \varepsilon\ , \\ f_1 &= \mathtt{b}\ , \\ f_2 &= \mathtt{a}\ , \\ f_n &= f_{n-1}f_{n-2} \qquad \text{pour } n \geq 3\ . \end{aligned}$$

Notons que la suite des longueurs des mots est exactement la suite des nombres de Fibonacci, c'est-à-dire que l'on a $F_n = |f_n|$. Voici en exemple les dix premiers nombres et mots de Fibonacci :

n	F_n	f_n
0	0	ε
1	1	b
2	1	a
3	2	ab
4	3	aba
5	5	abaab
6	8	abaababa
7	13	abaababaabaab
8	21	abaababaabaababaababa
9	34	abaababaabaababaababaabaababaabaab

L'intérêt des mots de Fibonacci est qu'ils satisfont des propriétés combinatoires intéressantes et contiennent un grand nombre de répétitions.

Les mots de de Bruijn considérés ici sont définis sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}\}$ et sont paramétrés par un naturel non nul. Un mot non vide $x \in A^+$ est un ***mot de de Bruijn*** d'ordre k si chaque mot sur A de longueur k apparait une fois et une seule dans x. Un premier exemple : ab et ba sont les deux seuls mots de de Bruijn d'ordre 1. Un second exemple : le mot aaababbbaa est un mot de de Bruijn d'ordre 3 puisque ses facteurs de longueur 3 sont les huit mots de A^3, à savoir aaa, aab, aba, abb, baa, bab, bba et bbb.

L'existence d'un mot de de Bruijn d'ordre $k \geq 2$ se vérifie à l'aide de l'***automate*** dont :

- les états sont les mots du langage A^{k-1} ;
- les flèches sont de la forme (av, b, vb) avec $a, b \in A$ et $v \in A^{k-2}$;

l'état initial et les états terminaux n'étant pas précisés (une illustration est montrée figure 1.3). On constate qu'exactement deux flèches sortent de chacun des états, l'une étiquetée par a, l'autre par b ; et qu'exactement deux flèches entrent dans chacun des états, toutes deux étiquetées par la même lettre. Le graphe associé à l'automate vérifie donc la condition d'Euler : le degré sortant et le degré entrant de chaque état sont identiques. Il s'en déduit qu'il existe un circuit eulérien partant de chaque

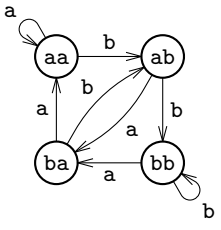


Figure 1.3 L'automate de de Bruijn à l'ordre 3 sur l'alphabet $\{\mathtt{a}, \mathtt{b}\}$. L'état initial de l'automate n'est pas précisé.

état. Maintenant, soit :

$\langle (u_0, a_0, u_1), (u_1, a_1, u_2), \ldots, (u_{n-1}, a_{n-1}, u_0) \rangle$

l'un des chemins correspondant. Alors le mot $u_0 a_0 a_1 \ldots a_{n-1}$ est un mot de de Bruijn d'ordre k, chacune des flèches du chemin étant identifiée à un facteur de longueur k. Il s'ensuit du même coup qu'un mot de de Bruijn d'ordre k a pour longueur $2^k + k - 1$ (soit $n = 2^k$ avec la notation précédente). On vérifie également que le nombre de mots de Bruijn d'ordre k est exponentiel en k.

Les mots de de Bruijn sont souvent utilisés comme exemple de cas limites en ce sens où ils contiennent tous les facteurs de longueur donnée.

Périodicité et bords

Soit x un mot non vide. Un entier p tel que $0 < p \leq |x|$ est une ***période*** de x si :

$x[i] = x[i + p]$

pour $i = 0, 1, \ldots, |x| - p - 1$. Remarquons que la longueur d'un mot non vide est une période de ce mot, de telle sorte que tout mot non vide possède au moins une période. On définit ainsi sans ambigüité aucune ***la période*** d'un mot non vide x comme la plus petite de ses périodes. On la note $pér(x)$. Par exemple, 3, 6, 7 et 8 sont des périodes du mot $x = \texttt{aabaabaa}$ et la période de x est $pér(x) = 3$.

On remarque que si p est une période de x, il en est de même de ses multiples kp, pour k entier satisfaisant $0 < k \leq \lfloor |x|/p \rfloor$.

Proposition 1.4
Soient x un mot non vide et p un entier tel que $0 < p \leq |x|$. Alors les cinq propriétés suivantes sont équivalentes :

1. *L'entier p est une période de x.*
2. *Il existe deux mots uniques $u \in A^*$ et $v \in A^+$ et un entier $k > 0$ tels que $x = (uv)^k u$ et $|uv| = p$.*
3. *Il existe un mot t et un entier $k > 0$ tels que $x \preceq_{préf} t^k$ et $|t| = p$.*
4. *Il existe trois mots u, v et w tels que $x = uw = wv$ et $|u| = |v| = p$.*
5. *Il existe un mot t tel que $x \preceq_{préf} tx$ et $|t| = p$.*

Preuve $1 \Rightarrow 2$: si $v \neq \varepsilon$ et $k > 0$, alors k est le dividende de la division entière de $|x|$ par p. Maintenant, si le triplet (u', v', k') satisfait les mêmes conditions que le triplet (u, v, k), on a $k' = k$ puis, pour une question de longueur, $|u'| = |u|$. Il vient à la suite que $u' = u$ et $v' = v$. Ce qui montre l'unicité de la décomposition sous réserve de son existence. Soient k et r le dividende et le reste de la division euclidienne de $|x|$ par p, puis u et v les deux facteurs de x définis par $u = x[0\,..\,r-1]$ et $v = x[r\,..\,p-1]$.

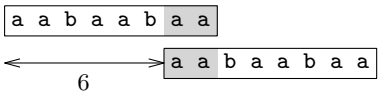


Figure 1.4 Dualité entre les notions de bords et périodes. Le mot `aa` est un bord du mot `aabaabaa` ; il lui correspond la période $6 = |\texttt{aabaabaa}| - |\texttt{aa}|$.

Alors $x = (uv)^k u$ et $|uv| = p$. Cela montre l'existence du triplet (u, v, k) et achève la preuve de la propriété.

$2 \Rightarrow 3$: il suffit de considérer le mot $t = uv$.

$3 \Rightarrow 4$: soit w le suffixe de x défini par $w = t^{-1}x$. Comme $x \preceq_{préf} t^k$, w est également un préfixe de x. D'où l'existence des deux mots u $(= t)$ et v tels que $x = uw = wv$ et $|u| = |v| = |t| = p$.

$4 \Rightarrow 5$: puisque $uw \preceq_{préf} uwv$, on a $x \preceq_{préf} tx$ avec $|t| = p$ en posant simplement $t = u$.

$5 \Rightarrow 1$: soit i un entier tel que $0 \leq i \leq |x| - p - 1$. Alors :

$$\begin{aligned} x[i+p] &= (tx)[i+p] && (\text{car } x \preceq_{préf} tx) \\ &= x[i] && (\text{car } |t| = p)\ . \end{aligned}$$

Ce qui montre que p est une période de x. ∎

On remarque en particulier que la propriété 3 s'exprime de façon un peu plus générale en remplaçant $\preceq_{préf}$ par $\preceq_{fact}$ (exercice 1.4).

Un ***bord*** d'un mot non vide x est un facteur propre de x qui est à la fois un préfixe et un suffixe de x. Ainsi, ε, `a`, `aa` et `aabaa` sont-ils les bords du mot `aabaabaa`.

Les notions de bords et de périodes sont duales comme le montre la propriété 4 de la proposition précédente (voir figure 1.4). La proposition qui suit exprime cette dualité en termes différents.

On introduit la fonction $Bord\colon A^* \to A^*$ définie pour tout mot non vide x par :

$Bord(x) =$ le plus long des bords de x .

On dit de $Bord(x)$ qu'il est ***le bord*** de x. Par exemple, le bord de tout mot de longueur 1 est le mot vide et celui du mot `aabaabaa` est `aabaa`. Remarquons également que, lorsqu'il est défini, le bord d'un bord d'un mot x donné est aussi un bord de x.

Proposition 1.5
Soient x un mot non vide et n le plus grand des entiers k pour lequel $Bord^k(x)$ est défini (soit $Bord^n(x) = \varepsilon$). Alors :

$$\langle Bord(x), Bord^2(x), \ldots, Bord^n(x)\rangle \qquad (1.1)$$

est la suite des bords de x classés par ordre décroissant de longueur, et :

$$\langle |x| - |Bord(x)|, |x| - |Bord^2(x)|, \ldots, |x| - |Bord^n(x)|\rangle \qquad (1.2)$$

est la suite des périodes de x classées en ordre croissant.

Preuve On procède par récurrence sur la longueur des mots. L'énoncé de la proposition est valide lorsque la longueur du mot x est égale à 1, la suite des bords étant réduite à $\langle \varepsilon \rangle$ et celle des périodes à $\langle |x| \rangle$.

Soit x un mot de longueur supérieure à 2. Alors tout bord de x différent de $Bord(x)$ est un bord de $Bord(x)$, et réciproquement. Il vient par récurrence que la suite (1.1) est exactement la suite des bords de x. Maintenant, si p est une période de x, la proposition 1.4 assure l'existence de trois mots u, v et w tels que $x = uw = wv$ et $|u| = |v| = p$. Alors w est un bord de x et $p = |x| - |w|$. Il s'ensuit que la suite (1.2) est bien la suite des périodes de x. ∎

Lemme 1.6 (lemme de périodicité)
Si p et q sont des périodes d'un mot non vide x telles que :

$$p + q - \mathrm{pgcd}(p, q) \leq |x| \ ,$$

alors $\mathrm{pgcd}(p, q)$ *est aussi une période de x.*

Preuve Par récurrence sur $\max\{p, q\}$. Le résultat est trivial lorsque $p = q = 1$ et, de façon plus générale, lorsque $p = q$. On peut dès lors supposer pour toute la suite que $p > q$.

D'après la proposition 1.4, le mot x s'écrit à la fois sous la forme uy avec $|u| = p$ et y un bord de x et sous la forme vz avec $|v| = q$ et z un bord de x.

Alors $p - q$ est une période de z. En effet, puisque $p > q$, y est un bord de x de longueur strictement inférieure à celle du bord z. Donc y est un bord de z. Il s'ensuit que $|z| - |y|$ est une période de z. Or $|z| - |y| = (|x| - q) - (|x| - p) = p - q$.

Mais q est également une période de z. En effet, puisque $p > q$ et $\mathrm{pgcd}(p, q) \leq p - q$, on obtient $q \leq p - \mathrm{pgcd}(p, q)$. On a d'autre part $p - \mathrm{pgcd}(p, q) = p + q - \mathrm{pgcd}(p, q) - q \leq |x| - q = |z|$. Il s'en déduit que $q \leq |z|$. Ce qui montre que la période q de x est également une période de son facteur z.

De plus, on a $(p-q)+q-\mathrm{pgcd}(p-q, q) = p-\mathrm{pgcd}(p, q)$, qui, comme on l'a vu ci-dessus, est une quantité inférieure à $|z|$.

On applique l'hypothèse de récurrence à $\max\{p - q, q\}$ relativement au mot z, et l'on obtient ainsi que $\mathrm{pgcd}(p, q)$ est une période de z.

Les conditions sur p et q (celle du lemme et $\mathrm{pgcd}(p, q) \leq p - q$) entrainent $q \leq |x|/2$. Et comme $x = vz$ et que z est un bord de x, v est un préfixe de z. Il a de plus une longueur qui est un multiple de $\mathrm{pgcd}(p, q)$. Soit t le préfixe de x de longueur $\mathrm{pgcd}(p, q)$. Alors v est une puissance de t et z est un préfixe d'une puissance de t. Il vient ensuite de la proposition 1.4 que x est un préfixe d'une puissance de t, et donc que $|t| = \mathrm{pgcd}(p, q)$ est une période de x. ∎

Pour illustrer ce résultat, considérons un mot x qui admette à la fois 5 et 8 comme périodes. Alors, si l'on suppose de plus que x est

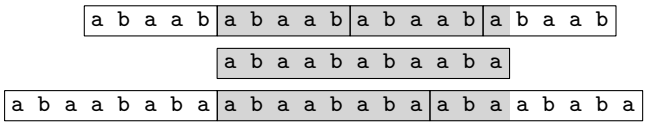


Figure 1.5 Application du lemme de périodicité. Le mot `abaababaaba` de longueur 11 possède 5 et 8 comme périodes. Il n'est pas possible de le prolonger sur la gauche comme sur la droite tout en conservant ces deux périodes. En effet, si 5 et 8 sont des périodes d'un certain mot, mais que 1 – le plus grand commun diviseur de 5 et 8 – ne l'est pas, alors ce mot est de longueur strictement inférieure à $5 + 8 - \text{pgcd}(5, 8) = 12$.

composé d'au moins deux lettres distinctes, $\text{pgcd}(5, 8) = 1$ n'est pas une période de x, et, par application du lemme, la longueur de x est strictement inférieure à $5 + 8 - \text{pgcd}(5, 8) = 12$. C'est par exemple le cas des quatre mots de longueur supérieure à 8 qui sont des préfixes du mot `abaababaaba` de longueur 11. Une autre illustration du résultat est proposée figure 1.5.

On veut montrer dans ce qui suit que l'on ne peut pas affaiblir la condition requise sur les périodes dans l'énoncé du lemme de périodicité. Plus précisément, on donne l'exemple de mots x qui possèdent deux périodes p et q telles que $p+q-\text{pgcd}(p, q) = |x|+1$ mais qui ne satisfont pas à la conclusion du lemme. (Voir aussi exercice 1.5.)

Soit $\beta\colon A^* \to A^*$ la fonction définie par :

$\beta(uab) = uba$

pour tout mot $u \in A^*$ et toutes lettres $a, b \in A$.

Lemme 1.7
Pour tout naturel $n \geq 3$, $\beta(f_n) = f_{n-2}f_{n-1}$.

Preuve Par récurrence sur n. Le résultat est trivial lorsque $3 \leq n \leq 4$. Si $n \geq 5$, on a :

$$\begin{aligned}
\beta(f_n) &= \beta(f_{n-1}f_{n-2}) && \text{(par définition de } f_n\text{)}\\
&= f_{n-1}\beta(f_{n-2}) && \text{(car } |f_{n-2}| = F_{n-2} \geq 2\text{)}\\
&= f_{n-1}f_{n-4}f_{n-3} && \text{(par hypothèse de récurrence)}\\
&= f_{n-2}f_{n-3}f_{n-4}f_{n-3} && \text{(par définition de } f_{n-1}\text{)}\\
&= f_{n-2}f_{n-2}f_{n-3} && \text{(par définition de } f_{n-2}\text{)}\\
&= f_{n-2}f_{n-1} && \text{(par définition de } f_{n-1}\text{) .}
\end{aligned}$$ ∎

Pour tout naturel $n \geq 3$, on définit le mot g_n comme le préfixe de longueur $F_n - 2$ de f_n, c'est-à-dire f_n privé de ses deux dernières lettres.

Lemme 1.8
Pour tout naturel $n \geq 6$, $g_n = {f_{n-2}}^2 g_{n-3}$.

Preuve On a :

$$\begin{array}{rcll} f_n & = & f_{n-1}f_{n-2} & \text{(par définition de } f_n\text{)} \\ & = & f_{n-2}f_{n-3}f_{n-2} & \text{(par définition de } f_{n-1}\text{)} \\ & = & f_{n-2}\beta(f_{n-1}) & \text{(d'après le lemme 1.7)} \\ & = & f_{n-2}\beta(f_{n-2}f_{n-3}) & \text{(par définition de } f_{n-1}\text{)} \\ & = & {f_{n-2}}^2\beta(f_{n-3}) & \text{(car } |f_{n-3}| = F_{n-3} \geq 2\text{)} \ . \end{array}$$

Le résultat annoncé s'en déduit immédiatement. ∎

Lemme 1.9
Pour tout naturel $n \geq 3$, $g_n \preceq_{préf} {f_{n-1}}^2$ et $g_n \preceq_{préf} {f_{n-2}}^3$.

Preuve On a :

$$\begin{array}{rcll} g_n & \preceq_{préf} & f_nf_{n-3} & \text{(car } g_n \preceq_{préf} f_n\text{)} \\ & = & f_{n-1}f_{n-2}f_{n-3} & \text{(par définition de } f_n\text{)} \\ & = & {f_{n-1}}^2 & \text{(par définition de } f_{n-1}\text{)} \ . \end{array}$$

La seconde relation est valide lorsque $3 \leq n \leq 5$. Lorsque $n \geq 6$, on a :

$$\begin{array}{rcll} g_n & = & {f_{n-2}}^2g_{n-3} & \text{(d'après le lemme 1.8)} \\ & \preceq_{préf} & {f_{n-2}}^2f_{n-3}f_{n-4} & \text{(car } g_{n-3} \preceq_{préf} f_{n-3}\text{)} \\ & = & {f_{n-2}}^3 & \text{(par définition de } f_{n-2}\text{)} \ . \end{array}$$ ∎

Maintenant, soit n un naturel tel que $n \geq 5$, de telle sorte que le mot g_n soit à la fois défini et de longueur supérieure à 2. Il vient alors :

$$\begin{array}{rcll} |g_n| & = & F_n - 2 & \text{(par définition de } g_n\text{)} \\ & = & F_{n-1} + F_{n-2} - 2 & \text{(par définition de } F_n\text{)} \\ & \geq & F_{n-1} & \text{(car } F_{n-2} \geq 2\text{)} \ . \end{array}$$

Il résulte de cette inégalité, du lemme 1.9 et de la proposition 1.4 que F_{n-1} et F_{n-2} sont deux périodes de g_n. Remarquons également que, puisque $\text{pgcd}(F_{n-1}, F_{n-2}) = 1$, on a aussi :

$$\begin{array}{rcl} F_{n-1} + F_{n-2} - \text{pgcd}(F_{n-1}, F_{n-2}) & = & F_n - 1 \\ & = & |g_n| + 1 \ . \end{array}$$

Si donc la conclusion du lemme de périodicité s'appliquait au mot g_n et ses deux périodes F_{n-1} et F_{n-2}, g_n serait la puissance d'un mot de longueur 1. Or les deux premières lettres de g_n sont distinctes. Ce qui indique que la condition du lemme de périodicité est en un certain sens optimale.

Puissances, primitivité et conjugaison

Lemme 1.10
Soient x et y deux mots. S'il existe deux naturels non nuls m et n tels que $x^m = y^n$, x et y sont des puissances d'un certain mot z.

Preuve Il suffit de montrer le résultat dans le cas non trivial où ni x ni y ne sont vides. Deux sous-cas peuvent alors être distingués, selon que $\min\{m, n\}$ est égal ou non à 1.

Si $\min\{m, n\} = 1$, il suffit de considérer le mot $z = y$ si $m = 1$ et $z = x$ sinon.

Sinon, $\min\{m, n\} \geq 2$. On remarque alors que $|x|$ et $|y|$ sont des périodes du mot $t = x^m = y^n$ qui vérifient la condition du lemme de périodicité : $|x| + |y| - \text{pgcd}(|x|, |y|) \leq |x| + |y| - 1 < |t|$. En conséquence de quoi il suffit de considérer le mot z défini comme le préfixe de t de longueur $\text{pgcd}(|x|, |y|)$ pour obtenir le résultat annoncé. ∎

Un mot non vide est ***primitif*** s'il n'est puissance d'aucun autre mot que lui-même. Autrement dit, un mot $x \in A^+$ est primitif si et seulement si toute décomposition de la forme $x = u^n$ avec $u \in A^*$ et $n \in \mathbf{N}$ implique $n = 1$. Par exemple, le mot `abaab` est primitif, tandis que les mots ε et `bababa` $= (\texttt{ba})^3$ ne le sont pas.

Lemme 1.11 (lemme de primitivité)
Un mot non vide est primitif si et seulement si il n'est un facteur de son carré qu'en tant que préfixe et suffixe. Autrement dit, pour tout mot non vide x :

x primitif

si et seulement si

$yx \preceq_{préf} x^2$ implique $y = \varepsilon$ ou $y = x$

si et seulement si

$pér(x^2) = |x|$.

Une illustration de ce résultat est proposée figure 1.6.

Preuve Si x est un mot non vide non primitif, il existe $z \in A^+$ et $n \geq 2$ tels que $x = z^n$. Puisque x^2 se décompose sous la forme $z \cdot z^n \cdot z^{n-1}$, le mot x apparait à la position $|z|$ sur x^2. Cela montre que tout mot non vide non primitif est un facteur de son carré sans en être seulement un préfixe et un suffixe.

Réciproquement, soit x un mot non vide tel que son carré x^2 s'écrive sous la forme yxz avec $y, z \in A^+$. Pour une question de longueur, il vient tout d'abord que $|y| < |x|$. Ensuite, et puisque $x \preceq_{préf} yx$, on obtient de

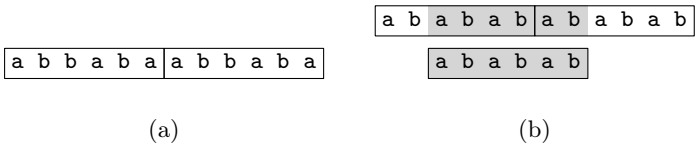


Figure 1.6 Application du lemme de primitivité. **(a)** Le mot $x = \texttt{abbaba}$ ne possède pas d'occurrence « non triviale » dans son carré x^2 – c'est-à-dire qui ne soit ni un préfixe ni un suffixe de x^2 – car x est primitif. **(b)** Le mot $x = \texttt{ababab}$ possède une occurrence « non triviale » dans son carré x^2 car x n'est pas primitif : $x = (\texttt{ab})^3$.

la proposition 1.4 que $|y|$ est une période de x. Ainsi $|x|$ et $|y|$ sont-elles des périodes de yx. D'après le lemme de périodicité, on en déduit que $p = \text{pgcd}(|x|, |y|)$ est également une période de yx. Maintenant, comme $p \leq |y| < |x|$, p est aussi une période de x. Et comme p divise $|x|$, on en déduit que x est de la forme t^n avec $|t| = p$ et $n \geq 2$. Cela montre que le mot x n'est pas primitif. ∎

Proposition 1.12
Pour tout mot non vide, il existe un et un seul mot primitif dont il est une puissance.

Preuve La preuve de l'existence découle d'une récurrence triviale sur la longueur des mots. On s'attache maintenant à montrer l'unicité.

Soit x un mot non vide. Si l'on suppose que $x = u^m = v^n$ pour deux mots primitifs u et v et deux naturels non nuls m et n, alors u et v sont nécessairement des puissances d'un mot $z \in A^+$ d'après le lemme 1.10. Il s'ensuit que $z = u = v$, ce qui montre l'unicité et termine la preuve. ∎

Si x est un mot non vide, on dit du mot primitif z dont x est la puissance qu'il est la ***racine*** de x, et du naturel n tel que $x = z^n$ qu'il est l'***exposant***[3] de x.

Deux mots x et y sont ***conjugués*** s'il existe deux mots u et v tels que $x = uv$ et $y = vu$. Par exemple, les mots $\texttt{abaab}$ et $\texttt{ababa}$ sont conjugués. Il est clair que la conjugaison est une relation d'équivalence. Elle n'est pas compatible avec le produit.

Proposition 1.13
Deux mots non vides sont conjugués si et seulement si leurs racines le sont également.

Preuve La preuve de la réciproque est immédiate.

Pour la preuve de l'implication directe, on considère deux mots conjugués non vides x et y dont on note z et t puis m et n les racines puis les

3. De manière plus générale, l'exposant de x est la quantité $|x|/pér(x)$ qui n'est pas nécessairement entière (voir exercice 9.2).

exposants respectifs. Puisque x et y sont conjugués, il existe $z', z'' \in A^+$ et $p, q \in \mathbf{N}$ tels que $z = z'z''$, $x = z^p z' \cdot z''z^q$, $y = z''z^q \cdot z^p z'$ et $m = p + q + 1$. On en déduit que $y = (z''z')^m$. Maintenant, comme t est primitif, le lemme 1.10 entraine que $z''z'$ est une puissance de t. D'où l'existence d'un naturel non nul k tel que $|z| = k|t|$. Par symétrie, il existe un naturel non nul ℓ tel que $|t| = \ell|z|$. Il s'ensuit que $k = \ell = 1$, que $|t| = |z|$, puis que $t = z''z'$. Cela montre que les mots z et t sont conjugués. ∎

Proposition 1.14
Deux mots non vides x et y sont conjugués si et seulement si il existe un mot z tel que $xz = zy$.

Preuve $\Rightarrow$: si x et y se décomposent sous la forme $x = uv$ et $y = vu$ avec $u, v \in A^*$, alors le mot $z = u$ convient puisque $xz = uvu = zy$.

$\Leftarrow$: dans le cas non trivial où $z \in A^+$, on obtient par une récurrence immédiate que $x^k z = zy^k$ pour tout $k \in \mathbf{N}$. Soit maintenant n le naturel (non nul) tel que $(n-1)|x| \leq |z| < n|x|$. Il existe donc $u, v \in A^*$ tels que $x = uv$, $z = x^{n-1}u$ et $vz = y^n$. Il s'en déduit que $y^n = vx^{n-1}u = (vu)^n$. Finalement, puisque $|y| = |x|$, on a $y = vu$, ce qui montre que x et y sont conjugués. ∎

1.3 Algorithmes et complexité

Dans cette section, on présente des éléments algorithmiques utilisés dans la suite de l'ouvrage. Ils comprennent les conventions d'écriture, l'évaluation de la complexité des algorithmes et quelques objets standard.

Convention d'écriture des algorithmes

Le style du langage d'expression algorithmique employé ici est relativement proche des langages de programmation réels mais se place à un niveau d'abstraction supérieur.

On adopte les conventions suivantes :

- l'indentation signifie la structure de bloc inhérente aux instructions composées ;
- les lignes de code sont numérotées à la seule fin de pouvoir être référencées dans le texte ;
- le symbole ▷ introduit un commentaire ;
- l'accès à un attribut particulier d'un objet est signifié par le nom de l'attribut suivi de l'identificateur associé à l'objet entre crochets ;
- une variable qui représente un objet donné (table, file, arbre, mot, automate) est un pointeur vers cet objet ;

– les paramètres passés lors d'appels de procédures ou de fonctions le sont par valeur ;
– la portée des variables des procédures et des fonctions est locale sauf mention contraire ;
– l'évaluation des expressions booléennes s'effectue de la gauche vers la droite de façon paresseuse.

On considère, à l'instar d'un langage comme le C, l'instruction itérative **faire-tantque** – utilisée en lieu et place de l'instruction traditionnelle **répéter-jusque** – et l'instruction **rupture** qui provoque la terminaison de la boucle la plus interne dans laquelle elle figure.

Adaptée au traitement séquentiel de mots, on utilise la formulation :

```
1  pour chaque lettre a de u, séquentiellement faire
2          traitement de a
```

pour tout mot u. Elle signifie que les lettres $u[i]$, $i = 0, 1, \ldots, |u| - 1$, composant u sont traitées les unes à la suite des autres dans le corps de la boucle : d'abord $u[0]$, puis $u[1]$, et ainsi de suite. Elle sous-entend que la longueur du mot u n'a pas nécessairement à être connue à l'avance, l'arrêt de la boucle pouvant se réaliser à l'aide d'un marqueur de fin de mot. Dans le cas où la longueur du mot u est connue, cette formulation est équivalente à une formulation du type :

```
1  pour i ← 0 à |u| − 1 faire
2          a ← u[i]
3          traitement de a
```

où la variable entière i est libre (son utilisation ne provoque pas de conflit avec l'environnement).

Algorithmes de recherche de motifs

Un ***motif*** représente un langage non vide ne contenant pas le mot vide. Il peut être décrit par un mot, par un ensemble fini de mots, ou autrement. Le problème de la ***recherche de motifs*** est celui de la localisation d'occurrences de mots du langage dans d'autres mots – ou dans des ***textes*** pour parler de manière moins formelle. Les notions d'occurrence, d'apparition et de position sur les mots sont étendues aux motifs.

Selon le problème spécifié, l'entrée d'un algorithme de recherche d'un motif est un mot x, un langage X ou autre, et un texte y, accompagnés ou non de leurs longueurs.

La sortie peut prendre plusieurs formes. En voici quelques unes :

– pour implanter un algorithme qui teste si le motif apparait dans le texte ou non, sans précision sur les positions des occurrences éventuelles, la sortie est simplement la valeur booléenne VRAI dans la première éventualité et FAUX dans la seconde ;

– lors d'une recherche séquentielle, il est opportun de produire un mot $\bar{y}$ sur l'alphabet $\{\mathtt{0}, \mathtt{1}\}$ qui code l'existence de positions droites d'occurrences. Le mot $\bar{y}$ est tel que $|\bar{y}| = |y|$ et $\bar{y}[i] = \mathtt{1}$ si et seulement si i est la position droite d'une occurrence du motif dans y ;
– la sortie peut aussi prendre la forme d'un ensemble P de positions gauches – ou droites – d'occurrences du motif dans y.

Soit e un prédicat de valeur VRAI si et seulement si une occurrence vient d'être localisée. Une fonction correspondant à la première des formes évoquées et prenant fin dès qu'une occurrence est localisée se doit d'intégrer dans son code une instruction comme :

```
1  si e alors
2          retourner VRAI
```

au cœur même de son procédé de recherche et de retourner la valeur FAUX à la terminaison de ce procédé. Pour la deuxième forme, il s'agit d'initialiser la variable $\bar{y}$ à ε, le mot vide, puis de modifier sa valeur par une instruction comme :

```
1  si e alors
2          ȳ ← ȳ · 1
3  sinon ȳ ← ȳ · 0
```

puis de la retourner à la terminaison. De même pour la troisième forme, où l'ensemble P est initialement vide, puis augmenté par une instruction comme :

```
1  si e alors
2          P ← P ∪ {la position courante sur y}
```

puis enfin retourné.

Afin de ne pas avoir à présenter plusieurs variantes du code d'un même algorithme, on considère l'instruction spéciale suivante :

SIGNALER-SI(e) signifie, à l'endroit où elle est donnée, la localisation d'une occurrence du motif à la position courante sur le texte à la condition que le prédicat e ait pour valeur VRAI.

Expression de la complexité

Le modèle de calcul pour l'évaluation de la complexité des algorithmes est le modèle standard de machine à accès direct.

De façon générale, la complexité des algorithmes se traduit par des expressions intégrant la taille de l'entrée. Ce qui inclut la longueur du langage représenté par le motif, la longueur du mot sur lequel s'effectue la recherche et la taille de l'alphabet. On suppose que les lettres de l'alphabet sont de taille comparable à celle des mots machine, et, qu'en

conséquence, la comparaison de deux lettres entre elles est une opération élémentaire qui s'effectue en temps constant.

On suppose que toute instruction SIGNALER-SI(e) s'exécute en temps constant [4] une fois le prédicat e évalué.

On utilise les notations préconisées par Knuth pour exprimer les ordres de grandeur. Soient f et g deux fonctions de $\mathbf{N}$ dans $\mathbf{N}$. On écrit « $f(n)$ est $O(g(n))$ » pour signifier qu'il existe une constante K et un naturel n_0 tels que $f(n) \leq Kg(n)$ pour tout $n \geq n_0$. De façon duale, on écrit « $f(n)$ est $\Omega(g(n))$ » s'il existe une constante K et un naturel n_0 tels que $f(n) \geq Kg(n)$ pour tout $n \geq n_0$. On écrit enfin « $f(n)$ est $\Theta(g(n))$ » pour signifier que f et g sont du même ordre, à savoir que $f(n)$ est à la fois $O(g(n))$ et $\Omega(g(n))$.

La fonction $f\colon \mathbf{N} \to \mathbf{N}$ est ***linéaire*** si $f(n)$ est $\Theta(n)$, ***quadratique*** si $f(n)$ est $\Theta(n^2)$, ***cubique*** si $f(n)$ est $\Theta(n^3)$, ***logarithmique*** si $f(n)$ est $\Theta(\log n)$, ***exponentielle*** s'il existe $a > 0$ pour lequel $f(n)$ est $\Theta(a^n)$.

On dit d'une fonction de deux paramètres $f\colon \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ qu'elle est linéaire lorsque $f(m, n)$ est $\Theta(m + n)$ et quadratique lorsque $f(m, n)$ est $\Theta(m \times n)$.

Quelques objets standard

Les files, les états et les automates sont des objets souvent utilisés dans la suite. Sans préjuger de leurs implantations effectives – pouvant d'ailleurs différer d'un algorithme à l'autre – on indique les attributs et les opérations définis au minimum sur ces objets.

Pour les files, on se contente de ne décrire que les opérations de base.

FILE-VIDE() crée puis retourne une file vide.

FILE-EST-VIDE(F) retourne VRAI si la file F est vide, et FAUX sinon.

ENFILER(F, x) ajoute l'élément x en queue de la file F.

TÊTE(F) retourne l'élément situé en tête de la file F.

DÉFILER(F) supprime l'élément situé en tête de la file F.

DÉFILEMENT(F) supprime l'élément situé en tête de la file F puis le retourne.

LONGUEUR(F) retourne la longueur de la file F.

Les états sont des objets qui possèdent au moins les deux attributs *terminal* et *Succ*. Le premier indique si l'état est terminal ou non et le second est une implantation de l'ensemble des successeurs étiquetés de l'état. L'attribut correspondant à une sortie d'un état est noté *sortie*.

4. On peut en fait toujours s'y ramener quand bien même le langage représenté par le motif ne serait pas réduit à un seul mot. Il suffit pour cela de ne produire qu'un descripteur – préalablement calculé – de l'ensemble des mots qui apparaissent à la position courante (au lieu par exemple de produire explicitement l'ensemble des mots). Reste ensuite à la charge d'un utilitaire de développer l'information si nécessaire.

Les deux opérations standard sur les états sont les fonctions NOUVEL-ÉTAT et CIBLE. Tandis que la première crée puis retourne un état non terminal d'ensemble des successeurs étiquetés vide, la seconde retourne la cible d'une flèche dont ne sont données que la source et l'étiquette, ou la valeur spéciale NIL si une telle flèche n'existe pas. Le code de ces deux fonctions s'écrit en quelques lignes :

```
NOUVEL-ÉTAT()
1  allouer un objet p de type état
2  terminal[p] ← FAUX
3  Succ[p] ← ∅
4  retourner p
```

```
CIBLE(p, a)
1  si il existe un état q tel que (a, q) ∈ Succ[p] alors
2        retourner q
3  sinon retourner NIL
```

Les objets du type automate possèdent au moins l'attribut *initial* qui spécifie l'état initial de l'automate. La fonction NOUVEL-AUTOMATE crée puis retourne un automate à un seul état posé comme étant son état initial, d'ensemble des successeurs étiquetés vide. Le code correspondant est le suivant :

```
NOUVEL-AUTOMATE()
1  allouer un objet M de type automate
2  q0 ← NOUVEL-ÉTAT()
3  initial[M] ← q0
4  retourner M
```

1.4 Mise en mémoire d'automates

Certains algorithmes de recherche de motifs reposent sur des implantations particulières des automates déterministes qu'ils considèrent. Cette section détaille plusieurs procédés, comprenant les structures de données et les algorithmes, qui peuvent être utilisés pour implanter ces objets en mémoire.

Implanter un automate déterministe (Q, q_0, T, F) revient à mettre en mémoire, soit l'ensemble F de ses flèches, soit les ensembles des successeurs étiquetés de ses états, soit sa fonction de transition δ. Ce sont là des problèmes équivalents qui rentrent dans le cadre général du problème de la représentation des ***fonctions partielles*** (exercice 1.15). On distingue deux familles d'implantations :

– la famille des implantations ***pleines*** dans lesquelles figurent toutes les transitions ;

	a	b	c
0	1	0	0
1	2	3	0
2	2	3	0
3	4	0	0
4	2	3	0

Figure 1.7 La matrice de transition de l'automate de la figure 1.2.

– la famille des implantations ***réduites*** qui font appel à des techniques plus ou moins élaborées de compression destinées à réduire l'espace mémoire de la représentation.

Le choix de l'implantation influence le temps nécessaire au calcul d'une transition, c'est-à-dire à l'exécution de CIBLE(p, a), pour tout état $p \in Q$ et toute lettre $a \in A$. Ce temps de calcul est appelé le ***délai*** en cela qu'il mesure également le temps nécessaire pour passer de la lettre courante sur l'entrée à la lettre qui la suit. Typiquement, deux modèles peuvent être opposés :

– le ***modèle branchements*** dans lequel δ est implantée à l'aide d'une matrice $Q \times A$ et où le délai est constant ;
– le ***modèle comparaisons*** dans lequel l'opération élémentaire est la comparaison de lettres et où le délai est $O(\log \operatorname{card} A)$, deux lettres quelconques pouvant être comparées en une unité de temps (hypothèse générale formulée dans la section 1.3).

On considère également dans la section suivante une technique élémentaire dite « modèle vecteur-binaire » dont le champ d'application est restreint : elle ne présente d'intérêt que lorsque la taille de l'automate est très faible.

Pour chacune des familles d'implantations, on précise les ordres de grandeur de l'espace mémoire nécessaire et du délai. Il y a toujours un compromis à trouver entre ces deux quantités.

Implantations pleines

La méthode la plus simple pour implanter la fonction δ est de ranger ses valeurs dans une matrice $Q \times A$, dite ***matrice de transition*** (une illustration est donnée figure 1.7). Il s'agit là d'une méthode de choix pour un automate déterministe complet sur un alphabet de taille relativement faible et lorsque les lettres peuvent être assimilées à des indices sur une table.

Proposition 1.15
Dans une implantation par matrice de transition, l'espace mémoire nécessaire est $O(\operatorname{card} Q \times \operatorname{card} A)$ *et le délai* $O(1)$. ∎

Dans le cas où l'automate n'est pas complet, la représentation reste correcte excepté le fait que l'exécution de l'automate sur le texte donné en entrée peut maintenant s'arrêter sur une transition non définie. La matrice peut toutefois être initialisée en temps $O(\operatorname{card} F)$ si l'on fait appel à une technique d'implantation des fonctions partielles comme celle proposée exercice 1.15. Les complexités annoncées plus haut tant pour l'espace mémoire nécessaire que pour le délai restent valides.

Un automate peut être implanté au moyen de matrices d'adjacence comme il est classique de le faire pour les graphes. On associe alors à chaque lettre de l'alphabet une matrice définie sur $Q \times Q$ à valeurs booléennes. Cette représentation n'est en général pas adaptée pour les applications développées dans cet ouvrage. Elle est cependant reliée à la méthode qui suit.

La méthode par ***liste de transitions*** consiste à implanter une liste des triplets (p, a, q) qui sont des flèches de l'automate. L'espace nécessaire est seulement $O(\operatorname{card} F)$. Ce faisant, on suppose que cette liste est rangée dans une table de hachage de façon à permettre un calcul rapide des transitions. La fonction de hachage correspondante est définie sur les couples $(p, a) \in Q \times A$. Étant donné un couple (p, a), l'accès à la transition (p, a, q), si elle est définie, se fait en moyenne en temps constant sous les habituelles hypothèses propres à ce type de technique.

Ces premiers types de représentations supposent implicitement que l'alphabet est fixé et connu à l'avance, ce qui les opposent aux représentations du modèle comparaisons considérées par la méthode exposée ci-dessous.

La méthode par ***ensembles des successeurs étiquetés*** consiste à utiliser une table t indicée sur Q dont chaque élément $t[p]$ donne accès à une implantation de l'ensemble des successeurs étiquetés de l'état p. L'espace nécessaire est $O(\operatorname{card} Q + \operatorname{card} F)$. Cette méthode est valable même lorsque la seule opération autorisée sur les lettres est la comparaison. En notant s le maximum des degrés sortants des états, le délai est $O(\log s)$ si l'on a recours à une implantation efficace des ensembles de successeurs étiquetés.

Proposition 1.16
Dans une implantation par ensembles des successeurs étiquetés, l'espace mémoire nécessaire est $O(\operatorname{card} Q + \operatorname{card} F)$ *et le délai* $O(\log s)$ *où* s *est le maximum des degrés sortants des états.* ∎

Remarquons que le délai est également $O(\log \operatorname{card} A)$ dans ce cas : en effet, l'automate étant supposé déterministe, le degré sortant de chacun des états est inférieur à $\operatorname{card} A$, soit $s \leq \operatorname{card} A$ avec les notations utilisées ci-dessus.

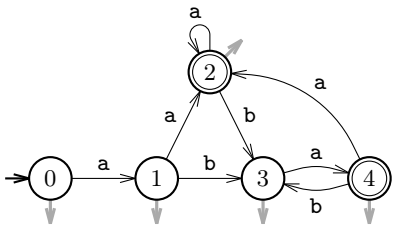


Figure 1.8 Implantation réduite par adjonction de successeurs par défaut. On considère l'automate de la figure 1.2 et l'on choisit l'état initial comme unique successeur par défaut (ce choix convient parfaitement pour les problèmes de recherche de motifs). Ceux des états qui admettent l'état initial comme successeur par défaut (en fait tous ici) sont indiqués par une courte flèche grisée. La cible de la transition de l'état 3 par la lettre `a` est l'état 4, et par toute autre lettre, ici `b` ou `c`, l'état initial 0.

Implantations réduites

Lorsque l'automate est complet, la complexité en espace peut être réduite en considérant un ***successeur par défaut*** pour le calcul des transitions à partir de n'importe quel état donné – l'état apparaissant le plus souvent dans un ensemble des successeurs étiquetés étant le meilleur des candidats possibles au titre de successeur par défaut. Le délai peut du même coup s'en trouver réduit puisque la taille des ensembles des successeurs étiquetés est moins importante. Pour les problèmes de recherche de motifs, le choix de l'état initial comme successeur par défaut convient parfaitement. Aussi convient-on d'indiquer qu'un état possède l'état initial comme successeur par défaut par une courte flèche grisée sortant de ce premier état (un exemple est montré figure 1.8.)

Une autre méthode consiste à utiliser une fonction de suppléance. L'idée est ici de réduire l'espace nécessaire à l'implantation de l'automate, en renvoyant, dans la plupart des cas, le calcul de la transition à partir de l'état courant à celui du calcul à partir d'un autre état mais pour la même lettre. Cette technique sert à implanter des automates déterministes dans le modèle comparaisons. Son principal avantage est – en général – de fournir des représentations de taille linéaire et d'obtenir simultanément un temps linéaire pour le calcul de séries de transitions quand bien même le calcul d'une seule transition ne se fait pas en temps constant.

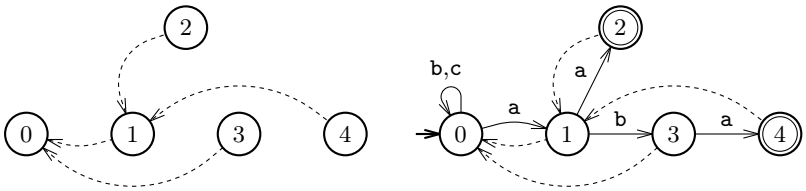
Formellement, soient :

$\gamma : Q \times A \to Q$

et :

$f : Q \to Q$

deux fonctions. On dit que le couple (γ, f) représente la fonction de



(a) (b)

Figure 1.9 Implantation réduite par adjonction d'une fonction de suppléance. On reprend l'exemple de l'automate de la figure 1.2. **(a)** Une fonction de suppléance donnée sous la forme d'un graphe orienté. Comme ce graphe ne possède pas de circuit, la fonction définit un ordre sur l'ensemble des états. **(b)** L'automate réduit correspondant. Chaque lien état-état suppléant est indiqué par un arc dirigé dessiné en pointillé. Le calcul de la transition de l'état 4 par la lettre `c` est renvoyé à l'état 1, puis à l'état 0. L'état 0 est en effet le premier des états 4, 1 et 0, dans cet ordre, à posséder une transition définie pour `c`. Au final, la cible de la transition de l'état 4 par `c` est l'état 0.

transition δ d'un automate complet si et seulement si γ est une sous-fonction de δ, f définit un ordre sur Q, et pour tout couple $(p, a) \in Q \times A$:

$$\delta(p,a) = \begin{cases} \gamma(p,a) & \text{si } \gamma(p,a) \text{ est défini} \ , \\ \delta(f(p),a) & \text{sinon} \ . \end{cases}$$

Lorsque qu'il est défini, on dit de l'état $f(p)$ qu'il est le ***suppléant*** de l'état p. On dit des fonctions γ et f qu'elle sont respectivement, et conjointement, une ***sous-transition*** et une ***fonction de suppléance*** de δ.

On convient d'indiquer le lien état-état suppléant par un arc dirigé dessiné en pointillé (voir l'exemple figure 1.9).

L'espace nécessaire pour représenter la fonction δ par les fonctions γ et f est $O(\operatorname{card} Q + \operatorname{card} F')$ dans le cas d'une implantation par ensembles des successeurs étiquetés où :

$$F' = \{(p,a,q) \in F : \gamma(p,a) \text{ est défini}\} \ .$$

Notons au passage que γ est la fonction de transition de l'automate (Q, q_0, T, F').

Un exemple intégré

La méthode présentée maintenant est une combinaison des précédentes alliant un calcul rapide des transitions et une représentation compacte des transitions dues à l'utilisation conjointe de tables et d'une fonction de suppléance. Elle est connue sous le nom de « compression de table de transition ».

Deux attributs supplémentaires, *suppléant* et *base*, sont adjoints aux états, le premier à valeurs dans Q et le second dans $\mathbf{N}$. On considère aussi

deux tables indexées dans $\mathbf{N}$ et à valeurs dans Q : *cible* et *contrôle*. Pour chaque couple $(p, a) \in Q \times A$, $base[p] + rang[a]$ est un indice sur *cible* et *contrôle*, en notant *rang* la fonction qui associe à toute lettre de A son rang dans A.

Le calcul de la transition d'un état $p \in Q$ par une lettre $a \in A$ procède comme suit :

1. Si $contrôle[base[p] + rang[a]] = p$, $cible[base[p] + rang[a]]$ est la cible de la flèche de source p et d'étiquette a.
2. Sinon le traitement est répété récursivement sur l'état $suppléant[p]$ et la lettre a (à supposer que *suppléant* soit bien une fonction de suppléance).

   Suit le code (non récursif) de la fonction correspondante.

CIBLE-PAR-COMPRESSION(p, a)

```
1  tantque contrôle[base[p] + rang[a]] ≠ p faire
2          p ← suppléant[p]
3  retourner cible[base[p] + rang[a]]
```

Dans le pire des cas, l'espace nécessaire est $O(\operatorname{card} Q \times \operatorname{card} A)$ et le délai $O(\operatorname{card} Q)$. Cela dit, cette méthode permet de réduire l'espace en $O(\operatorname{card} Q + \operatorname{card} A)$ avec un délai constant dans le meilleur des cas.

1.5 Techniques de base

On présente dans cette section des approches élémentaires au problème de la recherche de motifs. Elles comprennent la notion de fenêtre glissante commune à de nombreux algorithmes de recherche, l'utilisation d'heuristiques pour réduire le temps de calcul, la méthode générale basée sur un automate quand les textes doivent être traité de manière séquentielle et l'emploi de techniques qui reposent sur le codage binaire réalisé par les mots machine.

Notion de fenêtre glissante

Lorsque le motif est un mot non vide x de longueur m, il est agréable de considérer que le texte y de longueur n sur lequel s'effectue la localisation est examiné au moyen d'une ***fenêtre glissante***. La fenêtre délimite un facteur sur le texte – appelé le ***contenu de la fenêtre*** – qui a, dans la plupart des cas, la longueur du mot x. Elle glisse le long du texte de la gauche vers la droite.

La fenêtre étant à une position j donnée sur le texte, l'algorithme teste si le mot x apparait ou non à cette position, en comparant certaines lettres du contenu de la fenêtre avec les lettres du mot alignées en correspondance. On parle de ***tentative*** à la position j (voir l'exemple

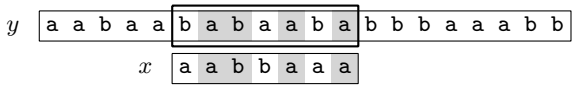


Figure 1.10 Une tentative de localisation du mot $x =$ `aabbaaa` dans le mot $y =$ `aabaababaababbbaaabb`. La tentative a lieu à la position 5 sur y. Le contenu de la fenêtre et le mot coïncident en quatre positions.

figure 1.10). Le cas échéant, l'occurrence est signalée. Lors de cette phase de test, l'algorithme acquiert une certaine quantité d'informations sur le texte laquelle peut être exploitée de deux façons :

- pour déterminer la longueur du prochain ***décalage*** de la fenêtre selon des règles qui sont propres à l'algorithme ;
- pour éviter des comparaisons lors des tentatives suivantes en mémorisant une partie de l'information recueillie.

Quand le décalage fait passer la fenêtre de la position j à la position $j + d$ ($d \geq 1$), on dit que le décalage est de ***longueur*** d. Pour répondre au problème posé, un décalage de longueur d pour une tentative à la position j doit être ***valide***, c'est-à-dire qu'il doit assurer que, pour $d \geq 2$, il n'y a pas d'occurrence du mot cherché x des positions $j+1$ à $j+d-1$ sur le texte y.

L'algorithme naïf

L'implantation la plus simple du mécanisme de la fenêtre glissante est l'algorithme dit naïf. La stratégie consiste ici à considérer une fenêtre de longueur m et à la faire glisser d'une position vers la droite après chaque tentative. Ce qui conduit, dès lors que la comparaison du contenu de la fenêtre et du mot est correctement implantée, à un algorithme correct.

On donne ci-dessous le code de l'algorithme. La variable j correspond à la position gauche de la fenêtre sur le texte. Il est entendu que la comparaison des mots à la ligne 2 s'effectue lettre à lettre selon un ordre préétabli.

```
LOCALISER-NAÏVEMENT(x, m, y, n)
1  pour j ← 0 à n − m faire
2        SIGNALER-SI(y[j . . j + m − 1] = x)
```

Dans le pire des cas, l'algorithme LOCALISER-NAÏVEMENT s'exécute en temps $\Theta(m \times n)$, comme par exemple lorsque x et y sont des puissances de la même lettre. Dans le cas moyen [5] en revanche, son comportement est plutôt bon, comme l'indique la proposition suivante.

5. Quand bien même les motifs et les textes considérés dans la pratique n'ont aucune raison d'être aléatoires, les cas moyens rendent compte de ce que l'on peut attendre d'un algorithme de recherche de motifs donné.

Proposition 1.17
Sous la double hypothèse d'un alphabet non réduit à une seule lettre et d'une distribution uniforme et indépendante des lettres de l'alphabet, le nombre moyen de comparaisons de lettres effectuées par l'opération LOCALISER-NAÏVEMENT(x, m, y, n) *est* $\Theta(n - m)$.

Preuve Soit c le cardinal de l'alphabet. Le nombre de comparaisons de lettres nécessaires pour déterminer si deux mots u et v de longueur m sont identiques ou non est en moyenne :

$$1 + 1/c + \cdots + 1/c^{m-1} \ ,$$

indépendamment de la permutation sur les positions suivant laquelle sont comparées entre elles les lettres des mots. Lorsque $c \geq 2$, cette quantité est inférieure à $1/(1 - 1/c)$, elle-même inférieure à 2.

Il s'ensuit que le nombre moyen de comparaisons de lettres comptées durant l'exécution de l'opération est inférieur à $2(n - m + 1)$. D'où le résultat puisque $n-m+1$ comparaisons sont effectuées au minimum. ∎

Heuristiques

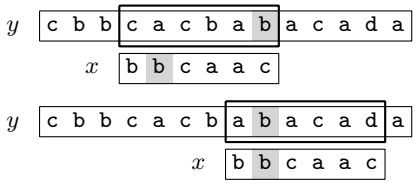
Certains procédés élémentaires améliorent sensiblement le comportement global des algorithmes. On en détaille ici quelques-uns des plus significatifs. Ils sont décrits en rapport avec l'algorithme naïf. Mais la plupart des autres algorithmes peuvent les inclure dans leur code, l'adaptation s'avérant plus ou moins aisée. On parle d'heuristiques en cela que l'on n'est pas capable de mesurer vraiment leur apport.

Quitte à localiser toutes les occurrences du mot x dans le texte y par la méthode naïve, autant commencer par localiser les occurrences de sa première lettre, $x[0]$, dans le préfixe $y[0\,..\,n-m+1]$ de y. Il restera ensuite à tester pour chaque apparition de $x[0]$ à une position j sur y l'identité éventuelle entre les deux mots $x[1\,..\,m-1]$ et $y[j+1\,..\,j+m-1]$. Comme l'opération de recherche de l'occurrence d'une lettre est généralement une opération de bas niveau des systèmes, la réduction du temps de calcul est souvent appréciable dans la pratique. Cette recherche élémentaire peut encore être améliorée de deux façons :

- en positionnant $x[0]$ comme sentinelle à la suite du texte y, de manière à avoir à tester le moins possible la fin du texte ;
- en cherchant non plus forcément $x[0]$, mais la lettre de x qui a la plus faible fréquence d'apparition dans les textes de la famille de y.

Il est à noter que la première technique présuppose qu'une telle altération de la mémoire est possible et qu'elle s'effectue en temps constant. Pour la seconde, outre la nécessité de posséder la fréquence des lettres de l'alphabet, le choix de la position de la lettre distinguée demande un calcul préalable sur x.

Un procédé d'ordre différent consiste à appliquer un décalage qui ne tienne compte que de la valeur de la lettre la plus à droite dans la fenêtre.



a	a	b	c	d
dern-occ$[a]$	1	4	3	6

(a) (b)

Figure 1.11 Décalage de la fenêtre glissante à l'aide de la table de la dernière occurrence, *dern-occ*, lorsque $x =$ `bbcaac`. **(a)** Les valeurs de la table *dern-occ* sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. **(b)** La fenêtre sur le texte y est à la position droite 8. La lettre à cette position, à savoir $y[8] = \mathtt{b}$, apparait à la position maximale $k = 1$ sur $x[0\,..\,|x|-2]$. Un décalage valide consiste à faire glisser la fenêtre de $|x| - 1 - k = 4 =$ *dern-occ*$[\mathtt{b}]$ positions sur la droite.

Soit j la position droite de la fenêtre. Deux cas antagonistes peuvent être envisagés selon que la lettre $y[j]$ apparait ou non dans $x[0\,..\,m-2]$:

- dans le cas où $y[j]$ n'apparait pas dans $x[0\,..\,m-2]$, le mot x ne peut apparaitre des positions droites $j+1$ à $j+m-1$ sur y ;
- dans l'autre cas, si k est la position maximale de l'occurrence de la lettre $y[j]$ sur $x[0\,..\,m-2]$, le mot x ne peut apparaitre des positions droites $j+1$ à $j+m-1-k-1$ sur y.

D'où des décalages valides à appliquer dans les deux cas : m pour le premier ; et $m-1-k$ pour le second. Remarquons qu'ils ne dépendent que de la lettre $y[j]$ et en aucune manière de sa position j sur y.

Pour formaliser l'observation précédente, on introduit la table :

dern-occ$: A \to \{1, 2, \ldots, m\}$

définie pour toute lettre $a \in A$ par :

$$\textit{dern-occ}[a] = \min(\{m\} \cup \{m-1-k : 0 \leq k \leq m-2 \text{ et } x[k] = a\}) \ .$$

On appelle *dern-occ* la ***table de la dernière occurrence***. Elle exprime un décalage valide, *dern-occ*$[y[j]]$, à appliquer après la tentative à la position droite j sur y. Une illustration est proposée figure 1.11. Suit le code du calcul de *dern-occ*. Il s'exécute en temps $\Theta(m + \operatorname{card} A)$.

DERNIÈRE-OCCURRENCE(x, m)
1 **pour** chaque lettre $a \in A$ **faire**
2 *dern-occ*$[a] \leftarrow m$
3 **pour** $k \leftarrow 0$ **à** $m-2$ **faire**
4 *dern-occ*$[x[k]] \leftarrow m-1-k$
5 **retourner** *dern-occ*

On donne maintenant le code complet de l'algorithme LOCALISER-RAPIDEMENT obtenu de celui de l'algorithme naïf par adjonction de la table *dern-occ*.

```
LOCALISER-RAPIDEMENT(x, m, y, n)
1  dern-occ ← DERNIÈRE-OCCURRENCE(x, m)
2  j ← m − 1
3  tantque j < n faire
4        SIGNALER-SI(y[j − m + 1 .. j] = x)
5        j ← j + dern-occ[y[j]]
```

Si la comparaison des mots à la ligne 4 débute en position $m-1$, la phase de recherche de l'algorithme LOCALISER-RAPIDEMENT s'exécute en temps $\Theta(n/m)$ dans le meilleur des cas. Comme par exemple lorsqu'aucune des lettres aux positions congrues modulo m à $m-1$ sur y n'apparait dans x ; dans ce cas, une seule comparaison entre lettres est effectuée lors de chaque tentative [6] et le décalage est constamment égal à m. Le comportement de l'algorithme sur des textes en langage naturel est très bon. On peut montrer toutefois que dans le cas moyen (sous la double hypothèse de la proposition 1.17 et pour l'ensemble des mots de même longueur), le nombre de comparaisons par lettre de texte est asymptotiquement minoré par $1/\operatorname{card} A$, laquelle borne est indépendante de la longueur du mot cherché.

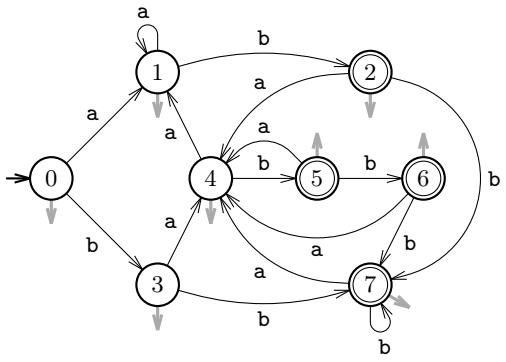
Machine de recherche

Certains automates peuvent servir de ***machine de recherche*** pour le traitement séquentiel de texte. On décrit dans cette partie deux algorithmes à base d'automate pour effectuer la localisation de motifs. On suppose les automates donnés ; le chapitre 2 présente la construction de certains de ces automates.

Considérons un motif $X \subseteq A^*$ et un automate déterministe M qui reconnait le langage A^*X (la figure 1.12(a) fournit un exemple). L'automate M reconnait les mots qui ont comme suffixe un mot de X. Pour localiser les mots de X qui apparaissent dans un texte y, il suffit de faire opérer l'automate M sur le texte y. Lorsque l'état courant est terminal, cela signifie que le préfixe courant de y – la partie de y déjà analysée par l'automate – appartient à A^*X ; ou, autrement dit, que la position courante sur y est la position droite d'une occurrence d'un mot de X. Cette remarque conduit à l'algorithme de localisation avec automate dont le code suit. Une illustration du fonctionnement de l'algorithme est présentée figure 1.12(b).

6. Remarquons qu'il s'agit là du meilleur des cas possibles pour un algorithme de localisation d'un mot de longueur m dans un texte de longueur n ; au moins $\lfloor n/m \rfloor$ lettres du texte doivent être inspectées pour conclure à la non-apparition du mot cherché dans le texte.

(a)



(b)

j	$y[j]$	état r	
		0	
0	c	0	
1	b	3	
2	a	4	
3	b	5	occurrence de ab
4	b	6	occurrences de babb et bb
5	a	4	

Figure 1.12 Localisation des occurrences d'un motif à l'aide d'un automate déterministe (à rapprocher de la figure 1.13). **(a)** Avec l'alphabet $A = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ et le motif $X = \{\texttt{ab}, \texttt{babb}, \texttt{bb}\}$, l'automate déterministe représenté ci-dessus reconnait le langage A^*X. Les flèches grisées issues de chacun des états figurent les flèches ayant pour source ces mêmes états, pour cible l'état initial 0, et pour étiquette une de celles non déjà représentées. Pour localiser les occurrences des mots de X dans un texte y, il suffit de faire opérer l'automate sur y et d'indiquer chaque fois qu'un état terminal est atteint. **(b)** Exemple d'analyse avec $y = \texttt{cbabba}$. De l'utilisation de l'automate, il résulte qu'il y a au moins une occurrence d'un mot de X aux positions 3 et 4 sur y, et aucune aux autres positions.

LA-DÉTERMINISTE(M, y)
1 $r \leftarrow \mathit{initial}[M]$
2 **pour** chaque lettre a de y, séquentiellement **faire**
3 $\quad r \leftarrow$ CIBLE(r, a)
4 $\quad$ SIGNALER-SI$(\mathit{terminal}[r])$

Proposition 1.18
Lorsque M est un automate déterministe qui reconnait le langage A^*X pour un motif $X \subseteq A^*$, l'opération LA-DÉTERMINISTE(M, y) *localise toutes les occurrences des mots de X dans le texte $y \in A^*$.*

Preuve Soit δ la fonction de transition de l'automate M. Comme l'automate est déterministe, il vient immédiatement que l'assertion :

$$r = \delta(\mathit{initial}[M], u) \ , \tag{1.3}$$

où u est le préfixe courant de y, est satisfaite à la suite de l'exécution de chacune des instructions de l'algorithme.

Si une occurrence d'un mot de X se termine à la position courante, le préfixe courant u appartient à A^*X. Et donc, par définition de M et d'après la propriété (1.3), l'état courant r est terminal. Comme l'état initial n'est pas terminal (car $\varepsilon \notin X$), il s'en déduit que l'opération signale cette occurrence.

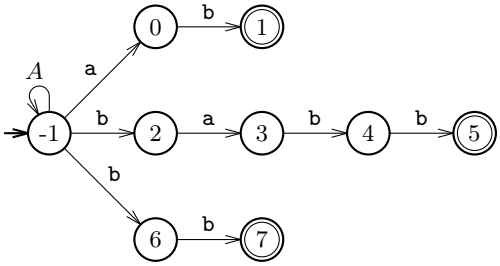
Réciproquement, supposons qu'une occurrence vienne d'être signalée. L'état courant r est donc terminal, ce qui, d'après la propriété (1.3) et par définition de M, implique que le préfixe courant u appartient à A^*X. Une occurrence d'un mot de X se termine donc à la position courante, ce qui achève la preuve du résultat annoncé. ∎

Le temps d'exécution et l'espace supplémentaire nécessaire au fonctionnement de l'algorithme LA-DÉTERMINISTE dépendent uniquement de l'implantation de l'automate M. Par exemple, dans une implantation par matrice de transition, le temps d'analyse du texte est $\Theta(|y|)$, car le délai est constant, et l'espace supplémentaire, en plus de la matrice, est constant (voir proposition 1.15).

Le second algorithme de cette partie s'applique lorsque l'on dispose d'un automate N reconnaissant le langage X lui-même, et non plus A^*X. En ajoutant à l'automate une flèche de son état initial vers lui-même étiquetée par a, pour chaque lettre $a \in A$, on obtient simplement un automate N' qui reconnait le langage A^*X. Mais l'automate N' n'est pas déterministe, ce qui empêche d'appliquer l'algorithme précédent. La figure 1.13(a) présente un exemple d'automate N' pour le même motif X que celui de la figure 1.12(a).

Dans une telle situation, la solution retenue habituellement consiste à simuler l'automate obtenu par déterminisation de N', en suivant en parallèle tous les chemins possibles d'étiquette donnée. Puisque seuls les états qui sont les fins des chemins permettent d'effectuer le test d'occurrence, on se contente de conserver l'ensemble R des états atteints. C'est ce que réalise l'algorithme LA-NON-DÉTERMINISTE ci-dessous. En réalité, il n'est même pas nécessaire de modifier l'automate N car les boucles sur son état initial peuvent également être simulées. Cela est réalisé à la ligne 4 de l'algorithme par l'ajout systématique de l'état initial à l'ensemble d'états. Durant l'exécution de l'automate sur l'entrée y, l'automate n'est pas dans un état donné, mais dans un ensemble d'états, R. Ce sous-ensemble de l'ensemble d'états est recalculé après l'analyse de la lettre courante de y. L'algorithme fait appel à la fonction CIBLES qui effectue une transition sur un ensemble d'états, laquelle fonction est une extension immédiate de CIBLE.

(a)



(b)

j	$y[j]$	ensemble d'états R	
		$\{-1\}$	
0	`c`	$\{-1\}$	
1	`b`	$\{-1, 2, 6\}$	
2	`a`	$\{-1, 0, 3\}$	
3	`b`	$\{-1, 1, 2, 4, 6\}$	occurrence de `ab`
4	`b`	$\{-1, 2, 5, 6, 7\}$	occurrences de `babb` et `bb`
5	`a`	$\{-1, 0, 3\}$	

Figure 1.13 Localisation des occurrences d'un motif à l'aide d'un automate non déterministe (à rapprocher de la figure 1.12). **(a)** L'automate non déterministe représenté reconnait le langage A^*X, avec l'alphabet $A = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ et le motif $X = \{\texttt{ab}, \texttt{babb}, \texttt{bb}\}$. Pour localiser les occurrences des mots de X qui apparaissent dans un texte y, il suffit de faire opérer l'automate sur y et de signaler une occurrence à chaque fois qu'un état terminal est atteint. **(b)** Un exemple de simulation lorsque $y =$ `cbabba`. Le calcul revient à suivre simultanément tous les chemins possibles. Il résulte que le motif apparait aux positions droites 3 et 4 et nulle part ailleurs.

LA-NON-DÉTERMINISTE(N, y)
1 $q_0 \leftarrow$ *initial*$[N]$
2 $R \leftarrow \{q_0\}$
3 **pour** chaque lettre a de y, séquentiellement **faire**
4 $\quad R \leftarrow$ CIBLES$(R, a) \cup \{q_0\}$
5 $\quad t \leftarrow$ FAUX
6 $\quad$ **pour** chaque état $p \in R$ **faire**
7 $\quad\quad$ **si** *terminal*$[p]$ **alors**
8 $\quad\quad\quad t \leftarrow$ VRAI
9 $\quad$ SIGNALER-SI(t)

CIBLES(R, a)
1 $S \leftarrow \emptyset$
2 **pour** chaque état $p \in R$ **faire**
3 $\quad$ **pour** chaque état q tel que $(a, q) \in Succ[p]$ **faire**
4 $\quad\quad S \leftarrow S \cup \{q\}$
5 **retourner** S

Les instructions aux lignes 5–8 de l'algorithme LA-NON-DÉTERMINISTE

donnent à la variable booléenne t la valeur VRAI lorsque l'intersection entre l'ensemble d'états R et l'ensemble des états terminaux est non vide. Une occurrence est ensuite signalée, ligne 9, le cas échéant. La figure 1.13(b) illustre le fonctionnement de l'algorithme.

Proposition 1.19
Lorsque N est un automate qui reconnait le langage X pour un motif $X \subseteq A^*$, l'opération LA-NON-DÉTERMINISTE(N, y) *localise toutes les occurrences des mots de X dans le texte $y \in A^*$.*

Preuve Notons q_0 l'état initial de l'automate N et, pour tout mot $v \in A^*$, R_v l'ensemble d'états défini par :

$R_v = \{q : q \text{ fin d'un chemin d'origine } q_0 \text{ et d'étiquette } v\}$.

On peut vérifier, par récurrence sur la longueur des préfixes de y, que l'assertion :

$$R = \bigcup_{v \preceq_{suff} u} R_v \ , \tag{1.4}$$

où u est le préfixe courant de y, est satisfaite à la suite de l'exécution de chacune des instructions de l'algorithme, hormis celle à la ligne 1.

Si une occurrence d'un mot de X se termine à la position courante, l'un des suffixes v du préfixe courant u appartient à X. Il vient alors, par définition de N, que l'un des états $q \in R_v$ est terminal, puis, d'après la propriété (1.4), que l'un des états de R est terminal. Il s'en déduit que l'opération signale cette occurrence puisqu'aucun des mots de X n'est vide.

Réciproquement, si une occurrence vient d'être signalée, c'est que l'un des états $q \in R$ est terminal. La propriété (1.4) et la définition de N impliquent alors l'existence d'un suffixe v du préfixe courant u qui appartient à X. Il s'ensuit qu'une occurrence d'un mot de X se termine à la position courante. Ce qui achève la preuve de la proposition. ∎

La complexité de l'algorithme LA-NON-DÉTERMINISTE dépend à la fois de l'implantation retenue pour l'automate N et de la réalisation choisie pour manipuler les ensembles d'états. Si, par exemple, l'automate est déterministe, que sa fonction de transition est implantée par matrice de transition, et que les ensembles d'états sont implantés par vecteurs booléens dont les indices sont les états, la fonction CIBLES s'exécute en temps et en espace $O(\operatorname{card} Q)$, où Q est l'ensemble des états. Dans ce cas, l'analyse du texte y a lieu en temps $O(|y| \times \operatorname{card} Q)$ et utilise un espace supplémentaire $O(\operatorname{card} Q)$.

Dans les paragraphes qui suivent, on considère un exemple de réalisation de la simulation ci-dessus adapté au cas d'un très petit automate qui possède une structure arborescente.

Modèle vecteur-binaire

Le ***modèle vecteur-binaire*** fait référence à la possibilité d'utiliser les mots machine pour coder les états des automates de localisation. Lorsque la longueur du langage associé au motif à localiser n'est pas plus grande que la taille d'un mot machine comptée en nombre de bits, cette technique donne des algorithmes efficaces faciles à implanter. La technique est notamment employée section 8.4.

Ici, le principe reprend la méthode de simulation d'automate déterministe des paragraphes précédents en codant l'ensemble des états atteints par un vecteur binaire, et en réalisant le changement d'état par simple décalage contrôlé par un masque associé à la lettre considérée.

Commençons par préciser les notations utilisées dans la suite pour les vecteurs binaires. On assimile un vecteur binaire à un mot sur l'alphabet $\{\mathtt{0},\mathtt{1}\}$. On note respectivement $\vee$ et $\wedge$ le « ou » et le « et » bit à bit. Ce sont des opérations binaires internes aux ensembles de vecteurs binaires de longueurs identiques qui, pour la première, met à `1` les bits du résultat si l'un des deux bits de même position des deux opérandes est égal à `1`, et à `0` sinon, et, pour la seconde, met à `1` les bits du résultat si les deux bits de même position des deux opérandes sont égaux à `1`, et à `0` sinon. On note encore $\dashv$ l'opération externe qui à un naturel k et un vecteur binaire fait correspondre le vecteur binaire de même longueur obtenu du premier en décalant les bits vers la droite de k positions et en complétant à gauche par des `0`. Ainsi, $\mathtt{1001} \vee \mathtt{0011} = \mathtt{1011}$, $\mathtt{1001} \wedge \mathtt{0011} = \mathtt{0001}$, et $2 \dashv \mathtt{1101} = \mathtt{0011}$.

Considérons un ensemble X fini non vide de mots tous non vides. Soit N l'automate obtenu à partir des $\operatorname{card} X$ automates déterministes élémentaires qui reconnaissent les mots de X en fusionnant les états initiaux en un seul, disons q_0. Soit N' l'automate construit sur N en ajoutant les flèches de la forme (q_0, a, q_0), pour chaque lettre $a \in A$. L'automate N' reconnait le langage A^*X. La recherche des occurrences de mots de X dans un texte y est réalisée ici comme dans les paragraphes ci-dessus en simulant l'automate déterminisé de N' au moyen de N (voir figure 1.13(a)).

Posons $m = |X|$ et numérotons les états de N depuis -1 jusqu'à $m-1$ en utilisant un parcours en profondeur à partir de l'état initial q_0 – c'est le cas dans l'exemple de la figure 1.13(a). Codons maintenant chaque ensemble d'états $R \setminus \{-1\}$ par un vecteur r de m bits avec la convention suivante :

$p \in R \setminus \{-1\}$ si et seulement si $r[p] = \mathtt{1}$.

Soient r le vecteur de m bits qui code l'état courant de la recherche, $a \in A$ la lettre courante de y, et s le vecteur de m bits qui code l'état suivant. Il est clair que le calcul de s à partir de r et de a observe la règle suivante : $s[p] = \mathtt{1}$ si et seulement si il existe une flèche d'étiquette a, soit de l'état -1 vers l'état p, soit de l'état $p-1$ à l'état p avec $r[p-1] = \mathtt{1}$.

Considérons *init* le vecteur de m bits défini par $init[p] = \mathtt{1}$ si et seulement si il existe une flèche de source l'état -1 et de cible l'état p. Considérons aussi la table *masq* indicée sur A et à valeurs dans l'ensemble des vecteurs de m bits, définie pour toute lettre $b \in A$ par $masq[b][p] = \mathtt{1}$ si et seulement si il existe une flèche d'étiquette b et de cible l'état p. Alors r, a et s satisfont l'identité :

$$s = (init \vee (1 \dashv r)) \wedge masq[a] \ .$$

Celle-ci traduit en termes d'opérations bit à bit la transition effectuée ligne 4 de l'algorithme LA-NON-DÉTERMINISTE, hors l'état initial. Le vecteur binaire *init* code les transitions potentielles à partir de l'état initial, et le décalage d'un bit vers la droite celles à partir des états atteints. La table *masq* valide les transitions étiquetées par la lettre courante.

Il ne reste plus maintenant qu'à indiquer comment tester si l'un des états représentés par un vecteur r de m bits qui code l'état courant de la recherche est terminal ou non. Dans ce but, soit *term* le vecteur de m bits défini par $term[p] = \mathtt{1}$ si et seulement si l'état p est terminal. Alors l'un des états représenté par r est terminal si et seulement si :

$$r \wedge term \neq \mathtt{0}^m \ .$$

Suivent le code de la fonction PETIT-AUTOMATE qui calcule les vecteurs *init* et *term* ainsi que les valeurs de la table *masq*, puis celui de l'algorithme de localisation.

```
PETIT-AUTOMATE(X, m)
 1  init ← 0^m
 2  term ← 0^m
 3  pour chaque lettre a ∈ A faire
 4        masq[a] ← 0^m
 5  p ← −1
 6  pour chaque mot x ∈ X faire
 7        init[p + 1] ← 1
 8        pour chaque lettre a de x, séquentiellement faire
 9              p ← p + 1
10              masq[a][p] ← 1
11        term[p] ← 1
12  retourner (init, term, masq)
```

```
LOCALISER-MOTS-COURTS(X, m, y)
1  (init, term, masq) ← PETIT-AUTOMATE(X, m)
2  r ← 0^m
3  pour chaque lettre a de y, séquentiellement faire
4        r ← (init ∨ (1 ⊣ r)) ∧ masq[a]
5        SIGNALER-SI(r ∧ term ≠ 0^m)
```

(a)

k	0	1	2	3	4	5	6	7
$init[k]$	1	0	1	0	0	0	1	0
$term[k]$	0	1	0	0	0	1	0	1
$masq$[a]$[k]$	1	0	0	1	0	0	0	0
$masq$[b]$[k]$	0	1	1	0	1	1	1	1
$masq$[c]$[k]$	0	0	0	0	0	0	0	0

(b)

j	$y[j]$	vecteur binaire r	
		00000000	
0	c	00000000	
1	b	00100010	
2	a	10010000	
3	b	01101010	occurrence de ab
4	b	00100111	occurrences de babb et bb
5	a	10010000	

Figure 1.14 Utilisation de vecteurs binaires pour la recherche des occurrences du motif $X = \{\texttt{ab}, \texttt{babb}, \texttt{bb}\}$ (voir figure 1.13). **(a)** Les vecteurs *init* et *term*, et les valeurs de la table de vecteurs *masq* sur l'alphabet $A = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. Ces vecteurs sont de longueur 8 car $|X| = 8$. Le premier vecteur code les transitions potentielles à partir de l'état initial. Le deuxième code les états terminaux. Les vecteurs de la table *masq* codent les occurrences d'une lettre de l'alphabet dans les mots de X. **(b)** Les valeurs successives du vecteur r qui code l'état courant de la recherche des occurrences des mots de X dans le texte $y =$ cbabba. La zone grisée qui marque certains bits indique qu'un état terminal vient d'être atteint.

Un exemple de calcul est traité figure 1.14.

Proposition 1.20
Pour s'exécuter, l'opération LOCALISER-MOTS-COURTS(X, m, y) *nécessite un temps* $\Theta(m \times \operatorname{card} A + m \times |y|)$. *L'espace mémoire supplémentaire requis est* $\Theta(m \times \operatorname{card} A)$.

Preuve Le temps nécessaire à l'initialisation des vecteurs binaires *init*, *term* et $masq[a]$ pour $a \in A$, est linéaire en leur taille, soit $\Theta(m \times \operatorname{card} A)$. Les instructions aux lignes 4 et 5 s'exécutent chacune en temps $\Theta(m)$. Les complexités annoncées s'en déduisent. ∎

Cela établi, lorsque la longueur m est inférieure au nombre de bits d'un mot machine, tout vecteur binaire de m bits peut s'implanter à l'aide d'un mot machine dont seuls les m premiers bits sont significatifs. Ce qui donne le résultat suivant.

Corollaire 1.21
Lorsque m est inférieur à la longueur d'un mot machine, l'opération LOCALISER-MOTS-COURTS(X, m, y) *s'exécute en temps* $\Theta(|y| + \operatorname{card} A)$ *avec un espace mémoire supplémentaire* $\Theta(\operatorname{card} A)$. ∎

1.6 Base de techniques élaborées

On présente dans cette section deux méthodes fondamentales des algorithmes efficaces de localisation de motifs ou de recherche de régularités dans les mots. Il s'agit de deux tables, la table des bords et la table des préfixes, qui toutes deux mémorisent les occurrences des préfixes d'un mot qui apparaissent au sein de lui-même. Les tables peuvent être calculées en temps linéaire. Les algorithmes de calcul fournissent aussi des méthodes de localisation de mots qui sont étudiées en détail dans les chapitres 2 et 3 (un prélude est proposé exercice 1.24).

Table des bords

Soit x un mot de longueur $m \geq 1$. On définit la table :

$bord\colon \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$

par :

$bord[k] = |Bord(x[0\,..\,k])|$

pour $k = 0, 1, \ldots, m-1$. On dit de la table *bord* qu'elle est la ***table des bords*** pour le mot x, sous-entendant qu'il s'agit des bords des préfixes non vides du mot. Voici en exemple la table des bords pour le mot x lorsque $x =$ `abbabaabbabaaaabbabbaa` :

k	0	1	2	3	4	5	6	7	8	9	10	11
$x[k]$	a	b	b	a	b	a	a	b	b	a	b	a
$bord[k]$	0	0	0	1	2	1	1	2	3	4	5	6

k	12	13	14	15	16	17	18	19	20	21
$x[k]$	a	a	a	b	b	a	b	b	a	a
$bord[k]$	7	1	1	2	3	4	5	3	4	1

Le lemme suivant fournit la relation de récurrence utilisée par la fonction Bords, donnée ensuite, pour effectuer le calcul de la table *bord*.

Lemme 1.22
Pour tout $(u, a) \in A^+ \times A$, on a :

$$Bord(ua) = \begin{cases} Bord(u)a & \text{si } Bord(u)a \preceq_{préf} u \ , \\ Bord(Bord(u)a) & \text{sinon} \ . \end{cases}$$

Preuve On remarque pour commencer que si $Bord(ua)$ est un mot non vide, il est de la forme wa où w est un bord de u.

Si $Bord(u)a \preceq_{préf} u$, le mot $Bord(u)a$ est alors un bord de ua, et la remarque précédente montre qu'il s'agit du plus long mot de la sorte. Il s'ensuit que $Bord(ua) = Bord(u)a$ dans ce cas.

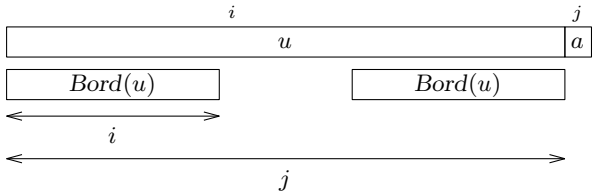


Figure 1.15 Schéma montrant la correspondance entre les variables i et j considérées en ligne 3 de la fonction BORDS et l'énoncé du lemme 1.22.

Dans le cas contraire, $Bord(ua)$ est à la fois un préfixe de $Bord(u)$ et un suffixe de $Bord(u)a$. Comme il est de longueur maximale avec cette propriété, c'est bien le mot $Bord(Bord(u)a)$. ∎

La figure 1.15 schématise la correspondance entre les variables i et j de la fonction dont le code suit et l'énoncé du lemme.

```
BORDS(x, m)
 1  i ← 0
 2  pour j ← 1 à m − 1 faire
 3        bord[j − 1] ← i
 4        tantque i ≥ 0 et x[j] ≠ x[i] faire
 5              si i = 0 alors
 6                    i ← −1
 7              sinon i ← bord[i − 1]
 8        i ← i + 1
 9  bord[m − 1] ← i
10  retourner bord
```

Proposition 1.23
La fonction BORDS *appliquée à un mot x et sa longueur m produit la table des bords pour x.*

Preuve La table *bord* est initialisée séquentiellement, du préfixe de x de longueur 1 à x lui-même. Lors de l'exécution de la boucle **tantque** aux lignes 4–7 est inspectée la suite des bords de $x[0\,..\,j-1]$, d'après la proposition 1.5. À la sortie de cette boucle, on a $|Bord(x[0\,..\,j])| = |x[0\,..\,i]| = i + 1$, conformément au lemme 1.22. La correction du code s'ensuit. ∎

Proposition 1.24
L'opération BORDS(x, m) *s'exécute en temps $\Theta(m)$. Le nombre de comparaisons entre les lettres du mot x est compris entre $m-1$ et $2m-3$ lorsque $m \geq 2$.*

On convient dans toute la suite de qualifier la comparaison de deux lettres données de ***positive*** lorsque ces deux lettres sont identiques, et de ***négative*** dans le cas contraire.

Preuve Remarquons que le temps d'exécution est linéaire en le nombre de comparaisons effectuées entre les lettres de x. Il suffit donc d'établir la borne sur le nombre de comparaisons.

La quantité $2j - i$ croît d'au moins une unité après chaque comparaison de lettres : les variables i et j sont incrémentées à la suite d'une comparaison positive ; la valeur de i est diminuée d'au moins une unité et celle de j reste inchangée à la suite d'une comparaison négative. Lorsque $m \geq 2$, cette quantité vaut 2 à la première comparaison ($i = 0$ et $j = 1$) et au plus $2m - 2$ lors de la dernière ($i \geq 0$ et $j = m - 1$). Le nombre total de comparaisons est donc bien majoré par $2m - 3$ comme annoncé.

La borne de $2m - 3$ comparaisons est précise : elle est atteinte pour tout mot x de la forme $a^{m-1}b$ avec $a, b \in A$ et $a \neq b$. Ce qui achève la preuve. ∎

Une autre preuve de la borne $2m - 3$ est proposée exercice 1.22.

Table des préfixes

Soit x un mot de longueur $m \geq 1$. On définit la table :

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

par :

$préf[k] = |lpc(x, x[k \,..\, m-1])|$

pour $k = 0, 1, \ldots, m-1$, où :

$lpc(u, v) =$ le ***plus long préfixe commun*** aux mots u et v .

La table *préf* est appelée la ***table des préfixes*** pour le mot x. Elle mémorise les préfixes de x qui apparaissent au sein du mot lui-même. On note que $préf[0] = |x|$. L'exemple qui suit montre la table des préfixes pour x dans le cas où $x =$ `abbabaabbabaaaabbabbaa`.

k	0	1	2	3	4	5	6	7	8	9	10	11
$x[k]$	a	b	b	a	b	a	a	b	b	a	b	a
$préf[k]$	22	0	0	2	0	1	7	0	0	2	0	1

k	12	13	14	15	16	17	18	19	20	21
$x[k]$	a	a	a	b	b	a	b	b	a	a
$préf[k]$	1	1	5	0	0	4	0	0	1	1

Certains algorithmes de recherche de mots (voir chapitre 3) utilisent la table *suff* qui n'est autre que l'analogue de la table des préfixes obtenue en considérant le renversé du mot x.

La méthode de calcul de *préf* présentée ci-dessous procède en déterminant $préf[i]$ par valeurs croissantes de la position i sur x. Une méthode naïve consisterait à évaluer chaque valeur $préf[i]$ indépendamment des valeurs précédentes par comparaisons directes ; mais on aboutirait alors à un calcul quadratique, dans le cas où x est la puissance d'une seule lettre, par exemple. L'utilisation de valeurs déjà calculées permet d'obtenir un algorithme linéaire. Pour cela on introduit, l'indice i étant fixé, deux valeurs g et f qui constituent les éléments clés de la méthode. Elles satisfont les relations :

$$g = \max\{j + préf[j] : 0 < j < i\} \tag{1.5}$$

et :

$$f \in \{j : 0 < j < i \text{ et } j + préf[j] = g\} \ . \tag{1.6}$$

On remarque que g et f sont définies quand $i > 1$. Le mot $x[f\,..\,g-1]$ est ainsi un préfixe de x, soit aussi un bord de $x[0\,..\,g-1]$. C'est le mot vide lorsque $f = g$. On peut noter de plus que si $g < i$ on a alors $g = i - 1$, et que dans le cas contraire, par définition de f, on a $f < i \leq g$.

Le lemme qui suit fournit la justification du principe de fonctionnement de la fonction Préfixes.

Lemme 1.25
Si $i < g$, on a :

$$préf[i] = \begin{cases} préf[i-f] & \text{si } préf[i-f] < g-i \ , \\ g-i & \text{si } préf[i-f] > g-i \ , \\ g-i+\ell & \text{sinon} \ , \end{cases}$$

où $\ell = |lpc(x[g-i\,..\,m-1], x[g\,..\,m-1])|$.

Preuve Posons $u = x[f\,..\,g-1]$. Le mot u est un préfixe de x par définition de f et g. Posons aussi $k = préf[i-f]$ et $k' = g - i$. Par définition de *préf*, le mot $x[i-f\,..\,i-f+k-1]$ est un préfixe de x mais $x[i-f\,..\,i-f+k]$ n'en est pas un.

Si $k < k'$, une occurrence de $x[i-f\,..\,i-f+k]$ débute à la position $i-f$ sur u, et donc également à la position i sur x. Ce qui montre que $x[i-f\,..\,i-f+k-1]$ est le plus long préfixe de x débutant à la position i. On a ainsi $préf[i] = k = préf[i-f]$.

Si $k > k'$, le mot $x[i-f\,..\,i-f+k']$ est un préfixe de x et le mot $x[i-f\,..\,i-f+k'-1]$ est un suffixe de u et donc il apparait dans x à la position i. Comme $x[i-f+k'] \neq x[g]$, par définition de f et g, le mot $x[i-f\,..\,i-f+k'-1]$ est le plus long préfixe de x qui apparait à la position i, ce qui implique $préf[i] = k' = g - i$.

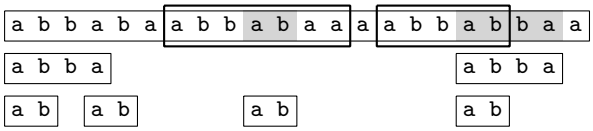


Figure 1.16 Illustration du principe de fonctionnement de la fonction PRÉFIXES. Les facteurs encadrés $x[6\,..\,12]$ et $x[14\,..\,18]$ et les facteurs grisés $x[9\,..\,10]$ et $x[17\,..\,20]$ sont des préfixes du mot $x = \texttt{abbabaabbabaaaabbabbaa}$. Pour $i = 9$, on a $f = 6$ et $g = 13$. La situation à cette position est la même qu'à la position $3 = 9 - 6$. On a $préf[9] = préf[3] = 2$ qui signifie que `ab`, de longueur 2, est le plus long facteur de position 9 qui est un préfixe de x. Pour $i = 17$, on a $f = 14$ et $g = 19$. Comme $préf[17-14] = 2 \geq 19-17$, on en déduit que $x[i\,..\,g-1] = \texttt{ab}$ est un préfixe de x. Il faut comparer x et $x[i\,..\,m-1]$ à partir des positions respectives 2 et g pour déterminer $préf[i] = 4$.

Sinon $k = k'$. Le mot $x[i\,..\,g-1]$, qui est un suffixe de u, est un préfixe de $x[i-f\,..\,i-f+k-1]$ donc de x. On en déduit immédiatement que $préf[i] = g - i + \ell$. ∎

Dans le calcul de *préf*, on initialise la variable g à 0 pour simplifier l'écriture du code de la fonction PRÉFIXES, et on laisse f initialement non défini. La première étape du calcul consiste ainsi à déterminer $préf[1]$ par comparaisons de lettres. L'utilité du résultat énoncé ci-dessus n'apparait au mieux qu'à partir du calcul de la valeur suivante.

```
PRÉFIXES(x, m)
 1  préf[0] ← m
 2  g ← 0
 3  pour i ← 1 à m − 1 faire
 4        si i < g et préf[i − f] < g − i alors
 5              préf[i] ← préf[i − f]
 6        sinonsi i < g et préf[i − f] > g − i alors
 7              préf[i] ← g − i
 8        sinon (f, g) ← (i, max{g, i})
 9              tantque g < m et x[g] = x[g − f] faire
10                    g ← g + 1
11              préf[i] ← g − f
12  retourner préf
```

Une illustration du principe de fonctionnement de la fonction est donné figure 1.16. Un schéma montrant la correspondance entre les variables de la fonction et les notations utilisées dans l'énoncé du lemme 1.25 et dans celui de sa preuve est donné figure 1.17.

Proposition 1.26
La fonction PRÉFIXES *appliquée à un mot x et à sa longueur m produit la table des préfixes pour x.*

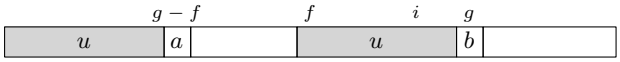


Figure 1.17 Variables i, f et g de la fonction PRÉFIXES. La boucle principale admet pour invariants : $u = lpc(x, x[f\,..\,m-1])$ et donc $a \neq b$ avec $a, b \in A$, ainsi que $f < i$ lorsque f est définie. Le schéma correspond à la situation dans laquelle $i < g$.

Preuve On peut vérifier que les variables f et g satisfont les relations (1.5) et (1.6) à chaque étape de l'exécution de la boucle.

On constate ensuite que, pour i fixé satisfaisant la condition $i < g$, la fonction applique la relation de l'énoncé du lemme 1.25, ce qui produit un calcul correct. Il reste donc à vérifier que le calcul est correct lorsque $i \geq g$. Mais dans cette situation, les instructions aux lignes 8–10 calculent $|lpc(x, x[i\,..\,m-1])| = |x[f\,..\,g-1]| = g-f$ qui est, par définition même, la valeur de $préf[i]$.

La fonction produit donc bien la table $préf$. ∎

Proposition 1.27
L'exécution de l'opération PRÉFIXES(x, m) *prend un temps* $\Theta(m)$. *Moins de* $2m$ *comparaisons entre les lettres du mot* x *sont effectuées.*

Preuve Les comparaisons de lettres sont effectuées à la ligne 9. Toute comparaison entre lettres égales conduit à incrémenter la variable g. Comme la valeur de g ne diminue jamais et qu'elle varie de 0 à m au maximum, il y a au plus m comparaisons négatives. Chaque comparaison négative fait passer à l'étape suivante de la boucle. Il y en a donc au plus m également. Soit au plus $2m$ comparaisons au total.

Le raisonnement précédent montre aussi que le temps total de toutes les exécutions de la boucle aux lignes 9–10 est $\Theta(m)$. Les autres instructions de la boucle 3–11 prennent un temps constant pour chaque valeur de i donnant à nouveau un temps global $\Theta(m)$ pour leurs exécutions et celle de la fonction. ∎

La borne de $2m$ sur le nombre de comparaisons effectuées par la fonction PRÉFIXES est relativement fine. Par exemple, on obtient $2m-3$ comparaisons pour un mot de la forme $a^{m-1}b$ avec $m \geq 2$, $a, b \in A$ et $a \neq b$. En effet, il faut $m-1$ comparaisons pour calculer $préf[1]$, puis une comparaison pour chacune des $m-2$ valeurs $préf[i]$ avec $1 < i < m$.

Relations entre bords et préfixes

Les tables $bord$ et $préf$ dont le calcul est décrit ci-dessus mémorisent toutes deux les occurrences des préfixes de x. On explicite ici une relation entre ces deux tables.

La relation n'est pas immédiate pour la raison qui suit, illustrée par la figure 1.18. Quand $préf[i] = \ell$, le facteur $u = x[i\,..\,i+\ell-1]$ est un préfixe

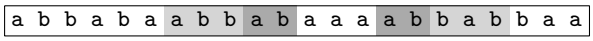


Figure 1.18 Bords et préfixes. Dans le mot $x = \texttt{abbabaabbabaaaabbabbaa}$, $préf[9] = 2$ et $bord[9 + 2 - 1] = 5 \neq 2$. On a également $bord[15] = 2$ et $préf[15 - 2 + 1] = 5 \neq 2$.

de x mais ce n'est pas nécessairement le bord de $x[0\,..\,i + \ell - 1]$, celui-ci pouvant être plus long que u. De même, lorsque $bord[j] = \ell$, le facteur $v = x[j - \ell + 1\,..\,j]$ est un préfixe de x mais ce n'est pas nécessairement le plus long préfixe de x apparaissant à la position $j - \ell + 1$.

La proposition qui suit montre comment la table *bord* s'exprime en fonction de la table *préf*. On peut en déduire un algorithme de calcul de la table *bord* à partir de la table *préf*.

Proposition 1.28
Soient $x \in A^+$ et j une position sur x. Alors :

$$bord[j] = \begin{cases} 0 & \text{si } I = \emptyset \ , \\ j - \min I + 1 & \text{sinon} \ , \end{cases}$$

où $I = \{i : 0 \leq i \leq j \text{ et } i + préf[i] - 1 \geq j\}$.

Preuve On remarque pour commencer que, pour $0 \leq i \leq j$, $i \in I$ si et seulement si $x[i\,..\,j] \preceq_{préf} x$. En effet, si $i \in I$, on a $x[i\,..\,j] \preceq_{préf} x[i\,..\,i + préf[i] - 1] \preceq_{préf} x$, donc $x[i\,..\,j] \preceq_{préf} x$. Réciproquement, si $x[i\,..\,j] \preceq_{préf} x$, on en déduit, par définition de $préf[i]$, $préf[i] \geq j - i + 1$. Et donc $i + préf[i] - 1 \geq j$. Ce qui montre que $i \in I$.

On remarque à la suite que $bord[j] = 0$ si et seulement si $I = \emptyset$.

Il s'ensuit que si $bord[j] \neq 0$ (soit $bord[j] > 0$) et $k = j - bord[j] + 1$, on a $k \leq j$ et $x[k\,..\,j] \preceq_{préf} x$. Aucun facteur $x[i\,..\,j]$, $i < k$, ne satisfait la relation $x[i\,..\,j] \preceq_{préf} x$ par définition de $bord[j]$. Donc $k = \min I$ par la première remarque, et $bord[j] = j - k + 1$ comme annoncé. ∎

Le calcul de la table *préf* à partir de la table *bord* peut conduire à une itération, et ne semble pas donner une expression simple, comparable à celle de l'énoncé précédent (voir exercice 1.23).

Notes

Le chapitre contient les éléments de base pour une étude précise des algorithmes sur les mots. La plupart des notions qui y sont introduites est dispersée dans différents ouvrages dont nous citons ici ceux qui sont souvent considérés comme des références dans leurs domaines.

Les aspects combinatoires sur les mots sont traités dans l'ouvrage collectif de Lothaire [96]. On peut se reporter au livre d'Aho, Hopcroft

et Ullman [87] pour les questions d'algorithmique : expression des algorithmes, structures de données et évaluation de la complexité. Nous nous sommes inspirés du livre de Cormen, Leiserson et Rivest [92] pour la présentation générale et le style des algorithmes. Concernant les automates et langages, on peut consulter le livre de Berstel [91] ou celui de Pin [100]. Les ouvrages de Berstel et Perrin (1985) ainsi que celui de Béal [89] contiennent des éléments de la théorie des codes (exercices 1.10 et 1.11 entre autres). Enfin, l'ouvrage d'Aho, Sethi et Ullman [88] décrit des méthodes d'implantation des automates.

La section 1.5 sur les techniques de base contient des éléments fréquemment retenus pour le développement final de logiciels utilisant des algorithmes de traitement de chaines de caractères. Il s'agit en particulier des heuristiques et de l'utilisation de mots machine. Cette dernière technique est reprise dans le chapitre 8 pour la localisation de motifs approchés. Ce type de technique a été initiée par Baeza-Yates et Gonnet (1992) et par Wu et Manber (1992). L'algorithme LOCALISER-RAPIDEMENT est dû à Horspool (1980). La recherche d'un mot au moyen d'une fonction de hachage est analysée par Karp et Rabin (1987).

Le traitement des notions de la section 1.6 est original. Le calcul de la table des bords est classique. Il s'inspire d'un algorithme de Morris et Pratt de 1970 (voir [12]) qui est à l'origine du premier algorithme de localisation d'un mot fonctionnant en temps linéaire. La table des préfixes synthétise différemment les mêmes informations sur un mot que la table précédente. La notion duale de table des suffixes est utilisée dans le chapitre 3. Gusfield [7] en fait un élément fondamental des méthodes de localisation de mots (son algorithme Z correspond à l'algorithme SUFFIXES du chapitre 3).

Du point de vue terminologique, un mot, encore appelé « chaine de caractères », a comme équivalent anglais *word*, *string* ou *sequence*. Le terme de facteur, qui traduit la structure algébrique de A^*, admet pour synonyme « segment » et se traduit souvent (en anglais) par *subword* ou *substring*, mais aussi par *factor*. Enfin, un sous-mot, qui est une sous-suite, se traduit (en anglais) par *subsequence*.

Exercices

1.1 (Calcul)
Quel est le nombre de préfixes, de suffixes, de facteurs et de sous-mots d'un mot donné ? Discuter si nécessaire.

1.2 (Morphisme de Fibonacci)
Un ***morphisme*** f sur A^* est une application de A^* dans A^* qui satisfait les règles :

$$f(\varepsilon) \;=\; \varepsilon \;,$$

$$f(x \cdot y) \quad = \quad f(x) \cdot f(y) \qquad \text{pour } x, y \in A^* \ .$$

Pour tout naturel n et tout mot $x \in A^*$, on note $f^n(x)$ le mot défini par $f^0(x) = \varepsilon$ et $f^k(x) = f^{k-1}(f(x))$ pour $k = 1, 2, \ldots, n$.

Considérons l'alphabet $A = \{\mathtt{a}, \mathtt{b}\}$. Soit φ le morphisme sur A^* défini par $\varphi(\mathtt{a}) = \mathtt{ab}$ et $\varphi(\mathtt{b}) = \mathtt{a}$. Montrer que le mot $\varphi^n(\mathtt{a})$ est identique à F_{n+2}, le mot de Fibonacci d'indice $n + 2$.

1.3 (Permutation)
On appelle permutation sur l'alphabet A un mot u qui satisfait la condition $\operatorname{card} alph(u) = |u| = \operatorname{card} A$. C'est donc un mot dans lequel toutes les lettres de l'alphabet apparaissent une fois et une seule.

Pour $k = \operatorname{card} A$, montrer qu'il existe un mot de longueur inférieure à $k^2 - 2k + 4$ qui possède comme sous-mots toutes les permutations sur A. Écrire un algorithme de construction d'un tel mot. [*Aide :* voir Mohanty (1980).]

1.4 (Période)
Montrer que la condition 3 de la proposition 1.4 peut être remplacée par la condition suivante : il existe un mot t et un entier $k > 0$ tels que $x \preceq_{fact} t^k$ et $|t| = p$.

1.5 (Cas limite)
Montrer que le mot $(\mathtt{ab})^k\mathtt{a}(\mathtt{ab})^k\mathtt{a}$ avec $k \geq 1$ est cas limite pour le lemme de périodicité.

1.6 (Trois périodes)
Sur les triplets d'entiers positifs (p_1, p_2, p_3) classés, $p_1 \leq p_2 \leq p_3$, on définit l'opération de dérivation par : le dérivé de (p_1, p_2, p_3) est le triplet classé constitué des entiers p_1, $p_2 - p_1$ et $p_3 - p_1$. Soit (q_1, q_2, q_3) le premier triplet obtenu en itérant la dérivation à partir de (p_1, p_2, p_3) tel que $q_1 = 0$.

Montrer que si le mot $x \in A^*$ possède p_1, p_2 et p_3 comme périodes et que :

$$|x| \geq \frac{1}{2}(p_1 + p_2 + p_3 - 2\,\mathrm{pgcd}(p_1, p_2, p_3) + q_2 + q_3) \ ,$$

alors il possède aussi $\mathrm{pgcd}(p_1, p_2, p_3)$ comme période. [*Aide :* voir Mignosi et Restivo [97].]

1.7 (Trois carrés)
Soient trois mots non vides u, v et w. Montrer que si l'on suppose que u est primitif et que $u^2 \prec_{préf} v^2 \prec_{préf} w^2$, on a $2|u| < |w|$ (voir proposition 9.17 pour une conséquence plus précise).

1.8 (Conjugués)
Montrer que deux mots conjugués non vides ont même exposant et des racines conjuguées.

Montrer que la classe de conjugaison de tout mot non vide x contient $|x|/k$ éléments où k est l'exposant de x.

1.9 (Périodes)
Soit p une période de x qui n'est pas multiple de $pér(x)$. Montrer que $p > |x| - pér(x)$.

Soient p et q deux périodes de x telles que $p < q$. Montrer que :

– $q - p$ est une période de $prem_{|x|-p}(x)$ et de $(prem_p(x))^{-1}x$;
– p et $q + p$ sont des périodes de $prem_q(x)x$.

(La définition de $prem_k$ est donnée en section 4.4.)

Montrer que si $x = uvw$, uv et vw ont comme période p et $|v| \geq p$, alors x a période p.

Supposons que x ait une période p et possède un facteur v de période r avec r diviseur de q. Montrer que r est aussi une période de x.

1.10 (Code)
Un langage $X \subseteq A^*$ est un ***code*** si tout mot de X^+ admet une décomposition unique en mots de X.

Montrer que le codage ASCII des caractères sur l'alphabet $\{\mathtt{0}, \mathtt{1}\}$ fournit un code au sens de cette définition.

Montrer que les langages $\{\mathtt{a}, \mathtt{b}\}^*$, $\mathtt{ab}^*$, $\{\mathtt{aa}, \mathtt{ba}, \mathtt{b}\}$, $\{\mathtt{aa}, \mathtt{baa}, \mathtt{ba}\}$ et $\{\mathtt{a}, \mathtt{ba}, \mathtt{bb}\}$ sont des codes. Montrer que ce n'est pas le cas des langages $\{\mathtt{a}, \mathtt{ab}, \mathtt{ba}\}$ et $\{\mathtt{a}, \mathtt{abbba}, \mathtt{babab}, \mathtt{bb}\}$.

Un langage $X \subseteq A^*$ est préfixe si la condition :

$u \preceq_{préf} v$ implique $u = v$

est satisfaite pour tous mots $u, v \in X$. La notion de langage suffixe est définie de façon duale.

Montrer que tout langage préfixe est un code. Même chose pour les langages suffixes.

1.11 (Théorème du défaut)
Soit $X \subseteq A^*$ un ensemble fini qui n'est pas un code. Soit $Y \subseteq A^*$ un code pour lequel Y^* est le plus petit ensemble de cette forme qui contient X^*. Montrer que $\operatorname{card} Y < \operatorname{card} X$. [*Aide :* tout mot $x \in X$ s'écrit sous la forme $y_1 y_2 \dots y_k$ avec $y_i \in Y$ pour $i = 1, 2, \dots, k$; montrer que la fonction $\alpha : X \to Y$ définie par $\alpha(x) = y_1$ est surjective mais n'est pas injective ; voir [96].]

1.12 (Commutation)
Montrer par le théorème du défaut (voir exercice 1.11), puis par le lemme de périodicité que, si $uv = vu$, pour deux mots $u, v \in A^*$, u et v sont des puissances d'un même mot.

1.13 (nlogn)
Soit $f\colon \mathbf{N} \to \mathbf{N}$ une fonction définie par :

$$\begin{aligned} f(1) &= a \ , \\ f(n) &= f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + bn \qquad \text{pour } n \geq 2 \ , \end{aligned}$$

avec $a \in \mathbf{N}$ et $b \in \mathbf{N} \setminus \{0\}$. Montrer qu'alors $f(n)$ est $\Theta(n \log n)$.

1.14 (Filtre)
On considère un système dans lequel les caractères sont codés sur 8 bits. On veut développer un algorithme de localisation par automate pour des mots écrits sur l'alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

Décrire des structures de données pour réaliser l'automate à l'aide d'une matrice de transitions de taille $4 \times m$ (et non pas $256 \times m$), où m est le nombre d'états de l'automate, utilisant éventuellement un espace supplémentaire dont la taille est indépendante de m.

1.15 (Implantation de fonctions partielles)
Soit $f\colon E \to F$ un fonction partielle où E est un ensemble fini. Décrire une implantation de f permettant de réaliser chacune des quatre opérations suivantes en temps constant :

- initialiser f, de sorte que $f(x)$ est indéfini pour chaque $x \in E$;
- fixer la valeur de $f(x)$ à $y \in F$, pour chaque $x \in E$;
- tester si $f(x)$ est défini ou non, pour chaque $x \in E$;
- donner la valeur de $f(x)$, pour chaque $x \in E$.

On peut utiliser un espace $O(\operatorname{card} E)$. [*Aide :* utiliser simultanément une table indexée par E et une liste des éléments x pour lesquels $f(x)$ est défini, avec des renvois entre table et liste.]

En déduire que la mise en table d'une telle fonction peut se faire en temps linéaire en le nombre d'éléments de E dont l'image par f est définie.

1.16 (Pas si naïf)
On considère ici une implantation un peu plus élaborée du mécanisme de la fenêtre glissante que celle décrite pour l'algorithme naïf. Parmi les mots x de longueur $m \geq 2$, elle distingue deux classes : celle pour laquelle les deux premières lettres sont identiques (soit $x[0] = x[1]$), et la classe antagoniste (soit $x[0] \neq x[1]$). Cette distinction élémentaire permet de décaler la fenêtre de deux positions vers la droite dans les cas suivants : le mot cherché x appartient à la première classe et $y[j+1] \neq x[1]$; le mot cherché x appartient à la seconde classe et $y[j+1] = x[1]$. D'autre part, si la comparaison du mot x et du contenu de la fenêtre s'effectue toujours lettre à lettre, celle-ci suit ici l'ordre $1, 2, \ldots, m-1$ puis 0 des positions sur x.

Donner le code d'un algorithme qui applique cette méthode.

Montrer que le nombre de comparaisons par lettre de texte est en moyenne (légèrement) strictement inférieur à 1 lorsque la moyenne est évaluée sur l'ensemble des mots de même longueur, que cette longueur est supérieure à 2 et que l'alphabet contient au moins quatre lettres. [*Aide :* voir Hancart (1993).]

1.17 (Bout de fenêtre)
Considérer la méthode qui, à l'instar de celle de l'algorithme LOCALISER-RAPIDEMENT utilisant la lettre la plus à droite dans la fenêtre pour effectuer un décalage, utilise les deux lettres les plus à droite dans la fenêtre (à supposer que le mot cherché soit de longueur supérieure à 2).

Donner le code d'un algorithme qui applique à cette méthode.

Dans quels cas semble-t-il intéressant ? [*Aide :* voir Zhu et Takaoka (1987) ou Baeza-Yates (1989).]

1.18 (Après la fenêtre)
Même énoncé que celui de l'exercice 1.17, mais en utilisant la lettre située immédiatement à droite de la fenêtre (attention au débordement à l'extrémité droite du texte). [*Aide :* voir Sunday (1990).]

1.19 (Sentinelle)
On revient encore sur le problème de la recherche des occurrences d'un mot x de longueur m dans un texte y de longueur n pour lequel on cherche à donner des implantations efficaces pour des solutions élémentaires.

La technique de la sentinelle peut être utilisée pour rechercher la lettre $x[m-1]$ en effectuant les décalages à l'aide de la table *dern-occ*. Les décalages pouvant être de longueur m, on fixe $y[n\,..\,n+m-1]$ à $x[m-1]^m$. Donner un code correct pour cette méthode à sentinelle.

De manière à accélérer le processus et diminuer les tests sur les lettres, il est possible d'enchainer plusieurs décalages sans tester les lettres du texte. Pour cela, on sauvegarde la valeur de $\mathit{dern\text{-}occ}[x[m-1]]$ dans une variable, disons d, puis on fixe la valeur de $\mathit{dern\text{-}occ}[x[m-1]]$ à 0. On peut ensuite enchainer des décalages jusqu'à ce que l'un d'entre eux soit de longueur 0. On teste alors les autres lettres de la fenêtre, en signalant une occurrence le cas échéant, et l'on applique un décalage de longueur d. Donner un code correct pour cette méthode. [*Aide :* voir Hume et Sunday (1991).]

1.20 (En C)
Réaliser une implantation en langage C de l'algorithme LOCALISER-MOTS-COURTS. À titre indicatif, les opérateurs $\vee$, $\wedge$ et $\dashv$ se codent `|`, `&` et `<<`. Étendre l'implantation pour qu'elle accepte un paramètre m quelconque (éventuellement strictement supérieur au nombre de bits d'un mot machine).

Comparer le code obtenu à celui de la source de la commande `agrep` d'Unix.

1.21 (Mots courts)
Décrire un algorithme de localisation de mots courts à la façon de l'algorithme LOCALISER-MOTS-COURTS, mais dans lequel les valeurs binaires `0` et `1` sont échangées.

1.22 (Borne)
Montrer que le nombre de comparaisons positives ainsi que le nombre de comparaisons négatives effectuées lors de l'opération $\textsc{Bords}(x, m)$ sont au plus égaux à $m - 1$. Retrouver ensuite la borne $2m - 3$ de la proposition 1.24.

1.23 (Table des préfixes)
Décrire un algorithme linéaire de calcul de la table *préf* connaissant la table *bord* du mot x.

1.24 (Localisation par les bords ou les préfixes)
Montrer que la table des bords pour le mot $x\$y$ peut être directement utilisée afin de localiser toutes les occurrences du mot x dans le mot y, où $\$ \notin alph(xy)$.

Même chose avec la table des préfixes pour le mot xy.

1.25 (Couverture)
Un mot u est une couverture d'un mot x si pour toute position i sur x, il existe une position j sur x pour laquelle $0 \leq j \leq i < j + |u| \leq |x|$ et $u = x[j\,..\,j + |u| - 1]$.

Écrire un algorithme de calcul de la plus courte couverture d'un mot. En énoncer la complexité.

1.26 (Long bord)
Soit u un bord non vide du mot $x \in A^*$.

Soit $v \in A^*$ tel que $|v| < |u|$. Montrer que v est un bord de u si et seulement si il est un bord de x.

Montrer que x possède un autre bord non vide si u satisfait l'inégalité $|x| < 2|u|$. Montrer que x ne possède aucun autre bord satisfaisant la même inégalité si $pér(x) > |x|/4$.

1.27 (Sans bord)
On dit qu'un mot non vide u est sans bord si $Bord(u) = \varepsilon$, ou, de manière équivalente, si $pér(u) = |u|$.

Soit $x \in A^*$. Montrer que $C = \{u : u \preceq_{préf} x \text{ et } u \text{ sans bord}\}$ est un code suffixe (voir exercice 1.10).

Montrer que x se factorise de manière unique en $x_k x_{k-1} \ldots x_1$ selon les mots de C ($x_i \in C$ pour $i = 1, 2, \ldots, k$). Montrer que x_1 est le plus court mot de C et que x_k en est le plus long.

Décrire un algorithme linéaire de calcul de la factorisation.

1.28 (Suffixe maximal)
On note $SM(\leq, u)$ le suffixe maximal de $u \in A^+$ pour l'ordre lexicographique où, dans cette notation, $\leq$ désigne l'ordre sur l'alphabet. Soit $x \in A^+$.

Montrer que $|SM(\leq, x)| < pér(x)$.

On suppose que $SM(\leq, x) = x$ et l'on note $w_1, w_2, \ldots, w_k$ les bords de x en ordre décroissant de longueur (on a $k > 0$ et $w_k = \varepsilon$). Soient $a_1, a_2, \ldots, a_k \in A$ et $z_1, z_2, \ldots, z_k \in A^*$ tels que :

$$x = w_1 a_1 z_1 = w_2 a_2 z_2 = \cdots = w_k a_k z_k \ .$$

Montrer que $a_1 \leq a_2 \leq \cdots \leq a_k$.

Écrire un algorithme linéaire de calcul du suffixe maximal (pour l'ordre lexicographique) d'un mot $x \in A^+$. [*Aide :* utiliser l'algorithme de calcul des bords de la section 1.6 ou voir Booth (1980) ; voir aussi [5].]

1.29 (Factorisation critique)
Soit $x \in A^+$. Pour chaque position i sur x, on note :

$$rép(i) = \min\{|u| : u \in A^+, A^*u \cup A^*x[0\,..\,i-1] \neq \emptyset \text{ et } uA^* \cup x[i\,..\,|x|-1]A^* \neq \emptyset\}$$

la période locale de x en i.

En notant $w = SM(\leq, x)$ (SM est défini dans l'exercice 1.28) et en supposant que $|w| \leq |SM(\leq^{-1}, x)|$, montrer que $rép_x(|x|-|w|) = pér(x)$. [*Aide :* noter que l'intersection des deux ordres induits sur les mots est l'ordre préfixe, et utiliser la proposition 1.4 ; voir Crochemore et Perrin (1991) ou [5].]

2 Automates de localisation

Dans ce chapitre, nous abordons le problème de la localisation d'un motif dans un texte lorsque ce motif représente un ensemble fini de mots. Nous présentons des solutions basées sur l'utilisation d'automates. Remarquons tout d'abord que l'utilisation d'un automate comme solution au problème est tout à fait naturelle : étant donné un langage fini $X \subseteq A^*$, localiser toutes les occurrences de mots appartenant à X dans un texte $y \in A^*$ revient à déterminer tous les préfixes de y qui se terminent par un mot de X ; ce qui revient à reconnaitre le langage A^*X ; et comme A^*X est un langage rationnel, cela peut être réalisé par un automate. Ajoutons ensuite que de telles solutions sont tout à fait adaptées aux cas où un motif est à localiser dans des données qui doivent être traitées de façon séquentielle : ligne de communication, téléchargement, virus.

L'utilisation d'un automate pour localiser un motif a déjà été abordée section 1.5. Nous complétons ici le sujet en précisant notamment comment obtenir les automates déterministes évoqués au début de cette section de référence. Les complexités des méthodes exposées en fin de section 1.5 valides pour les automates non déterministes sont également comparées avec celles présentées dans ce chapitre.

Le plan se décompose comme suit. Nous exhibons un type d'automates déterministes et complets reconnaissant le langage A^*X. Nous envisageons deux implantations réduites de ce type d'automates. La première utilise une fonction de suppléance et la seconde l'état initial comme successeur par défaut (notions introduites dans la section 1.4). Chacune des deux implantations possède son propre avantage : tandis que la première réalise une implantation de taille linéaire en la somme des longueurs des mots de X, la seconde assure naturellement une localisation en temps réel lorsque l'alphabet est considéré comme constant. Nous considérons à part le cas où l'ensemble X est réduit à un seul mot et nous montrons que le délai de l'algorithme de localisation est logarithmique en la longueur du mot pour les deux implantations envisagées.

2.1 Arbre d'un dictionnaire

Soient $X \subseteq A^*$ un ***dictionnaire*** (sur A), c'est-à-dire un langage fini non vide ne contenant pas le mot vide ε, et $y \in A^*$ le texte dans lequel on cherche à localiser toutes les occurrences de mots de X.

La méthode décrite dans la suite du chapitre est basée sur un automate qui reconnait X. On le note $\mathcal{A}(X)$. C'est l'automate dont :

- l'ensemble des états est $\mathit{Préf}(X)$;
- l'état initial est le mot vide ε ;
- l'ensemble des états terminaux est X ;
- les flèches sont de la forme (u, a, ua).

Proposition 2.1
L'automate $\mathcal{A}(X)$ est déterministe. Il reconnait X.

Preuve Immédiate. ∎

On appelle $\mathcal{A}(X)$ l'***arbre*** du dictionnaire X (que l'on confond avec l'arborescence dont le sommet distingué – la racine – est l'état initial de l'automate). La figure 2.1 illustre le propos.

La fonction ARBRE, dont le code est donné ci-dessous, produit l'arbre de tout dictionnaire X. Elle considère successivement chacun des mots de X dans la boucle **pour** des lignes 2–10 et l'insère au sein de la structure en construction lettre par lettre pendant l'exécution de la boucle **pour** des lignes 4–9.

```
ARBRE(X)
 1  M ← NOUVEL-AUTOMATE()
 2  pour chaque mot x ∈ X faire
 3        t ← initial[M]
 4        pour chaque lettre a de x, séquentiellement faire
 5              p ← CIBLE(t, a)
 6              si p = NIL alors
 7                    p ← NOUVEL-ÉTAT()
 8                    Succ[t] ← Succ[t] ∪ {(a, p)}
 9              t ← p
10        terminal[t] ← VRAI
11  retourner M
```

Proposition 2.2
L'opération ARBRE(X) *produit l'automate $\mathcal{A}(X)$.* ∎

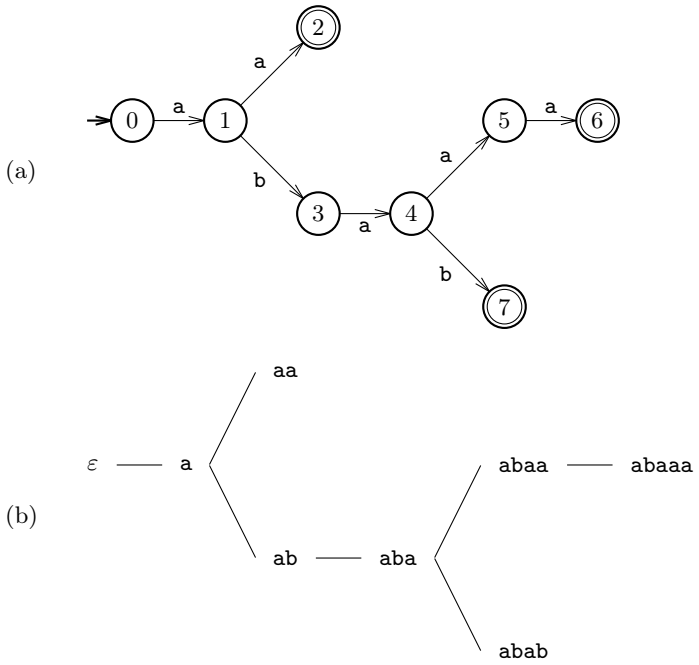


Figure 2.1 **(a)** L'arbre $\mathcal{A}(X)$ lorsque $X = \{\texttt{aa}, \texttt{abaaa}, \texttt{abab}\}$. Le langage reconnu par l'automate $\mathcal{A}(X)$ est X. Les états sont identifiés aux préfixes des mots de X. Par exemple l'état 3 correspond au préfixe de longueur 2 de `abaaa` et `abab`. **(b)** Représentation arborescente de X.

2.2 Localisation de plusieurs mots

Nous présentons dans cette section un automate déterministe et complet qui reconnait le langage A^*X. La particularité de cet automate est que ses états sont les préfixes des mots de X : lors d'une lecture séquentielle du texte, il suffit effectivement – nous allons le voir – de ne retenir que le plus long des suffixes de la partie de texte déjà lue qui est un préfixe d'un mot de X. L'automate que nous considérons n'est pas minimal dans le cas général, mais il est relativement simple à construire. Il est également à la base des diverses constructions abordées dans les sections ultérieures. L'automate possède les mêmes états que $\mathcal{A}(X)$ et le même état initial. Il contient les états terminaux et les flèches de $\mathcal{A}(X)$.

Dans la suite, nous indiquons une construction de l'automate dissociée de la phase de localisation. On peut aussi envisager de le construire de façon « paresseuse », c'est-à-dire au fur et à mesure des besoins de la localisation. Cette construction est laissée en exercice (exercice 2.4).

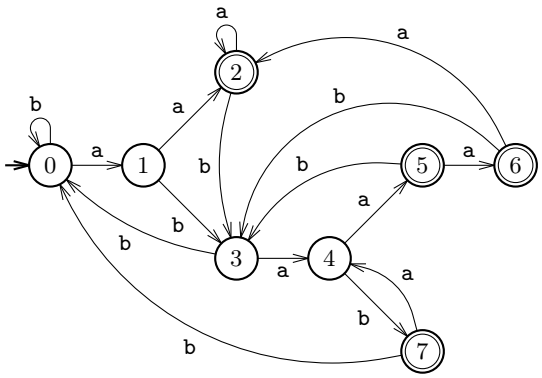


Figure 2.2 L'automate-dictionnaire $\mathcal{D}(X)$ lorsque $X = \{\texttt{aa}, \texttt{abaaa}, \texttt{abab}\}$ et $A = \{\texttt{a}, \texttt{b}\}$. L'automate $\mathcal{D}(X)$ reconnait le langage A^*X. Comparé à l'arbre du même dictionnaire illustré figure 2.1, on remarque que l'état 5 est terminal : il correspond à `abaa` dont un suffixe – `aa` en l'occurrence – appartient à X.

Automate-dictionnaire

Afin de formaliser l'automate de localisation du dictionnaire $X \subseteq A^*$, on introduit la fonction :

$h\colon A^* \to \mathit{Préf}(X)$

définie par :

$h(u) =$ le plus long suffixe de u qui appartient à $\mathit{Préf}(X)$

pour tout mot $u \in A^*$. Maintenant, soit $\mathcal{D}(X)$ l'automate dont :

- l'ensemble des états est $\mathit{Préf}(X)$;
- l'état initial est le mot vide ε ;
- l'ensemble des états terminaux est $\mathit{Préf}(X) \cap A^*X$;
- les flèches sont de la forme $(u, a, h(ua))$.

La preuve de la proposition suivante, qui s'appuie sur le lemme 2.4, est reportée après la preuve de celui-ci.

Proposition 2.3
L'automate $\mathcal{D}(X)$ est déterministe et complet. Il reconnait A^*X.

On appelle $\mathcal{D}(X)$ l'***automate-dictionnaire*** de X. Une illustration est donnée figure 2.2. La preuve de la proposition repose sur le résultat suivant.

Lemme 2.4
La fonction h satisfait les propriétés suivantes :

1. *$u \in A^*X$ si et seulement si $h(u) \in A^*X$, pour tout $u \in A^*$.*

2. $h(\varepsilon) = \varepsilon$.
3. $h(ua) = h(h(u)a)$, *pour tout* $(u, a) \in A^* \times A$.

Preuve Soient $u \in A^*$ et $a \in A$.

Supposons que $u \in A^*X$. Le mot u se décompose alors sous la forme vx avec $v \in A^*$ et $x \in X$. Or, par définition de h, $x \preceq_{suff} h(u)$. Il s'ensuit que $h(u) \in A^*X$. Réciproquement, supposons que $h(u) \in A^*X$. Comme $h(u) \preceq_{suff} u$, $u \in A^*X$. Ce qui prouve la propriété 1.

La propriété 2 est clairement vérifiée.

Il reste à montrer la propriété 3. Les mots $h(ua)$ et $h(u)a$ étant tous deux des suffixes de ua, l'un de ces deux mots est un suffixe de l'autre. Nous envisageons consécutivement les deux éventualités.

Première éventualité : $h(u)a \prec_{suff} h(ua)$. Soit v le mot défini par :

$$v = h(ua)a^{-1}.$$

Alors $h(u) \prec_{suff} v \preceq_{suff} u$. Et comme $h(ua) \in \mathit{Préf}(X)$, $v \in \mathit{Préf}(X)$. Il s'ensuit que v est un mot qui contredit la maximalité de $h(u)$. Cette première éventualité est donc impossible.

Seconde éventualité : $h(ua) \preceq_{suff} h(u)a$. Alors $h(ua) \preceq_{suff} h(h(u)a)$. Et comme $h(u)a \preceq_{suff} ua$, $h(h(u)a) \preceq_{suff} h(ua)$. D'où $h(ua) = h(h(u)a)$.

Ce qui établit la propriété 3 et achève la preuve. ∎

Preuve de la proposition 2.3 Soit $z \in A^*$. D'après les propriétés 2 et 3 du lemme 2.4, il vient que la suite des flèches de la forme

$$(h(z[0\,..\,i-1]), z[i], h(z[0\,..\,i]))\ ,$$

avec $i = 0, 1, \ldots, |z| - 1$, est un chemin dans $\mathcal{D}(X)$ des états ε à $h(z)$ étiqueté par z. Puis, comme $h(z) \in \mathit{Préf}(X)$, il se déduit du lemme 2.4 que $z \in A^*X$ si et seulement si $h(z) \in \mathit{Préf}(X) \cap A^*X$. Ce qui montre que $\mathcal{D}(X)$ reconnait le langage A^*X. ∎

Construction de l'automate-dictionnaire

L'algorithme de construction de l'automate-dictionnaire $\mathcal{D}(X)$ à partir de l'arbre $\mathcal{A}(X)$ proposé dans la suite utilise un parcours en largeur de l'arbre. Conjointement à la fonction h définie plus haut, on introduit la fonction :

$$f\colon A^* \to \mathit{Préf}(X)$$

définie par :

$f(u) =$ le plus long suffixe propre de u qui appartient à $\mathit{Préf}(X)$

pour tout mot $u \in A^+$ et non définie en ε.

Les trois résultats qui suivent montrent qu'il suffit de connaitre l'état $f(u)$ pour chacun des états $u \neq \varepsilon$ rencontrés lors du parcours de l'arbre, afin d'assurer une construction correcte de $\mathcal{D}(X)$.

Lemme 2.5
Pour tout $(u, a) \in A^* \times A$ *on a :*

$$h(ua) = \begin{cases} ua & \text{si } ua \in \mathit{Préf}(X) \ , \\ h(f(u)a) & \text{si } u \neq \varepsilon \text{ et } ua \notin \mathit{Préf}(X) \ , \\ \varepsilon & \text{sinon} \ . \end{cases}$$

Preuve L'identité est triviale lorsque $ua \in \mathit{Préf}(X)$ ou lorsque $u = \varepsilon$ et $ua \notin \mathit{Préf}(X)$. Reste à examiner le cas où $u \neq \varepsilon$ et $ua \notin \mathit{Préf}(X)$. Si l'on suppose l'existence d'un suffixe v de ua tel que $v \in \mathit{Préf}(X)$ et $|v| > |f(u)a|$, il vient que va^{-1} est un suffixe propre de u qui appartient à $\mathit{Préf}(X)$; ce qui contredit la maximalité de $f(u)$. Il s'ensuit que $h(f(u)a)$ est le plus long suffixe de ua qui appartient à $\mathit{Préf}(X)$, ce qui valide la dernière identité qui restait à établir et termine la preuve. ∎

Lemme 2.6
Pour tout $(u, a) \in A^* \times A$ *on a :*

$$f(ua) = \begin{cases} h(f(u)a) & \text{si } u \neq \varepsilon \ , \\ \varepsilon & \text{sinon} \ . \end{cases}$$

Preuve Examinons le cas où $u \in A^+$. Si l'on suppose l'existence d'un suffixe v de ua tel que $v \in \mathit{Préf}(X)$ et $|v| > |f(u)a|$, il vient que va^{-1} est un suffixe propre de u qui appartient à $\mathit{Préf}(X)$, ce qui contredit la maximalité de $f(u)$. Il s'ensuit que $f(ua)$, le plus long suffixe propre de ua qui appartient à $\mathit{Préf}(X)$, est un suffixe de $f(u)a$. Par maximalité de h, le suffixe en question est également $h(f(u)a)$, ce qui achève la preuve. ∎

Lemme 2.7
Pour tout $u \in A^*$ *on a :*

$u \in A^*X$ *si et seulement si* $u \in X$ *ou* $(u \neq \varepsilon$ *et* $f(u) \in A^*X)$.

Preuve Il est clairement suffisant de montrer que :

$u \in (A^*X) \setminus X$ implique $f(u) \in A^*X$,

car alors $u \neq \varepsilon$ puisque $\varepsilon \notin X$. Ainsi, soit $u \in (A^*X) \setminus X$. Le mot x est de la forme vw où $v \in A^*$ et w est un suffixe propre de u appartenant à X. Il s'ensuit que, par définition de f, w est un suffixe de $f(u)$. Donc $f(u) \in A^*X$. Ce qui termine la preuve. ∎

La fonction ALP-COMPLET, dont le code suit, implante l'algorithme de construction de $\mathcal{D}(X)$. Les trois premières lettres de son identificateur signifient « automate de localisation de plusieurs mots ». Une étape de l'exécution de la fonction est illustrée figure 2.3.

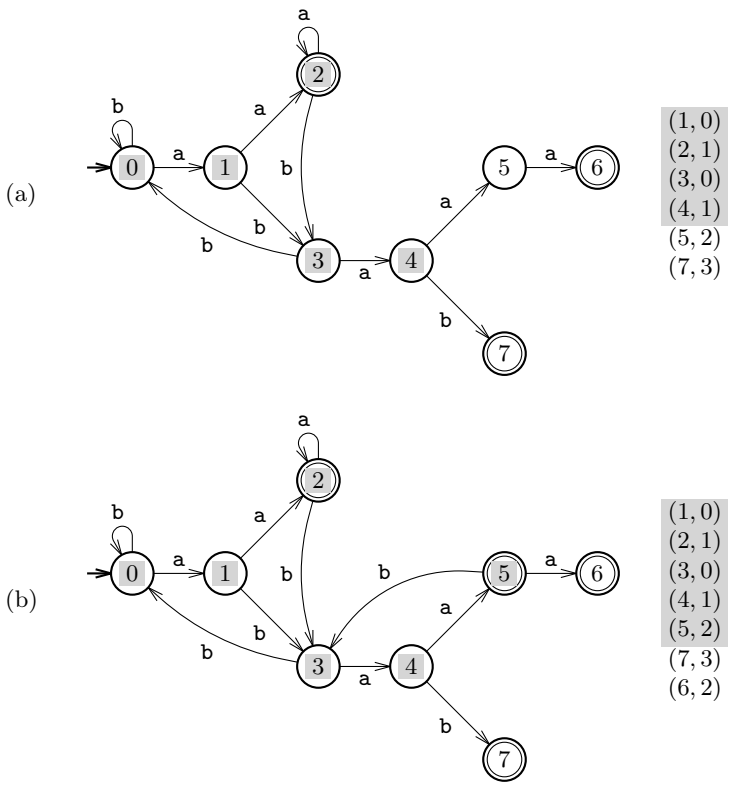


Figure 2.3 Une étape de l'exécution de l'opération ALP-COMPLET(X) avec $X = \{\mathtt{aa}, \mathtt{abaaa}, \mathtt{abab}\}$ et $A = \{\mathtt{a}, \mathtt{b}\}$. **(a)** Les états 0, 1, 2, 3 et 4 de l'automate ont déjà été visités. La structure en construction coïncide avec $\mathcal{D}(X)$ sur ces états, et avec $\mathcal{A}(X)$ sur ceux qui restent à visiter. La file contient deux éléments : les couples $(5, 2)$ et $(7, 3)$. **(b)** L'étape, à proprement parler. L'élément $(5, 2)$ est retiré de la file. Comme l'état 2 est terminal, l'état 5 est à son tour rendu terminal. Cela correspond au fait que `aa`, qui appartient à X, est un suffixe de `abaa`, mot associé à l'état 5. La fonction ALP-COMPLET considère ensuite les deux flèches qui sortent de l'état 2, à savoir les flèches $(2, \mathtt{a}, 2)$ et $(2, \mathtt{b}, 3)$. Pour la première, et puisqu'il existe déjà une transition par la lettre `a` à partir de l'état 5 et que la cible de la flèche associée est l'état 6, elle ajoute le couple $(6, 2)$ à la file. Tandis que pour la seconde, elle ajoute la flèche $(5, \mathtt{b}, 3)$ à la structure.

```
ALP-COMPLET(X)
 1  M ← ARBRE(X)
 2  q0 ← initial[M]
 3  F ← FILE-VIDE()
 4  pour chaque lettre a ∈ A faire
 5        q ← CIBLE(q0, a)
 6        si q = NIL alors
 7              Succ[q0] ← Succ[q0] ∪ {(a, q0)}
 8        sinon ENFILER(F, (q, q0))
 9  tantque non FILE-EST-VIDE(F) faire
10        (p, r) ← DÉFILEMENT(F)
11        si terminal[r] alors
12              terminal[p] ← VRAI
13        pour chaque lettre a ∈ A faire
14              q ← CIBLE(p, a)
15              s ← CIBLE(r, a)
16              si q = NIL alors
17                    Succ[p] ← Succ[p] ∪ {(a, s)}
18              sinon ENFILER(F, (q, s))
19  retourner M
```

La fonction ALP-COMPLET procède comme suit. Elle commence par construire l'automate $\mathcal{A}(X)$ à la ligne 1. Elle initialise ensuite, des lignes 3 à 8, la file F avec les couples d'états qui correspondent aux couples de la forme (a, ε) avec $a \in A \cap \mathit{Préf}(X)$. Dans le même temps, elle ajoute à l'état initial q_0 les flèches de la forme (q_0, a, q_0) pour $a \in A \setminus \mathit{Préf}(X)$. On peut dès lors supposer qu'à chaque couple d'états (p, r) dans la file F correspond un couple de la forme $(u, f(u))$ avec $u \in \mathit{Préf}(X) \setminus \{\varepsilon\}$, que l'ensemble des successeurs étiquetés de chacun des états déjà visités est celui qu'il admet dans $\mathcal{D}(X)$, et dans $\mathcal{A}(X)$ pour les autres. Cela constitue un invariant de la boucle **tantque** des lignes 9–18. En effet, à chacune des $\operatorname{card} A$ flèches (r, a, s) considérées ligne 15 correspond une flèche de la forme $(f(u), a, h(f(u)a))$. À ce point, de deux choses l'une :

– S'il n'y a pas déjà de transition définie de source p et d'étiquette a dans la structure, c'est que $ua \notin \mathit{Préf}(X)$. Le lemme 2.5 indique alors que $h(ua) = h(f(u)a)$. Il suffit donc bien d'ajouter la flèche (p, a, s) comme réalisé à la ligne 17.

– Sinon $ua \in \mathit{Préf}(X)$, auquel cas $h(ua) = ua$, mot qui correspond à l'état q. L'instruction à la ligne 18 ajoute alors le couple (q, s) à la file F de manière à pouvoir poursuivre le parcours en largeur. Du reste, le lemme 2.6 indique que $h(f(u)a) = f(ua)$. Cela montre que le couple (q, s) est bien de la forme annoncée.

Quant aux états terminaux spécifiques à $\mathcal{D}(X)$, ils sont marqués par la conditionnelle **si** des lignes 11–12 conformément au lemme 2.7. Cela prouve le résultat suivant.

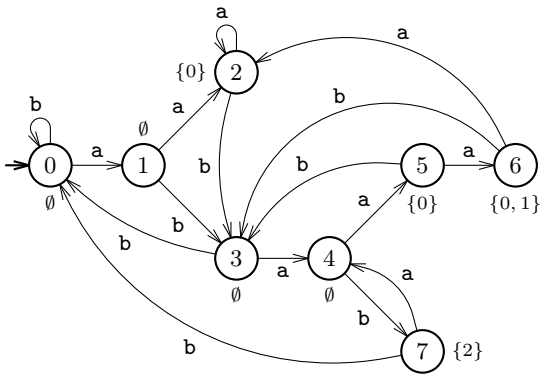


Figure 2.4 Version avec sorties de l'automate-dictionnaire de la figure 2.2 obtenue en numérotant les mots du dictionnaire : 0 pour `aa`, 1 pour `abaaa` et 2 pour `abab`. La sortie de chaque état correspond à l'ensemble des mots de X qui sont des suffixes du préfixe associé à l'état. Les états terminaux sont ceux qui possèdent une sortie non vide.

Proposition 2.8
L'opération ALP-COMPLET(X) *produit l'automate* $\mathcal{D}(X)$. ∎

Report des occurrences

Pour faire opérer l'automate $\mathcal{D}(X)$ sur le texte y, on peut utiliser l'algorithme LA-DÉTERMINISTE décrit section 1.5. Celui-ci signale une occurrence à chaque fois qu'une occurrence de l'un des mots de X se termine à la position courante. Le marquage des états terminaux peut toutefois être plus précis afin de permettre de repérer quels sont les mots de X qui apparaissent à une position donnée sur le texte. On associe pour cela une sortie à chaque état.

Notons $x_0, x_1, \ldots, x_{k-1}$ les $k = \operatorname{card} X$ mots de X. On définit la sortie d'un état u de $\mathcal{D}(X)$ comme l'ensemble des indices i pour lesquels x_i est un suffixe de u (voir l'illustration donnée figure 2.4). Ainsi une occurrence du mot x_i de X se termine-t-elle à la position courante sur le texte si et seulement si l'indice i est un élément de la sortie de l'état courant. En remarquant que seuls les états terminaux possèdent une sortie non vide, le calcul des sorties – en lieu et place de celui des états terminaux – peut procéder comme suit :

1. L'instruction à la ligne 2 de la fonction NOUVEL-ÉTAT est remplacée par l'affectation $\mathit{sortie}[p] \leftarrow \emptyset$ (la fonction NOUVEL-ÉTAT est appelée à la ligne 7 de la fonction ARBRE ; son code est donné section 1.3).
2. L'instruction à la ligne 10 de la fonction ARBRE est remplacée par l'affectation $\mathit{sortie}[t] \leftarrow \{i\}$ où i est l'indice du mot de X pris en compte par la boucle **pour** aux lignes 2–10.

3. L'instruction aux lignes 11–12 de la fonction ALP-COMPLET est remplacée par l'affectation $\mathit{sortie}[p] \leftarrow \mathit{sortie}[p] \cup \mathit{sortie}[r]$.

Le test d'occurrence ligne 4 de l'algorithme LA-DÉTERMINISTE devient $\mathit{sortie}[r] \neq \emptyset$. Dans le cas où ce test se révèle positif, des occurrences de mots de X en particulier peuvent alors être signalées (voir la note 4 page 20).

Implantation par matrice de transition

L'automate $\mathcal{D}(X)$ étant complet, il est naturel d'implanter sa fonction de transition par matrice de transition.

Proposition 2.9
Lorsque l'automate $\mathcal{D}(X)$ est implanté à l'aide d'une matrice de transition, la taille de l'implantation est $O(|X| \times \operatorname{card} A)$, et le temps pour la construire par la fonction ALP-COMPLET *est $O(|X| \times \operatorname{card} A)$. L'espace supplémentaire nécessaire à l'exécution de la fonction* ALP-COMPLET *est $O(\operatorname{card} X)$.*

Preuve Le nombre d'états de $\mathcal{D}(X)$ est égal à $\operatorname{card} \mathit{Préf}(X)$, nombre lui-même inférieur à $|X| + 1$. D'autre part, la fonction de transition étant implantée par une matrice, chacune des consultations (fonction CIBLE) ou modifications (ajouts aux ensembles des successeurs étiquetés) prend un temps $O(1)$. Enfin, la file – qui constitue l'essentiel de l'espace nécessaire au calcul – contient toujours moins d'éléments qu'il n'y a de branches dans l'arbre, soit moins de $\operatorname{card} X$ éléments. Les complexités annoncées s'en déduisent. ∎

L'algorithme LA-DÉTERMINISTE de la section 1.5 peut être utilisé pour faire opérer l'automate $\mathcal{D}(X)$ sur le texte y. On a alors le résultat suivant, lequel est une conséquence immédiate de la proposition 1.15.

Proposition 2.10
La localisation des occurrences des mots de X dans un texte y s'effectue en temps $O(|y|)$ si l'on utilise l'automate $\mathcal{D}(X)$ implanté à l'aide d'une matrice de transition. Le délai ainsi que l'espace mémoire supplémentaire sont constants. ∎

Remarquer que, par comparaison avec les résultats de la section 1.5, le passage par l'automate $\mathcal{D}(X)$ permet de faire gagner un facteur $O(|X|)$ dans les complexités spatiotemporelles de la phase de localisation : si celle-ci est réalisée par l'algorithme LA-NON-DÉTERMINISTE à partir de l'automate $\mathcal{A}(X)$ considéré dans le même modèle (le modèle branchements, avec implantation par matrice de transitions) et que les ensembles d'états de $\mathcal{A}(X)$ sont codés à l'aide de vecteurs booléens, le temps est en effet $O(|X| \times |y|)$, le délai et l'espace mémoire supplémentaire $O(|X|)$.

Cela dit, l'espace mémoire et le temps nécessaires afin de mémoriser et construire l'automate peuvent conduire à écarter une telle implantation lorsque la taille de l'alphabet A est grande relativement à X. On montre dans les deux sections suivantes deux méthodes qui implantent l'automate en temps et en espace indépendants de l'alphabet dans le modèle comparaisons.

2.3 Implantation avec fonction de suppléance

Nous présentons dans cette section une implantation réduite de l'automate $\mathcal{D}(X)$ dans le modèle comparaisons à l'aide d'une fonction de suppléance (voir section 1.4). Cette fonction n'est autre que la fonction f, définie section 2.2, qui à tout mot non vide associe son plus long suffixe propre appartenant à l'ensemble des préfixes des mots de X.

Nous commençons par préciser l'implantation. Nous nous intéressons ensuite à son utilisation pour la localisation des occurrences des mots de X dans un texte, puis à sa construction. Enfin, nous indiquons une optimisation possible de la fonction de suppléance.

Définition de l'implantation

Soit :

$\gamma\colon \mathit{Préf}(X) \times A \to \mathit{Préf}(X)$

la fonction partiellement définie par :

$$\gamma(u,a) = \begin{cases} ua & \text{si } ua \in \mathit{Préf}(X)\ , \\ \varepsilon & \text{si } u = \varepsilon \text{ et } a \notin \mathit{Préf}(X)\ . \end{cases}$$

Proposition 2.11
Les fonctions γ et f sont respectivement une sous-transition et une fonction de suppléance de la fonction de transition de $\mathcal{D}(X)$.

Preuve Pour tout préfixe non vide u de $\mathit{Préf}(X)$, $f(u)$ est défini et l'on a $|f(u)| < |u|$. Il s'en déduit que f définit un ordre sur $\mathit{Préf}(X)$, ensemble des états de $\mathcal{D}(X)$. La fonction $\delta\colon \mathit{Préf}(X) \times A \to \mathit{Préf}(X)$ définie par $\delta(u,a) = h(ua)$ pour tout couple $(u,a) \in \mathit{Préf}(X) \times A$ est la fonction de transition de l'automate. On vérifie facilement à l'aide du lemme 2.5 que :

$$\delta(u,a) = \begin{cases} \gamma(u,a) & \text{si } \gamma(u,a) \text{ est défini}\ , \\ \delta(f(u),a) & \text{sinon}\ . \end{cases}$$

Cela montre que γ et f sont bien une sous-transition et une fonction de suppléance de δ comme attendu (voir section 1.4). ∎

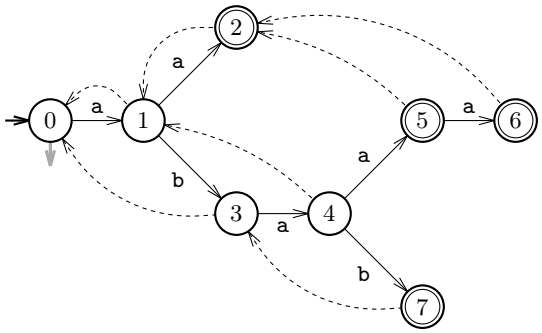


Figure 2.5 Implantation $\mathcal{D}_{\mathrm{S}}(X)$ de l'automate-dictionnaire $\mathcal{D}(X)$ lorsque $X = \{\mathtt{aa}, \mathtt{abaaa}, \mathtt{abab}\}$. (Se reporter figure 2.3 où les suppléants des états, calculés lors du parcours en largeur de l'arbre $\mathcal{A}(X)$, sont également indiqués en marge.)

Soit $\mathcal{D}_{\mathrm{S}}(X)$ la structure constituée :

- de l'automate $\mathcal{A}(X)$ dont la fonction de transition est implantée par ensembles des successeurs étiquetés ;
- de l'état initial de $\mathcal{A}(X)$ comme successeur par défaut de lui-même ;
- de la fonction de suppléance f.

Une illustration est donnée figure 2.5.

Théorème 2.12
Soit X un dictionnaire. Alors $\mathcal{D}_{\mathrm{S}}(X)$ est une implantation de l'automate-dictionnaire $\mathcal{D}(X)$ de taille $O(|X|)$.

Preuve Le fait que $\mathcal{D}_{\mathrm{S}}(X)$ soit une implantation de $\mathcal{D}(X)$ est une conséquence des définitions de $\mathcal{A}(X)$ et de γ ainsi que de la proposition 2.11. Quant à la taille de l'implantation, elle est linéaire en le nombre d'états de $\mathcal{A}(X)$, lequel nombre est majoré par $|X| + 1$. ∎

Phase de localisation

La localisation des occurrences des mots de X dans un texte y à l'aide de l'implantation $\mathcal{D}_{\mathrm{S}}(X)$ nécessite la simulation des transitions de l'automate $\mathcal{D}(X)$ avec le successeur par défaut et la fonction de suppléance. On considère l'attribut *suppléant* adjoint à chacun des objets états. Pour l'état initial de l'objet automate M, on pose :

$\mathit{suppléant}[\mathit{initial}[M]] = \textsc{nil}$

pour signifier que le suppléant n'est pas défini pour cet état. Le code donné ci-dessous réalise la simulation. L'objet automate M est global.

CIBLE-PAR-SUPPLÉANCE(p, a)
1 **tantque** $p \neq$ NIL et CIBLE$(p, a) =$ NIL **faire**
2 $\quad p \leftarrow$ *suppléant*$[p]$
3 **si** $p =$ NIL **alors**
4 $\quad$ **retourner** *initial*$[M]$
5 **sinon retourner** CIBLE(p, a)

La boucle **tantque** des lignes 1–2 et le retour de la fonction aux lignes 3–5 sont corrects puisque conformes à la notion de suppléant.

On adapte l'algorithme LA-DÉTERMINISTE (section 1.5) pour localiser les occurrences en modifiant toutefois son code : la construction de l'automate est effectuée au sein de l'algorithme par la fonction ALP-PAR-SUPPLÉANCE précisée plus loin ; la ligne 3, qui correspond à la ligne 4 dans le code ci-dessous, fait appel à la fonction CIBLE-PAR-SUPPLÉANCE à la place de la fonction CIBLE, ce qui donne le code suivant.

LA-DÉTERMINISTE-PAR-SUPPLÉANCE(X, y)
1 $M \leftarrow$ ALP-PAR-SUPPLÉANCE(X)
2 $r \leftarrow$ *initial*$[M]$
3 **pour** chaque lettre a de y, séquentiellement **faire**
4 $\quad r \leftarrow$ CIBLE-PAR-SUPPLÉANCE(r, a)
5 $\quad$ SIGNALER-SI(*terminal*$[r]$)

Lemme 2.13
Le nombre de tests CIBLE$(p, a) =$ NIL *réalisés durant la phase de localisation de l'opération* LA-DÉTERMINISTE-PAR-SUPPLÉANCE(X, y) *est inférieur à* $2|y| - 1$.

Preuve Notons u le préfixe courant de y et $|p|$ le niveau dans $\mathcal{A}(X)$ de l'état p de la fonction CIBLE-PAR-SUPPLÉANCE en posant $|p| = -1$ lorsque $p =$ NIL. Alors la quantité $2|u| - |p|$ croît d'au moins une unité après chacun des tests en question. En effet : les quantités $|p|$ et $|u|$ sont incrémentées à la suite d'un résultat positif au test ; la quantité $|p|$ est diminuée d'au moins une unité et la quantité $|u|$ reste inchangée à la suite d'un résultat négatif au test. D'autre part, la quantité $2|u| - |p|$ vaut $2 \times 0 - 0 = 0$ au premier test et au plus $2 \times (|y| - 1) - 0 = 2|y| - 2$ au dernier. Il s'ensuit que le nombre de tests est bien majoré par $2|y| - 1$ comme annoncé. ∎

Lemme 2.14
Le maximum des degrés sortants des états de l'automate $\mathcal{A}(X)$ *est inférieur à* $\min\{\operatorname{card} \mathit{alph}(X), \operatorname{card} X\}$.

Preuve L'automate étant déterministe, les flèches sortant de l'un quelconque de ses états sont étiquetées par des lettres deux à deux distinctes qui apparaissent dans les mots de X. De plus, il possède au plus $\operatorname{card} X$

états (nœuds) externes, donc chaque état possède au plus card X flèches sortantes. La borne s'en déduit. ∎

Théorème 2.15
Le temps d'exécution de la phase de localisation des occurrences des mots de X dans un texte y pour l'algorithme LA-DÉTERMINISTE-PAR-SUPPLÉANCE *est $O(|y| \times \log s)$ et le délai $O(\ell \times \log s)$ où*

$s \leq \min\{\operatorname{card} alph(X), \operatorname{card} X\}$

est le maximum des degrés sortants des états de l'arbre $\mathcal{A}(X)$ et ℓ le maximum des longueurs des mots de X.

Preuve Comme le cout de chaque test CIBLE$(p, a) =$ NIL est $O(\log s)$ (proposition 1.16), que le nombre de ces tests est linéaire en la longueur de y (lemme 2.13), que le temps d'exécution de ces tests est représentatif du temps total de la localisation, ce dernier est $O(|y| \times \log s)$.

Lors de l'exécution de l'opération CIBLE-PAR-SUPPLÉANCE(p, a), le nombre d'exécutions du corps de la boucle **tantque** des lignes 1–2 ne peut excéder le niveau de l'état dans l'arbre, d'où la borne du délai, par application de la proposition 1.16.

Reste à ajouter que la majoration sur s provient du lemme 2.14. ∎

Pour terminer la partie consacrée à la phase de localisation, nous montrons que la borne du nombre de tests donnée dans le lemme 2.13 est optimale sur tout alphabet d'au moins deux lettres.

Proposition 2.16
Lorsque card $A \geq 2$, *il existe un dictionnaire X et un texte non vide y pour lesquels le nombre de tests* CIBLE$(p, a) =$ NIL *réalisés lors de la localisation des occurrences des mots de X dans y par l'algorithme* LA-DÉTERMINISTE-PAR-SUPPLÉANCE *est égal à* $2|y| - 1$.

Preuve Soient a et b deux lettres distinctes de A. Considérons un dictionnaire X qui compte dans $Préf(X)$ le mot ab mais pas le mot aa. Supposons de plus que $y \in \{a\}^*$. Alors le test en question est exécuté une fois sur la première lettre de y, et deux fois sur chacune des suivantes. Le résultat annoncé s'en déduit. ∎

Construction de l'implantation

On construit l'implantation $\mathcal{D}_S(X)$ à partir d'un parcours en largeur de l'arbre $\mathcal{A}(X)$. Mais le procédé à mettre en œuvre est plus simple que celui donné pour $\mathcal{D}(X)$ car :

– aucune flèche n'est à ajouter à la structure ;
– il n'est plus besoin de ranger dans la file un état et son suppléant.

Suit le code de la fonction ALP-PAR-SUPPLÉANCE qui produit l'implantation $\mathcal{D}_S(X)$.

```
ALP-PAR-SUPPLÉANCE(X)
 1  M ← ARBRE(X)
 2  suppléant[initial[M]] ← NIL
 3  F ← FILE-VIDE()
 4  ENFILER(F, initial[M])
 5  tantque non FILE-EST-VIDE(F) faire
 6        t ← DÉFILEMENT(F)
 7        pour chaque couple (a, p) ∈ Succ[t] faire
 8              r ← CIBLE-PAR-SUPPLÉANCE(suppléant[t], a)
 9              suppléant[p] ← r
10              si terminal[r] alors
11                    terminal[p] ← VRAI
12              ENFILER(F, p)
13  retourner M
```

Le seul point délicat du code se situe aux lignes 8–9 dans le cas où t est l'état initial de l'automate. Remarquons maintenant que la fonction CIBLE-PAR-SUPPLÉANCE produit l'état initial lorsqu'elle considère un paramètre de type état de valeur NIL en entrée. Cela suffit pour montrer que les instructions aux lignes 8–9 sont bien conformes au lemme 2.6.

Proposition 2.17
L'opération ALP-PAR-SUPPLÉANCE(X) *produit* $\mathcal{D}_{\mathrm{s}}(X)$*, implantation de* $\mathcal{D}(X)$ *par fonction de suppléance.* ∎

Théorème 2.18
Le temps d'exécution pour l'opération ALP-PAR-SUPPLÉANCE(X) *est* $O(|X| \times \log \min\{\operatorname{card} \mathit{alph}(X), \operatorname{card} X\})$*. L'espace mémoire supplémentaire requis par cette opération est* $O(\operatorname{card} X)$*.*

Preuve Temps d'exécution : renommons s la variable de type état de la fonction CIBLE-PAR-SUPPLÉANCE ; on procède de la même façon que pour la preuve du lemme 2.13, mais en s'intéressant cette fois-ci aux quantités $2|t| - |s|$ considérées le long de chacune des différentes branches de l'arbre $\mathcal{A}(X)$; on remarque ensuite que la somme des longueurs des branches est majorée par $|X|$; puis on utilise le lemme 2.14. Espace supplémentaire : voir la preuve de la proposition 2.9. ∎

Optimisation de la fonction de suppléance

La phase de localisation peut être sensiblement améliorée si les appels inutiles à la fonction de suppléance sont éliminés.

Reprenons l'exemple donné figure 2.5 et étudions deux cas.

– Supposons que l'état 6 soit atteint. Quelle que soit la valeur de la lettre courante du texte, la fonction de suppléance est appelée deux fois de suite, avant d'atteindre finalement l'état 1. Il est donc préférable de choisir 1 comme suppléant de 6.

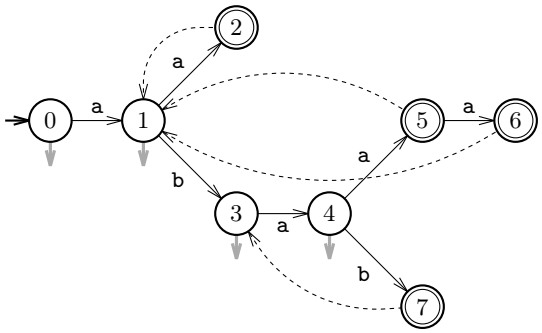


Figure 2.6 La représentation optimisée de l'implantation $\mathcal{D}_{\mathrm{s}}(X)$ lorsque $X = \{\mathtt{aa}, \mathtt{abaaa}, \mathtt{abab}\}$ (l'implantation originelle est donnée figure 2.5).

– Supposons maintenant que l'état 4 soit atteint. Si la lettre courante n'est ni `a`, ni `b`, il est inutile de transiter par les états 1 puis 0 pour finalement revenir à l'état initial 0 et passer à la lettre suivante. Le calcul peut là aussi se faire en une seule étape en considérant l'état initial comme successeur par défaut pour l'état 4.

En suivant un raisonnement analogue pour chacun des états, on obtient la représentation optimisée donnée figure 2.6.

Formellement, l'implantation $\mathcal{D}_{\mathrm{s}}(X)$ de l'automate $\mathcal{D}(X)$ peut être optimisée pour la phase de localisation en considérant une autre fonction de suppléance que la fonction f. Notons $\mathit{Suivant}(u)$ l'ensemble défini pour tout mot $u \in \mathit{Préf}(X)$ par :

$$\mathit{Suivant}(u) = \{a : a \in A, ua \in \mathit{Préf}(X)\} \ .$$

Soit maintenant f' la fonction de $\mathit{Préf}(X)$ dans lui-même définie par $f'(u) = f^k(u)$ pour tout mot $u \in \mathit{Préf}(X) \setminus \{\varepsilon\}$ pour lequel le naturel

$$k = \min\{\ell : \mathit{Suivant}(f^\ell(u)) \not\subseteq \mathit{Suivant}(u)\}$$

est défini, et non définie partout ailleurs. Alors la structure constituée :

– de l'automate $\mathcal{A}(X)$, comme pour l'implantation $\mathcal{D}_{\mathrm{s}}(X)$;
– de l'état initial de $\mathcal{A}(X)$ comme successeur par défaut pour les états dont l'image par f' n'est pas définie ;
– de la fonction de suppléance f' ;

est une implantation de l'automate-dictionnaire $\mathcal{D}(X)$ dans le modèle comparaisons.

Cela dit, quand bien même f' est substituée à f pour la phase de localisation, l'amélioration n'est pas quantifiable au moyen des mesures que nous utilisons. En particulier, le délai reste proportionnel au maximum des longueurs des mots du dictionnaire dans le pire des cas. C'est ce que montre l'exemple suivant.

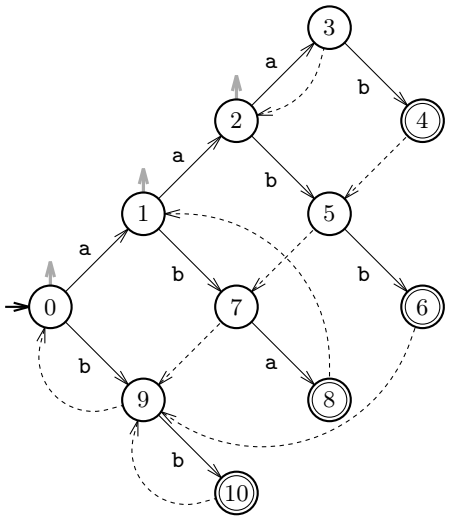


Figure 2.7 Dans le pire des cas, le délai de l'algorithme LA-DÉTERMINISTE-PAR-SUPPLÉANCE est proportionnel au maximum des longueurs des mots du dictionnaire X. Ce constat reste valable quand bien même l'on considère la version optimisée f' de la fonction de suppléance f de l'implantation $\mathcal{D}_{\mathrm{S}}(X)$ de $\mathcal{D}(X)$. Comme par exemple lorsque $X = L(4) = \{\mathtt{aaab}, \mathtt{aabb}, \mathtt{aba}, \mathtt{bb}\}$ et que `aaabc` est un facteur du texte : quatre appels successifs à f' sont effectués, l'état courant prenant successivement les valeurs 4, 5, 7, 9 puis 0.

Supposons que l'alphabet A contienne (au moins) les trois lettres `a`, `b` et `c`. Soit $L(m)$ le langage défini pour tout entier $m \geq 1$ par :

$$\begin{aligned} L(m) = &\{\mathtt{a}^{m-1}\mathtt{b}\} \\ &\cup \{\mathtt{a}^{2k-1}\mathtt{ba} : 1 \leq k < \lceil m/2 \rceil\} \\ &\cup \{\mathtt{a}^{2k}\mathtt{bb} : 0 \leq k < \lfloor m/2 \rfloor\} \ . \end{aligned}$$

Pour un certain entier $m \geq 1$, posons $X = L(m)$. Alors, si le mot $\mathtt{a}^{m-1}\mathtt{bc}$ est un facteur du texte, m appels à la fonction de suppléance (ligne 2 de la fonction CIBLE-PAR-SUPPLÉANCE) sont effectués lorsque `c` est la lettre courante. Et l'entier m est exactement la longueur du mot $\mathtt{a}^{m-1}\mathtt{b}$, un plus long mot de X. (Voir l'illustration proposée figure 2.7.)

2.4 Implantation avec successeur par défaut

Dans cette section, nous étudions l'implantation $\mathcal{D}_{\mathrm{D}}(X)$ obtenue de l'automate $\mathcal{D}(X)$ en supprimant toutes les flèches dont la cible est l'état initial et en ajoutant l'état initial comme successeur par défaut (voir l'illustration proposée figure 2.8). Cette implantation réduite de $\mathcal{D}(X)$ dans le modèle comparaisons s'avère particulièrement intéressante, tant

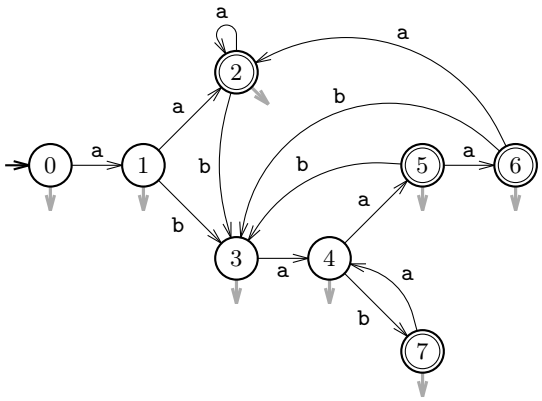


Figure 2.8 Implantation $\mathcal{D}_{\mathrm{D}}(X)$ de l'automate-dictionnaire $\mathcal{D}(X)$ lorsque $X = \{\mathtt{aa}, \mathtt{abaaa}, \mathtt{abab}\}$. Tous les états ont comme successeur par défaut l'état initial. Cette représentation est à comparer à celle donnée figure 2.2.

au niveau de son initialisation qu'à celui de son utilisation, lorsque les ensembles des successeurs étiquetés sont ordonnés selon l'alphabet.

Le plan suivi est le suivant : nous commençons par montrer que la taille de l'implantation $\mathcal{D}_{\mathrm{D}}(X)$ est à la fois raisonnable et indépendante de la taille de l'alphabet ; nous nous intéressons ensuite à la construction de cette implantation particulière ; puis nous exprimons les complexités de la phase de localisation ; nous comparons enfin entre elles les diverses implantations de $\mathcal{D}(X)$ exposées jusqu'ici.

Taille de l'implantation

Dans l'implantation $\mathcal{D}_{\mathrm{D}}(X)$, appelons ***flèche avant*** une flèche de la forme (u, a, ua) – autrement dit une flèche de l'arbre $\mathcal{A}(X)$ – et ***flèche arrière*** toute autre flèche. L'automate représenté figure 2.8 possède ainsi sept flèches avant et six flèches arrière. Plus généralement maintenant, nous avons le résultat suivant.

Proposition 2.19
Le nombre de flèches avant dans $\mathcal{D}_{\mathrm{D}}(X)$ est inférieur à $|X|$, et son nombre de flèches arrière est inférieur à $|X| \times \operatorname{card} X$.

La preuve de ce résultat sera établie après celle du lemme qui suit. Auparavant, appelons décalage d'une flèche (u, a, u') dans $\mathcal{D}_{\mathrm{D}}(X)$ l'entier $|ua| - |u'|$, et disons d'une flèche qu'elle est dirigée de $x \in X$ vers $x' \in X$ si sa source est un préfixe de x et sa cible un préfixe de x'.

Lemme 2.20
Si x et x' sont des mots de X, alors les décalages de flèches arrière distinctes dirigées de x vers x' sont distincts.

Preuve Par l'absurde. Supposons l'existence de deux flèches arrière distinctes (u, a, u') et (v, b, v') dirigées de x vers x' de décalages identiques, c'est-à-dire telles que :

$$|ua| - |u'| = |vb| - |v'| \ . \tag{2.1}$$

Si l'on suppose $u = v$, on obtient, d'après (2.1), $|u'| = |v'|$, puis, comme u' et v' sont des préfixes de x', $u' = v'$. D'autre part, comme les deux flèches sont des flèches arrière, elle n'entrent pas dans l'état initial ; donc $x'[|u'| - 1] = a$ et $x'[|v'| - 1] = b$. Il s'ensuit que $a = b$. Ce qui contredit l'hypothèse de deux flèches distinctes.

On peut dès lors supposer sans perte de généralité que $v \prec_{préf} u$. Pour des questions de longueur, il vient de (2.1) que $|v'| < |u'|$, puis, comme u' et v' sont des préfixes de x', $v' \prec_{préf} u'$. Maintenant, puisque $u' \preceq_{suff} ua$, on calcule à l'aide de (2.1) : $x'[|v'|-1] = x[|v'|-1+|ua|-|u'|] = x[|v|]$. Ce qui est impossible : d'une part $x'[|v'|-1] = b$, la flèche (v, b, v') n'entrant pas dans l'état initial puisqu'il s'agit d'une flèche arrière ; d'autre part $x[|v|] \neq b$, sans quoi cette même flèche serait une flèche avant. D'où le résultat. ∎

Preuve de la proposition 2.19 Dans $\mathcal{D}_{\mathrm{D}}(X)$, chaque flèche avant s'identifie à sa cible, laquelle appartient à $Préf(X) \setminus \{\varepsilon\}$. La première borne annoncée s'en déduit.

Si x et x' sont deux mots de X, les décalages des éventuelles flèches arrière dirigées de x vers x' sont distincts d'après le lemme 2.20 et sont compris entre 1 et $|x|$. Il s'ensuit que le nombre de flèches arrière dirigées de x vers x' est majoré par $|x|$. Le nombre total de flèches arrière de l'implantation $\mathcal{D}_{\mathrm{D}}(X)$ est donc majoré par $\sum_{x \in X} |x| \times \operatorname{card} X$, ce qui établit la seconde borne. ∎

Nous venons d'établir des bornes sur le nombre total de flèches dans $\mathcal{D}_{\mathrm{D}}(X)$. Remarquons maintenant que localement, en chacun des états, on a le résultat suivant.

Lemme 2.21
Le maximum des degrés sortant des états dans $\mathcal{D}_{\mathrm{D}}(X)$ *est inférieur à* $\operatorname{card} alph(X)$.

Preuve Cela résulte du fait que les flèches issues d'un même état sont étiquetées par des lettres de $alph(X)$. ∎

Théorème 2.22
L'implantation $\mathcal{D}_{\mathrm{D}}(X)$ *de l'automate-dictionnaire* $\mathcal{D}(X)$ *est de taille* $O(|X| \times \min\{\operatorname{card} alph(X), \operatorname{card} X\})$.

Preuve L'espace total nécessaire pour mémoriser l'automate $\mathcal{D}(X)$ sous la forme $\mathcal{D}_{\mathrm{D}}(X)$ est linéaire en le nombre d'états et le nombre de

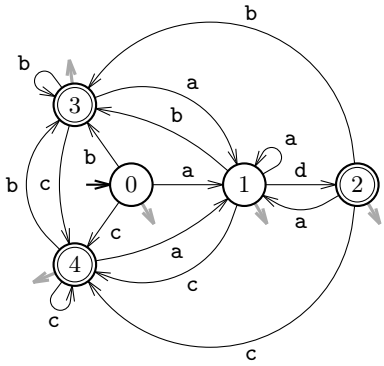


Figure 2.9 Une implantation $\mathcal{D}_{\mathrm{D}}(X)$ avec un maximum de flèches à $\operatorname{card} X$ et $|X|$ fixés, soit $(|X|+1) \times \operatorname{card} X$ flèches. Ici $X = \{\mathtt{ad}, \mathtt{b}, \mathtt{c}\}$.

flèches dans $\mathcal{D}_{\mathrm{D}}(X)$. Le premier de ces nombres est inférieur à $|X|+1$. Quant au second, il est inférieur à $|X| \times (1 + \operatorname{card} X)$ d'après la proposition 2.19. D'où la borne $O(|X| \times \operatorname{card} X)$. Quant à la borne $O(|X| \times \operatorname{card} \mathit{alph}(X))$, c'est une conséquence immédiate du lemme 2.21. ∎

Lorsque le nombre et la somme des longueurs des mots de X sont fixés, la majoration du nombre de flèches donnée dans la proposition 2.19 peut être atteinte. C'est ce que suggère l'exemple de la figure 2.9, et ce qu'établit la proposition qui suit pour le cas général.

Proposition 2.23
Pour tout entier naturel non nul $k < \operatorname{card} A$, pour tout entier $m \geq k$, il existe un dictionnaire X tel que $\operatorname{card} X = k$ et $|X| = m$, pour lequel le nombre de flèches dans $\mathcal{D}_{\mathrm{D}}(X)$ est égal à $m \times (k+1)$.

Preuve Choisissons $k+1$ lettres deux à deux distinctes $a_0, a_1, \ldots, a_k$ dans A, puis considérons le dictionnaire X formé du mot $a_0 {a_k}^{m-k}$ d'une part, et des mots $a_1, a_2, \ldots, a_{k-1}$ d'autre part. Alors, dans l'implantation $\mathcal{D}_{\mathrm{D}}(X)$, il sort exactement k flèches arrière étiquetées chacune par l'une des lettres $a_0, a_1, \ldots, a_{k-1}$ de chacun des m états différents de l'état initial. Comme l'implantation possède également m flèches avant, elle possède en tout $k \times m + m = m \times (k+1)$ flèches. ∎

Construction de l'implantation

Pour construire l'implantation $\mathcal{D}_{\mathrm{D}}(X)$, on reprend le code de la fonction ALP-COMPLET qui produit l'automate $\mathcal{D}(X)$. On le modifie (lignes 4–8 et 17 auxquelles correspondent ici les lignes 4–5 et 14–15) afin de ne pas générer les flèches de $\mathcal{D}(X)$ qui ont pour cible l'état initial.

```
ALP-PAR-DÉFAUT(X)
 1  M ← ARBRE(X)
 2  q_0 ← initial[M]
 3  F ← FILE-VIDE()
 4  pour chaque couple (a, q) ∈ Succ[q_0] faire
 5        ENFILER(F, (q, q_0))
 6  tantque non FILE-EST-VIDE(F) faire
 7        (p, r) ← DÉFILEMENT(F)
 8        si terminal[r] alors
 9              terminal[p] ← VRAI
10        pour chaque lettre a ∈ A faire
11              q ← CIBLE(p, a)
12              s ← CIBLE-PAR-DÉFAUT(r, a)
13              si q = NIL alors
14                    si s ≠ q_0 alors
15                          Succ[p] ← Succ[p] ∪ {(a, s)}
16              sinon ENFILER(F, (q, s))
17  retourner M
```

La ligne 12 fait appel à la fonction CIBLE-PAR-DÉFAUT qui simule les transitions vers l'état initial dans la partie déjà construite de l'implantation. Le code de cette fonction est précisé ci-dessous. L'objet automate M est supposé être global.

```
CIBLE-PAR-DÉFAUT(p, a)
1  si il existe un état q tel que (a, q) ∈ Succ[p] alors
2        retourner q
3  sinon retourner initial[M]
```

Proposition 2.24
L'opération ALP-PAR-DÉFAUT(X) *produit* $\mathcal{D}_{\mathrm{D}}(X)$*, implantation avec successeur par défaut de* $\mathcal{D}(X)$*.*

Preuve Immédiate d'après la proposition 2.8. ∎

Le théorème qui suit établit les complexités de la fonction ALP-PAR-DÉFAUT. Il précise en particulier que le fait de maintenir ordonnés les ensembles des successeurs étiquetés selon l'alphabet en assure une exécution efficace.

Théorème 2.25
Si, durant l'exécution de l'opération ALP-PAR-DÉFAUT(X)*, on prend soin de considérer les ensembles des successeurs étiquetés comme des listes ordonnées selon l'alphabet, le temps d'exécution de cette opération est du même ordre que la taille de l'implantation* $\mathcal{D}_{\mathrm{D}}(X)$ *de l'automate* $\mathcal{D}(X)$ *qu'elle produit, soit* $O(|X| \times \min\{\operatorname{card} \mathit{alph}(X), \operatorname{card} X\})$*. L'espace mémoire supplémentaire nécessaire à l'exécution est* $O(\operatorname{card} X)$*.*

Preuve Durant la construction de l'arbre $\mathcal{A}(X)$, chaque appel à la fonction CIBLE et chaque ajout dans une liste de successeurs étiquetés a un cout au plus linéaire en le maximum des degrés sortants des états. Comme le nombre de chacune de ces deux opérations est au plus égal à $|X|$, il se déduit du lemme 2.14 que le cout total de la ligne 1 est $O(|X| \times \min\{\operatorname{card} \mathit{alph}(X), \operatorname{card} X\})$.

Ensuite, les listes de successeurs étiquetés de p (comme dans $\mathcal{A}(X)$) et de r (comme dans $\mathcal{D}_{\text{D}}(X)$) étant ordonnées selon l'alphabet, la boucle **pour** des lignes 10–16 s'implante comme une opération de fusion sans doublon des deux listes. Elle se réalise donc en temps linéaire en la longueur de la liste résultante de p (comme dans $\mathcal{D}_{\text{D}}(X)$ maintenant). Il vient alors, par application du théorème 2.22, que le temps d'exécution des lignes 6–16 est également $O(|X| \times \min\{\operatorname{card} \mathit{alph}(X), \operatorname{card} X\})$.

Le temps total de l'exécution de la fonction ALP-PAR-DÉFAUT s'en déduit. Quant à la justification de la taille de l'espace nécessaire au calcul, elle a déjà été donnée lors de la preuve de la proposition 2.9. ∎

Phase de localisation

Pour localiser les occurrences des mots du dictionnaire X dans le texte y à partir de l'implantation $\mathcal{D}_{\text{D}}(X)$, on utilise, comme pour l'automate $\mathcal{D}(X)$, l'algorithme LA-DÉTERMINISTE. On modifie toutefois son code puisqu'il s'agit ici de simuler les transitions vers l'état initial. La ligne 3, qui correspond à la ligne 4 dans le code ci-dessous, fait désormais appel à la fonction CIBLE-PAR-DÉFAUT en lieu et place de la fonction CIBLE. Le code de l'algorithme de localisation associé suit.

```
LA-DÉTERMINISTE-PAR-DÉFAUT(X, y)
1  M ← ALP-PAR-DÉFAUT(X)
2  r ← initial[M]
3  pour chaque lettre a de y, séquentiellement faire
4        r ← CIBLE-PAR-DÉFAUT(r, a)
5        SIGNALER-SI(terminal[r])
```

Lemme 2.26
Le nombre de comparaisons entre lettres effectuées lors de la phase de localisation de l'opération LA-DÉTERMINISTE-PAR-DÉFAUT(X, y) *est inférieur à* $(1 + \operatorname{card} X) \times |y| - 1$ *lorsque le texte* y *est non vide, quel que soit l'ordre dans lequel sont examinés les éléments des ensembles de successeurs étiquetés lors du calcul des transitions (chacun des éléments étant inspecté au plus une fois lors d'un calcul).*

Preuve Lors du calcul d'une transition, le résultat d'une comparaison entre la lettre courante du texte avec la lettre de l'élément courant de l'ensemble des successeurs étiquetés courant est, soit positif, soit négatif. Dans le second cas, le calcul se poursuit. Il peut s'arrêter dans le premier cas, mais cela n'a pas d'importance vu le résultat que l'on veut établir.

Remarquons que puisque l'on veut obtenir une majoration, on peut toujours supposer – quitte à augmenter l'alphabet d'une lettre – que la dernière lettre de y n'apparait dans aucun des mots de X. Le nombre de comparaisons positives est majoré par $|y| - 1$ dans un tel cas. Nous montrons plus loin que le nombre de comparaisons négatives est majoré par $\operatorname{card} X \times |y|$, ce qui achèvera la preuve.

Notons pour commencer que si (a, u) est un élément de l'ensemble des successeurs étiquetés courant inspecté avec un résultat négatif en position i sur le texte y (soit $y[i] \neq a$), la valeur $i - |u| + 1$ est également une position sur y, c'est-à-dire qu'elle satisfait la double inégalité

$$0 \leq i - |u| + 1 \leq |y| - 1 \ , \tag{2.2}$$

puisque $u \neq \varepsilon$, $ua^{-1} \preceq_{préf} y[0\,..\,i-1]$, et $i < |y|$.

Supposons maintenant l'existence de deux éléments (a, u) et (b, v) inspectés négativement aux positions respectives i et j. Alors, si l'on a

$$i - |u| + 1 = j - |v| + 1 \ , \tag{2.3}$$

u et v sont des préfixes de deux mots distincts de X. Afin de prouver cette assertion, on envisage successivement les deux éventualités $i = j$ et $i < j$ (l'éventualité $i > j$ étant symétrique de la seconde).

Première éventualité : $i = j$. De (2.3) il vient alors que $|u| = |v|$. Comme a et b sont les dernières lettres de u et v, respectivement, on a nécessairement $a \neq b$. Cela montre que $u \neq v$, puis que l'assertion est satisfaite dans ce cas.

Seconde éventualité : $i < j$. De (2.3) il s'ensuit que $|u| < |v|$. Pour montrer que l'assertion est satisfaite, il suffit de montrer que u n'est pas un préfixe de v. Par l'absurde, supposons un instant que $u \prec_{préf} v$. On a alors :

$$\begin{array}{lcll} y[i] & = & v[i - j + |v| - 1] & (\text{car } v[0\,..\,|v|-2] \preceq_{suff} y[0\,..\,j-1]) \\ & = & v[|u| - 1] & (\text{d'après (2.3)}) \\ & = & a & (\text{car } u \prec_{préf} v) \ . \end{array}$$

D'où la contradiction avec l'hypothèse d'une comparaison négative en i pour l'élément (a, u), ce qui achève la preuve de l'assertion.

Autrement dit, pour chaque mot $x \in X$, les quantités $i - |u| + 1$ avec $u \preceq_{préf} x$ qui sont associées aux comparaisons négatives sont deux à deux distinctes. Il vient donc, à l'aide de (2.2), qu'au plus $|y|$ comparaisons négatives sont associées à chaque mot de X. Au total, le nombre de comparaisons négatives est donc bien majoré par $\operatorname{card} X \times |y|$, comme annoncé. ∎

Théorème 2.27
L'opération LA-DÉTERMINISTE-PAR-DÉFAUT(X, y) *a une phase de localisation qui s'exécute en temps* $O(|y| \times \min\{\log \operatorname{card} alph(X), \operatorname{card} X\})$ *et un délai qui est* $O(\log \operatorname{card} alph(X))$.

Preuve La borne pour le délai est une conséquence du lemme 2.21 et de la proposition 1.16, les ensembles des successeurs étiquetés pouvant être construits ordonnés selon l'alphabet, sans surcout supplémentaire. La borne $O(|y| \times \log \operatorname{card} \mathit{alph}(X))$ pour le temps de localisation s'en déduit. La borne $O(|y| \times \operatorname{card} X)$ provient du lemme 2.26. ∎

À l'appui du résultat du lemme 2.26, nous montrons ci-dessous que la borne du nombre de comparaisons est atteinte à $\operatorname{card} X$ et $|y|$ fixés.

Proposition 2.28
Pour tout entier naturel non nul $k < \operatorname{card} A$, pour tout entier naturel non nul n, il existe un dictionnaire X de k mots et un texte y de longueur n tels que le nombre de comparaisons entre lettres effectuées lors de la phase de localisation de l'opération LA-DÉTERMINISTE-PAR-DÉFAUT(X, y) *soit égal à $(k+1) \times n - 1$, l'ordre dans lequel les ensembles des successeurs étiquetés sont examinés étant quelconque.*

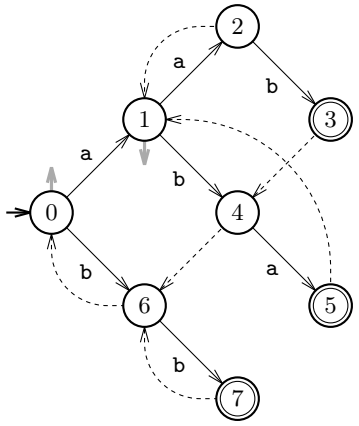
Preuve Choisissons un entier $m \geq k + 1$, considérons le dictionnaire X défini lors de la preuve de la proposition 2.23, puis le texte $y = {a_0}^n$. Alors, durant la localisation, la première lettre de y est comparée aux lettres $a_0, a_1, \ldots, a_{k-1}$, lesquelles sont les étiquettes des flèches sortant de l'état initial ε ; et les autres lettres de y sont comparées aux lettres $a_0, a_1, \ldots, a_k$, étiquettes des flèches sortant de l'état a_0. Dans le pire des cas, k comparaisons sont effectuées sur la première lettre de y et $k + 1$ sur les suivantes. Soit $(k + 1) \times n - 1$ comparaisons au total. ∎

Un exemple qui illustre le pire des cas qui vient d'être évoqué dans la preuve : pour $k = 3$ et $m = 4$, on prend $X = \{\mathtt{ad}, \mathtt{b}, \mathtt{c}\}$ et $y \in \{\mathtt{a}\}^*$; l'implantation $\mathcal{D}_{\text{D}}(X)$ est montrée figure 2.9.

Challenge d'implantations

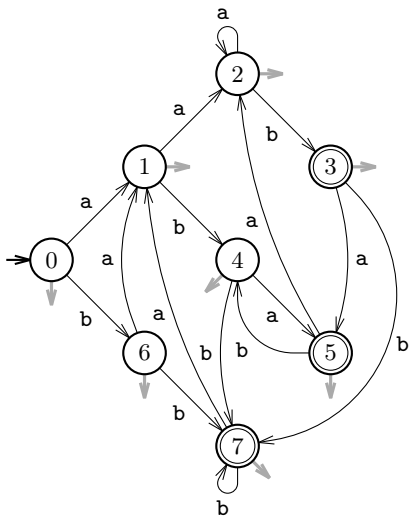
Les implantations $\mathcal{D}_{\text{S}}(X)$ et $\mathcal{D}_{\text{D}}(X)$ sont deux implantations concurrentes de l'automate-dictionnaire $\mathcal{D}(X)$ dans le modèle comparaisons. Les résultats établis dans cette section et la précédente plaident plutôt en faveur de la première : ordre de la taille de l'implantation, celui de sa construction et l'ordre du temps de la phase de localisation (théorèmes 2.12, 2.18 et 2.15 contre théorèmes 2.22, 2.25 et 2.27). Seul l'ordre du délai compté pour les phases de localisation peut être mis à l'actif de la seconde implantation (théorèmes 2.15 et 2.27 encore une fois). Il est toutefois possible, en renonçant à ce résultat sur le délai, d'améliorer l'implantation $\mathcal{D}_{\text{D}}(X)$ de telle sorte que jamais plus de comparaisons entre lettres ne soient effectuées lors de la phase de localisation qu'avec $\mathcal{D}_{\text{S}}(X)$ (version originelle ou version optimisée), sans augmentation de l'ordre des autres complexités. C'est ce qu'expriment le paragraphe suivant et la figure 2.10.

(a)



p	$Succ[p], Succ[suppléant[p]], \ldots$
0	$\{(\mathtt{a},1),(\mathtt{b},6)\}$
1	$\{(\mathtt{a},2),(\mathtt{b},4)\}$
2	$\{(\mathtt{b},3)\}$, $\{(\mathtt{a},2),(\mathtt{b},4)\}$
3	$\emptyset$, $\{(\mathtt{a},5)\}$, $\{(\mathtt{b},7)\}$, $\{(\mathtt{a},1),(\mathtt{b},6)\}$
4	$\{(\mathtt{a},5)\}$, $\{(\mathtt{b},7)\}$, $\{(\mathtt{a},1),(\mathtt{b},6)\}$
5	$\emptyset$, $\{(\mathtt{a},2),(\mathtt{b},4)\}$
6	$\{(\mathtt{b},7)\}$, $\{(\mathtt{a},1),(\mathtt{b},6)\}$
7	$\emptyset$, $\{(\mathtt{b},7)\}$, $\{(\mathtt{a},1),(\mathtt{b},6)\}$

(b)



p	partition de $Succ[p]$
0	$\{(\mathtt{a},1),(\mathtt{b},6)\}$
1	$\{(\mathtt{a},2),(\mathtt{b},4)\}$
2	$\{(\mathtt{b},3)\}$, $\{(\mathtt{a},2)\}$
3	$\{(\mathtt{a},5)\}$, $\{(\mathtt{b},7)\}$
4	$\{(\mathtt{a},5)\}$, $\{(\mathtt{b},7)\}$
5	$\{(\mathtt{a},2),(\mathtt{b},4)\}$
6	$\{(\mathtt{b},7)\}$, $\{(\mathtt{a},1)\}$
7	$\{(\mathtt{b},7)\}$, $\{(\mathtt{a},1)\}$

Figure 2.10 Deux optimisations pour les implantations $\mathcal{D}_{\mathrm{S}}(X)$ et $\mathcal{D}_{\mathrm{D}}(X)$ de l'automate-dictionnaire $\mathcal{D}(X)$. Ici avec $X = \{\mathtt{aab}, \mathtt{aba}, \mathtt{bb}\}$ (soit $X = L(3)$ avec la notation de la fin de la section 2.3). **(a)** Implantation $\mathcal{D}_{\mathrm{S}}(X)$ avec la version optimisée f' de la fonction de suppléance f. En marge, la suite des ensembles de successeurs étiquetés qui peuvent être examinés pour le calcul d'une transition à partir de l'état courant. L'examen prend fin dès que la lettre courante apparait dans l'un des éléments appartenant à l'ensemble courant ou lorsque la liste est épuisée. **(b)** Pour n'effectuer jamais plus de comparaisons que l'implantation $\mathcal{D}_{\mathrm{S}}(X)$, l'implantation $\mathcal{D}_{\mathrm{D}}(X)$ peut considérer elle aussi une suite d'ensembles des successeurs étiquetés : pour chaque état, la partition de l'ensemble de ses successeurs étiquetés obtenue en classant les successeurs étiquetés par niveaux décroissants dans l'arbre $\mathcal{A}(X)$ (les états de niveaux identiques sont situés sur une même verticale sur le dessin). Une telle optimisation s'obtient sans altération des ordres de grandeur des complexités pour la construction de l'implantation.

L'inconvénient avec l'implantation $\mathcal{D}_{\text{D}}(X)$, c'est que, lors du calcul des transitions durant la phase de localisation, l'ensemble des successeurs de chaque état courant peut être considéré dans son entier. Alors que s'il est considéré par parties (disjointes bien évidemment, pour ne pas faire deux fois la même comparaison), la partie avec les cibles de plus grand niveau d'abord, celle avec les cibles de niveau immédiatement inférieur ensuite, et ainsi de suite, les parties considérées sont toutes de cardinal au plus égal à leurs homologues dans l'implantation $\mathcal{D}_{\text{S}}(X)$. Il s'ensuit qu'avec cet examen particulier des ensembles des successeurs, le nombre de comparaisons entre lettres comptées pour $\mathcal{D}_{\text{D}}(X)$ est au plus égal à celui des comparaisons entre lettres comptées pour $\mathcal{D}_{\text{S}}(X)$. Pour construire la partition dans le même temps et le même espace que celui requis pour la version originelle de $\mathcal{D}_{\text{D}}(X)$, il suffit par exemple de maintenir, pour chaque état p : une liste des successeurs étiquetés, disons $S_0(p)$, ordonnée selon l'alphabet ; une partition des successeurs étiquetés, disons $S_1(p)$, classée par niveaux décroissants ; des pointeurs de chaque élément de $S_0(p)$ vers son correspondant dans $S_1(p)$. Les pointeurs permettent de supprimer les doublons dans une copie de $S_1(r)$ lors de la fusion de $S_0(r)$ et $S_0(p)$ (pour donner $S_0(p)$, voir lignes 10–16 de la fonction ALP-PAR-DÉFAUT) en temps constant. La partition $S_1(p)$ s'obtient ensuite en concaténant à la suite de sa valeur originelle (l'ensemble des successeurs étiquetés de p dans $\mathcal{A}(X)$) la copie éventuellement modifiée de $S_1(r)$ (les éléments de $S_1(r)$ étant de niveaux strictement inférieurs à celui des précédents).

Pour les deux implantations maintenant, et en concurrence avec une localisation de type non déterministe à partir de l'arbre du dictionnaire (ou d'un automate encore plus rudimentaire qui reconnait également X, comme celui évoqué section 1.5), le facteur de la complexité en temps qui multiplie la longueur du texte est lié au nombre de mots dans le dictionnaire, alors qu'il présente au moins un facteur lié à la somme des longueurs des mots du dictionnaire dans le second cas (dû à la gestion des ensembles d'états), ce qui montre l'intérêt des deux implantations réduites lorsque $|X|$ est grand.

Si l'on considère l'alphabet comme constant, le temps de la construction et l'espace nécessaire pour la mémorisation des implantations (avec fonction de suppléance, état initial comme successeur par défaut ou même par matrice de transition) est linéaire en la somme des longueurs des mots et le temps de la phase de localisation linéaire en la longueur du texte.

Ajoutons encore que les implantations $\mathcal{D}_{\text{S}}(X)$ et $\mathcal{D}_{\text{D}}(X)$ peuvent être réalisées avec un espace $O(|X| \times \operatorname{card} A)$ mais avec des temps de construction dépendant uniquement de X en utilisant une technique d'implantation des fonction partielles (voir section 1.4 et exercice 1.15) ; le temps de la phase de localisation est alors également linéaire en la longueur du texte.

2.5 Localisation d'un mot

Dans toute la suite du chapitre (sections 2.5 à 2.7), nous étudions le cas particulier où le dictionnaire X n'est constitué que d'un seul mot. Nous reprenons, en les adaptant, certains des résultats établis précédemment. Nous les complétons en donnant notamment :

- des méthodes de construction de l'automate-dictionnaire et de ses implantations mieux adaptées au cas particulier envisagé ;
- des bornes précises du délai pour les implantations réduites (implantation avec fonction de suppléance section 2.6 et implantation avec l'état initial comme successeur par défaut section 2.7).

Dans la présente section, nous revenons essentiellement sur la construction de l'automate-dictionnaire en montrant que celle-ci peut s'effectuer séquentiellement sur le mot considéré. En outre, elle produit sans aucune modification un automate minimal.

Pour toute la suite du chapitre, nous considérons un mot x non vide, et nous posons $X = \{x\}$.

Commençons par réécrire les fonctions h et f avec la notion de bord. Pour tout couple $(u, a) \in A^* \times A$ on a :

$$h(ua) = \begin{cases} ua & \text{si } ua \preceq_{préf} x \ , \\ Bord(ua) & \text{sinon} \ . \end{cases}$$

Et pour tout mot $u \in A^+$, on a :

$$f(u) = Bord(u) \ .$$

Pour l'égalité concernant la fonction f, il s'agit d'une conséquence de sa définition et de celle de la fonction *Bord*. Pour l'égalité concernant la fonction h, c'est une conséquence des lemmes 2.5 et 2.6, et de la réécriture de f. Quant à l'automate $\mathcal{D}(\{x\})$ de la section 2.2, il est défini par :

$$\mathcal{D}(\{x\}) = (Préf(x), \varepsilon, \{x\}, F_x) \tag{2.4}$$

avec :

$$F_x = \{(u, a, ua) : u \in A^*, a \in A, ua \preceq_{préf} x\} \cup \\ \{(u, a, Bord(ua)) : u \in A^*, u \preceq_{préf} x, a \in A, ua \not\preceq_{préf} x\} \ .$$

Remarquons au passage – on en a besoin pour établir simplement certains des résultats qui suivent – que l'identité (2.4) peut être étendue au mot vide, l'automate $(Préf(\varepsilon), \varepsilon, \{\varepsilon\}, F_\varepsilon)$ reconnaissant le mot vide.

La construction de l'automate $\mathcal{D}(\{x\})$ peut se faire de façon séquentielle sur x, ce qui signifie qu'elle n'a recours ni à la construction préalable de l'automate $\mathcal{A}(\{x\})$ comme dans la section 2.2, ni à la fonction *Bord*. C'est ce que suggère le résultat suivant.

Proposition 2.29
On a $F_\varepsilon = \{(\varepsilon, b, \varepsilon) : b \in A\}$. *De plus, pour tout couple* $(u, a) \in A^* \times A$, *on a* $F_{ua} = F' \cup F''$ *avec :*

$F' = (F_u \setminus \{(u, a, \mathit{Bord}(ua))\}) \cup \{(u, a, ua)\}$

et :

$F'' = \{(ua, b, v) : (\mathit{Bord}(ua), b, v) \in F'\}$.

Preuve La propriété est clairement vérifiée pour F_ε. Ensuite, soient u, a, F' et F'' comme dans l'énoncé de la proposition.

Chaque flèche dans F_{ua} qui sort d'un état de longueur inférieure à $|u|$ est dans F'. La réciproque est également vraie.

Il reste à montrer que toute flèche dans F_{ua} sortant de l'état ua appartient à F'', et réciproquement. Ce qui revient à montrer que, pour toute lettre $b \in A$, les cibles v et v' des flèches (ua, b, v) et $(\mathit{Bord}(ua), b, v')$ sont identiques. Or, par définition de $\mathcal{D}(\{ua\})$, on a $v = \mathit{Bord}(uab)$; et si $\mathit{Bord}(ua)b \preceq_{préf} ua$, $v' = \mathit{Bord}(ua)b$, et $v' = \mathit{Bord}(\mathit{Bord}(ua)b)$ sinon. D'où l'on déduit, par application du lemme 1.22, que $v = v'$. ∎

Il est agréable d'interpréter « visuellement » le résultat précédent : on obtient $\mathcal{D}(\{ua\})$ de $\mathcal{D}(\{u\})$ en « dépliant » la flèche de source u et d'étiquette a ; la cible est dupliquée avec ses flèches sortantes. Une illustration est proposée figure 2.11.

Suit le code de la fonction ALU-COMPLET qui construit puis retourne l'automate $\mathcal{D}(\{x\})$. Les trois premières lettres de l'identificateur de la fonction signifient « automate de localisation d'un seul mot ».

```
ALU-COMPLET(x)
 1  M ← NOUVEL-AUTOMATE()
 2  q0 ← initial[M]
 3  pour chaque lettre b ∈ A faire
 4        Succ[q0] ← Succ[q0] ∪ {(b, q0)}
 5  t ← q0
 6  pour chaque lettre a de x, séquentiellement faire
 7        p ← NOUVEL-ÉTAT()
 8        r ← CIBLE(t, a)
 9        Succ[t] ← Succ[t] \ {(a, r)}
10        Succ[t] ← Succ[t] ∪ {(a, p)}
11        Succ[p] ← Succ[r]
12        t ← p
13  terminal[t] ← VRAI
14  retourner M
```

Une propriété invariante de la boucle **pour** des lignes 6–12 est que la structure construite coïncide avec l'automate-dictionnaire du préfixe courant du mot x, hormis en ce qui concerne l'état terminal. Ce détail est réglé au final ligne 13.

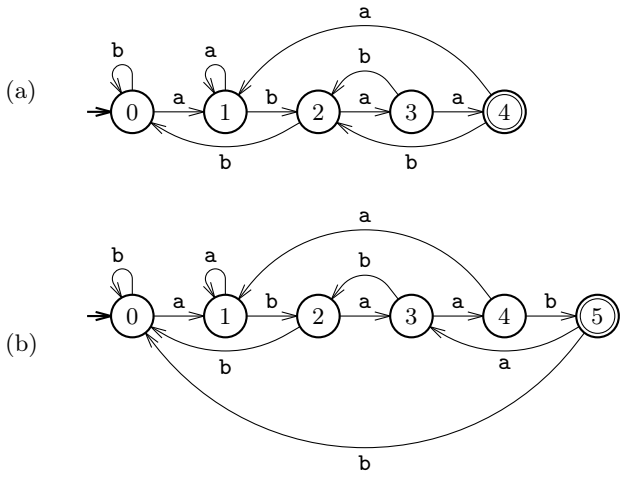


Figure 2.11 L'automate $\mathcal{D}(\{ua\})$, $u \in A^*$ et $a \in A$, s'obtient de l'automate $\mathcal{D}(\{u\})$ en « dépliant » la flèche $(u, a, Bord(ua))$ dans ce dernier. Par exemple, de l'automate $\mathcal{D}(\{\texttt{abaa}\})$ (a), on obtient l'automate $\mathcal{D}(\{\texttt{abaab}\})$ (b) en créant un nouvel état, 5 en l'occurrence, en « redirigeant » la flèche $(4, \texttt{b}, 2)$ vers l'état 5, puis en donnant à l'état 5 le même ensemble des successeurs étiquetés que celui de l'état 2 une fois l'opération effectuée. L'ordre d'exécution de ces opérations ne peut être changé.

Proposition 2.30
L'opération ALU-COMPLET(x) *produit* $\mathcal{D}(\{x\})$.

Preuve Il suffit de vérifier que le code applique correctement la proposition 2.29. ∎

2.6 Localisation d'un mot et fonction de suppléance

Nous étudions les implantations de l'automate-dictionnaire $\mathcal{D}(\{x\})$ avec la fonction de suppléance f et sa version optimisée f' introduites section 2.3. Nous commençons par établir quelques propriétés satisfaites par f'. Ces propriétés montrent que la fonction f' peut être directement utilisée lors de la phase de construction de l'implantation $\mathcal{D}_{\mathrm{s}}(\{x\})$ – alors qu'elle se déduit après coup de la fonction f dans le cas général d'un dictionnaire quelconque. Nous abordons ensuite précisément la phase de construction, pour en venir finalement à l'analyse de la phase de localisation. Dans cette dernière subdivision, nous montrons en particulier que le délai est logarithmique en la longueur du mot cherché lorsque la fonction de suppléance f' est utilisée.

Propriétés de la fonction de suppléance optimisée

La fonction $f'\colon Préf(x) \to Préf(x)$ – tout comme, plus haut, les fonctions h et f – se réécrit plus simplement avec la notion de bord. Elle se reformule en :

$f'(x) = Bord(x)$

en x, puis en :

$f'(u) = v$

en tout $u \prec_{préf} x$ pour lequel il existe un mot v tel que :

$v =$ le plus long bord de u avec $x[|u|] \neq x[|v|]$,

et est non définie partout ailleurs.

De cette reformulation, on déduit les deux propriétés qui suivent.

Lemme 2.31
Pour tout mot $u \prec_{préf} x$ pour lequel $f'(u)$ est défini, on a :

$$f'(u) = \begin{cases} Bord(u) & si\ x[|u|] \neq x[|Bord(u)|] \ , \\ f'(Bord(u)) & sinon\ . \end{cases}$$

Preuve Le mot $f'(u)$ est un bord de u. Si le plus long bord de u ne convient pas, parce que $x[|u|] = x[|Bord(u)|]$, $f'(u)$ est exactement $f'(Bord(u))$, lequel mot assure que les deux lettres $x[|f'(Bord(u))|]$ et $x[|Bord(u)|]$ sont distinctes. ∎

Lemme 2.32
Soit $ua \preceq_{préf} x$ avec $u \neq \varepsilon$ et $a \in A$. Si $a \neq x[|Bord(u)|]$, alors $Bord(ua)$ est, soit le plus long des mots de la forme $x[0\,..\,|{f'}^k(Bord(u))|]$, avec $k \geq 1$, satisfaisant $x[|{f'}^k(Bord(u))|] = a$, soit ε lorsqu'aucun naturel k ne convient.

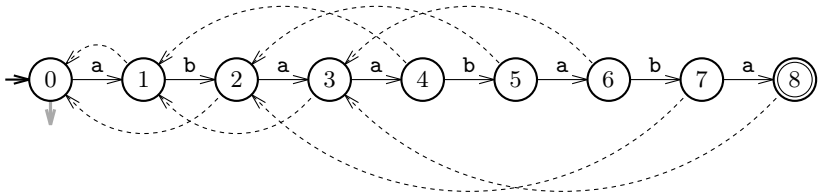
Preuve Analogue à la seconde partie de la preuve du lemme 1.22. ∎

Mise en table des fonctions de suppléance

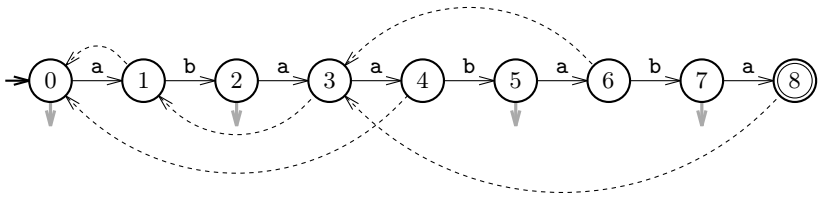
On choisit pour toute la suite une structure de données particulièrement bien adaptée au cas étudié, dans laquelle chaque état dans l'arbre $\mathcal{A}(\{x\})$ est représenté par son niveau. Il suffit ainsi pour représenter $\mathcal{A}(\{x\})$, son état terminal et l'une des fonctions de suppléance (f ou f') :

- du mot x ;
- de sa longueur $m = |x|$;
- d'une table indicée de 0 à m à valeurs dans $\{-1, 0, \ldots, m-1\}$;

(a)



(b)



(c)

i	0	1	2	3	4	5	6	7	8
$x[i]$	a	b	a	a	b	a	b	a	
bord$[i]$	0	0	1	1	2	3	2	3	
bon-préf$[i]$	-1	0	0	1	1	2	3	2	3
meil-préf$[i]$	-1	0	-1	1	0	-1	3	-1	3

Figure 2.12 Mise en table des fonctions de suppléance f et f' de l'implantation $\mathcal{D}_{\mathrm{S}}(\{x\})$ de l'automate-dictionnaire $\mathcal{D}(\{x\})$ lorsque $x =$ `abaababa`. **(a)** L'implantation $\mathcal{D}_{\mathrm{S}}(\{x\})$ et sa fonction de suppléance f (qui n'est autre que la fonction *Bord*). **(b)** L'implantation $\mathcal{D}_{\mathrm{S}}(\{x\})$ et sa fonction de suppléance optimisée f'. **(c)** Les tables *bord*, *bon-préf* et *meil-préf*. La première est donnée à titre de rappel. La deuxième correspond à f, et la troisième à f'.

la valeur NIL pour les états coïncidant, par convention, avec la valeur entière -1. Les tables correspondant respectivement aux fonctions de suppléance f et f' sont notées *bon-préf* et *meil-préf*. La première est appelée la ***table du bon préfixe***, la seconde la ***table du meilleur préfixe***. Elle sont donc définies par :

$$\textit{bon-préf}[i] = \begin{cases} |\mathit{Bord}(x[0\,..\,i-1])| & \text{si } i \neq 0 \ , \\ -1 & \text{sinon} \ , \end{cases}$$

et :

$$\textit{meil-préf}[i] = \begin{cases} |f'(x[0\,..\,i-1])| & \text{si } f'(x[0\,..\,i-1]) \text{ est défini} \ , \\ -1 & \text{sinon} \ , \end{cases}$$

pour $i = 0, 1, \ldots, m$. On remarque que :

$$\textit{bon-préf}[i] = \textit{bord}[i-1]$$

pour $i = 1, 2, \ldots, m$ (la table *bord* est introduite section 1.6). Un exemple est montré figure 2.12.

Les deux fonctions qui suivent produisent la table *bon-préf* et la table *meil-préf*. Pour la première, il s'agit d'une adaptation du code

de la fonction BORDS qui produit la table *bord*. Le « décalage » d'une unité sur les indices autorise une formulation algorithmique plus simple, notamment au niveau de la boucle aux lignes 6–7 sur les bords du préfixe $x[0\,..\,j-1]$. On suit le même schéma pour la seconde fonction, en appliquant les résultats des lemmes 2.31 et 2.32.

```
BON-PRÉFIXE(x, m)
 1  bon-préf[0] ← −1
 2  i ← 0
 3  pour j ← 1 à m − 1 faire
 4        ▷ Ici, x[0 .. i − 1] = Bord(x[0 .. j − 1])
 5        bon-préf[j] ← i
 6        tantque i ≥ 0 et x[j] ≠ x[i] faire
 7              i ← bon-préf[i]
 8        i ← i + 1
 9  bon-préf[m] ← i
10  retourner bon-préf
```

```
MEILLEUR-PRÉFIXE(x, m)
 1  meil-préf[0] ← −1
 2  i ← 0
 3  pour j ← 1 à m − 1 faire
 4        ▷ Ici, x[0 .. i − 1] = Bord(x[0 .. j − 1])
 5        si x[j] = x[i] alors
 6              meil-préf[j] ← meil-préf[i]
 7        sinon meil-préf[j] ← i
 8              faire  i ← meil-préf[i]
 9              tantque i ≥ 0 et x[j] ≠ x[i]
10        i ← i + 1
11  meil-préf[m] ← i
12  retourner meil-préf
```

Théorème 2.33
Les opérations BON-PRÉFIXE(x, m) *et* MEILLEUR-PRÉFIXE(x, m) *produisent respectivement la table du bon préfixe et la table du meilleur préfixe du mot x de longueur non nulle m.*

Preuve C'est une conséquence des définitions des tables *bon-préf* et *meil-préf*, de la proposition 1.23, des lemmes 2.31 et 2.32. ∎

Théorème 2.34
L'exécution de l'opération BON-PRÉFIXE(x, m) *prend un temps $\Theta(m)$ et nécessite au plus $2m - 3$ comparaisons entre lettres du mot x. Même chose pour l'opération* MEILLEUR-PRÉFIXE(x, m).

Preuve Voir la preuve de la proposition 1.24. ∎

Rappelons que la borne de $2m-3$ comparaisons a été établie par un raisonnement sur des variables locales de la fonction BORDS (à savoir i et j) et non par une étude combinatoire sur les mots de longueur m. On a montré qu'elle est atteinte dans le cas du calcul de la table *bord*. Elle l'est donc également pour *bon-préf*. Mais elle se trouve ne pas être si précise que cela pour *meil-préf*. On peut en effet montrer qu'elle n'est jamais atteinte lorsque $m \geq 3$, et que seuls les mots de la forme aba^{m-2} ou $aba^{m-3}c$ avec $a, b, c \in A$ et $a \neq b \neq c \neq a$ requièrent $2m-4$ comparaisons. L'établissement de cette borne précise est proposée en exercice (exercice 2.7).

Phase de localisation

Le code de l'algorithme qui réalise la localisation du mot non vide x de longueur m à l'aide de l'une des deux tables *bon-préf* ou *meil-préf* dans un texte y est donné ci-dessous. Le paramètre π qui y apparait représente l'une quelconque des deux tables. L'instruction conditionnelle [1] aux lignes 4–5 fait en sorte que $x[0\,..\,i-1]$ soit le plus long préfixe propre du mot x qui soit également un suffixe de y ; comme une occurrence de x vient d'être localisée, ce préfixe est le bord de x.

```
LOCALISER-SELON-PRÉFIXE(x, m, π, y)
 1  i ← 0
 2  pour chaque lettre a de y, séquentiellement faire
 3        ▷ Ici, x[0..i−1] est le plus long préfixe de x
           ▷   qui est également un suffixe de y
 4        si i = m alors
 5              i ← π[m]
 6        tantque i ≥ 0 et a ≠ x[i] faire
 7              i ← π[i]
 8        i ← i + 1
 9        SIGNALER-SI(i = m)
```

Théorème 2.35
Que le paramètre π représente la table bon-préf ou la table meil-préf, l'opération LOCALISER-SELON-PRÉFIXE(x, m, π, y) *s'exécute en temps $\Theta(|y|)$ et le nombre de comparaisons entre lettres de x et lettres de y qu'elle effectue n'excède jamais $2|y|-1$.*

Preuve La borne du nombre de comparaisons peut s'établir en considérant la quantité $2|u|-i$ où u est le préfixe courant de y (se reporter à la preuve du lemme 2.13). La linéarité en $|y|$ pour la complexité en temps s'en déduit. ∎

1. Noter que cette instruction peut être supprimée si l'on peut positionner une lettre qui n'apparait pas dans y à la suite de x, à l'indice m.

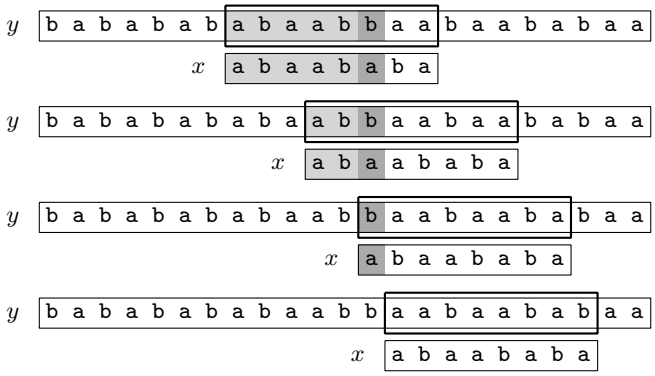


Figure 2.13 Comportements locaux pour l'implantation $\mathcal{D}_{\mathrm{S}}(\{x\})$ selon que la fonction de suppléance utilisée est f ou sa version optimisée f', avec ici $x = \texttt{abaababa}$. (Se reporter à la figure 2.12 pour voir les valeurs des deux tables *bon-préf* et *meil-préf* correspondantes.) L'illustration a recours aux artifices employés ailleurs pour les algorithmes de localisation avec fenêtre glissante. Le suffixe de longueur 5 du préfixe courant du texte (sa partie déjà analysée) et le préfixe de longueur 5 de x sont identiques (zones en gris clair). La comparaisons des lettres aux positions suivantes est négative (zones en gris sombre). Avec la fonction de suppléance f, la fenêtre est décalée de $5 - \mathit{bon\text{-}préf}[5] = \mathit{pér}(\texttt{abaab}) = 3$ positions ; les deux comparaisons suivantes étant encore négatives, la fenêtre est décalée de $2 - \mathit{bon\text{-}préf}[2] = \mathit{pér}(\texttt{ab}) = 2$ positions, puis de $0 - \mathit{bon\text{-}préf}[0] = 1$ position. Soit 3 comparaisons en tout sur la même lettre du texte, pour un décalage de 6 positions au final. Alors que si la version optimisée f' est utilisée, la fenêtre est directement décalée de $5 - \mathit{meil\text{-}préf}[5] = 6$ positions, après seulement une comparaison.

Comme indiqué dans la preuve de la proposition 2.16, le pire des cas de $2|y| - 1$ comparaisons est atteint lorsque, pour $a, b \in A$ avec $a \neq b$, ab est un préfixe de x tandis que y n'est formé que de a.

Si elle ne se traduit pas sur la borne du pire des cas du nombre de comparaisons, l'utilisation de la fonction de suppléance optimisée est qualitativement appréciable : une lettre du texte y n'est jamais comparée à deux lettres identiques du mot x consécutivement. Deux exemples sont donnés figure 2.13. On profite de cette illustration pour montrer que la localisation d'un mot par un automate (ou de l'une de ses implantations) peut très bien s'interpréter à l'aide du mécanisme de la fenêtre glissante. Dans le cas présent, l'affectation $i \leftarrow \pi[i]$ à la ligne 7 correspond à un décalage de la fenêtre de $i - \pi[i]$ positions vers la droite ; et pour l'affectation $i \leftarrow \pi[m]$ à la ligne 5, il s'agit d'un décalage de la période du mot x.

Plus généralement maintenant, si le nombre de comparaisons sur une même lettre du texte peut atteindre m avec la fonction de suppléance f (lorsque $x = a^m$ avec $a \in A$ et qu'une lettre différente de a se trouve être alignée avec la dernière lettre de x), il est inférieur à $\log_\Phi(m + 1)$

avec la fonction de suppléance f'. C'est ce qu'indique le corollaire 2.38, établi à la suite du lemme 2.36 et du théorème 2.37.

Lemme 2.36
On a :

$f'^2(u)$ *défini implique* $|u| \geq |f'(u)| + |f'^2(u)| + 2$

pour tout $u \prec_{préf} x$.

Preuve Puisque les mots $f'(u)$ et $f'^2(u)$ sont des bords de u, les entiers $p = |u| - |f'(u)|$ et $q = |u| - |f'^2(u)|$ sont des périodes de u. Par l'absurde, si l'on suppose que $|u| \leq |f'(u)| + |f'^2(u)| + 1$, on a également $(|u| - |f'(u)|) + (|u| - |f'^2(u)|) - 1 \leq |u|$, soit $p + q - 1 \leq |u|$. Le lemme de périodicité indique alors que $q-p$ est une période de u. En conséquence de quoi les deux lettres $x[|f'^2(u)|]$ et $x[|f'(u)|]$ de u sont identiques puisque situées à des positions distantes d'une quantité $q - p$, ce qui contredit la définition de $f'^2(u)$ et achève la preuve. ∎

Théorème 2.37
Lors de l'opération LOCALISER-SELON-PRÉFIXE$(x, m, meil\text{-}préf, y)$, *le nombre de comparaisons consécutives effectuées sur une même lettre du texte* y *est inférieur au plus grand entier* k *qui satisfait l'inégalité* $|x| + 1 \geq F_{k+2}$.

Preuve Soit k le plus grand entier associé aux suites

$\langle u, f'(u), f'^2(u), \ldots, f'^{k-1}(u) \rangle$

où $u \prec_{préf} x$ et $f'^k(u)$ n'est pas défini. Cet entier k majore le nombre de comparaisons considéré dans l'énoncé du théorème. Nous allons montrer par récurrence sur k que :

$$|u| \geq F_{k+2} - 2 \ . \tag{2.5}$$

L'inégalité (2.5) est vérifiée lorsque $k = 1$ (car $F_3 - 2 = 0$) et $k = 2$ (car $F_4 - 2 = 1$, et il est nécessaire que u soit non vide afin que $f'(u)$ soit défini). Supposons pour la suite $k \geq 3$. Dans ce cas, $f'(u)$ et $f'^2(u)$ existent, et la récurrence s'applique à ces deux mots. Il s'ensuit donc que :

$$\begin{array}{lcll} |u| & \geq & |f'(u)| + |f'^2(u)| + 2 & \text{(d'après le lemme 2.36)} \\ & \geq & (F_{k+1} - 2) + (F_k - 2) + 2 & \text{(récurrence)} \\ & = & F_{k+2} - 2 \ . & \end{array}$$

Ce qui achève la preuve par récurrence de l'inégalité (2.5).

Puisque $u \prec_{préf} x$, il s'ensuit que $|x| + 1 \geq |u| + 2 \geq F_{k+2}$, ce qui est le résultat annoncé. ∎

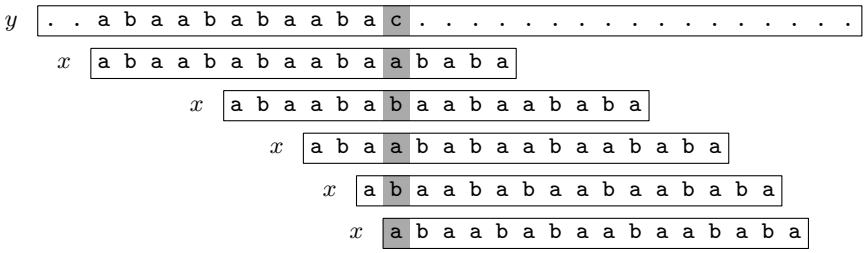


Figure 2.14 Lorsque le mot x est un préfixe d'un mot de Fibonacci et que k est l'entier tel que $F_{k+2} \leq |x| + 1 < F_{k+3}$, le nombre de comparaisons consécutives effectuées sur une même lettre du texte y lors d'une localisation à l'aide de l'implantation $\mathcal{D}_{\text{S}}(\{x\})$ peut être égal à k. Ici, $x =$ `abaababaabaababa`. C'est un préfixe de f_8 ; $F_7 = 13$, $|x| + 1 = 17$, $F_8 = 21$, donc $k = 5$; et cinq comparaisons sont effectivement réalisées sur la lettre `c` de y.

Corollaire 2.38
Lors de l'opération LOCALISER-SELON-PRÉFIXE$(x, m, \textit{meil-préf}, y)$, *le nombre de comparaisons consécutives effectuées sur une même lettre du texte y est inférieur à* $\log_\Phi(|x|+1)$. *Le délai de l'opération est* $O(\log|x|)$.

Preuve Si k est le maximum du nombre de comparaisons consécutives effectuées sur une même lettre du texte, on a, d'après le théorème 2.37 :

$$|x| + 1 \geq F_{k+2} \ .$$

De l'inégalité classique

$$F_{n+2} \geq \Phi^n \ ,$$

il vient $|x| + 1 \geq \Phi^k$, soit $\log_\Phi(|x| + 1) \geq k$. Les résultats annoncés s'en déduisent. ∎

La borne sur la longueur de x donnée dans l'énoncé du théorème 2.37 est très précise : elle est atteinte lorsque x est un préfixe d'un mot de Fibonacci. Un exemple est donné figure 2.14.

2.7 Localisation d'un mot et successeur par défaut

Nous considérons à nouveau l'implantation de l'automate-dictionnaire avec l'état initial comme successeur par défaut (voir section 2.4) en l'appliquant au cas particulier d'un dictionnaire composé d'un seul mot non vide x. Nous montrons que, contrairement au cas général, il n'est pas nécessaire de maintenir les ensembles de successeurs étiquetés ordonnés selon l'alphabet afin d'assurer la linéarité de la construction de l'implantation $\mathcal{D}_{\text{D}}(\{x\})$ en la longueur du mot x à localiser. Nous montrons

également que le délai est logarithmique en la longueur du mot cherché, indépendamment du maintien ou non d'un ordre éventuel dans les ensembles des successeurs étiquetés.

Construction de l'implantation

Il vient directement de la définition de $\mathcal{D}(\{x\})$, des propositions 2.19 et 2.23 le résultat suivant.

Théorème 2.39
La taille de $\mathcal{D}_{\text{D}}(\{x\})$ est $O(|x|)$. Plus précisément, $\mathcal{D}_{\text{D}}(\{x\})$ possède $|x|+1$ états, $|x|$ flèches avant, et au plus $|x|$ flèches arrière. ∎

La méthode de construction de l'implantation suit celle développée pour l'automate complet (section 2.2). Il s'agit simplement ici de ne pas générer les flèches de $\mathcal{D}(\{x\})$ qui entrent dans l'état initial. À cette fin, on adapte le code de la fonction ALU-COMPLET en supprimant la boucle **pour** des lignes 3–4 et en insérant, après la ligne 8, des instructions pour simuler le retour à l'état initial de certaines flèches. On obtient le code que voici :

```
ALU-PAR-DÉFAUT(x)
 1  M ← NOUVEL-AUTOMATE()
 2  q_0 ← initial[M]
 3  t ← q_0
 4  pour chaque lettre a de x, séquentiellement faire
 5        p ← NOUVEL-ÉTAT()
 6        r ← CIBLE(t, a)
 7        si r = NIL alors
 8              r ← q_0
 9        sinon Succ[t] ← Succ[t] \ {(a, r)}
10        Succ[t] ← Succ[t] ∪ {(a, p)}
11        Succ[p] ← Succ[r]
12        t ← p
13  terminal[t] ← VRAI
14  retourner M
```

Proposition 2.40
L'opération ALU-PAR-DÉFAUT(x) *produit $\mathcal{D}_{\text{D}}(\{x\})$, implantation de l'automate $\mathcal{D}(\{x\})$ par successeur par défaut.* ∎

On établit maintenant un résultat sur la construction de l'implantation qui est bien plus qu'une adaptation immédiate du théorème 2.25.

Théorème 2.41
L'opération ALU-PAR-DÉFAUT(x) *s'effectue en temps $O(|x|)$ en utilisant un espace supplémentaire constant, que les ensembles de successeurs étiquetés soient ordonnés ou non selon l'alphabet.*

Preuve Les opérations sur l'ensemble des successeurs étiquetés d'un état qui n'est ni l'état initial, ni l'état terminal de l'automate sont celles des lignes 11, 5, 6, éventuellement 9, puis 10 de la fonction ALU-PAR-DÉFAUT. Chacune d'entre elles se réalise en temps au plus linéaire en le cardinal final de l'ensemble. Il est clair qu'il en est de même pour l'état initial et l'état terminal. Il vient au total que le temps de la construction est au plus linéaire en la somme des cardinaux des ensembles des successeurs étiquetés, laquelle est $O(|x|)$ d'après le théorème 2.39. ∎

Phase de localisation

Le théorème suivant se déduit directement du lemme 2.26 et de la proposition 2.28.

Théorème 2.42
Pour l'opération LA-DÉTERMINISTE-PAR-DÉFAUT$(\{x\}, y)$, *la phase de localisation s'exécute en temps* $O(|y|)$. *Plus précisément, le nombre de comparaisons effectuées entre lettres de* x *et de* y *est au plus égal* $2|y|-1$ *lorsque* $y \neq \varepsilon$, *quel que soit l'ordre dans lequel sont examinés les éléments des ensembles des successeurs étiquetés.* ∎

Il reste à préciser l'ordre de grandeur du délai. Rappelons que celui-ci dépend directement du maximum des degrés sortant des états de l'implantation $\mathcal{D}_{\text{D}}(\{x\})$. Plus loin, on note deg_u la fonction qui à tout état v dans $\mathcal{D}_{\text{D}}(\{u\})$ associe le degré sortant de l'état v. Les lemmes 2.43 et 2.44 expriment des relations de récurrence sur les degrés sortant.

Lemme 2.43
Soit $(u, a) \in A^* \times A$. *Alors, pour tout* $w \preceq_{\text{préf}} ua$, *on a :*

$$deg_{ua}(w) = \begin{cases} deg_u(Bord(ua)) & \text{si } w = ua \ , \\ deg_u(u) + 1 & \text{si } w = u \text{ et } Bord(ua) = \varepsilon \ , \\ deg_u(w) & \text{sinon} \ . \end{cases}$$

Preuve C'est une conséquence directe de la proposition 2.29. ∎

Lemme 2.44
On a :

$$deg_x(x) = deg_x(Bord(x)) \ .$$

De plus, pour tout $va \preceq_{\text{préf}} x$, *avec* $v \prec_{\text{préf}} x$ *et* $a \in A$, *on a :*

$$deg_x(v) = \begin{cases} deg_x(Bord(v)) + 1 & \text{si } v \neq \varepsilon \text{ et } Bord(va) = \varepsilon \ , \\ deg_x(Bord(v)) & \text{si } v \neq \varepsilon \text{ et } Bord(va) \neq \varepsilon \ , \\ 1 & \text{si } v = \varepsilon \ . \end{cases}$$

Preuve C'est une conséquence directe du lemme 2.43. ∎

Le résultat qui suit est « la pierre angulaire » de la preuve de la borne logarithmique qui sera donnée dans le lemme 2.46.

Lemme 2.45
Pour tout préfixe non vide u de x, on a :

$2|Bord(u)| \geq |u|$ *implique* $deg_x(Bord(u)) = deg_x(Bord^2(u))$.

Preuve Posons $k = 2|Bord(u)| - |u|$, puis $w = u[0\,..\,k-1]$ et $a = w[k]$. Remarquons alors que, puisque wa est un bord de $Bord(u)a$, le bord de $Bord(u)a$ n'est pas vide. On applique ensuite le lemme 2.44 au préfixe $va = Bord(u)a$ de x. ∎

Lemme 2.46
Pour tout $u \preceq_{préf} x$, on a :

$deg_x(u) \leq \lfloor \log_2(|u|+1) \rfloor + 1$.

Preuve Nous montrons la propriété par récurrence sur la longueur $|u|$ des préfixes propres u de x. Si $|u| = 0$, la propriété est vraie puisque $deg_x(\varepsilon) = 1$. Pour l'étape de récurrence, posons $|u| \geq 1$ et la propriété vraie pour tous les préfixes de x de longueur strictement inférieure à celle de u. Soit $i \in \mathbf{N}$ tel que :

$$2^i \leq |u| + 1 < 2^{i+1} \ , \tag{2.6}$$

puis soit $j \in \mathbf{N}$ tel que :

$$|Bord^{j+1}(u)| + 1 < 2^i \leq |Bord^j(u)| + 1 \ . \tag{2.7}$$

Pour $k = 0, 1, \ldots, j-1$, nous avons :

$$\begin{array}{lcll} 2|Bord^{k+1}(u)| & \geq & 2^{i+1} - 2 & \text{(d'après l'inégalité (2.7))} \\ & \geq & |u| & \text{(d'après l'inégalité (2.6))} \\ & \geq & |Bord^k(u)| \ . & \end{array}$$

Il s'ensuit alors, par application du lemme 2.45, que :

$deg_x(|Bord^{k+1}(u)|) = deg_x(|Bord^{k+2}(u)|)$

pour $k = 0, 1, \ldots, j-1$. Cela conduit à ce que :

$$deg_x(|Bord(u)|) = deg_x(|Bord^{j+1}(u)|) \ . \tag{2.8}$$

En conséquence, nous avons :

$$\begin{array}{lcll} deg_x(u) & \leq & deg_x(Bord(u)) + 1 & \text{(d'après le lemme 2.44)} \\ & = & deg_x(Bord^{j+1}(u)) + 1 & \text{(d'après l'égalité (2.8))} \\ & \leq & \lfloor \log_2(|Bord^{j+1}(u)| + 1) \rfloor + 2 & \text{(récurrence) ,} \\ & \leq & i + 1 & \text{(par définition de } j\text{)} \\ & = & \lfloor \log_2(|u| + 1) \rfloor + 1 & \text{(par définition de } i\text{) .} \end{array}$$

La propriété est donc vraie pour tout mot de longueur $|u|$, ce qui achève la preuve par récurrence. ∎

Théorème 2.47
Le degré de n'importe lequel des états de l'implantation $\mathcal{D}_{\text{D}}(\{x\})$ *est inférieur à* $\min\{\operatorname{card} alph(x), 1 + \lfloor \log_2 |x| \rfloor\}$.

Preuve La borne qui dépend de l'alphabet du mot résulte de la proposition 2.21. Quant à la borne qui dépend de la longueur du mot, c'est une conséquence directe des lemmes 2.44 (pour l'état x) et 2.46 (pour les autres états). ∎

Le théorème 2.47 a pour conséquence le corollaire suivant.

Corollaire 2.48
Pour l'opération LA-DÉTERMINISTE-PAR-DÉFAUT$(\{x\}, y)$, *le délai est* $O(s)$ *où*

$$s = \min\{\operatorname{card} alph(x), 1 + \lfloor \log_2 |x| \rfloor\} \ ,$$

quel que soit l'ordre dans lequel sont examinés les éléments des ensembles des successeurs étiquetés. Lorsque ces ensembles sont ordonnés selon l'alphabet, le délai devient $O(\log s)$. ∎

La borne du degré donnée dans le théorème 2.47 est optimale à $|x|$ fixé (et en tenant compte de l'alphabet). Considérons

$$\xi\colon A^* \to A^*$$

la fonction définie par la récurrence :

$$\begin{aligned} \xi(\varepsilon) &= \varepsilon \\ \xi(ua) &= \xi(u) \cdot a \cdot \xi(u) \qquad \text{pour } (u, a) \in A^* \times A \ . \end{aligned}$$

Alors, lorsque le mot $\xi(a_1 a_2 \dots a_{k-1}) a_k$ est un préfixe du mot x avec $k = \min\{\operatorname{card} A, 1 + \lfloor \log_2 |x| \rfloor\}$ et $a_1, a_2, \dots, a_k$ des lettres deux à deux distinctes, le degré sortant de l'état $\xi(a_1 a_2 \dots a_{k-1})$ est exactement k. Un exemple est montré figure 2.15 sur un alphabet composé d'au moins les lettres `a`, `b`, `c` et `d` et avec $x = \xi(\texttt{abc})\texttt{d}$.

Challenge d'implantations pour la localisation d'un mot

Les observations de la fin de la section 2.4 concernant les deux implantations avec fonction de suppléance et avec successeur par défaut se doivent d'être partiellement amendées. Dans le cas de la localisation d'un seul mot, les complexités en temps et en espace sont toutes linéaires, soit en la longueur du mot à localiser pour les structures à mémoriser, soit en la longueur du texte dans lequel s'effectue la recherche du mot pour la localisation. L'implantation avec l'état initial comme successeur par défaut présente toutefois quelques avantages supplémentaires :

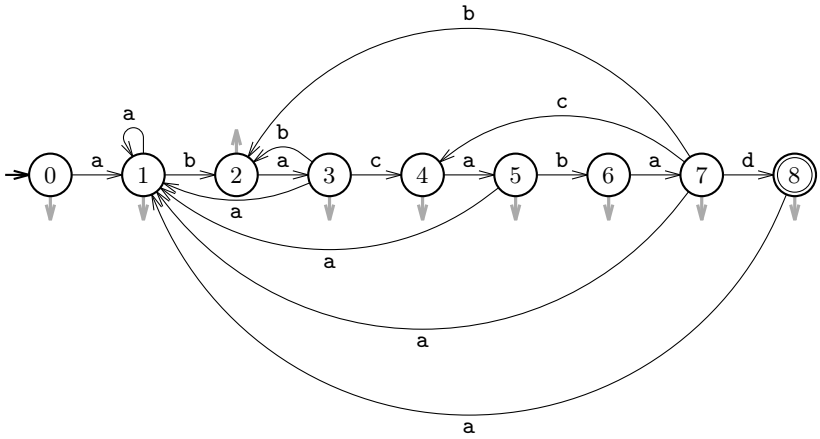


Figure 2.15 L'implantation $\mathcal{D}_{\mathrm{D}}(\{x\})$ lorsque $x =$ `abacabad`. Le maximum des degrés sortant des états est égal à 4 (état 7), ce qui est le maximum possible pour un mot qui, comme x, a une longueur comprise entre 2^3 et $2^4 - 1$ et est formé d'au moins quatre lettres distinctes.

1. Temps réel sur un alphabet considéré constant, c'est-à-dire délai borné par une constante.
2. Délai logarithmique de base 2, ce qui est mieux qu'un délai logarithmique de base Φ.
3. Nombre de comparaisons entre lettres du mot et du texte moindre lorsque les successeurs étiquetés des états sont inspectés par ordre décroissant de niveau (voir l'exemple figure 2.16).
4. Ordre d'inspection des successeurs modifiable à souhait, sans aucune conséquence sur la linéarité des complexités.

Notes

Les résultats présentés dans ce chapitre proviennent initialement des travaux de Knuth, Morris et Pratt (1977) et d'Aho et Corasick (1974).

La localisation par suppléance pour un dictionnaire de la section 2.3 est adaptée d'Aho et Corasick (voir [12]). Le traitement de la section 2.4 est original ; il poursuit les travaux de Simon (1990) et de Hancart (1993) pour le cas de la localisation d'un seul mot.

L'algorithme de localisation par les préfixes (section 2.6) est de Morris et Pratt (1970). Sa version optimisée (même section) est une adaptation de celle donnée par Knuth, Morris et Pratt (1977). La linéarité de la taille de l'implantation $\mathcal{D}_{\mathrm{D}}(\{x\})$ (section 2.7) est due à Simon (1990) (voir [36]). Il a montré simultanément la linéarité de la construction et de la localisation associées. Le fait que l'ordre d'inspection ne modifie pas

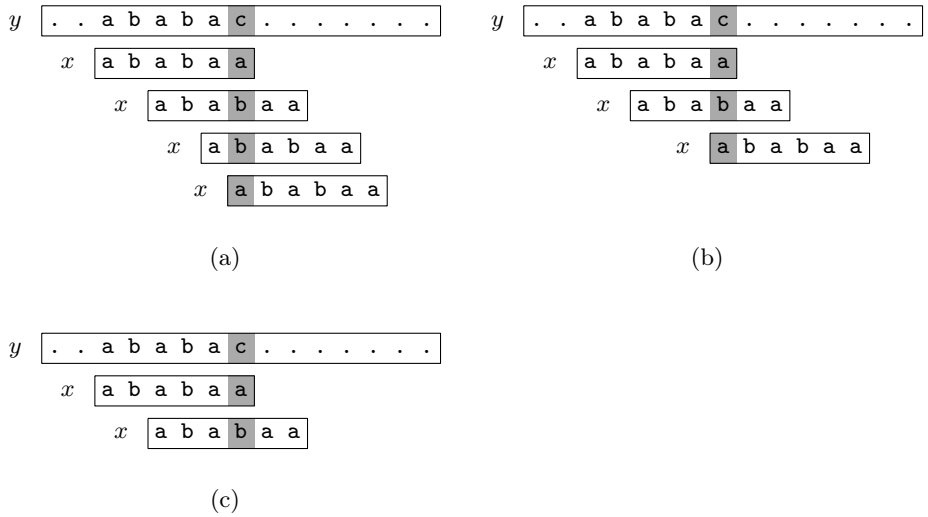


Figure 2.16 Comportement de trois algorithmes séquentiels de localisation d'un mot lorsque la dernière lettre du mot $x =$ `ababaa` est alignée avec la lettre `c` qui apparait dans le texte y. **(a)** Implantation $\mathcal{D}_{\mathrm{S}}(\{x\})$ avec la fonction de suppléance f ; 4 comparaisons entre les lettres de x et la lettre courante de y, dont 2 redondantes. **(b)** Sa version avec f' ; 3 comparaisons, la dernière étant redondante. **(c)** Implantation $\mathcal{D}_{\mathrm{D}}(\{x\})$ lorsque les éléments des ensembles des successeurs étiquetés sont examinés par niveau décroissant ; 2 comparaisons, et l'on ne peut faire mieux.

la linéarité des phases de construction et de localisation est dû à Hancart (1993). Il a donné la borne exacte du délai. La borne exacte du nombre de comparaisons pour le problème de la localisation séquentielle d'un mot dans le modèle comparaisons a été donnée à la suite par Hancart (1993 ; voir exercice 2.9) et, pour un problème voisin, par Breslauer, Colussi et Toniolo (1998 ; voir exercice 2.13).

Exercices

2.1 (Rationnel)
Comparer les complexités des algorithmes de localisation des occurrences des mots d'un dictionnaire décrits dans ce chapitre – dont l'une des entrées est le dictionnaire – avec ceux – standard – dont l'entrée est une expression rationnelle décrivant le dictionnaire.

2.2 (Déterminisation)
Montrer que si l'on considère la construction par sous-ensemble comme méthode de déterminisation de l'automate $\mathcal{A}(X)$ augmenté d'une boucle

sur l'état initial, les états du déterminisé sont de la forme :

$\{u, f(u), f^2(u), \ldots, \varepsilon\}$

avec $u \in \mathit{Préf}(X)$.

2.3 (Suppléance)
Montrer que la fonction f peut s'exprimer indépendamment de la fonction h, à savoir qu'elle vérifie pour tout $(u, a) \in A^* \times A$ la relation :

$$f(ua) = \begin{cases} f(u)a & \text{si } u \neq \varepsilon \text{ et } f(u)a \in \mathit{Préf}(X) \ , \\ f(f(u)a) & \text{si } u \neq \varepsilon \text{ et } f(u)a \notin \mathit{Préf}(X) \ , \\ \varepsilon & \text{sinon} \ . \end{cases}$$

2.4 (Paresse)
Donner de chacun des algorithmes de localisation du chapitre une version paresseuse qui construit l'automate associé – ou l'une de ses implantations particulières – au fur et à mesure des besoins de la localisation.

2.5 (Boucle rapide)
Réaliser l'implantation de $\mathcal{D}_{\text{S}}(X)$ avec une boucle rapide sur l'état initial à l'aide d'une table sur l'alphabet. [*Aide :* il s'agit là de l'algorithme original d'Aho et Corasick ; voir [12].]

2.6 (Bille en tête)
On considère que les ensembles de successeurs étiquetés de l'implantation $\mathcal{D}_{\text{D}}(X)$ sont des listes. On applique à ces listes la technique de recherche autoadaptative qui, pour chaque recherche fructueuse d'un élément particulier dans une liste, réorganise la liste en plaçant l'élément trouvé en tête. Quelle est la complexité de la construction de cette implantation ? Quelle est la complexité de la phase de localisation (incluant le maintien de telles listes) ?

2.7 (Borne)
Montrer que $2m - 4$ est la borne exacte du nombre maximal de comparaisons entre lettres effectuées lors du calcul de la table du meilleur préfixe des mots x de longueur $m \geq 3$ lors de l'opération MEILLEUR-PRÉFIXE(x, m). [*Aide :* montrer éventuellement que la borne des $2m - 3$ comparaisons n'est atteinte que pour les mots de la forme $a^{m-1}b$ avec $m \geq 2$, $a, b \in A$ et $a \neq b$ lors de l'opération BON-PRÉFIXE(x, m). Montrer ensuite que seuls les mots de la forme aba^{m-2} ou $aba^{m-3}c$ avec $m \geq 3$, $a, b, c \in A$ et $a \neq b \neq c \neq a$ requièrent $2m - 4$ comparaisons.]

2.8 (Le pire dévoilé)
Montrer que les préfixes des mots de Fibonacci atteignent la borne sur le nombre de comparaisons consécutives du corollaire 2.38.

2.9 (Les premières à la queue)
Montrer que le nombre de comparaisons effectuées lors de la localisation de tout mot x de longueur non nulle m dans un texte de longueur n est inférieur à $(2-1/m)n$ si l'on utilise l'implantation $\mathcal{D}_{\text{D}}\{x\}$ en inspectant la flèche avant en dernier lors de chaque transition.

Montrer que la borne $(2-1/m)n$ est une borne inférieure du pire des cas de la localisation séquentielle dans le modèle comparaisons. [*Aide :* voir Hancart (1993).]

2.10 (Temps réel)
Montrer que la localisation d'un mot ou de plusieurs mots peut s'effectuer en temps réel lorsque les lettres de l'alphabet sont codées en binaire.

2.11 (Conjugués)
Donner un algorithme qui teste si deux mots u et v sont conjugués ou non et qui fonctionne en temps $O(|u| + |v|)$.

2.12 (Palindromes)
On note P l'ensemble des palindromes de longueur paire. Montrer que l'on peut tester si un mot x appartient ou non à P^* – appelé ensemble des *palstars*, pour *even palindromes starred* – en temps et en espace $O(|x|)$. [*Aide :* voir Knuth, Morris et Pratt (1977).]

2.13 (Localisation de préfixes)
Le problème du *prefix-matching* consiste, pour un mot x et un texte y donnés, à déterminer en chacune des positions sur le texte le plus long préfixe de x dont l'occurrence s'y termine. Montrer que ce problème admet des solutions et des bornes (voir exercice 2.9) identiques à celui de la localisation séquentielle d'un mot dans un texte dans le modèle comparaisons. [*Aide :* voir Breslauer, Colussi et Toniolo (1998).]

2.14 (Test de codicité)
On considère un dictionnaire $X \subset A^*$ et le graphe associé $\mathcal{G} = (Q, F)$ dans lequel $Q = \mathit{Préf}(X) \setminus \{\varepsilon\}$ et F est l'ensemble de couple (u, v) de mots de Q qui sont tels que $uv \in X$ (arc croisé) ou, à la fois, $v \notin X$ et $uz = v$ pour un mot $z \in X$ (arc avant).

Montrer que X est un code (voir exercice 1.10) si et seulement si le graphe $\mathcal{G}$ ne possède pas de chemin qui relie deux éléments de X.

Écrire un algorithme de construction de $\mathcal{G}$ qui utilise l'automate-dictionnaire associé à X.

Compléter l'algorithme pour obtenir un test de codicité de X.

Quelle est la complexité de l'algorithme ? [*Aide :* voir Sardinas et Patterson (1953).]

3 Localisation avec fenêtre glissante

On considère dans ce chapitre le problème de la localisation de toutes les occurrences d'un mot fixe dans un texte. Les méthodes décrites ici sont basées sur des propriétés combinatoires. Elles s'appliquent lorsque le mot et le texte sont en mémoire centrale ou qu'une portion seulement de ce dernier est en mémoire tampon. Contrairement aux solutions présentées dans le chapitre précédent, la localisation ne traite pas le texte de façon strictement séquentielle.

Les algorithmes du chapitre examinent le texte au travers d'une fenêtre de même longueur que le mot. Le traitement qui consiste à déterminer si le contenu de la fenêtre coïncide ou non avec le mot est appelé une tentative, suivant en cela le mécanisme de la fenêtre glissante décrit section 1.5. À la fin de chaque tentative la fenêtre est décalée en direction de la fin du texte. Les exécutions de ces algorithmes sont donc des successions de tentatives suivies de décalages.

On considère les algorithmes qui, au cours de chaque tentative, effectuent les comparaisons entre lettres du mot et de la fenêtre de la droite vers la gauche, c'est-à-dire en sens inverse du sens de lecture usuel. Ces algorithmes repèrent ainsi des suffixes du mot à l'intérieur du texte. L'intérêt de cette technique est, qu'au cours d'une tentative, l'algorithme accumule de l'information sur une portion du texte qui sera vraisemblablement traitée ensuite.

Nous présentons trois versions de plus en plus efficaces en termes de nombre de comparaisons entre lettres effectuées par les algorithmes. La première ne mémorise aucune information, la deuxième mémorise l'information sur la tentative précédente et la troisième l'information sur toutes les tentatives précédentes. Le nombre de comparaisons permet d'évaluer le gain obtenu. Nous considérons ensuite une méthode qui généralise le procédé pour localiser les mots d'un dictionnaire dans un texte.

3.1 Localisation sans mémoire

Soit x un mot de longueur m dont on veut localiser toutes les occurrences dans un texte y de longueur n. Dans la suite, on dit de x qu'il est ***périodique*** lorsque $pér(x) \leq m/2$.

Nous présentons tout d'abord une méthode qui réalise la localisation au moyen du principe décrit en introduction. Lorsque le mot cherché x est non périodique, elle effectue moins de $3n$ comparaisons entre lettres. L'une des caractéristiques de cet algorithme est qu'il ne garde aucune mémoire des tentatives précédentes.

Cette section contient en plus une version affaiblie de la méthode, dont l'analyse est donnée à la suite. La phase de prétraitement de cette version, qui s'exécute en temps et en espace $O(m)$, est plus simple ; elle est développée dans la section 3.3. Le prétraitement de la version initiale, basé sur un automate, est décrit en section 3.4.

On considère dans ce chapitre que lorsqu'une tentative a lieu à la position j sur le texte y, la fenêtre contient le facteur $y[j-m+1\,..\,j]$ du texte y. L'indice j est donc la position droite du facteur. Le ***plus long suffixe commun*** à deux mots u et v étant noté

$$lsc(u, v) \ ,$$

pour une tentative T à la position j sur le texte y, on pose :

$$z = lsc(y[0\,..\,j], x) \ ,$$

et d la longueur du décalage appliqué juste après la tentative T.

La situation générale à la fin de la tentative T est la suivante : le suffixe z de x a été identifié dans le texte y et, si $|z| < |x|$, une comparaison négative est intervenue entre la lettre $a = x[m - |z| - 1]$ du mot et la lettre $b = y[j - |z|]$ du texte. Autrement dit, en posant $i = m - |z| - 1$, on a $z = x[i+1\,..\,m-1] = y[j-m+i+2\,..\,j]$ et, soit $i = -1$, soit $i \geq 0$ avec $a = x[i]$, $b = y[j - m + i + 1\,..\,j]$ et $a \neq b$ (voir figure 3.1).

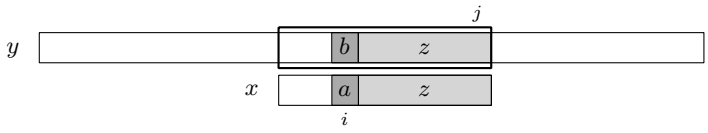


Figure 3.1 Situation générale à la fin d'une tentative à la position j. La comparaison du contenu de la fenêtre $y[j - m + 1\,..\,j]$ avec le mot x procède par comparaisons lettre à lettre, de la droite vers la gauche. Le mot z est le plus long suffixe commun à $y[0\,..\,j]$ et x (zone de comparaisons positives, signifiée en gris clair). Lorsque ce suffixe du mot x n'est pas x tout entier, la position i sur x en laquelle survient une comparaison négative (en gris foncé) vérifie $i = m - |z| - 1$.

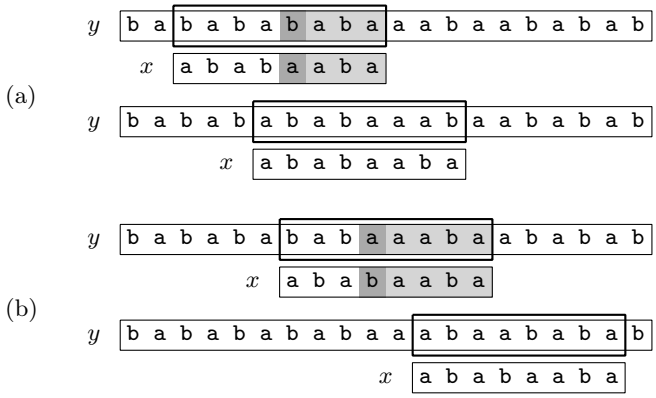


Figure 3.2 Décalages à la suite de tentatives. **(a)** Lors de la tentative à la position 9, on détecte le suffixe `aba` du mot dans le texte. Une comparaison négative intervient entre $x[4] = \texttt{a}$ et $y[6] = \texttt{b}$. Le décalage à appliquer consiste à aligner le facteur `baba` du texte avec son occurrence (droite) dans x. On applique ici un décalage de longueur 3. **(b)** Lors de la tentative à la position 13, on détecte le suffixe `aaba` du mot dans le texte. Une comparaison négative intervient entre $x[3] = \texttt{b}$ et $y[9] = \texttt{a}$. Le facteur `aaaba` n'apparait pas dans x ; le décalage à appliquer consiste à aligner un plus long préfixe du mot coïncidant avec un suffixe du facteur `aaaba` du texte. Ici, ce préfixe est `aba` et la longueur du décalage est 5.

Compte tenu des informations collectées sur le texte y au cours de la tentative, le décalage naturel à appliquer consiste à aligner le facteur bz du texte avec son occurrence la plus à droite dans x. Si bz n'est pas facteur de x, il faut alors effectuer l'alignement (par la droite) avec le plus long préfixe de x qui est également un suffixe de z. Ces deux cas sont illustrés dans la figure 3.2.

Dans les deux situations qui viennent d'être examinées, le calcul du décalage consécutif à T est indépendant du texte. Il peut être calculé au préalable pour chaque position du mot et pour chaque lettre de l'alphabet. Dans ce but, on définit deux conditions qui correspondent au cas où le mot z est le suffixe $x[i+1\,..\,m-1]$ de x. Il s'agit de la ***condition de suffixe*** *Cs* et de la ***condition d'occurrence*** de lettre *Co*. Elles sont définies, pour toute position i sur x, tout décalage d de x et toute lettre $b \in A$, par :

$$Cs(i,d) = \begin{cases} 0 < d \leq i+1 \text{ et } x[i-d+1\,..\,m-d-1] \preceq_{suff} x \\ \text{ou} \\ i+1 < d \text{ et } x[0\,..\,m-d-1] \preceq_{suff} x \end{cases}$$

et :

$$Co(b,i,d) = \begin{cases} 0 < d \leq i \text{ et } x[i-d] = b \\ \text{ou} \\ i < d \ . \end{cases}$$

Puis, la ***fonction du meilleur facteur***, notée *meil-fact*, est définie de la manière suivante, pour toute position i sur x et toute lettre $b \in A$:

$$\textit{meil-fact}(i, b) = \min\{d : \mathit{Cs}(i, d) \text{ et } \mathit{Co}(b, i, d) \text{ satisfaites}\} \ .$$

On note que $\textit{meil-fact}(i, b)$ est toujours défini car les conditions sont satisfaites pour $d = m$.

Une implantation directe de la fonction du meilleur facteur nécessite un espace mémoire $O(m \times \operatorname{card} A)$. En fait, une solution plus fine à base d'automate n'a besoin que d'un espace $O(m)$. Elle est présentée dans la section 3.4. On introduit plus loin une version affaiblie de la fonction pour laquelle la linéarité de l'implantation est immédiate.

Phase de recherche

Au cours d'une tentative à la position j sur le texte y, et lorsqu'une comparaison négative intervient entre la lettre $x[i]$ du mot et la lettre $y[j - m + 1 + i]$ du texte, on applique un décalage de longueur

$$d = \textit{meil-fact}(i, y[j - m + 1 + i]) \ .$$

Une fois le décalage effectué, la première condition, $\mathit{Cs}(i, d)$, assure que le facteur $y[j - m + 2 + i \,..\, j]$ du texte et le facteur (ou préfixe) du mot avec lequel il est aligné sont identiques. Tandis que la deuxième condition $\mathit{Co}(y[j - m + 1 + i], i, d)$ assure que si une lettre du mot est alignée avec $y[j - m + 1 + i]$ alors elle coïncide avec celle-ci.

On note que, si au cours d'une tentative, une occurrence du mot est découverte dans le texte (ce qui correspond à $i = -1$), le décalage à appliquer est de longueur $\mathit{pér}(x)$. On note en plus que :

$$\textit{meil-fact}(0, b) = \mathit{pér}(x)$$

pour toute lettre $b \in A$.

L'algorithme LS-SANS-MÉMOIRE, dont le code est donné ci-dessous, implante la méthode qui vient d'être décrite.

```
LS-SANS-MÉMOIRE(x, m, y, n)
 1  j ← m − 1
 2  tantque j < n faire
 3        i ← m − 1
 4        tantque i ≥ 0 et x[i] = y[j − m + 1 + i] faire
 5              i ← i − 1
 6        SIGNALER-SI(i < 0)
 7        si i < 0 alors
 8              j ← j + pér(x)
 9        sinon j ← j + meil-fact(i, y[j − m + 1 + i])
```

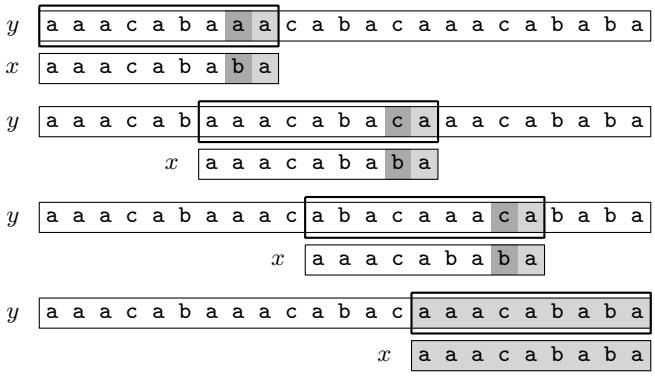


Figure 3.3 Exemple d'exécution de l'algorithme LS-SANS-MÉMOIRE. Ici, quinze comparaisons entre lettres du mot et lettres du texte sont effectuées.

Un exemple d'exécution est donné figure 3.3. Les valeurs des mots x et y considérées dans cet exemple seront réutilisées dans la suite. Elles servent à illustrer les différences de comportement des divers algorithmes de localisation présentés dans le chapitre.

Théorème 3.1
L'algorithme LS-SANS-MÉMOIRE *localise toutes les occurrences du mot x dans le texte y.*

Preuve Par définition des fonctions *meil-fact* et *pér*, tous les décalages appliqués par l'algorithme LS-SANS-MÉMOIRE sont valides. L'algorithme ne peut donc manquer aucune occurrence de x dans y. ∎

On trouve facilement des cas pour lesquels le comportement de l'algorithme LS-SANS-MÉMOIRE est quadratique, $O(m \times n)$, par exemple lorsque $x = \mathtt{a}^m$ et $y = \mathtt{a}^n$.

Version faible

Il est possible d'approximer la fonction du meilleur facteur afin d'éviter le recours à un automate, ce qui simplifie la réalisation de l'ensemble. L'approximation est donnée par une fonction dite du bon suffixe implantée par une table. Il est traditionnel de lui adjoindre la table *dern-occ* introduite à la section 1.5.

On définit la nouvelle condition *Co-aff* par :

$$\textit{Co-aff}(i,d) = \begin{cases} 0 < d \leq i \text{ et } x[i-d] \neq x[i] \\ \text{ou} \\ i < d \ . \end{cases}$$

La ***table du bon suffixe*** est définie pour une position i sur x par :

$$\textit{bon-suff}[i] = \min\{d : \textit{Cs}(i,d) \text{ et } \textit{Co-aff}(i,d) \text{ satisfaites}\} \ .$$

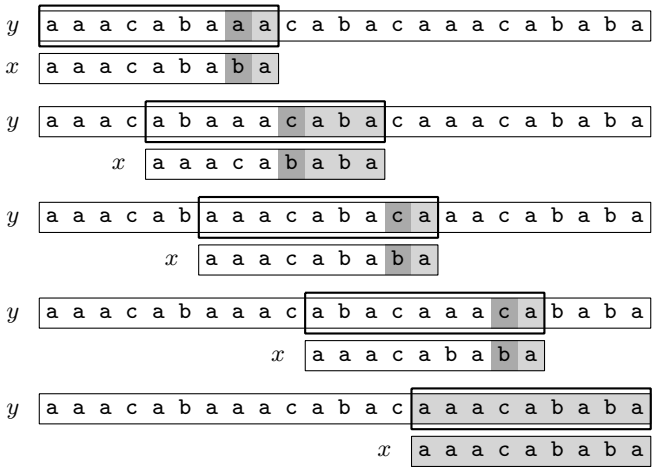


Figure 3.4 Exécution, sur l'exemple de la figure 3.3, de l'algorithme LS-SANS-MÉMOIRE-FAIBLE qui utilise la table du bon suffixe. Avec cet algorithme, dix-neuf comparaisons entre lettres du mot et lettres du texte sont effectuées.

La condition *Co-aff*(i, d) assure que si une lettre c du mot est alignée avec la lettre $b = y[j - m + 1 + i]$ après le décalage, alors c est différente de la lettre $a = x[i]$ avec laquelle b était alignée juste avant le décalage. Cela affaiblit la condition $Co(b, i, d)$ qui impose l'identité des lettres c et b (ligne 9 de l'algorithme LS-SANS-MÉMOIRE). On peut remarquer que dans le cas d'un alphabet binaire, l'utilisation de la table du bon suffixe dans l'algorithme de recherche coïncide avec l'utilisation de la fonction du meilleur facteur.

La phase de préparation de l'algorithme LS-SANS-MÉMOIRE-FAIBLE se résume ainsi au calcul de la seule table *bon-suff*. Elle est présentée dans la section 3.3. Comme précédemment, on note que *bon-suff*$[0]$ a pour valeur *pér*(x).

Dans le code qui suit, on utilise uniquement la table *bon-suff*, l'ajout de l'heuristique *dern-occ* étant une variante immédiate. Un exemple d'exécution de l'algorithme est montré figure 3.4.

```
LS-SANS-MÉMOIRE-FAIBLE(x, m, bon-suff, y, n)
 1  j ← m − 1
 2  tantque j < n faire
 3          i ← m − 1
 4          tantque i ≥ 0 et x[i] = y[j − m + 1 + i] faire
 5                  i ← i − 1
 6          SIGNALER-SI(i < 0)
 7          si i < 0 alors
 8                  j ← j + pér(x)
 9          sinon j ← j + bon-suff[i]
```

Théorème 3.2
L'algorithme LS-SANS-MÉMOIRE-FAIBLE *localise toutes les occurrences du mot x dans le texte y.*

Preuve Par définition de la table *bon-suff* et de la fonction *pér*, tous les décalages appliqués par l'algorithme LS-SANS-MÉMOIRE-FAIBLE sont valides. L'algorithme ne peut donc manquer aucune occurrence du mot x dans le texte y. ∎

3.2 Temps de recherche

Nous montrons dans cette partie que l'algorithme LS-SANS-MÉMOIRE-FAIBLE effectue au plus $4n$ comparaisons entre des lettres du mot et des lettres du texte lorsqu'il est utilisé pour localiser un mot x tel que $pér(x) > m/3$ dans un texte de longueur n.

Nous commençons par montrer trois résultats techniques qui sont à la base de la preuve.

Lemme 3.3
Soient x un mot, y un texte, v un mot primitif et k un entier tels que $v^2 \preceq_{suff} x$, $y = v^k$ et $k \geq 2$. Durant l'exécution de l'opération LS-SANS-MÉMOIRE-FAIBLE$(x, m, bon\text{-}suff, y, n)$*, s'il existe une tentative T_0 à une position j_0 sur y qui n'est pas de la forme $\ell|v| - 1$ ($\ell \in \mathbf{N}$), celle-ci est suivie, immédiatement ou non, d'une tentative à la position*

$$j = \min\{h : h = \ell|v| - 1, h > j_0, \ell \in \mathbf{N}\} \ .$$

Preuve Puisque v est primitif, au plus $|v|$ comparaisons sont effectuées durant la tentative T_0. Soit $a = x[i]$ la lettre du mot qui provoque la comparaison négative ($b = y[j_0 - m + 1 + i]$ et $a \neq b$). Soit $d_0 = j - j_0$. La condition $Cs(i, d_0)$ est satisfaite : $d_0 \leq i$ et $x[i - d_0 + 1 \,..\, m - d_0 - 1] \preceq_{suff} x$. De même pour $Co\text{-}aff(i, d_0)$: $d_0 \leq i$ et $b = x[i - d_0] \neq x[i] = a$. Il s'ensuit que $bon\text{-}suff[i] \leq d_0$ (voir figure 3.5).

Si $bon\text{-}suff[i] < d_0$, la tentative T_1, qui suit immédiatement T_0, est à une position j_1 sur le texte y telle que $0 < d_1 = j - j_1 < d_0$. Le raisonnement appliqué à la tentative T_0 s'applique donc à la tentative T_1. Une suite finie de telles tentatives mène ainsi à une tentative à la position j. ∎

Soit T une tentative à la position j sur y. On suppose que les propriétés suivantes sont remplies : $bz \preceq_{suff} y[0 \,..\, j]$, $az \preceq_{suff} x$, $a \neq b$, $z = wv^k$, $w \prec_{suff} v$, $aw \preceq_{suff} x$, $k \geq 2$ et v primitif. Ces propriétés sont des hypothèses des deux lemmes qui suivent ainsi que de leur corollaire.

Lemme 3.4
Sous les hypothèses ci-dessus, il n'y a pas de tentative aux positions $j - \ell|v|$, $1 \leq \ell \leq k - 1$, avant la tentative T.

Figure 3.5 Élément de la preuve du lemme 3.3. On recherche $x = \mathtt{a}(\mathtt{aaab})^2$ dans $y = (\mathtt{aaab})^4$. À la suite de la tentative à la position (droite) 8, en 3 décalages (chacun de longueur 1), on arrive à une tentative à la position 11 qui correspond à une position droite d'un facteur `aaab` dans y, ce qui recale la recherche sur la période de y.

Preuve Supposons, par l'absurde, qu'il y ait eu une tentative à une position $j_0 = j - \ell_0|v|$ pour un certain ℓ_0 tel que $1 \leq \ell_0 \leq k-1$. On aurait $bwv^{k-\ell_0} \preceq_{suff} y[0\,..\,j_0]$ et $awv^{k-\ell_0} \preceq_{suff} x$. Pour i_0 tel que $i_0 = m - |w| - (k-\ell_0)|v|$, on a alors $d_0 = bon\text{-}suff[i_0] > \ell_0|v|$.

Tout décalage plus petit et non multiple de $|v|$ contredirait le fait que v est primitif. Tout décalage plus petit et multiple de $|v|$ alignerait une lettre a dans le mot en face de la lettre b dans le texte. Il s'ensuit que le décalage appliqué après une tentative à la position $j_0 = j - \ell_0|v|$ est plus long que ℓ_0. D'où la contradiction (voir figure 3.6). ∎

Lemme 3.5
Sous les hypothèses ci-dessus, avant la tentative T, il n'y a pas de tentative aux positions ℓ telles que $j - |z| + |v| \leq \ell \leq j - |v|$.

Preuve Du lemme 3.4, on déduit qu'il ne peut y avoir de tentative aux positions $j - \ell|v|$ pour $1 \leq \ell \leq k-1$. Et du lemme 3.3, que toute tentative à une autre position comprise entre $j - |z| + |v|$ et $j - |v|$ est suivie (immédiatement ou non) par une tentative à une position $j - \ell|v|$ avec $1 \leq \ell \leq k-1$. Ce qui donne le résultat. ∎

Corollaire 3.6
Sous les hypothèses ci-dessus, avant la tentative T, au plus $3|v|-3$ lettres du facteur z de y ont été comparées à des lettres de x.

Preuve D'après le lemme 3.5, les tentatives précédant la tentative T et dans lesquelles les lettres de z ont été comparées n'ont pu intervenir qu'à des positions se trouvant dans les intervalles $[j - |z| + 1, j - |z| + |v| - 1]$

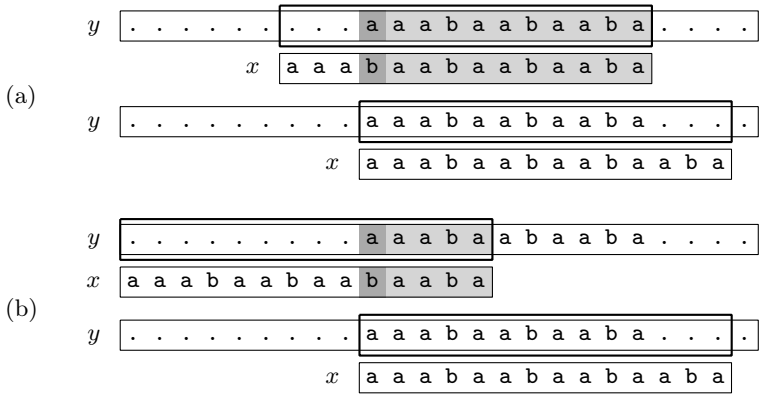


Figure 3.6 Élément de la preuve du lemme 3.4. **(a)** Soit j la position sur y de la tentative courante. On détecte le suffixe $\mathtt{a}(\mathtt{aba})^3$ de x. Une comparaison négative survient entre les lettres `b` et `a` qui précèdent ce facteur dans le mot et le texte respectivement. Le décalage à appliquer est de longueur 3. **(b)** Si la tentative ici décrite avait existé préalablement, elle aurait conduit à la même situation finale que celle de la partie (a). Ce qui contredirait l'existence de la tentative en j.

d'une part et $[j-|v|+1, j-1]$ d'autre part. Pour le premier intervalle, le préfixe de z soumis à comparaisons est de longueur maximale $|v|-1$. Pour le second qui contient $|v|-1$ positions, le facteur de z éventuellement soumis à comparaisons est $z[|z|-2|v|+1\,..\,|z|-2]$. En effet, le nombre de comparaisons effectuées lors d'une tentative à une position de l'intervalle $[j-|v|+1, j-1]$ est strictement inférieur à $|v|$ puisque v est primitif. Le nombre d'occurrences de lettres comparées est donc majoré par la somme des longueurs des deux facteurs de z considérés, c'est-à-dire $3(|v|-1)$. Ce que l'on voulait montrer. ∎

Théorème 3.7
Lors de la localisation d'un mot x de longueur m tel que pér$(x) > m/3$ dans un texte y de longueur n, l'algorithme LS-SANS-MÉMOIRE-FAIBLE *effectue moins de $4n$ comparaisons entre des lettres de x et des lettres de y.*

Preuve Pour une tentative T à la position j, on note t le nombre d'occurrences de lettres comparées pour la première fois au cours de cette tentative, et d la longueur du décalage qui suit. Nous allons borner le nombre de comparaisons effectuées lors de la tentative T par $3d+t$.

Posons $z = lsc(x, y[0\,..\,j])$.

Si $|z| \leq 3d$, le nombre de comparaisons effectuées lors de la tentative T est au plus $|z|+1$ et la lettre $y[j]$ n'avait pas été comparée avant la tentative T. Donc $|z|+1 \leq 3d+1$.

Si $|z| > 3d$, cela implique $z = wv^k$, $bz \preceq_{suff} y[0\,..\,j]$, $az \preceq_{suff} x$, $a \neq b$, $k \geq 1$, $w \prec_{suff} v$ et v primitif, sinon cela contredirait l'hypothèse $pér(x) > m/3$. De plus $k \geq 2$, $aw \prec_{suff} v$ et $d \geq |v|$. Donc, par le corollaire 3.6, au plus $3|v| - 3$ lettres de z ont été comparées avant la tentative T. Il s'ensuit que $t \geq |z| - 3|v| + 3 \geq |z| - 3d + 3$. Le nombre de comparaisons effectuées lors de la tentative T, $|z| + 1$, qui est inférieur à $3d + |z| - 3d + 3 = |z| + 3$, est donc bien inférieur à $3d + t$.

Puisque la somme des longueurs de tous les décalages est inférieure à n et que le nombre de lettres pouvant être comparées pour la première fois est inférieur à n, le résultat suit. ∎

La borne $4n$ du théorème précédent n'est pas optimale. En fait, on peut montrer le résultat suivant que nous énonçons sans preuve.

Théorème 3.8
Lors de la localisation d'un mot non périodique x de longueur m (c'est-à-dire tel que $pér(x) > m/2$) dans un texte y de longueur n, l'algorithme LS-SANS-MÉMOIRE-FAIBLE *effectue au plus $3n$ comparaisons entre des lettres de x et des lettres de y.* ∎

Le théorème ne s'applique pas au cas où le mot x est périodique. Pour ces mots-là, il suffit de modifier légèrement l'algorithme LS-SANS-MÉMOIRE-FAIBLE afin d'obtenir un algorithme linéaire. En effet, l'indice i peut continuer à varier de $m - 1$ à 0 sauf lorsqu'une occurrence vient d'être signalée auquel cas il peut varier de $m - 1$ à $m - pér(x)$. L'algorithme LS-FAIBLE-LINÉAIRE ci-dessous implante cette technique, appelée « mémorisation de préfixe ».

LS-FAIBLE-LINÉAIRE$(x, m, \mathit{bon\text{-}suff}, y, n)$
```
 1  ℓ ← 0
 2  j ← m − 1
 3  tantque j < n faire
 4          i ← m − 1
 5          tantque i ≥ ℓ et x[i] = y[j − m + 1 + i] faire
 6                  i ← i − 1
 7          SIGNALER-SI(i < ℓ)
 8          si i < ℓ alors
 9                  ℓ ← m − pér(x)
10                  j ← j + pér(x)
11          sinon ℓ ← 0
12                  j ← j + bon-suff[i]
```

La borne donnée dans le théorème 3.8 est quasi optimale comme le montre l'exemple suivant. Soient $x = \mathtt{a}^{k-1}\mathtt{b}\mathtt{a}^{k-1}$ et $y = \mathtt{a}^{k-1}(\mathtt{aba}^{k-1})^{\ell}$ avec $k \geq 2$ (on a alors $m = 2k - 1$ et $n = \ell(k+1) + (k-1)$). Sur chacun des $\ell - 1$ premiers facteurs $\mathtt{aba}^{k-1}$ (de longueur $k + 1$) de y, le nombre

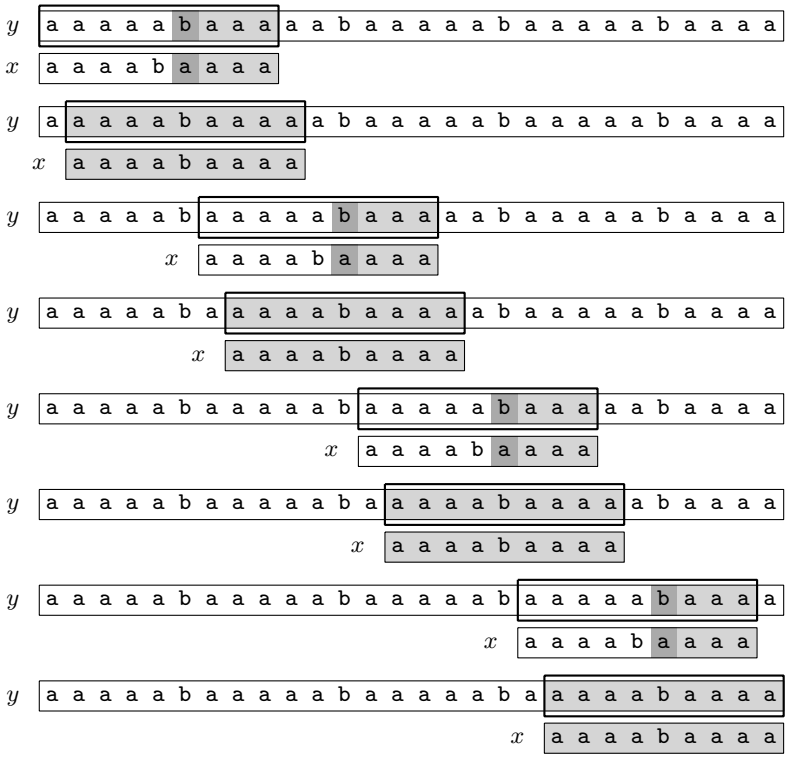


Figure 3.7 Illustration de la borne du théorème 3.8 avec $x = \mathtt{a}^4\mathtt{ba}^4$ et $y = \mathtt{a}^4(\mathtt{aba}^4)^4$. Le mot x est de longueur 9, le texte y de longueur 28, et cinquante-deux comparaisons sont effectuées. Pour chaque facteur `abaaaa` (de longueur 6) du texte, treize comparaisons sont effectuées.

de comparaisons effectuées est $(k-1)+(k+1)+(k-2) = 3k-2$. Sur le dernier facteur de la sorte, $(k-1)+(k+1) = 2k$ comparaisons sont effectuées. Et sur le préfixe de longueur $k-1$ de y, $k-2$ comparaisons sont effectuées. Au total, l'algorithme LS-SANS-MÉMOIRE-FAIBLE effectue

$$\frac{3k-2}{k+1}(n-k+1) = \left(n - \frac{m-1}{2}\right)\left(3 - \frac{10}{m+3}\right)$$

comparaisons. La figure 3.7 illustre la borne avec les valeurs $k = 5$ et $\ell = 4$.

Corollaire 3.9
L'algorithme LS-SANS-MÉMOIRE-FAIBLE *effectue la localisation de la première occurrence d'un mot de longueur m dans un texte de longueur n en temps $O(n)$ et en espace $O(m)$.*

Preuve Le résultat est une conséquence du théorème 3.2 et du théorème 3.7 (ou du théorème 3.8). ∎

3.3 Calcul de la table du bon suffixe

Nous considérons dans cette section le prétraitement que doit subir le mot x pour effectuer sa localisation au moyen de l'algorithme LS-SANS-MÉMOIRE-FAIBLE. Celui-ci consiste au calcul de la table du bon suffixe, *bon-suff*, et de la période de x. Ce dernier calcul est contenu dans le premier car on a déjà remarqué que $pér(x) = bon\text{-}suff[0]$. Deux autres calculs de la table *bon-suff* sont proposés en exercice (exercices 3.9 et 3.10).

Algorithme

Pour calculer la table *bon-suff* qui figure dans l'algorithme LS-SANS-MÉMOIRE-FAIBLE, on utilise la table *suff* définie sur le mot x comme suit. Pour $i = 0, 1, \ldots, m-1$:

$$suff[i] = |lsc(x, x[0\,..\,i])| \ ,$$

c'est-à-dire que $suff[i]$ est la longueur maximale des suffixes de x qui apparaissent à la position droite i sur x. La table *suff* est l'analogue, obtenue en inversant le sens de lecture, de la table *préf* de la section 1.6. Cette dernière fournit les longueurs maximales des préfixes de x commençant en chacune de ses positions. La figure 3.8 donne les deux tables *suff* et *bon-suff* pour le mot $x =$ `aaacababa`.

Le calcul de la table *suff* est effectué par l'algorithme SUFFIXES ci-dessous qui est directement adapté de l'algorithme PRÉFIXES calculant la table *préf* (voir section 1.6).

(a)

i	0	1	2	3	4	5	6	7	8
$x[i]$	a	a	a	c	a	b	a	b	a
$suff[i]$	1	1	1	0	1	0	3	0	9
$bon\text{-}suff[i]$	8	8	8	8	8	2	8	4	1

(b)

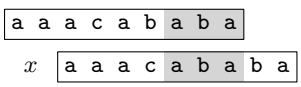


Figure 3.8 On considère le mot $x =$ `aaacababa`. **(a)** Les valeurs des tables *suff* et *bon-suff*. **(b)** On a $suff[6] = 3$. Cela indique que le plus long suffixe de x se terminant à la position 6 est `aba`, lequel mot est de longueur 3. Comme $suff[6] = 3$, on a $bon\text{-}suff[9-1-3] = 9-1-6 = 2$, valeur qui est calculée à la ligne 8 de l'algorithme BON-SUFFIXE.

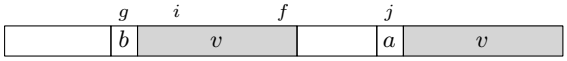


Figure 3.9 Variables i, j, f et g de l'algorithme SUFFIXES. La boucle principale admet pour invariants : $v = lsc(x, x[0\,..\,f])$ et donc $a \neq b$ $(a, b \in A)$, $j = g + m - 1 - f$, ainsi que $i < f$. Le schéma correspond à la situation dans laquelle $g < i$.

```
SUFFIXES(x, m)
 1  g ← m − 1
 2  suff[m − 1] ← m
 3  pour i ← m − 2 à 0 pas −1 faire
 4        si i > g et suff[i + m − 1 − f] < i − g alors
 5              suff[i] ← suff[i + m − 1 − f]
 6        sinonsi i > g et suff[i + m − 1 − f] > i − g alors
 7              suff[i] ← i − g
 8        sinon (f, g) ← (i, min{g, i})
 9              tantque g ≥ 0 et x[g] = x[g + m − 1 − f] faire
10                    g ← g − 1
11              suff[i] ← f − g
12  retourner suff
```

Le schéma de la figure 3.9 décrit les variables de l'algorithme SUFFIXES et les invariants de sa boucle principale. La preuve de correction de l'algorithme est similaire à celle de PRÉFIXES.

Nous pouvons maintenant formuler l'algorithme BON-SUFFIXE qui calcule la table *bon-suff* au moyen de la table *suff*.

```
BON-SUFFIXE(x, m, suff)
 1  j ← 0
 2  pour i ← m − 2 à −1 pas −1 faire
 3        si i = −1 ou suff[i] = i + 1 alors
 4              tantque j < m − 1 − i faire
 5                    bon-suff[j] ← m − 1 − i
 6                    j ← j + 1
 7  pour i ← 0 à m − 2 faire
 8        bon-suff[m − 1 − suff[i]] ← m − 1 − i
 9  retourner bon-suff
```

Le schéma de la figure 3.10 présente les invariants de la seconde boucle de BON-SUFFIXE. Nous montrons que cet algorithme calcule bien la table *bon-suff*. Pour cela, nous commençons par énoncer deux lemmes intermédiaires.

Lemme 3.10
Pour $0 \leq i \leq m - 2$, *si* $suff[i] = i + 1$ *alors, pour* $0 \leq j < m - 1 - i$, $bon\text{-}suff[j] \leq m - 1 - i$.

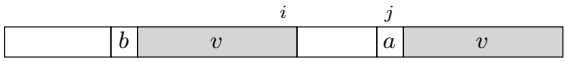


Figure 3.10 Variables i et j de l'algorithme BON-SUFFIXE. Situation dans laquelle $\mathit{suff}[i] < i+1$. La boucle des lignes 7–8 admet les invariants suivants : $v = \mathit{lsc}(x, x[0\,..\,i])$ et donc $a \neq b$ $(a, b \in A)$ et $\mathit{suff}[i] = |v|$. On en déduit $\mathit{bon\text{-}suff}[j] \leq m-1-i$ avec $j = m-1-\mathit{suff}[i]$.

Preuve L'hypothèse $\mathit{suff}[i] = i+1$ est équivalente à $x[0\,..\,i] \preceq_{\mathit{suff}} x$. Ainsi $m - \mathit{suff}[i] = m-1-i$ est une période de x. Soit j un indice qui satisfait $0 \leq j < m-1-i$. La condition $\mathit{Cs}(j, m-1-i)$ est satisfaite car $m-1-i > j$ et $x[0\,..\,m-(m-1-i)-1] = x[0\,..\,i] \preceq_{\mathit{suff}} x$. Il en va de même pour la condition $\mathit{Co\text{-}aff}(j, m-1-i)$ car $m-1-i > j$. Cela montre, par définition de *bon-suff*, que $\mathit{bon\text{-}suff}[j] \leq m-1-i$ comme annoncé. ∎

Lemme 3.11
Pour $0 \leq i \leq m-2$, on a $\mathit{bon\text{-}suff}[m-1-\mathit{suff}[i]] \leq m-1-i$.

Preuve Si $\mathit{suff}[i] < i+1$, la condition $\mathit{Cs}(m-1-\mathit{suff}[i], m-1-i)$ est satisfaite car on a d'une part $m-1-i \leq m-1-\mathit{suff}[i]$ et d'autre part $x[i-\mathit{suff}[i]+1\,..\,i] = x[m-1-\mathit{suff}[i]+1\,..\,m-1]$. De plus, la condition $\mathit{Co\text{-}aff}(m-1-\mathit{suff}[i], m-1-i)$ est également satisfaite puisque $x[i-\mathit{suff}[i]] \neq x[m-1-\mathit{suff}[i]]$ par définition de *suff*. Donc $\mathit{bon\text{-}suff}[m-1-\mathit{suff}[i]] \leq m-1-i$.

Maintenant si $\mathit{suff}[i] = i+1$, par le lemme 3.10, on a en particulier pour $j = m-1-\mathit{suff}[i] = m-i-2$, l'inégalité $\mathit{bon\text{-}suff}[j] \leq m-1-i$. Ce qui termine la preuve. ∎

Proposition 3.12
L'algorithme BON-SUFFIXE *calcule la table bon-suff du mot x au moyen de la table suff du même mot.*

Preuve Il faut montrer, pour chaque indice j, $0 \leq j < m$, que la valeur finale d attribuée à $\mathit{bon\text{-}suff}[j]$ par BON-SUFFIXE est la valeur minimale qui satisfait les conditions $\mathit{Cs}(j, d)$ et $\mathit{Co\text{-}aff}(j, d)$.

Supposons d'abord que d résulte d'une affectation au cours de l'exécution de la boucle des lignes 2–6. Ainsi la première partie de la condition *Cs* n'est pas satisfaite. On vérifie alors au moyen du lemme 3.10 que d est la valeur minimale qui satisfait la seconde partie de la condition $\mathit{Cs}(j, d)$. Dans ce cas, $d = m-1-i$ pour une valeur i qui est telle que $\mathit{suff}[i] = i+1$ et $j < m-1-i$. Cette dernière inégalité montre que la condition $\mathit{Co\text{-}aff}(j, d)$ est aussi satisfaite. Cela prouve le résultat dans cette situation, c'est-à-dire $d = \mathit{bon\text{-}suff}[j]$.

Supposons maintenant que d résulte d'une affectation au cours de l'exécution de la boucle des lignes 7–8. On a ainsi $j = m-1-\mathit{suff}[i]$

et $d = m - 1 - i$, et, d'après le lemme 3.11, *bon-suff*$[j] \leq d$. On a aussi $0 < d \leq i$, ce qui montre que les secondes parties des conditions $Cs(j, d)$ et *Co-aff*(j, d) ne peuvent être satisfaites. Comme la quantité $m-1-i$ va en décroissant pendant l'exécution de la boucle, d est la plus petite valeur de $m-1-i$ pour laquelle $j = m-1-$*suff*$[i]$. On a donc $d =$ *bon-suff*$[j]$. Ce qui termine la preuve. ∎

Complexité du calcul

Le temps de préparation de la table *bon-suff*, utilisée par l'algorithme LS-SANS-MÉMOIRE-FAIBLE, est linéaire. On peut noter que ce temps ne dépend pas de la taille de l'alphabet.

Proposition 3.13
L'algorithme SUFFIXES *appliqué à un mot de longueur m s'exécute en temps $O(m)$ et nécessite un espace supplémentaire constant.*

Preuve La preuve se déduit de celle qui concerne l'algorithme PRÉFIXES dans le section 1.6. Rappelons simplement que l'ensemble des exécutions de la boucle aux lignes 9–10 prend un temps $O(m)$ car les valeurs de g ne font que décroitre. L'exécution des autres instructions prend un temps constant par valeur de i, soit encore globalement $O(m)$.

L'algorithme a besoin d'espace supplémentaire pour quelques variables entières uniquement, soit un espace constant. ∎

Proposition 3.14
L'algorithme BON-SUFFIXE *appliqué à un mot de longueur m s'exécute en temps $O(m)$ (même en incluant le temps de calcul de la table intermédiaire suff) et nécessite un espace supplémentaire $O(m)$.*

Preuve L'espace nécessaire au calcul (en sus du mot x et de la table *suff*) comprend la table *bon-suff* et quelques variables entières. Soit un espace $O(m)$.

L'exécution de la boucle aux lignes 2–6 prend un temps $O(m)$ car chaque opération s'exécute en temps constant par valeur de i ou par valeur de j, et car ces variables prennent $m + 1$ valeurs distinctes.

La boucle des lignes 7–8 s'exécute elle aussi en temps $O(m)$, ce qui montre le résultat. Inclure le temps de calcul de la table *suff* donne la même conclusion d'après la proposition 3.13. ∎

3.4 Automate du meilleur facteur

Nous montrons dans cette section que la fonction de décalage du meilleur facteur – fonction utilisée dans l'algorithme de localisation d'un mot,

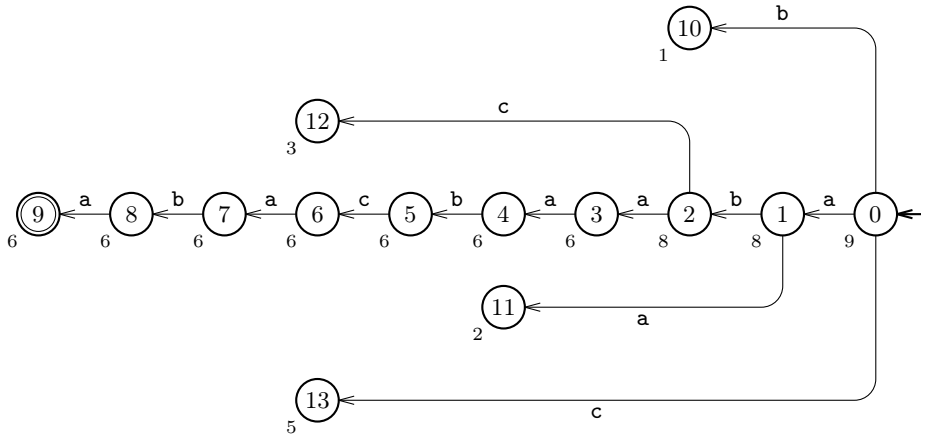


Figure 3.11 L'automate du meilleur facteur de $x =$ `abacbaaba`. Les sorties des états indiquent la longueur du décalage à appliquer à la fenêtre lorsque l'état ne possède aucun successeur ou alors lorsqu'aucune des transitions sortant de l'état courant n'a d'étiquette identique à la lettre courante dans la fenêtre.

LS-SANS-MÉMOIRE, présenté section 3.1 – peut être implantée en espace $O(m)$. Le support retenu pour exprimer cette implantation est celui d'un automate. Au-delà du complément théorique, on ne met en évidence un gain quelconque dans aucune démonstration sur les complexités asymptotiques.

On appelle ***automate du meilleur facteur*** du mot x l'automate dont :

- les états sont le mot vide ε et les facteurs de x de la forme cz avec $c \in A$ et $z \prec_{suff} x$;
- l'état initial est le mot vide ε ;
- l'état terminal est x ;
- les flèches sont de la forme (z, c, cz).

De plus, chaque état est muni d'une sortie qui correspond à la longueur d'un décalage de la fenêtre utilisée lors de la localisation de x. La définition de la sortie est donnée ci-dessous. Elle diffère selon que l'état est un suffixe ou non de x :

1. La sortie d'un état z avec $z \preceq_{suff} x$ est la longueur du plus court suffixe z' de x tel que $x \preceq_{suff} zz'$.
2. La sortie d'un état de la forme cz, $c \in A$ et $z \preceq_{suff} x$, avec $cz \not\preceq_{suff} x$, est la longueur du plus court suffixe z' de x tel que $czz' \preceq_{suff} x$.

Un exemple d'automate du meilleur facteur est montré figure 3.11.

En reprenant les termes de la section 3.1 dans le cas d'une comparaison négative pour une tentative à une position j sur le texte, c'est-à-dire en notant i la position courante sur x ($i \geq 0$), $b = y[j - m + 1 + i]$

$(b \neq x[i])$, $z = x[i+1\,..\,m-1]$, puis en notant δ la fonction de transition de l'automate, on a :

$$\textit{meil-fact}(i,b) = \begin{cases} \text{sortie de } \delta(z,b) & \text{si } \delta(z,b) \text{ est défini ,} \\ \text{sortie de } z & \text{sinon .} \end{cases}$$

Quant à l'algorithme de localisation qui utilise l'automate, il s'exprime comme suit.

```
LA-MEILLEUR-FACTEUR(x, m, y, n)
 1  soit M l'automate du meilleur facteur de x
 2  j ← m − 1
 3  tantque j < n faire
 4        p ← initial[M]
 5        k ← m − 1
 6        tantque Succ[p] ≠ ∅
            et CIBLE(p, y[j − m + 1 + k]) ≠ NIL faire
 7              p ← CIBLE(p, y[j − m + 1 + k])
 8              k ← k − 1
 9        SIGNALER-SI(terminal[p])
10        j ← j + sortie[p]
```

L'intérêt de l'automate du meilleur facteur est triple : il synthétise parfaitement tentatives et décalages ; sa taille est $O(m)$; sa construction peut se réaliser en temps $O(m)$. Pour la taille, on peut montrer directement que le nombre d'états de l'automate qui ne sont pas des suffixes de x (ou, de manière équivalente, des flèches qui entrent dans ces états, puisque le degré entrant de tous les états, hormis l'état initial, vaut 1) est au plus égal à $m-1$ (voir exercice 3.5). Une autre preuve de cette borne est donnée dans la preuve du théorème 3.16.

Théorème 3.15
La taille de l'automate du meilleur facteur de tout mot x de longueur m est $O(m)$.

Preuve L'automate possède $m+1$ états qui sont des suffixes de x et au plus $m-1$ états qui n'en sont pas (voir exercice 3.5). Il possède également m flèches qui entrent dans des états qui sont des suffixes de x et au plus $m-1$ flèches qui entrent dans des états qui ne sont pas des suffixes de x. Sa taille totale est donc $O(m)$. ∎

Dans les paragraphes suivants, on détaille une méthode de construction de l'automate dont on montrera qu'elle peut s'effectuer en temps $O(m)$.

Notons M_x la structure qui correspond à l'automate du meilleur facteur de x mais dans laquelle les sorties des états z qui sont des suffixes de x (états de type 1) ne sont pas définies. Remarquons maintenant que

pour ces états, la sortie est la plus petite des périodes de x supérieure à $|x| - |z|$. Il s'ensuit que si l'on dispose de M_x et, par exemple, de la table des longueurs des bords des suffixes non vides (analogue à la table des bords pour les préfixes de la section 1.6), le calcul des sorties des états de type 1 peut se faire en temps $O(m)$. Il reste donc à construire M_x.

La construction de M_x peut se faire de façon séquentielle sur x, en procédant par suffixes de longueur croissante. La structure M_ε se réduit à l'état ε, à la fois initial et terminal. Soit at un suffixe de x avec $a \in A$ et $t \prec_{suff} x$, et supposons M_t construit. La structure M_{at} contient :

- les états et les flèches de M_t, l'état t ne portant pas la marque des états terminaux ;
- l'état terminal at, de type 1, et la flèche (t, a, at) ;
- les états de type 2 de la forme az, avec $z \prec_{suff} t$, dont la sortie est $|t| - |z|$, et les flèches associées de la forme (a, z, az).

Intéressons-nous au calcul des objets du dernier point. L'état az étant du type 2, z est un bord de t. De plus, la longueur de z est nécessairement supérieure à celle du mot $Bord(at)$. En effet, dans le cas contraire, la sortie de az serait de longueur inférieure à $|Bord(at)| - 1 - |z|$, quantité strictement inférieure à $|t| - |z|$; ce qui est contraire à l'hypothèse. Maintenant, parmi tous les bords z de t de longueur supérieure à $|Bord(at)|$, seuls ceux pour lesquels l'état az n'est pas déjà dans la structure sont à insérer dans celle-ci, avec $|t| - |z|$ comme sortie, les flèches associées (z, a, az) étant insérées à l'avenant. Remarquons que tester la présence de tels états dans M_{at} revient à tester s'il sort de z une transition étiquetée par a. Ajoutons également que l'accès aux bords de t est immédiat dès lors que, en parallèle à la construction de M_{at}, est calculée la table des longueurs des bords des suffixes de x.

Théorème 3.16
La construction de l'automate du meilleur facteur de tout mot x de longueur m se réalise en temps $O(m)$ si l'on dispose d'un espace supplémentaire $O(m + \operatorname{card} A)$.

Preuve La construction de l'automate proposée ci-dessus utilise la table des longueurs des bords des suffixes de x. Celle-ci est calculée en parallèle de la construction. L'espace supplémentaire pour la mémoriser est $O(m)$.

Avec les notations précédentes, les seuls bords z de t qui sont retenus comme candidats à une possible insertion de l'état az dans la structure M_{at} sont les suffixes de t qui sont précédés d'une lettre distincte de a. Ils correspondent donc aux comparaisons négatives entre lettres effectuées lors du calcul de la table ; on sait que leur nombre est au plus égal à $m - 1$ (voir exercice 1.22). Cela confirme au passage les bornes données plus haut quant au nombre d'états qui ne sont pas des suffixes de x et au nombre de flèches qui entrent dans ces états.

Dans le modèle comparaisons, un test préalable à chaque insertion, puis, le cas échéant, l'insertion elle-même prend un temps $O(\log m)$; ce qui donnerait une complexité $O(m \times \log m)$. On peut en revanche ajouter sans test les états et flèches dans un premier temps : la surstructure de M_x ainsi obtenue est toujours de taille $O(m)$ d'après le résultat précédent. Puis, à l'aide d'une table sur l'alphabet, on élague en éliminant les flèches indésirables (pour une lettre donnée, seule la flèche à sortie minimale est conservée). Ce qui s'effectue en temps $O(m)$.

Enfin, comme mentionné plus haut, le calcul des sorties des états qui sont des suffixes de x se fait en temps $O(m)$ avec la table des longueurs des bords des suffixes de x. Ce qui achève la preuve. ∎

La construction effective de la fonction *meil-fact* par l'intermédiaire de l'automate du meilleur facteur de x est laissée en exercice (exercice 3.8). On déduit des preuves ci-dessus un autre calcul de la table du bon suffixe que celui présenté section 3.3 (voir exercice 3.9).

3.5 Localisation avec une mémoire

Cette section présente un algorithme moins « amnésique » que celui de la section 3.1. Durant la recherche, il retient au moins une information sur les coïncidences précédentes : celle du dernier suffixe du mot rencontré dans le texte. Il s'agit d'une technique nommée « mémorisation de facteur » qui prolonge la technique de mémorisation de préfixe mise en œuvre par l'algorithme LS-SANS-MÉMOIRE-FAIBLE-LINÉAIRE. Elle a besoin d'un espace supplémentaire constant par rapport à l'algorithme LS-SANS-MÉMOIRE, ainsi que d'une adaptation de son fonctionnement. En contrepartie, le comportement de l'algorithme n'est plus quadratique et moins de $2n$ comparaisons sont effectuées afin de rechercher toutes les occurrences d'un mot dans un texte de longueur n. En outre, la phase de préparation de cet algorithme est la même que celle de l'algorithme LS-SANS-MÉMOIRE ou de sa version LS-SANS-MÉMOIRE-FAIBLE.

Phase de recherche

Après chacun de ses décalages, l'algorithme LS-SANS-MÉMOIRE oublie toutes les informations disponibles lors des tentatives antérieures. On améliore le comportement de cet algorithme en prenant en compte la dernière occurrence d'un suffixe du mot x reconnue dans le texte y. La mémorisation de ce facteur du texte reconnu lors de la tentative précédente présente deux avantages pour la tentative en cours :

- elle permet éventuellement d'effectuer un « saut » au-dessus de ce facteur ;
- elle permet éventuellement d'allonger le décalage qui suit.

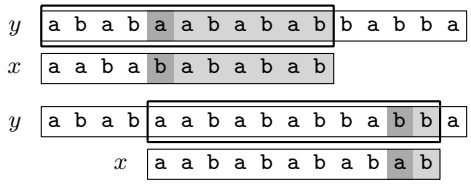


Figure 3.12 Conditions d'un turbo-décalage. Lors de la tentative à la position 10, on reconnait le suffixe `ababab` du mot. On décale de 4 positions comme pour l'algorithme LS-SANS-MÉMOIRE (on note que le suffixe `abababababab` du mot admet 4 comme période). Ainsi le facteur `aababab` de y coïncide-t-il avec le facteur de x aligné avec lui. Lors de la tentative suivante, on reconnait le suffixe `b` uniquement. Les lettres $y[9] = \texttt{a}$ et $y[13] = \texttt{b}$ montrent que cette portion du texte n'a pas 4 comme période. Donc le suffixe `abababababab` du mot, qui admet 4 comme période, ne peut être aligné simultanément avec $y[9]$ et $y[13]$. Ce qui entraine que le décalage à appliquer soit au moins de longueur $|\texttt{ababab}| - |\texttt{b}| = 5$.

Ces possibilités sont partiellement exploitées dans l'algorithme de cette section. La mémorisation d'un seul facteur est effectuée dans un cas bien précis et l'allongement du décalage qu'elle procure est réalisé par ce que l'on désigne sous le terme de ***turbo-décalage***.

On décrit plus précisément ces techniques. La situation générale lors d'une tentative T de la phase de recherche de l'algorithme LS-UNE-MÉMOIRE est illustrée figure 3.12. Lors de la tentative précédente T', à la position j', un suffixe z' du mot a été reconnu dans le texte et un décalage de longueur $d' = \mathit{meil\text{-}fact}(m-1-|z'|, y[j'-|z'|])$ a été appliqué.

Lors de la tentative courante T à la position $j = j' + d'$, un saut au-dessus du facteur z' de y peut intervenir si le suffixe du mot de longueur d' est reconnu dans y à cette position. Dans ce cas, il est inutile de comparer les facteurs z' du mot et du texte puisque, par la définition même du décalage, il est certain qu'ils coïncident. Un turbo-décalage peut être appliqué si le suffixe z reconnu lors de la tentative courante est plus court que z'. La longueur du turbo-décalage est $|z'| - |z|$.

Dans le cas où la valeur du turbo-décalage est strictement supérieure à $\mathit{meil\text{-}fact}(m - |z|, y[j - |z|])$, on constate que le décalage à appliquer après la tentative courante peut être, en plus, strictement plus long que $|z|$. La mémorisation d'un facteur n'intervient qu'après un décalage donné par *meil-fact*, la preuve de validité étant basée sur des arguments de périodicité.

Nous pouvons maintenant donner le code de l'algorithme LS-UNE-MÉMOIRE. Le code fait référence à la fonction *meil-fact* de la section 3.4. Mais il est aussi possible d'utiliser la table *bon-suff* pour calculer la longueur des décalages.

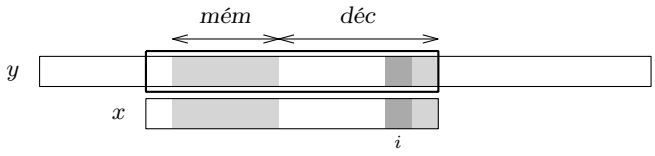


Figure 3.13 Variables i, *mém* et *déc* en ligne 10 de l'algorithme LS-UNE-MÉMOIRE. Les parties en gris clair coïncident, et celles en gris foncé diffèrent.

```
LS-UNE-MÉMOIRE(x, m, y, n)
 1  déc ← 0
 2  mém ← 0
 3  j ← m − 1
 4  tantque j < n faire
 5        i ← m − 1
 6        tantque i ≥ 0 et x[i] = y[j − m + 1 + i] faire
 7              si i = m − déc alors
 8                    i ← i − mém − 1      ▷ Saut
 9              sinon i ← i − 1
10        SIGNALER-SI(i < 0)
11        si i < 0 alors
12              déc ← pér(x)
13              mém ← m − déc
14        sinon turbo ← mém − m + 1 + i
15              si turbo ≤ meil-fact(i, y[j − m + 1 + i]) alors
16                    déc ← meil-fact(i, y[j − m + 1 + i])
17                    mém ← min{m − déc, m − i}
18              sinon déc ← max{turbo, m − 1 − i}
19                    mém ← 0
20        j ← j + déc                      ▷ Décalage
```

Le schéma de la figure 3.13 donne une indication sur la signification des variables.

Théorème 3.17
L'algorithme LS-UNE-MÉMOIRE *localise toutes les occurrences du mot x dans le texte y.*

Preuve Les différences entre les algorithmes LS-SANS-MÉMOIRE et LS-UNE-MÉMOIRE interviennent au niveau du calcul des décalages. Il suffit donc de montrer que le décalage calculé à la ligne 18 est valide. On montre pour commencer que le turbo-décalage de longueur *turbo* est valide. On montre ensuite que le décalage de longueur $m-1-i$ est également valide. Noter que l'instruction de la ligne 18 est exécutée lorsque l'on a $turbo > meil\text{-}fact(i, y[j-m+1+i])$, ce qui implique $turbo > 1$.

La valeur de la variable *mém* est la longueur du suffixe z' reconnu lors de la tentative précédente T'. La longueur du suffixe $z = x[i+1\,..\,m-1]$

reconnu lors de la tentative courante T est $m-1-i$. La valeur de la variable *turbo* est $|z'|-|z|$. Soit $a = x[i]$ la lettre qui précède le suffixe z dans le mot et soit $b = y[j-m+1+i]$ la lettre qui précède l'occurrence correspondante de z dans le texte. Soit $u = x[m-d\,..\,i]$ (on a $z'uz \preceq_{suff} x$). Puisque z est moins long que z', az est un suffixe de z'. Il s'ensuit que les lettres a et b apparaissent à une distance $d = |uz|$ dans le texte. Mais comme le suffixe $z'uz$ du mot a une période $d = |uz|$ (car z' est un bord de $z'uz$), les décalages de longueur inférieure à $|z'|-|z| = turbo$ conduisent à des inégalités. Donc le décalage de longueur *turbo* est valide (voir figure 3.12).

On montre maintenant qu'un décalage de longueur $|z| = m-1-i$ est valide. En effet, posons $\ell = meil\text{-}fact(i, b)$. Par définition de *meil-fact*, on a $x[i-\ell] \neq x[i]$. Comme l'entier ℓ est une période de z, les deux lettres $x[i-\ell]$ et $x[i]$ ne peuvent être toutes les deux alignées avec des lettres de l'occurrence de z en question dans y. On en déduit que le décalage de longueur $|z| = m-1-i$ est valide.

En conclusion des deux points ci-dessus, le décalage de la ligne 18 dont la longueur est le maximum des longueurs de deux décalages valides, est lui aussi valide. Cela termine la preuve. ∎

Deux exemples d'exécution de l'algorithme LS-UNE-MÉMOIRE, l'un utilisant la fonction *meil-fact*, l'autre la table *bon-suff*, sont montrés dans la figure 3.14.

Temps d'exécution de la phase de recherche

Nous allons montrer que l'algorithme LS-UNE-MÉMOIRE a un comportement linéaire dans le pire des cas.

Théorème 3.18
Lors de la recherche de toutes les occurrences d'un mot x de longueur m dans un texte y de longueur n l'algorithme LS-UNE-MÉMOIRE *effectue au plus $2n$ comparaisons de lettres.*

Preuve En reprenant les notations de la preuve du théorème 3.17, on dit que le décalage, de longueur d, appliqué après la tentative T, est court si $2d < |z|+1$ et long sinon.

Nous allons considérer trois types de tentatives :

1. Les tentatives suivies par une tentative effectuant un « saut ».
2. Les tentatives qui ne sont pas de type 1 et qui sont suivies d'un décalage long.
3. Les tentatives qui ne sont pas de type 1 et qui sont suivies d'un décalage court.

L'idée de la preuve est d'amortir les comparaisons avec les décalages.

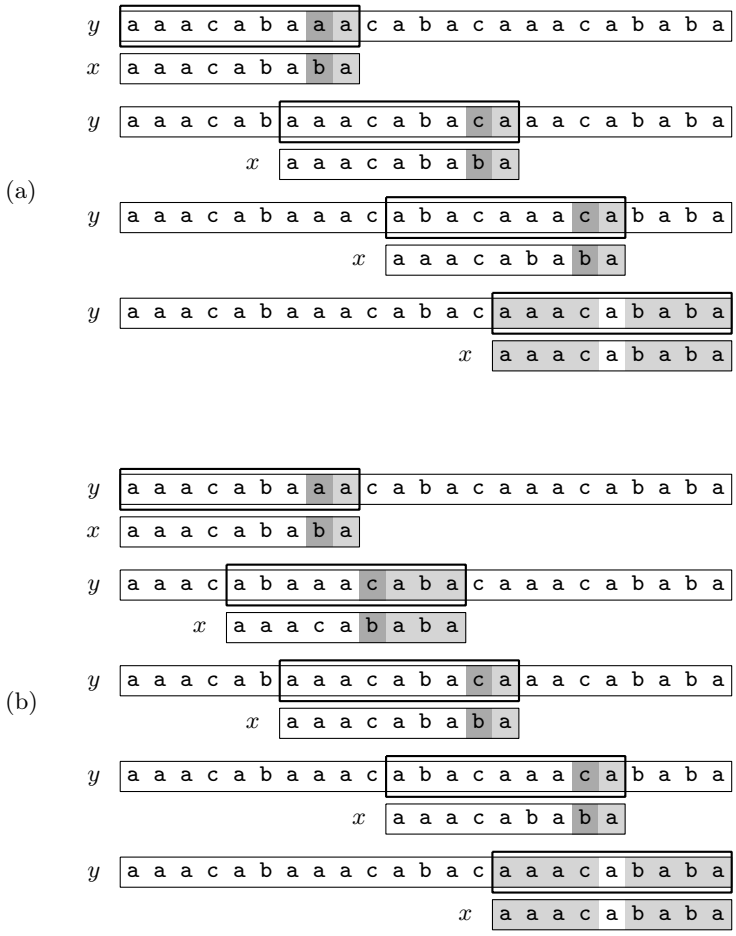


Figure 3.14 Deux exemples d'exécution de l'algorithme LS-UNE-MÉMOIRE. **(a)** Avec utilisation de la fonction *meil-fact*. Dans ce cas, quatorze comparaisons entre lettres du mot et du texte sont effectuées. **(b)** Avec utilisation de la table *bon-suff*. Dans ce cas, dix-huit comparaisons entre lettres du mot et du texte sont effectuées.

Le cout $cout(T)$ d'une tentative T est défini comme suit :

$$cout(T) = \begin{cases} 1 & \text{si } T \text{ est de type 1 ,} \\ 1 + |z| & \text{si } T \text{ est de type 2 ou 3 .} \end{cases}$$

Dans le cas d'une tentative de type 1, le cout correspond au test qui produit l'inégalité. Les autres comparaisons sont reportées sur la tentative suivante. Par conséquent, le nombre total de comparaisons effectuées durant l'exécution de l'algorithme LS-UNE-MÉMOIRE est égal à la somme des couts de toutes les tentatives. Nous allons montrer que $\sum_T cout(T) \leq 2 \sum_T d \leq 2n$.

Pour une tentative T_0 de type 1 : $cout(T_0) = 1 < 2d_0$ puisque $d_0 \geq 1$.

Pour une tentative T_0 de type 2 : $cout(T_0) = |z_0| + 1 \leq 2d_0$ par définition.

Il reste à considérer une tentative T_0 à la position j_0 sur y de type 3. Puisque dans ce cas $d_0 < |z_0|$, on a $d_0 = \textit{meil-fact}(m - |z_0|, y[j_0 - |z_0|])$. Ce qui veut dire que lors de la tentative suivante T_1 à la position j_1 sur y, il peut y avoir un turbo-décalage.

Considérons les deux cas suivants :

a. $|z_0| + d_0 \leq m$. Alors, par définition du turbo-décalage, nous avons : $d_1 \geq |z_0| - |z_1|$. Donc : $cout(T_0) = |z_0| + 1 \leq |z_1| + d_1 + 1 \leq d_0 + d_1$.

b. $|z_0| + d_0 > m$. Alors, par définition du turbo-décalage, nous avons : $|z_1| + d_0 + d_1 \geq m$. Donc : $cout(T_0) \leq m \leq 2d_0 - 1 + d_1$.

On peut toujours supposer que le cas b se produit lors de la tentative T_1 puisque cela donne une borne plus grande sur la valeur de $cout(T_0)$.

Lorsque la tentative T_1 est de type 1, on a alors $cout(T_1) = 1$ et $cout(T_0) + cout(T_1) \leq 2d_0 + d_1$. Ce qui est mieux que le résultat escompté.

Lorsque la tentative T_1 est de type 2 ou lorsque $|z_1| \leq d_1$, on a alors $cout(T_0) + cout(T_1) \leq 2d_0 + 2d_1$.

Il reste à considérer le cas où la tentative T_1 est de type 3 et où $|z_1| > d_1$. Cela signifie, comme après la tentative T_0, que l'on a $d_1 = \textit{meil-fact}(m - |z_1|, y[j_1 - |z_1|])$. Le raisonnement appliqué à la tentative T_1 s'applique donc également à la tentative suivante T_2. Seul le cas a peut se produire lors de la tentative T_2. Il en résulte que $cout(T_1) \leq d_1 + d_2$. Finalement, $cout(T_0) + cout(T_1) \leq 2d_0 + 2d_1 + d_2$.

Ce dernier argument donne le pas d'une preuve par induction : si toutes les tentatives $T_0, T_1, \ldots, T_k$ sont de type 3 avec $|z_j| > d_j$ pour $j = 0, 1, \ldots, k$, alors :

$$cout(T_0) + cout(T_1) + \cdots + cout(T_k) \leq 2d_0 + 2d_1 + \cdots + 2d_k + d_{k+1} \ .$$

Soit $T_{k'}$ la première tentative après la tentative T_0 telle que $|z_{k'}| \leq d_{k'}$. Cette tentative existe car, dans le cas contraire, cela signifierait qu'il existe une suite infinie de tentatives provoquant des décalages de plus en plus courts. Alors $cout(T_0) + cout(T_1) + \cdots + cout(T_{k'}) \leq 2d_0 + 2d_1 + \cdots + 2d_{k'}$.

Nous avons donc montré que $\sum_T cout(T) < 2 \sum_T d_T \leq 2n$ comme annoncé. ∎

La borne de $2n$ comparaisons du théorème 3.18 est quasi optimale, comme le montre l'exemple suivant. Posons $x = \mathtt{a}^k\mathtt{b}\mathtt{a}^k$ et $y = (\mathtt{a}^{k+1}\mathtt{b})^\ell$ avec $k \geq 1$. On a $m = 2k+1$ et $n = \ell(k+2)$. Hormis sur la première et la dernière occurrence de $\mathtt{a}^{k+1}\mathtt{b}$ (de longueur $k+2$) dans y, l'algorithme LS-UNE-MÉMOIRE effectue $2k+2$ comparaisons. Sur la première, il effectue $k+2$ comparaisons et sur la dernière il en effectue k. On obtient donc au total

$$(\ell-1)(2k+2) = 2n\left(\frac{m+1}{m+3}\right) - m - 1$$

comparaisons. La figure 3.15 illustre cet exemple avec les valeurs $k = 3$ et $\ell = 6$.

Corollaire 3.19
L'algorithme LS-UNE-MÉMOIRE *localise toutes les occurrences d'un mot dans un texte de longueur n en temps $O(n)$ avec un espace supplémentaire constant par rapport à l'algorithme* LS-SANS-MÉMOIRE.

Preuve C'est une conséquence directe des théorèmes 3.17 et 3.18. ∎

3.6 Localisation avec plusieurs mémoires

Dans cette section, on considère un algorithme du même type que les précédents mais qui travaille en mémorisant plus d'informations. Il a besoin d'un espace de travail supplémentaire $O(m)$ par rapport à l'algorithme LS-SANS-MÉMOIRE mais cela permet de réduire le nombre de comparaisons de lettres puisque celui-ci tombe à $1{,}5n$ (contre $3n$ et $2n$ respectivement pour les algorithmes LS-SANS-MÉMOIRE et LS-UNE-MÉMOIRE).

L'algorithme de cette section mémorise toutes les occurrences de suffixes du mot rencontrés dans le texte. Il utilise cette information, conjointement avec la table *suff* (section 3.3), pour effectuer des « sauts », à la façon de l'algorithme LS-UNE-MÉMOIRE, ainsi que pour augmenter la longueur de certains décalages. Celle-ci est calculée au moyen de la fonction du meilleur facteur *meil-fact*, mais peut tout aussi bien être déterminée à l'aide de la table du bon suffixe (voir section 3.1).

Phase de recherche

On décrit les éléments essentiels de la méthode. Après chaque tentative à une position j' sur le texte y, la longueur du plus long suffixe de x reconnu à la position droite j', $|z'|$, est mémorisée dans la table notée S ($S[j'] = |z'|$). Ainsi, lorsque durant la tentative courante à la position j sur le texte y, on est amené à examiner la position j', $j' < j$, (on a $y[j'+1\,..\,j] \prec_{suff} x$) pour laquelle la valeur $k = S[j']$ est définie, on sait que $y[j'-k+1\,..\,j'] \preceq_{suff} x$. Soit $i = m-1-j+j'$. Il suffit alors de

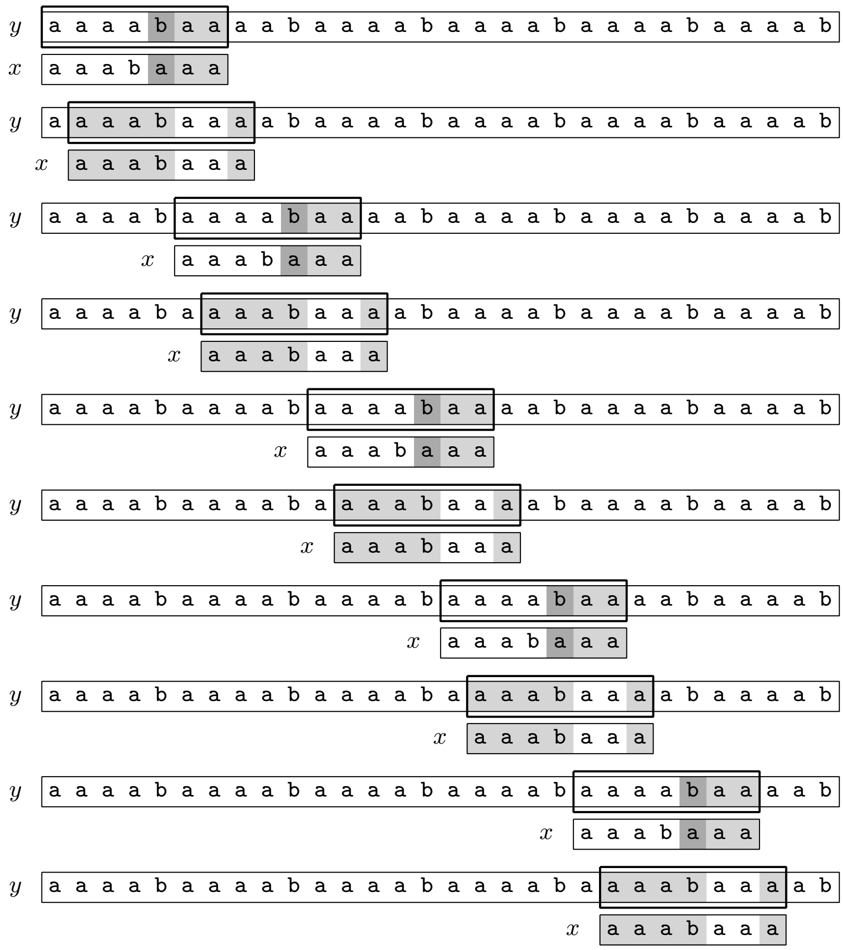


Figure 3.15 Pire des cas pour l'algorithme LS-UNE-MÉMOIRE. Illustration de la borne du théorème 3.18 avec les valeurs $k = 3$ et $\ell = 6$. Le mot x est de longueur 7, le texte y de longueur 30, et quarante comparaisons sont effectuées. Pour chacun des quatre facteurs du texte `aaaab` de longueur 5, à compter de la position 5, on effectue huit comparaisons.

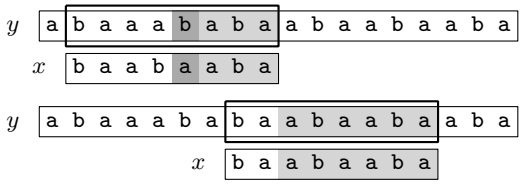


Figure 3.16 Lors de la tentative à la position 14, on reconnait, par comparaisons lettre à lettre, le suffixe `abaaba` de longueur 6 du mot. On arrive à la position 8 sur y où l'on sait (grâce à la tentative en position 8) que le plus long suffixe de x qui se termine à cette position est de longueur 3. Par ailleurs, on sait que le plus long suffixe de x se terminant à la position 1 sur x est de longueur 2. Les hypothèses du lemme 3.20 sont vérifiées. Une occurrence du mot est donc détectée à la position 14, sans avoir à recomparer $y[7\,..\,8]$.

connaitre la longueur $s = \mathit{suff}[i]$, du plus long suffixe de x se terminant à la position i sur x, pour achever la tentative dans la majorité des situations.

Quatre cas peuvent se produire. Nous les détaillons dans les lemmes 3.20 à 3.23 associés aux figures 3.16 à 3.19.

Lemme 3.20
Lorsque $s \leq k$ et $s = i + 1$, une occurrence de x apparait à la position droite j sur le texte y. On a $S[j] = m$ et le décalage de longueur pér(x) est valide.

Preuve Si $s = i+1$ et $s \leq k$ (voir figure 3.16), $y[j'-k+1\,..\,j']$ et $x[0\,..\,i]$ sont des suffixes de x et $x[0\,..\,i]$ est de longueur s. Comme $s \leq k$, on en déduit $x[0\,..\,i] \preceq_{\mathit{suff}} y[j'-k+1\,..\,j']$ et donc $y[j'-s+1\,..\,j'] = x[0\,..\,i]$. D'où $y[j-m+1\,..\,j] = x$ comme annoncé. La valeur de $S[j]$ est alors m et le décalage de longueur pér(x) est valide. ∎

Lemme 3.21
Lorsque $s \leq i$ et $s < k$, on a $S[j] = m - 1 - i + s$ et, en posant $j' = j - m + 1 + i$, le décalage de longueur meil-fact$(i - s, y[j' - s])$ est valide.

Preuve Si $s \leq i$ et $s < k$ (voir figure 3.17), on a $x[i-s+1\,..\,i] \preceq_{\mathit{suff}} x$ et $x[i-s\,..\,i] \not\preceq_{\mathit{suff}} x$, et donc $x[i-s] \neq y[j'-s]$. La valeur de $S[j]$ est alors $m-1-i+s$ (puisque $x[i-s+1\,..\,m-1] = y[j'-s+1\,..\,j]$ et $x[i-s] \neq y[j'-s]$) et le décalage de longueur meil-fact$(i-s, y[j'-s])$ est valide. ∎

Lemme 3.22
Lorsque $k < s$, on a $S[j] = m-1-i+k$ et, en posant $j' = j-m+1+i$, le décalage de longueur meil-fact$(i-k, y[j'-k])$ est valide.

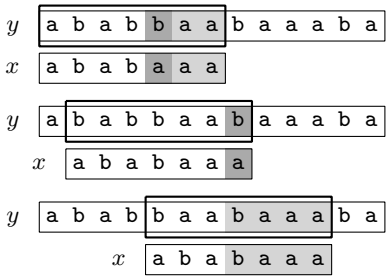


Figure 3.17 Lors de la tentative à la position 10, on reconnait le suffixe `baaa` de longueur 4 du mot. On arrive à la position 6 sur y, où l'on sait (grâce à la tentative à la position 6) que le plus long suffixe de x qui se termine à cette position est de longueur 2. Par ailleurs, on sait que le plus long suffixe de x se terminant à la position 2 sur x est de longueur 1. Donc, sans avoir à recomparer $y[4\,..\,6]$, on sait qu'il y a une inégalité entre $x[1] = \texttt{b}$ et $y[5] = \texttt{a}$.

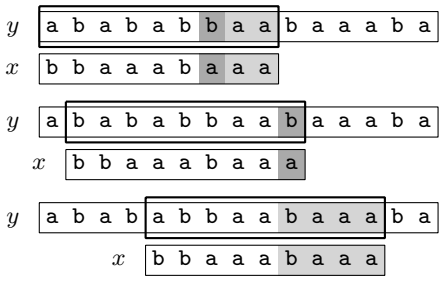


Figure 3.18 Lors de la tentative à la position 12, on reconnait le suffixe de longueur 4 du mot, et l'on arrive à la position 8, où l'on sait (grâce à la tentative à la position 8) que le plus long suffixe du mot qui se termine à cette position est de longueur 2. Par ailleurs, on sait que le plus long suffixe du mot se terminant à la position 4 sur le mot est de longueur 4. Donc, sans effectuer de comparaison de lettres, on sait qu'il y a une inégalité entre $x[2] = \texttt{a}$ et $y[6] = \texttt{b}$.

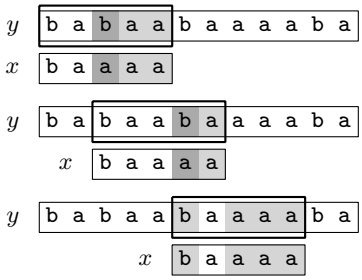


Figure 3.19 Lors de la tentative à la position 9, on reconnait le suffixe de longueur 3 du mot, et l'on arrive à la position 6, où l'on sait (grâce à la tentative à la position 6) que le plus long suffixe du mot qui se termine à cette position est de longueur 1. Par ailleurs, on sait que le plus long suffixe du mot se terminant à la position 1 sur le mot est aussi de longueur 1. On peut donc effectuer un « saut » au-dessus de $y[6]$, et reprendre les comparaisons entre $x[0]$ et $y[5]$.

Preuve Si $k < s$ (voir figure 3.18), on a $x[i-k+1\,..\,i] \preceq_{suff} x$ et $x[i-k\,..\,i] \not\preceq_{suff} x$, et donc $x[i-k] \neq y[j'-k]$. La valeur de $S[j]$ est alors $m-1-i+k$ (puisque $x[i-k+1\,..\,m-1] = y[j'-k+1\,..\,j]$ et $x[i-k] \neq y[j'-k]$) et le décalage de longueur $\mathit{meil\text{-}fact}(i-k, y[j'-k])$ est valide. ∎

Lemme 3.23
Lorsque $k = s$, on a, en posant $j' = j-m+1+i$:

$$x[i-s+1\,..\,m-1] = y[j'-s+1\,..\,j]$$

et :

$$S[j] = m-1-i+s+|\mathit{lsc}(x[0\,..\,i-s], y[j-m+1\,..\,j'-s])| \qquad (3.1)$$

Preuve Si $k = s$ (voir figure 3.19), les deux mots $x[i-s+1\,..\,i]$ et $y[j'-s+1\,..\,j']$ de même longueur sont des suffixes de x. On a donc $x[i-s+1\,..\,i] = y[j'-s+1\,..\,j']$. Et puisque que l'on suppose que l'on a $x[i+1\,..\,m-1] = y[j'+1\,..\,j]$, il s'ensuit $x[i-s+1\,..\,m-1] = y[j'-s+1\,..\,j]$.

Dans le cas où $s = i+1$, on a $S[j] = m-1-i+s$ d'une part et $\mathit{lsc}(x[0\,..\,i-s], y[j-m+1\,..\,j'-s]) = \varepsilon$ d'autre part.

Lorsque $s \leq i$ maintenant, on sait en plus que $x[i-s] \neq x[m-1-s]$ et $y[j'-s] \neq x[m-1-s]$, ce qui ne permet pas de conclure sur la comparaison entre $x[i-s]$ et $y[j'-s]$. L'égalité (3.1) est une conséquence directe de l'égalité précédente. ∎

Le code de l'algorithme LS-PLUSIEURS-MÉMOIRES est donné ci-après. Il utilise la fonction *meil-fact* de la section 3.4 et la table *suff* de la section 3.3.

La mémorisation des suffixes de x qui apparaissent dans le texte est effectuée par la table S. Les valeurs de cette table sont initialisées à 0 préalablement à la phase de recherche.

```
LS-PLUSIEURS-MÉMOIRES(x, m, y, n)
 1  pour j ← 0 à n − 1 faire
 2        S[j] ← 0
 3  j ← m − 1
 4  tantque j < n faire
 5        i ← m − 1
 6        tantque i ≥ 0 faire
 7              si S[j − m + 1 + i] > 0 alors
 8                    k ← S[j − m + 1 + i]
 9                    s ← suff[i]
10                    si s ≠ k alors
11                          i ← i − min{s, k}
12                          rupture
13                    sinon i ← i − k        ▷ Saut
14              sinonsi x[i] = y[j − m + 1 + i] alors
15                    i ← i − 1
16              sinon rupture
17        SIGNALER-SI(i < 0)
18        si i < 0 alors
19              S[j] ← m
20              j ← j + pér(x)
21        sinon S[j] ← m − 1 − i
22              j ← j + meil-fact(i, y[j − m + 1 + i])
```

Théorème 3.24
L'algorithme LS-PLUSIEURS-MÉMOIRES *localise toutes les occurrences d'un mot x dans un texte y.*

Preuve La preuve est essentiellement une conséquence des lemmes 3.20 à 3.23. ∎

Deux exemples d'exécution de l'algorithme LS-PLUSIEURS-MÉMOIRES sont montrés figure 3.20. Le premier utilise la fonction du meilleur facteur, et le second la table du bon suffixe.

Complexité de la phase de recherche

Nous examinons successivement la complexité en espace puis en temps d'exécution de l'algorithme LS-PLUSIEURS-MÉMOIRES.

Proposition 3.25
Pour localiser un mot x de longueur m, l'algorithme LS-PLUSIEURS-MÉMOIRES *peut être implanté dans un espace $O(m)$.*

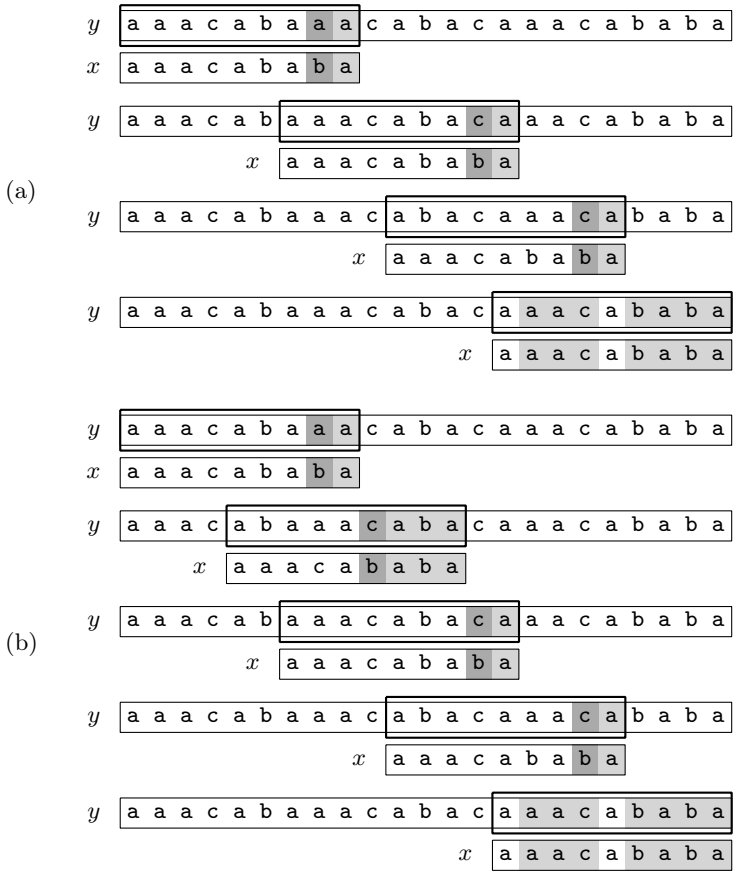


Figure 3.20 Deux exemples d'exécution de l'algorithme LS-PLUSIEURS-MÉMOIRES. **(a)** Avec utilisation de la fonction *meil-fact*. Dans ce cas, treize comparaisons entre lettres du mot et lettres du texte sont effectuées. **(b)** Avec utilisation de la table *bon-suff*. Dans ce cas, dix-sept comparaisons sont effectuées.

Preuve L'espace de travail est utilisé pour mémoriser la table *suff*, une implantation de la fonction *meil-fact* ou la table *bon-suff*, la table S et quelques variables. Les trois premiers éléments occupe un espace $O(m)$ (voir section 3.4 pour la fonction *meil-fact*). Pour la table S, on note que seule sa portion $S[j-m+1\,..\,j]$ est utile quand la fenêtre de recherche est en position droite j. Une gestion circulaire de table ou une réalisation par liste des éléments $S[\ell]$ utiles réduit l'espace à $O(m)$ (sans pénaliser le temps d'exécution). Cela donne le résultat annoncé. ∎

Nous montrons ensuite que l'algorithme LS-PLUSIEURS-MÉMOIRES a un temps d'exécution $O(n)$. Il effectue au plus $1{,}5n$ comparaisons de lettres pour localiser toutes les occurrences de x dans y.

Remarquons tout d'abord que, si durant la phase de recherche, l'occurrence d'une lettre du texte est comparée positivement, alors cette lettre ne sera plus jamais comparée dans la suite de l'exécution de l'algorithme. Il y a donc au plus n comparaisons de la sorte (c'est par exemple le cas lorsque l'on recherche a^m dans a^n, $a \in A$). Les seules lettres recomparées sont donc celles qui ont été préalablement comparées négativement.

Lemme 3.26
Si au cours d'une tentative de l'algorithme LS-PLUSIEURS-MÉMOIRES, *k comparaisons positives ont lieu sur des lettres du texte qui ont déjà été comparées, le décalage qui suit cette tentative est de longueur k au moins.*

Preuve Soit T une tentative de l'algorithme LS-PLUSIEURS-MÉMOIRES pendant laquelle k lettres du texte ayant déjà été comparées sont recomparées positivement (voir figure 3.21). Ces lettres avaient été comparées négativement durant k tentatives précédentes. Notons

$b_0v_0b_1u_1v_1b_2u_2v_2\ldots b_ku_kv_k$

le facteur du texte examiné durant la tentative T avec :

- b_0 est la lettre qui provoque une comparaison négative durant la tentative T ;
- $v_0b_1u_1v_1b_2u_2v_2\ldots b_ku_kv_k$ est un suffixe du mot x ;
- les lettres b_ℓ, $1 \leq \ell \leq k$, sont les k lettres qui sont recomparées positivement durant la tentative T ;
- les facteurs u_ℓ, $1 \leq \ell \leq k$, sont les suffixes (éventuellement vides) du mot qui avaient été reconnus lors des k tentatives où les b_ℓ avaient été comparées négativement. Ces facteurs sont « sautés » durant la tentative T ;
- les facteurs v_ℓ, $1 \leq \ell \leq k$, sont les facteurs du texte qui sont comparés (pour la première fois) positivement lors de la tentative T.

Figure 3.21 Lors de la tentative à une position j sur le texte, on reconnait le suffixe du mot de longueur 1, puis on décale de six positions. On reconnait ensuite le suffixe de longueur 3, on décale de quatre positions. Puis on reconnait le suffixe de longueur 2 et l'on décale de cinq positions. Lors de la tentative à la position $j+15$, on reconnait le suffixe du mot de longueur 19, en effectuant trois recomparaisons sur les lettres $y[j+8]$, $y[j+3]$ et $y[j-1]$. Le décalage qui suit cette tentative ne peut être de longueur inférieure ou égale à 3 d'après le lemme 3.26. En effet le suffixe `aabaaabaaaabaaaaaaa` du mot ne peut avoir de période inférieure ou égale à 3.

De par leur définition, les mots $b_\ell u_\ell$, $1 \leq \ell \leq k$, ne sont pas des suffixes du mot.

Raisonnons par l'absurde et supposons que le décalage d appliqué juste après la tentative T soit de longueur strictement inférieure à k. Soit w le suffixe de x de longueur d. Par définition, le mot

$$v_0 b_1 u_1 v_1 b_2 u_2 v_2 \dots b_k u_k v_k w$$

est un suffixe de x et a pour période $d = |w|$.

Pour deux indices différents $\ell' \neq \ell''$, $u_{\ell'}$ et $u_{\ell''}$ sont alignés avec une même position dans un facteur w, puisqu'il y a moins de $k-1$ positions possibles. Cela implique que $b_{\ell'} u_{\ell'} = b_{\ell''} u_{\ell''}$. Les décalages appliqués après les deux tentatives où $b_{\ell'}$ et $b_{\ell''}$ ont été comparés sont de même longueur. Cela implique que $b_{\ell'+1} u_{\ell'+1} = b_{\ell''+1} u_{\ell''+1}$. Il existe donc un indice $\ell < k$ tel que $b_\ell u_\ell = b_k u_k$, ce qui contredit le fait le mot x aurait du être aligné comme pendant la tentative T avant celle-ci.

Il s'ensuit que la longueur du décalage appliqué après la tentative T est au moins k. ∎

Lemme 3.27
L'algorithme LS-PLUSIEURS-MÉMOIRES *effectue moins de $n/2$ comparaisons impliquant des lettres du texte ayant déjà été comparées.*

Preuve On groupe les tentatives par paquets, deux tentatives étant dans un même paquet lorsqu'elles effectuent des comparaisons sur des

lettres du texte communes. Un paquet p de tentatives qui effectuent k recomparaisons positives de lettres du texte contient au moins $k+1$ tentatives. Parmi ces tentatives au moins k d'entre elles appliquent un décalage de longueur au moins 1 et une applique un décalage de longueur au moins k (par le lemme 3.26). Donc la longueur totale de tous les décalages des tentatives du paquet p est au moins égale à $2k$.

La somme totale de tous les décalages appliqués au cours de l'algorithme LS-PLUSIEURS-MÉMOIRES est inférieure à n. Le nombre total de recomparaisons est donc bien inférieur à $n/2$. ∎

Théorème 3.28
Lors de la recherche de toutes les occurrences d'un mot dans un texte y de longueur n, l'algorithme LS-PLUSIEURS-MÉMOIRES *effectue au plus* $1{,}5n$ *comparaisons entre des lettres du mot et des lettres du texte.*

Preuve Le résultat découle directement du lemme 3.27 et du fait qu'il y a au plus n comparaisons positives. ∎

Corollaire 3.29
L'algorithme LS-PLUSIEURS-MÉMOIRES *effectue la recherche de toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(n)$ avec un espace supplémentaire $O(m)$ par rapport à l'algorithme* LS-SANS-MÉMOIRE.

Preuve C'est une conséquence du théorème 3.24, en notant que le temps d'exécution est asymptotiquement équivalent au nombre de comparaisons, et du théorème 3.28. ∎

La borne de $1{,}5n$ comparaisons du théorème 3.28 est quasi atteinte lors de la recherche du mot $x = \mathtt{a}^{k-1}\mathtt{ba}^k\mathtt{b}$ dans le texte $y = (\mathtt{a}^{k-1}\mathtt{ba}^k\mathtt{b})^\ell$, avec $k \geq 1$. L'algorithme effectue alors exactement

$$2k + 1 + (3k+1)(\ell - 1) = \frac{3k+1}{2k+1}n - k$$

comparaisons entre lettres du mot et lettres du texte. La figure 3.22 illustre cette borne avec les valeurs $k = 3$ et $\ell = 4$.

3.7 Localisation d'un dictionnaire

Avec la technique de fenêtre glissante, il est possible de résoudre efficacement le problème de la recherche de toutes les occurrences des mots appartenant à un dictionnaire de k mots $X = \{x_0, x_1, \ldots, x_{k-1}\}$ dans un texte y. Dans cette section, on désigne respectivement par m' et m'' la longueur du plus court mot et du plus long mot de X.

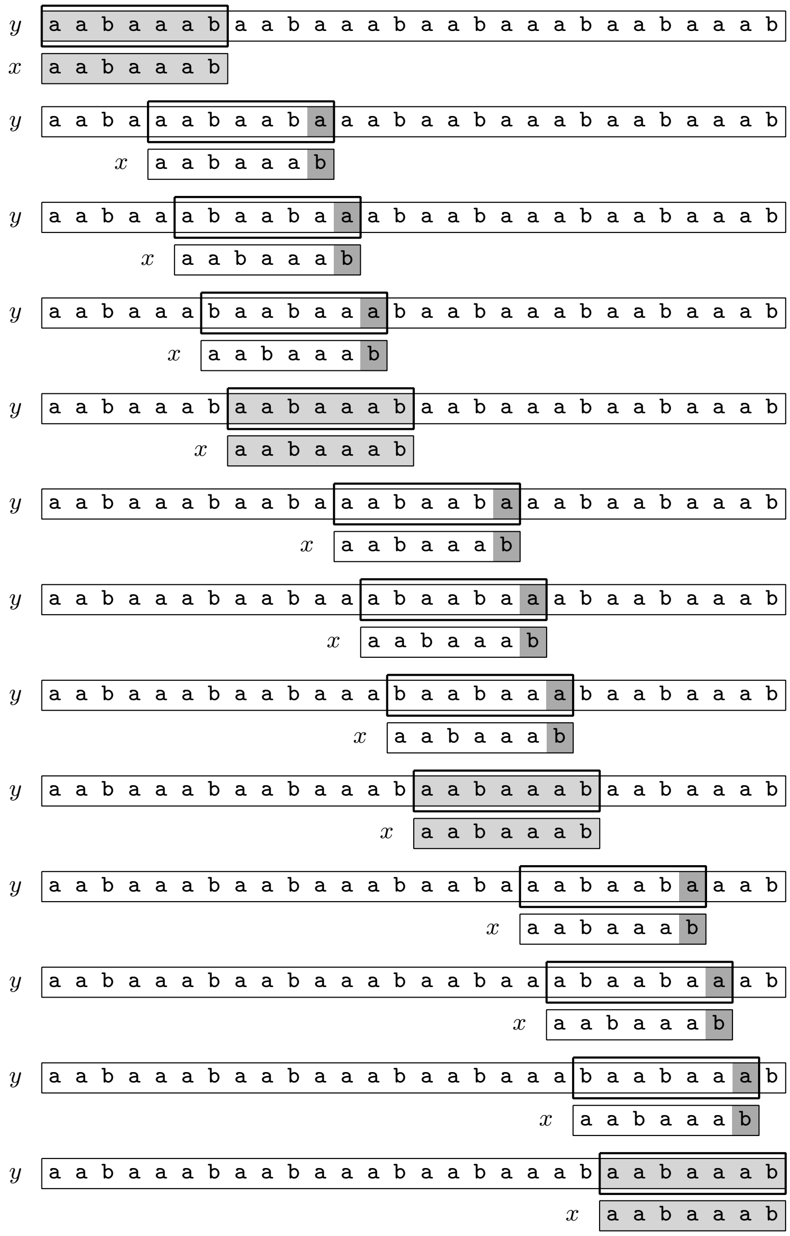


Figure 3.22 Illustration de la borne du théorème 3.28 avec les valeurs $k = 3$ et $\ell = 4$. Le mot est de longueur 7, le texte de longueur 28, et trente-sept comparaisons sont effectuées. Pour chacune des trois dernières occurrences du facteur `aabaaab` de longueur 7, l'algorithme LS-PLUSIEURS-MÉMOIRES effectue dix comparaisons.

L'examen du texte durant une tentative consiste à déterminer le plus long facteur des mots de X qui est un suffixe du contenu de la fenêtre. Cette façon de procéder allonge le suffixe de la fenêtre qui est examiné par rapport au suffixe considéré dans les méthodes des sections précédentes. Cela permet de recueillir plus d'informations sur le texte et conduit souvent à des décalages plus longs. Pour mettre en œuvre cette méthode, on utilise un automate des suffixes des renversés des mots de X (voir chapitre 5). Pendant l'examen du texte y, l'automate contient les informations suffisantes pour produire les positions des occurrences des mots de X dans y.

Le but de l'algorithme est de produire les mots de X qui sont des suffixes du contenu de la fenêtre de longueur m''. Le principe du calcul consiste à déterminer à chaque tentative les préfixes des mots de X qui sont des suffixes du contenu de la fenêtre. Dans le même temps, on produit les mots de X qui apparaissent dans la fenêtre et l'on retient la longueur minimale des décalages valides, sachant que celle-ci ne peut être strictement supérieure à m'.

On décrit maintenant la technique utilisée dans ce but. Soit $X^{\sim}$ l'ensemble des renversés des mots de X. On considère un automate N qui reconnait les suffixes des renversés des mots de X avec sa fonction de transition associée δ. L'automate accepte le langage

$$\mathit{Suff}(X^{\sim}) = \{v \in A^* : uv = x^{\sim}, u \in A^*, x \in X\} \ .$$

En d'autres termes, N reconnait de manière déterministe les préfixes des mots de X par lecture de droite à gauche. Pour chaque état terminal de l'automate, atteint par un mot de $X^{\sim}$, on pose :

$$\mathit{sortie}[q] = \{i : 0 \leq i \leq k-1 \text{ et } \overline{\delta}(q_0, {x_i}^{\sim}) = q\} \ ,$$

où $\overline{\delta}$ est l'extension aux mots de la fonction de transition δ de l'automate et q_0 est son état initial (voir section 1.1).

Une tentative à la position j sur le texte y consiste à analyser les lettres de y de droite à gauche à partir de $y[j]$ à l'aide de N. À chaque fois qu'un état q est atteint par une lettre $y[j']$, on vérifie si $\mathit{sortie}[q]$ est non vide ; si tel est le cas, le mot x_i apparait dans le texte y à la position j' quand

$$i \in \mathit{sortie}[q] \text{ et } j - j' + 1 = |x_i| \ .$$

En outre, si l'état q est terminal aucun décalage valide ne peut être de longueur strictement supérieure à $m' - (j - j' + 1)$ lorsque cette quantité est positive. On peut donc calculer simultanément la longueur minimale d des décalages valides. Enfin, la tentative se termine lorsqu'il n'existe plus de transition définie pour la lettre courante depuis l'état courant. Un exemple de recherche est montré figure 3.23. L'algorithme qui suit implante cette méthode.

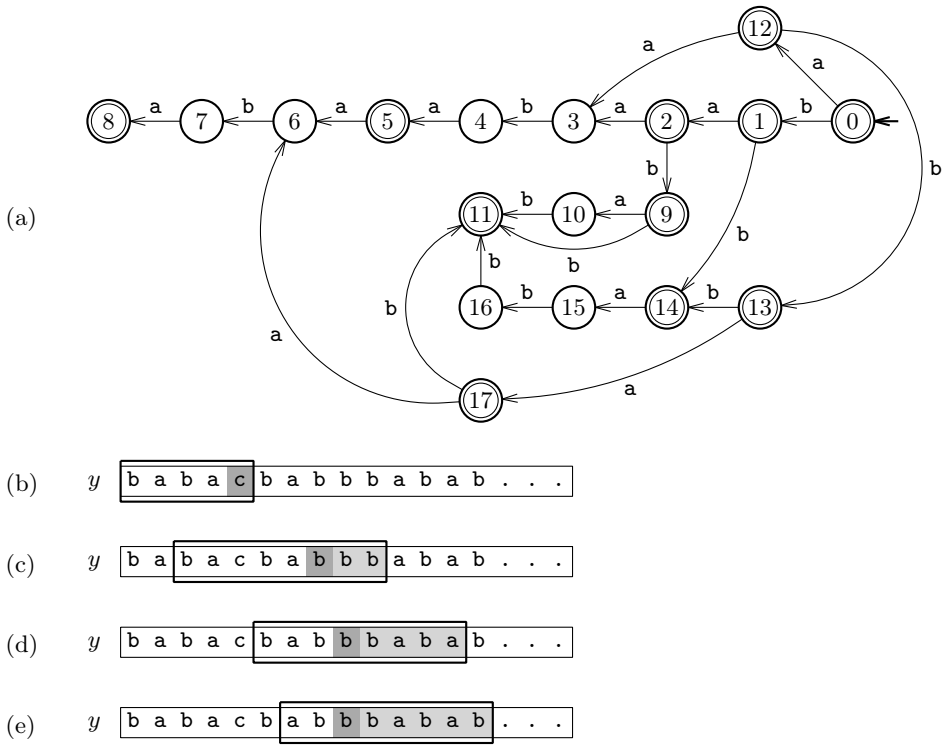


Figure 3.23 Exemple d'exécution de l'algorithme LS-MULTIPLE dans le cas où $X = \{\texttt{abaabaab}, \texttt{babab}, \texttt{bbabba}\}$ et y commence par `babacbabbbabab`. On a $m' = 5$. **(a)** Un automate qui reconnait les suffixes de $X^{\sim}$. **(b)** Première tentative. La longueur de la fenêtre est égale à la longueur du plus long mot de X, soit 8, mais la première tentative doit commencer en se calant sur le plus petit mot de X soit $y[0\,..\,4]$. À chaque tentative la recherche commence dans l'état initial 0. Il n'y a pas de transition définie depuis l'état 0 par la lettre $y[4] = \texttt{c}$, donc la tentative se termine et un décalage de longueur 5 est appliqué. **(c)** Deuxième tentative. La fenêtre de longueur 8 est positionnée sur le facteur du texte $y[2\,..\,9] = \texttt{bacabbb}$. Depuis l'état initial 0, on atteint l'état terminal 14 après transition par le mot `bb`. Il n'y a pas de transition définie depuis cet état par la lettre $y[7] = \texttt{b}$, donc la tentative se termine et un décalage de longueur $m' - |\texttt{bb}| = 3$ est appliqué. **(d)** Troisième tentative. On transite par les états 0, 12, 13, 17 et 11. Un décalage de longueur $m' - |\texttt{baba}| = 1$ est appliqué. **(e)** Quatrième tentative. On transite par les états 0, 1, 2, 9, 10 et 11. Un décalage de longueur 1 est appliqué.

```
LS-MULTIPLE(X, m', y, n)
 1  soit N un automate qui reconnait les suffixes des renversés
      des mots de X
 2  j ← m' − 1
 3  tantque j ≤ n faire
 4        q ← initial[N]
 5        j' ← j
 6        d ← m'
 7        tantque j' ≥ 0 et CIBLE(q, y[j']) ≠ NIL faire
 8              q ← CIBLE(q, y[j'])
 9              si terminal[q] alors
10                    pour chaque i ∈ sortie[q] faire
11                          SIGNALER-SI(|x_i| = j − j' + 1)
12                    si m' − j + j' − 1 > 0 alors
13                          d ← min{d, m' − j + j' − 1}
14                    sinon d ← 1
15              j' ← j' − 1
16        j ← j + d
```

Théorème 3.30
L'algorithme LS-MULTIPLE *localise toutes les occurrences des mots d'un dictionnaire X dans un texte y.*

Preuve Notons que l'algorithme ne détecte que des occurrences de mots de X (lignes 10–11). Vérifions qu'il n'en oublie aucune.

Soit ℓ la position droite sur y d'une occurrence d'un mot $x_i \in X$. Soit j la position droite de la fenêtre de recherche. On montre dans la suite que si $j \leq \ell$, l'occurrence de x_i est détectée. Remarquons que l'on peut supposer en outre $\ell < j + m'$ puisque, la longueur des décalages étant limitée à m', la variable j prend une valeur qui satisfait les deux conditions. On effectue la preuve par récurrence sur la quantité $\ell - j$.

Si $j = \ell$, l'automate N reconnaissant les préfixes des mots de X, il accepte x_i. À la position $j' = j - |x_i| + 1$, l'état courant est terminal, sa sortie contient i et la condition $|x_i| = j - j' + 1$ de la ligne 11 est remplie, donc l'occurrence est signalée.

Supposons maintenant $j < \ell$, et soit $x_i = uv$ où v est de longueur $\ell - j$. Dans cette situation, u est un suffixe du contenu de la fenêtre. À la position $j' = j - |u| + 1$, l'état courant est terminal, car u est un préfixe de x_i, et la condition $j - j' + 1 \leq m'$ est remplie. Cela limite la longueur du décalage à $m' - j + j' - 1 \leq |v|$. La prochaine valeur de j, disons j'', sera donc telle que $j'' \leq \ell$ avec $\ell - j'' < \ell - j$. L'hypothèse de récurrence permet alors de conclure que l'occurrence de x_i est détectée, ce qui termine la récurrence.

Enfin, on vérifie qu'initialement on a bien $j \leq \ell$ pour toute position droite d'un mot de X car j est initialisée à $m' - 1$. Donc toute occurrence d'un mot de X est signalée. ∎

Bien que l'algorithme LS-MULTIPLE ait un très bon comportement en pratique, son temps d'exécution est $O(k \times m'' \times n)$ dans le pire des cas. En effet, les instructions de la boucle **tantque** des lignes 3–16 peuvent être exécutées n fois au maximum, celles des lignes 7–15 m'' fois, et celles de la boucle **pour** des lignes 10–11 k fois. Son temps d'exécution peut toutefois être rendu linéaire par application de techniques standard.

Notes

L'algorithme LS-SANS-MÉMOIRE est dû à Boyer et Moore (1977). Les preuves des théorèmes 3.7 et 3.8 ont été établies par Cole (1994). L'idée de la localisation des occurrences d'un mot en utilisant une fenêtre de longueur égale à la période du mot a été exposée par Galil (1979). Hancart (1993) a proposé le calcul de l'automate du meilleur facteur et le calcul de la fonction du meilleur facteur de la section 3.4.

L'algorithme LS-UNE-MÉMOIRE est dû à Crochemore et co-auteurs (1994). Il est connu sous l'appellation *Turbo-BM*.

Apostolico et Giancarlo (1986) ont présenté l'idée de l'algorithme LS-PLUSIEURS-MÉMOIRES. La version qui en est donnée ici, ainsi que la preuve du théorème 3.28, ont été données par Crochemore et Lecroq (1997).

L'algorithme LS-MULTIPLE est dû à Crochemore et co-auteurs (1999). Les mêmes auteurs ont proposé une version linéaire de cet algorithme. Raffinot (1997) a décrit une variante de ce dernier algorithme implantée par la commande `vfgrep` sous le système UNIX.

Les automates de Boyer-Moore (exercice 3.6) ont été introduits par Knuth (1977). On ne sait pas si la taille de ces automates est polynomiale.

Une animation d'algorithmes de recherche de mots (dont ceux de ce chapitre) est proposée sur le site [71], développé par Charras et Lecroq.

Exercices

3.1 (Implantation)
Écrire l'algorithme LS-SANS-MÉMOIRE en utilisant deux variables qui ont pour valeurs celles de i et $j-m+1+i$ du code de la section 3.1. Redéfinir *meil-fact* en conséquence. Même question pour les autres versions de l'algorithme.

3.2 (Période)
Donner le code d'un algorithme qui localise toutes les occurrences de x dans y en utilisant une fenêtre de longueur $pér(x)$ et qui effectue moins de $3n$ comparaisons entre lettres du mot et du texte. [*Aide :* voir Galil (1979).]

3.3 (Mieux)
Donner l'exemple d'un mot et d'un texte pour lesquels l'algorithme LS-SANS-MÉMOIRE effectue moins de comparaisons lorsque l'on utilise la table *bon-suff* plutôt que la fonction *meil-fact*.

3.4 (Moins bien)
Donner des exemples de mots et de textes pour lesquels tous les algorithmes de ce chapitre effectuent plus de comparaisons que l'algorithme LOCALISER-RAPIDEMENT de la section 1.5.

3.5 (Nombre de flèches)
Montrer par un argument direct – c'est-à-dire sans faire appel à une méthode de construction – que le nombre de flèches de l'automate du meilleur facteur de tout mot non vide x qui entrent dans un état qui n'est pas un suffixe de x est au plus égal à $|x| - 1$. [*Aide :* comme pour la preuve de la proposition 2.19, montrer que les sorties des états en question sont deux à deux distinctes et sont comprises entre 1 et $|x| - 1$.]

3.6 (Automate de Boyer-Moore)
L'automate de Boyer-Moore est un automate déterministe des configurations de la fenêtre de recherche rencontrées pendant l'exécution de l'algorithme LS-SANS-MÉMOIRE. Les états sont porteurs de l'information déjà connue sur le contenu de la fenêtre après les comparaisons précédentes.

On note B l'automate de localisation associé au mot $x \in A^+$ de longueur m. Son ensemble d'états est noté Q, son ensemble de flèches F. Il possède une fonction de décalage d qui donne la longueur du décalage à réaliser, et une fonction de sortie booléenne s qui signale une occurrence de x, toutes deux définies sur F.

Les états sont définis comme suit :

- Q est la partie de $(A \cup \{\#\})^*$ accessible depuis d'état initial. La lettre # qui n'appartient pas à l'alphabet A représente l'absence d'information sur la lettre correspondante de la fenêtre ;
- l'état initial est le mot $\#^m$.

L'ensemble des flèches F et les fonctions d et s sont définis comme suit, pour $u \in (A \cup \{\#\})^*$, $v \prec_{suff} x$, $w' \in \{\#\}^*$ et $a \in A$:

- $f = (u\#v, a, uav) \in F$ si $av \prec_{suff} x$ et $uav \neq x$. On a $d(f) = 0$ et $s(f) =$ FAUX ;
- $f = (u\#v, a, Bord(x)w') \in F$ si $uav = x$. On a $d(f) = |w'| = pér(x)$ et $s(f) =$ VRAI ;
- le triplet $f = (u\#v, a, ww') \in F$ si $av \not\prec_{suff} x$ et w est le plus long mot pour lequel $w \prec_{suff} uav$ et, pour $i = 0, 1, \ldots, |w| - 1$, $w[i] = x[i]$ ou $w[i] = \#$. On a $d(f) = |w'|$ et $s(f) =$ FAUX.

Pour $q \in Q$, $q[i] = \#$ signifie que la lettre du texte alignée avec $x[i]$ n'a jamais été inspectée. La stratégie avec laquelle les comparaisons lettre à

lettre sont effectuées est celle des algorithmes de localisation du chapitre : de la droite vers la gauche, mais en commençant par la première lettre non inspectée.

Donner l'automate de recherche pour le mot $x =$ `aabbabb`. Écrire un algorithme de localisation de x utilisant l'automate B. Écrire un algorithme de construction de l'automate B. Donner une borne précise à la taille de B.

3.7 (Des preuves !)
Adapter la preuve de complexité de l'algorithme LS-SANS-MÉMOIRE-FAIBLE à l'algorithme LS-SANS-MÉMOIRE.

3.8 (Meilleur facteur)
Déduire de la section 3.4 une implantation de la fonction *meil-fact*. Écrire un algorithme qui construit l'automate du meilleur facteur de tout mot x en temps et en espace $O(|x|)$.

3.9 (Bon suffixe)
Écrire un algorithme qui calcule la table *bon-suff* uniquement à l'aide de la table *bordd* (et de m) définie pour x par :

$$bordd[i] = |Bord(x[i\,..\,m-1])| \ ,$$

pour toute position i sur x.

3.10 (Bis)
Soit BON-SUFFIXE-BIS l'algorithme dont le code suit.

```
BON-SUFFIXE-BIS(x, m)
 1  pour i ← 0 à m − 1 faire
 2          bon-suff[i] ← 0
 3  f[m − 1] ← m
 4  j ← m − 1
 5  pour i ← m − 2 à 0 pas −1 faire
 6          f[i] ← j
 7          tantque j < m et x[i] ≠ x[j] faire
 8                  si bon-suff[j] = 0 alors
 9                          bon-suff[j] ← j − i
10                  j ← f[j]
11          j ← j − 1
12  pour i ← 0 à m − 1 faire
13          si bon-suff[i] = 0 alors
14                  bon-suff[i] ← j + 1
15          si i = j alors
16                  j ← f[j]
17  retourner bon-suff
```

Montrer qu'on a $f[i] = m - 1 - bordd[i+1]$ pour toute position i sur x à la fin de l'exécution de l'algorithme (la table *bordd* est définie dans l'exercice 3.9).

Montrer que l'algorithme calcule effectivement la table *bon-suff*.

3.11 (Témoins)
Soient $y \in A^+$ et w un préfixe propre de $x \in A^+$. On suppose que w est périodique, c'est-à-dire $|w| \geq 2pér(w)$

Montrer que le mot $w[0\,..\,2pér(w) - 2]$ n'est pas périodique.

Soient $p = |w| - pér(w)$ et $q = |w|$, et supposons que $x[p] \neq x[q]$ (les entiers p et q sont des témoins de non périodicité $q - p$ de x). Montrer que si simultanément $y[j + p] = x[p]$ et $y[j + q] = x[q]$ alors le mot x ne possède aucune occurrence aux positions $j + 1$, $j + 2$, ..., $j + p$ sur y.

Déduire des propriétés précédentes un algorithme de localisation des occurrences de x dans y qui effectue au plus $2|y|$ comparaisons entre lettres de x et de y pendant la recherche et qui n'utilise qu'un espace supplémentaire constant. [*Aide :* on distinguera les trois cas : aucun préfixe de x n'est périodique ; w est un plus long préfixe périodique de x et x n'est pas périodique ; x est périodique. Voir aussi Gąsieniec, Plandowski et Rytter (1995).]

3.12 (Heuristique)
Montrer que, dans l'algorithme LS-UNE-MÉMOIRE, si l'on utilise l'heuristique *dern-occ* et la table *bon-suff*, et si le décalage est donné par l'heuristique, alors la longueur du décalage doit être supérieure à $|z|$ (le mot z étant le suffixe du mot reconnu lors de la tentative).

Donner le code complet de l'algorithme modifié avec inclusion de l'heuristique et utilisation de la propriété ci-dessus.

3.13 (Tout seul)
Adapter l'algorithme LS-MULTIPLE au cas de la recherche d'un seul mot. [*Aide :* voir Crochemore et co-auteurs (1994).]

3.14 (Linéaire)
Combiner les techniques de la localisation d'un dictionnaire présentés dans le chapitre 2 avec celles mises en œuvre dans l'algorithme LS-MULTIPLE afin d'obtenir un algorithme de localisation fonctionnant en temps linéaire. [*Aide :* voir Crochemore et co-auteurs (1999).]

4 Table des suffixes

On considère dans ce chapitre le problème de la localisation d'un mot dans une liste de mots mémorisés dans une table. La table est supposée fixe et peut donc être préparée pour accélérer les accès ultérieurs. La recherche d'un mot dans un lexique ou un dictionnaire qui peuvent être stockés en mémoire centrale d'un ordinateur est une application de cette question.

On indique comment classer lexicographiquement les mots de la liste (en temps maximal proportionnel à la longueur totale des mots) de façon à pouvoir appliquer un algorithme de recherche par dichotomie. En réalité, le classement ne suffit pas entièrement pour obtenir une recherche efficace. Le précalcul et l'utilisation des préfixes communs aux mots de la liste sont des éléments supplémentaires qui rendent la technique très performante. La recherche d'un mot de longueur m dans une liste de n mots prend ainsi un temps $O(m + \log n)$.

La table des suffixes d'un texte est une structure de données qui applique la technique précédente aux n suffixes (non vides) d'un texte de longueur n. Elle permet de déterminer toutes les occurrences d'un facteur du texte, en temps $O(m+\log n)$ comme ci-dessus, et fournit une solution complémentaire à celles décrites dans les chapitres 2 et 3. Le texte est fixe et sa préparation procure un accès efficace à ses suffixes. Dans ce cas, la préparation du texte, classement lexicographique de ses suffixes et calcul de leurs préfixes communs, peut être adaptée pour s'exécuter en temps $O(n \times \log n)$ alors que la somme des longueurs des suffixes est quadratique.

Le chapitre présente donc une solution algorithmique aux problèmes de la recherche d'un mot dans une liste fixe et dans les facteurs d'un texte fixe. Le chapitre 5 complète l'étude en proposant une solution basée sur des structures de données adaptées à la mémorisation des suffixes d'un texte. Enfin, le chapitre 9 présente une solution alternative à la question de la préparation d'une table des suffixes.

L'intérêt de considérer les suffixes d'un mot réside dans les applications de recherche de motifs et d'implantation d'index qui sont décrites

dans le chapitre 6. La technique de recherche en liste permet en fait de calculer l'intervalle des mots de la liste qui possèdent un préfixe donné, et c'est pour cette raison qu'elle s'adapte à la localisation de motifs.

Tout cela suppose l'existence d'une relation d'ordre sur l'alphabet. Mais ce n'est pas une contrainte en pratique puisque les données stockées en mémoire d'un ordinateur sont codées en binaire et que l'on bénéficie ainsi de l'ordre lexicographique sur les suites binaires.

4.1 Recherche dans une table de mots

On considère une liste L de n mots de A^* supposés mémorisés dans une table : $L_0, L_1, \ldots, L_{n-1}$. Dans cette section et la suivante, on suppose que les mots sont en ordre lexicographique croissant, $L_0 \leq L_1 \leq \cdots \leq L_{n-1}$. Le classement de la liste est étudié en section 4.3.

Le problème de base considéré dans le chapitre est celui de la localisation d'un mot $x \in A^*$ dans la liste. Dans les applications, il est souvent plus intéressant de répondre à une question plus précise qui prend en compte la structure des éléments de la liste, à savoir, déterminer quels sont les mots de la liste possédant x comme préfixe. C'est ce problème qui est à l'origine d'une implantation d'index présentée dans le chapitre 6 et qui permet entre autres la localisation du mot x dans un texte y.

Problème de l'intervalle : soient $n \geq 0$ et $L_0, L_1, \ldots, L_{n-1} \in A^*$, satisfaisant la condition $L_0 \leq L_1 \leq \cdots \leq L_{n-1}$. Pour $x \in A^*$, calculer les indices d et f, $-1 \leq d < f \leq n$, pour lesquels : $d < i < f$ si et seulement si $x \preceq_{préf} L_i$.

Le choix des bornes -1 et n dans l'énoncé simplifie l'écriture de l'algorithme qui le résout (algorithme Intervalle de la section 4.2). On procède comme si la liste était précédée par un mot plus petit (dans l'ordre lexicographique) que tout autre, et qu'elle était suivie par un mot plus grand que tout autre.

On peut poser le test d'appartenance de x à la liste L en des termes qui réduisent le problème précédent et en rendent la programmation plus directe. Plus que l'appartenance d'ailleurs, la réponse au problème fournit une localisation précise de x par rapport aux éléments ordonnés de la liste même s'il n'y figure pas.

Problème de l'appartenance : soient $n \geq 0$ et $L_0, L_1, \ldots, L_{n-1} \in A^*$, satisfaisant la condition $L_0 \leq L_1 \leq \cdots \leq L_{n-1}$. Pour $x \in A^*$, calculer un indice i, $-1 < i < n$, pour lequel $x = L_i$ si x figure dans la liste L, ou des indices d et f, $-1 \leq d < f \leq n$, pour lesquels $d + 1 = f$ et $L_d < x < L_f$ sinon.

La recherche de x dans la liste L peut se faire de façon séquentielle sans aucune préparation de la liste, sans avoir besoin d'ailleurs qu'elle soit ordonnée. Le temps d'exécution est alors de l'ordre de $m \times n$. En appliquant cette méthode, on ne retire aucun gain du fait que la liste

est classée et que l'on peut la préparer avant la recherche. Une seconde solution consiste à appliquer une recherche par dichotomie comme il est classique de le faire sur des tables classées d'éléments. L'algorithme de recherche s'écrit alors comme ci-dessous. Il fournit une réponse assez efficace au problème de l'appartenance, solution qui est améliorée dans la section suivante. Le code de l'algorithme fait appel à la fonction *lpc* qui est définie, pour $u, v \in A^*$, par :

$lpc(u, v) =$ le plus long préfixe commun à u et v .

Dans le code ci-dessous, on note que $L_i[\ell]$ est la lettre de position ℓ sur le mot d'indice i de la liste L. On remarque aussi que l'initialisation de d et f revient à considérer, comme on l'a déjà mentionné, que la liste possède deux mots supplémentaires L_{-1} et L_n de longueur 1, le mot L_{-1} étant constitué d'une lettre strictement inférieure à toutes les lettres des mots $x, L_0, L_1, \ldots, L_{n-1}$, le mot L_n étant lui constitué d'une lettre strictement supérieure.

```
RECHERCHE-SIMPLE(L, n, x, m)
 1  d ← −1
 2  f ← n
 3  tantque d + 1 < f faire
 4         ▷ Invariant : L_d < x < L_f
 5         i ← ⌊(d + f)/2⌋
 6         ℓ ← |lpc(x, L_i)|
 7         si ℓ = m et ℓ = |L_i| alors
 8                retourner i
 9         sinonsi (ℓ = |L_i|) ou (ℓ ≠ m et L_i[ℓ] < x[ℓ]) alors
10                d ← i
11         sinon f ← i
12  retourner (d, f)
```

L'algorithme RECHERCHE-SIMPLE considère un ensemble de couples d'entiers (d, f) qui sont structurés en un arbre, l'arbre de la recherche dichotomique. L'exécution de l'algorithme correspond à un parcours le long d'une branche de l'arbre depuis la racine $(-1, n)$. La descente s'arrête sur un nœud externe de l'arbre lorsque le mot x ne figure pas dans la liste, sinon elle s'arrête avant. La figure 4.1 montre l'arbre de la recherche dichotomique lorsque $n = 6$.

L'ensemble N des nœuds de l'arbre est défini inductivement par les conditions :

- $(-1, n) \in N$;
- si $(d, f) \in N$ et $d + 1 < f$, alors à la fois $(d, \lfloor (d + f)/2 \rfloor) \in N$ et $(\lfloor (d + f)/2 \rfloor, f) \in N$.

Les nœuds externes de l'arbre sont les couples (d, f), $-1 \leq d < f \leq n$, pour lesquels $d + 1 = f$. Un nœud interne (d, f), $-1 \leq d + 1 < f \leq n$, de l'arbre possède deux enfants : $(d, \lfloor (d + f)/2 \rfloor)$ et $(\lfloor (d + f)/2 \rfloor, f)$.

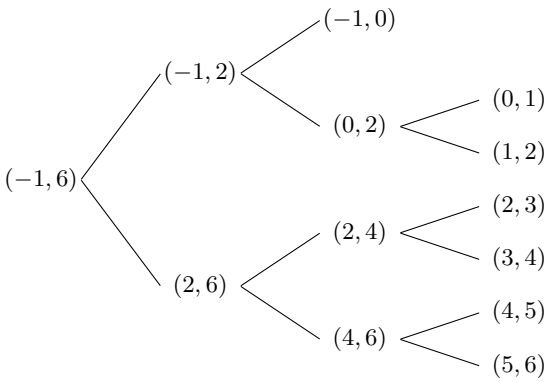


Figure 4.1 Arbre de la recherche dichotomique au sein d'une liste à six éléments. L'arbre possède $2 \times 6 + 1 = 13$ nœuds, 6 internes et 7 externes.

Lemme 4.1
L'arbre de la recherche dichotomique associé à une liste à n éléments possède $2n + 1$ nœuds.

Preuve L'arbre de la recherche dichotomique possède les $n+1$ nœuds externes $(-1, 0), (0, 1), \ldots, (n-1, n)$. Dans ce type d'arbre, le nombre de nœuds internes est d'une unité de moins que le nombre de nœuds externes (simple preuve par récurrence sur le nombre de nœuds). Il y en a donc n, ce qui donne le résultat. ∎

Proposition 4.2
L'algorithme RECHERCHE-SIMPLE *localise un mot x de longueur m dans une liste classée de taille n (problème de l'appartenance) en temps $O(m \times \log n)$ avec un maximum de $m \times \lceil \log_2(n+1) \rceil$ comparaisons de lettres.*

Preuve L'algorithme s'arrête car la différence $f - d$ diminue strictement à chaque exécution des lignes 5 à 11, ce qui résulte de l'inégalité $d + 1 < f$. On peut aussi vérifier que la propriété de la ligne 4 est invariante. En effet, le test de la ligne 7 contrôle l'égalité des mots x et L_i. Et en cas d'inégalité, le test de la ligne 9 permet de déterminer lequel est supérieur à l'autre dans l'ordre lexicographique. On en déduit que l'algorithme résout le problème de l'appartenance.

Chaque comparaison de mots demande au plus m comparaisons en comptant le calcul de $|lpc(x, L_i)|$ par comparaisons lettre à lettre. La longueur de l'intervalle entier (d, f) varie de $n+1$ jusqu'à 1 au minimum. Cette longueur moins une unité est divisée par deux à chaque étape, soit $\lceil \log_2(n+1) \rceil$ étapes au plus. D'où l'on déduit le résultat sur le nombre de comparaisons qui est représentatif du temps d'exécution.

L'exemple ci-dessous montre que la borne sur le nombre de comparaisons est atteinte. ∎

Le résultat de la proposition n'est pas surprenant et la majoration du temps d'exécution est précise quand x n'est pas plus long que les éléments de la liste. En effet, le nombre maximal de comparaisons est atteint avec l'exemple suivant. On choisit pour liste de mots

$$L = \langle \mathtt{a}^{m-1}\mathtt{b}, \mathtt{a}^{m-1}\mathtt{c}, \mathtt{a}^{m-1}\mathtt{d}, \ldots \rangle$$

et pour mot à rechercher $x = \mathtt{a}^m$. On suppose l'ordre habituel sur les lettres : $\mathtt{a} < \mathtt{b}$, $\mathtt{b} < \mathtt{c}$, etc. Le résultat de l'algorithme RECHERCHE-SIMPLE est le couple $(-1, 0)$ qui indique que x est inférieur à tous les mots de L. Si les comparaisons entre mots sont effectuées de la gauche vers la droite (par positions croissantes), exactement m comparaisons de lettres sont exécutées à chaque étape ; comme leur nombre est $\lceil \log_2(n+1) \rceil$, cela donne la borne de la proposition.

Lorsque x est plus long que les éléments de L, l'ordre $O(\ell \times \log n)$ du temps d'exécution, où ℓ est la longueur maximale des mots de L, est une expression mieux adaptée.

4.2 Recherche avec les préfixes communs

En reprenant la méthode par dichotomie de la section précédente, il est possible d'accélérer la recherche de x dans la liste L grâce à l'utilisation d'une information supplémentaire sur les mots de la liste : leurs préfixes communs. Le temps de la recherche passe de $O(m \times \log n)$ (algorithme RECHERCHE-SIMPLE de la section précédente) à $O(m + \log n)$ pour l'algorithme RECHERCHE ci-dessous. Le stockage des longueurs de préfixes communs demande un espace mémoire supplémentaire $O(n)$.

L'idée de l'amélioration est contenue dans la proposition 4.3 qui est une remarque sur les préfixes communs. Dans l'énoncé de la proposition, les valeurs *ld* et *lf* et celles des variables associées dans l'algorithme RECHERCHE sont définies par :

$$ld = |lpc(x, L_d)|$$

et

$$lf = |lpc(x, L_f)| \ .$$

La proposition porte sur deux situations rencontrées au cours de l'exécution de l'algorithme RECHERCHE et qui sont illustrées par la figure 4.2. Un troisième cas est décrit dans la figure 4.3 ; c'est celui pour lequel des comparaisons lettre à lettre sont nécessaires. Trois autres cas symétriques sont à considérer quand on suppose $ld > lf$.

Proposition 4.3
Soient d, f, i trois entiers, $0 \leq d < i < f < n$. Sous les hypothèses $L_d \leq L_{d+1} \leq \cdots \leq L_f$ et $L_d < x < L_f$, soient $ld = |lpc(x, L_d)|$ et

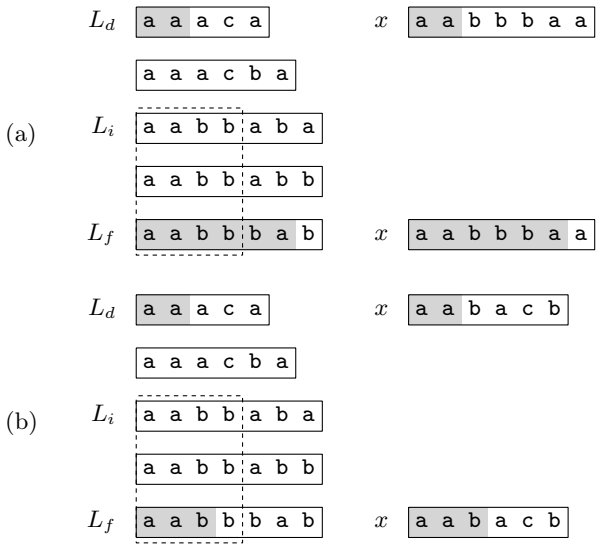


Figure 4.2 Illustration pour la preuve de la proposition 4.3 dans le cas $ld \leq lf$. **(a)** Soient $u = lpc(L_i, L_f)$ et a, b les lettres distinctes pour lesquelles $ua \preceq_{préf} L_i$ et $ub \preceq_{préf} L_f$. La liste étant classée, on a $a < b$. Alors, si $|u| = |lpc(L_i, L_f)| < lf$, ub est aussi un préfixe de x et donc $L_i < x < L_f$. Le raisonnement s'adapte au cas où $u = L_i$ et donne le même résultat. **(b)** Soient $v = lpc(x, L_f)$ et a, b les lettres distinctes pour lesquelles $va \preceq_{préf} x$ et $vb \preceq_{préf} L_f$. Comme $L_d < x < L_f$, on a $a < b$. Alors, si $|lpc(L_i, L_f)| > lf = |v|$, vb est aussi un préfixe de L_i et donc $L_d < x < L_i$. Le raisonnement s'adapte au cas où $v = x$ et donne le même résultat.

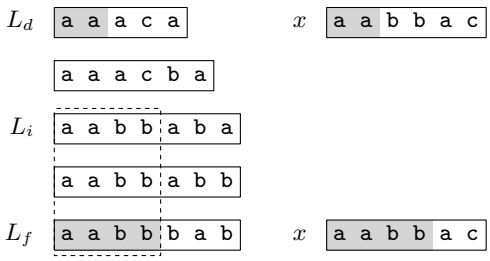


Figure 4.3 Illustration du fonctionnement de l'algorithme RECHERCHE pour un cas complémentaire de ceux de la proposition 4.3. On considère toujours le cas $ld \leq lf$. Soient $u = lpc(L_i, L_f)$ et a, b les lettres distinctes pour lesquelles $ua \preceq_{préf} L_i$ et $ub \preceq_{préf} L_f$. La liste étant classée, on en déduit $a < b$. Si $|u| = |lpc(L_i, L_f)| = lf$, ua' est un préfixe de x pour une lettre $a' < b$. Dans cette situation, il faut comparer lettre à lettre x et L_i pour localiser x dans la liste. Les comparaisons de lettres, effectuées de la gauche vers la droite, ne sont nécessaires qu'à partir de la position lf, où se trouvent les lettres a et a' dans leurs mots respectifs. L'algorithme prend en compte les possibilités $u = x$ et $u = L_i$.

$lf = |lpc(x, L_f)|$ satisfaisant $ld \leq lf$. On a alors :

$|lpc(L_i, L_f)| < lf$ *implique* $L_i < x < L_f$

et :

$|lpc(L_i, L_f)| > lf$ *implique* $L_d < x < L_i$.

Preuve La preuve se déduit de la légende de la figure 4.2. ∎

Le code de l'algorithme qui exploite la proposition 4.3 est donné ci-dessous. Il fait appel à la fonction *Lpc* définie comme suit. Pour (d, f), $-1 \leq d < f \leq n$, couple d'indices de la recherche dichotomique, on note

$Lpc(d, f) = |lpc(L_d, L_f)|$

la longueur maximale des préfixes communs à L_d et L_f.

```
RECHERCHE(L, n, Lpc, x, m)
 1  (d, ld) ← (−1, 0)
 2  (f, lf) ← (n, 0)
 3  tantque d + 1 < f faire
 4        ▷ Invariant : L_d < x < L_f
 5        i ← ⌊(d + f)/2⌋
 6        si ld ≤ Lpc(i, f) et Lpc(i, f) < lf alors
 7              (d, ld) ← (i, Lpc(i, f))              ▷ Figure 4.2(a)
 8        sinonsi ld ≤ lf et lf < Lpc(i, f) alors
 9              f ← i                                 ▷ Figure 4.2(b)
10        sinonsi lf ≤ Lpc(d, i) et Lpc(d, i) < ld alors
11              (f, lf) ← (i, Lpc(d, i))
12        sinonsi lf ≤ ld et ld < Lpc(d, i) alors
13              d ← i
14        sinon ℓ ← max{ld, lf}                       ▷ Figure 4.3
15              ℓ ← ℓ + |lpc(x[ℓ .. m − 1], L_i[ℓ .. |L_i| − 1])|
16              si ℓ = m et ℓ = |L_i| alors
17                    retourner i
18              sinonsi (ℓ = |L_i|) ou (ℓ ≠ m et L_i[ℓ] < x[ℓ]) alors
19                    (d, ld) ← (i, ℓ)
20              sinon (f, lf) ← (i, ℓ)
21  retourner (d, f)
```

Un exemple de fonctionnement de l'algorithme est donné figure 4.4.

On évalue la complexité de l'algorithme RECHERCHE sous l'hypothèse que le classement de la liste et le calcul des préfixes communs sont effectués au préalable. Cette préparation est étudiée dans la section suivante. Il s'en déduit que le calcul de $Lpc(r, s)$, $-1 \leq r < s \leq n$, se ramène à un accès à des valeurs tabulées et peut donc être exécuté en temps constant, propriété qui est utilisée dans la preuve de la proposition suivante.

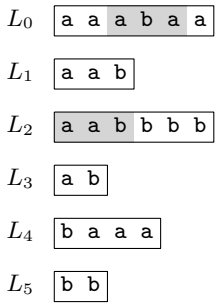


Figure 4.4 Pour rechercher le mot $x =$ `aaabb` dans la liste, l'algorithme RECHERCHE effectue six comparaisons de lettres (lettres grisées). Le résultat est le couple $(0, 1)$ qui indique que $L_0 < x < L_1$.

Proposition 4.4
L'algorithme RECHERCHE *localise un mot x de longueur m dans une liste classée de n mots (problème de l'appartenance) en temps $O(m + \log n)$ avec un maximum de $m + \lceil \log_2(n+1) \rceil$ comparaisons de lettres. L'algorithme demande un espace mémoire supplémentaire $O(n)$.*

Preuve Le code de l'algorithme RECHERCHE est une modification de celui de l'algorithme RECHERCHE-SIMPLE. Il prend en compte le résultat de la proposition 4.3. Le bon fonctionnement de l'algorithme résulte donc essentiellement des propositions 4.2 et 4.3, et de la légende de la figure 4.3.

Pour l'évaluation du temps d'exécution, on remarque que chaque comparaison positive accroit strictement la valeur de $\max\{ld, lf\}$ qui varie de 0 à m au maximum. Il y donc au plus m comparaisons de ce genre. Par ailleurs, chaque comparaison négative conduit à diviser par deux la quantité $f - d - 1$. Les comparaisons entre ld, lf et les valeurs des Lpc précalculées ont le même effet lorsqu'elles ne conduisent pas à des comparaisons de lettres. Il y a donc au plus $\lceil \log_2(n+1) \rceil$ comparaisons de ce genre. D'où le résultat annoncé sur le temps d'exécution quand le calcul de $Lpc(r, s)$, $-1 \leq r < s \leq n$, s'exécute en temps constant, ce que l'implantation décrite dans la section suivante réalise.

L'espace mémoire supplémentaire est utilisé pour stocker les informations sur le classement de la liste et sur les préfixes communs nécessaires aux calculs des $Lpc(r, s)$, $-1 \leq r < s \leq n$. L'implantation décrite dans la section suivante montre qu'un espace $O(n)$ est suffisant, résultat qui provient essentiellement de ce que $2n + 1$ couples (r, s) seulement interviennent dans la dichotomie d'après le lemme 4.1. ∎

L'algorithme RECHERCHE fournit une solution au problème de l'appartenance. Il se transforme facilement en une solution du problème de l'intervalle : l'algorithme INTERVALLE. Puisque l'on cherche maintenant

les mots de la liste dont x est préfixe, il faut pour cela détecter le cas où $x \preceq_{préf} L_i$. Cela se fait par modification du test de la ligne 16. Il reste ensuite à déterminer les bornes de l'intervalle cherché. On procède par dichotomie à gauche, puis à droite du mot L_i dont x est un préfixe, en recherchant les indices j pour lesquels $Lpc(i,j) \geq |x|$. Le principe du calcul repose sur le lemme 4.6 de la section 4.3.

L'algorithme INTERVALLE est obtenu en remplaçant les lignes 16–17 de l'algorithme RECHERCHE par les lignes qui suivent.

```
 1  ▷ Les lignes suivantes remplacent les lignes 16–17 de RECHERCHE
 2  si ℓ = m alors
 3        e ← i
 4        tantque d + 1 < e faire
 5              j ← ⌊(d + e)/2⌋
 6              si Lpc(j, e) < m alors
 7                    d ← j
 8              sinon e ← j
 9        si Lpc(d, e) ≥ m alors
10              d ← max{d − 1, −1}
11        e ← i
12        tantque e + 1 < f faire
13              j ← ⌊(e + f)/2⌋
14              si Lpc(e, j) < m alors
15                    f ← j
16              sinon e ← j
17        si Lpc(e, f) ≥ m alors
18              f ← min{f + 1, n}
19        retourner (d, f)
```

Les comparaisons de lettres effectuées par l'algorithme INTERVALLE sont les mêmes que celles exécutées par l'algorithme RECHERCHE. La majoration asymptotique du temps d'exécution n'est pas altérée par la modification ci-dessus. On obtient ainsi le résultat suivant.

Proposition 4.5
L'algorithme INTERVALLE *résout le problème de l'intervalle pour un mot de longueur m et une liste classée de n mots en temps $O(m + \log n)$ avec un maximum de $m + \lceil \log_2(n+1) \rceil$ comparaisons de lettres. L'algorithme demande un espace mémoire supplémentaire $O(n)$.* ∎

Il va de soi que la complexité temporelle est aussi $O(\ell + \log n)$ avec $\ell = \max\{|L_i| : i = 0, 1, \ldots, n-1\}$, majoration qu'il convient de retenir si x est plus long que les mots de la liste.

4.3 Préparation de la liste

L'algorithme RECHERCHE (ainsi que l'algorithme INTERVALLE) de la section précédente travaille sur une liste de mots L classée lexicographiquement et pour laquelle on connait les préfixes communs. On montre dans cette section comment effectuer ces opérations sur la liste.

Le tri d'une telle liste se réalise usuellement au moyen d'une méthode de classement par sélection de places (tri par bacs) analogue à la méthode utilisée par l'algorithme TRI de la section suivante. Par ce procédé de calcul, le classement s'exécute en temps $O(|L|)$, où, par abus de notation, $|L|$ désigne la somme des longueurs des mots de la liste.

On décrit ensuite une implantation des longueurs des préfixes communs nécessaires à l'algorithme RECHERCHE. Celle-ci est réalisée par mémorisation des valeurs dans une table à laquelle l'algorithme accède par l'intermédiaire de la fonction Lpc. On note

$LPC\colon \{0, 1, \ldots, 2n\} \to \mathbf{N}$

la table utilisée pour mémoriser les longueurs des préfixes communs. Elle est définie par :

- $LPC[f] = |lpc(L_{f-1}, L_f)|$, pour $0 \leq f \leq n$;
- $LPC[n+1+i] = |lpc(L_d, L_f)|$, pour $i = \lfloor (d+f)/2 \rfloor$ milieu d'un segment (d, f), $0 \leq d+1 < f \leq n$, de la recherche dichotomique,

en supposant que $lpc(L_r, L_s) = \varepsilon$ quand $r = -1$ ou $s = n$. La représentation des valeurs dans la table LPC ne provoque pas d'ambigüité car chaque indice i sur la table ne se rapporte qu'à un seul couple (d, f) intervenant pendant la recherche dichotomique.

L'égalité qui suit établit le lien entre la table LPC et la fonction Lpc :

$$Lpc(d, f) = \begin{cases} LPC[f] & \text{si } d+1 = f \ , \\ LPC[n+1+\lfloor (d+f)/2 \rfloor] & \text{sinon} \ . \end{cases}$$

On en déduit une implantation de la fonction Lpc qui s'exécute en temps constant. Ce résultat est une hypothèse utilisée dans la section précédente pour évaluer le temps d'exécution de l'algorithme RECHERCHE.

```
LPC(d, f)
1  si d + 1 = f alors
2          retourner LPC[f]
3  sinon retourner LPC[n + 1 + ⌊(d + f)/2⌋]
```

Le calcul de la table LPC se fait sur la liste L classée en ordre croissant. Le calcul de $LPC[f]$ pour $0 \leq f \leq n$ résulte de comparaisons lettre à lettre. Le lemme suivant fournit un moyen pour calculer les autres valeurs.

Lemme 4.6
On suppose $L_0 \leq L_1 \leq \cdots \leq L_{n-1}$. Soient d, i et f des entiers tels que $-1 < d < i < f < n$. Alors :

$$|lpc(L_d, L_f)| = \min\{|lpc(L_d, L_i)|, |lpc(L_i, L_f)|\} \ .$$

Preuve Soient $u = lpc(L_d, L_i)$ et $v = lpc(L_i, L_f)$. On suppose, par exemple, $|u| \leq |v|$, l'autre cas étant analogue. Les mots u et v étant des préfixes de L_i, on a alors $u \preceq_{préf} v$ et $u \preceq_{préf} L_f$.

Si $u = L_d$, on a ainsi $u = lpc(L_d, L_f)$, ce qui donne l'égalité cherchée.

Sinon, il existe trois lettres a, b, c telles que $ua \preceq_{préf} L_d$, $ub \preceq_{préf} L_i$ et $uc \preceq_{préf} L_f$. On a $a \neq b$ par définition de u, et même $a < b$ car la suite est en ordre croissant. Si de plus $b = c$, on obtient $u = lpc(L_d, L_f)$, ce qui donne la conclusion. Si par contre $b \neq c$, la suite étant en ordre croissant, on a $b < c$, ce qui donne la même conclusion et achève la preuve. ∎

L'algorithme DÉF-LPC implante le calcul de la table *LPC*. L'exécution commence par l'appel DÉF-LPC$(-1, n)$, pour $n \geq 0$, qui a pour effet de calculer toutes les entrées de la table. Le résultat correspond à sa définition ci-dessus, et le calcul utilise le lemme précédent à la ligne 8.

```
DÉF-LPC(d, f)
 1  ▷ On a d < f
 2  si d + 1 = f alors
 3        si d = −1 ou f = n alors
 4              LPC[f] ← 0
 5        sinon LPC[f] ← |lpc(L_d, L_f)|
 6        retourner LPC[f]
 7  sinon i ← ⌊(d + f)/2⌋
 8        LPC[n + 1 + i] ← min{DÉF-LPC(d, i), DÉF-LPC(i, f)}
 9        retourner LPC[n + 1 + i]
```

On vérifie que le temps d'exécution de DÉF-LPC$(-1, n)$ est $O(|L|)$ comme conséquence du lemme 4.1. La proposition qui suit résume les éléments discutés dans la section.

Proposition 4.7
La préparation de la liste L, tri puis calcul de la table LPC, prend un temps $O(|L|)$. ∎

4.4 Tri des suffixes

La technique des sections précédentes peut être appliquée à la liste des suffixes d'un mot et est à la base d'une implantation d'index décrite dans le chapitre 6. Le problème de l'intervalle et sa solution, l'algorithme

INTERVALLE, prennent tout leur intérêt dans ce type d'application à laquelle ils s'adaptent sans modification.

Dans cette section on montre comment classer en ordre lexicographique les suffixes d'un mot y de longueur n, condition préalable pour exécuter l'algorithme INTERVALLE sur la liste des suffixes de y. Dans la section suivante, on complète la préparation du mot y en montrant comment calculer efficacement les préfixes communs aux suffixes de y. La permutation qui résulte du classement et la table des préfixes communs constituent la ***table des suffixes*** du mot à indexer.

L'objectif du classement est de calculer une permutation p des indices sur y qui satisfait la condition :

$$y[p[0]\,..\,n-1] < y[p[1]\,..\,n-1] < \cdots < y[p[n-1]\,..\,n-1] \ . \tag{4.1}$$

On note que les inégalités sont strictes car deux suffixes apparaissant à des positions distinctes ne peuvent être identiques.

La mise en œuvre d'une méthode de classement lexicographique standard, comme celle qui est suggérée à la section 4.3, conduit à un algorithme dont le temps d'exécution est $O(n^2)$ car la somme des longueurs des suffixes de y est quadratique. La méthode de classement que l'on utilise ici repose sur une technique d'identification partielle des suffixes de y au moyen de leurs k premières lettres. Les valeurs de k croissent de façon exponentielle, ce qui produit un classement en $\lceil \log_2 n \rceil$ étapes. Chaque étape est réalisée en temps linéaire à l'aide d'un tri lexicographique sur des paires d'entiers de taille limitée, classement qui peut être réalisé par sélection de places (tri par bacs).

Soit un entier k, $k > 0$. On note, pour $u \in A^*$:

$$prem_k(u) = \begin{cases} u & \text{si } |u| \leq k \ , \\ u[0\,..\,k-1] & \text{sinon} \ , \end{cases}$$

le début d'ordre k du mot u. On définit pour les positions $0, 1, \ldots, n-1$ sur y une suite de fonctions de rang R_k de la manière suivante. On note $R_k[i]$ le rang (compté à partir de 0) de $prem_k(y[i\,..\,n-1])$ dans la liste en ordre croissant des mots de l'ensemble $\{prem_k(u) : u \preceq_{suff} y \text{ et } u \neq \varepsilon\}$. Cet ensemble contient en général moins de n éléments pour de faibles valeurs de k, ce qui fait que des positions différentes peuvent se voir attribuer une même valeur par R_k. Ainsi, R_k induit une relation d'équivalence sur les positions. Elle est notée $\equiv_k$, et définie par :

$i \equiv_k j$

si et seulement si

$R_k[i] = R_k[j]$.

Lorsque $k = 1$, l'équivalence $\equiv_1$ revient à identifier les lettres de y. Pour $k \in \mathbf{N}$ quelconque, deux suffixes de longueur au moins k sont équivalents vis-à-vis de $\equiv_k$ si leurs préfixes de longueur k sont égaux. Lorsque

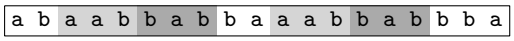


Figure 4.5 Doublement. Le rang du facteur `aabbab`, $R_6[2]$, est déterminé par les rangs $R_3[2]$ et $R_3[5]$ de `aab` et `bab` respectivement. En particulier, `aabbab` apparait aux positions 2 et 10 car `aab` apparait aux positions 2 et 10, et à la fois `bab` apparait aux positions 5 et 13.

$k \geq n$, l'équivalence $\equiv_k$ est discrète : chaque suffixe n'est équivalent qu'à lui-même.

Pour simplifier l'énoncé de la propriété qui est à la base de l'algorithme de classement des suffixes ci-dessous, on étend la définition de R_k en posant $R_k[i] = -1$ pour $i \geq n$.

Lemme 4.8 (lemme de doublement)
Pour k, i entiers, $k > 0$ et $0 \leq i < n$, $R_{2k}[i]$ est le rang du couple $(R_k[i], R_k[i+k])$ dans la liste classée en ordre lexicographique croissant de ces couples.

Preuve Poser $R_k[i] = -1$ pour un entier $i \geq n$ revient à considérer le mot infini ya^∞, où a est une lettre strictement inférieure à toutes celles qui apparaissent dans y. Ainsi, lorsque $i \geq n$, le facteur de longueur k apparaissant à la position i est a^k qui est inférieur à tous les autres mots de même longueur apparaissant dans y. Son rang est donc strictement inférieur aux autres rangs, ce qui est compatible avec la convention de lui donner la valeur -1.

Par définition, $R_{2k}[i]$ est le rang de $prem_{2k}(y[i\,..\,i+2k-1])$ dans la liste classée des facteurs de longueur $2k$ du mot ya^∞. On note :

$$u(i) = prem_k(y[i\,..\,i+k-1])$$

et :

$$v(i) = prem_k(y[i+k\,..\,i+2k-1])\ .$$

De l'égalité :

$$prem_{2k}(y[i\,..\,i+2k-1]) = u(i) \cdot v(i)$$

(voir figure 4.5) on déduit, pour $0 \leq i \neq j < n$, que l'inégalité :

$$prem_{2k}(y[i\,..\,i+2k-1]) < prem_{2k}(y[j\,..\,j+2k-1])$$

est équivalente à :

$$(u(i), v(i)) < (u(j), v(j))$$

qui elle-même est équivalente à :

$$(R_k[i], R_k[i+k]) < (R_k[j], R_k[j+k])\ ,$$

par définition de R_k. Le rang $R_{2k}[i]$ de $prem_{2k}(y[i\,..\,i+2k-1])$ est donc égal au rang de $(R_k[i], R_k[i+k])$ dans la suite croissante de ces couples, ce qui achève la preuve. ∎

Relativement au paramètre k, on note enfin p_k une permutation des positions sur y qui satisfait, pour $0 \leq r < s < n$,

$R_k[p_k(r)] \leq R_k[p_k(s)]$.

La permutation est associée à la suite classée des débuts de longueur k des suffixes de y. Lorsque $k \geq n$, les mots $prem_k(u)$ (où u est suffixe non vide de y) étant deux à deux distincts, l'inégalité précédente devient stricte. On obtient alors une seule permutation satisfaisant la condition, c'est la permutation définie par la table p et utilisée dans la section 6.1 pour effectuer la recherche de facteurs dans le mot y. L'algorithme TRI-SUFFIXES calcule cette permutation au moyen des tables R_k.

```
TRI-SUFFIXES(y, n)
 1  pour r ← 0 à n − 1 faire
 2        p[r] ← r
 3  k ← 1
 4  pour i ← 0 à n − 1 faire
 5        R1[i] ← rang de y[i] dans la liste classée
             des lettres de alph(y)
 6  p ← TRI(p, n, R1, 0)
 7  i ← card alph(y) − 1
 8  tantque i < n − 1 faire
 9        p ← TRI(p, n, Rk, k)
10        p ← TRI(p, n, Rk, 0)
11        i ← 0
12        R2k[p[0]] ← i
13        pour r ← 1 à n − 1 faire
14              si Rk[p[r]] ≠ Rk[p[r − 1]]
                   ou Rk[p[r] + k] ≠ Rk[p[r − 1] + k] alors
15                    i ← i + 1
16              R2k[p[r]] ← i
17        k ← 2k
18  retourner p
```

Une illustration du fonctionnement de l'algorithme TRI-SUFFIXES est donnée figure 4.6.

L'algorithme utilise la propriété énoncée dans le lemme 4.8 en faisant appel à l'algorithme de classement TRI décrit ci-dessous. En entrée de TRI, p est une permutation des entiers $0, 1, \ldots, n-1$, R est une table sur ces entiers à valeurs dans l'ensemble $\{-1, 0, \ldots, n-1\}$, et k est un entier. L'algorithme TRI classe la suite d'entiers

$p[0] + k, p[1] + k, \ldots, p[n-1] + k$

en ordre croissant de leur clé R. C'est-à-dire que la valeur p' produite par $\text{TRI}(p, n, R, k)$ est une permutation de $\{0, 1, \ldots, n-1\}$ qui satisfait

(a)

i	0	1	2	3	4	5	6	7	8	9	10
$y[i]$	a	a	b	a	a	b	a	a	b	b	a

(b)

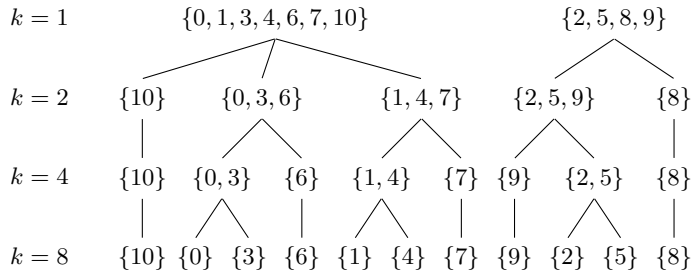


Figure 4.6 Calcul par doublement des partitions associées aux équivalences $\equiv_k$ sur le mot $y =$ `aabaabaabba`. **(a)** Les positions sur le mot y. **(b)** Les classes de positions selon $\equiv_k$ sont notées de gauche à droite en ordre croissant du rang commun de leurs éléments. Ainsi, ligne $k = 2$, $R_2[10] = 0$, $R_2[0] = R_2[3] = R_2[6] = 1$, $R_2[1] = R_2[4] = R_2[7] = 2$, etc. Pour $k = 8$, la suite des positions fournit les suffixes en ordre croissant.

les inégalités :

$$R[p'[0] + k] \leq R[p'[1] + k] \leq \cdots \leq R[p'[n-1] + k] \ .$$

De plus, Tri satisfait une condition de stabilité qui le rend approprié au classement lexicographique. À savoir que l'algorithme ne modifie pas la position relative dans la liste de deux éléments qui possèdent la même clé de classement. En d'autres termes, si r et t, $0 \leq r \neq t < n$, sont deux entiers pour lesquels $R[p[r] + k] = R[p[t] + k]$, on a $p[r] < p[t]$ si et seulement si $p'[r] < p'[t]$. L'implantation ci-dessous satisfait les propriétés requises.

```
TRI(p, n, R, k)
 1  pour i ← −1 à n − 1 faire
 2        Bac[i] ← FILE-VIDE()
 3  pour r ← 0 à n − 1 faire
 4        si p[r] + k < n alors
 5              i ← R[p[r] + k]
 6        sinon i ← −1
 7        ENFILER(Bac[i], p[r])
 8  r ← −1
 9  pour i ← −1 à n − 1 faire
10        tantque non FILE-EST-VIDE(Bac[i]) faire
11              j ← DÉFILEMENT(Bac[i])
12              r ← r + 1
13              p'[r] ← j
14  retourner p'
```

Proposition 4.9
L'algorithme TRI-SUFFIXES *appliqué au mot y classe ses suffixes en ordre croissant.*

Preuve Les instructions de la boucle **tantque** servent d'une part à classer les entiers $p[0], p[1], \ldots, p[n-1]$, puis à définir les nouveaux rangs qui leur sont associés. Cette deuxième partie est assurée par la boucle interne **pour** et se vérifie aisément. La partie importante est constituée par le tri.

On montre ci-dessous que la condition :

$$\langle prem_k(y[p[r]\,..\,n-1]) : r = 0, 1, \ldots, n-1\rangle \text{ est croissante} \tag{4.2}$$

est invariante par la boucle **tantque**.

Soient $p' = \text{TRI}(p, n, R_k, k)$ et $p'' = \text{TRI}(p', n, R_k, 0)$ pendant une exécution des instructions de la boucle **tantque**. La condition de stabilité imposée à TRI a pour conséquence, pour $0 \leq r < n$ et $0 \leq t < n$, que :

$$p''[r] < p''[t]$$

implique

$$(R_k[p[r]], R_k[p[r]+k]) \leq (R_k[p[t]], R_k[p[t]+k]) \ .$$

Ainsi, d'après le lemme 4.8, le rang attribué à $p''[r]$, $0 \leq r < n$, est $R_{2k}[p[r]]$. Ce qui signifie que juste avant l'exécution de l'instruction 17 on a : $\langle prem_{2k}(y[p''[r]\,..\,n-1]) : r = 0, 1, \ldots, n-1\rangle$ est croissante. Juste après l'exécution de l'instruction 17 la condition (4.2) est donc remplie, ce qui prouve son invariance.

On vérifie directement que la condition (4.2) est satisfaite avant l'exécution de la boucle **tantque** grâce à l'instruction de la ligne 6. Elle l'est donc encore après l'exécution de cette boucle (pour la terminaison, voir la preuve de la proposition 4.10). On a alors $i \leq n$ et, plus exactement, $i = n-1$, car la valeur finale de i est le rang maximal des facteurs $prem_k(u)$, pour u suffixe non vide de y, qui ne peut être supérieur à $n-1$.

On montre que les suffixes sont en ordre croissant relativement à la permutation p. Soient u, w deux suffixes non vides de y ($u \neq w$) de positions respectives $p[r]$ et $p[t]$, $p[r] < p[t]$. Par la condition (4.2) on obtient l'inégalité $R_k[p[r]] \leq R_k[p[t]]$. Mais les rangs étant deux à deux distincts, on en déduit même $R_k[p[r]] < R_k[p[t]]$, ce qui est équivalent à $prem_k(u) < prem_k(w)$. Cette inégalité signifie que, ou bien $prem_k(u)$ est un préfixe propre de $prem_k(w)$, ou bien $prem_k(u) = vau'$ et $prem_k(w) = vbw'$ avec $v, u', w' \in A^*$, $a, b \in A$ et $a < b$. Dans le premier cas, on a nécessairement $prem_k(u) = u$ qui est donc un préfixe propre de $prem_k(w)$ et ainsi de w. Donc $u < w$. Dans le deuxième cas, on a $u = vau''$ et $w = vbw''$ pour deux mots u'' et w'', ce qui montre que l'on a encore $u < w$.

La permutation p qui est produite par l'algorithme TRI-SUFFIXES satisfait la condition (4.1) et correspond donc à la suite croissante des suffixes de y. ∎

Proposition 4.10
Le temps d'exécution de l'algorithme TRI-SUFFIXES *appliqué au mot y de longueur n est $O(n \times \log n)$. L'algorithme travaille dans un espace $O(n \times \log n)$ lorsque les tables R_k doivent être conservées, et dans un espace $O(n)$ dans le cas contraire.*

Preuve Les lignes 4–5 qui se réfèrent implicitement à un classement des lettres de y s'exécutent en temps $O(n \times \log n)$. Les autres instructions situées avant la boucle **tantque**, internes à cette boucle et après cette boucle s'exécutent en temps $O(n)$. Le temps d'exécution global dépend donc du nombre d'itérations de cette boucle.

Comme pour $k \geq n$, les mots $prem_k(u)$ (u suffixe non vide de y) sont deux à deux distincts, leur rang maximal est exactement $n-1$, ce qui constitue la condition d'arrêt de la boucle **tantque**. Les valeurs successives de k sont $2^0, 2^1, 2^2, \ldots$ jusqu'à $2^{\lceil \log_2(n-1) \rceil}$ au plus, ce qui limite le nombre d'itérations de la boucle à $\lceil \log_2(n-1) \rceil$. D'où la majoration du temps d'exécution de TRI-SUFFIXES.

Une autre conséquence en est que le nombre de tables R_k utilisées par l'algorithme est majoré par $\lceil \log_2(n-1) \rceil + 1$ (une nouvelle table par itération). Ce qui nécessite un espace $O(n \times \log n)$ si elles doivent être toutes conservées. Dans le cas contraire, on constate qu'une seule table R suffit au calcul, soit un espace supplémentaire $O(n)$ pour cette table. La même quantité est nécessaire pour implanter les bacs utilisés par le tri. ∎

4.5 Préfixes communs des suffixes

On décrit dans cette section le deuxième élément constitutif d'une table des suffixes, la table des longueurs des préfixes communs aux suffixes. Cette donnée complète la permutation des suffixes étudiée dans la section précédente, et permet l'utilisation de l'algorithme INTERVALLE de la section 4.2.

Le calcul des plus longs préfixes communs reprend la méthode de la section 4.3, réalisée par l'algorithme DÉF-LPC, en l'adaptant toutefois pour réduire son temps d'exécution. L'algorithme DÉF-LPC s'applique à une liste classée de mots. La liste L que l'on considère ici est la liste ordonnée des suffixes de y, c'est-à-dire :

$$y[p[0]\,..\,n-1], y[p[1]\,..\,n-1], \ldots, y[p[n-1]\,..\,n-1] \ ,$$

où p est la permutation, calculée par l'algorithme TRI-SUFFIXES, qui

(a)

i	0	1	2	3	4	5	6	7	8	9	10	11
$y[i]$	a	a	b	a	a	b	a	a	b	b	a	
$p[i]$	10	0	3	6	1	4	7	9	2	5	8	
$LPC[i]$	0	1	6	3	1	5	2	0	2	4	1	0

(b)

i	12	13	14	15	16	17	18	19	20	21	22
$LPC[i]$	0	1	0	1	1	0	0	0	0	0	0

Figure 4.7 Table des suffixes du mot $y =$ `aabaabaabba` qui comprend les tables p et LPC. **(a)** La table p donne la liste des suffixes en ordre croissant : le premier commence à la position 10, le second à la position 0, etc. La table LPC contient les longueurs des plus longs préfixes communs. Pour $0 \leq i \leq 11$, $LPC[i] = |lpc(y[p[i]\,..\,10], y[p[i-1]\,..\,10])|$. Les positions -1 et 11 sont assimilées à des lettres qui sont respectivement plus petite et plus grande que toutes les autres lettres de y. **(b)** Pour $12 \leq i \leq 22$, $LPC[i] = |lpc(y[p[d]\,..\,10], y[p[f]\,..\,10])|$ où (d, f) est un couple qui intervient lors de la recherche dichotomique ($0 \leq d+1 < f \leq 11$) et $i = 12 + \lfloor (d+f)/2 \rfloor$. Par exemple la valeur de $LPC[15]$ ($= 1$) est celle de $|lpc(y[p[2]\,..\,10], y[p[5]\,..\,10])|$ (car $15 = 12 + \lfloor (2+5)/2 \rfloor$), c'est-à-dire celle de $|lpc(y[3\,..\,10], y[4\,..\,10])|$ (car $p[2] = 3$ et $p[5] = 4$), soit $|lpc($`aabaabba`$,$ `abaabba`$)|$ qui est bien 1.

satisfait la condition (4.1) :

$$y[p[0]\,..\,n-1] < y[p[1]\,..\,n-1] < \cdots < y[p[n-1]\,..\,n-1] \ .$$

Le but de cette section est de montrer comment on peut calculer la table LPC associée à la liste L comme dans la section 4.3. La figure 4.7 illustre le résultat recherché pour le mot `aabaabaabba`.

L'utilisation directe de DÉF-LPC (section 4.3) pour faire le calcul conduit à un temps d'exécution $O(n^2)$ puisque la somme des longueurs des suffixes est quadratique. On décrit un algorithme, DÉF-LPC-SUFF, qui effectue le calcul en temps linéaire. L'adaptation de DÉF-LPC réside dans une optimisation du calcul du plus long préfixe commun à deux suffixes consécutifs dans l'ordre lexicographique. Dans DÉF-LPC (ligne 5) le calcul est censé se faire par comparaisons lettre à lettre. Il est d'ailleurs difficile de faire autrement sans autre information sur les mots de la liste. La situation est différente pour les suffixes de y car ceux-ci ne sont pas indépendants les uns des autres. C'est cette dépendance qui permet de réduire le temps de calcul au moyen d'un algorithme assez simple, basé sur le lemme qui suit illustré par la figure 4.8.

Lemme 4.11
Soient i, j, i' des positions sur y pour lesquelles $j = p[i]$ et $j - 1 = p[i']$. Alors $LPC[i'] - 1 \leq LPC[i]$.

Preuve Soit u le plus long préfixe commun à $y[j-1\,..\,n-1]$ et à son prédécesseur dans l'ordre lexicographique, disons $y[k\,..\,n-1]$. On a $LPC[i'] = |u|$ par définition.

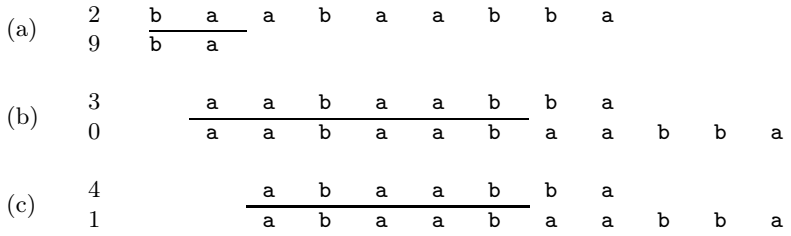


Figure 4.8 Illustration du lemme 4.11 sur le mot $y =$ `aabaabaabba` de la figure 4.7. On considère les plus longs préfixes communs des suffixes aux positions 2, 3 et 4 avec leurs prédécesseurs dans l'ordre lexicographique. **(a)** $p[8] = 2$, $p[7] = 9$ et $LPC[8] = lpc(y[2\,..\,10], y[9\,..\,10]) = 2$. **(b)** Avec les notations du lemme, en choisissant $j = 3$ on obtient $i = 2$ car $p[2] = 3$, et $i' = 8$ car $p[8] = p[2] - 1 = 2$. Dans ce cas $LPC[2] = lpc(y[3\,..\,10], y[0\,..\,10]) = 6$, quantité qui est strictement supérieure à $LPC[8] - 1$. **(c)** En choisissant $j = 4$ on obtient $i = 5$ car $p[5] = 4$, et $i' = 2$ car $p[2] = p[5] - 1 = 3$. On a $LPC[5] = lpc(y[4\,..\,10], y[1\,..\,10]) = 5$. Dans ce cas on a l'égalité $LPC[5] = LPC[2] - 1$.

Si u est le mot vide le résultat est satisfait car $LPC[i] \geq 0$. Sinon, u s'écrit cv où $c = y[j-1]$ et $v \in A^*$. Le mot $y[j-1\,..\,n-1]$ admet alors pour préfixe cvb pour une lettre b, et son prédécesseur admet pour préfixe cva pour une lettre a telle que $a < b$ à moins qu'il soit égal à cv.

Ainsi, v est un préfixe commun à $y[j\,..\,n-1]$ et $y[k+1\,..\,n-1]$. De plus, $y[k+1\,..\,n-1]$, qui commence par va ou est égal à v, est inférieur à $y[j\,..\,n-1]$ qui commence par vb. Donc $LPC[i]$ qui est la longueur maximale des préfixes communs à $y[j\,..\,n-1]$ et à son prédécesseur dans l'ordre lexicographique ne peut être inférieure à $|u|$ (conséquence du lemme 4.6). On a donc $LPC[i] \geq |u| = LPC[i'] - 1$ ce qui donne le résultat également lorsque u est non vide. ∎

Fort de la conclusion du lemme précédent, pour calculer $LPC[i]$ lorsque $0 < i \leq n$, c'est-à-dire $|lpc(y[p[i]\,..\,n-1], y[p[i-1]\,..\,n-1])|$, on peut commencer les comparaisons lettre à lettre exactement à la position à laquelle s'est arrêté le calcul précédent, celui de $LPC[i']$. Pour cela il suffit de procéder en considérant les suffixes du plus long au plus court, et non pas dans l'ordre lexicographique qui parait pourtant plus naturel. C'est ce que réalise l'algorithme DÉF-DEMI-LPC qui calcule les valeurs $LPC[i]$ pour $0 \leq i \leq n$, c'est-à-dire les longueurs maximales des préfixes communs des suffixes consécutifs dans l'ordre lexicographique. Notons que pour déterminer la position i associée à la position j, l'algorithme utilise l'inverse de la permutation p qui est calculée dans une première étape (lignes 1–2). Cette fonction est représentée par la table notée R car elle indique le rang de chaque suffixe dans la liste classée des suffixes de y. La seconde étape de l'algorithme applique le lemme 4.11.

```
DÉF-DEMI-LPC(y, n, p)
 1  pour i ← 0 à n − 1 faire
 2        R[p[i]] ← i
 3  ℓ ← 0
 4  pour j ← 0 à n − 1 faire
 5        ℓ ← max{0, ℓ − 1}
 6        i ← R[j]
 7        si i ≠ 0 alors
 8              j′ ← p[i − 1]
 9              tantque j + ℓ < n et j′ + ℓ < n
                  et y[j + ℓ] = y[j′ + ℓ] faire
10                    ℓ ← ℓ + 1
11        sinon ℓ ← 0          ▷ instruction facultative
12        LPC[i] ← ℓ
13  LPC[n] ← 0
```

Proposition 4.12
Appliqué au mot y de longueur n et à la permutation p de ses suffixes classés, l'algorithme DÉF-DEMI-LPC *calcule $LPC[i]$, pour $0 \leq i \leq n$, en temps $O(n)$.*

Preuve Considérons d'abord l'exécution des instructions aux lignes 5–12 pour $j = 0$. Si $i = 0$ à la ligne 7, la valeur de ℓ est nulle et c'est par définition celle de $LPC[i]$ car y lui-même est son suffixe minimal. Sinon, comme $\ell = 0$ avant l'exécution de la boucle **tantque**, juste après, on a $\ell = |lpc(y[j\,..\,n-1], y[j'\,..\,n-1])|$. D'après le calcul de la table R effectué aux lignes 1–2, et comme p est bijective, on a $j = p[i]$. Et d'après la ligne 8, on a aussi $j' = p[i-1]$. Donc la valeur de ℓ est bien celle de $LPC[i]$, c'est-à-dire $|lpc(y[p[i]\,..\,n-1], y[p[i-1]\,..\,n-1])|$.

Considérons ensuite une position j, $0 < j < n$. Si $i = 0$, l'argument utilisé précédemment est aussi valable ici car $y[j\,..\,n-1]$ est alors le suffixe minimal de y. Supposons ensuite que $i = R[j]$ est non nul, soit $i > 0$. Par définition $LPC[i] = |lpc(y[p[i]\,..\,n-1], y[p[i-1]\,..\,n-1])|$ et donc, d'après l'égalité $j = p[i]$ et la valeur de j' calculée à la ligne 8, $LPC[i] = |lpc(y[j\,..\,n-1], y[j'\,..\,n-1])|$. Les comparaisons effectuées pendant l'exécution de la boucle **tantque** calcule donc la longueur maximale des préfixes communs à $y[j\,..\,n-1]$ et $y[j'\,..\,n-1]$ à partir de la position d'indice ℓ sur $y[j\,..\,n-1]$ (c'est-à-dire $j + \ell$ sur y) par application du lemme 4.11. Le résultat est correct à condition que la valeur initiale de ℓ à cette étape soit bien égale à $LPC[i']$ pour la position i' telle que $p[i'] = j - 1$. Mais ceci résulte d'un raisonnement par itération qui commence par la validité du calcul pour $j = 0$ qui est démontrée ci-dessus.

Enfin, la valeur de $LPC[n]$ est correctement calculée à la ligne 13 car celle-ci est nulle par définition.

La plupart des instructions de l'algorithme s'exécutent une fois pour

chacune des n valeurs de i ou de j. Il reste à vérifier que le temps d'exécution de la boucle **tantque** est aussi $O(n)$. Ceci provient du fait que chaque comparaison positive de lettres à la ligne 9 accroit d'une unité la valeur de $j + \ell$ qui n'est jamais diminuée par la suite et varie de 0 à n au maximum. Ceci termine la preuve. ∎

Remarquons que l'instruction à la ligne 11 de l'algorithme DÉF-DEMI-LPC est facultative. On peut prouver cette remarque par un raisonnement analogue à celui de la preuve du lemme 4.11 et en notant que $y[j\,..\,n-1]$ est le suffixe minimal de y dans la situation concernée.

L'algorithme DÉF-LPC-SUFF ci-dessous complète le calcul de la table *LPC* (valeurs $LPC[i]$ pour $n+1 \leq i \leq 2n$). Il s'applique à la table *LPC* partiellement calculée par l'algorithme précédent. Par rapport à l'algorithme DÉF-LPC, les lignes 3–5 de celui-ci sont supprimées car la valeur de $LPC[f]$ considérée est déjà connue. Pour alléger l'écriture de l'algorithme, la permutation p est étendue en posant $p[-1] = -1$ et $p[n] = n$.

DÉF-LPC-SUFF(d, f)
1 ▷ On a $d < f$
2 **si** $d + 1 = f$ **alors**
3 **retourner** $LPC[f]$ ▷ déjà calculé par DÉF-DEMI-LPC
4 **sinon** $i \leftarrow \lfloor (d+f)/2 \rfloor$
5 $LPC[n+1+i] \leftarrow \min \begin{cases} \text{DÉF-LPC-SUFF}(d, i) \\ \text{DÉF-LPC-SUFF}(i, f) \end{cases}$
6 **retourner** $LPC[n+1+i]$

Proposition 4.13
Les exécutions successives de DÉF-DEMI-LPC(y, n, p) *et de* DÉF-LPC-SUFF$(-1, n)$ *appliqués à la liste en ordre croissant des suffixes du mot y de longueur n produisent la table de leurs préfixes communs, LPC, en temps $O(n)$.*

Preuve La validité du calcul repose sur celle de DÉF-DEMI-LPC (proposition 4.12) et sur le lemme 4.6 pour celle de DÉF-LPC-SUFF.

Le temps d'exécution de DÉF-DEMI-LPC(y, n, p) est linéaire d'après la proposition 4.12. En dehors des appels récursifs, l'exécution de DÉF-LPC-SUFF(d, f) prend un temps constant pour chaque couple (d, f). Comme il y a $2n + 1$ couples de la sorte (lemme 4.1) on obtient encore un temps linéaire pour la seconde exécution, ce qui donne le résultat annoncé. ∎

Avec cette section se terminent les algorithmes de préparation d'une table des suffixes, table qui sert de base à une implantation d'index (voir chapitre 6).

Notes

La table des suffixes d'un mot, avec l'algorithme de recherche associé basé sur la connaissance des préfixes communs (section 4.2), est due à Manber et Myers (1993). On obtient de cette façon une méthode pour la réalisation d'index (voir chapitre 6) qui, sans être optimale, est assez légère à implanter et économique en espace mémoire comparativement aux structures du chapitre 5.

Le classement des suffixes d'un mot tel que présenté dans ce chapitre est une variation sur un procédé introduit par Karp, Miller et Rosenberg (1972). Cette technique, appelée nommage, qui comprend essentiellement l'utilisation de la fonction de rang et le lemme de doublement, a été une des premières méthodes efficaces pour la recherche de répétitions et la localisation de motifs dans les données textuelles. Le nommage s'adapte aussi à des données non séquentielles comme les images ou les arbres. La méthode utilisée dans la section 4.5 pour calculer les préfixes communs aux suffixes classés est due à Kasai et co-auteurs (2001). Le chapitre 9 présente un autre moyen pour préparer une table des suffixes, moyen qui est plus proche de celui proposé originellement par Manber et Myers.

Il est bien sûr possible de préparer une table des suffixes en utilisant les structures de données développées dans le chapitre suivant. Mais on perd alors une partie des avantages de la méthode car les structures sont plus gourmandes en mémoire. D'ailleurs, la table des suffixes peut aussi être vue comme une implantation particulière de l'arbre des suffixes du chapitre suivant.

Exercices

4.1 (Tous les préfixes communs)
Soit L une liste classée de n mots, $L_0 \leq L_1 \leq \cdots \leq L_{n-1}$, de longueur commune n. Décrire un algorithme de calcul des $|lpc(L_i, L_j)|$, $0 \leq i \neq j < n$, qui fonctionne en temps (optimal) $O(n^2)$.

4.2 (Gain en mémoire)
Étudier la possibilité de réduire l'espace nécessaire à la mémorisation de la table *LPC* sans modifier les majorations des temps d'exécution des algorithmes qui utilisent ou calculent la table.

4.3 (Triche)
Décrire le calcul des tables p et *LPC* au moyen d'une des structures d'automates du chapitre suivant, arbre compact des suffixes et automate des suffixes. Quelle est la complexité, temps et espace, de la préparation des tables ? Que deviennent ces valeurs lorsque l'alphabet est fixé ? Montrer

qu'en particulier dans cette situation, le temps de construction des tables est linéaire.

4.4 (Tst !)
Soit $X = \langle x_0, x_1, \ldots, x_{k-1} \rangle$ une suite de k mots deux-à-deux distincts sur l'alphabet A, muni d'une relation d'ordre. En notant $a = x_0[0]$, on définit G comme la sous-suite des mots de X qui commencent par une lettre strictement inférieure à a, et D comme celle des mots qui commencent par une lettre strictement supérieure à a. On note en plus $C = \langle u_0, u_1, \ldots, u_{\ell-1} \rangle$ la suite pour laquelle $\langle au_0, au_1, \ldots, au_{\ell-1} \rangle$ est la sous-suite des mots de X qui commencent par la lettre a.

L'arbre digital de recherche associé à X, noté $\mathcal{T}(X)$, est la structure R définie comme suit :

$$R = \begin{cases} \text{vide} & \text{si } k = 0 \ , \\ \langle r \rangle & \text{si } k = 1 \ , \\ \langle r, g(R), (a, c(R)), d(R) \rangle & \text{sinon} \ , \end{cases}$$

où r est la racine de R ; $g(R)$, son sous-arbre gauche, est $\mathcal{T}(G)$; $c(R)$, son sous-arbre central, est $\mathcal{T}(C)$; et $d(R)$, son sous-arbre droit, est $\mathcal{T}(D)$. Les feuilles de l'arbre sont étiquetées par des mots : dans la définition précédente, si $k = 1$, l'arbre est constitué d'une seule feuille r dont l'étiquette est le mot x_0. On notera que $R = \mathcal{T}(X)$ est un arbre ternaire, et que le lien entre sa racine et celle de son sous-arbre central porte comme étiquette $a = x_0[0]$, première lettre du premier mot de X.

Décrire les algorithmes de gestion des arbres digitaux de recherche (recherche, insertion, suppression, ...). Évaluer la complexité des opérations. [*Aide :* voir les *Ternary Search Trees* de Bentley et Sedgewick (1997).]

4.5 (En moyenne)
Montrer que la longueur moyenne des préfixes communs aux suffixes d'un mot y est $O(\log |y|)$. Que peut-on en déduire sur le temps moyen de la recherche de x dans y au moyen de la table des suffixes de y ? Que peut-on en déduire sur les temps moyens du tri des suffixes et du calcul des préfixes communs par les algorithmes des sections 4.4 et 4.5 ?

4.6 (Doublement d'images)
Sur des images, matrices de lettres, adapter la fonction $prem_k$ et prouver un lemme de doublement correspondant.

4.7 (Facultatif)
Montrer que l'instruction ligne 11 de l'algorithme DÉF-DEMI-LPC peut être supprimée sans que cela altère le bon fonctionnement de l'algorithme ni son temps d'exécution.

4.8 (Suffixes d'images)
Sur des images, introduire une relation d'ordre permettant de profiter de la méthodologie présentée dans ce chapitre.

5 Structures pour index

On présente dans ce chapitre des structures de données pour mémoriser les suffixes d'un texte. Ces structures sont conçues pour donner un accès rapide aux facteurs du texte. Elles permettent de travailler sur les facteurs du mot un peu comme le fait la table des suffixes du chapitre 4, mais une part plus importante est accordée à la structuration des données.

L'application principale de ces techniques est de fournir la base d'une implantation d'index qui est décrite au chapitre 6. L'accès direct aux facteurs d'un mot autorise un grand nombre d'autres applications. En particulier, les structures peuvent être utilisées pour faire de la recherche de motifs en les considérant comme des machines de recherche (voir chapitre 6).

Deux types d'objets sont considérés dans ce chapitre, les arbres et les automates, avec leurs versions compactes. Les premiers ont pour effet de mettre en commun les préfixes des mots de l'ensemble. Les seconds regroupent en plus leurs suffixes communs. Les structures sont présentées en ordre de taille décroissante.

La représentation des suffixes d'un mot par arbre (section 5.1) a l'avantage d'être simple mais peut conduire à un espace mémoire quadratique en la longueur du mot considéré. L'arbre compact des suffixes (section 5.2) est assuré, lui, de tenir dans un espace mémoire linéaire.

La minimisation (au sens des automates) de l'arbre de suffixes donne l'automate minimal des suffixes décrit dans la section 5.4. Compaction et minimisation donnent l'automate compact des suffixes de la section 5.5.

La plupart des algorithmes de construction des structures présentées dans le chapitre fonctionnent en temps $O(n \times \log \operatorname{card} A)$ sur un texte de longueur n en supposant que l'alphabet est muni d'une relation d'ordre. Leur temps d'exécution est donc linéaire lorsque l'alphabet est fini et fixé.

5.1 Arbre des suffixes

L'***arbre des suffixes*** d'un mot est l'automate déterministe qui reconnait l'ensemble des suffixes du mot et dans lequel deux chemins différents de même origine ont toujours des fins distinctes. Ainsi, la structure de graphe sous-jacente à l'automate est-elle un arbre dont les arcs sont étiquetés par des lettres. Les méthodes de la section 1.4 peuvent être utilisées pour l'implantation de ces automates. Cependant, la structure d'arbre en autorise une représentation simplifiée.

Le fait de considérer un arbre implique que les états terminaux de l'arbre sont en bijection avec les mots du langage reconnu. L'arbre n'est donc fini que si son langage l'est aussi. En conséquence, la représentation explicite de l'arbre n'a d'intérêt algorithmique que pour les langages finis.

On impose quelquefois aux arbres de n'avoir pour états terminaux que des nœuds externes de l'arbre. Avec cette contrainte, un langage L est représentable par un arbre seulement si aucun préfixe propre d'un mot de L n'est dans L. Il résulte de cette remarque que si y est un mot non vide, seul $\mathit{Suff}(y)\backslash\{\varepsilon\}$ est représentable par un arbre possédant cette propriété, et cela a lieu uniquement lorsque la dernière lettre de y n'apparait qu'une seule fois dans y. C'est pour cette raison que l'on ajoute parfois une lettre spéciale à la fin du mot. Nous préférons attribuer une sortie aux nœuds de l'arbre, ce qui est plus conforme à la notion utilisée, celle d'automate. Seuls les nœuds dont la sortie est définie sont considérés comme terminaux. Par ailleurs, il n'y a pas beaucoup de différences entre les implantations des deux procédés.

L'arbre des suffixes d'un mot y est noté $\mathcal{A}(\mathit{Suff}(y))$. Ses nœuds sont les facteurs de y, ε en est l'état initial, et les suffixes de y sont les états terminaux. La fonction de transition δ de $\mathcal{A}(\mathit{Suff}(y))$ est définie par $\delta(u, a) = ua$ si ua est un facteur de y et $a \in A$. La sortie d'un état terminal, qui est alors un suffixe, est la position de ce suffixe dans y. Un exemple d'automate est présenté dans la figure 5.1.

La construction de $\mathcal{A}(\mathit{Suff}(y))$ est généralement effectuée par ajouts successifs des suffixes de y dans l'arbre en construction, en partant du plus long suffixe, y lui-même, jusqu'au plus court, le mot vide.

La situation courante consiste en l'insertion du suffixe de position i, $y[i\,..\,n-1]$, dans la structure qui contient déjà tous les suffixes plus longs. On appelle ***tête*** du suffixe son plus long préfixe commun à un suffixe de position (strictement) inférieure. C'est aussi le plus long préfixe de $y[i\,..\,n-1]$ qui est l'étiquette d'un chemin issu de l'état initial dans l'automate en construction. L'état fin du chemin est appelé une fourche (deux chemins divergent à partir de cet état). Si $y[i\,..\,k-1]$ est la tête du suffixe de position i, le mot $y[k\,..\,n-1]$ est appelé la ***queue*** du suffixe. La figure 5.2 illustre ces notions.

Plus précisément, on appelle ***fourche*** de l'automate tout état qui est de degré (sortant) au moins 2, ou qui est à la fois de degré 1 et terminal.

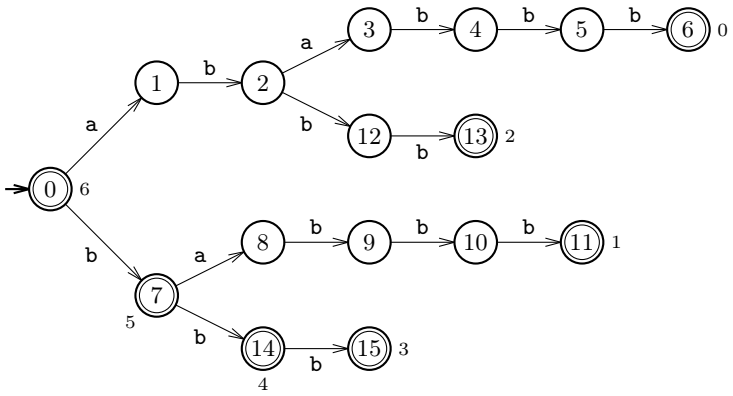


Figure 5.1 Arbre des suffixes du mot `ababbb`, $\mathcal{A}(Suff(\texttt{ababbb}))$. À chacun des états terminaux – indiqués par des doubles cercles – est associée une sortie qui est la position du suffixe correspondant à l'état dans le mot `ababbb`.

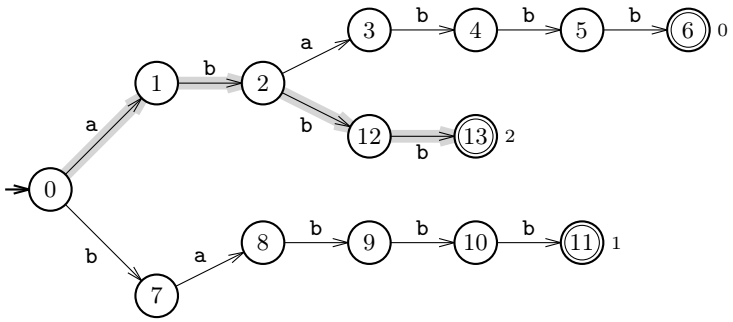


Figure 5.2 L'automate $\mathcal{A}(Suff(\texttt{ababbb}))$ (voir figure 5.1) en cours de construction, juste après l'insertion du suffixe `abbb`. La fourche, état 2, correspond à la tête de ce suffixe, `ab`, qui est le plus long préfixe de `abbb` apparaissant avant la position concernée. La queue du suffixe est `bb`, étiquette du chemin greffé à cette étape à partir de la fourche.

L'algorithme ARBRE-SUFFIXES construit l'arbre des suffixes de y. Son code est donné ci-dessous. On suppose que l'automate est représenté au moyen d'ensembles de successeurs étiquetés (voir section 1.4). Les états de l'automate possèdent l'attribut *sortie* dont la valeur, lorsqu'elle est définie, est une position sur le mot y. À la création des états par la fonction NOUVEL-ÉTAT(), la valeur de l'attribut est indéfinie. Seule la sortie des états terminaux est définie par l'algorithme. L'insertion du suffixe courant $y[i\,..\,n-1]$ dans l'automate en construction, M, commence par la détermination de sa tête, $y[i\,..\,k-1]$, et de la fourche associée, $p = \delta(\mathit{initial}[M], y[i\,..\,k-1])$, à partir de laquelle on doit greffer la queue du suffixe (où δ est la fonction de transition de M). La valeur de la fonction DESC-LENTE appliquée au couple $(\mathit{initial}[M], i)$ est précisément le couple (p, k) cherché. La création du chemin d'origine p et d'étiquette $y[k\,..\,n-1]$ avec la définition de la sortie de sa fin est réalisée aux lignes 5–9 du code.

La fin de l'exécution de l'algorithme, insertion du suffixe vide, consiste juste à définir la sortie de l'état initial, dont la valeur est $n = |y|$ par convention (ligne 10).

```
ARBRE-SUFFIXES(y, n)
 1  M ← NOUVEL-AUTOMATE()
 2  pour i ← 0 à n − 1 faire
 3        (fourche, k) ← DESC-LENTE(initial[M], i)
 4        p ← fourche
 5        pour j ← k à n − 1 faire
 6              q ← NOUVEL-ÉTAT()
 7              Succ[p] ← Succ[p] ∪ {(y[j], q)}
 8              p ← q
 9        sortie[p] ← i
10  sortie[initial[M]] ← n
11  retourner M
```

```
DESC-LENTE(p, k)
1  tantque k < n et CIBLE(p, y[k]) ≠ NIL faire
2        (p, k) ← (CIBLE(p, y[k]), k + 1)
3  retourner (p, k)
```

Proposition 5.1
L'algorithme ARBRE-SUFFIXES *construit l'arbre des suffixes d'un mot de longueur n en temps $\Omega(n^2)$.*

Preuve La preuve de bon fonctionnement de l'algorithme se vérifie simplement sur le code de l'algorithme.

Pour l'évaluation du temps d'exécution, considérons l'étape i. Supposons que $y[i\,..\,n-1]$ ait pour tête $y[i\,..\,k-1]$ et pour queue $y[k\,..\,n-1]$. On vérifie que l'appel à DESC-LENTE (ligne 3) exécute $k - i$ opérations

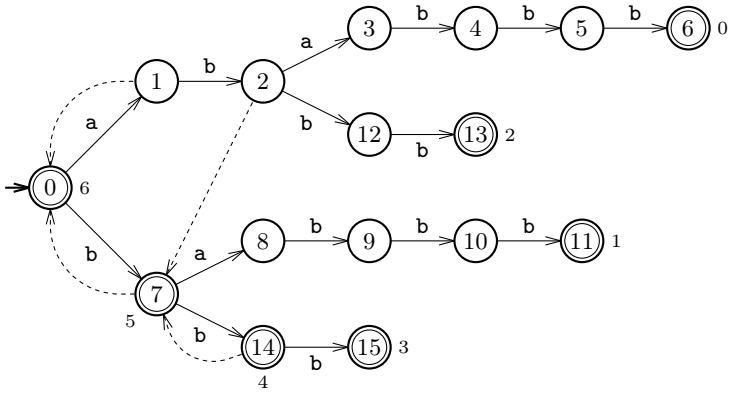


Figure 5.3 L'automate $\mathcal{A}(\mathit{Suff}(\texttt{ababbb}))$ avec les liens suffixes des fourches et de leurs ancêtres indiqués par des flèches dessinées en pointillé.

et que la boucle **pour** des lignes 5–8 en exécute $n - k$, soit un total de $n - i$ opérations. Ainsi la boucle **pour** des lignes 2–9 indexée par i exécute-t-elle $n + (n-1) + \cdots + 1$ opérations, ce qui donne un temps d'exécution total $\Omega(n^2)$. ∎

Liens suffixes

Il est possible d'accélérer la construction précédente en améliorant la recherche des fourches. La technique décrite ici est reprise dans la section suivante où elle conduit à un gain dans le temps d'exécution, mesurable par la majoration asymptotique.

Soit av un suffixe de y qui a une tête non vide az avec $a \in A$. Le préfixe z de v apparait donc dans y avant l'occurrence considérée. Cela implique que z est un préfixe de la tête du suffixe v. La recherche de cette tête, et de la fourche correspondante, peut donc se faire à partir de l'état z au lieu de commencer systématiquement par l'état initial comme dans l'algorithme précédent. Néanmoins, cela suppose que, l'état az étant connu, on ait un accès rapide à l'état z. Pour cela on introduit une fonction sur les états de l'automate, appelée ***lien suffixe***. Elle est notée s et définie par $s(az) = z$ pour chaque état az ($a \in A$, $z \in A^*$). L'état $s(az)$ est appelé la ***cible suffixe*** de az. La figure 5.3 montre le lien suffixe de l'arbre de la figure 5.1.

L'algorithme dont le code suit implante et utilise le lien suffixe pour le calcul de l'arbre des suffixes de y. Le lien est réalisé au moyen d'un attribut attaché à chacun des états et noté *ls* ; l'attribut est supposé être initialisé à la valeur NIL. Les cibles suffixes ne sont effectivement calculées par l'algorithme que pour les fourches et leurs ancêtres (en dehors de l'état initial) car les cibles des autres nœuds ne sont pas utiles pour la construction. Le code n'est qu'une adaptation de l'algorithme

DESC-LENTE pour que celui-ci intègre la définition des cibles suffixes. Le nouvel algorithme est appelé DESC-LENTE-BIS.

```
DESC-LENTE-BIS(p, k)
 1  tantque k < n et CIBLE(p, y[k]) ≠ NIL faire
 2        q ← CIBLE(p, y[k])
 3        (e, f) ← (p, q)
 4        tantque e ≠ initial[M] et ls[f] = NIL faire
 5              ls[f] ← CIBLE(ls[e], y[k])
 6              (e, f) ← (ls[e], ls[f])
 7        si ls[f] = NIL alors
 8              ls[f] ← initial[M]
 9        (p, k) ← (q, k + 1)
10  retourner (p, k)
```

```
ARBRE-SUFFIXES-BIS(y, n)
 1  M ← NOUVEL-AUTOMATE()
 2  ls[initial[M]] ← initial[M]
 3  (fourche, k) ← (initial[M], 0)
 4  pour i ← 0 à n − 1 faire
 5        k ← max{k, i}
 6        (fourche, k) ← DESC-LENTE-BIS(ls[fourche], k)
 7        p ← fourche
 8        pour j ← k à n − 1 faire
 9              q ← NOUVEL-ÉTAT()
10              Succ[p] ← Succ[p] ∪ {(y[j], q)}
11              p ← q
12        sortie[p] ← i
13  sortie[initial[M]] ← n
14  retourner M
```

Proposition 5.2
L'algorithme ARBRE-SUFFIXES-BIS *construit l'arbre des suffixes de y en temps $\Omega(\operatorname{card} Q)$, où Q est l'ensemble des états de $\mathcal{A}(\mathit{Suff}(y))$.*

Preuve Les opérations de la boucle principale, en dehors de la ligne 6 et de la boucle **pour** des lignes 8–11, s'exécutent en temps constant, ce qui donne un temps $O(|y|)$ pour leur exécution globale.

Chaque opération de la boucle interne à l'algorithme DESC-LENTE-BIS qui est appelé à la ligne 6 a pour effet de créer une cible suffixe. Le nombre total de cibles étant majoré par $\operatorname{card} Q$, le temps cumulé de toutes les exécutions de la ligne 6 est $O(\operatorname{card} Q)$.

Le temps d'exécution de la boucle 8–11 est proportionnel au nombre d'états qu'elle crée. Le temps cumulé de toutes les exécutions des lignes 8–11 est donc encore $O(\operatorname{card} Q)$.

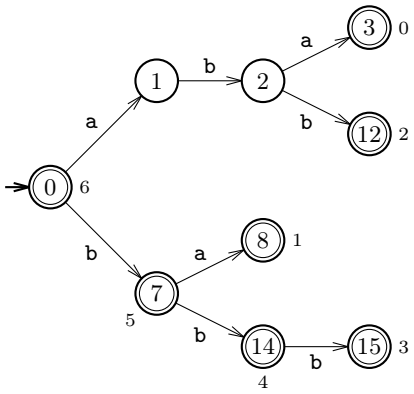


Figure 5.4 Arbre des positions du mot `ababbb`. Il reconnait les plus courts facteurs ou suffixes qui identifient les positions du mot.

Enfin, comme $|y| < \operatorname{card} Q$ et qu'il y a création de $\operatorname{card} Q$ états, le temps total de construction est $\Omega(\operatorname{card} Q)$, comme annoncé. ∎

La taille de $\mathcal{A}(\mathit{Suff}(y))$ peut être quadratique. C'est par exemple le cas pour un mot dont les lettres sont deux à deux distinctes. Pour cette catégorie de mots l'algorithme Arbre-suffixes-bis n'est en fait pas plus rapide que Arbre-suffixes.

Pour certains mots, il suffit d'élaguer les branches pendantes (au-dessous des fourches) de $\mathcal{A}(\mathit{Suff}(y))$ pour obtenir une structure dont la taille est linéaire. Ce genre d'élagage donne l'arbre des positions de y (un exemple est montré figure 5.4) qui représente les plus courts facteurs n'apparaissant qu'à une seule position dans y ainsi que les suffixes qui identifient les autres positions. Toutefois, la considération de l'arbre des positions ne résout pas totalement les questions d'espace mémoire car cette structure peut également avoir une taille quadratique. On constate par exemple que le mot $\mathtt{a}^k\mathtt{b}^k\mathtt{a}^k\mathtt{b}^k$ ($k \in \mathbf{N}$) de longueur $4k$ possède un arbre des suffixes élagué qui contient plus de k^2 nœuds.

La structure d'arbre compact de la section suivante est une solution pour obtenir une structure de taille linéaire. Les automates des sections 5.4 et 5.5 fournissent un autre type de solution.

5.2 Arbre compact des suffixes

L'***arbre compact des suffixes*** de y, noté $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$, est obtenu en supprimant les nœuds de degré 1 qui ne sont pas terminaux dans son arbre des suffixes. C'est ce que l'on appelle la ***compaction*** de l'arbre. L'arbre compact ne conserve que les fourches et les nœuds externes de l'arbre des suffixes. Les étiquettes des flèches deviennent alors des mots

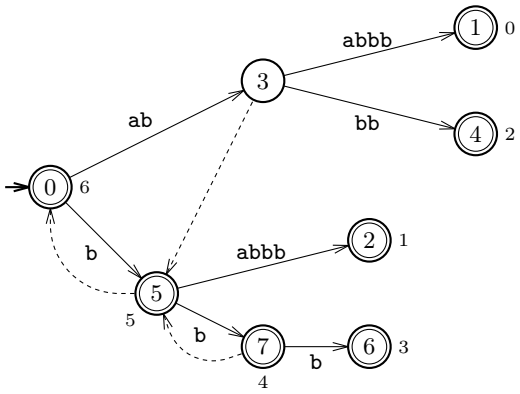


Figure 5.5 L'arbre compact des suffixes $\mathcal{A}_C(Suff(\texttt{ababbb}))$ avec ses liens suffixes.

de longueur variable (strictement positive). On remarque que si deux arcs issus d'un même nœud sont étiquetés par les mots u et v, alors leurs premières lettres sont distinctes, c'est-à-dire que l'on a $u[0] \neq v[0]$. Cela vient de ce que l'arbre des suffixes est un automate déterministe.

La figure 5.5 montre l'arbre compact des suffixes obtenu par compaction de l'arbre des suffixes de la figure 5.1. La figure 5.6 présente un arbre compact des suffixes adapté au cas où le mot se termine par une lettre spéciale.

Proposition 5.3
L'arbre compact des suffixes d'un mot de longueur $n > 0$ possède entre $n + 1$ et $2n$ nœuds. Le nombre de fourches du même type d'arbre est compris entre 1 et n.

Preuve L'arbre contient $n + 1$ nœuds terminaux distincts correspondant aux $n + 1$ suffixes qu'il représente. Cela donne la minoration.

Chaque fourche de l'arbre qui n'est pas terminale possède au moins deux enfants. Pour un nombre fixé de nœuds externes, le nombre maximal de ces fourches est obtenu lorsque chacun de ces nœuds possède exactement deux enfants. Dans ce cas, on obtient au plus n fourches (terminales ou non). Comme pour $n > 0$ l'état initial est à la fois une fourche et un nœud terminal, on obtient la majoration $(n+1)+n-1 = 2n$ du nombre total de nœuds. ∎

Le fait d'avoir un nombre linéaire de nœuds n'implique pas la linéarité de la représentation de l'arbre compact des suffixes de y, car celle-ci dépend aussi de la taille totale des étiquettes des arcs. L'exemple d'un mot de longueur n qui possède n lettres distinctes montre que cette taille peut fort bien être quadratique. Néanmoins, les étiquettes des arcs étant toutes des facteurs de y, chacune peut être représentée par un

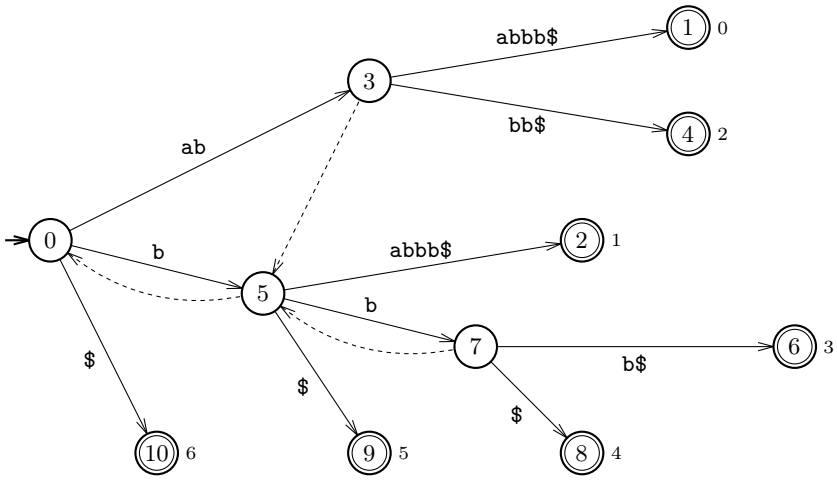


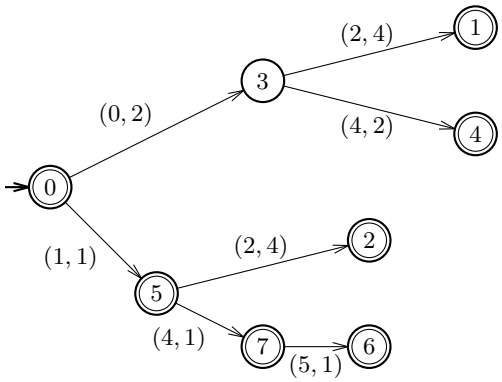
Figure 5.6 Adaptation de l'arbre compact des suffixes pour un mot marqué à droite par une lettre spéciale. Seuls les nœuds externes sont terminaux. Ils correspondent aux suffixes du mot `ababbb`.

couple position-longueur (ou aussi position de début-position de fin), à condition que le mot y réside en mémoire avec l'arbre pour permettre un accès aux étiquettes. Si le mot u est l'étiquette d'un arc (p, q), il sera représenté par le couple $(i, |u|)$ où i est la position d'une occurrence de u dans y. On note $\mathit{étiq}(p, q) = (i, |u|)$ et on suppose que l'implantation de l'arbre procure un accès direct à cette étiquette. Cette représentation des étiquettes est illustrée dans la figure 5.7 pour l'arbre de la figure 5.5.

Proposition 5.4
En représentant les étiquettes des flèches par des couples d'entiers, la taille totale de l'arbre compact des suffixes d'un mot est linéaire en sa longueur, c'est-à-dire que la taille de $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ *est* $O(|y|)$.

Preuve Le nombre de nœuds de $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ est $O(|y|)$ d'après la proposition 5.3. Le nombre d'arcs de $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ est une unité de moins que le nombre de nœuds. L'hypothèse sur la représentation des flèches a pour conséquence que chaque arc occupe un espace constant, ce qui donne le résultat. ∎

Le lien suffixe introduit dans la section précédente trouve sa pleine utilité dans la construction de l'arbre compact des suffixes. Il permet une construction rapide quand, en plus, l'algorithme de descente lente de la section précédente est remplacé par l'algorithme de descente rapide ci-après qui a une fonction analogue. La possibilité de ne retenir que les fourches de l'arbre des suffixes, en plus des états terminaux repose sur le lemme suivant.



i	0	1	2	3	4	5
$y[i]$	a	b	a	b	b	b

Figure 5.7 Représentation des étiquettes dans l'arbre compact des suffixes $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ avec $y =$ `ababbb` (à comparer à la figure 5.5). L'étiquette $(2, 4)$ de l'arc $(3, 1)$ représente le facteur de longueur 4 et de position 2 sur y, c'est-à-dire le mot `abbb`.

Proposition 5.5
Dans l'arbre des suffixes d'un mot, la cible suffixe d'une fourche (non vide) est une fourche.

Preuve Pour une fourche non vide, il y a deux cas à considérer suivant que la fourche, disons au ($a \in A$, $u \in A^*$) est de degré au moins 2, ou simultanément de degré 1 et terminale.

Supposons que le degré de au soit au moins 2. Pour deux lettres distinctes, b et c, aub et auc sont des facteurs de y. La même propriété vaut donc pour $u = s(au)$ qui est ainsi de degré au moins 2 et est donc une fourche.

Si la fourche au est de degré 1 et terminale, pour une lettre b, le mot aub est un facteur de y et simultanément au est un suffixe de y. Donc, ub est un facteur de y et u est un suffixe de y, ce qui montre que $u = s(au)$ est encore une fourche. ∎

La propriété suivante sert de base au calcul des cibles suffixes dans l'algorithme de construction de l'arbre des suffixes Arbre-c-suffixes. On note δ la fonction de transition de $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$.

Lemme 5.6
Soient (p, q) un arc de $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ et $y[j\,..\,k-1]$, $j < k$, son étiquette. Si q est une fourche de l'arbre, on a :

$$s(q) = \begin{cases} \delta(p, y[j+1\,..\,k-1]) & \textit{si } p \textit{ est l'état initial ,} \\ \delta(s(p), y[j\,..\,k-1]) & \textit{sinon .} \end{cases}$$

Preuve Comme q est une fourche, $s(q)$ est défini d'après la proposition 5.5. Si p est l'état initial de l'arbre, c'est-à-dire si $p = \varepsilon$, on a $s(q) = \delta(\varepsilon, y[j+1\,..\,k-1])$ par définition de s.

Dans le cas contraire, il existe un chemin unique d'origine l'état initial et de fin p car $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ est un arbre. Soit av l'étiquette non vide de ce chemin avec $a \in A$ et $v \in A^*$ (c'est-à-dire que l'on a $p = av$). On a donc $\delta(\varepsilon, v) = s(p)$ et $\delta(\varepsilon, v \cdot y[j\,..\,k-1]) = s(q)$. Il s'ensuit que $s(q) = \delta(s(p), y[j\,..\,k-1])$ car l'automate est déterministe, comme annoncé. ∎

La stratégie retenue pour construire l'arbre compact des suffixes de y consiste à insérer successivement les suffixes de y dans la structure, du plus long au plus court, comme dans la construction de l'arbre (non compact) des suffixes de la section précédente. Comme pour l'algorithme ARBRE-SUFFIXES-BIS, l'insertion de la queue du suffixe courant se fait après descente lente à partir de la cible suffixe de la fourche courante.

```
ARBRE-C-SUFFIXES(y, n)
 1  M ← NOUVEL-AUTOMATE()
 2  ls[initial[M]] ← initial[M]
 3  (fourche, k) ← (initial[M], 0)
 4  pour i ← 0 à n − 1 faire
 5        k ← max{k, i}
 6        si ls[fourche] = NIL alors
 7              t ← parent de fourche
 8              (j, ℓ) ← étiq(t, fourche)
 9              si t = initial[M] alors
10                    ℓ ← ℓ − 1
11              ls[fourche] ← DESC-RAPIDE(ls[t], k − ℓ, k)
12        (fourche, k) ← DESC-LENTE-C(ls[fourche], k)
13        si k < n alors
14              q ← NOUVEL-ÉTAT()
15              Succ[fourche] ← Succ[fourche] ∪ {((k, n − k), q)}
16        sinon q ← fourche
17        sortie[q] ← i
18  sortie[initial[M]] ← n
19  retourner M
```

Lorsque ce lien n'existe pas, il est créé (lignes 6–11) en utilisant la propriété de l'énoncé précédent. Le calcul est réalisé par l'algorithme DESC-RAPIDE, dit de descente rapide, qui satisfait :

$\textsc{Desc-rapide}(r, j, k) = \delta(r, y[j\,..\,k-1])$

pour r état de l'arbre et j, k positions sur y qui sont tels que :

$r \cdot y[j\,..\,k-1] \preceq_{\mathit{fact}} y$.

À la ligne 7, l'accès au parent de *fourche* doit être compris comme une explicitation de la valeur de t. Celle-ci peut être récupérée au moyen d'un chainage vers le parent. On préfèrera une mémorisation permanente du

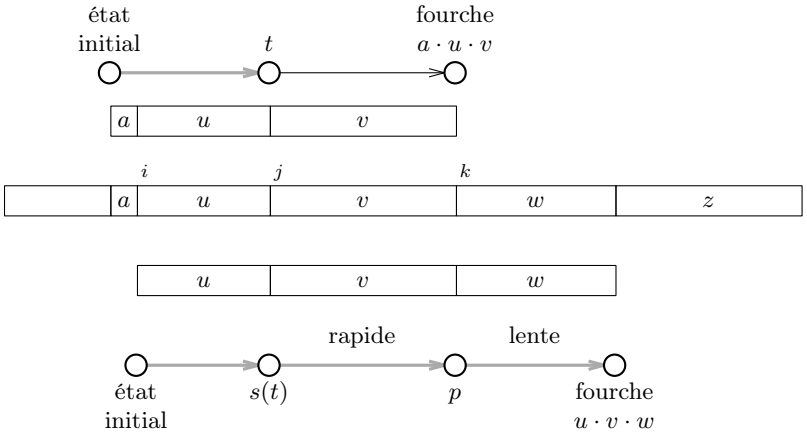


Figure 5.8 Schéma pour l'insertion du suffixe $y[i\,..\,n-1] = u \cdot v \cdot w \cdot z$ de y dans l'arbre compact des suffixes en cours de construction quand le lien suffixe n'est pas défini sur la fourche $a \cdot u \cdot v$. Soit t le parent de cette fourche et v l'étiquette de la flèche associée. On calcule d'abord $p = \delta(s(t), v)$ par descente rapide, puis la fourche du suffixe par descente lente comme dans la section 5.1.

parent de la fourche (cela peut conduire à considérer un nœud artificiel, parent de l'état initial). Le schéma pour l'insertion d'un suffixe à l'intérieur de l'arbre en construction est présenté dans la figure 5.8.

L'algorithme de descente lente voit son code adapté par rapport à l'algorithme DESC-LENTE pour prendre en compte le fait que les étiquettes des flèches sont des mots. Quand la cible cherchée tombe au milieu d'une flèche, il faut aussi couper cette flèche.

```
DESC-LENTE-C(p, k)
 1  tantque k < n et CIBLE(p, y[k]) ≠ NIL faire
 2        q ← CIBLE(p, y[k])
 3        (j, ℓ) ← étiq(p, q)
 4        i ← j
 5        faire  i ← i + 1
 6               k ← k + 1
 7        tantque i < j + ℓ et k < n et y[i] = y[k]
 8        si i < j + ℓ alors
 9               Succ[p] ← Succ[p] \ {((j, ℓ), q)}
10               r ← NOUVEL-ÉTAT()
11               Succ[p] ← Succ[p] ∪ {((j, i − j), r)}
12               Succ[r] ← Succ[r] ∪ {((i, ℓ − i + j), q)}
13               retourner (r, k)
14        p ← q
15  retourner (p, k)
```

Remarquons que CIBLE(p, a), s'il existe, est l'état q pour lequel a est la

première lettre de l'étiquette de la flèche de source p et de cible q. Les étiquettes peuvent être des mots de longueur strictement supérieure à 1 ; on n'a donc pas en général $\textsc{Cible}(p, a) = \delta(p, a)$.

L'amélioration du temps d'exécution du calcul d'un arbre des suffixes par l'algorithme Arbre-c-suffixes repose, en plus de la compaction de la structure de données, sur un élément algorithmique supplémentaire : l'implantation de Desc-rapide. Le recours à cet algorithme particulier décrit par le code ci-dessous est essentiel pour obtenir le temps d'exécution de l'algorithme de construction de l'arbre énoncé par le théorème 5.9.

L'algorithme Desc-rapide est utilisé pour calculer une fourche. Il n'est appliqué à l'état r et au mot $y[j\,..\,k-1]$ que lorsque la condition

$$r \cdot y[j\,..\,k-1] \prec_{fact} y$$

est satisfaite. Dans cette situation, il existe un chemin d'origine l'état r et dont l'étiquette a pour préfixe $y[j\,..\,k-1]$. De plus, comme l'automate est déterministe, le plus court de ces chemins est unique. L'algorithme utilise cette propriété pour déterminer les flèches du chemin par seul examen de la première lettre de leur étiquette. Le code ci-dessous, ou du moins la partie principale, implante la relation de récurrence donnée dans la preuve du lemme 5.7.

L'algorithme Desc-rapide sert plus précisément à l'évaluation de $\delta(r, y[j \cdot k-1])$ (ou à celle de $\delta(r, v)$ pour reprendre les notations du lemme 5.7). Quand la fin du chemin parcouru n'est pas l'état cherché, il y a création d'un état p qui s'intercale entre les deux derniers états rencontrés.

```
Desc-rapide(r, j, k)
 1  ▷ Calcul de δ(r, y[j..k−1])
 2  si j ≥ k alors
 3        retourner r
 4  sinon q ← Cible(r, y[j])
 5        (j′, ℓ) ← étiq(r, q)
 6        si j + ℓ ≤ k alors
 7              retourner Desc-rapide(q, j + ℓ, k)
 8        sinon Succ[r] ← Succ[r] \ {((j′, ℓ), q)}
 9              p ← Nouvel-état()
10              Succ[r] ← Succ[r] ∪ {((j′, k − j), p)}
11              Succ[p] ← Succ[p] ∪ {((j′ + k − j, ℓ − k + j), q)}
12              retourner p
```

Le fonctionnement des algorithmes de descente lente et de descente rapide est illustré par la figure 5.9.

Le lemme qui suit sert à l'évaluation du temps d'exécution de Desc-rapide(r, j, k). Il est un élément de la preuve du théorème 5.9. Il indique que le temps de calcul est proportionnel (à un coefficient multiplicatif près qui provient du temps de calcul des transitions) au nombre de nœuds

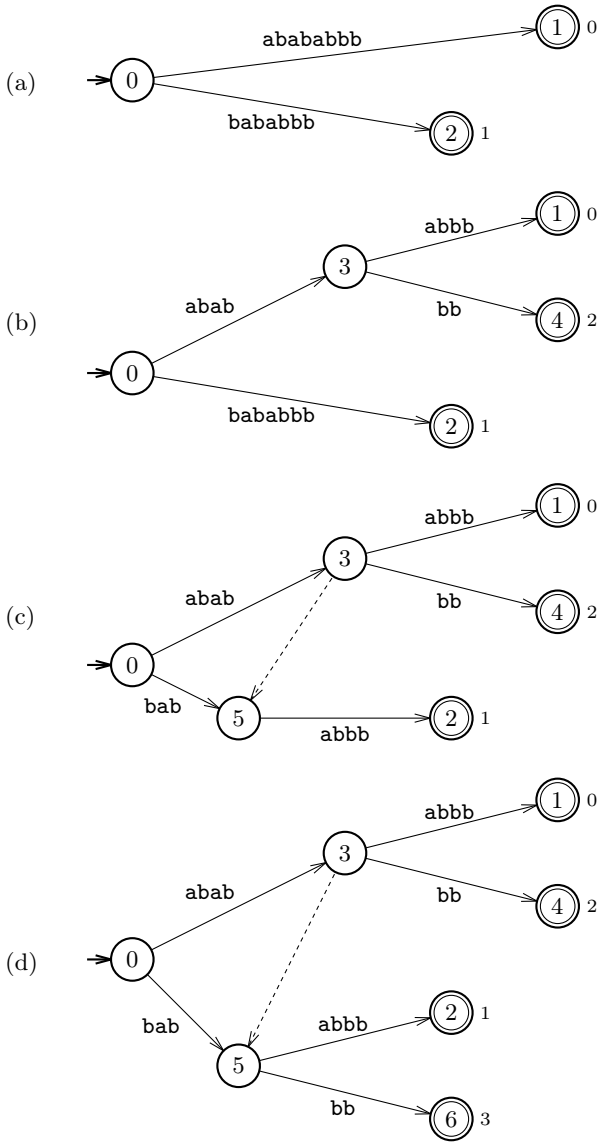


Figure 5.9 Durant la construction de $\mathcal{A}_{\mathrm{C}}$(`abababbb`), insertion des suffixes `ababbb` et `babbb`. **(a)** Automate obtenu après l'insertion des suffixes `abababbb` et `bababbb`. La fourche courante est l'état initial 0. **(b)** On ajoute le suffixe `ababbb` par comparaisons lettre à lettre (descente lente) à partir de l'état 0. Cela conduit à créer la fourche 3. Le lien suffixe de 3 n'est pas encore défini. **(c)** La première étape de l'insertion du suffixe `babbb` commence par la définition du lien suffixe de l'état 3 qui est l'état 5. On procède par descente rapide depuis l'état 0 par le mot `bab`. **(d)** La seconde étape de l'insertion de `babbb` conduit à la création de l'état 6. L'état 5, qui est la fourche du suffixe `babbb`, devient la fourche courante pour la suite de la construction.

du chemin parcouru et non à la longueur de l'étiquette du chemin, résultat que l'on obtiendrait de façon immédiate en appliquant l'algorithme DESC-LENTE (section 5.1).

Pour un état r de $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ et un mot $v \preceq_{\mathit{fact}} y$ satisfaisant l'inégalité $r \cdot v \preceq_{\mathit{fact}} y$, on note $\mathit{but}(r, v)$ la fin du plus court chemin d'origine r dont l'étiquette a pour préfixe v. On note que $\mathit{but}(r, v) = \delta(r, v)$ uniquement si v est l'étiquette du chemin.

Lemme 5.7
Soient r un nœud de $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ et v un mot tels que $r \cdot v \preceq_{\mathit{fact}} y$. Soit $\langle r, r_1, \ldots, r_\ell \rangle$ le chemin d'origine r et de fin $r_\ell = \mathit{but}(r, v)$ dans $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$. Le calcul de $\mathit{but}(r, v)$ peut être réalisé en temps $O(\ell \times \log \operatorname{card} A)$ dans le modèle comparaisons.

Preuve On remarque que le chemin $\langle r, r_1, \ldots, r_\ell \rangle$ existe par la condition $r \cdot v \preceq_{\mathit{fact}} y$ et est unique car l'arbre est un automate déterministe.

Si $v = \varepsilon$, on a $\mathit{but}(r, v) = r$. Sinon, soient $r_1 = \textsc{Cible}(r, v[0])$ et v' l'étiquette de l'arc (r, r_1). On constate que :

$$\mathit{but}(r, v) = \begin{cases} r_1 & \text{si } |v| \leq |v'| \text{ (soit } v \preceq_{\mathit{préf}} v') \ , \\ \mathit{but}(r_1, v'^{-1}v) & \text{sinon } . \end{cases}$$

Cette relation montre que chaque étape du calcul prend un temps $\alpha + \beta$ où α est une constante, qui comprend le temps d'accès à l'étiquette de l'arc (r, r_1), et β est le temps de calcul de $\textsc{Cible}(r, v[0])$. Cela donne le temps $O(\log \operatorname{card} A)$ dans le modèle comparaisons.

Le calcul de r_ℓ qui comprend le parcours du chemin $\langle r, r_1, \ldots, r_\ell \rangle$ prend donc un temps $O(\ell \times \log \operatorname{card} A)$ comme annoncé. ∎

Corollaire 5.8
Soient r un nœud de $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ et j, k deux positions sur y, $j < k$, tels que $r \cdot y[j \,..\, k-1] \preceq_{\mathit{fact}} y$. Soit ℓ le nombre d'états de l'arbre visités pendant le calcul de $\textsc{Desc-rapide}(r, j, k)$. Le temps d'exécution de $\textsc{Desc-rapide}(r, j, k)$ est alors $O(\ell \times \log \operatorname{card} A)$ dans le modèle comparaisons.

Preuve Posons $v = y[j \,..\, k-1]$ et notons $\langle r, r_1, \ldots, r_\ell \rangle$ le chemin dont la fin est $\mathit{but}(r, v)$. Le calcul de $\mathit{but}(r, v)$ est effectué par DESC-RAPIDE qui implante la relation de récurrence de la preuve du lemme 5.7. Il prend donc un temps $O(\ell \times \log \operatorname{card} A)$. Au dernier appel récursif, il y a éventuellement création de l'état p et modification des arcs. Cette opération prend encore le temps $O(\log \operatorname{card} A)$, ce qui donne le temps global $O(\ell \times \log \operatorname{card} A)$ de l'énoncé. ∎

Théorème 5.9
L'opération $\textsc{Arbre-c-suffixes}(y, n)$, qui produit $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$, prend un temps $O(n \times \log \operatorname{card} A)$ dans le modèle comparaisons.

Preuve Le fait que l'opération ARBRE-C-SUFFIXES(y, n) produise l'automate $\mathcal{A}_C(\mathit{Suff}(y))$ s'appuie principalement sur le lemme 5.6 en vérifiant que l'algorithme reprend la technique élémentaire de la section 5.1.

L'évaluation du temps d'exécution repose sur les observations suivantes (voir figure 5.8) :

- chaque étape du calcul effectué par DESC-RAPIDE, sauf peut-être la dernière, conduit au parcours d'un état et augmente strictement la valeur de $k - \ell$ (j sur la figure) ;
- chaque étape du calcul effectué par DESC-LENTE, sauf peut-être la dernière, augmente strictement la valeur de k ;
- chaque autre instruction de la boucle **pour** conduit à une incrémentation de i.

Le nombre d'étapes exécutées par DESC-RAPIDE est donc majoré par n, ce qui donne un temps $O(n \times \log \operatorname{card} A)$ pour ces étapes d'après le corollaire 5.8. Le même raisonnement vaut pour le nombre d'étapes exécutées par DESC-LENTE et pour les autres étapes, donnant encore un temps $O(n \times \log \operatorname{card} A)$.

On obtient ainsi un temps d'exécution total $O(n \times \log \operatorname{card} A)$. ∎

5.3 Contextes des facteurs

On présente dans cette section les bases formelles de la construction de l'automate minimal qui accepte les suffixes d'un mot. Certaines propriétés concourent à la preuve de la construction de l'automate (théorèmes 5.19 et 5.28 plus loin).

L'automate (minimal) des suffixes d'un mot y est noté $\mathcal{S}(y)$. Ses états sont les classes de l'équivalence (ou congruence) syntaxique associée à l'ensemble $\mathit{Suff}(y)$, c'est-à-dire des ensembles de facteurs de y possédant le même contexte droit au sein de y (voir section 1.1). Ces états sont en bijection avec les contextes (droits) des facteurs de y dans y lui-même. Rappelons que le contexte (droit) d'un mot u relativement à y est $u^{-1}\mathit{Suff}(y)$. On note $\equiv_{\mathit{Suff}(y)}$ la congruence syntaxique qui est définie, pour $u, v \in A^*$, par :

$u \equiv_{\mathit{Suff}(y)} v$

si et seulement si

$u^{-1}\mathit{Suff}(y) = v^{-1}\mathit{Suff}(y)$.

On peut également identifier les états de $\mathcal{S}(y)$ à des ensembles d'indices sur y qui sont les positions droites d'occurrences de facteurs équivalents.

Les contextes droits satisfont quelques propriétés énoncées ci-dessous et qui sont utilisées dans la suite. La première remarque concerne le lien

entre la relation $\preceq_{suff}$ et l'inclusion des contextes. Pour tout facteur u de y, on note :

$posd(u) = \min\{|w| - 1 : w \preceq_{préf} y \text{ et } u \preceq_{suff} w\}$

la position droite de la première occurrence de u dans y.

Lemme 5.10
Soient $u, v \preceq_{fact} y$ avec $|u| \leq |v|$. Alors :

$u \preceq_{suff} v$ *implique* $v^{-1}Suff(y) \subseteq u^{-1}Suff(y)$

et :

$v^{-1}Suff(y) = u^{-1}Suff(y)$ *implique* $posd(v) = posd(u)$ *et* $u \preceq_{suff} v$.

Preuve Supposons $u \preceq_{suff} v$. Soit $z \in v^{-1}Suff(y)$. Par définition, $vz \preceq_{suff} y$ et, comme $u \preceq_{suff} v$, on a également $uz \preceq_{suff} y$. Donc, $z \in u^{-1}Suff(y)$, ce qui prouve la première implication.

Supposons maintenant $v^{-1}Suff(y) = u^{-1}Suff(y)$. Soient w, z tels que $y = w \cdot z$ avec $|w| = posd(u) + 1$. Par définition de $posd$, u est un suffixe de w. Donc z est le plus long mot de $u^{-1}Suff(y)$. L'hypothèse implique que z est aussi le plus long mot de $v^{-1}Suff(y)$, ce qui entraine $|w| = posd(v) + 1$. Les mots u et v sont donc tous deux des suffixes de w, et comme u est plus court que v, on obtient $u \preceq_{suff} v$. Cela termine la preuve de la seconde implication. ∎

Une autre propriété très utile de la congruence est qu'elle partitionne les suffixes d'un facteur de y en intervalles relativement à leur longueur.

Lemme 5.11
Soient $u, v, w \preceq_{fact} y$. Alors :

$u \preceq_{suff} v$, $v \preceq_{suff} w$ *et* $u \equiv_{Suff(y)} w$

implique

$u \equiv_{Suff(y)} v \equiv_{Suff(y)} w$.

Preuve Par le lemme 5.10, l'hypothèse implique :

$w^{-1}Suff(y) \subseteq v^{-1}Suff(y) \subseteq u^{-1}Suff(y)$.

L'équivalence $u \equiv_{Suff(y)} w$, qui signifie $u^{-1}Suff(y) = w^{-1}Suff(y)$, entraine alors la conclusion. ∎

La propriété suivante a pour conséquence que l'inclusion induit sur les contextes droits une structure d'arbre. Dans cet arbre, le lien parent est constitué par l'inclusion propre des ensembles. Ce lien important pour la construction rapide de l'automate correspond à la fonction suffixe définie ensuite.

Corollaire 5.12
Soient $u, v \in A^*$. Les contextes de u et v sont comparables pour l'inclusion ou disjoints, c'est-à-dire que l'une au moins des trois conditions suivantes est satisfaite :

1. $u^{-1}\mathit{Suff}(y) \subseteq v^{-1}\mathit{Suff}(y)$;
2. $v^{-1}\mathit{Suff}(y) \subseteq u^{-1}\mathit{Suff}(y)$;
3. $u^{-1}\mathit{Suff}(y) \cap v^{-1}\mathit{Suff}(y) = \emptyset$.

Preuve On prouve la propriété en montrant que la condition

$u^{-1}\mathit{Suff}(y) \cap v^{-1}\mathit{Suff}(y) \neq \emptyset$

implique

$u^{-1}\mathit{Suff}(y) \subseteq v^{-1}\mathit{Suff}(y)$ ou $v^{-1}\mathit{Suff}(y) \subseteq u^{-1}\mathit{Suff}(y)$.

Soit $z \in u^{-1}\mathit{Suff}(y) \cap v^{-1}\mathit{Suff}(y)$. Alors les mots uz et vz sont des suffixes de y, et donc u et v sont des suffixes de yz^{-1}. En conséquence l'un des deux mots u ou v est un suffixe de l'autre. On obtient finalement la conclusion par le lemme 5.10. ∎

Fonction suffixe

Sur l'ensemble $\mathit{Fact}(y)$, on considère la fonction notée [1] s, appelée ***fonction suffixe*** relativement à y. Elle est définie, pour tout $v \in \mathit{Fact}(y) \setminus \{\varepsilon\}$, par :

$s(v)$ = le plus long mot $u \prec_{suff} v$ tel que $u \not\equiv_{\mathit{Suff}(y)} v$.

Après le lemme 5.10, on déduit la définition équivalente :

$s(v)$ = le plus long mot $u \prec_{suff} v$ tel que $v^{-1}\mathit{Suff}(y) \subset u^{-1}\mathit{Suff}(y)$.

On note que, par définition, $s(v)$ est un suffixe propre de v (c'est-à-dire, $|s(v)| < |v|$). Le lemme qui suit montre que la fonction suffixe s induit une fonction de suppléance (voir section 1.4) sur les états de $\mathcal{S}(y)$.

Lemme 5.13
Soient $u, v \in \mathit{Fact}(y) \setminus \{\varepsilon\}$. Alors :

$u \equiv_{\mathit{Suff}(y)} v$ *implique* $s(u) = s(v)$.

Preuve Par le lemme 5.10 on peut supposer sans perte de généralité que $u \preceq_{suff} v$. Ainsi, u et $s(v)$ sont-ils des suffixes de v, et donc l'un est un suffixe de l'autre. Le mot u ne peut être un suffixe de $s(v)$ car le lemme 5.11 impliquerait $s(v) \equiv_{\mathit{Suff}(y)} v$, ce qui contredit la définition de $s(v)$. En conséquence, $s(v)$ est un suffixe de u. Puisque, par définition, $s(v)$ est le plus long suffixe de v qui ne lui est pas équivalent et qu'il n'est pas équivalent à u, c'est aussi $s(u)$. Donc, $s(u) = s(v)$. ∎

1. Bien que nous utilisions la même notation, la définition de la fonction suffixe est syntaxique en ce sens qu'elle fait intervenir le langage de référence, alors que celle du lien suffixe en section 5.1 est de nature purement algorithmique.

Lemme 5.14
Soit $y \in A^+$. Le mot $s(y)$ est le plus long suffixe de y qui apparait deux fois au moins dans y lui-même.

Preuve Le contexte $y^{-1}Suff(y)$ est $\{\varepsilon\}$. Comme y et $s(y)$ ne sont pas équivalents, $s(y)^{-1}Suff(y)$ contient un mot z non vide. Alors, $s(y)z$ et $s(y)$ sont des suffixes de y, ce qui montre que $s(y)$ apparait deux fois au moins dans y.

Tout suffixe w de y, plus long que $s(y)$, est équivalent à y par définition de $s(y)$. Il satisfait donc $w^{-1}Suff(y) = y^{-1}Suff(y) = \{\varepsilon\}$. Cela montre que w n'apparait qu'une fois dans y en tant que suffixe et termine la preuve. ∎

Le lemme suivant montre que l'image d'un facteur de y par la fonction suffixe est un mot de longueur maximale dans sa classe d'équivalence.

Lemme 5.15
Soit $u \in Fact(y) \setminus \{\varepsilon\}$. Alors, tout mot équivalent à $s(u)$ en est un suffixe.

Preuve On note $w = s(u)$ et soit $v \equiv_{Suff(y)} w$. Le mot w est un suffixe propre de u. Si la conclusion de l'énoncé est fausse, on obtient $w \prec_{suff} v$ d'après le lemme 5.10. Soit alors $z \in u^{-1}Suff(y)$. Comme w est un suffixe de u équivalent à v, on a $z \in w^{-1}Suff(y) = v^{-1}Suff(y)$. Alors, u et v sont des suffixes de yz^{-1}, ce qui implique que l'un est un suffixe de l'autre. Mais cela contredit soit la définition de $w = s(u)$ soit la conclusion du lemme 5.11, ce qui prouve que v est nécessairement un suffixe de $w = s(u)$. ∎

La propriété précédente est utilisée dans la section 6.6 où l'automate sert de machine de recherche de motifs. On peut vérifier que la propriété de s n'est pas satisfaite en général sur l'automate minimal qui accepte les facteurs (et pas seulement les suffixes) d'un mot, ou, plus exactement, ne l'est pas sur la fonction équivalente définie à partir de la congruence $\equiv_{Fact(y)}$.

Évolution de la congruence

La séquentialité de la construction de l'automate des suffixes repose sur des relations entre $\equiv_{Suff(wa)}$ et $\equiv_{Suff(w)}$ que l'on examine ici. Ce faisant, on considère que le mot générique y est égal à wa pour une certaine lettre a. Les propriétés énoncées permettent de déduire des bornes précises sur la taille de l'automate dans la section suivante.

La première relation (lemme 5.16) énonce que $\equiv_{Suff(wa)}$ est un raffinement de $\equiv_{Suff(w)}$.

Lemme 5.16
Soient $w \in A^*$ et $a \in A$. La congruence $\equiv_{Suff(wa)}$ est un raffinement de $\equiv_{Suff(w)}$, c'est-à-dire que, pour tous mots $u, v \in A^*$:

$u \equiv_{Suff(wa)} v$ implique $u \equiv_{Suff(w)} v$.

Preuve Supposons $u \equiv_{Suff(wa)} v$, soit $u^{-1}Suff(wa) = v^{-1}Suff(wa)$, et montrons $u \equiv_{Suff(w)} v$, soit $u^{-1}Suff(w) = v^{-1}Suff(w)$. On ne montre que $u^{-1}Suff(w) \subseteq v^{-1}Suff(w)$ car l'inclusion opposée s'en déduit par symétrie.

Si l'ensemble $u^{-1}Suff(w)$ est vide, l'inclusion est triviale. Sinon, soit $z \in u^{-1}Suff(w)$. On a alors $uz \preceq_{suff} w$, ce qui implique $uza \preceq_{suff} wa$. L'hypothèse donne $vza \preceq_{suff} wa$, et donc $vz \preceq_{suff} w$ ou $z \in v^{-1}Suff(w)$, ce qui termine la preuve. ∎

La congruence $\equiv_{Suff(w)}$ partitionne A^* en classes d'équivalence. Le lemme 5.16 revient à dire que ces classes sont unions de classes relativement à $\equiv_{Suff(wa)}$ $(a \in A)$. Il s'avère que seules une ou deux classes relativement à $\equiv_{Suff(w)}$ se partagent en deux sous-classes pour donner la partition induite par $\equiv_{Suff(wa)}$. Une de ces deux classes est celle qui provient des mots n'apparaissant pas dans w. Elle contient le mot wa lui-même qui produit une nouvelle classe et un nouvel état de l'automate de suffixes (voir lemme 5.17). Le théorème 5.19 et ses corollaires donnent des conditions pour le partage d'une autre classe et indiquent comment celle-ci se partage.

Lemme 5.17
Soient $w \in A^*$ et $a \in A$. Soit z le plus long suffixe de wa qui apparait dans w. Si u est un suffixe de wa strictement plus long que z, on a alors l'équivalence $u \equiv_{Suff(wa)} wa$.

Preuve C'est une conséquence directe du lemme 5.14 car z apparait au moins deux fois dans wa. ∎

Avant d'énoncer le théorème principal on donne une relation supplémentaire concernant les contextes droits.

Lemme 5.18
Soient $w \in A^*$ et $a \in A$. Alors, pour chaque mot $u \in A^*$, on a :

$$u^{-1}Suff(wa) = \begin{cases} \{\varepsilon\} \cup u^{-1}Suff(w)a & \text{si } u \preceq_{suff} wa \ , \\ u^{-1}Suff(w)a & \text{sinon} \ . \end{cases}$$

Preuve On remarque d'abord que $\varepsilon \in u^{-1}Suff(wa)$ est équivalent à $u \preceq_{suff} wa$. Il suffit donc de montrer $u^{-1}Suff(wa) \setminus \{\varepsilon\} = u^{-1}Suff(w)a$.

Soit z un mot non vide de $u^{-1}Suff(wa)$. On a $uz \preceq_{suff} wa$. Le mot uz s'écrit $uz'a$ avec $uz' \preceq_{suff} w$. Dès lors, $z' \in u^{-1}Suff(w)$, et ainsi $z \in u^{-1}Suff(w)a$.

Réciproquement, soit z un mot (non vide) de $u^{-1}Suff(w)a$. Il s'écrit $z'a$ pour $z' \in u^{-1}Suff(w)$. Donc, $uz' \preceq_{suff} w$, ce qui implique $uz = uz'a \preceq_{suff} wa$, soit $z \in u^{-1}Suff(wa)$, ce qui prouve la réciproque et termine la preuve. ∎

Théorème 5.19
Soient $w \in A^*$ et $a \in A$. Soit z le plus long suffixe de wa qui apparait dans w. Soit z' le plus long facteur de w pour lequel $z' \equiv_{Suff(w)} z$. Alors, pour chaque $u, v \preceq_{fact} w$:

$u \equiv_{Suff(w)} v$ *et* $u \not\equiv_{Suff(w)} z$ *implique* $u \equiv_{Suff(wa)} v$.

De plus, pour chaque u tel que $u \equiv_{Suff(w)} z$, on a :

$$u \equiv_{Suff(wa)} \begin{cases} z & \text{si } |u| \leq |z| \ , \\ z' & \text{sinon} \ . \end{cases}$$

Preuve Soient $u, v \preceq_{fact} w$ tels que $u \equiv_{Suff(w)} v$. Par définition de l'équivalence, on a $u^{-1}Suff(w) = v^{-1}Suff(w)$. On suppose, pour commencer, $u \not\equiv_{Suff(w)} z$ et l'on montre $u^{-1}Suff(wa) = v^{-1}Suff(wa)$, ce qui donne l'équivalence $u \equiv_{Suff(wa)} v$.

D'après le lemme 5.18, on a simplement à montrer que $u \preceq_{suff} wa$ est équivalent à $v \preceq_{suff} wa$. En fait, il suffit de montrer que $u \preceq_{suff} wa$ implique $v \preceq_{suff} wa$ puisque l'implication opposée s'en déduit par symétrie.

Supposons ainsi $u \preceq_{suff} wa$. On déduit de $u \preceq_{fact} w$ et de la définition de z que u est un suffixe de z. On peut donc considérer le plus grand indice $j \geq 0$ pour lequel $|u| \leq |{s_w}^j(z)|$. Notons que ${s_w}^j(z)$ est un suffixe de wa (de la même façon que z), et que le lemme 5.11 assure que $u \equiv_{Suff(w)} {s_w}^j(z)$. D'où, $v \equiv_{Suff(w)} {s_w}^j(z)$ par transitivité.

Comme $u \not\equiv_{Suff(w)} z$, on a $j > 0$. Le lemme 5.15 implique alors que v est un suffixe de ${s_w}^j(z)$, et donc aussi de wa comme souhaité. Cela montre la première partie de l'énoncé.

Considérons maintenant un mot u tel que $u \equiv_{Suff(w)} z$.

Quand $|u| \leq |z|$, pour montrer $u \equiv_{Suff(wa)} z$ en utilisant l'argument ci-dessus, on a seulement à vérifier que $u \preceq_{suff} wa$ car $z \preceq_{suff} wa$. Cela est en fait une simple conséquence du lemme 5.10.

Supposons $|u| > |z|$. L'existence d'un tel mot u implique $z' \neq z$ et $|z'| > |z|$ ($z \prec_{suff} z'$). En conséquence, par la définition de z, u et z' ne sont pas des suffixes de wa. Utilisant de nouveau l'argument ci-dessus, ceci prouve $u \equiv_{Suff(wa)} z'$ et termine la preuve. ∎

Les deux corollaires du théorème précédent énoncés ci-dessous font référence à des situations simples à gérer dans la construction de l'automate des suffixes.

Corollaire 5.20
Soient $w \in A^*$ et $a \in A$. Soit z le plus long suffixe de wa qui apparait dans w. Soit z' le plus long mot tel que $z' \equiv_{Suff(w)} z$. Supposons $z' = z$. Alors, pour chaque $u, v \preceq_{fact} w$, on a :

$u \equiv_{Suff(w)} v$ *implique* $u \equiv_{Suff(wa)} v$.

Preuve Soient $u, v \preceq_{fact} w$ tels que $u \equiv_{Suff(w)} v$. On montre l'équivalence $u \equiv_{Suff(wa)} v$. La conclusion vient directement du théorème 5.19 si $u \not\equiv_{Suff(w)} z$. Sinon, $u \equiv_{Suff(w)} z$; par l'hypothèse faite sur z et le lemme 5.10, on obtient $|u| \leq |z|$. Finalement, le théorème 5.19 donne la même conclusion. ∎

Corollaire 5.21
Soient $w \in A^*$ et $a \in A$. Si la lettre a n'apparait pas dans w, on a, pour chaque $u, v \preceq_{fact} w$:

$u \equiv_{Suff(w)} v$ *implique* $u \equiv_{Suff(wa)} v$.

Preuve Comme a n'apparait pas dans w, le mot z du corollaire 5.20 est le mot vide. Il est bien sûr le plus long de sa classe, ce qui permet d'appliquer le corollaire 5.20 et donne la même conclusion. ∎

5.4 Automate des suffixes

L'***automate des suffixes*** d'un mot y, noté $\mathcal{S}(y)$, est l'automate minimal qui accepte l'ensemble des suffixes de y. La structure est destinée à être utilisée comme un index sur le mot mais constitue également un dispositif pour la recherche des facteurs de y au sein d'un autre texte (voir chapitre 6). La propriété la plus surprenante de cet automate est que sa taille est linéaire en la longueur de y bien que le nombre de facteurs de y puisse être quadratique. La construction de l'automate prend également un temps linéaire sur un alphabet fixé. La figure 5.10 montre un exemple de tel automate.

Comme nous n'imposons pas à l'automate d'être complet, la classe des mots qui n'apparaissent pas dans y, dont le contexte droit est vide, n'est pas un état de $\mathcal{S}(y)$.

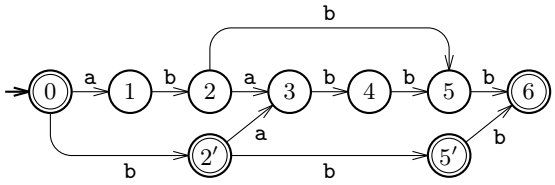


Figure 5.10 L'automate minimal des suffixes de `ababbb`.

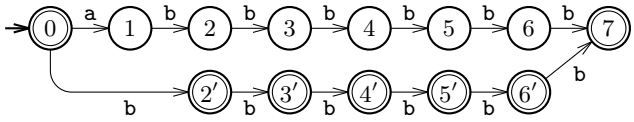


Figure 5.11 Un automate des suffixes avec le maximum d'états.

Taille de l'automate

La taille d'un automate est exprimée à la fois par le nombre de ses états et par le nombre de ses flèches. On montre que $\mathcal{S}(y)$ possède moins de $2|y|$ états et moins de $3|y|$ flèches, pour une taille totale $O(|y|)$, résultat qui s'appuie sur le théorème 5.19 de la section précédente. La figure 5.11 montre un automate qui possède le maximum d'états pour un mot de longueur 7.

Proposition 5.22
Soient $y \in A^*$ un mot de longueur n et $e(y)$ le nombre d'états de $\mathcal{S}(y)$. Pour $n = 0$, on a $e(y) = 1$; pour $n = 1$, on a $e(y) = 2$; pour $n > 1$ enfin, on a :

$$n + 1 \leq e(y) \leq 2n - 1 \ ,$$

et la borne supérieure est atteinte si et seulement si y est de la forme ab^{n-1}, pour deux lettres distinctes a, b.

Preuve Les égalités concernant les mots courts se vérifient directement. Supposons $n > 1$ pour la suite. Le nombre minimal d'états de $\mathcal{S}(y)$ est évidemment $n + 1$ (sinon le chemin d'étiquette y contiendrait un cycle entrainant un nombre infini de mots reconnus par l'automate), minimum qui est atteint avec $y = a^n$ ($a \in A$).

Démontrons la borne supérieure. Par le théorème 5.19, chaque lettre $y[i]$, $2 \leq i \leq n - 1$, accroit de deux au plus le nombre d'états de $\mathcal{S}(y[0\,..\,i-1])$. Comme le nombre d'états de $\mathcal{S}(y[0]y[1])$ est 3, il s'ensuit que $e(y) \leq 3 + 2(n - 2) = 2n - 1$, comme annoncé.

La construction d'un mot de longueur n dont l'automate des suffixes possède $2n - 1$ états est encore une simple application du théorème 5.19 en notant que chacune des lettres $y[2], y[3], \ldots, y[n-1]$ doit effectivement conduire à la création de deux états pendant la construction. On constate qu'après le choix des deux premières lettres qui doivent être différentes, les autres lettres sont imposées, ce qui produit la seule forme possible donnée dans l'énoncé. ∎

Lemme 5.23
Soient $y \in A^+$ et $f(y)$ le nombre de flèches de $\mathcal{S}(y)$. Alors :

$$f(y) \leq e(y) + |y| - 2 \ .$$

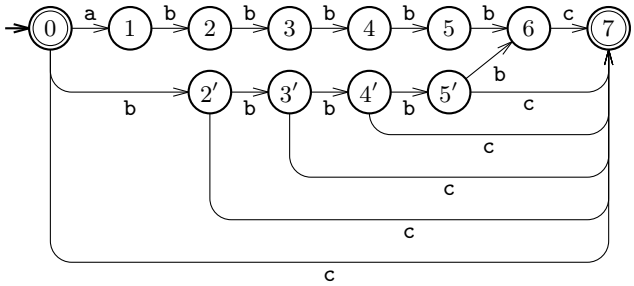


Figure 5.12 Un automate des suffixes avec le maximum de flèches.

Preuve Notons q_0 l'état initial de $\mathcal{S}(y)$, et considérons l'arbre recouvrant des plus longs chemins d'origine q_0 dans $\mathcal{S}(y)$. L'arbre contient $e(y) - 1$ flèches de $\mathcal{S}(y)$ car exactement une flèche entre sur chaque état sauf sur l'état initial.

À chaque autre flèche (p, a, q) de l'automate, on associe le suffixe uav de y défini comme suit : u est l'étiquette du chemin de l'arbre d'origine q_0 et de fin p ; v est l'étiquette du plus long chemin d'origine q ayant pour fin un état terminal. De cette façon, on obtient une injection de l'ensemble des flèches concernées dans celui des suffixes de y. Les suffixes y et ε ne sont pas considérés car ils sont étiquettes de chemins de l'arbre recouvrant. Cela montre qu'il y a au plus $|y| - 1$ flèches supplémentaires.

Soit un total de $e(y) + |y| - 2$ flèches au maximum. ∎

La figure 5.12 montre un automate qui possède le maximum de flèches pour un mot de longueur 7, comme le montre la proposition suivante.

Proposition 5.24
Soient $y \in A^*$ de longueur n et $f(y)$ le nombre de flèches de $\mathcal{S}(y)$. Pour $n = 0$, on a $f(y) = 0$; pour $n = 1$, on a $f(y) = 1$; pour $n = 2$, on a $f(y) = 2$ ou $f(y) = 3$; pour $n > 2$ enfin, on a :

$$n \leq f(y) \leq 3n - 4 \ ,$$

et la borne supérieure est atteinte pour y de la forme $ab^{n-2}c$, où a, b et c sont trois lettres deux-à-deux distinctes.

Preuve On vérifie directement les résultats sur les mots courts. Considérons $n > 2$. La borne inférieure est immédiate et atteinte pour le mot $y = a^n$ $(a \in A)$.

Examinons ensuite la borne supérieure. Par la proposition 5.22 et le lemme 5.23, on obtient $f(y) \leq (2n - 1) + n - 2 = 3n - 3$. La quantité $2n - 1$ est le nombre maximal d'états obtenus uniquement si $y = ab^{n-1}$ $(a, b \in A, a \neq b)$. Mais pour ce mot le nombre de flèches n'est que $2n - 1$. Donc, $f(y) \leq 3n - 4$.

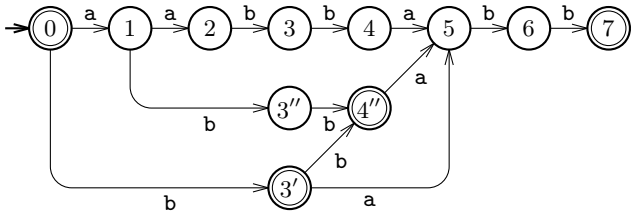


Figure 5.13 L'automate des suffixes $\mathcal{S}(\texttt{aabbabb})$. Les liens suffixes des états sont : $F[1] = 0$, $F[2] = 1$, $F[3] = 3''$, $F[3''] = 3'$, $F[3'] = 0$, $F[4] = 4''$, $F[4''] = 3'$, $F[5] = 1$, $F[6] = 3''$, $F[7] = 4''$. Le chemin suffixe de 7 est $\langle 7, 4'', 3', 0\rangle$ qui contient tous les états terminaux de l'automate (voir corollaire 5.27).

On vérifie que l'automate $\mathcal{S}(ab^{n-2}c)$ $(a, b, c \in A,\ \mathrm{card}\{a, b, c\} = 3)$ possède $2n - 2$ états et $3n - 4$ flèches. ∎

L'énoncé qui suit est une conséquence immédiate des propositions 5.22 et 5.24.

Théorème 5.25
La taille totale de l'automate des suffixes d'un mot est linéaire en la longueur du mot. ∎

Lien suffixe et chemins suffixes

Le théorème 5.19 et ses deux corollaires consécutifs fournissent la trame de la construction séquentielle de l'automate des suffixes $\mathcal{S}(y)$. L'algorithme contrôle les conditions qui apparaissent dans ces énoncés au moyen d'une fonction définie sur les états de l'automate, le lien suffixe, et d'une classification des flèches en solides et non solides. Nous définissons ces deux notions ci-après.

Soit p un état de $\mathcal{S}(y)$, différent de l'état initial. L'état p est une classe de facteurs de y congruents vis-à-vis de l'équivalence $\equiv_{Suff(y)}$. Soit u un mot quelconque de la classe ($u \neq \varepsilon$ car p n'est pas l'état initial). On définit la ***cible suffixe*** de p, notée $f(p)$, comme la classe de congruence de $s(u)$. La fonction f est appelée le ***lien suffixe*** de l'automate. D'après le lemme 5.13 la valeur de $s(u)$ est indépendante du mot u choisi dans la classe p, ce qui rend la définition de f cohérente. Le lien suffixe est une fonction de suppléance au sens de la section 1.4, c'est-à-dire que $f(p)$ est le suppléant de p. Le lien est utilisé avec cette signification en section 6.6. Un exemple est donné dans la figure 5.13.

Pour un état p de $\mathcal{S}(y)$, on note $lg(p)$ la longueur du plus long mot u de la classe de congruence p. C'est aussi la longueur du plus long chemin d'origine l'état initial et de fin p, chemin qui est étiqueté par u. Les plus longs chemins issus de l'état initial forment un arbre recouvrant de $\mathcal{S}(y)$ (conséquence du lemme 5.10). Les flèches qui appartiennent à cet arbre

sont qualifiées de ***solides***. De façon équivalente :

la flèche (p, a, q) est solide

si et seulement si

$lg(q) = lg(p) + 1$.

Cette notion de solidité des flèches est utilisée dans la construction de l'automate pour tester la condition du théorème 5.19.

Les cibles suffixes induisent par itération des chemins suffixes dans $\mathcal{S}(y)$ (voir figure 5.13). On peut noter que :

$q = f(p)$ implique $lg(q) < lg(p)$.

Ainsi, la suite

$\langle p, f(p), f^2(p), \ldots \rangle$

est finie et se termine par l'état initial (qui n'a pas de cible suffixe). Elle est appelée le ***chemin suffixe*** de p dans $\mathcal{S}(y)$, et notée $CS(p)$.

Soit *dernier* l'état de $\mathcal{S}(y)$ qui est la classe de y lui-même. Cet état est caractérisé par le fait qu'il n'est l'origine d'aucune flèche (sinon $\mathcal{S}(y)$ accepterait des mots plus longs que y). Le chemin suffixe de *dernier*,

$\langle dernier, f(dernier), f^2(dernier), \ldots, f^{k-1}(dernier) = q_0 \rangle$,

où q_0 est l'état initial de l'automate, joue un rôle important dans l'algorithme de construction séquentielle. Il est utilisé pour tester efficacement les conditions du théorème 5.19 et de ses corollaires (δ fonction de transition de $\mathcal{S}(y)$).

Proposition 5.26
Soient $u \in Fact(y) \setminus \{\varepsilon\}$ et $p = \delta(q_0, u)$. Alors, pour chaque entier $j \geq 0$ pour lequel $s^j(u)$ est défini, on a :

$f^j(p) = \delta(q_0, s^j(u))$.

Preuve On prouve le résultat par récurrence sur j. Si $j = 0$, $f^j(p) = p$ et $s^j(u) = u$, donc l'égalité est satisfaite par hypothèse. Soit alors $j > 0$ tel que $s^j(u)$ est défini et supposons par hypothèse de récurrence que :

$f^{j-1}(p) = \delta(i, s^{j-1}(u))$.

Par définition de f, $f(f^{j-1}(p))$ est la classe de congruence du mot $s(s^{j-1}(u))$. Il vient en conséquence que :

$f^j(p) = \delta(q_0, s^j(u))$.

Ce qui achève la récurrence et la preuve. ∎

Corollaire 5.27
Les états terminaux de $\mathcal{S}(y)$ sont les états du chemin suffixe de dernier, CS(dernier).

Preuve On montre pour commencer que les états du chemin suffixe sont terminaux. Soit p un état du chemin suffixe de *dernier*. On a $p = f^j(\mathit{dernier})$ pour un entier $j \geq 0$. Comme $\mathit{dernier} = \delta(q_0, y)$, la proposition 5.26 implique $p = \delta(q_0, s^j(y))$. Et comme $s^j(y)$ est un suffixe de y, p est un état terminal.

Réciproquement, soit p un état terminal de $\mathcal{S}(y)$. Soit alors $u \preceq_{\mathit{suff}} y$ tel que $p = \delta(q_0, u)$. Comme $u \preceq_{\mathit{suff}} y$, on peut considérer le plus grand entier $j \geq 0$ pour lequel $|u| \leq |s^j(y)|$. Par le lemme 5.11, on obtient $u \equiv_{\mathit{Suff}(y)} s^j(y)$. Donc, $p = \delta(q_0, s^j(y))$ par définition de $\mathcal{S}(y)$. Ainsi, la proposition 5.26 appliquée à y implique $p = f^j(\mathit{dernier})$, ce qui prouve que p apparait dans $CS(\mathit{dernier})$. Cela termine la preuve. ∎

Construction séquentielle

Il est possible de construire l'automate des suffixes de y par application d'algorithmes standard de minimisation d'automates appliqués à l'arbre des suffixes de la section 5.1. Mais l'arbre des suffixes peut être de taille quadratique, ce qui donne la complexité de cette approche. On présente un algorithme de construction séquentielle qui évite ce problème et travaille en espace linéaire avec un temps d'exécution $O(|y| \times \log \operatorname{card} A)$.

L'algorithme traite les préfixes de y du plus court, ε, au plus long, y lui-même. À chaque étape, juste après avoir traité le préfixe w, on possède les informations suivantes :

- l'automate des suffixes $\mathcal{S}(w)$ avec sa fonction de transition δ ;
- l'attribut F, défini sur les états de $\mathcal{S}(w)$, qui implante la fonction suffixe f_w ;
- l'attribut L, défini sur les états de $\mathcal{S}(w)$, qui implante la fonction de longueur lg_w ;
- l'état *dernier*.

Les états terminaux de $\mathcal{S}(w)$ ne sont pas marqués explicitement, ils sont donnés implicitement par le chemin suffixe de *dernier* (corollaire 5.27). L'implantation de $\mathcal{S}(w)$ avec ces éléments supplémentaires est discutée ci-dessous juste avant l'analyse de la complexité du calcul.

L'algorithme de construction AUTO-SUFFIXES est basé sur l'utilisation de la procédure EXTENSION donnée plus loin. Cette procédure traite la lettre courante a du mot y. Elle transforme l'automate des suffixes $\mathcal{S}(w)$ déjà construit en l'automate des suffixes $\mathcal{S}(wa)$ ($wa \preceq_{\mathit{préf}} y$, $a \in A$). Un exemple de son fonctionnement est donné figure 5.14.

```
AUTO-SUFFIXES(y, n)
 1  M ← NOUVEL-AUTOMATE()
 2  L[initial[M]] ← 0
 3  F[initial[M]] ← NIL
 4  dernier[M] ← initial[M]
 5  pour chaque lettre a de y, séquentiellement faire
 6        ▷ Extension de M par la lettre a
 7        EXTENSION(a)
 8  p ← dernier[M]
 9  faire  terminal[p] ← VRAI
10        p ← F[p]
11  tantque p ≠ NIL
12  retourner M

EXTENSION(a)
 1  nouveau ← NOUVEL-ÉTAT()
 2  L[nouveau] ← L[dernier[M]] + 1
 3  p ← dernier[M]
 4  faire  Succ[p] ← Succ[p] ∪ {(a, nouveau)}
 5        p ← F[p]
 6  tantque p ≠ NIL et CIBLE(p, a) = NIL
 7  si p = NIL alors
 8        F[nouveau] ← initial[M]
 9  sinon q ← CIBLE(p, a)
10        si (p, a, q) est solide, soit L[p] + 1 = L[q] alors
11              F[nouveau] ← q
12        sinon clone ← NOUVEL-ÉTAT()
13              L[clone] ← L[p] + 1
14              pour chaque couple (b, q′) ∈ Succ[q] faire
15                    Succ[clone] ← Succ[clone] ∪ {(b, q′)}
16              F[nouveau] ← clone
17              F[clone] ← F[q]
18              F[q] ← clone
19              faire  Succ[p] ← Succ[p] \ {(a, q)}
20                    Succ[p] ← Succ[p] ∪ {(a, clone)}
21                    p ← F[p]
22              tantque p ≠ NIL et CIBLE(p, a) = q
23  dernier[M] ← nouveau
```

Théorème 5.28
L'algorithme AUTO-SUFFIXES *construit un automate des suffixes, c'est-à-dire que l'opération* AUTO-SUFFIXES(y) *produit l'automate* $\mathcal{S}(y)$*, pour* $y \in A^*$.

Preuve On montre par récurrence sur $|y|$ que l'automate est calculé correctement, ainsi que les attributs L et F et la variable *dernier*. On

montre ensuite que les états terminaux sont calculés correctement.

Si $|y| = 0$, l'algorithme construit un automate comprenant un seul état qui est à la fois initial et terminal. Aucune transition n'est définie. L'automate reconnait donc le langage $\{\varepsilon\}$ qui est $Suff(y)$. Les éléments F, L et *dernier* sont aussi correctement calculés.

On considère maintenant que $|y| > 0$, et soit $y = wa$, pour $a \in A$ et $w \in A^*$. On suppose, par récurrence, que l'automate courant M est $\mathcal{S}(w)$ avec sa fonction de transition δ_w, que $q_0 = initial[M]$, $dernier = \delta_w(q_0, w)$, que l'attribut L satisfait $L[p] = lg_w(p)$ pour tout état p, et que l'attribut F satisfait $F[p] = f_w(p)$ pour tout état p différent de l'état initial.

On montre d'abord que la procédure Extension effectue correctement la transformation de l'automate M, de la variable *dernier*, et des attributs L et F.

La variable p de la procédure Extension parcourt les états du chemin suffixe $CS(dernier)$ de $\mathcal{S}(w)$. La première boucle crée des transitions étiquetées par a et de cible le nouvel état *nouveau* en accord avec le lemme 5.17. On a aussi l'égalité $L[nouveau] = lg(nouveau)$.

Quand la première boucle s'arrête, trois cas disjoints se présentent :

1. p n'est pas défini.
2. (p, a, q) est une flèche solide.
3. (p, a, q) est une flèche non solide.

Cas 1. Cette situation se produit lorsque la lettre a n'apparait pas dans w ; on a alors $f_y(nouveau) = q_0$. On obtient donc, après l'exécution de l'instruction de la ligne 8, l'égalité $F[nouveau] = f_y(nouveau)$. Pour les autres états r, on a $f_w(r) = f_y(r)$ d'après le corollaire 5.21, ce qui donne les égalités $F[r] = f_y(r)$ à la fin de l'exécution de la procédure Extension.

Cas 2. Soit u le plus long mot pour lequel $\delta_w(q_0, u) = p$. Par récurrence et par le lemme 5.15, on a $|u| = lg_w(p) = L[p]$. Le mot ua est le plus long suffixe de y qui est un facteur de w. Il s'ensuit que $f_y(nouveau) = q$. Ce qui montre que l'on a $F[nouveau] = f_y(nouveau)$ après l'instruction de la ligne 11.

Comme la flèche (p, a, q) est solide, par récurrence de nouveau, on a $|ua| = L[q] = lg(q)$, ce qui montre que les mots équivalents à ua selon $\equiv_{Suff(w)}$ ne sont pas plus longs que ua. Le corollaire 5.20 s'applique avec $z = ua$. Et comme dans le cas 1, $F[r] = f_y(r)$ pour tous les états différents de *nouveau*.

Cas 3. Soit u le plus long mot pour lequel $\delta_w(q_0, u) = p$. Le mot ua est le plus long suffixe de y qui est un facteur de w. Comme la flèche (p, a, q) n'est pas solide, ua n'est pas le plus long mot de sa classe de congruence selon $\equiv_{Suff(w)}$. Le théorème 5.19 s'applique avec $z = ua$, et z' le plus long mot pour lequel $\delta_w(q_0, z') = q$. La classe de ua selon $\equiv_{Suff(w)}$ se partage en deux sous-classes selon $\equiv_{Suff(y)}$ correspondant aux états q et *clone*.

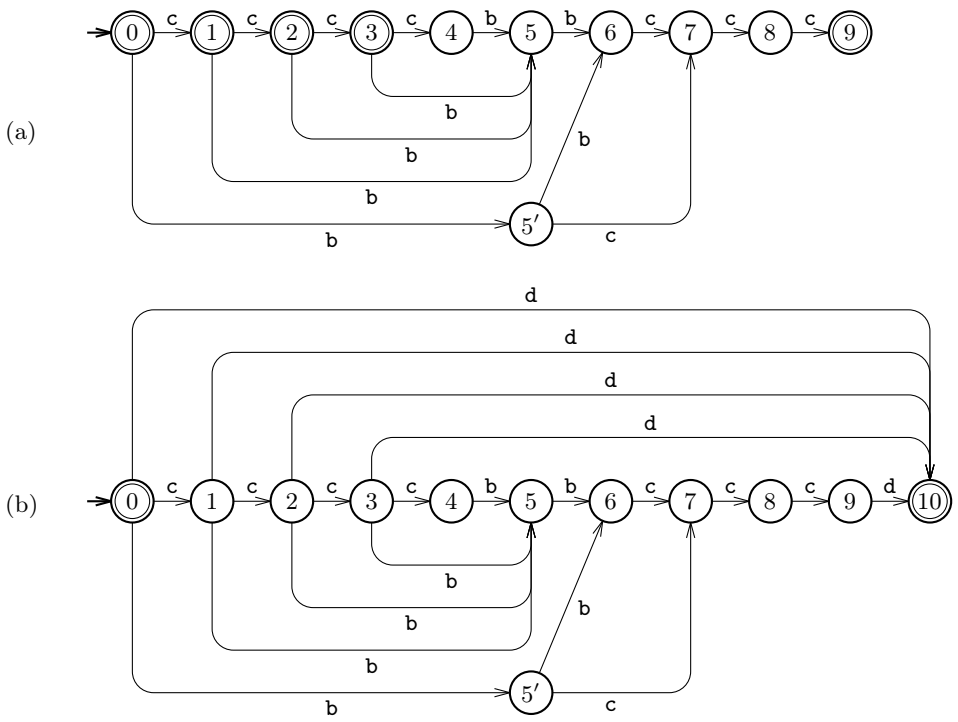
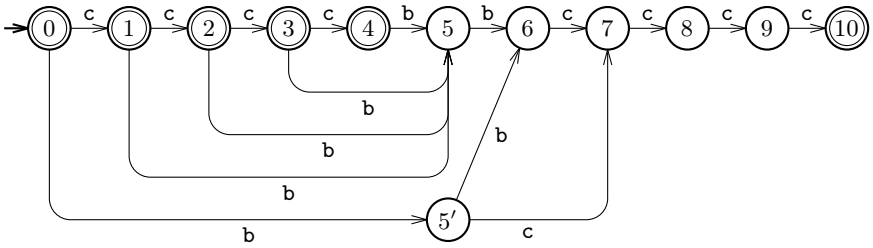
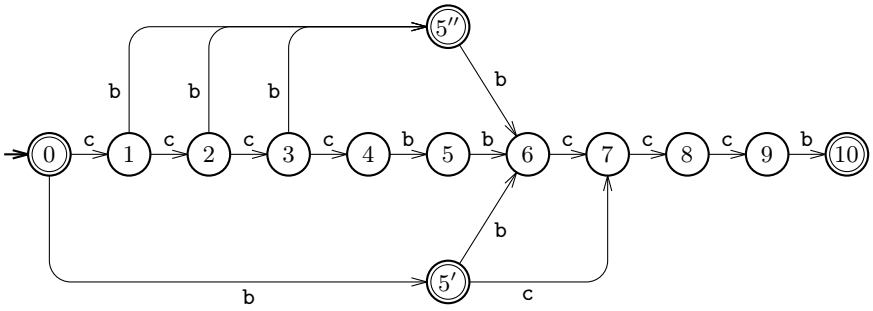


Figure 5.14 Illustration du fonctionnement de la procédure EXTENSION(a) sur l'automate des suffixes $\mathcal{S}(\texttt{ccccbbccc})$ selon trois cas. **(a)** L'automate $\mathcal{S}(\texttt{ccccbbccc})$. **(b)** Cas où $a = \texttt{d}$. Pendant l'exécution de la première boucle de la procédure, l'état p parcourt le chemin suffixe $\langle 9, 3, 2, 1, 0\rangle$. En même temps, des flèches étiquetées par la lettre `d` sont créées, sortant de ces états et aboutissant sur 10, le dernier état créé. Le parcours s'arrête sur l'état initial. Cette situation correspond au corollaire 5.21. **(c)** Cas où $a = \texttt{c}$. La première boucle de la procédure s'arrête sur l'état $3 = F[9]$ parce qu'une flèche étiquetée par `c` sort de cet état. De plus, la flèche $(3, \texttt{c}, 4)$ est solide. On obtient directement la cible suffixe du nouvel état créé : $F[10] = \delta(3, \texttt{c}) = 4$. Il n'y a rien d'autre à faire selon le corollaire 5.20. **(d)** Cas où $a = \texttt{b}$. La première boucle de la procédure s'arrête sur l'état $3 = F[9]$ parce qu'une flèche étiquetée par `b` sort de cet état. Dans l'automate $\mathcal{S}(\texttt{ccccbbccc})$, la flèche $(3, \texttt{b}, 5)$ n'est pas solide. Le mot `cccb` est suffixe de `ccccbbcccb` mais `ccccb` ne l'est pas, bien que ces deux mots conduisent, dans $\mathcal{S}(\texttt{ccccbbccc})$, à l'état 5. Afin d'obtenir `ccccbbcccb`, cet état est dupliqué en l'état terminal $5''$ qui est la classe des facteurs `cccb`, `ccb` et `cb`. Les flèches $(3, \texttt{b}, 5)$, $(2, \texttt{b}, 5)$ et $(1, \texttt{b}, 5)$ de $\mathcal{S}(\texttt{ccccbbccc})$ sont redirigées vers $5''$ conformément au théorème 5.19. Et $F[10] = 5''$, $F[5] = 5''$, $F[5''] = 5'$.

(c)



(d)



Les mots v plus courts que ua et tels que $v \equiv_{Suff(w)} ua$ sont de la forme $v'a$ avec $v' \preceq_{suff} u$ (conséquence du lemme 5.10). Avant l'exécution de la dernière boucle, tous ces mots v satisfont $q = \delta_w(q_0, v)$. En conséquence, après l'exécution de la boucle, ils satisfont $clone = \delta_y(q_0, v)$, comme indiqué par le théorème 5.19. Les mots v plus longs que ua et tels que $v \equiv_{Suff(w)} ua$ satisfont $q = \delta_y(q_0, v)$ après l'exécution de la boucle comme indiqué par le théorème 5.19 à nouveau. On vérifie que l'attribut F est mis à jour correctement.

Enfin, dans chacun des trois cas, on vérifie que la valeur de *dernier* est correctement calculée à la fin de l'exécution de la procédure EXTENSION.

Finalement, la récurrence montre que l'automate M, l'état *dernier*, ainsi que les attributs L et F sont corrects après l'exécution de la procédure EXTENSION.

Il reste à vérifier que les états terminaux sont correctement marqués lors de l'exécution de la dernière boucle de l'algorithme AUTO-SUFFIXES. Mais cela est une conséquence du corollaire 5.27 car la variable p parcourt le chemin suffixe de *dernier*. ∎

Complexité

Pour analyser la complexité de l'algorithme AUTO-SUFFIXES, on commence par décrire une implantation possible des éléments nécessaires à la construction.

On suppose que l'automate est représenté par ensembles de successeurs étiquetés. Ce faisant, les opérations d'ajout, d'accès et de mise à jour concernant une flèche s'exécutent en temps $O(\log \operatorname{card} A)$ avec une

implantation efficace des ensembles dans le modèle comparaisons (voir section 1.4). La fonction f est réalisée par l'attribut F qui donne accès à $f(p)$ en temps constant.

Afin d'implanter la solidité des flèches, on utilise l'attribut L, représentant la fonction lg, comme le suggère la description de la procédure EXTENSION (ligne 10). Une autre façon de faire consiste à utiliser un booléen par flèche de l'automate. Cela entraine une légère modification de l'algorithme qui peut se décrire ainsi : chaque première flèche créée au cours de l'exécution des boucles aux lignes 4–6 et aux lignes 19–22 doit être marquée comme solide, les autres flèches créées sont marquées comme étant non solides. Ce type d'implantation ne nécessite pas l'utilisation de l'attribut L qui peut alors être éliminé, ce qui permet un gain en espace mémoire. Néanmoins, l'attribut L trouve son utilité dans des applications comme celles du chapitre 6. On retient que les deux types d'implantations permettent d'accéder en temps constant à la qualité d'une flèche (solide ou non solide).

Théorème 5.29
L'algorithme AUTO-SUFFIXES *peut être implanté de telle sorte que la construction de* $\mathcal{S}(y)$ *prenne un temps* $O(|y| \times \log \operatorname{card} A)$ *dans un espace mémoire* $O(|y|)$.

Preuve On choisit une implantation de la fonction de transition par ensembles de successeurs étiquetés. Les états de $\mathcal{S}(y)$ et les attributs F et L requièrent un espace $O(e(y))$, les ensembles de successeurs étiquetés un espace $O(f(y))$. Ainsi, l'implantation complète prend un espace $O(|y|)$, comme conséquence des propositions 5.22 et 5.24.

Une autre conséquence de ces propositions est que toutes les opérations exécutées une fois par état ou une fois par flèche de l'automate final prennent un temps total $O(|y| \times \log \operatorname{card} A)$. Le même résultat vaut pour les opérations qui sont exécutées une fois par lettre de y. Il reste donc à montrer que le temps passé à l'exécution des deux boucles aux lignes 4–6 et 19–22 de la procédure EXTENSION est du même ordre, à savoir $O(|y| \times \log \operatorname{card} A)$.

On examine d'abord le cas de la première boucle aux lignes 4–6. Considérons l'exécution de la procédure EXTENSION pendant la transformation de $\mathcal{S}(w)$ en $\mathcal{S}(wa)$ ($wa \preceq_{préf} y$, $a \in A$). Soit u le plus long mot de l'état p durant le test à la ligne 6. La valeur initiale de u est $s_w(w)$, et sa valeur finale satisfait $ua = s_{wa}(wa)$ (si p est défini). Soit $k = |w| - |u|$, position de l'occurrence suffixe de u dans w. Alors, chaque test accroit strictement la valeur de k pendant un appel de la procédure. De plus, la valeur initiale de k au début de l'exécution du prochain appel n'est pas plus petite que sa valeur finale atteinte en fin d'exécution de l'appel courant. Donc, les tests et instructions de cette boucle sont exécutés au plus $|y|$ fois au cours de tous les appels à EXTENSION.

Un argument similaire vaut pour la seconde boucle aux lignes 19–22

de la procédure EXTENSION. Soit v le plus long mot de l'état p pendant le test de la boucle. La valeur initiale de v est ${s_w}^j(w)$, pour $j \geq 2$, et sa valeur finale satisfait $va = {s_{wa}}^2(wa)$ (si p est défini). Alors, la position de v comme suffixe de w s'accroit strictement à chaque test au cours des appels successifs de la procédure. De nouveau, tests et instructions de la boucle sont exécutés au plus $|y|$ fois.

En conséquence, le temps cumulé des exécutions des deux boucles est $O(|y| \times \log \operatorname{card} A)$, ce qui termine la preuve. ∎

Sur un petit alphabet, on peut choisir une implantation de l'automate plus efficace encore que celle par ensembles de successeurs étiquetés, au détriment toutefois de l'espace mémoire. Il suffit d'utiliser pour cela une matrice de transition gérée comme une matrice creuse. Avec cette gestion particulière, les opérations sur les flèches s'exécutent en temps constant, ce qui conduit au résultat suivant.

Théorème 5.30
Dans le modèle branchements, le construction de $\mathcal{S}(y)$ par l'algorithme AUTO-SUFFIXES *prend un temps $O(|y|)$.*

Preuve On peut utiliser, afin d'implanter la matrice de transitions, la technique de représentation des matrices creuses qui donne un accès direct à chacune de ses entrées tout en évitant d'initialiser complètement chacune d'elles (voir exercice 1.15). ∎

5.5 Automate compact des suffixes

Dans cette section, on décrit succinctement une méthode pour construire un ***automate compact des suffixes***, noté $\mathcal{S}_{\mathrm{C}}(y)$ pour $y \in A^*$. Cet automate peut être vu comme la version compacte de l'automate des suffixes de la section précédente, c'est-à-dire qu'il est obtenu par suppression des états qui ne possèdent qu'un seul successeur et ne sont pas terminaux. C'est le procédé qui est utilisé sur l'arbre de suffixes à la section 5.2 pour obtenir une structure de taille linéaire.

L'automate compact des suffixes est aussi la version minimisée, au sens des automates, de l'arbre compact des suffixes. Il s'obtient en identifiant les sous-arbres qui reconnaissent les mêmes mots.

La figure 5.15 présente l'automate compact des suffixes de `ababbb` que l'on peut comparer à l'arbre compact de la figure 5.5 et à l'automate de la figure 5.10.

Exactement comme pour l'arbre $\mathcal{A}(\mathit{Suff}(y))$, on appelle ***fourche***, dans l'automate $\mathcal{S}(y)$, tout état qui est de degré (sortant) au moins 2, ou qui est à la fois de degré 1 et terminal. Les fourches de l'automate des suffixes satisfont la même propriété que celle des fourches de l'arbre des

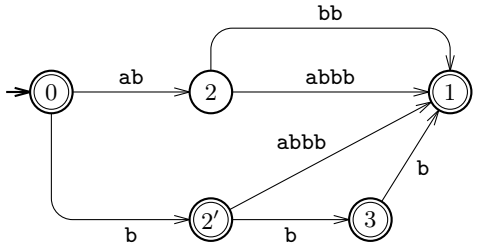
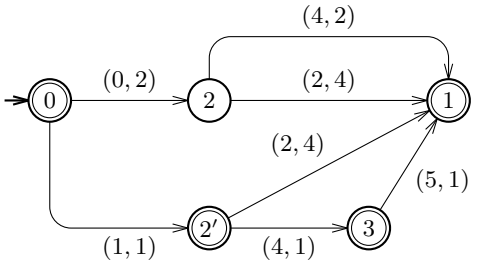


Figure 5.15 L'automate compact des suffixes $\mathcal{S}_{\mathrm{C}}(\texttt{ababbb})$.



i	0	1	2	3	4	5
$y[i]$	a	b	a	b	b	b

Figure 5.16 Représentation des étiquettes dans l'automate compact des suffixes $\mathcal{S}_{\mathrm{C}}(y)$ avec $y = \texttt{ababbb}$ (à comparer à la figure 5.15).

suffixes, propriété qui permet la compaction de l'automate. La preuve de la proposition qui suit est une adaptation immédiate de celle de la proposition 5.5 et est laissée au lecteur.

Proposition 5.31
Dans l'automate des suffixes d'un mot,la cible suffixe d'une fourche (différente de l'état initial) est une fourche. ∎

Lorsqu'on supprime les états qui ont un degré sortant de 1 et qui ne sont pas terminaux, les flèches de l'automate doivent être étiquetées par des mots (non vides) et plus seulement par des lettres. Afin d'obtenir une structure de taille linéaire en la longueur de y, il est nécessaire de ne pas mémoriser ces étiquettes sous une forme explicite. On procède comme pour l'arbre compact des suffixes en les représentant en espace constant au moyen d'un couple d'entiers. Si le mot u est l'étiquette d'une flèche (p, q), il est représenté par le couple $(i, |u|)$ pour lequel i est la position d'une occurrence de u dans y. On note $\mathit{étiq}(p, q) = (i, |u|)$ et l'on suppose que l'implantation de l'automate procure un accès direct à cette étiquette. Cela impose de stocker le mot y en même temps que la structure. La figure 5.16 indique comment sont représentées les étiquettes de l'automate compact des suffixes de ababbb.

La taille de l'automate compact des suffixes s'évalue assez directement à partir de celles de l'arbre compact et de l'automate des suffixes.

Proposition 5.32
Soient $y \in A^*$ de longueur n et $e_{\mathrm{C}}(y)$ le nombre d'états de $\mathcal{S}_{\mathrm{C}}(y)$. Pour $n = 0$, on a $e_{\mathrm{C}}(y) = 1$; pour $n > 0$, on a :

$$2 \leq e_{\mathrm{C}}(y) \leq n + 1 \ ,$$

et la borne supérieure est atteinte pour $y = a^n$, $a \in A$.

Preuve Le résultat se vérifie directement pour le mot vide.

Supposons $n > 0$. Soit \$ une lettre spéciale, \$ $\notin$ *alph*(y), et considérons l'arbre $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y\$))$. Cet arbre possède exactement $n + 1$ nœuds externes sur chacun desquels entre une flèche dont l'étiquette se termine précisément par la lettre \$. Il possède au plus n nœuds internes car ceux-ci sont de degré au moins 2. Lorsqu'on le minimise pour obtenir un automate compact, tous les nœuds externes sont identifiés en un seul état, ce qui réduit le nombre d'états à $n+1$ au plus. La suppression de la lettre \$ n'augmente pas cette valeur, ce qui donne la borne supérieure. Il est immédiat de vérifier que $\mathcal{S}_{\mathrm{C}}(a^n)$ possède exactement $n+1$ états et que la borne inférieure évidente est atteinte lorsque l'alphabet de y est de taille n, card *alph*$(y) = n$, pour $n > 0$. ∎

Proposition 5.33
Soient $y \in A^*$ de longueur n et $f_{\mathrm{C}}(y)$ le nombre de flèches de $\mathcal{S}_{\mathrm{C}}(y)$. Pour $n = 0$, on a $f_{\mathrm{C}}(y) = 0$; pour $n = 1$, on a $f_{\mathrm{C}}(y) = 1$; pour $n > 1$ enfin, on a :

$$f_{\mathrm{C}}(y) \leq 2(n-1) \ ,$$

et la borne supérieure est atteinte pour $y = a^{n-1}b$, a, b étant deux lettres distinctes.

Preuve Après vérification des résultats pour les mots courts, on note que si x est de la forme a^n, $n > 1$, on a $f_{\mathrm{C}}(y) = n - 1$, laquelle quantité est inférieure à $2(n-1)$.

Supposons maintenant que card *alph*$(y) \geq 2$. On poursuit la preuve du lemme précédent en considérant toujours le mot $y\$$, \$ $\notin$ *alph*(y). Son arbre compact possède au plus $2n$ nœuds, sa racine étant de degré au moins 3. Il possède donc au plus $2n - 1$ flèches qui après compaction donnent $2n - 2$ flèches puisque celle qui est étiquetée par \$ d'origine l'état initial disparait. Cela donne la majoration annoncée. On vérifie enfin directement que l'automate $\mathcal{S}_{\mathrm{C}}(a^{n-1}b)$ possède exactement n états et $2n - 2$ flèches. ∎

La construction de l'automate $\mathcal{S}_{\mathrm{C}}(y)$ peut être effectuée à partir de l'arbre $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ ou de l'automate $\mathcal{S}(y)$ (voir exercices 5.12 et 5.13). Cependant, pour économiser l'espace mémoire au cours de la construction, on a intérêt à utiliser une construction directe. C'est le schéma de cette construction que l'on décrit ici.

La construction emprunte des éléments aux algorithmes ARBRE-C-SUFFIXES et AUTO-SUFFIXES. Ainsi, les flèches de l'automate sont-elles marquées comme solides ou non solides. Les flèches créées vers de nouvelles feuilles de l'arbre deviennent des flèches vers l'état *dernier*. On utilise aussi les notions de descentes lente et rapide de la construction de l'arbre compact. Ce sont sur ces deux procédures que les changements sont essentiels et que l'on retrouve les duplications d'états et les redirections de flèches de la construction de l'automate des suffixes.

Pendant l'exécution d'une descente lente, la tentative de traversée d'une flèche non solide conduit au clonage de sa cible, c'est-à-dire à une duplication analogue à celle qui intervient pendant l'exécution de la procédure EXTENSION aux lignes 12–22. On peut noter que certaines flèches peuvent se trouver redirigées par ce processus.

Le deuxième point important dans l'adaptation des algorithmes des sections précédentes porte sur la procédure de descente rapide. L'algorithme y fait appel pour la définition d'une cible suffixe comme dans l'algorithme ARBRE-C-SUFFIXES. La différence intervient ici lors de la création de la cible suffixe d'une fourche nouvellement créée (voir lignes 8–11 dans la procédure DESC-RAPIDE). Si le nouvel état doit être créé par coupure d'une flèche solide, le même procédé s'applique. En revanche, si la flèche est non solide, il n'y a dans un premier temps que redirection de la flèche vers la fourche en question, avec mise à jour de son étiquette. Cela laisse indéfinie la cible suffixe et conduit à une itération du même procédé.

Les phénomènes qui viennent d'être décrits interviennent dans toute construction séquentielle de ce type d'automate. Leur prise en compte est nécessaire au bon fonctionnement de l'algorithme de calcul séquentiel de $\mathcal{S}_{\mathrm{C}}(y)$. Ils sont présents dans la construction de $\mathcal{S}_{\mathrm{C}}(\texttt{ababbb})$ (voir figure 5.15) pour laquelle trois étapes sont détaillées dans la figure 5.17.

En conclusion de cette section, on énonce le résultat sur la construction directe de l'automate compact des suffixes. La description et la preuve formelles de l'algorithme sont laissées au soin du lecteur.

Proposition 5.34
Le calcul de l'automate compact des suffixes $\mathcal{S}_{\mathrm{C}}(y)$ peut être réalisé en temps $O(|y| \times \log \operatorname{card} A)$ dans un espace $O(|y|)$. Dans le modèle branchements, le calcul s'exécute en temps $O(|y|)$. ∎

Notes

La notion d'arbre des positions est due à Weiner (1973) qui a présenté un algorithme de calcul de sa version compacte. L'algorithme de la section 5.2 est de McCreight (1976). Une version strictement séquentielle de la construction de l'arbre compact a été décrite par Ukkonen (1995).

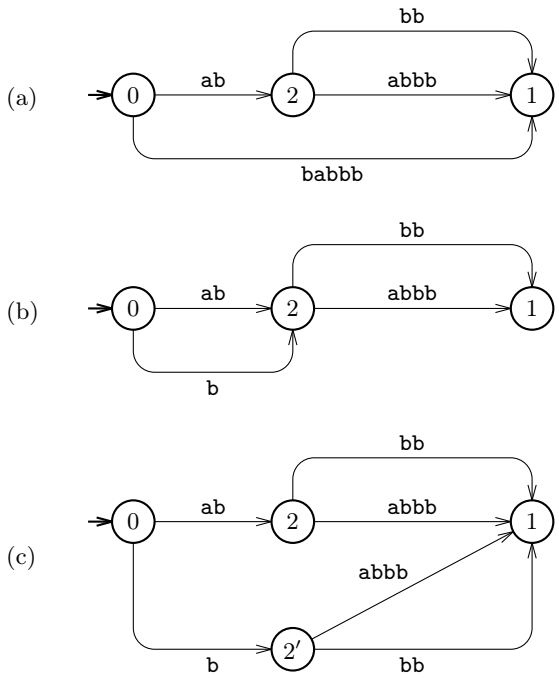


Figure 5.17 Trois étapes de la construction de $\mathcal{S}_{\mathrm{C}}(\texttt{ababbb})$. **(a)** L'automate juste après l'insertion des trois plus longs suffixes du mot `ababbb`. Le lien suffixe sur l'état 2 reste à définir. **(b)** Le calcul par descente rapide du lien suffixe de l'état 2 a conduit à transformer la flèche $(0, \texttt{babbb}, 1)$ en $(0, \texttt{b}, 2)$. Dans le même temps le suffixe `bbb` a été inséré. **(c)** L'insertion du suffixe suivant, `bb`, se fait par descente lente à partir de l'état 0. La flèche $(0, \texttt{b}, 2)$ n'étant pas solide, sa cible, l'état 2, a été dupliquée en $2'$ qui possède les mêmes transitions que 2. Pour terminer l'insertion du suffixe `bb`, il reste à couper la flèche $(2', \texttt{bb}, 1)$ afin de créer un état 3. Finalement, le reste de la construction équivaut à déterminer les états terminaux, et l'on obtient l'automate de la figure 5.15.

Pour les questions se rapportant aux langages formels, comme les notions de congruences syntaxiques et d'automates minimaux, on pourra se reporter aux livres de Berstel [91] et de Pin [100].

L'automate des suffixes d'un texte est aussi connu sous l'appellation de *DAWG*, pour *Directed Acyclic Word Graph*. Sa linéarité a été découverte par Blumer et co-auteurs (1983), qui en ont donné une construction linéaire (sur un alphabet fixé). La minimalité de la structure en tant qu'automate est due à Crochemore (1984), qui a montré comment construire avec la même complexité l'automate des facteurs d'un texte (voir exercices 5.9, 5.10 et 5.11).

Un algorithme de compaction de l'automate des suffixes ainsi qu'un algorithme de construction directe de l'automate compact des suffixes ont été présentés par Crochemore et Vérin (1997).

Pour l'analyse en moyenne de la taille des différentes structures présentées dans le chapitre, on peut se reporter aux articles de Blumer, Ehrenfeucht et Haussler (1989) et de Raffinot (1997), qui font appel à des méthodes décrites dans le livre de Sedgewick et Flajolet [101].

Lorsque l'alphabet est potentiellement infini, les algorithmes de construction de l'arbre compact et de l'automate des suffixes sont optimaux car ils impliquent un classement des lettres de l'alphabet. Sur des alphabets d'entiers particuliers, Farach (1997) a montré que la construction peut se faire en temps linéaire.

Par ailleurs, Allauzen, Crochemore et Raffinot (1999), ont introduit une structure réduite sous le nom d'« oracle des suffixes » et qui possède des applications voisines de celles de l'automate des suffixes.

Pour l'implantation de structures d'index en mémoire externe, on peut se référer à Ferragina et Grossi (1999).

Exercices

5.1 (Devinette)
On considère l'arbre des suffixes construit par l'algorithme ARBRE-C-SUFFIXES. Soit (p, q) un arc de l'arbre et $(i, \ell) = \mathit{étiq}(p, q)$ son étiquette. Est-ce que l'égalité $\mathit{pos}(y[i\,..\,i+\ell-1]) = i$ est toujours satisfaite ?

5.2 (Temps)
Vérifier que l'exécution de ARBRE-C-SUFFIXES(a^n) $(a \in A)$ prend un temps $\Omega(n)$. Vérifier que celle de ARBRE-C-SUFFIXES(y) se fait en temps $\Omega(n \log n)$ quand $\operatorname{card} \mathit{alph}(y) = |y| = n$.

5.3 (En particulier)
Combien y-a-t-il de nœuds dans l'arbre compact des suffixes d'un mot de de Bruijn ? et dans celui d'un mot de Fibonacci ? Même question pour leurs automates des suffixes compacts et non compacts.

5.4 (Facteur commun)
Donner un algorithme de calcul de $LCF(x, y)$ $(x, y \in A^*)$, longueur maximale des facteurs communs à x et y, connaissant l'arbre $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(x{\cdot}c{\cdot}y))$, où $c \in A$ et $c \notin \mathit{alph}(x \cdot y)$. Quelle est la complexité du calcul en temps et espace (voir une autre solution en section 6.6) ?

5.5 (Cubes)
Donner une majoration précise du nombre de cubes de mots primitifs pouvant apparaitre dans un mot de longueur n. Même question pour les carrés. [*Aide :* utiliser l'arbre compact des suffixes du mot.]

5.6 (Fusion)
Écrire un algorithme de fusion de deux arbres des suffixes, tous deux compacts, ou tous deux non compacts.

5.7 (Plusieurs mots)
Décrire un algorithme linéaire (sur un alphabet fixé) en temps et espace pour la construction de l'arbre des suffixes, arbre compact ou non, d'un ensemble fini de mots. Montrer que les mots peuvent être incorporés un à un dans la structure en construction.

5.8 (Compaction)
Décrire une version compactée de l'arbre digital de recherche $\mathcal{T}(X)$ (voir exercice 4.4). Adapter les opérations de recherche, insertion, suppression à la nouvelle structure.

5.9 (Automate des facteurs)
Soit y un mot dans lequel la dernière lettre n'apparait pas ailleurs. Montrer que $\mathcal{F}(y)$, l'automate déterministe et minimal qui reconnait les facteurs de y, possède les mêmes états et les mêmes flèches que $\mathcal{S}(y)$ (seuls les états terminaux diffèrent).

5.10 (Bornes)
Donner les bornes précises du nombre d'états et du nombre de flèches de l'automate des facteurs $\mathcal{F}(y)$.

5.11 (Construction)
Écrire un algorithme de construction séquentielle et linéaire (alphabet fini et fixé) en temps et espace pour la construction de l'automate des facteurs $\mathcal{F}(y)$.

5.12 (Autre)
Décrire un algorithme de construction de $\mathcal{S}_{\mathrm{C}}(y)$ à partir de $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$.

5.13 (Encore)
Décrire un algorithme de construction de $\mathcal{S}_{\mathrm{C}}(y)$ à partir de $\mathcal{S}(y)$.

5.14 (Programme)
Écrire en détail le code de l'algorithme de construction directe de $\mathcal{S}_{\mathrm{C}}(y)$, décrit informellement dans la section 5.5.

5.15 (Plusieurs mots, suite)
Décrire un algorithme linéaire (sur un alphabet fixé) en temps et espace pour la construction de l'automate des suffixes d'un ensemble fini de mots. [*Aide :* voir Blumer *et. al.* (1987).]

5.16 (Facteurs bornés)
Soit $A_{\mathrm{C}}(y, k, \ell)$ l'arbre compact qui accepte les facteurs du mot y qui ont une longueur comprise entre k et ℓ. Décrire un algorithme de construction de $A_{\mathrm{C}}(y, k, l)$ qui utilise un espace mémoire proportionnel à la taille de l'arbre (et non $O(|y|)$) et qui s'exécute dans le même temps asymptotique que la construction de l'arbre des suffixes de y.

6 Index

Les techniques introduites dans les deux chapitres précédents trouvent des applications immédiates pour la réalisation de l'index d'un texte. L'utilité de considérer les suffixes d'un texte pour ce genre d'application provient de la remarque évidente que tout facteur d'un mot est le préfixe d'un suffixe du texte (voir figure 6.1). On obtient ainsi une sorte d'accès direct aux facteurs d'un mot ou d'un langage, et c'est certainement le principal intérêt de ces techniques. De cette propriété découle assez directement une implantation de la notion d'index sur un texte ou sur une famille de textes, avec des algorithmes efficaces pour les opérations de base (section 6.2) telles que les questions d'appartenance, de localisation et de calcul de listes d'occurrences de motifs. La section 6.3 donne une solution sous la forme d'un transducteur. On déduit assez directement aussi des solutions aux problèmes de détection de répétitions (section 6.4) ou de mots interdits (section 6.5). La section 6.6 présente une application quelque peu détournée des techniques en question en utilisant l'index d'un motif afin d'aider à sa propre recherche. Cette méthode se prolonge de façon particulièrement efficace à la recherche des conjugués (ou rotations) d'un mot.

6.1 Implantation d'un index

Le but d'un index est de permettre de répondre à certaines questions concernant le contenu d'un texte fixe. Ce mot est noté y ($y \in A^*$) et sa longueur n ($n \in \mathbf{N}$). Un ***index*** sur y peut être considéré comme un type abstrait de données dont l'ensemble de base est celui des facteurs de y,

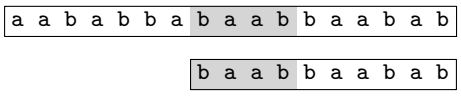


Figure 6.1 Tout facteur d'un texte est le préfixe d'un suffixe du texte.

$Fact(y)$, et qui possède des opérations donnant accès aux informations relatives à ces facteurs. La notion est analogue est celle de l'index d'un livre qui renvoie au texte à partir de mots sélectionnés. On considère plutôt ce qu'on peut appeler un index généralisé dans lequel tous les facteurs du texte sont présents. On s'intéresse à l'index sur un seul mot, mais l'extension des méthodes à un ensemble fini de mots ne pose en général pas de problème particulier.

On considère quatre opérations principales sur l'index d'un texte. Elles concernent un mot x que l'on recherche au sein de y : appartenance, position, nombre d'occurrences et liste des positions. Cette liste est en général étendue dans les applications réelles, en rapport avec la nature des données représentées par y, dans le but de conduire à des systèmes de recherche documentaire. Mais les quatre opérations considérées constituent la base technique à partir de laquelle peuvent se développer des systèmes d'interrogation plus larges.

On choisit de traiter deux méthodes principales d'implantation qui conduisent à des algorithmes efficaces sinon optimaux. La première utilise la table des suffixes du mot y, la seconde s'appuie sur une structure de données pour représenter les suffixes de y. Le choix de la structure produit des variantes de la seconde méthode. Dans cette section on rappelle pour chacune de ces implantations les éléments qui doivent être disponibles pour réaliser les opérations de l'index à partir des éléments étudiés dans les chapitres 4 et 5. Les opérations elles-mêmes sont traitées dans la section suivante.

La technique de table des suffixes (chapitre 4) est la première méthode envisagée. Elle porte de façon centrale sur une recherche par dichotomie dans les suffixes de y. Elle fournit une solution au problème de l'intervalle, qui se prolonge en une méthode de localisation de mots. Afin d'en bénéficier, il est nécessaire de classer les suffixes en ordre lexicographique et de calculer la table LPC correspondante. Bien que $\operatorname{card} Fact(y)$ soit $O(n^2)$, le tri et le calcul de LPC peuvent être réalisés en temps et espace $O(n \log n)$ (voir sections 4.4 et 4.5).

La permutation des suffixes de y qui fournit le classement lexicographique est une table notée p et définie, pour $r = 0, 1, \ldots, n-1$, par :

$p[r] = i$

si et seulement si

$y[i \,..\, n-1]$ est le r-ième plus petit suffixe non vide de y

pour l'ordre lexicographique. En d'autres termes, r est le rang du suffixe $y[i \,..\, n-1]$ dans L, liste ordonnée des suffixes non vides de y. La recherche de motifs au sein de y se base sur la remarque : les suffixes de y commençant par un même mot u sont consécutifs dans la liste L.

Les structures de données pour l'utilisation de la table des suffixes du mot y comportent :

– le mot y lui-même, stocké dans une table ;

- la table p: $\{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, n-1\}$ qui fournit les indices des suffixes dans l'ordre lexicographique croissant de ces mots ;
- la table LPC: $\{0, 1, \ldots, 2n\} \to \{0, 1, \ldots, n-1\}$ qui donne la longueur maximale des préfixes communs aux suffixes, comme indiqué dans les sections 4.3 et 4.5.

Le calcul des tables p et LPC est présenté dans les sections 4.4 et 4.5.

La seconde méthode retenue pour l'implantation d'un index s'appuie sur les structures d'automates de suffixes (chapitre 5). Ainsi, l'arbre compact des suffixes de y, $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$, fournit une base pour la réalisation d'un index. Rappelons que les structures de données nécessaires à son utilisation comprennent :

- le mot y lui-même stocké dans une table ;
- une implantation de l'automate sous forme de matrice de transitions ou d'ensemble de successeurs étiquetés pour représenter la fonction de transition δ, l'accès à l'état initial, et un marquage des états terminaux, par exemple ;
- l'attribut ls, défini sur les états, qui réalise le lien suffixe de l'arbre.

On note que le mot doit être maintenu en mémoire car l'étiquetage des flèches y fait référence (voir section 5.2). Le lien suffixe est utile pour certaines applications seulement, il peut bien sûr être éliminé lorsque les opérations implantées ne l'utilisent pas.

On peut aussi avoir recours à l'automate des suffixes de y, $\mathcal{S}(y)$, qui produit de façon naturelle un index sur les facteurs du texte. La structure comprend :

- une implantation de l'automate comme pour l'arbre ci-dessus ;
- l'attribut F qui réalise la fonction de suppléance définie sur les états ;
- l'attribut L qui indique pour chaque état la longueur maximale des mots qui permettent d'atteindre cet état.

Pour cet automate, il n'est pas nécessaire de mémoriser le mot y. Il figure dans l'automate comme étiquette du plus long chemin partant de l'état initial. Les attributs F et L peuvent être omis si non utiles pour les opérations retenues.

Enfin, la version compacte de l'automate des suffixes peut être utilisée avec profit afin de réduire encore l'espace mémoire nécessaire au stockage de la structure. Son implantation utilise de façon standard les mêmes éléments que l'automate des suffixes (en version non compacte) avec en supplément le mot y pour l'accès aux étiquettes des flèches comme pour l'arbre des suffixes. On retire un gain appréciable en espace de stockage à utiliser cette structure plutôt que les précédentes.

Dans la section qui suit, on examine plusieurs types de solutions pour la réalisation des opérations de base sur les index.

6.2 Opérations de base

On considère dans cette section quatre opérations relatives aux facteurs d'un texte y : l'appartenance (à $Fact(y)$), la première position, le nombre d'occurrences et la liste des positions. Les algorithmes sont présentés après la description globale de ces quatre opérations.

La première opération sur un index est l'appartenance d'un mot x à l'index, c'est-à-dire la question de savoir si x est un facteur de y. Cette question peut être précisée de deux manières complémentaires suivant que l'on s'attend à trouver une occurrence de x ou non dans y. Si x n'apparait pas dans y, il est souvent intéressant en pratique de connaitre le plus long début de x qui est un facteur de y. C'est le type de réponse habituelle nécessaire pour les outils de recherche séquentielle d'un éditeur de texte.

Problème de l'appartenance à l'index : étant donné $x \in A^*$, trouver le plus long préfixe de x qui appartient à $Fact(y)$.

Dans le cas contraire ($x \preceq_{fact} y$), les méthodes produisent sans grande modification la position d'une occurrence de x, et même la position de la première ou dernière occurrence de x dans y.

Problème de la position : étant donné $x \preceq_{fact} y$, trouver la position gauche de sa première (respectivement dernière) occurrence dans y.

Sachant que x est dans l'index, une autre information pertinente est constituée par son nombre d'occurrences dans y. Cette information peut orienter différemment les recherches ultérieures.

Problème du nombre d'occurrences : étant donné $x \preceq_{fact} y$, trouver combien de fois x apparait dans y.

Enfin, sous la même hypothèse que précédemment, une information complète sur la localisation de x dans y est fournie par la liste des positions de ses occurrences.

Problème de la liste des positions : étant donné $x \preceq_{fact} y$, produire la liste des positions des occurrences de x dans y.

La table des suffixes d'un mot présentée au chapitre 4 fournit une solution simple et élégante pour la réalisation des opérations ci-dessus. La table des suffixes de y est constituée par le couple de tables p et LPC comme rappelé dans la section précédente.

Proposition 6.1
Au moyen de la table des suffixes de y (couple de tables p et LPC) qui occupe un espace mémoire $O(|y|)$ on peut calculer le plus long préfixe u d'un mot $x \in A^*$ pour lequel $u \preceq_{fact} y$ en temps $O(|u| + \log |y|)$.

Lorsque $x \preceq_{fact} y$, on peut calculer la position de la première occurrence de x dans y et son nombre d'occurrences en temps $O(|x| + \log |y|)$, et produire toutes les positions des occurrences en temps supplémentaire proportionnel à leur nombre.

Preuve L'algorithme s'obtient à partir de l'algorithme INTERVALLE (section 4.5). Soit (d, f) le résultat de cet algorithme appliqué à la liste classée des suffixes de y (classement qui est fourni par p) et utilisant la table *LPC*. Par construction de l'algorithme (problème de l'intervalle et proposition 4.5), si $d+1 < f$ le mot x possède $f-d-1$ occurrences dans y ; elles sont aux positions $p[d+1], p[d+2], \ldots, p[f-1]$. En revanche, si $d+1 = f$, le mot x n'apparait pas dans y et l'on constate par simple examen de la preuve de la proposition 4.4 que le temps de la recherche est $O(|u| + \log |y|)$.

Produire les positions $p[d+1], p[d+2], \ldots, p[f-1]$ prend un temps proportionnel à leur nombre, $f-d-1$. Cela termine la preuve. ∎

On aborde ensuite les solutions obtenues en utilisant les structures de données du chapitre 5. L'espace mémoire occupé par les arbres ou automates est un peu plus important que celui qui est nécessaire à la réalisation de la table des suffixes. Il faut aussi remarquer que les structures nécessitent parfois d'être enrichies pour garantir une exécution optimale des algorithmes. Cependant, les temps d'exécution des opérations sont différents, et les structures autorisent d'autres applications qui font l'objet des sections suivantes et dont les solutions utilisent les algorithmes de base donnés ci-dessous.

Proposition 6.2
Que ce soit au moyen de $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$, $\mathcal{S}(y)$ ou $\mathcal{S}_{\mathrm{C}}(y)$, le calcul du plus long préfixe u de x qui est facteur de y ($u \preceq_{\mathit{fact}} y$) peut être réalisé en temps $O(|u| \times \log \operatorname{card} A)$ dans un espace mémoire $O(|y|)$.

Preuve Au moyen de $\mathcal{S}(y)$, afin de déterminer le mot u, il suffit de suivre un chemin d'étiquette x à partir de l'état initial de l'automate. On arrête le parcours lorsque qu'une transition manque ou lorsque x est épuisé. Cela produit le plus long préfixe de x qui est aussi préfixe de l'étiquette d'un chemin issu de l'état initial, c'est-à-dire qui apparait dans y puisque tous les facteurs de y sont des étiquettes des chemins considérés. Au total, on effectue ainsi $|u|$ transitions réussies et éventuellement une transition infructueuse (lorsque $u \prec_{\mathit{préf}} x$) à la fin du test. Comme chaque transition nécessite un temps $O(\log \operatorname{card} A)$ pour une implantation en espace $O(|y|)$ (section 1.4), on obtient un temps global $O(|u| \times \log \operatorname{card} A)$.

Le même procédé fonctionne avec $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ et $\mathcal{S}_{\mathrm{C}}(y)$. ∎

Compte tenu de la représentation des structures compactes, certaines transitions se font par simples comparaisons de lettres. Cela accélère quelque peu l'exécution de l'opération considérée même si cela ne modifie pas sa majoration asymptotique.

Position

On examine maintenant les opérations pour lesquelles il est supposé que x est facteur de y. Le test d'appartenance qui peut être réalisé séparément comme dans la proposition précédente, peut aussi être intégré aux solutions des autres problèmes qui nous intéressent ici. L'utilisation de transducteurs, qui prolongent les automates de suffixes, pour ce type de question est abordé dans la section suivante.

Trouver la position de la première occurrence de x dans y, $pos(x)$, revient à calculer sa position droite $posd(x)$ (voir section 5.3) car

$$pos(x) = posd(x) - |x| + 1 \ .$$

De plus, cela est aussi équivalent à calculer la longueur maximale des contextes droits de x dans y,

$$lc(x) = \max\{|z| : z \in x^{-1}Suff(y)\} \ ,$$

parce que

$$pos(x) = |y| - lc(x) - |x| \ .$$

De façon symétrique, trouver la position $posf(x)$ de la dernière occurrence de x dans y revient à calculer la longueur minimale $cc(x)$ de ses contextes droits car

$$posf(x) = |y| - cc(x) - |x| \ .$$

Afin de répondre rapidement aux requêtes portant sur la première ou la dernière positions des facteurs de y, les seules structures d'index ne sont pas suffisantes, du moins si on cherche à obtenir des temps d'exécution optimaux. En conséquence, on précalcule deux attributs sur les états de l'automate retenu, lesquelles représentent les fonctions lc et cc. On obtient ainsi le résultat qui suit.

Proposition 6.3
Les automates $\mathcal{A}_{\mathrm{C}}(Suff(y))$, $\mathcal{S}(y)$ et $\mathcal{S}_{\mathrm{C}}(y)$ peuvent être traités en temps $O(|y|)$ de telle sorte que la première (ou dernière) position dans y d'un mot $x \preceq_{fact} y$, ainsi que le nombre d'occurrences de x, puissent être calculés en temps $O(|x| \times \log \operatorname{card} A)$ dans un espace mémoire $O(|y|)$.

Preuve Notons M l'automate choisi, δ sa fonction de transition, F son ensemble de flèches, q_0 son état initial et T l'ensemble de ses états terminaux.

Considérons pour commencer le calcul de $pos(x)$. Le prétraitement de l'automate porte sur le calcul d'un attribut LC défini sur les états de M dans le but de représenter la fonction lc. Pour un état p et un mot $u \in A^*$ avec $p = \delta(q_0, u)$, on pose :

$$LC[p] = lc(u) \ ,$$

quantité indépendante du mot u qui permet d'atteindre l'état p (voir lemme 5.10). Cette valeur est encore la longueur maximale des chemins issus de p et d'extrémité un état terminal dans l'automate $\mathcal{S}(y)$. Pour $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ et $\mathcal{S}_{\mathrm{C}}(y)$ cette remarque vaut encore en définissant la longueur d'une flèche comme celle de son étiquette.

L'attribut LC satisfait la relation de récurrence :

$$LC[p] = \begin{cases} 0 & \text{si } \mathit{deg}(p) = 0 \ , \\ \max\{\ell + LC[q] : (p, v, q) \in F \text{ et } |v| = \ell\} & \text{sinon } . \end{cases}$$

La relation montre que le calcul des valeurs $LC[p]$, pour tous les états de l'automate M, se fait par simple parcours en profondeur du graphe de la structure. Comme son nombre de nœuds et son nombre de flèches sont linéaires (voir sections 5.2, 5.4 et 5.5) et que l'accès à la longueur de l'étiquette d'une flèche se fait en temps constant d'après la représentation décrite en section 5.2, le calcul de l'attribut prend un temps $O(|y|)$ (indépendant de l'alphabet).

Une fois le calcul de l'attribut LC effectué, le calcul de $\mathit{pos}(x)$ se fait par recherche de $p = \delta(q_0, x)$, puis calcul de $|y| - LC[p] - |x|$. On obtient alors le même temps d'exécution asymptotique que pour le problème de l'appartenance, à savoir $O(|x| \times \log \operatorname{card} A)$. Notons que si

$$\mathit{but}(q_0, x) = \delta(q_0, xw)$$

avec w non vide, la valeur de $\mathit{pos}(x)$ est alors $|y| - LC[p] - |xw|$, ce qui ne modifie pas l'évaluation asymptotique du temps d'exécution.

Le calcul de la position de la dernière occurrence de x dans y se résout de façon analogue en considérant l'attribut CC défini par :

$$CC[p] = \mathit{cc}(u) \ ,$$

avec les notations ci-dessus. La relation

$$CC[p] = \begin{cases} 0 & \text{si } p \in T \ , \\ \min\{\ell + CC[q] : (p, v, q) \in F \text{ et } |v| = \ell\} & \text{sinon } , \end{cases}$$

montre que le précalcul de CC nécessite un temps $O(|y|)$, et que le calcul de $\mathit{posf}(x)$ requiert ensuite le temps $O(|x| \times \log \operatorname{card} A)$.

Enfin, pour l'accès au nombre d'occurrences de x on précalcule un attribut NB défini par :

$$NB[p] = \operatorname{card}\{z \in A^* : \delta(p, z) \in T\} \ ,$$

qui est précisément la quantité cherchée lorsque $p = \mathit{but}(q_0, x)$. Le précalcul linéaire se déduit de la relation

$$NB[p] = \begin{cases} 1 + \sum_{(p,v,q) \in F} NB[q] & \text{si } p \in T \ , \\ \sum_{(p,v,q) \in F} NB[q] & \text{sinon } . \end{cases}$$

Ensuite, le nombre d'occurrences de x est obtenu par calcul de l'état $p = \mathit{but}(q_0, x)$ et accès à $NB[p]$, ce qui se fait dans le même temps que pour les opérations ci-dessus.

Cela termine la preuve. ∎

Nombre de facteurs

Un argument similaire au dernier élément de la preuve précédente permet un calcul efficace du nombre de facteurs de y, c'est-à-dire de la taille de $\mathit{Fact}(y)$. Pour cela on évalue la quantité $CS[p]$, pour tous les états p de l'automate, en utilisant la relation :

$$CS[p] = \begin{cases} 1 & \text{si } \mathit{deg}(p) = 0 \ , \\ 1 + \sum_{(p,v,q)\in F}(|v| - 1 + CS[q]) & \text{sinon } . \end{cases}$$

Si $p = \delta(q_0, u)$ pour un facteur u de y, $CS[p]$ est le nombre de facteurs de y commençant par u. Cela donne un calcul linéaire de $\operatorname{card} \mathit{Fact}(y) = CS[q_0]$, c'est-à-dire en temps $O(|y|)$ indépendamment de l'alphabet A, l'automate étant donné.

Liste des positions

Proposition 6.4
Au moyen, soit de l'arbre $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$, soit de l'automate $\mathcal{S}_{\mathrm{C}}(y)$, la liste L des positions dans y des occurrences d'un mot $x \preceq_{\mathit{fact}} y$ peut être calculée en temps $O(|x| \times \log \operatorname{card} A + k)$ dans un espace mémoire de $O(|y|)$, où k est le nombre d'éléments de L.

Preuve On considère l'arbre $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ dont on note q_0 l'état initial. Rappelons de la section 5.1 qu'un état q de l'arbre est un facteur de y, et que, s'il est terminal, il possède une sortie qui est la position de l'occurrence suffixe de q dans y (on a dans ce cas $q \preceq_{\mathit{suff}} y$ et $\mathit{sortie}[q] = \mathit{pos}(q) = |y| - |q|$). Les positions des occurrences de x dans y sont celles des suffixes dont x est un préfixe. On obtient donc ces positions en recherchant les états terminaux du sous-arbre de racine $p = \mathit{but}(q_0, x)$ (voir section 5.2). Le parcours de ce sous-arbre prend un temps proportionnel à sa taille ou encore à son nombre de nœuds terminaux puisque chaque nœud non terminal possède au moins deux enfants par définition de l'arbre. Enfin, le nombre de nœuds terminaux est précisément le nombre k d'éléments de la liste L.

En résumé, le calcul de L demande celui de p, puis le parcours du sous-arbre. La première phase s'exécute en temps $O(|x| \times \log \operatorname{card} A)$, la seconde en temps $O(k)$, ce qui donne le résultat annoncé pour l'utilisation de $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$.

Un raisonnement analogue vaut pour $\mathcal{S}_{\mathrm{C}}(y)$. À partir de l'état $p = \mathit{but}(q_0, x)$, on explore l'automate en profondeur en mémorisant la longueur du chemin courant (la longueur d'une flèche est celle de son étiquette). Un état terminal q auquel on accède par un chemin de longueur ℓ correspond à un suffixe de longueur ℓ qui est donc à la position $|y| - \ell$. Cette quantité est alors la position d'une occurrence de x dans y. Le parcours complet prend un temps $O(k)$ car il est équivalent au parcours du sous-arbre de $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(y))$ décrit ci-dessus. On obtient ainsi le même résultat qu'avec l'arbre compact des suffixes. ∎

Remarquons que le résultat sur le calcul des listes de positions s'obtient sans prétraitement des automates. En revanche, l'utilisation de l'automate (non compact) des suffixes de y demande un prétraitement qui consiste à créer des raccourcis pour lui superposer la structure de $\mathcal{S}_{\mathrm{C}}(y)$ si on souhaite obtenir un calcul de même complexité.

6.3 Transducteur des positions

Certaines des questions de localisation de facteurs au sein du mot y peuvent se décrire au moyen de transducteurs, c'est-à-dire d'automates dans lesquels les flèches possèdent une sortie. À titre d'exemple, la fonction *pos* se réalise au moyen du transducteur des positions de y, noté $T(y)$. La figure 6.2 en donne une illustration.

Le transducteur $T(y)$ est défini à partir de $\mathcal{S}(y)$ par ajout de sortie aux flèches et par modification des sorties associées aux états terminaux. Les flèches de $T(y)$ sont de la forme $(p, (a, s), q)$ où p et q sont des états, et (a, s) l'étiquette de la flèche. La lettre $a \in A$ en est l'***entrée*** et l'entier $s \in \mathbf{N}$ la ***sortie***. Le chemin

$$(p_0, (a_0, s_0), p_1), (p_1, (a_1, s_1), p_2), \ldots, (p_{k-1}, (a_{k-1}, s_{k-1}), p_k)$$

du transducteur a pour étiquette d'entrée le mot $a_0 a_1 \ldots a_{k-1}$, concaténation des entrées des étiquettes des flèches du chemin, et pour sortie la somme $s_0 + s_1 + \cdots + s_{k-1}$.

Le transducteur des positions $T(y)$ a pour base l'automate $\mathcal{S}(y)$. La transformation de $\mathcal{S}(y)$ en $T(y)$ se fait ainsi. Quand (p, a, q) est une flèche

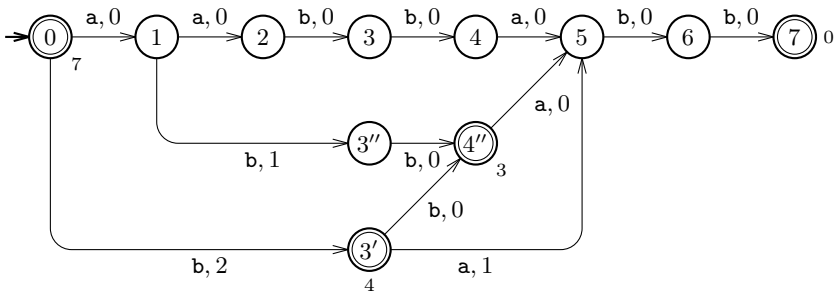


Figure 6.2 Transducteur qui réalise de façon séquentielle la fonction *pos* relative à $y =$ `aabbabb`. Chaque flèche est étiquetée par un couple (a, s), où a est l'entrée de la flèche et s sa sortie. À la lecture de `abb`, le transducteur produit l'entier 1 $(= 0 + 1 + 0)$ qui est la position de la première occurrence de `abb` dans y. L'état d'arrivée possédant la sortie 3, on en déduit que `abb` est un suffixe à la position 4 $(= 1 + 3)$ de y.

de $\mathcal{S}(y)$ elle devient la flèche $(p,(a,s),q)$ de $T(y)$ avec pour sortie :

$s = posd(q) - posd(p) - 1$

qui est aussi :

$LC[p] - LC[q] - 1$

avec la notation LC utilisée dans la preuve de la proposition 6.3. La sortie associée à un état terminal p est définie comme étant $LC[p]$.

Proposition 6.5
Soit u l'étiquette d'entrée d'un chemin issu de l'état initial dans le transducteur $T(y)$. Alors, l'étiquette de sortie du chemin est $pos(u)$.

De plus, si l'extrémité du chemin est un état terminal de sortie t, u est un suffixe de y et la position de cette occurrence de u est $pos(u) + t$.

Preuve On raisonne par récurrence sur la longueur de u. L'amorce de la récurrence, pour $u = \varepsilon$, est immédiate. Supposons $u = va$ avec $v \in A^*$ et $a \in A$. L'étiquette de sortie du chemin d'étiquette d'entrée va est $r + s$ où r et s sont respectivement les étiquettes de sortie correspondant aux entrées v et a. Par hypothèse de récurrence, on a $r = pos(v)$. Par définition des étiquettes dans $T(y)$, on a :

$s = posd(u) - posd(v) - 1$.

Donc la sortie associée à u est

$pos(v) + posd(u) - posd(v) - 1$,

soit, comme $posd(w) = pos(w) + |w| - 1$,

$pos(u) + |u| - |v| - 1$,

qui est bien $pos(u)$ comme souhaité. Cela termine la preuve de la première partie de l'énoncé.

Si l'extrémité du chemin considéré est un état terminal, sa sortie t est, par définition, $LC[u]$ qui est $|y| - posd(u) - 1$ ou encore $|y| - pos(u) - |u|$. Donc $pos(u) + t = |y| - |u|$, ce qui est bien la position du suffixe u comme annoncé. ∎

On a vu dans la preuve de la proposition 6.3 comment calculer l'attribut LC qui sert à la définition du transducteur $T(y)$. On en déduit un calcul des sorties associées aux flèches et aux états terminaux du transducteur. La transformation s'effectue donc en temps linéaire.

Proposition 6.6
Le calcul du transducteur des positions $T(y)$ à partir de l'automate des suffixes $\mathcal{S}(y)$ peut être réalisé en temps linéaire, $O(|y|)$. ∎

L'existence du transducteur des positions décrit ci-dessus montre que la position d'un facteur de y peut être calculée séquentiellement au fur et à mesure de la lecture du facteur. Le calcul a même lieu en temps réel lorsque les transitions sont effectuées en temps constant.

6.4 Répétitions

Dans cette section, on examine deux questions concernant les répétitions de facteurs au sein du texte y. Il y a deux problèmes duaux qui se résolvent efficacement par l'utilisation d'une table ou d'un automate des suffixes :

- calculer un plus long facteur répété de y ;
- trouver un plus court facteur de y qui n'y apparait qu'une seule fois.

On peut aussi généraliser la question en cherchant des facteurs qui apparaissent au moins k fois dans y, pour un entier $k > 0$ donné.

Problème de la plus longue répétition : trouver un plus long mot possédant au moins deux occurrences dans y.

La table des suffixes de y étant donnée (couple de tables p et LPC), une plus longue répétition est aussi un mot qui est le plus long préfixe commun à deux suffixes distincts. Deux de ces suffixes sont alors consécutifs dans l'ordre lexicographique comme conséquence du lemme 4.6. En se souvenant que $LPC[f] = |lpc(y[p[f-1]\,..\,n-1], y[p[f]\,..\,n-1])|$, pour $0 < f < n$, la longueur de la plus longue répétition est donc :

$$\max\{LPC[f] : f = 1, 2, \ldots, n-1\} \ .$$

Soient r cette valeur et f un indice pour lequel $LPC[f] = r$. On en déduit que :

$$y[p[f-1]\,..\,p[f-1]+r-1] = y[p[f]\,..\,p[f]+r-1] \ ,$$

et que ce mot est une plus longue répétition dans y.

Considérons maintenant l'utilisation d'un automate des suffixes, par exemple $\mathcal{S}(y)$ pour fixer les idées. Si la table NB définie dans la preuve de la proposition 6.3 est disponible, le problème de la plus longue répétition se réduit à trouver un état p de $\mathcal{S}(y)$ qui est le plus profond dans l'automate, et pour lequel $NB[p] > 1$. L'étiquette du plus long chemin de l'état initial vers p est alors une solution au problème. En fait, le problème se résout sans le recours à l'attribut NB de la façon suivante. On recherche simplement un état, le plus profond possible, satisfaisant une des deux conditions :

- au moins deux flèches partent de p ;
- une flèche part de p et p est terminal.

L'état p est alors une fourche et sa recherche peut se faire par simple parcours de l'automate. En procédant de cette façon, aucun traitement préalable de $\mathcal{S}(y)$ n'est nécessaire et on conserve néanmoins un temps de calcul linéaire. On peut noter que le temps d'exécution ne dépend pas du temps de branchement dans l'automate car aucune transition n'est effectuée, la recherche ne faisant qu'emprunter les flèches existantes.

Les deux descriptions ci-dessus sont résumées dans la proposition qui suit.

Proposition 6.7
Le calcul d'un plus long facteur répété de y peut être réalisé en temps $O(|y|)$ au moyen de la table des suffixes de y ou de l'un des automates $\mathcal{A}_{\mathrm{C}}(Suff(y))$, $\mathcal{S}(y)$ ou $\mathcal{S}_{\mathrm{C}}(y)$. ∎

Le second problème traité dans cette section est celui de la recherche d'un marqueur. Le facteur est ainsi appelé car il marque une position précise sur y.

Problème du marqueur : trouver un plus court mot possédant exactement une occurrence dans y.

Le recours à l'automate des suffixes fournit une solution au problème de la même veine que celle de la recherche d'une répétition. Il s'agit de rechercher dans l'automate un état le moins profond possible et qui est l'origine d'un seul chemin vers un état terminal. De nouveau, un simple parcours de l'automate résout la question, ce qui donne le résultat suivant.

Proposition 6.8
Au moyen de l'automate des suffixes $\mathcal{S}(y)$, le calcul d'un marqueur, plus court mot n'apparaissant qu'une seule fois dans y, peut être réalisé en temps et espace $O(|y|)$. ∎

6.5 Mots interdits

La recherche de mots interdits est complémentaire des questions de la section précédente. Elle intervient en particulier dans la description de certains algorithmes de compression de textes.

Un mot $u \in A^*$ est dit ***interdit*** dans le mot $y \in A^*$ s'il n'est pas facteur de y. Et le mot u est dit ***interdit minimal*** si, en supplément, tous ses facteurs propres sont facteurs de y. En d'autres termes, la minimalité est relative à la relation d'ordre $\preceq_{fact}$. Cette notion est en fait plus pertinente que la précédente. On note $I(y)$ l'ensemble de mots interdits minimaux dans y.

On peut remarquer que, si u est un mot de longueur k, on a :

$u \in I(y)$

si et seulement si

$u[1\,..\,k-1] \preceq_{fact} y, u[0\,..\,k-2] \preceq_{fact} y$ et $u \not\preceq_{fact} y$,

ce qui se traduit par :

$I(y) = (A \cdot Fact(y)) \cap (Fact(y) \cdot A) \cap (A^* \setminus Fact(y))$.

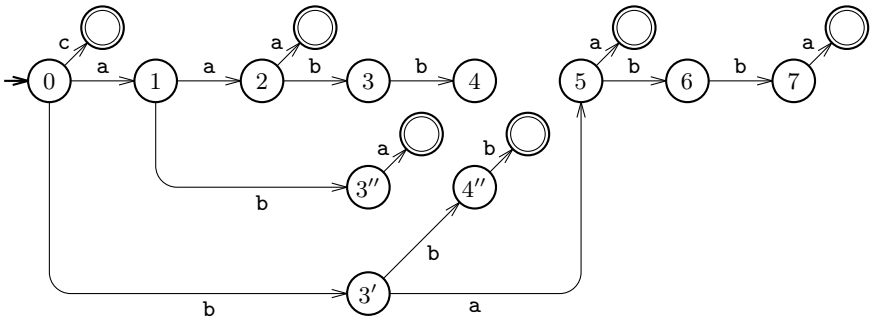


Figure 6.3 Arbre des mots interdits minimaux du mot `aabbabb` sur l'alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ tel qu'il est construit par l'algorithme INTERDITS. Les états qui ne sont pas terminaux sont ceux de l'automate $\mathcal{S}(\mathtt{aabbabb})$ de la figure 5.13. On note que les états 3 et 4 ainsi que les flèches qui y aboutissent peuvent être supprimés. Le mot interdit `babba`, reconnu par l'arbre, est minimal car `babb` et `abba` sont facteurs de `aabbabb`.

L'identité montre en particulier que le langage $I(y)$ est fini. Il est donc possible de représenter $I(y)$ au moyen d'un arbre dans lequel seuls les nœuds externes sont terminaux du fait de la minimalité des mots.

L'algorithme INTERDITS, dont le code est donné ci-dessous, construit l'arbre acceptant $I(y)$ à partir de l'automate $\mathcal{S}(y)$.

```
INTERDITS(S(y))
 1  M ← NOUVEL-AUTOMATE()
 2  L ← FILE-VIDE()
 3  ENFILER(L, (initial[S(y)], initial[M]))
 4  tantque non FILE-EST-VIDE(L) faire
 5        (p, p') ← DÉFILEMENT(L)
 6        pour chaque lettre a ∈ A faire
 7              si CIBLE(p, a) = NIL et
                  (p = initial[S(y)] ou CIBLE(F[p], a) ≠ NIL) alors
 8                    q' ← NOUVEL-ÉTAT()
 9                    terminal[q'] ← VRAI
10                    Succ[p'] ← Succ[p'] ∪ {(a, q')}
11              sinonsi CIBLE(p, a) ≠ NIL
                  et CIBLE(p, a) non encore atteint alors
12                    q' ← NOUVEL-ÉTAT()
13                    Succ[p'] ← Succ[p'] ∪ {(a, q')}
14                    ENFILER(L, (CIBLE(p, a), q'))
15  retourner M
```

Dans l'algorithme, la file L est utilisée pour parcourir l'automate $\mathcal{S}(y)$ par niveaux. La figure 6.3 présente l'exemple de l'arbre des mots interdits dans le mot `aabbabb`, obtenu à partir de l'automate de la figure 5.13.

Proposition 6.9
Pour $y \in A^*$, l'algorithme INTERDITS *produit, à partir de l'automate $\mathcal{S}(y)$, un arbre qui accepte le langage $I(y)$. L'exécution peut être réalisée en temps $O(|y| \times \log \operatorname{card} A)$.*

Preuve On remarque que les flèches créées à la ligne 13 dupliquent les flèches de l'arbre recouvrant des plus courts chemins du graphe de $\mathcal{S}(y)$, car le parcours de l'automate est effectué en ordre des niveaux croissants (la file L est utilisée dans ce but). Les autres flèches sont créées à la ligne 10 et sont de la forme (p', a, q') avec $q' \in T'$, en notant T' l'ensemble des états terminaux de M. Notons δ' la fonction de transition associée aux flèches de M calculées par l'algorithme. Par construction, le mot u pour lequel $\delta'(\mathit{initial}[M], u) = p'$ est le plus court mot qui permet d'atteindre l'état $p = \delta(\mathit{initial}[\mathcal{S}(y)], u)$ dans $\mathcal{S}(y)$.

On commence par montrer que tout mot reconnu par l'arbre que produit l'algorithme est interdit minimal. Soit ua un tel mot nécessairement non vide ($u \in A^*$, $a \in A$). Par hypothèse, la flèche (p', a, q') a été créée à la ligne 10 et $q' \in T'$. Si $u = \varepsilon$, on a $p' = \mathit{initial}[M]$ et on remarque que, par construction, $a \notin \mathit{alph}(y)$ donc ua est bien interdit minimal. Si $u \neq \varepsilon$, notons-le bv avec $b \in A$ et $v \in A^*$. On a

$$s = \delta(\mathit{initial}[\mathcal{S}(y)], v)$$

et $s \neq p$ car $|v| < |u|$ et, par construction, u est le plus court mot qui satisfait $p = \delta(\mathit{initial}[M], u)$. Donc $F[p] = s$, par définition du lien suffixe. Alors, toujours par construction, $\delta(s, a)$ est défini, ce qui implique $va \preceq_{\mathit{fact}} y$. Le mot $ua = bva$ est donc interdit minimal, car $bv, va \preceq_{\mathit{fact}} y$ et $ua \not\preceq_{\mathit{fact}} y$.

On montre ensuite que tout mot interdit est reconnu par l'arbre que construit l'algorithme. Soit ua un tel mot nécessairement non vide ($u \in \mathit{Fact}(y)$, $a \in A$). Si $u = \varepsilon$, la lettre a n'apparait pas dans y, et donc $\delta(\mathit{initial}[\mathcal{S}(y)], a)$ n'est pas défini. La condition à la ligne 7 est satisfaite et a pour effet de créer une flèche qui entraine la reconnaissance du mot ua par l'automate M. Si $u \neq \varepsilon$, notons-le bv avec $b \in A$ et $v \in A^*$. Soit

$$p = \delta(\mathit{initial}[\mathcal{S}(y)], u) \ .$$

Comme $v \prec_{\mathit{suff}} u$ et $va \preceq_{\mathit{fact}} y$ alors que $ua \not\preceq_{\mathit{fact}} y$, si l'on note

$$s = \delta(\mathit{initial}[\mathcal{S}(y)], v) \ ,$$

on a nécessairement $p \neq s$ et donc $s = F[p]$ par définition du lien suffixe. La condition à la ligne 7 est donc encore satisfaite dans ce cas et a le même effet que précédemment. En conclusion, ua est reconnu par l'arbre créé par l'algorithme, ce qui termine la preuve. ∎

On note que $y \in \{\mathtt{a}, \mathtt{b}\}^*$ possède au plus $|y|$ mots interdits minimaux (essentiellement parce que pour tout préfixe za de y, il existe au

plus un mot interdit minimal de la forme ub avec $u \preceq_{suff} z$ et $a \neq b$). Une conséquence remarquable et inattendue de l'existence de l'arbre des mots interdits, donné par la construction précédente, est une majoration du nombre de mots interdits minimaux d'un mot sur un alphabet quelconque. Si l'alphabet est réduit à deux lettres, la majoration est $|y| + 1$ essentiellement car les mots interdits sont associés aux positions sur y.

Proposition 6.10
Un mot $y \in A^*$ de longueur $|y| \geq 2$ ne possède pas plus de

$$\operatorname{card} A + (2|y| - 3) \times (\operatorname{card} alph(y) - 1)$$

mots interdits minimaux. Il en possède $\operatorname{card} A$ *si* $|y| < 2$.

Preuve D'après la proposition précédente, le nombre de mots interdits minimaux dans y est égal au nombre d'états terminaux de l'arbre reconnaissant $I(y)$, ce qui est aussi le nombre de flèches entrant dans ces états.

Il y a exactement $\operatorname{card} A - \alpha$ telles flèches sortant de l'état initial, en notant $\alpha = \operatorname{card} alph(y)$. Il y en a au plus α sortant de l'état correspondant à l'unique état de $\mathcal{S}(y)$ qui est sans successeur. Des autres états en partent au plus $\alpha - 1$. Comme, pour $|y| \geq 2$, $\mathcal{S}(y)$ possède au plus $2|y| - 1$ états (proposition 5.22), on obtient :

$$\operatorname{card} I(y) \leq (\operatorname{card} A - \alpha) + \alpha + (2|y| - 3) \times (\alpha - 1) \ ,$$

d'où :

$$\operatorname{card} I(y) \leq \operatorname{card} A + (2|y| - 3) \times (\alpha - 1) \ ,$$

comme annoncé.

Enfin, on a $I(\varepsilon) = A$ et, pour $a \in A$, $I(a) = (A \setminus \{a\}) \cup \{aa\}$. Donc $\operatorname{card} I(y) = \operatorname{card} A$ quand $|y| < 2$. ∎

6.6 Machine de recherche

Les automates de suffixes peuvent être utilisés comme des machines de recherche pour la localisation d'occurrences de motifs. On considère dans cette section l'automate $\mathcal{S}(x)$ qu'on retient afin de rechercher x dans un mot y. Les autres structures, l'arbre compact $\mathcal{A}_C(Suff(x))$ et l'automate compact $\mathcal{S}_C(x)$, peuvent être utilisées de la même manière.

L'algorithme repose sur la considération d'un transducteur représenté par fonction de suppléance (section 1.4). Le transducteur calcule séquentiellement les longueurs ℓ_i définies ci-dessous. Il est basé sur l'automate $\mathcal{S}(x)$, et la fonction de suppléance n'est rien d'autre que le lien suffixe f défini sur les états de l'automate. Le principe de fonctionnement de

la méthode de recherche est celui qui est décrit dans la section 1.4 et utilisé pour faire de la recherche de mots dans les sections 2.3 et 2.6. La recherche est donc exécutée séquentiellement le long du mot y. L'adaptation et l'analyse de l'algorithme à l'arbre $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(x))$ ne sont pas totalement immédiates car le lien suffixe de cette structure n'est pas une fonction de suppléance au sens précis de cette notion.

L'avantage qu'apporte l'algorithme sur celui de la section 2.6 réside dans un temps réduit de traitement d'une lettre de y et une analyse plus directe de la complexité du calcul. Le prix pour cette amélioration est un besoin plus important en espace mémoire destiné à stocker l'automate au lieu d'une simple table.

Longueurs des facteurs communs

La recherche du mot x est basée sur un calcul des longueurs des facteurs de x apparaissant en toute position de y. Plus précisément, l'algorithme calcule, en toute position i sur y, la longueur

$$\ell_i = \max\{|u| : u \in \mathit{Fact}(x) \cap \mathit{Suff}(y[0\,..\,i])\}$$

du plus long facteur de x se terminant à cette position. La détection des occurrences de x suit alors la remarque :

x apparait à la position $i - |x| + 1$ dans y

si et seulement si

$\ell_i = |x|$.

L'algorithme qui calcule les longueurs $\ell_0, \ell_1, \ldots, \ell_{|y|-1}$ est donné ci-dessous. Il utilise les attributs F et L définis sur les états de l'automate (section 5.4). L'attribut F sert à réinitialiser la longueur courante du facteur reconnu, après le calcul d'une cible suffixe (ligne 8). La correction de cette instruction est une conséquence du lemme 5.15.

```
LONGUEURS-DES-FACTEURS(S(x), y, n)
 1  (ℓ, p) ← (0, initial[S(x)])
 2  pour i ← 0 à n − 1 faire
 3        si CIBLE(p, y[i]) ≠ NIL alors
 4              (ℓ, p) ← (ℓ + 1, CIBLE(p, y[i]))
 5        sinon faire  p ← F[p]
 6              tantque p ≠ NIL et CIBLE(p, y[i]) = NIL
 7              si p ≠ NIL alors
 8                    (ℓ, p) ← (L[p] + 1, CIBLE(p, y[i]))
 9              sinon (ℓ, p) ← (0, initial[S(x)])
10        sortir ℓ
```

Une simulation du fonctionnement de l'algorithme est donnée dans la figure 6.4.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$y[i]$	a	a	a	b	b	b	a	b	b	a	a	b	b	a	b	b	b
ℓ_i	1	2	2	3	4	2	3	4	5	4	2	3	4	5	6	7	2
p_i	1	2	2	3	4	$4''$	5	6	7	5	2	3	4	5	6	7	$4''$

Figure 6.4 Au moyen de l'automate $\mathcal{S}$(aabbabb) (se reporter figure 5.13), l'algorithme LONGUEURS-DES-FACTEURS détermine les facteurs communs à aabbabb et y. Les valeurs ℓ_i et p_i sont les valeurs des variables ℓ et p de l'algorithme relatives à la position i. En position 8 par exemple, on a $\ell_8 = 5$, ce qui indique que le plus long facteur de aabbabb se terminant là est de longueur 5 ; c'est bbabb ; l'état courant est 7. On détecte une occurrence du motif lorsque $\ell_i = 7 = |\texttt{aabbabb}|$, comme à la position 15.

Théorème 6.11
L'algorithme LONGUEURS-DES-FACTEURS *appliqué à l'automate $\mathcal{S}(x)$ et au mot y ($x, y \in A^*$) produit les longueurs $\ell_0, \ell_1, \ldots, \ell_{|y|-1}$.*

Il effectue moins de $2|y|$ transitions dans $\mathcal{S}(x)$ et s'exécute en temps $O(|y| \times \log \operatorname{card} A)$ dans un espace $O(|x|)$.

Preuve La correction de l'algorithme se fait par récurrence sur la longueur des préfixes de y. On montre plus exactement que les égalités :

$$\ell = \ell_i$$

et :

$$p = \delta(\mathit{initial}[\mathcal{S}(x)], y[i - \ell + 1 \,..\, i])$$

sont des invariants de la boucle **pour**, en notant δ la fonction de transition de $\mathcal{S}(x)$.

Soit $i \geq 0$. Le préfixe déjà traité est de longueur i et la lettre courante est $y[i]$. On suppose que la condition est satisfaite pour $i - 1$. Ainsi, $u = y[i-\ell \,..\, i-1]$ est le plus long facteur de x se terminant à la position $i - 1$.

Soit w le suffixe de $y[0 \,..\, i]$ de longueur ℓ_i. Supposons d'abord $w \neq \varepsilon$; donc w s'écrit $v \cdot y[i]$ pour $v \in A^*$. On remarque que v ne peut être plus long que u parce que cela contredirait la définition de u, et donc v est un suffixe de u.

Si $v = u$, $\delta(p, y[i])$ est défini et fournit la prochaine valeur de p. De plus, $\ell_i = \ell + 1$. Ces deux points correspondent à la mise à jour du couple (ℓ, p) effectuée à la ligne 4, ce qui montre que la condition est satisfaite pour i dans cette situation.

Quand $v \prec_{\mathit{suff}} u$, on considère le plus grand entier k, $k > 0$, pour lequel $v \preceq_{\mathit{suff}} {s_x}^k(u)$ où s_x est la fonction suffixe relative à x (section 5.3). Le lemme 5.15 a pour conséquence que $v = {s_x}^k(u)$ et que la longueur de ce mot est $\mathit{lg}_x(q)$ où $q = \delta(\mathit{initial}[\mathcal{S}(x)], v)$. La nouvelle valeur de p est donc $\delta(q, y[i])$, et celle de ℓ est $\mathit{lg}_x(q) + 1$. C'est le résultat de l'instruction à la ligne 8 sachant que F et L implantent respectivement la

fonction suffixe et la fonction de longueur de l'automate, et d'après la proposition 5.26 qui fait le lien avec la fonction s_x.

Quand $w = \varepsilon$, cela signifie que la lettre $y[i] \notin alph(x)$. Il convient donc de réinitialiser le couple (ℓ, p), ce qui est fait à la ligne 9.

Enfin, on note que la preuve vaut aussi pour le traitement de la première lettre de y, ce qui termine la preuve de l'invariance de la condition qui prouve la correction de l'algorithme.

Pour la complexité de l'algorithme, on remarque que chaque calcul de transition, réussi ou infructueux, conduit à une incrémentation de i ou à une augmentation stricte de la valeur de $i - \ell$. Comme chacune de ces deux expressions varie de 0 à $|y|$, on en déduit que le nombre de transitions effectuées par l'algorithme n'est pas plus grand que $2|y|$. De plus, comme le temps d'exécution des transitions est représentatif du temps d'exécution total, celui-ci est $O(|y| \times \log \operatorname{card} A)$.

L'espace mémoire nécessaire au fonctionnement de l'algorithme est utilisé principalement par l'automate $\mathcal{S}(x)$ qui a une taille $O(|x|)$ d'après le théorème 5.25. Cela donne le dernier résultat énoncé et termine la preuve. ∎

L'algorithme Longueurs-des-facteurs autorise par exemple un calcul efficace de $LCF(x, y)$, la longueur maximale des facteurs communs aux mots x et y. Cette quantité intervient par exemple dans la définition de la distance, dite « distance des facteurs » :

$$d(x, y) = |x| + |y| - 2LCF(x, y) \ .$$

Corollaire 6.12
Le calcul du plus long facteur commun à deux mots x et y tels que $|x| \leq |y|$ peut être réalisé en temps $O((|x| + |y|) \times \log \operatorname{card} alph(x))$ dans un espace $O(|x|)$, ou en temps $O(|x|+|y|)$ dans un espace $O(|x| \times \operatorname{card} A)$.

Preuve On effectue le calcul en deux étapes. La première produit $\mathcal{S}(x)$, l'automate des suffixes de x. Dans la seconde, on exécute l'opération Longueurs-des-facteurs sur $\mathcal{S}(x)$ et y en ayant soin de mémoriser au cours du calcul la plus grande valeur de la variable ℓ et la position correspondante sur y. L'exécution fournit ainsi un plus long facteur commun à x et y, d'après le théorème précédent, localisé grâce à sa longueur et à sa position droite.

La complexité du calcul résulte de celle du calcul de l'automate $\mathcal{S}(x)$ (théorèmes 5.29 et 5.30) et de celle du calcul des longueurs (théorème 6.11), en notant pour ce dernier que si l'automate est implanté par matrice de transitions, il fonctionne en temps $O(|x|+|y|)$ dans un espace $O(|x| \times \operatorname{card} A)$. ∎

Optimisation du lien suffixe

Lorsqu'on cherche à calculer le délai de l'algorithme LONGUEURS-DES-FACTEURS qui travaille de façon séquentielle, on s'aperçoit rapidement qu'il est possible de modifier la fonction suffixe de façon à réduire ce délai. On suit en cela une méthode voisine à celle qui est appliquée dans la section 2.3.

L'optimisation est basée sur les ensembles de lettres, étiquettes des flèches sortant d'un état. On définit, pour p état de $\mathcal{S}(x)$, l'ensemble :

$$\mathit{Suivant}(p) = \{a : a \in A \text{ et } \delta(p, a) \text{ est défini}\} \ .$$

Alors, le nouveau lien suffixe F' est défini, pour p état de $\mathcal{S}(x)$, par la relation :

$$F'[p] = \begin{cases} F[p] & \text{si } \mathit{Suivant}(p) \subset \mathit{Suivant}(F[p]) \ , \\ F'[F[p]] & \text{sinon, si cette valeur est définie} \ . \end{cases}$$

La relation peut laisser $F'[p]$ non défini (auquel cas on peut lui donner la valeur NIL). L'idée de cette définition reprend l'optimisation réalisée sur la fonction de suppléance de l'automate-dictionnaire d'un seul mot (section 2.3).

On remarque que dans l'automate $\mathcal{S}(x)$ on a toujours :

$$\mathit{Suivant}(p) \subseteq \mathit{Suivant}(F[p]) \ .$$

On peut alors reformuler la définition de F' en :

$$F'[p] = \begin{cases} F[p] & \text{si } \mathit{deg}(p) \neq \mathit{deg}(F[p]) \ , \\ F'[F[p]] & \text{sinon, si cette valeur est définie} \ . \end{cases}$$

Le calcul de F' peut donc être réalisé en temps linéaire par simple considération des degrés sortants (deg) des états de l'automate.

L'optimisation du lien suffixe conduit à une réduction du délai de l'algorithme LONGUEURS-DES-FACTEURS. Le délai peut être évalué par le nombre d'exécutions de l'instruction à la ligne 5. On obtient le résultat suivant qui montre que l'algorithme traite les lettres de y en un temps indépendant de la longueur de x et même en temps réel quand l'alphabet est fixé.

Proposition 6.13
Pour l'algorithme LONGUEURS-DES-FACTEURS *utilisant le lien suffixe F', à la place de F, le traitement d'une lettre de y prend un temps $O(\mathrm{card}\, \mathit{alph}(x))$.*

Preuve Le résultat est une conséquence immédiate des inclusions

$$\mathit{Suivant}(p) \subset \mathit{Suivant}(F'[p]) \subseteq A$$

pour chaque état p pour lequel $F'[p]$ est défini. ∎

6.7 Recherche d'un conjugué

La suite des longueurs $\ell_0, \ell_1, \ldots, \ell_{|y|-1}$ de la section précédente est une information très riche sur les ressemblances entre les mots x et y. Elle peut être exploitée de diverses façons dans des algorithmes de comparaison de mots.

On s'intéresse ici à la recherche d'un conjugué de mot au sein d'un texte. La solution présentée dans cette section est une autre conséquence du calcul des longueurs des facteurs communs à deux mots. On rappelle qu'un conjugué du mot x est un mot de la forme $v \cdot u$ tel que $x = u \cdot v$.

Problème de la recherche d'un conjugué : soit $x \in A^*$. Localiser toutes les occurrences de conjugués de x qui apparaissent dans un mot y.

Une première solution consiste à appliquer l'algorithme de recherche d'un ensemble fini de mots (section 2.3) après avoir construit le dictionnaire des conjugués de x. Le temps de la recherche est alors proportionnel à $|y|$ (au facteur de branchement près), mais le dictionnaire peut avoir une taille quadratique, $O(|x|^2)$, comme peut l'être la taille de l'arbre (non compact) des suffixes de x.

La solution basée sur l'utilisation d'un automate des suffixes n'a pas cet inconvénient tout en conservant un temps d'exécution équivalent. La technique dérive du calcul des longueurs de la section précédente. On considère pour cela l'automate des suffixes du mot x^2, en remarquant que tout conjugué de x est facteur de x^2. On pourrait même considérer le mot $x \cdot wA^{-1}$ où w est la racine primitive de x, mais cela ne change pas le résultat qui suit.

Proposition 6.14
Soient $x, y \in A^*$. La localisation des conjugués de x dans y peut être effectuée en temps $O(|y| \times \log \operatorname{card} A)$ dans un espace $O(|x|)$.

Preuve On considère une variante de l'algorithme LONGUEURS-DES-FACTEURS qui produit les positions des occurrences des facteurs de longueur supérieure ou égale à un entier k donné. La transformation est immédiate puisqu'à chaque étape de l'algorithme la longueur du facteur courant est mémorisée dans la variable ℓ.

L'algorithme modifié est appliqué à l'automate $\mathcal{S}(x^2)$ et au mot y avec pour paramètre $k = |x|$. L'algorithme détermine donc les facteurs de longueur $|x|$ de x^2 qui apparaissent dans y. La conclusion s'en déduit en remarquant que l'ensemble des facteurs de longueur $|x|$ de x^2 est exactement l'ensemble des conjugués de x. ∎

Notes

La notion d'index est très utilisée pour la recherche documentaire. On peut se reporter au livre de Frakes et Baeza-Yates [77] ou celui de Baeza-Yates et Ribeiro-Neto [75] afin de s'initier à ce sujet, ou encore à celui de Salton [82].

Les systèmes d'indexation individuels ou ceux des robots de recherche sur la Toile utilisent souvent des techniques plus simples comme la constitution de lexiques de mots sélectionnés manuellement, de mots rares ou de k-grammes (c'est-à-dire des facteurs de longueur k) avec k relativement faible.

La plupart des sujets traités dans ce chapitre sont classiques. Le livre de Gusfield [7] contient de nombreux problèmes dont les solutions algorithmiques reposent sur l'utilisation d'une structure d'index.

La notion de répétition considérée dans la section 6.4 est voisine de celle de « facteur spécial » : un tel facteur se prolonge d'au moins deux façons différentes dans le texte. Les facteurs spéciaux interviennent en particulier dans des questions de combinatoires sur les mots.

Les mots interdits de la section 6.5 sont utilisés dans l'algorithme de compression de texte DCA (Crochemore et co-auteurs, 1999).

L'utilisation de l'automate des suffixes comme machine de recherche est due à Crochemore (1987). L'utilisation de l'arbre compact des suffixes produit une solution immédiate mais moins efficace (voir exercice 6.9).

Exercices

6.1 (Plusieurs occurrences)
Soit un entier $k > 0$. Implanter un algorithme, basé sur la table des suffixes de $y \in A^*$, qui détermine les facteurs apparaissant au moins k fois dans y.

6.2 (Idem)
Soit un entier $k > 0$. Implanter un algorithme, basé sur un des automates de suffixes de $y \in A^*$, qui détermine les facteurs apparaissant au moins k fois dans y.

6.3 (Sans chevauchement)
Pour $y \in A^*$, écrire un algorithme de calcul de la longueur maximale des facteurs de y qui possèdent deux occurrences non chevauchantes (c'est-à-dire que si u est un tel facteur, il apparait dans y à deux positions, i et j, telles que $i + |u| \leq j$).

6.4 (Marqueur)
Écrire un algorithme de calcul d'un marqueur pour $y \in A^*$, reposant sur l'utilisation de la table des suffixes de y.

6.5 (Code interdit)
Montrer que $I(y)$, $y \in A^*$, est un code (voir exercice 1.10).

6.6 (Éviter)
On dit d'un langage $M \subseteq A^*$ qu'il évite un mot $u \in A^*$ si u n'est le facteur d'aucun mot de M. Soit L le langage qui évite tous les mots d'un ensemble fini $I \subseteq A^*$. Montrer que L est reconnu par un automate. Donner un algorithme de construction d'un automate qui accepte L à partir de l'arbre des mots de I. [*Aide :* s'inspirer de l'algorithme de calcul de la fonction de suppléance donné section 2.3.]

6.7 (Automate des facteurs)
Donner une construction de l'automate $\mathcal{F}(y)$ (automate déterministe et minimal qui reconnait les facteurs de y) à partir de l'arbre des mots interdits $I(y)$. [*Aide :* voir Crochemore, Mignosi et Restivo (1998).]

6.8 (Délai)
Donner une borne précise au délai de l'algorithme LONGUEURS-DES-FACTEURS utilisant le lien suffixe non optimisé F sur l'automate des suffixes.

6.9 (Longueur des facteurs)
Décrire un algorithme, basé sur l'utilisation de l'arbre compact des suffixes, qui calcule les longueurs des facteurs communs à deux mots comme le fait l'algorithme LONGUEURS-DES-FACTEURS de la section 6.6. Analyser le temps de calcul de l'algorithme ainsi obtenu et son délai. Indiquer comment optimiser le lien suffixe et analyser la complexité de l'algorithme utilisant ce nouveau lien.

6.10 (Distance)
Montrer que la fonction d introduite section 6.6 est une distance sur A^* (la notion de distance sur les mots est définie section 7.1).

6.11 (Gros dictionnaire)
Donner une famille infinie de mots pour laquelle chaque mot possède un automate-dictionnaire de ses conjugués de taille quadratique en la longueur du mot.

6.12 (Conjugué)
Donner un algorithme de localisation des conjugués de x dans y (avec $x, y \in A^*$), connaissant l'arbre $\mathcal{A}_{\mathrm{C}}(\mathit{Suff}(x \cdot x \cdot c \cdot y))$, où $c \in A$ et $c \notin \mathit{alph}(x \cdot y)$. Quelle est la complexité du calcul en temps et en espace ?

7 Alignements

Les alignements constituent l'un des procédés utilisés pour comparer des mots. Ils permettent de visualiser la ressemblance entre les mots. Le chapitre traite de plusieurs méthodes qui effectuent la comparaison de deux mots en ce sens. L'extension des méthodes à la comparaison de plus de deux mots est délicate, conduit à des algorithmes dont le temps d'exécution est au moins exponentiel, et n'est pas traitée ici.

Les alignements sont basés sur des notions de distance ou de similarité entre les mots. Les calculs sont effectués usuellement par programmation dynamique. Un exemple type en est celui du calcul du plus long sous-mot commun à deux mots car il montre bien les techniques algorithmiques à mettre en œuvre pour obtenir un calcul efficace. En particulier, la réduction de l'espace mémoire obtenue par l'un des algorithmes constitue une stratégie qui peut souvent être appliquée dans les solutions de problèmes voisins.

Après la présentation de quelques distances définies sur les mots, des notions d'alignement et de graphe d'édition, la section 7.2 décrit les techniques de base pour le calcul de la distance d'édition (ou d'alignement) et la production des alignements associés. La méthode retenue permet de mettre en évidence une ressemblance globale entre deux mots avec des hypothèses qui simplifient le calcul. La méthode est étendue dans la section 7.4 à un problème voisin. La recherche de similarités locales entre deux mots est examinée dans la section 7.5.

La possibilité de réduction de l'espace mémoire nécessaire aux calculs est présentée dans la section 7.3 à propos du calcul de plus long sous-mots communs. Enfin, la section 7.6 présente une méthode qui est à la base d'un des logiciels les plus communément utilisés (Blast) pour la comparaison de séquences biologiques et la recherche dans les banques de séquences. Cette méthode approchée contient des heuristiques qui accélèrent le temps d'exécution dans les cas pratiques, car les autres méthodes sont souvent trop lentes pour rechercher des analogies dans les grandes banques de données.

7.1 Comparaison de mots

Dans cette section, nous introduisons les notions de distance sur les mots, d'opérations d'édition, d'alignement et de graphe d'édition.

Distance et opérations d'édition

On s'intéresse à la notion de ressemblance ou de similarité entre deux mots x et y de longueurs respectives m et n, ou de manière duale, à la distance entre ces deux mots.

On dit d'une fonction $d\colon A^* \times A^* \to \mathbf{R}$ qu'elle est une ***distance*** sur A^* si les quatre propriétés suivantes sont vérifiées pour tous $u, v \in A^*$:

Positivité : $d(u, v) \geq 0$.

Séparation : $d(u, v) = 0$ si et seulement si $u = v$.

Symétrie : $d(u, v) = d(v, u)$.

Inégalité triangulaire : $d(u, v) \leq d(u, w) + d(w, v)$ pour tout $w \in A^*$.

Plusieurs distances sur les mots peuvent être considérées à partir des factorisations des mots. Ce sont les distances préfixe, suffixe et facteur. Leur intérêt est essentiellement théorique.

Distance préfixe : définie, pour tous $u, v \in A^*$, par :
$d_{préf}(u, v) = |u| + |v| - 2 \times |lpc(u, v)|$,
où $lpc(u, v)$ est le plus long préfixe commun à u et v.

Distance suffixe : distance définie symétriquement à la distance préfixe, pour tous $u, v \in A^*$, par :
$d_{suff}(u, v) = |u| + |v| - 2 \times |lsc(u, v)|$,
où $lsc(u, v)$ est le plus long suffixe commun à u et v.

Distance facteur : distance définie de manière analogue aux deux distances précédentes (voir aussi section 6.6), pour tous $u, v \in A^*$, par :
$d_{fact}(u, v) = |u| + |v| - 2 \times LCF(u, v)$,
où $LCF(u, v)$ est la longueur maximale des facteurs communs à u et v.

La ***distance de Hamming*** fournit un moyen simple mais pas toujours pertinent pour comparer deux mots. Elle est définie pour deux mots de même longueur comme le nombre de positions en lesquelles les deux mots possèdent des lettres différentes (voir également chapitre 8).

Les distances auxquelles nous nous intéressons dans la suite sont définies à partir d'opérations qui permettent de transformer x en y. Trois types d'opérations élémentaires sont considérées. Elles sont appelées les ***opérations d'édition*** :

– la ***substitution*** d'une lettre de x à une position donnée par une lettre de y ;

Opération	Mot résultant	Cout
substituer `A` par `A`	`A C G A`	0
substituer `C` par `T`	`A T G A`	1
substituer `G` par `G`	`A T G A`	0
insérer `C`	`A T G C A`	1
insérer `T`	`A T G C T A`	1
substituer `A` par `A`	`A T G C T A`	0

Figure 7.1 Exemple illustrant la notion de distance d'édition. On montre dans le tableau ci-dessus une suite d'opérations élémentaires afin de passer du mot `ACGA` au mot `ATGCTA`. Si, pour toutes lettres $a, b \in A$, on a les couts $Sub(a, a) = 0$, $Sub(a, b) = 1$ lorsque $a \neq b$, et $Dél(a) = Ins(a) = 1$, le cout total de la suite des opérations d'édition est $0+1+0+1+1+0 = 3$. On vérifie aisément que l'on ne peut pas faire mieux avec de tels couts. Autrement dit, la distance d'édition entre les mots, $Lev(\texttt{ACGA}, \texttt{ATGCTA})$, est égale à 3.

- la ***suppression*** ou ***délétion***[1] d'une lettre de x à une position donnée ;
- l'***insertion*** d'une lettre de y dans x à une position donnée.

Un cout (de valeur entière positive) est associé à chacune des opérations. Pour $a, b \in A$, on note :

- $Sub(a, b)$ le cout de la substitution de la lettre a par la lettre b ;
- $Dél(a)$ le cout de la suppression de la lettre a ;
- $Ins(b)$ le cout de l'insertion de la lettre b.

Ce faisant, on suppose implicitement que ces couts sont indépendants des positions auxquelles les opérations sont réalisées. Une hypothèse différente est examinée en section 7.4. À partir des couts élémentaires, on pose :

$$Lev(x, y) = \min\{\text{cout de } \sigma : \sigma \in \Sigma_{x,y}\} \ ,$$

où $\Sigma_{x,y}$ est l'ensemble des suites d'opérations d'édition élémentaires permettant de transformer x en y et le cout d'un élément $\sigma \in \Sigma_{x,y}$ est la somme des couts des opérations d'édition de la suite σ. Dans la suite du chapitre, on suppose que les conditions énoncées dans la proposition qui suit sont satisfaites. La fonction Lev est alors une distance sur A^*, qui s'appelle la ***distance d'édition*** ou ***distance d'alignement***. La figure 7.1 illustre les notions qui viennent d'être introduites.

La distance de Hamming mentionnée plus haut est un cas particulier de distance d'édition pour laquelle seule l'opération de substitution est considérée. Cela revient à poser $Dél(a) = Ins(a) = +\infty$, pour chaque lettre a de l'alphabet, en se rappelant que pour cette distance, les deux mots sont supposés être de même longueur.

1. En biologie, la délétion dénote la perte d'un fragment de chromosome.

Proposition 7.1
La fonction Lev est une distance sur A^* si et seulement si Sub est une distance sur A et si $Dél(a) = Ins(a) > 0$ pour tout $a \in A$.

Preuve $\Rightarrow$: nous supposons que *Lev* est une distance et montrons les hypothèses sur les opérations élémentaires. Comme $Lev(a, b) = Sub(a, b)$ pour $a, b \in A$, on constate que *Sub* satisfait les conditions pour être une distance sur l'alphabet. Et, du fait que $Dél(a) = Lev(a, \varepsilon) = Lev(\varepsilon, a) = Ins(a)$, on a $Dél(a) = Ins(a) > 0$, pour $a \in A$, ce qui montre l'implication directe.

$\Leftarrow$: nous montrons que les quatre propriétés de positivité, de séparation, de symétrie et d'inégalité triangulaire sont satisfaites sous les hypothèses faites sur les opérations élémentaires.

Positivité. Les couts élémentaires des opérations de substitution, de suppression et d'insertion étant tous positifs, le cout de toute suite d'opérations d'édition est positif. Il s'ensuit que $Lev(u, v)$ est lui-même positif.

Séparation. Il est clair que si $u = v$, alors $Lev(u, v) = 0$, la substitution d'une lettre par elle-même ayant un cout nul car *Sub* est une distance sur A. Réciproquement, si $Lev(u, v) = 0$, alors $u = v$, puisque la seule des opérations d'édition de cout nul est la substitution d'une lettre par elle-même.

Symétrie. Comme *Sub* est symétrique et que les couts de la suppression et de l'insertion de toute lettre donnée sont identiques, la fonction *Lev* est elle aussi symétrique (la suite de cout minimal des opérations qui transforment v en u est la suite obtenue de la suite de cout minimal des opérations qui transforment u en v par lecture inverse, en changeant chaque opération de suppression en opération d'insertion et *vice versa*).

Inégalité triangulaire. Par l'absurde, supposons l'existence de $w \in A^*$ tel que $Lev(u, w) + Lev(w, v) < Lev(u, v)$. Alors la suite obtenue en concaténant les deux suites de cout minimal des opérations d'édition transformant u en w et w en v, dans cet ordre, a un cout strictement inférieur à celui de toute suite d'opérations transformant u en v, ce qui contredit la définition de $Lev(u, v)$.

Cela termine la réciproque et la preuve. ∎

Le problème du calcul de $Lev(x, y)$ consiste à déterminer une suite d'opérations d'édition pour transformer x en y et qui minimise le cout total des opérations utilisées. Calculer la distance entre x et y revient généralement aussi à maximiser une certaine notion de similarité entre ces deux mots. Toute solution, qui n'est pas nécessairement unique, peut s'énoncer comme une suite d'opérations élémentaires de substitution, suppression et insertion. Elle peut également se représenter de manière similaire sous forme d'un alignement.

Opération	Paire alignée	Cout
substituer `A` par `A`	(`A`, `A`)	0
substituer `C` par `T`	(`C`, `T`)	1
substituer `G` par `G`	(`G`, `G`)	0
insérer `C`	(`-`, `C`)	1
insérer `T`	(`-`, `T`)	1
substituer `A` par `A`	(`A`, `A`)	0

Figure 7.2 Suite de l'exemple de la figure 7.1. Les paires alignées sont indiquées dans le tableau ci-dessus. L'alignement correspondant est noté :

$$\begin{pmatrix} \texttt{A} & \texttt{C} & \texttt{G} & \texttt{-} & \texttt{-} & \texttt{A} \\ \texttt{A} & \texttt{T} & \texttt{G} & \texttt{C} & \texttt{T} & \texttt{A} \end{pmatrix} .$$

Cet alignement est optimal car son cout, $0 + 1 + 0 + 1 + 1 + 0 = 3$, est la distance d'édition entre les deux mots.

Alignements

Un ***alignement*** entre deux mots $x, y \in A^*$, dont les longueurs respectives sont m et n, est une façon de visualiser leurs similitudes. Une illustration est donnée figure 7.2. Formellement un alignement entre x et y est un mot z sur l'alphabet

$$(A \cup \{\varepsilon\}) \times (A \cup \{\varepsilon\}) \setminus \{(\varepsilon, \varepsilon)\}$$

dont la projection sur la première composante est x et la projection sur la seconde composante est y. Ainsi, si z est un alignement de longueur p entre x et y, on a :

$$\begin{aligned} z &= (\bar{x}_0, \bar{y}_0)(\bar{x}_1, \bar{y}_1) \dots (\bar{x}_{p-1}, \bar{y}_{p-1}) \ , \\ x &= \bar{x}_0 \bar{x}_1 \dots \bar{x}_{p-1} \ , \\ y &= \bar{y}_0 \bar{y}_1 \dots \bar{y}_{p-1} \ , \end{aligned}$$

avec $\bar{x}_i \in A \cup \{\varepsilon\}$ et $\bar{y}_i \in A \cup \{\varepsilon\}$ pour $i = 0, 1, \dots, p-1$. Un alignement

$$(\bar{x}_0, \bar{y}_0)(\bar{x}_1, \bar{y}_1) \dots (\bar{x}_{p-1}, \bar{y}_{p-1})$$

de longueur p se note également :

$$\begin{pmatrix} \bar{x}_0 \\ \bar{y}_0 \end{pmatrix} \begin{pmatrix} \bar{x}_1 \\ \bar{y}_1 \end{pmatrix} \dots \begin{pmatrix} \bar{x}_{p-1} \\ \bar{y}_{p-1} \end{pmatrix} \ ,$$

ou encore :

$$\begin{pmatrix} \bar{x}_0 & \bar{x}_1 & \dots & \bar{x}_{p-1} \\ \bar{y}_0 & \bar{y}_1 & \dots & \bar{y}_{p-1} \end{pmatrix} .$$

Une ***paire alignée*** du type (a, b) avec $a, b \in A$ dénote la substitution de la lettre a par la lettre b. Une paire alignée du type (a, ε) avec $a \in A$ dénote la suppression de la lettre a. Enfin, une paire alignée du type

(ε, b) avec $b \in A$ dénote l'insertion de la lettre b. Dans les alignements ou les paires alignées, le symbole « - » se substitue souvent au symbole ε, il est appelé un ***trou***.

On définit le cout d'une paire alignée par :

$$\begin{aligned} \mathit{cout}(a,b) &= \mathit{Sub}(a,b) \ , \\ \mathit{cout}(a,\varepsilon) &= \mathit{Dél}(a) \ , \\ \mathit{cout}(\varepsilon,b) &= \mathit{Ins}(b) \ , \end{aligned}$$

pour $a, b \in A$. Le cout d'un alignement est alors défini comme la somme des couts associés à chacune de ses paires alignées.

Le nombre d'alignements entre deux mots est exponentiel. La proposition qui suit précise cette quantité pour un type particulier d'alignements et donne ainsi une minoration au nombre total d'alignements.

Proposition 7.2
Soient $x, y \in A$ de longueurs respectives m et n avec $m \leq n$. Le nombre d'alignements entre x et y ne contenant pas de suppressions consécutives de lettres de x est $\binom{2n+1}{m}$.

Preuve On peut vérifier que chaque alignement du type considéré est caractérisé uniquement par l'emplacement des substitutions aux n positions sur y et à celui des suppressions entre les lettres de y. Il y a exactement $n+1$ emplacements de cette deuxième catégorie en comptant une possibilité de suppression avant $y[0]$ et une après $y[n-1]$.

L'alignement est donc caractérisé par le choix des m substitutions ou suppressions aux $2n+1$ emplacements possibles, ce qui donne le résultat annoncé. ∎

Graphe d'édition

Un alignement se traduit en termes de graphe. Pour cela, on introduit le ***graphe d'édition*** $G(x,y)$ de deux mots $x, y \in A^*$ de longueurs respectives m et n comme suit. La figure 7.3 illustre la notion.

On note Q l'ensemble des sommets de $G(x,y)$ et F son ensemble de flèches. Les flèches sont étiquetées par la fonction *étiq*, dont les valeurs sont des paires alignées, et valuées par le cout de ces paires.

L'ensemble Q des sommets est :

$$Q = \{-1, 0, \ldots, m-1\} \times \{-1, 0, \ldots, n-1\} \ ,$$

l'ensemble F des flèches est :

$$\begin{aligned} F = &\{((i-1,j-1),(i,j)) : (i,j) \in Q \text{ et } i \neq -1 \text{ et } j \neq -1\} \\ &\cup \{((i-1,j),(i,j)) : (i,j) \in Q \text{ et } i \neq -1\} \\ &\cup \{((i,j-1),(i,j)) : (i,j) \in Q \text{ et } j \neq -1\} \ , \end{aligned}$$

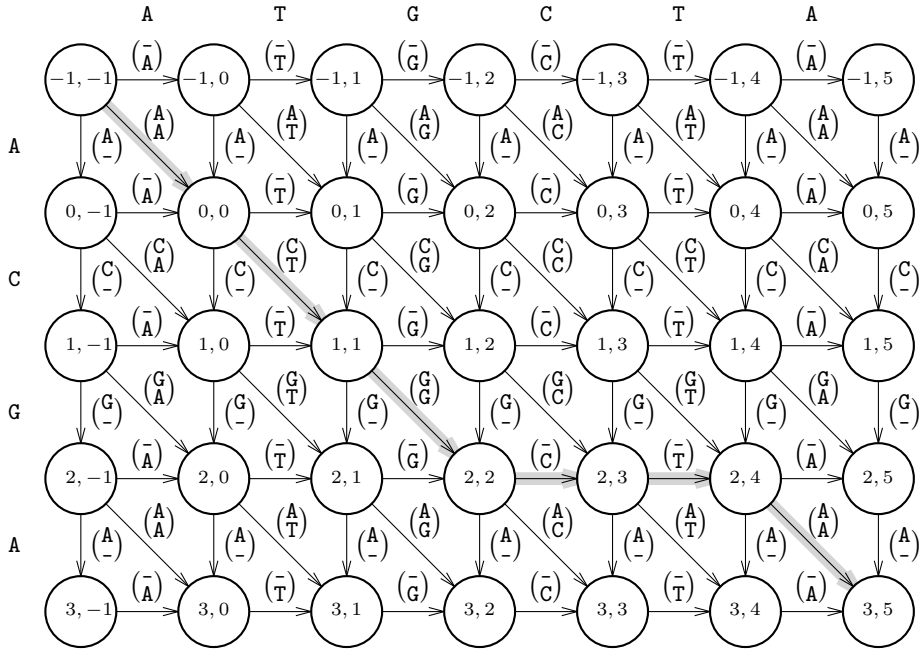


Figure 7.3 Suite de l'exemple des figures 7.1 et 7.2. On montre ici le graphe d'édition $G(\texttt{ACGA}, \texttt{ATGCTA})$ sans faire figurer les couts. Tout chemin du sommet $(-1,-1)$ au sommet $(3,5)$ correspond à un alignement entre ACGA et ATGCTA. Le chemin surligné en gris correspond à l'alignement optimal de la figure 7.2.

et la fonction :

$$\textit{étiq}\colon F \to (A \cup \{\varepsilon\}) \times (A \cup \{\varepsilon\}) \setminus \{(\varepsilon,\varepsilon)\}$$

est définie par :

$$\begin{aligned} \textit{étiq}((i-1,j-1),(i,j)) &= (x[i], y[j]) \ , \\ \textit{étiq}((i-1,j),(i,j)) &= (x[i], \varepsilon) \ , \\ \textit{étiq}((i,j-1),(i,j)) &= (\varepsilon, y[j]) \ . \end{aligned}$$

Tout chemin d'origine $(-1,-1)$ et de fin $(m-1,n-1)$ est étiqueté par un alignement entre x et y. Ainsi, en choisissant $(-1,-1)$ pour état initial et $(m-1,n-1)$ pour état terminal, le graphe d'édition $G(x,y)$ devient un automate qui reconnait tous les alignements entre x et y. Le cout d'une flèche f de $G(x,y)$ est celui de son étiquette, c'est-à-dire, $\textit{cout}(\textit{étiq}(f))$.

Le calcul d'un alignement optimal ou celui de $\textit{Lev}(x,y)$ se ramène, sur le graphe $G(x,y)$, à la recherche d'un chemin de cout minimal d'origine $(-1,-1)$ et de fin $(m-1,n-1)$. Les chemins de cout minimal de $(-1,-1)$ à $(m-1,n-1)$ sont en bijection avec les alignements optimaux entre x et y. Puisque le graphe $G(x,y)$ est acyclique, il est possible de trouver un chemin de cout minimal en considérant une et une seule fois chaque

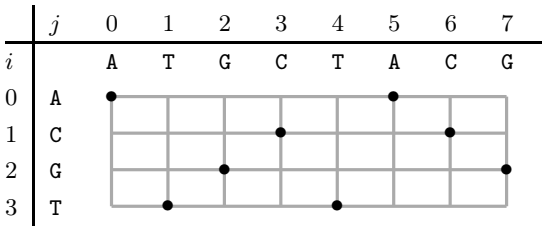


Figure 7.4 Visualisation du tracé entre x = ACGT et y = ATGCTACG. Un jeton (noir) apparait en (i, j) si et seulement si $x[i] = y[j]$. On voit apparaitre des diagonales de jetons qui signalent des similarités. Ainsi la diagonale $\langle(0,5),(1,6),(2,7)\rangle$ indique que le préfixe ACG de x est un suffixe de y, alors que l'antidiagonale $\langle(3,1),(2,2),(1,3)\rangle$ montre que le renversé du facteur CGT de x est un facteur de y.

sommet. Il suffit pour cela de les considérer suivant un ordre topologique. Un tel ordre peut être obtenu en considérant les sommets ligne par ligne de haut en bas et de gauche à droite à l'intérieur d'une ligne. Il est également possible d'obtenir le résultat en considérant les sommets, colonne par colonne, de gauche à droite, et de haut en bas, à l'intérieur d'une colonne, ou encore en les parcourant suivant les antidiagonales par exemple. Le problème se résout par programmation dynamique comme expliqué dans la section suivante.

Tracé

Il existe une méthode très simple pour mettre en évidence les similitudes entre deux mots x et y de longueurs respectives m et n. On définit pour cela une table Tr de taille $m \times n$, appelée le ***tracé*** entre x et y. Les valeurs de la table Tr sont définies pour toute position i sur x et toute position j sur y par :

$$Tr[i,j] = \begin{cases} \text{VRAI} & \text{si } x[i] = y[j] \ , \\ \text{FAUX} & \text{sinon} \ . \end{cases}$$

Pour visualiser le tracé, on dispose des jetons sur une grille pour signifier la valeur VRAI (un exemple est donné figure 7.4). Les zones de similitudes entre les deux mots apparaissent alors comme des suites de jetons sur les diagonales de la grille.

Il est possible de déduire d'un tracé (confondu avec sa visualisation) un alignement global entre les deux mots en joignant des suites de jetons. Des jonctions diagonales correspondent à des substitutions, des jonctions horizontales correspondent à des insertions et des jonctions verticales correspondent à des suppressions. Les alignements globaux correspondent alors à des chemins commençant à proximité du coin supérieur gauche et se terminant à proximité du coin inférieur droit.

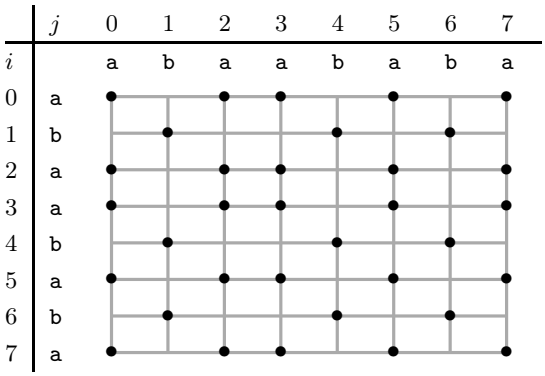


Figure 7.5 Visualisation du tracé du mot `abaababa` avec lui-même. Entre autres éléments apparaissent les bords du mot : ce sont les diagonales de jetons partant du haut de la grille (en dehors de la diagonale principale). On distingue les bords non vides `a` et `aba`. Les antidiagonales centrées sur la diagonale principale indiquent des facteurs de x qui sont des palindromes : l'antidiagonale $\langle(7,3),(6,4),(5,5),(4,6),(3,7)\rangle$ correspond au palindrome `ababa`.

Il est à noter que lorsque l'on utilise cette technique avec $x = y$, les bords de x apparaissent comme des diagonales de jetons commençant et se terminant sur les bordures de la grille. La figure 7.5 illustre cela.

7.2 Alignement optimal

Dans cette section, on présente la méthode de base pour le calcul d'un alignement optimal entre deux mots. Le procédé utilise très simplement une technique dite de programmation dynamique. Elle consiste à mémoriser les résultats de calculs intermédiaires pour éviter d'avoir à les recalculer. La production d'un alignement entre deux mots x et y est basée sur le calcul de la distance d'édition entre ces deux mots. On commence donc par expliquer comment effectuer ce calcul. On décrit ensuite comment déterminer les alignements optimaux associés.

Calcul de la distance d'édition

À partir des deux mots $x, y \in A^*$ de longueurs respectives m et n, on définit la table T à $m+1$ lignes et $n+1$ colonnes par :

$$T[i,j] = Lev(x[0\,..\,i], y[0\,..\,j])$$

pour $i = -1, 0, \ldots, m-1$ et $j = -1, 0, \ldots, n-1$. Ainsi, $T[i,j]$ est-il aussi le cout minimal d'un chemin de $(-1,-1)$ à (i,j) dans le graphe d'édition $G(x,y)$.

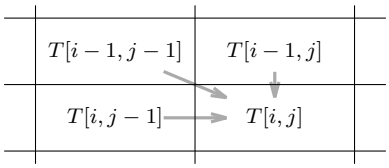


Figure 7.6 La valeur $T[i,j]$ ne dépend que des valeurs aux trois positions voisines : $T[i-1,j-1]$, $T[i-1,j]$ et $T[i,j-1]$ (lorsque $i,j \geq 0$).

Pour calculer $T[i,j]$, on utilise la formule de récurrence énoncée dans la proposition qui suit et dont la preuve est donnée plus loin.

Proposition 7.3
Pour $i = 0,1,\ldots,m-1$ et $j = 0,1,\ldots,n-1$, on a :

$$\begin{aligned}
T[-1,-1] &= 0 \ , \\
T[i,-1] &= T[i-1,-1] + \mathit{Dél}(x[i]) \ , \\
T[-1,j] &= T[-1,j-1] + \mathit{Ins}(y[j]) \ , \\
T[i,j] &= \min \begin{cases} T[i-1,j-1] + \mathit{Sub}(x[i],y[j]) \ , \\ T[i-1,j] + \mathit{Dél}(x[i]) \ , \\ T[i,j-1] + \mathit{Ins}(y[j]) \ . \end{cases}
\end{aligned}$$

La valeur à la position $[i,j]$ dans la table T, avec $i,j \geq 0$, ne dépend ainsi que des valeurs aux positions $[i-1,j-1]$, $[i-1,j]$ et $[i,j-1]$ (voir figure 7.6). Une illustration du calcul est présentée figure 7.7.

L'algorithme Calcul-générique, dont le code est donné ci-dessous, effectue le calcul de la distance d'édition en utilisant la table T, la valeur cherchée étant $T[m-1,n-1] = \mathit{Lev}(x,y)$ (corollaire 7.5).

```
Calcul-générique(x, m, y, n)
1  T[−1, −1] ← 0
2  pour i ← 0 à m − 1 faire
3          T[i, −1] ← T[i − 1, −1] + Dél(x[i])
4  pour j ← 0 à n − 1 faire
5          T[−1, j] ← T[−1, j − 1] + Ins(y[j])
6          pour i ← 0 à m − 1 faire
7                  T[i, j] ← min { T[i − 1, j − 1] + Sub(x[i], y[j])
                                 { T[i − 1, j] + Dél(x[i])
                                 { T[i, j − 1] + Ins(y[j])
8  retourner T[m − 1, n − 1]
```

Nous allons maintenant prouver la validité du procédé de calcul en énonçant d'abord un résultat intermédiaire.

(a)

Sub	A	C	D	E	G	K	L	P	Q	R	W	Y
A	0	3	3	3	3	3	3	3	3	3	3	3
C	3	0	3	3	3	3	3	3	3	3	3	3
D	3	3	0	3	3	3	3	3	3	3	3	3
E	3	3	3	0	3	3	3	3	3	3	3	3
G	3	3	3	3	0	3	3	3	3	3	3	3
K	3	3	3	3	3	0	3	3	3	3	3	3
L	3	3	3	3	3	3	0	3	3	3	3	3
P	3	3	3	3	3	3	3	0	3	3	3	3
Q	3	3	3	3	3	3	3	3	0	3	3	3
R	3	3	3	3	3	3	3	3	3	0	3	3
W	3	3	3	3	3	3	3	3	3	3	0	3
Y	3	3	3	3	3	3	3	3	3	3	3	0

	Dél	*Ins*
A	1	1
C	1	1
D	1	1
E	1	1
G	1	1
K	1	1
L	1	1
P	1	1
Q	1	1
R	1	1
W	1	1
Y	1	1

(b)

T	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	E	R	D	A	W	C	Q	P	G	K	W	Y
-1	$x[i]$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	E	1	0	1	2	3	4	5	6	7	8	9	10	11
1	A	2	1	2	3	2	3	4	5	6	7	8	9	10
2	W	3	2	3	4	3	2	3	4	5	6	7	8	9
3	A	4	3	4	5	4	3	4	5	6	7	8	9	10
4	C	5	4	5	6	5	4	3	4	5	6	7	8	9
5	Q	6	5	6	7	6	5	4	3	4	5	6	7	8
6	G	7	6	7	8	7	6	5	4	5	4	5	6	7
7	K	8	7	8	9	8	7	6	5	6	5	4	5	6
8	L	9	8	9	10	9	8	7	6	7	6	5	6	7

(c)

```
E - - A W A C Q - G K - - L
E R D A W - C Q P G K W Y -

E - - A W A C Q - G K - L -
E R D A W - C Q P G K W - Y

E - - A W A C Q - G K L - -
E R D A W - C Q P G K - W Y
```

Figure 7.7 Calcul de la distance d'édition entre les mots `EAWACQGKL` et `ERDAWCQPGKWY` et alignements correspondants. **(a)** Matrice d'édition : valeurs des couts des opérations d'édition avec ici $Sub(a, b) = 3$ pour $a \neq b$ et $Dél(a) = Ins(a) = 1$. **(b)** Table T, calculée par l'algorithme CALCUL-GÉNÉRIQUE. On obtient $Lev(\texttt{EAWACQGKL}, \texttt{ERDAWCQPGKWY}) = T[8, 11] = 7$. Sont également figurés les chemins de cout minimal entre les positions $[-1, -1]$ et $[8, 11]$ sur la table. Ceux-ci peuvent être calculés par l'algorithme LES-ALIGNEMENTS ; ils sont au nombre de trois. **(c)** Les trois alignements optimaux associés. On remarque qu'ils font apparaitre le sous-mot `EAWCQGK` commun aux deux mots qui est en fait de longueur maximale. On constate en plus que la distance ci-dessus est aussi $|\texttt{EAWACQGKL}| + |\texttt{ERDAWCQPGKWY}| - 2 \times |\texttt{EAWCQGK}| = 7$ (voir section 7.3).

Lemme 7.4
Pour tous $a, b \in A$, $u, v \in A^*$, on a :

$$\begin{aligned} Lev(ua, \varepsilon) &= Lev(u, \varepsilon) + Dél(a) \ , \\ Lev(\varepsilon, vb) &= Lev(\varepsilon, v) + Ins(b) \ , \\ Lev(ua, vb) &= \min \begin{cases} Lev(u, v) + Sub(a, b) \ , \\ Lev(u, vb) + Dél(a) \ , \\ Lev(ua, v) + Ins(b) \ . \end{cases} \end{aligned}$$

Preuve La suite des opérations d'édition qui transforment le mot ua en le mot vide se termine nécessairement par la suppression de la lettre a. Le reste de la suite transforme le mot u en le mot vide. On a donc :

$$\begin{aligned} Lev(ua, \varepsilon) &= \min\{\text{cout de } \sigma : \sigma \in \Sigma_{ua,\varepsilon}\} \\ &= \min\{\text{cout de } \sigma' \cdot (a, \varepsilon) : \sigma' \in \Sigma_{u,\varepsilon}\} \\ &= \min\{\text{cout de } \sigma' : \sigma' \in \Sigma_{u,\varepsilon}\} + Dél(a) \\ &= Lev(u, \varepsilon) + Dél(a) \ . \end{aligned}$$

D'où la première identité. La validité de la deuxième identité s'établit selon le même schéma. Pour la troisième, il suffit de distinguer les cas où la dernière opération d'édition est une substitution, une suppression ou une insertion. ∎

Preuve de la proposition 7.3 C'est une conséquence directe de l'égalité $Lev(\varepsilon, \varepsilon) = 0$ et du lemme 7.4 en posant $a = x[i]$, $b = y[j]$, $u = x[0\,..\,i-1]$ et $v = y[0\,..\,j-1]$. ∎

Corollaire 7.5
L'algorithme Calcul-générique *produit la distance d'édition entre x et y.*

Preuve C'est une conséquence de la proposition 7.3 : le calcul effectué par l'algorithme applique la relation de récurrence énoncée. ∎

Alors que la programmation directe de la formule de récurrence de la proposition 7.3 mène à un algorithme de complexité exponentielle, on voit immédiatement que le temps d'exécution de l'opération Calcul-générique(x, m, y, n) est quadratique.

Proposition 7.6
L'algorithme Calcul-générique*, appliqué à deux mots de longueur m et n, s'exécute en temps $O(m \times n)$ dans un espace $O(\min\{m, n\})$.*

Preuve Le calcul de la valeur de chaque position de la table T ne dépend que des trois positions voisines et ce calcul s'exécute en temps constant. Il y a $m \times n$ valeurs dans la table T calculées de cette façon, après une initialisation en temps $O(m+n)$, ce qui donne le résultat sur le temps d'exécution. Pour l'espace, il suffit de remarquer que seules deux lignes ou deux colonnes de la table T suffisent pour réaliser le calcul. ∎

On obtient un résultat analogue à l'énoncé de la proposition en effectuant le calcul des valeurs de la table T selon les antidiagonales. Il suffit dans ce cas de ne mémoriser que trois antidiagonales consécutives pour mener à bien le calcul.

Calcul d'un alignement optimal

L'algorithme CALCUL-GÉNÉRIQUE ne calcule que le cout de la transformation de x en y. Afin d'obtenir une suite d'opérations d'édition qui transforme x en y, ou l'alignement correspondant, on peut effectuer le calcul en « remontant » dans la table T depuis la position $[m-1, n-1]$ jusqu'à la position $[-1, -1]$. À partir d'une position $[i, j]$, on visite, parmi les trois positions voisines $[i-1, j-1]$, $[i-1, j]$ et $[i, j-1]$, l'une de celles dont la valeur associée a produit celle de $T[i, j]$. L'algorithme UNALIGNEMENT, dont le code est donné plus loin, implante cette méthode qui produit un alignement optimal.

La validité du procédé peut s'expliquer au moyen de la notion de ***flèche active*** dans le graphe d'édition $G(x, y)$. Ce sont les flèches qui sont à considérer pour obtenir un alignement optimal. En reprenant l'exemple des figures 7.1 et 7.2, l'algorithme CALCUL-GÉNÉRIQUE calcule la table T qui est donnée dans la figure 7.8. Le graphe d'édition associé est celui de la figure 7.3 et la figure 7.9 en montre le sous-graphe des flèches actives qui se déduit de la table. Formellement, on dit que la flèche $((i', j'), (i, j))$ d'étiquette (a, b) est active lorsque :

$$\begin{array}{lcl} T[i,j] & = & T[i',j'] + \mathit{Sub}(a,b) \text{ si } i - i' = j - j' = 1 \ , \\ T[i,j] & = & T[i',j'] + \mathit{Dél}(a) \text{ si } i - i' = 1 \text{ et } j = j' \ , \\ T[i,j] & = & T[i',j'] + \mathit{Ins}(b) \text{ si } i = i' \text{ et } j - j' = 1 \ , \end{array}$$

avec $i, i' \in \{-1, 0, \ldots, m-1\}$, $j, j' \in \{-1, 0, \ldots, n-1\}$, $a, b \in A$.

Lemme 7.7
L'étiquette d'un chemin (non réduit à un seul sommet) de $G(x, y)$ joignant (k, ℓ) à (i, j) est un alignement optimal entre $x[k\,..\,i]$ et $y[\ell\,..\,j]$ si et seulement si toutes ses flèches sont actives. On a :

$$\mathit{Lev}(x[k\,..\,i], y[\ell\,..\,j]) = T[i,j] - T[k,\ell] \ .$$

Preuve On note que l'alignement est optimal, par définition, si le cout du chemin est minimal. De plus, on a dans ce cas :

$$\mathit{Lev}(x[k\,..\,i], y[\ell\,..\,j]) = T[i,j] - T[k,\ell] \ .$$

Montrons l'équivalence par récurrence sur la longueur strictement positive du chemin (comptée en nombre de flèches). Soit (i', j') le sommet qui précède (i, j) sur le chemin.

(a)

T	j	-1	0	1	2	3	4	5
i		$y[j]$	A	T	G	C	T	A
-1	$x[i]$	0	1	2	3	4	5	6
0	A	1	0	1	2	3	4	5
1	C	2	1	1	2	2	3	4
2	G	3	2	2	1	2	3	4
3	A	4	3	3	2	2	3	3

(b) $\begin{pmatrix} \texttt{A} & \texttt{-} & \texttt{-} & \texttt{C} & \texttt{G} & \texttt{A} \\ \texttt{A} & \texttt{T} & \texttt{G} & \texttt{C} & \texttt{T} & \texttt{A} \end{pmatrix}$ $\begin{pmatrix} \texttt{A} & \texttt{C} & \texttt{G} & \texttt{-} & \texttt{-} & \texttt{A} \\ \texttt{A} & \texttt{T} & \texttt{G} & \texttt{C} & \texttt{T} & \texttt{A} \end{pmatrix}$

Figure 7.8 Suite de l'exemple de la figure 7.3. Calcul de la distance d'édition entre les deux mots ACGA et ATGCTA et alignements correspondants. **(a)** La table T, comme calculée lors de l'exécution l'algorithme Calcul-générique avec les couts élémentaires $Sub(a, b) = 1$ pour $a \neq b$ et $Dél(a) = Ins(a) = 1$. On obtient $Lev(\texttt{ACGA}, \texttt{ATGCTA}) = T[3, 5] = 3$. Figurent également les chemins de cout minimal entre les positions $[-1, -1]$ et $[3, 5]$. **(b)** Les deux alignements optimaux associés.

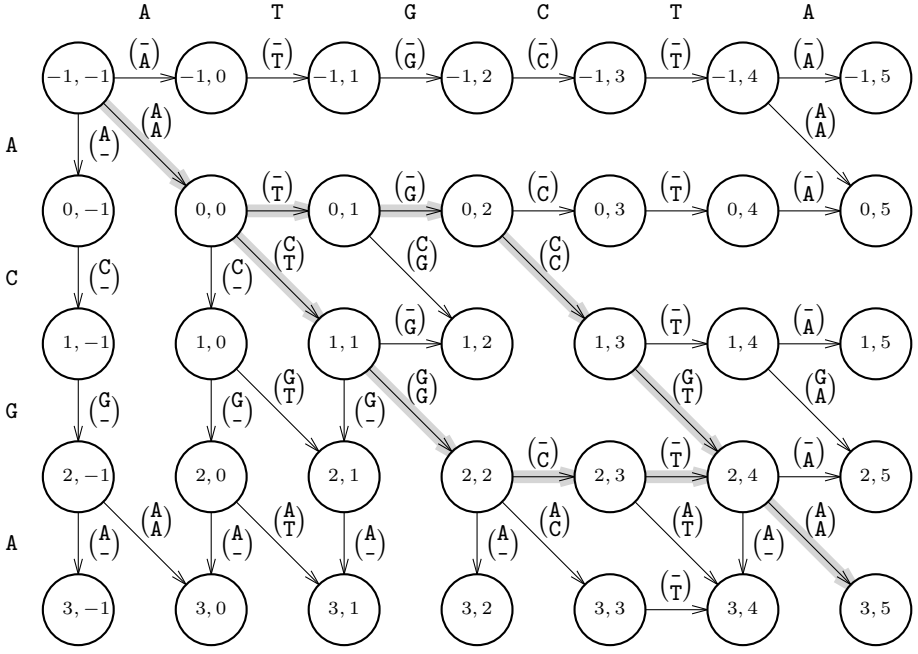


Figure 7.9 Flèches actives du graphe d'édition de la figure 7.3. Les chemins surlignés en gris joignent les sommets $(-1, -1)$ et $(3, 5)$; ils correspondent aux alignements optimaux (voir figure 7.8). Les flèches de ces chemins constituent les flèches de l'automate des alignements optimaux.

Supposons d'abord le chemin de longueur 1, soit $(k,\ell) = (i',j')$. Si le cout du chemin est minimal, sa valeur est :

$$Lev(x[k\,..\,i], y[\ell\,..\,j]) = T[i,j] - T[k,\ell] \ ,$$

et comme celle-ci est aussi $Sub(x[i], y[j])$, $Dél(x[i])$ ou $Ins(y[j])$ suivant le cas considéré, on en déduit que la flèche est active, par définition. Réciproquement, si la flèche du chemin est active on a par définition, ou $T[i,j] - T[k,\ell] = Sub(x[i], y[j])$, ou $T[i,j] - T[k,\ell] = Dél(x[i])$, ou $T[i,j] - T[k,\ell] = Ins(y[j])$, suivant le cas considéré. Mais ces valeurs sont aussi la distance entre les deux mots qui sont de longueur inférieure à 1. Donc le chemin est de cout minimal.

Supposons ensuite le chemin de longueur strictement supérieure à 1.

Si le chemin est de cout minimal, il en est de même de sa portion joignant (k,ℓ) à (i',j') et de la flèche $((i',j'),(i,j))$. L'hypothèse de récurrence appliquée à la première portion indique alors qu'elle est constituée de flèches actives. La minimalité du cout de la dernière flèche revient aussi à dire qu'elle est active (voir proposition 7.3).

Réciproquement, supposons que les flèches du chemin soient toutes actives. En appliquant l'hypothèse de récurrence à la portion du chemin joignant (k,ℓ) à (i',j'), on déduit que celle-ci est de cout minimal et

$$T[i',j'] - T[k,\ell] = Lev(x[k\,..\,i'], y[\ell\,..\,j']) \ .$$

Comme la dernière flèche est active, son cout est minimal et vaut $T[i,j] - T[i',j']$ d'après la proposition 7.3. Le chemin complet est donc de cout minimal :

$$\begin{aligned} Lev(x[k\,..\,i], y[\ell\,..\,j]) &= (T[i,j] - T[i',j']) + (T[i',j'] - T[k,\ell]) \\ &= T[i,j] - T[k,\ell] \ . \end{aligned}$$

Cela termine la preuve. ∎

On note qu'en tout sommet du graphe d'édition, hormis en $(-1,-1)$, entre au moins une flèche active d'après la relation de récurrence satisfaite par la table T (proposition 7.3). Le travail effectué par l'algorithme Un-alignement consiste ainsi à remonter le long des flèches actives, ce qui ne s'arrête que lorsque le sommet $(-1,-1)$ est atteint. On considère que la variable z de l'algorithme est un mot sur l'alphabet $(A \cup \{\varepsilon\}) \times (A \cup \{\varepsilon\})$, et que, sur cet alphabet, la concaténation a lieu composante par composante.

```
UN-ALIGNEMENT(x, m, y, n)
 1  z ← (ε, ε)
 2  (i, j) ← (m − 1, n − 1)
 3  tantque i ≠ −1 et j ≠ −1 faire
 4        si T[i, j] = T[i − 1, j − 1] + Sub(x[i], y[j]) alors
 5              z ← (x[i], y[j]) · z
 6              (i, j) ← (i − 1, j − 1)
 7        sinonsi T[i, j] = T[i − 1, j] + Dél(x[i]) alors
 8              z ← (x[i], ε) · z
 9              i ← i − 1
10        sinon z ← (ε, y[j]) · z
11              j ← j − 1
12  tantque i ≠ −1 faire
13        z ← (x[i], ε) · z
14        i ← i − 1
15  tantque j ≠ −1 faire
16        z ← (ε, y[j]) · z
17        j ← j − 1
18  retourner z
```

Proposition 7.8
L'algorithme UN-ALIGNEMENT *produit un alignement optimal entre x et y, c'est-à-dire un alignement de cout $Lev(x, y)$. Le calcul s'exécute en temps et espace supplémentaire $O(m + n)$.*

Preuve La preuve formelle repose sur le lemme 7.7. On constate que les conditions aux lignes 4 et 7 permettent de tester l'activité des flèches du graphe d'édition associé au calcul. Le troisième cas traité aux lignes 10–11 correspond à la troisième condition de la définition d'une flèche active, car il en entre toujours au moins une sur chaque sommet différent de $(-1, -1)$ du graphe. Le calcul complet produit donc l'étiquette d'un chemin d'origine $(-1, -1)$ et de fin $(m - 1, n - 1)$ constitué uniquement de flèches actives. D'après le lemme 7.7, cette étiquette est un alignement optimal entre x et y.

Chaque opération significative du temps d'exécution de l'algorithme conduit à diminuer la valeur de i ou celle de j qui varient respectivement de $m - 1$ et $n - 1$ jusqu'à -1. Cela donne le temps $O(m + n)$. L'espace supplémentaire est utilisé pour mémoriser le mot z qui est de longueur maximale $m + n$. Le problème se résout donc en espace $O(m + n)$. ∎

On note que les tests de validité des trois flèches entrant dans le sommet (i, j) du graphe d'édition peuvent être effectués dans un ordre quelconque. Il existe donc $3! = 6$ écritures possibles des lignes 4–11. Celle qui est présentée privilégie un chemin contenant des flèches diagonales. À titre d'exemple, on obtient le chemin le plus haut (relativement au dessin du graphe d'édition comme en figure 7.9) en échangeant les lignes

4–6 avec les lignes 7–9. On peut aussi programmer le calcul de façon à obtenir un alignement aléatoire parmi les alignements optimaux.

Pour calculer un alignement, il est également possible de mémoriser les flèches actives sous forme de « flèches de retour » dans une table supplémentaire lors du calcul des valeurs de la table T. Le calcul d'un alignement revient alors à suivre ces flèches depuis la position $[m-1, n-1]$ jusqu'à la position $[-1, -1]$ dans la table des flèches. Cela nécessite un espace $O(m \times n)$ comme celui occupé par la table T. Il convient de remarquer qu'il suffit de mémoriser, pour chaque position, une direction de retour parmi trois ; ce qui se code sur deux bits.

Le procédé retenu dans cette section afin de calculer un alignement optimal fait appel à la table T et nécessite donc un espace quadratique. Il est toutefois possible de trouver un alignement optimal en espace linéaire en utilisant une méthode « diviser pour régner » décrite section 7.3.

Calcul de tous les alignements optimaux

Si tous les alignements optimaux entre x et y doivent être exhibés, on peut faire appel à l'algorithme LES-ALIGNEMENTS dont le code est donné ci-après. Elle fait elle-même appel à la procédure récursive LA, dont le code est donné à la suite, pour laquelle les variables x, y et T sont globales. Le principe de fonctionnement est identique à l'algorithme précédent, basé sur la notion de flèche active.

```
LES-ALIGNEMENTS(x, m, y, n)
  1  LA(m − 1, n − 1, (ε, ε))

LA(i, j, z)
  1  si i ≠ −1 et j ≠ −1
       et T[i, j] = T[i − 1, j − 1] + Sub(x[i], y[j]) alors
  2        LA(i − 1, j − 1, (x[i], y[j]) · z)
  3  si i ≠ −1
       et T[i, j] = T[i − 1, j] + Dél(x[i]) alors
  4        LA(i − 1, j, (x[i], ε) · z)
  5  si j ≠ −1
       et T[i, j] = T[i, j − 1] + Ins(y[j]) alors
  6        LA(i, j − 1, (ε, y[j]) · z)
  7  si i = −1 et j = −1 alors
  8        signaler que z est un alignement
```

Proposition 7.9
L'algorithme LES-ALIGNEMENTS *produit tous les alignements optimaux entre ses mots d'entrée. Son temps d'exécution est proportionnel à la somme des longueurs de tous les alignements produits.*

Preuve On constate que les tests aux lignes 1, 3 et 5 sont utilisés pour tester l'activité de la flèche concernée. Le test à la ligne 7 permet de

produire l'alignement courant lorsque celui est complet. Le reste de la preuve est semblable à celle de l'algorithme UN-ALIGNEMENT.

Le temps d'exécution de chaque test est constant, et celui-ci conduit à augmenter d'une paire l'alignement courant. D'où le résultat sur le temps d'exécution total. ∎

La mémorisation de flèches de retour évoquée ci-dessus peut aussi servir au calcul de tous les alignements. Il est néanmoins nécessaire ici de mémoriser trois flèches au plus par position, ce qui se code sur trois bits.

Produire tous les alignements n'est pas judicieux si ceux-ci sont trop nombreux (voir proposition 7.2). Il est préférable de produire un graphe contenant toute l'information interrogeable par la suite.

Automate des alignements optimaux

Les alignements optimaux entre le mot x et le mot y sont représentés dans le graphe d'alignement par les chemins d'origine $(-1,-1)$ et de fin $(m-1,n-1)$ constitués de flèches actives. Le graphe des flèches actives apparaissant sur ces chemins et des sommets associés est un sous-graphe de $G(x,y)$. Quand on choisit $(-1,-1)$ pour état initial et $(m-1,n-1)$ pour état terminal, il devient un automate qui reconnait les alignements optimaux entre x et y (voir figure 7.9).

La construction de l'automate des alignements optimaux est donnée par l'algorithme dont le code suit. Le calcul revient à déterminer la partie coaccessible (depuis le sommet $(m-1,n-1)$) du graphe des flèches actives. La table E utilisée dans l'algorithme permet d'avoir un accès direct à l'état associé à chaque position sur la table T considérée pendant le déroulement de l'algorithme.

AUTO-ALIGN-OPT(x,m,y,n,T)
1 $M \leftarrow$ NOUVEL-AUTOMATE()
2 initialiser E
3 $E[-1,-1] \leftarrow \mathit{initial}[M]$
4 $E[m-1,n-1] \leftarrow$ NOUVEL-ÉTAT()
5 $\mathit{terminal}[E[m-1,n-1]] \leftarrow$ VRAI
6 AA$(m-1,n-1)$
7 **retourner** M

```
AA(i, j)
 1  si i ≠ −1 et j ≠ −1
      et T[i, j] = T[i − 1, j − 1] + Sub(x[i], y[j]) alors
 2        si E[i − 1, j − 1] = NIL alors
 3              E[i − 1, j − 1] ← NOUVEL-ÉTAT()
 4              AA(i − 1, j − 1)
 5        Succ[E[i − 1, j − 1]] ←
            Succ[E[i − 1, j − 1]] ∪ {((x[i], y[j]), E[i, j])}
 6  si i ≠ −1
      et T[i, j] = T[i − 1, j] + Dél(x[i]) alors
 7        si E[i − 1, j] = NIL alors
 8              E[i − 1, j] ← NOUVEL-ÉTAT()
 9              AA(i − 1, j)
10        Succ[E[i − 1, j]] ← Succ[E[i − 1, j]] ∪ {((x[i], ε), E[i, j])}
11  si j ≠ −1
      et T[i, j] = T[i, j − 1] + Ins(y[j]) alors
12        si E[i, j − 1] = NIL alors
13              E[i, j − 1] ← NOUVEL-ÉTAT()
14              AA(i, j − 1)
15        Succ[E[i, j − 1]] ← Succ[E[i, j − 1]] ∪ {((ε, y[j]), E[i, j])}
```

Les arguments pour prouver la validité du procédé sont identiques à ceux utilisés pour les algorithmes de production d'alignements optimaux. On remarque l'utilisation de la table E de taille $O(m \times n)$ qui permet de ne traiter chaque sommet de $G(x, y)$ qu'une seule fois (il est possible de la remplacer par une table de taille linéaire ; voir exercice 7.5).

Proposition 7.10
Soient e le nombre d'états de l'automate des alignements optimaux et f son nombre de flèches. L'algorithme AUTO-ALIGN-OPT *construit l'automate au moyen de la table T en temps $O(e + f)$.*

Preuve Les trois tests effectués dans la procédure AA servent à vérifier l'activité des flèches. Il suffit alors de vérifier que les flèches de l'automate en construction correspondent aux flèches actives de $G(x, y)$ qui se trouvent sur un chemin d'origine $(-1, -1)$ et de fin $(m - 1, n - 1)$.

Concernant le temps d'exécution, le seul point délicat est celui de l'initialisation de la table E (ligne 2). Celui-ci peut être $\Omega(m \times n)$ si elle est effectuée sans précaution. Il convient d'utiliser une technique d'implantation des fonctions partielles (voir exercice 1.15) qui permet de les initialiser en temps constant. ∎

On notera que l'automate des alignements optimaux peut être de taille linéaire $O(m + n)$, dans le cas où ils sont en nombre borné par exemple, et que dans cette situation l'algorithme AUTO-ALIGN-OPT le produit en temps linéaire. On remarque que le temps d'exécution de

l'algorithme est $O(m \times n)$ contrairement à celui de l'algorithme LES-ALIGNEMENTS.

7.3 Plus long sous-mot commun

On s'intéresse dans cette section au calcul d'un plus long sous-mot commun à deux mots. Ce problème est une spécialisation de la notion de distance d'édition dans laquelle on ne considère pas l'opération de substitution. Deux mots x et y peuvent avoir plusieurs plus longs sous-mots communs. L'ensemble de ces mots est noté $Smc(x, y)$. La longueur (unique) des mots de $Smc(x, y)$ est notée $smc(x, y)$.

Si l'on pose :

$Sub(a, a) = 0$

et :

$Dél(a) = Ins(a) = 1$

pour $a \in A$, et si l'on suppose :

$Sub(a, b) > Dél(a) + Ins(b) = 2$

pour $a, b \in A$ et $a \neq b$, la valeur $T[m-1, n-1]$ (voir section 7.2) représente ce que l'on appelle la ***distance par les sous-mots*** entre x et y, distance que l'on note $d_{smot}(x, y)$. Le calcul de cette distance est un problème dual du calcul de la longueur d'un ***plus long sous-mot commun*** à x et y grâce à la proposition qui suit (voir figure 7.7). C'est pourquoi l'on s'intéresse dans cette section à la recherche de plus longs sous-mots communs.

Proposition 7.11
La distance par les sous-mots vérifie l'égalité :

$$d_{smot}(x, y) = |x| + |y| - 2 \times smc(x, y) \ . \qquad (7.1)$$

Preuve Par définition, $d_{smot}(x, y)$ est le cout minimal des alignements entre les deux mots, comptabilisé à partir des couts élémentaires *Sub*, *Dél* et *Ins* qui satisfont les hypothèses ci-dessus. Soit z un alignement de cout $d_{smot}(x, y)$. L'inégalité :

$Sub(a, b) > Dél(a) + Ins(b)$

entraine que z ne contient pas de substitution de deux lettres différentes car une suppression de a et une insertion de b coute moins qu'une substitution de a par b quand $a \neq b$. Comme $Dél(a) = Ins(a) = 1$, la valeur $d_{smot}(x, y)$ est le nombre d'insertions et de suppressions contenues dans z. Les autres paires alignées de z correspondent à des égalités,

leur nombre est $smc(x, y)$ (il ne peut être plus petit sinon on obtient une contradiction avec la définition de $d_{smot}(x, y)$). Si chacune de ces paires est remplacée par une insertion suivie d'une suppression de la même lettre, on obtient un alignement qui ne contient que des insertions et des suppressions ; il est alors de longueur $|x| + |y|$. Le cout de z est donc $|x| + |y| - 2 \times smc(x, y)$, ce qui donne l'égalité de l'énoncé. ∎

Une méthode naïve pour calculer $smc(x, y)$ consiste à considérer tous les sous-mots de x, à vérifier s'ils sont des sous-mots de y et à conserver les plus longs. Comme le mot x de longueur m peut posséder 2^m sous-mots distincts, cette méthode par énumération est inapplicable pour de grandes valeurs de m.

Calcul par programmation dynamique

En utilisant la méthode de programmation dynamique, de façon analogue à la démarche de la section 7.2, il est possible de calculer $Smc(x, y)$ et $smc(x, y)$ en temps et en espace $O(m \times n)$. La méthode mène naturellement à calculer les longueurs de plus longs sous-mots communs à des préfixes de plus en plus longs des deux mots x et y.

Pour cela, nous considérons la table S à deux dimensions, $m+1$ lignes et $n+1$ colonnes, définie pour $i = -1, 0, \ldots, m-1$ et $j = -1, 0, \ldots, n-1$ par :

$$S[i, j] = \begin{cases} 0 & \text{si } i = -1 \text{ ou } j = -1 \ , \\ smc(x[0\,..\,i], y[0\,..\,j]) & \text{sinon} \ . \end{cases}$$

Calculer

$$smc(x, y) = S[m-1, n-1]$$

repose sur une simple observation qui conduit à la relation de récurrence de l'énoncé suivant (voir aussi figure 7.7).

Proposition 7.12
Pour $i = 0, 1, \ldots, m-1$ et $j = 0, 1, \ldots, n-1$, on a :

$$S[i, j] = \begin{cases} S[i-1, j-1] + 1 & \textit{si } x[i] = y[j] \ , \\ \max\{S[i-1, j], S[i, j-1]\} & \textit{sinon} \ . \end{cases}$$

Preuve Soient $ua = x[0\,..\,i]$ et $vb = y[0\,..\,j]$ $(u, v \in A^*,\ a, b \in A)$. Si $a = b$, un plus long sous-mot commun à ua et vb se termine par a ; sinon on pourrait le prolonger par a, ce qui contredirait la maximalité de sa longueur. Il en résulte qu'il est de la forme wa où w est un plus long sous-mot commun à u et v. D'où $S[i, j] = S[i-1, j-1] + 1$ dans ce cas.

Si $a \neq b$ et si ua et vb possède un plus long sous-mot commun qui ne se termine pas par a, on a $S[i, j] = S[i-1, j]$. De façon symétrique, s'il ne se termine pas par b, on a $S[i, j] = S[i, j-1]$. C'est-à-dire que dans ce cas $S[i, j] = \max\{S[i-1, j], S[i, j-1]\}$. ∎

(a)

S	j	-1	0	1	2	3	4	5	6	7	8	9
i		$y[j]$	C	A	G	A	T	C	A	G	A	G
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0	0
0	A	0	0	1	1	1	1	1	1	1	1	1
1	G	0	0	1	2	2	2	2	2	2	2	2
2	C	0	1	1	2	2	2	3	3	3	3	3
3	T	0	1	1	2	2	3	3	3	3	3	3
4	G	0	1	1	2	2	3	3	3	4	4	4
5	A	0	1	2	2	3	3	3	4	4	5	5

(b)

$$\begin{pmatrix} \texttt{-} & \texttt{A} & \texttt{G} & \texttt{C} & \texttt{-} & \texttt{T} & \texttt{-} & \texttt{-} & \texttt{G} & \texttt{A} & \texttt{-} \\ \texttt{C} & \texttt{A} & \texttt{G} & \texttt{-} & \texttt{A} & \texttt{T} & \texttt{C} & \texttt{A} & \texttt{G} & \texttt{A} & \texttt{G} \end{pmatrix} \quad \begin{pmatrix} \texttt{-} & \texttt{A} & \texttt{G} & \texttt{-} & \texttt{-} & \texttt{C} & \texttt{T} & \texttt{-} & \texttt{G} & \texttt{A} & \texttt{-} \\ \texttt{C} & \texttt{A} & \texttt{G} & \texttt{A} & \texttt{T} & \texttt{C} & \texttt{-} & \texttt{A} & \texttt{G} & \texttt{A} & \texttt{G} \end{pmatrix}$$

$$\begin{pmatrix} \texttt{-} & \texttt{A} & \texttt{G} & \texttt{-} & \texttt{C} & \texttt{T} & \texttt{-} & \texttt{-} & \texttt{G} & \texttt{A} & \texttt{-} \\ \texttt{C} & \texttt{A} & \texttt{G} & \texttt{A} & \texttt{-} & \texttt{T} & \texttt{C} & \texttt{A} & \texttt{G} & \texttt{A} & \texttt{G} \end{pmatrix} \quad \begin{pmatrix} \texttt{-} & \texttt{A} & \texttt{G} & \texttt{-} & \texttt{-} & \texttt{C} & \texttt{-} & \texttt{T} & \texttt{G} & \texttt{A} & \texttt{-} \\ \texttt{C} & \texttt{A} & \texttt{G} & \texttt{A} & \texttt{T} & \texttt{C} & \texttt{-} & \texttt{-} & \texttt{G} & \texttt{A} & \texttt{G} \end{pmatrix}$$

Figure 7.10 Calcul d'un plus long sous-mot commun aux mots $x = \texttt{AGCTGA}$ et $y = \texttt{CAGATCAGAG}$. **(a)** La table S et les chemins de cout maximal entre les positions $[-1,-1]$ et $[5,9]$ sur la table. **(b)** Les quatre alignements associés. Il en résulte que les mots `AGCGA` et `AGTGA` sont les plus long sous-mots communs à x et y.

L'égalité donnée dans l'énoncé précédent est utilisée par l'algorithme SMC-SIMPLE afin de calculer toutes les valeurs de la table S et produire $smc(x,y) = S[m-1, n-1]$.

SMC-SIMPLE(x, m, y, n)
1 **pour** $i \leftarrow -1$ **à** $m-1$ **faire**
2 $\quad S[i,-1] \leftarrow 0$
3 **pour** $j \leftarrow 0$ **à** $n-1$ **faire**
4 $\quad S[-1,j] \leftarrow 0$
5 $\quad$ **pour** $i \leftarrow 0$ **à** $m-1$ **faire**
6 $\quad\quad$ **si** $x[i] = y[j]$ **alors**
7 $\quad\quad\quad S[i,j] \leftarrow S[i-1,j-1]+1$
8 $\quad\quad$ **sinon** $S[i,j] \leftarrow \max\{S[i-1,j], S[i,j-1]\}$
9 **retourner** $S[m-1,n-1]$

Un exemple de calcul est montré figure 7.10.

Proposition 7.13
L'algorithme SMC-SIMPLE *calcule la longueur maximale des sous-mots communs à x et y. Il s'exécute en temps et espace $O(m \times n)$.*

Preuve La correction de l'algorithme résulte de la relation de récurrence de la proposition 7.12.

Il est immédiat que le temps de calcul et l'espace mémoire sont tous les deux $O(m \times n)$. ∎

Il est possible, après le calcul de la table S, de trouver un plus long sous-mot commun à x et à y en remontant dans la table S depuis la position $[m-1, n-1]$ (voir figure 7.10), comme effectué dans la section 7.2. Le code qui suit effectue ce calcul à la façon de l'algorithme UN-ALIGNEMENT.

```
UN-PLUS-LONG-SOUS-MOT-COMMUN(x, m, y, n, S)
 1  z ← ε
 2  (i, j) ← (m − 1, n − 1)
 3  tantque i ≠ −1 et j ≠ −1 faire
 4        si x[i] = y[j] alors
 5              z ← x[i] · z
 6              (i, j) ← (i − 1, j − 1)
 7        sinonsi S[i − 1, j] > S[i, j − 1] alors
 8              i ← i − 1
 9        sinon j ← j − 1
10  retourner z
```

Il est bien sûr possible de calculer comme dans la section 7.2 tous les plus longs sous-mots communs à x et y en prolongeant la technique utilisée dans l'algorithme précédent.

Calcul de la longueur en espace linéaire

Si seule la longueur d'un plus long sous-mot commun est désirée, il est simple de voir que la mémorisation de deux colonnes (ou deux lignes) de la table S sont suffisantes pour en faire le calcul (il est même possible de n'utiliser qu'une seule colonne ou une seule ligne pour effectuer ce calcul ; voir l'exercice 7.3). C'est précisément ce que réalise l'algorithme SMC-COLONNE dont le code figure ci-après.

```
SMC-COLONNE(x, m, y, n)
 1  pour i ← −1 à m − 1 faire
 2        C1[i] ← 0
 3  pour j ← 0 à n − 1 faire
 4        C2[−1] ← 0
 5        pour i ← 0 à m − 1 faire
 6              si x[i] = y[j] alors
 7                    C2[i] ← C1[i − 1] + 1
 8              sinon C2[i] ← max{C1[i], C2[i − 1]}
 9        C1 ← C2
10  retourner C1
```

Proposition 7.14
L'algorithme SMC-COLONNE *calcule une table C dont chaque valeur $C[i]$, $i = -1, 0, \ldots, m-1$, est égale à $smc(x[0\,..\,i], y)$. Le calcul est réalisé en temps $O(m \times n)$ et en espace $O(m)$.*

Preuve La table produite par l'algorithme est la table C_1. On obtient le résultat énoncé en montrant, par récurrence sur la valeur de j, que $C_1[i] = S[i, j]$, pour $i = -1, 0, \ldots, m-1$. En effet, lorsque $j = n-1$ à la fin de l'exécution de l'algorithme, on obtient $C_1[i] = S[i, n-1] = smc(x[0\,..\,i], y)$, pour $i = -1, 0, \ldots, m-1$, par définition de la table S.

Juste avant l'exécution de la ligne 3, ce qui précède peut être assimilé au traitement du cas $j = -1$; on a $C_1[i] = 0$ pour chaque valeur de i. On a également $S[i, -1] = 0$, ce qui prouve la relation pour $j = -1$.

Supposons maintenant que j ait une valeur positive. La valeur de la table C_1 correspondante est calculée aux lignes 4–9 de l'algorithme. D'après l'instruction à la ligne 9, il suffit de montrer que la table C_2 satisfait la relation ci-dessus lorsque, par hypothèse de récurrence, C_1 la satisfait pour la valeur $j-1$. On suppose donc que $C_1[i] = S[i, j-1]$ pour $i = -1, 0, \ldots, m-1$ et l'on montre qu'après l'exécution des lignes 4–8, on a $C_2[i] = S[i, j]$ pour $i = -1, 0, \ldots, m-1$.

La preuve se fait par récurrence sur la valeur de i. Pour $i = -1$, ce qui correspond à l'initialisation de la table C_2 à la ligne 4, on a $C_2[-1] = 0 = S[-1, j]$. Lorsque $i \geq 0$ deux cas sont à considérer. Si $x[i] = y[j]$, l'instruction associée conduit à poser $C_2[i] = C_1[i-1]+1$, ce qui est égal à $S[i-1, j-1]+1$ par application de l'hypothèse de récurrence sur j. Cette valeur est aussi $S[i, j]$ d'après la proposition 7.12, ce qui donne finalement $C_2[i] = S[i, j]$. Si $x[i] \neq y[j]$, l'instruction à la ligne 8 donne $C_2[i] = \max\{C_1[i], C_2[i-1]\}$, ce qui vaut $\max\{S[i, j-1], C_2[i-1]\}$, d'après l'hypothèse de récurrence sur j, puis $\max\{S[i, j-1], S[i-1, j]\}$ d'après l'hypothèse de récurrence sur i. On obtient finalement le résultat cherché, $C_2[i] = S[i, j]$, de nouveau par la proposition 7.12.

Cela termine les récurrences sur i et j, et donne le résultat. ∎

L'utilisation de l'algorithme SMC-COLONNE pour calculer la longueur maximale des sous-mots communs à x et y ne permet plus simplement la production d'un plus long sous-mot commun comme décrit précédemment (car la table S n'est pas complètement mémorisée). Mais l'algorithme est utilisé dans les calculs intermédiaires de la méthode qui suit.

Calcul d'un plus long sous-mot en espace linéaire

On montre à présent comment exhiber un plus long sous-mot commun par une approche du type « diviser pour régner ». La méthode s'exécute entièrement en espace linéaire. L'idée du calcul se décrit sur le graphe d'édition associé à x et y. Elle consiste à déterminer un sommet de la forme $(k-1, \lfloor n/2 \rfloor - 1)$, avec $0 \leq k \leq m$, par lequel passe un chemin de cout maximal d'origine $(-1, -1)$ et de fin $(m-1, n-1)$ dans le graphe. Une fois ce sommet connu, il ne reste plus qu'à calculer les deux portions du chemin, de $(-1, -1)$ à $(k-1, \lfloor n/2 \rfloor - 1)$ et de $(k-1, \lfloor n/2 \rfloor - 1)$ à $(m-1, n-1)$. Cela revient à trouver un plus long sous-mot u commun

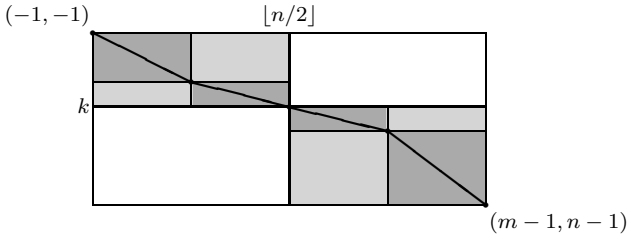


Figure 7.11 Schéma d'une méthode « diviser pour régner » afin de calculer un plus long sous-mot commun à deux mots en espace linéaire. Le temps du calcul de chaque étape est proportionnel à la surface des rectangles considérés. Comme celle-ci est divisée par deux à chaque niveau de la récurrence, on obtient un temps total $O(m \times n)$.

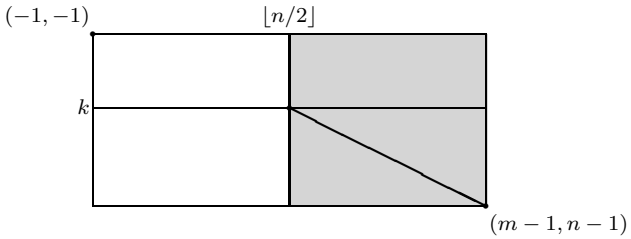


Figure 7.12 Lors du calcul de la seconde moitié des valeurs (partie grisée), on mémorise pour chaque sommet (i, j) une position de la colonne du milieu par laquelle passe un chemin de cout minimal d'origine $(-1, -1)$ et d'extrémité (i, j). Seul le pointeur vers $(m-1, n-1)$ est utilisé pour la suite du calcul.

à $x[0\,..\,k-1]$ et $y[0\,..\,\lfloor n/2\rfloor - 1]$, puis, un plus long sous-mot v commun à $x[k\,..\,m-1]$ et $y[\lfloor n/2\rfloor\,..\,n-1]$. Ces deux calculs sont effectués en appliquant la même méthode récursivement (voir figure 7.11). Le mot $z = u \cdot v$ est alors un plus long sous-mot commun à x et y. La récurrence s'arrête lorsqu'un des deux mots est vide ou réduit à une seule lettre. Dans ce cas, un test simple permet de conclure.

Il reste à décrire comment obtenir l'indice k qui identifie le sommet $(k-1, \lfloor n/2\rfloor - 1)$ cherché. L'entier k est, par définition, un indice compris entre 0 et m pour lequel la quantité

$$\begin{aligned} &\mathit{smc}(x[0\,..\,k-1], y[0\,..\,\lfloor n/2\rfloor - 1]) \\ &\quad + \mathit{smc}(x[k\,..\,m-1], y[\lfloor n/2\rfloor\,..\,n-1]) \end{aligned}$$

est maximale (figure 7.12). Pour le trouver, l'algorithme SMC – son code est donné plus loin –, commence par calculer la colonne d'indice $\lfloor n/2\rfloor - 1$ de la table S en appelant (ligne 7) SMC-COLONNE$(x, m, y, \lfloor n/2\rfloor)$. Pour la suite du calcul à cette étape (lignes 8–18), et avant les appels récursifs, l'algorithme procède sur la deuxième moitié de la table S comme le fait l'algorithme SMC-COLONNE sur la première moitié, mais en mémorisant

en plus des pointeurs vers la colonne du milieu. Le calcul utilise toujours deux tables C_1 et C_2 afin de calculer les valeurs de S, mais aussi deux tables supplémentaires P_1 et P_2 pour stocker les pointeurs.

Ces deux dernières tables implantent la table P définie, pour $j = \lfloor n/2 \rfloor - 1, \lfloor n/2 \rfloor, \ldots, n-1$ et $i = -1, 0, \ldots, m-1$, par :

$P[i,j] = k$

si et seulement si

$0 \leq k \leq i+1$

et

$$\begin{aligned} &smc(x[0\,..\,i], y[0\,..\,j]) = \\ &\quad smc(x[0\,..\,k-1], y[0\,..\,\lfloor n/2 \rfloor - 1]) \\ &\quad + smc(x[k\,..\,i], y[\lfloor n/2 \rfloor\,..\,j]) \ . \end{aligned} \tag{7.2}$$

La proposition qui suit fournit le moyen utilisé par l'algorithme SMC, pour calculer les valeurs de la table P. On constate que la récurrence qui y est énoncée permet un calcul colonne par colonne comme pour celui de la table S effectué par SMC-COLONNE. C'est en partie cette propriété qui conduit à un calcul d'un plus long sous-mot commun en espace linéaire. On montre figure 7.13 un exemple d'exécution de la méthode.

Proposition 7.15
La table P satisfait les relations de récurrence suivantes :

$P[i, \lfloor n/2 \rfloor - 1] = i + 1$,

pour $i = -1, 0, \ldots, m-1$,

$P[-1, j] = 0$,

pour $j \geq \lfloor n/2 \rfloor$, *et*

$$P[i,j] = \begin{cases} P[i-1, j-1] & \text{si } x[i] = y[j] \ , \\ P[i-1, j] & \text{si } x[i] \neq y[j] \text{ et } S[i-1,j] > S[i,j-1] \ , \\ P[i, j-1] & \text{sinon} \ , \end{cases}$$

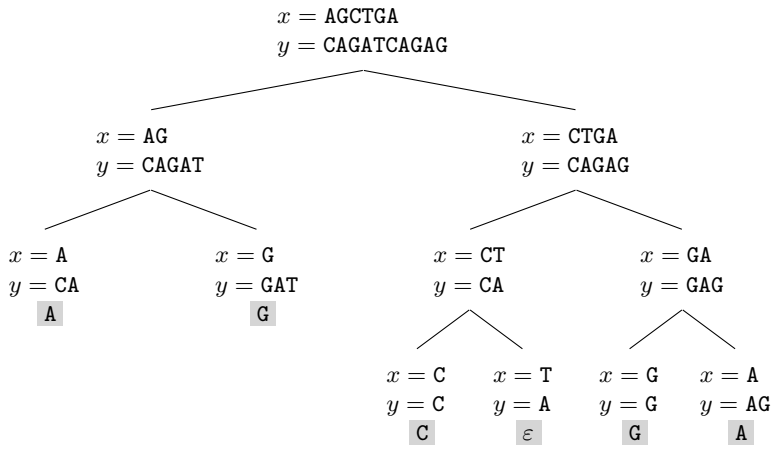
pour $i = 0, 1, \ldots, m-1$ *et* $j = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots, n-1$.

Preuve On montre la propriété par récurrence sur le couple (i,j) (couples ordonnés par l'ordre lexicographique).

Si $j = \lfloor n/2 \rfloor - 1$, par définition de P, $k = i+1$ car le second terme de la somme dans l'équation (7.2) est nul du fait que $y[\lfloor n/2 \rfloor\,..\,j]$ est le mot vide. L'initialisation de la récurrence est donc correcte.

Considérons maintenant que $j \geq \lfloor n/2 \rfloor$. Si $i = -1$, par définition de P, $k = 0$ et l'équation (7.2) est trivialement satisfaite puisque les facteurs de x considérés sont vides.

(a)



(b)

i	$j=5$	$j=6$	$j=7$	$j=8$	$j=9$
-1	0	0	0	0	0
0	1	0	0	0	0
1	2	2	0	0	0
2	2	2	2	2	2
3	2	2	2	2	2
4	2	2	2	2	2
5	2	2	2	2	2

Figure 7.13 Illustration de l'exécution de l'algorithme SMC avec les mots `AGCTGA` et `CAGATCAGAG`. **(a)** Arbre des appels récursifs. Le plus long sous-mot commun, `AGCGA`, produit par l'algorithme est obtenu par concaténation des résultats obtenus sur les feuilles de l'arbre parcourues de gauche à droite. **(b)** Les valeurs de la table P_1 des pointeurs après chacune des itérations de la boucle **pour** des lignes 10–18 lors de l'appel initial. La valeur de k calculée lors de cet appel est $P_1[5] = 2$ obtenue après le traitement $j = 9$, ce qui correspond à la décomposition $smc(\texttt{AGCTGA}, \texttt{CAGATCAGAG}) = smc(\texttt{AG}, \texttt{CAGAT}) + smc(\texttt{CTGA}, \texttt{CAGAG})$.

Il reste à traiter le cas général $j \geq \lfloor n/2 \rfloor$ et $i \geq 0$. Supposons que l'on ait $x[i] = y[j]$. Il existe alors, dans le graphe d'édition, un chemin de cout maximal, d'origine $(-1,-1)$ et de fin (i,j), passant par $(i-1, j-1)$. Donc il en existe un passant par $(k-1, \lfloor n/2 \rfloor - 1)$ où $k = P[i-1, j-1]$. Dit autrement, on a d'après la proposition 7.12 $smc(x[0\,..\,i], y[0\,..\,j]) = smc(x[0\,..\,i-1], y[0\,..\,j-1]) + 1$, et par récurrence, $smc(x[0\,..\,i-1], y[0\,..\,j-1]) = smc(x[0\,..\,k-1], y[0\,..\,\lfloor n/2 \rfloor - 1]) + smc(x[k\,..\,i-1], y[\lfloor n/2 \rfloor\,..\,j-1])$. L'hypothèse $x[i] = y[j]$ impliquant également $smc(x[k\,..\,i], y[\lfloor n/2 \rfloor\,..\,j]) = smc(x[k\,..\,i-1], y[\lfloor n/2 \rfloor\,..\,j-1]) + 1$, on en déduit $smc(x[0\,..\,i], y[0\,..\,j]) = smc(x[0\,..\,k-1], y[0\,..\,\lfloor n/2 \rfloor - 1]) + smc(x[k\,..\,i], y[\lfloor n/2 \rfloor\,..\,j])$, ce qui donne, par définition de P, $P[i,j] = k = P[i-1, j-1]$, comme indiqué dans l'énoncé.

Les deux derniers cas se traitent de façon analogue. ∎

Le code de l'algorithme SMC est donné ci-dessous sous la forme d'une fonction récursive.

```
SMC(x, m, y, n)
 1  si m = 1 et x[0] ∈ alph(y) alors
 2        retourner x[0]
 3  sinonsi n = 1 et y[0] ∈ alph(x) alors
 4        retourner y[0]
 5  sinonsi m = 0 ou m = 1 ou n = 0 ou n = 1 alors
 6        retourner ε
 7  C_1 ← SMC-COLONNE(x, m, y, ⌊n/2⌋)
 8  pour i ← −1 à m − 1 faire
 9        P_1[i] ← i + 1
10  pour j ← ⌊n/2⌋ à n − 1 faire
11        (C_2[−1], P_2[−1]) ← (0, 0)
12        pour i ← 0 à m − 1 faire
13              si x[i] = y[j] alors
14                    (C_2[i], P_2[i]) ← (C_1[i − 1] + 1, P_1[i − 1])
15              sinonsi C_1[i] > C_2[i − 1] alors
16                    (C_2[i], P_2[i]) ← (C_1[i], P_1[i])
17              sinon (C_2[i], P_2[i]) ← (C_2[i − 1], P_2[i − 1])
18        (C_1, P_1) ← (C_2, P_2)
19  k ← P_1[m − 1]
20  u ← SMC(x[0 .. k − 1], k, y[0 .. ⌊n/2⌋ − 1], ⌊n/2⌋)
21  v ← SMC(x[k .. m − 1], m − k, y[⌊n/2⌋ .. n − 1], n − ⌊n/2⌋)
22  retourner u · v
```

Proposition 7.16
L'opération $\mathrm{SMC}(x, m, y, n)$ *produit un plus long sous-mot commun aux mots* x *et* y *de longueurs respectives* m *et* n.

Preuve La preuve se fait par récurrence sur la longueur n du mot y. Il s'agit d'une simple vérification lorsque $n = 0$ ou $n = 1$.

Considérons ensuite que $n > 1$. Si $m = 0$ ou $m = 1$, on vérifie aussi simplement que l'opération fournit bien un plus long sous-mot commun à x et y. On peut donc supposer maintenant que $m > 1$.

On constate que les instructions des lignes 10–18 poursuivent le calcul de la table C_1 commencé par l'appel à l'algorithme SMC-COLONNE ligne 7 en appliquant la même relation de récurrence. On constate en plus que la table P_1 qui implante la table P est calculée au moyen des relations de récurrence de la proposition 7.15, ce qui résulte du calcul correct de la table C_1. On a donc immédiatement après l'exécution de la ligne 19 l'égalité $k = P_1[m-1] = P[m-1, n-1]$, ce qui signifie que $smc(x, y) = smc(x[0\,..\,k-1], y[0\,..\,\lfloor n/2 \rfloor - 1]) + smc(x[k\,..\,m-1], y[\lfloor n/2 \rfloor\,..\,n-1])$ par définition de P. Comme par hypothèse de récurrence les appels à l'algorithme aux lignes 20 et 21 fournissent des plus longs sous-mots communs à leurs mots d'entrée (qui sont correctement choisis), leur concaténation est un plus long sous-mot commun à x et y.

Cela termine la récurrence et la preuve de la proposition. ∎

Proposition 7.17
L'opération $\mathrm{SMC}(x, m, y, n)$ *s'exécute en temps* $\Theta(m \times n)$. *Elle peut être réalisée dans un espace* $\Theta(m)$.

Preuve Lors de l'appel initial à l'algorithme, les instructions aux lignes 1–19 s'exécutent en temps $\Theta(m \times n)$.

Les instructions aux mêmes lignes lors des appels immédiatement successifs effectués aux lignes 20 et 21 prennent respectivement des temps proportionnels à $k \times \lfloor n/2 \rfloor$ et $(m-k) \times (n - \lfloor n/2 \rfloor)$, soit $(m \times n)/2$ (voir figure 7.11).

Il s'ensuit que le temps d'exécution global est $O(m \times n)$ puisque $\sum_i (m \times n)/2^i \leq 2m \times n$. Mais il est aussi $\Omega(m \times n)$ en raison de la première étape, ce qui donne le premier résultat de l'énoncé.

L'espace mémoire est utilisé par l'algorithme SMC pour mémoriser les tables C_1, C_2, P_1 et P_2 en dehors de quelques variables qui occupent un espace constant. L'ensemble occupe donc un espace $O(m)$. Et comme les appels récursifs à l'algorithme ne nécessitent pas de conserver les informations stockées dans les tables, leur espace peut être réutilisé pour la suite du calcul. D'où le résultat. ∎

Le théorème suivant fournit la conclusion de la section.

Théorème 7.18
Il est possible de calculer un plus long sous-mot commun à deux mots de longueurs m *et* n *en temps* $O(m \times n)$ *dans un espace* $O(\min\{m, n\})$.

Preuve C'est une conséquence directe des propositions 7.16 et 7.17 en choisissant pour mot x le plus court des deux mots d'entrée de l'algorithme SMC. ∎

7.4 Alignement avec brèches

Une ***brèche*** est une suite consécutive de trous dans un alignement. L'utilisation des alignements sur des séquences génétiques montre qu'il est quelquefois souhaitable de pénaliser la formation de longues brèches, dans un calcul d'alignement, au lieu de pénaliser individuellement la suppression ou l'insertion de telle ou telle lettre de l'alphabet. Les trous ne sont donc plus comptabilisés indépendamment de leur position. Mais aucune information extérieure aux mots n'est utilisée dans la définition.

Dans ce contexte, le cout minimal d'une suite d'opérations d'édition est une distance sous des conditions analogues à celles de la proposition 7.1, essentiellement car la symétrie entre suppression et insertion est respectée. On introduit la fonction

$$br\colon \mathbf{N} \to \mathbf{R} \ ,$$

dont chaque image $br(k)$ désigne le cout d'une brèche de longueur k. L'algorithme Calcul-générique de la section 7.2 ne s'applique pas directement au calcul d'une distance prenant en compte l'hypothèse ci-dessus, mais son adaptation est relativement immédiate.

Afin de calculer un alignement optimal dans cette situation, on utilise trois tables : D, I et T. La valeur $D[i,j]$ désigne le cout d'un meilleur alignement entre $x[0\,..\,i]$ et $y[0\,..\,j]$ se terminant par des suppressions de lettres de x. La valeur $I[i,j]$ désigne le cout d'un meilleur alignement entre $x[0\,..\,i]$ et $y[0\,..\,j]$ se terminant par des insertions de lettres de y. Enfin, la valeur $T[i,j]$ donne le cout d'un meilleur alignement entre $x[0\,..\,i]$ et $y[0\,..\,j]$. Les tables sont liées par les relations de récurrence de la proposition qui suit.

Proposition 7.19
Le cout $T[i,j]$, d'un meilleur alignement entre $x[0\,..\,i]$ et $y[0\,..\,j]$, est donné par les relations de récurrence suivantes :

$$\begin{array}{rcl} D[-1,-1] = D[i,-1] = D[-1,j] & = & \infty \ , \\ I[-1,-1] = I[i,-1] = I[-1,j] & = & \infty \ , \end{array}$$

et :

$$\begin{array}{rcl} T[-1,-1] & = & 0 \ , \\ T[i,-1] & = & br(i+1) \ , \\ T[-1,j] & = & br(j+1) \ , \\ D[i,j] & = & \min\{T[\ell,j] + br(i-\ell) : \ell = 0,1,\ldots,i-1\} \ , \\ I[i,j] & = & \min\{T[i,k] + br(j-k) : k = 0,1,\ldots,j-1\} \ , \\ T[i,j] & = & \min\{T[i-1,j-1] + \mathit{Sub}(x[i],y[j]), D[i,j], I[i,j]\} \ , \end{array}$$

pour $i = 0,1,\ldots,m-1$ et $j = 0,1,\ldots,n-1$.

Preuve La preuve peut s'obtenir en utilisant des arguments similaires à ceux de la proposition 7.3. Elle se décompose alors en trois cas, car un alignement optimal entre $x[0\,..\,i]$ et $y[0\,..\,j]$ peut se terminer uniquement de trois manières : soit par une substitution de $x[i]$ par $y[j]$; soit par la suppression de ℓ lettres à la fin de x ; soit par l'insertion de k lettres à la fin de y avec $0 \leq \ell < i$ et $0 \leq k < j$. ∎

Si aucune restriction n'est faite sur la fonction br, on peut vérifier que le problème du calcul d'un alignement optimal entre x et y se résout en temps $O(m \times n \times (m+n))$. En revanche, on montre que le problème se résout en temps $O(m \times n)$ si la fonction br est une fonction affine, c'est-à-dire de la forme

$$br(k) = g + h \times (k-1)$$

avec g et h deux constantes entières positives. Ce type de fonction revient à pénaliser l'ouverture d'une brèche d'une quantité g et à pénaliser la prolongation d'une brèche d'une quantité h. Dans les applications réelles, on choisit habituellement les deux constantes de façon à ce que $h < g$. Les relations de récurrence de la proposition ci-dessus deviennent :

$$\begin{aligned}
D[i,j] &= \min\{D[i-1,j]+h, T[i-1,j]+g\}\ , \\
I[i,j] &= \min\{I[i,j-1]+h, T[i,j-1]+g\}\ , \\
T[i,j] &= \min\{T[i-1,j-1]+Sub(x[i],y[j]), D[i,j], I[i,j]\}\ ,
\end{aligned}$$

pour $i = 0, 1, \ldots, m-1$ et $j = 0, 1, \ldots, n-1$. On pose en plus :

$$\begin{aligned}
D[-1,-1] &= D[i,-1] = D[-1,j] = \infty\ , \\
I[-1,-1] &= I[i,-1] = I[-1,j] = \infty\ ,
\end{aligned}$$

pour $i = 0, 1, \ldots, m-1$ et $j = 0, 1, \ldots, n-1$, et :

$$\begin{aligned}
T[-1,-1] &= 0\ , \\
T[0,-1] &= g\ , \\
T[-1,0] &= g\ , \\
T[i,-1] &= T[i-1,-1]+h\ , \\
T[-1,j] &= T[-1,j-1]+h\ ,
\end{aligned}$$

pour $i = 1, 2, \ldots, m-1$ et $j = 1, 2, \ldots, n-1$.

L'algorithme Brèche, dont le code suit, utilise ces relations de récurrence. Les tables D, I et T considérées dans le code sont de dimension $(m+1) \times (n+1)$. Un exemple d'exécution de l'algorithme est montré figure 7.14.

(a)

D	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	E	R	D	A	W	C	Q	P	G	K	W	Y
-1	$x[i]$	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
0	E	∞	6	7	8	9	10	11	12	13	14	15	16	17
1	A	∞	3	6	7	8	9	10	11	12	13	14	15	16
2	W	∞	4	6	8	7	10	11	12	13	14	15	16	17
3	A	∞	5	7	9	8	7	10	11	12	13	14	15	16
4	C	∞	6	8	10	9	8	10	12	13	14	15	16	17
5	Q	∞	7	9	11	10	9	10	13	14	15	16	17	18
6	G	∞	8	10	12	11	10	11	10	13	14	15	16	17
7	K	∞	9	11	13	12	11	12	11	13	13	16	17	18
8	L	∞	10	12	14	13	12	13	12	14	14	13	16	17

(b)

I	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	E	R	D	A	W	C	Q	P	G	K	W	Y
-1	$x[i]$	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
0	E	∞	6	3	4	5	6	7	8	9	10	11	12	13
1	A	∞	7	6	6	7	7	8	9	10	11	12	13	14
2	W	∞	8	7	8	9	10	7	8	9	10	11	12	13
3	A	∞	9	8	9	10	9	10	10	11	12	13	14	15
4	C	∞	10	9	10	11	12	11	10	11	12	13	14	15
5	Q	∞	11	10	11	12	13	12	13	10	11	12	13	14
6	G	∞	12	11	12	13	14	13	14	13	13	13	14	15
7	K	∞	13	12	13	14	15	14	15	14	15	16	13	14
8	L	∞	14	13	14	15	16	15	16	15	16	17	16	16

(c)

T	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	E	R	D	A	W	C	Q	P	G	K	W	Y
-1	$x[i]$	0	3	4	5	6	7	8	9	10	11	12	13	14
0	E	3	0	3	4	5	6	7	8	9	10	11	12	13
1	A	4	3	3	6	4	7	8	9	10	11	12	13	14
2	W	5	4	6	6	7	4	7	8	9	10	11	12	13
3	A	6	5	7	9	6	7	7	10	11	12	13	14	15
4	C	7	6	8	10	9	8	7	10	11	12	13	14	15
5	Q	8	7	9	11	10	9	10	7	10	11	12	13	14
6	G	9	8	10	12	11	10	11	10	10	10	13	14	15
7	K	10	9	11	13	12	11	12	11	13	13	10	13	14
8	L	11	10	12	14	13	12	13	12	14	14	13	13	16

(d)

$$\begin{pmatrix} \texttt{E} & \texttt{-} & \texttt{-} & \texttt{A} & \texttt{W} & \texttt{A} & \texttt{C} & \texttt{Q} & \texttt{-} & \texttt{G} & \texttt{K} & \texttt{-} & \texttt{L} \\ \texttt{E} & \texttt{R} & \texttt{D} & \texttt{A} & \texttt{W} & \texttt{-} & \texttt{C} & \texttt{Q} & \texttt{P} & \texttt{G} & \texttt{K} & \texttt{W} & \texttt{Y} \end{pmatrix}$$

$$\begin{pmatrix} \texttt{E} & \texttt{-} & \texttt{-} & \texttt{A} & \texttt{W} & \texttt{A} & \texttt{C} & \texttt{Q} & \texttt{-} & \texttt{G} & \texttt{K} & \texttt{L} & \texttt{-} \\ \texttt{E} & \texttt{R} & \texttt{D} & \texttt{A} & \texttt{W} & \texttt{-} & \texttt{C} & \texttt{Q} & \texttt{P} & \texttt{G} & \texttt{K} & \texttt{W} & \texttt{Y} \end{pmatrix}$$

Figure 7.14 Exemple de calcul effectué avec l'algorithme Brèche sur les mots de la figure 7.7, `EAWACQGKL` et `ERDAWCQPGKWY`. On considère les valeurs $g = 3$, $h = 1$, $Sub(a, a) = 0$ et $Sub(a, b) = 3$ pour toutes lettres $a, b \in A$ telles que $a \neq b$. **(a)**–**(c)** Tables D, I et T, dans cet ordre. **(d)** Les deux alignements optimaux obtenus grâce à une méthode similaire à celle de l'algorithme Les-alignements.

```
BRÈCHE(x, m, y, n)
 1  pour i ← −1 à m − 1 faire
 2        (D[i, −1], I[i, −1]) ← (∞, ∞)
 3  T[−1, −1] ← 0
 4  T[0, −1] ← g
 5  pour i ← 1 à m − 1 faire
 6        T[i, −1] ← T[i − 1, −1] + h
 7  T[−1, 0] ← g
 8  pour j ← 1 à n − 1 faire
 9        T[−1, j] ← T[−1, j − 1] + h
10  pour j ← 0 à n − 1 faire
11        (D[−1, j], I[−1, j]) ← (∞, ∞)
12        pour i ← 0 à m − 1 faire
13              D[i, j] ← min{D[i − 1, j] + h, T[i − 1, j] + g}
14              I[i, j] ← min{I[i, j − 1] + h, T[i, j − 1] + g}
15              t ← T[i − 1, j − 1] + Sub(x[i], y[j])
16              T[i, j] ← min{t, D[i, j], I[i, j]}
17  retourner T[m − 1, n − 1]
```

Les tables D, I et T utilisées dans l'algorithme peuvent être réduites dans le but d'occuper un espace linéaire en adaptant la technique de la section 7.3. L'énoncé qui suit résume le résultat de la section.

Proposition 7.20
Avec une fonction de cout des brèches qui est affine, l'alignement optimal de mots de longueurs m et n se calcule en temps $O(m \times n)$ et espace $O(\min\{m, n\})$. ∎

7.5 Meilleur alignement local

Au lieu de s'intéresser à un alignement global entre x et y, il est souvent plus pertinent de déterminer un meilleur alignement entre un facteur de x et un facteur de y. La notion de distance n'est pas appropriée pour résoudre cette question. En effet, lorsque l'on cherche à minimiser une distance, les facteurs qui amènent les plus petites valeurs sont les facteurs qui apparaissent simultanément dans les deux mots x et y, facteurs qui risquent d'être réduits à quelques lettres seulement. On utilise donc plutôt une notion de similarité entre mots, pour laquelle les égalités entre lettres sont valuées positivement, et les inégalités, insertions et suppressions sont valuées négativement. La recherche d'un facteur similaire consiste alors à maximiser une quantité représentative de la similarité entre mots.

Similarité

Afin de mesurer le degré de similarité entre deux mots x et y, on utilise une ***fonction de score***. Cette fonction, notée Sub_{S}, mesure le degré de ressemblance entre deux lettres de l'alphabet. La valeur $Sub_{\mathrm{S}}(a, b)$ est d'autant plus grande que les deux lettres a et b sont semblables. On suppose que la fonction satisfait :

$Sub_{\mathrm{S}}(a, a) > 0$

pour $a \in A$ et :

$Sub_{\mathrm{S}}(a, b) < 0$

pour $a, b \in A$ avec $a \neq b$. La fonction Sub_{S} est symétrique. Mais elle n'est pas une distance car elle ne satisfait pas la condition de positivité, ni celle de séparation, ni même l'inégalité triangulaire. En effet, on peut attribuer des scores différents à plusieurs égalités de lettres : on peut avoir $Sub_{\mathrm{S}}(a, a) \neq Sub_{\mathrm{S}}(b, b)$. Cela autorise un meilleur contrôle des égalités plus vivement souhaitées. Les fonctions d'insertion et de suppression doivent elles aussi être valuées négativement (leurs valeurs sont entières) :

$Ins_{\mathrm{S}}(a) < 0$

et :

$Dél_{\mathrm{S}}(a) < 0$

pour $a \in A$.

On définit ensuite la ***similarité*** $sim(u, v)$ entre les mots u et v par

$$sim(u, v) = \max\{\text{score de } \sigma : \sigma \in \Sigma_{u,v}\} \ ,$$

où $\Sigma_{u,v}$ est l'ensemble des suites d'opérations d'édition permettant de transformer u en v. Le score d'un élément $\sigma \in \Sigma_{u,v}$ est la somme des scores des opérations d'édition de σ.

On peut montrer la propriété suivante qui établit la relation entre les notions de distance et de similarité (voir les notes).

Proposition 7.21
Étant données Sub, une distance sur les lettres, Ins et Dél, deux fonctions sur les lettres, de valeur constante g, et une constante ℓ, on définit un système de score de la façon suivante :

$Sub_{\mathrm{S}}(a, b) = \ell - Sub(a, b)$

et :

$$Ins_{\mathrm{S}}(a) = Dél_{\mathrm{S}}(a) = -g + \frac{\ell}{2}$$

pour toutes les lettres $a, b \in A$. Alors on a :

$$Lev(u, v) + sim(u, v) = \frac{\ell}{2}(|u| + |v|)$$

pour tous mots $u, v \in A^*$. ∎

Calcul d'un meilleur alignement local

Un meilleur alignement local entre les mots x et y est un couple de mots (u, v) pour lequel $u \preceq_{fact} x$, $v \preceq_{fact} y$ et $sim(u, v)$ est maximal. Pour en effectuer le calcul par un procédé analogue à celui de la section 7.2, on considère une table S définie, pour $i = -1, 0, \ldots, m-1$ et $j = -1, 0, \ldots, n-1$, par : $S[i, j]$ est la similarité maximale entre un suffixe de $x[0\,..\,i]$ et un suffixe de $y[0\,..\,j]$. Soit encore :

$$S[i,j] = \max\{sim(x[\ell\,..\,i], y[k\,..\,j]) : 0 \leq \ell \leq i \text{ et } 0 \leq k \leq j\} \cup \{0\} \ ,$$

le score de l'alignement local en $[i, j]$. Les meilleurs alignements locaux sont alors calculés à partir des valeurs maximales dans la table.

Proposition 7.22
La table S satisfait les relations de récurrence :

$$S[i,j] = \max \begin{cases} 0 \ , \\ S[i-1, j-1] + Sub_{\mathrm{S}}(x[i], y[j]) \ , \\ S[i-1, j] + Dél_{\mathrm{S}}(x[i]) \ , \\ S[i, j-1] + Ins_{\mathrm{S}}(y[j]) \ , \end{cases}$$

et :

$$S[-1,-1] = S[i,-1] = S[-1,j] = 0$$

pour $i = 0, 1, \ldots, m-1$ et $j = 0, 1, \ldots, n-1$.

Preuve La preuve est analogue à celle de la proposition 7.3. ∎

L'algorithme ALIGNEMENT-LOCAL, dont le code est donné ci-dessous, utilise la relation de récurrence de la proposition précédente.

ALIGNEMENT-LOCAL(x, m, y, n)
1 **pour** $i \leftarrow -1$ **à** $m-1$ **faire**
2 $\quad S[i,-1] \leftarrow 0$
3 **pour** $j \leftarrow 0$ **à** $n-1$ **faire**
4 $\quad S[-1,j] \leftarrow 0$
5 $\quad$ **pour** $i \leftarrow 0$ **à** $m-1$ **faire**
6 $\quad\quad S[i,j] \leftarrow \max \begin{cases} 0 \\ S[i-1, j-1] + Sub_{\mathrm{S}}(x[i], y[j]) \\ S[i-1, j] + Dél_{\mathrm{S}}(x[i]) \\ S[i, j-1] + Ins_{\mathrm{S}}(y[j]) \end{cases}$
7 **retourner** S

Proposition 7.23
L'algorithme ALIGNEMENT-LOCAL *calcule les scores de tous les alignements locaux entre x et y.*

(a)

T	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	E	R	D	A	W	C	Q	P	G	K	W	Y
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0	0	0	0
0	E	0	1	0	0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	0	1	0	0	0	0	0	0	0	0
2	W	0	0	0	0	0	2	1	0	0	0	0	1	0
3	A	0	0	0	0	1	1	0	0	0	0	0	0	0
4	C	0	0	0	0	0	0	2	1	0	0	0	0	0
5	Q	0	0	0	0	0	0	1	3	2	1	0	0	0
6	G	0	0	0	0	0	0	0	2	1	3	2	1	0
7	K	0	0	0	0	0	0	0	1	0	2	4	3	2
8	L	0	0	0	0	0	0	0	0	0	1	3	2	1

(b) $\begin{pmatrix} \texttt{A} & \texttt{W} & \texttt{A} & \texttt{C} & \texttt{Q} & \texttt{-} & \texttt{G} & \texttt{K} \\ \texttt{A} & \texttt{W} & \texttt{-} & \texttt{C} & \texttt{Q} & \texttt{P} & \texttt{G} & \texttt{K} \end{pmatrix}$

Figure 7.15 Calcul d'un alignement local optimal pour les mots `EAWACQGKL` et `ERDAWCQPGKWY` lorsque $Sub_S(a,a) = 1$, $Sub_S(a,b) = -3$ et $Dél_S(a) = Ins_S(a) = -1$ pour $a, b \in A$, $a \neq b$. **(a)** Table T des couts de tous les alignements locaux, et le chemin aboutissant sur la position contenant la plus grande valeur. **(b)** L'alignement local optimal correspondant.

Preuve C'est une application immédiate de la proposition 7.22. car l'algorithme utilise les relations de cette proposition. ∎

Pour trouver un alignement local optimal, il suffit de rechercher une plus grande valeur dans la table T. On retrace ensuite le chemin à partir de la position de cette valeur en remontant dans la table (voir section 7.2). On arrête le parcours, en général, sur une valeur nulle. Un exemple est fourni figure 7.15.

7.6 Heuristique pour l'alignement local

Les méthodes d'alignement sont souvent utilisées pour comparer des mots sélectionnés. Mais elles sont aussi précieuses pour la recherche de ressemblance entre un mot choisi et ceux d'une banque de données. Dans ce cas, on désire rechercher les similarités entre un mot $x \in A^*$ et chacun des mots d'un ensemble fini $Y \subset A^*$. Il est primordial d'effectuer chaque alignement en un temps raisonnable, car le procédé doit être répété sur tous les mots $y \in Y$. Il faut donc trouver des solutions plus rapides que celle fournie par la méthode de programmation dynamique. Les solutions retenues habituellement font appel à des heuristiques et sont des méthodes approchées : elles peuvent manquer quelques bonnes réponses

au problème posé et peuvent aussi donner quelques réponses erronées. Mais elles ont un comportement satisfaisant sur les exemples réels.

La méthode décrite ici permet de trouver un bon alignement local entre un facteur de x et un facteur de y sans autoriser d'insertions ni de suppressions. Cette hypothèse simplifie le problème. La comparaison est itérée sur chacun des mots y de la banque Y.

Pour deux entiers ℓ et k donnés, on considère l'ensemble des mots de longueur ℓ qui sont à distance au plus k d'un facteur de longueur ℓ du mot x. On considère ici une généralisation de la distance de Hamming prenant en compte le cout d'une substitution. Les mots ainsi définis à partir de tous les facteurs de longueur ℓ de x sont appelés les ***voisins fréquentables*** des facteurs de longueur ℓ de x.

L'analyse du texte y consiste à y localiser la plus longue suite d'occurrences de voisins fréquentables ; elle produit un facteur de y similaire à un facteur de x. Afin de réaliser la localisation, on utilise un automate qui reconnait l'ensemble des voisins fréquentables. La construction de l'automate est un élément important de la méthode.

On considère la distance d définie par :

$$d(u,v) = \sum_{i=0}^{|u|-1} \mathit{Sub}(u[i], v[i])$$

pour deux mots u et v de même longueur (on suppose que *Sub* est une distance sur l'alphabet). Pour tout entier naturel ℓ, on note $\mathit{Fact}_\ell(x)$ l'ensemble des facteurs de longueur ℓ du mot x, et $V_k(\mathit{Fact}_\ell(x))$ son ensemble de voisins fréquentables :

$$V_k(\mathit{Fact}_\ell(x)) = \{z \in A^\ell : d(w,z) \leq k \text{ pour } w \in \mathit{Fact}_\ell(x)\} \ .$$

Pour construire l'ensemble $V_k(\mathit{Fact}_\ell(x))$ en temps $O(\operatorname{card} V_k(\mathit{Fact}_\ell(x)))$, on suppose que l'on dispose, pour chaque lettre $a \in A$, de la liste des lettres de l'alphabet triée dans l'ordre croissant du cout de leur substitution à a. Les éléments de ces listes sont des couples de la forme $(b, \mathit{Sub}(a,b))$. On accède au premier élément de tels objets par l'attribut *lettre*, et au second par l'attribut *cout*. Ces listes sont rangées dans une table à deux dimensions, notée L. Pour $a \in A$ et $i = 0, 1, \ldots, \operatorname{card} A - 1$, l'objet $L[a,i]$ est le couple correspondant à la $(i+1)$-ième lettre la plus proche de la lettre a. Un exemple est donné figure 7.16.

L'algorithme GÉNÉRER-VOISINS produit l'ensemble $V_t(\mathit{Fact}_k(x))$. Il fait appel à la procédure récursive GV. L'appel $\textsc{Gv}(i, \varepsilon, 0, 0, 0)$ (ligne 6 de l'algorithme) calcule tous les voisins fréquentables de $x[i\,..\,i+\ell-1]$ et les stocke dans l'ensemble implanté par la variable globale V. Au début de l'opération $\textsc{Gv}(i', v, j', p, t)$, $p = d(v[0\,..\,j'-1], x[i'-j'\,..\,i'-1]) \leq k$ et l'on essaie d'étendre v par la lettre du couple $L[x[i'], t]$ dans le cas où $j' < \ell$.

(a)

a	$L[a,0]$	$L[a,1]$	$L[a,2]$	$L[a,3]$	$L[a,4]$	$L[a,5]$
A	(A, 0)	(D, 2)	(E, 2)	(G, 3)	(K, 3)	(Q, 3)
C	(C, 0)	(Y, 2)	(A, 4)	(D, 5)	(E, 5)	(G, 5)
E	(E, 0)	(A, 2)	(D, 2)	(G, 3)	(K, 3)	(Q, 3)
G	(G, 0)	(A, 3)	(D, 3)	(E, 3)	(Q, 3)	(K, 4)
K	(K, 0)	(D, 2)	(A, 3)	(E, 3)	(Q, 3)	(R, 3)
L	(L, 0)	(Y, 3)	(A, 4)	(Q, 4)	(W, 4)	(D, 5)
Q	(Q, 0)	(D, 2)	(A, 3)	(E, 3)	(G, 3)	(K, 3)
W	(W, 0)	(R, 2)	(Y, 3)	(L, 4)	(K, 5)	(Q, 5)

a	$L[a,6]$	$L[a,7]$	$L[a,8]$	$L[a,9]$	$L[a,10]$
A	(C, 4)	(L, 4)	(R, 4)	(Y, 5)	(W, 6)
C	(K, 5)	(Q, 5)	(R, 5)	(L, 6)	(W, 7)
E	(R, 3)	(C, 5)	(L, 5)	(Y, 5)	(W, 6)
G	(C, 5)	(L, 5)	(R, 5)	(Y, 5)	(W, 6)
K	(G, 4)	(C, 5)	(L, 5)	(W, 5)	(Y, 5)
L	(E, 5)	(G, 5)	(K, 5)	(R, 5)	(C, 6)
Q	(R, 3)	(L, 4)	(C, 5)	(W, 5)	(Y, 5)
W	(A, 6)	(D, 6)	(E, 6)	(G, 6)	(C, 7)

(b)

facteur	voisins fréquentables
EAW	EAW, EAR, EDW, EEW, AAW, DAW
AWA	AWA, AWD, AWE, ARA, DWA, EWA
WAC	WAC, WAY, WDC, WEC, RAC
ACQ	ACQ, ACD, AYQ, DCQ, ECQ
CQG	CQG, CDG, YQG
QGK	QGK, QGD, DGK
GKL	GKL, GDL

Figure 7.16 Illustration pour la méthode heuristique d'alignement local. **(a)** La table L qui implante, pour chaque lettre a, les listes de couples de la forme $(b, Sub(a,b))$, pour $b \in A$, ordonnées selon la seconde composante du couple, ici sur l'alphabet formé par les lettres des mots $x =$ EAWACQGKL et $y =$ ERDAWCQPGKWY. **(b)** Les voisins fréquentables à distance maximale $k = 2$ des facteurs de x de longueur $\ell = 3$.

```
GÉNÉRER-VOISINS(ℓ)
 1  V ← ∅
 2  seuil[ℓ − 1] ← k
 3  pour i ← 0 à m − ℓ faire
 4        pour j ← ℓ − 1 à 1 pas −1 faire
 5              seuil[j − 1] ← seuil[j] − cout(L[x[i + j], 0])
 6        GV(i, ε, 0, 0, 0)
 7  retourner V

GV(i′, v, j′, p, t)
 1  si j′ = ℓ alors
 2        V ← V ∪ {v}
 3  sinonsi t < card A alors
 4        c ← L[x[i′], t]
 5        si p + cout[c] ≤ seuil[j′] alors
 6              v ← v · lettre[c]
 7              GV(i′ + 1, v[0 . . j′], j′ + 1, p + cout[c], 0)
 8              GV(i′, v[0 . . j′ − 1], j′, p, t + 1)
```

Lorsque l'on connait l'ensemble de tous les voisins fréquentables des facteurs de longueur ℓ du mot x, on peut construire un automate reconnaissant le langage défini par $V_k(\mathit{Fact}_\ell(x))$. On peut aussi le construire au fur et à mesure de la production des voisins fréquentables. Le texte y est ensuite analysé à l'aide de l'automate pour y trouver les positions des éléments de $V_k(\mathit{Fact}_\ell(x))$. La méthode utilisée repère la plus longue suite de telles positions. Elle essaie ensuite d'étendre, par programmation dynamique, vers la gauche ou vers la droite, le segment de forte similarité ainsi trouvé. On en déduit un alignement local entre x et y.

Notes

Les techniques décrites dans ce chapitre sont très employées en biologie moléculaire pour comparer des séquences provenant de chaines d'acides nucléiques (ADN ou ARN) ou d'acides aminés (protéines). Les matrices de substitution (*Sub*) les plus connues sont les matrices PAM et BLOSUM (voir Attwood et Parry-Smith [74]). Ces matrices de score, calculées empiriquement, témoignent de propriétés physico-chimiques ou évolutives des molécules étudiées. Les livres de Waterman [86] et de Setubal et Meidanis [84] constituent d'excellentes introductions aux problèmes du domaine. Le livre de Sankoff et Kruskal [83] contient de nombreuses applications des alignements.

La distance par les sous-mots de la section 7.3 est souvent attribuée à Levenshtein (1966). La notion de plus long sous-mot commun à deux mots est utilisée pour la comparaison de fichiers. La commande `diff` du

système UNIX implante un algorithme basé sur cette notion en considérant que les lignes des fichiers sont des lettres de l'alphabet. Parmi les algorithmes à la base de cette commande figurent ceux de Hunt et Szymanski (1977) (exercice 7.7) et de Myers (1986). Une présentation générale des algorithmes de recherche de sous-mots communs se trouve dans un article d'Apostolico (1997). Wong et Chandra (1976) ont montré que l'algorithme SMC-SIMPLE est optimal dans un modèle où l'on restreint l'accès aux lettres à des tests d'égalité. Sans cette condition, Hirschberg (1976) a donné une borne (inférieure) $\Omega(n \times \log n)$. Sur un alphabet borné, Masek et Paterson ont donné un algorithme fonctionnant en temps $O(n^2/\log n)$. L'extension de ce résultat au calcul général d'alignements est une question ouverte (voir Apostolico et Giancarlo, 1998).

L'algorithme initial d'alignement global, dû à Needleman et Wunsch (1970), fonctionne en temps cubique. L'algorithme de Wagner et Fischer (1974), de même que l'algorithme d'alignement local de Smith et Waterman (1981), fonctionnent en temps quadratique (voir [7], page 234). La méthode de programmation dynamique a été introduite par Bellman (1957 ; voir [92]). Sankoff (2000) discute de l'introduction de la programmation dynamique dans le traitement des séquences moléculaires.

L'algorithme SMC est dû à Hirschberg (1975). La présentation qui en est faite ici se réfère au livre de Durbin et co-auteurs [76]. Une généralisation de la méthode a été proposée par Myers et Miller (1988).

L'algorithme BRÈCHE est dû à Gotoh (1982). Un survol des méthodes d'alignement avec brèches a été présenté par Giancarlo en 1997. La preuve de la proposition 7.21 est présentée dans [84].

La méthode heuristique de la section 7.6 est à la base du logiciel Blast (voir Altschul et co-auteurs, 1990). Les paramètres ℓ et k de la section correspondent respectivement aux paramètres W (*word size*) et T (*word score threshold*) du logiciel. Le terme « voisin fréquentable » a été introduit par Mouchard (1995).

Charras et Lecroq ont créé le site [72], accessible sur la Toile, où sont disponibles des animations d'algorithmes d'alignement.

Exercices

7.1 (Distances)
Montrer que $d_{préf}$, d_{suff}, d_{fact} et d_{smot} sont des distances.

7.2 (Transposition)
Concevoir une distance entre mots qui, en plus des opérations élémentaires d'édition, retient la transposition de deux lettres consécutives. Écrire un algorithme de calcul de cette distance.

7.3 (Une colonne)
Donner une version de l'algorithme CALCUL-GÉNÉRIQUE utilisant une seule table de taille $\min\{m, n\}$ en plus des mots et d'un espace constant.

7.4 (Distingués)
Étant donnés deux mots différents x et y, donner un algorithme qui permet de trouver le plus petit sous-mot les distinguant, c'est-à-dire permettant de trouver un mot z de longueur minimale tel que, soit $z \preceq_{smot} x$ et $z \not\preceq_{smot} y$, soit $z \not\preceq_{smot} x$ et $z \preceq_{smot} y$ [*Aide :* voir Lothaire [96], chapitre 6.]

7.5 (Automate)
Donner une méthode pour produire l'automate des alignements optimaux entre deux mots x et y à partir de la table T de la section 7.2 en n'utilisant qu'un espace supplémentaire linéaire (par opposition à l'algorithme AUTO-ALIGN-OPT qui utilise la table E de taille $O(|x| \times |y|)$). [*Aide :* mémoriser une liste de sommets courants appartenant à une ou deux antidiagonales consécutives.]

7.6 (Alternative)
Il existe une autre méthode que celle qui est utilisée par l'algorithme SMC (section 7.3) pour trouver l'indice k de l'équation (7.2). Cette méthode consiste à calculer les valeurs de la dernière colonne C_1 de la table T pour x et $y[0\,..\,\lfloor n/2 \rfloor - 1]$ et de calculer les valeurs de la dernière colonne C_2 de la table pour le renversé de x et le renversé de $y[\lfloor n/2 \rfloor\,..\,n-1]$. L'indice k est alors une valeur telle que $-1 \leq k \leq m-1$ qui maximise la somme $C_1[k] + C_2[m-2-k]$.

Écrire l'algorithme qui calcule un plus long sous-mot commun à deux mots, en espace linéaire, en utilisant cette méthode. [*Aide :* voir Hirschberg (1975).]

7.7 (Boulier)
Il existe une méthode pour calculer un plus long sous-mot commun à deux mots x et y efficace lorsque x et y ont peu de lettres communes. Les lettres de y sont traitées séquentiellement de la première à la dernière. Considérons la situation où $y[0\,..\,j-1]$ a déjà été traité. L'algorithme maintient une partition des positions sur x en classes $I_0, I_1, \ldots, I_k, \ldots$ définies par :

$$I_k = \{i : smc(x[0\,..\,i], y[0\,..\,j-1]) = k\} \ .$$

En d'autres termes, les positions dans la classe I_k correspondent aux préfixes de x qui ont un plus long sous-mot commun de longueur k avec $y[0\,..\,j-1]$.

Le traitement de $y[j]$ consiste alors à considérer les positions ℓ sur x dans l'ordre décroissant telles que $x[\ell] = y[j]$. Soient ℓ une telle position et I_k sa classe. Si $\ell - 1$ appartient également à la classe I_k, on fait glisser

toutes les positions supérieures ou égales à ℓ de I_k vers la classe I_{k+1} (imaginer un boulier où chaque boule représente une position sur x et où chaque regroupement de boules représente une classe).

Implanter cette méthode pour calculer un plus long sous-mot commun à deux mots. Montrer que l'on peut la réaliser en temps $O(m \times n \times \log m)$ et espace $O(m)$. Donner une condition sur x et y réduisant le temps à $O(m + n \times \log m)$. [*Aide :* voir Hunt et Szymanski (1977).]

7.8 (Automate des sous-mots)
Donner le nombre d'états et de flèches de l'automate $\mathcal{SM}(x)$, automate minimal reconnaissant $SMot(x)$, l'ensemble des sous-mots de x ($x \in A^*$).

Écrire un algorithme séquentiel de calcul de $\mathcal{SM}(x)$, puis un second algorithme de calcul examinant le mot de droite à gauche. Quel est la complexité des deux algorithmes ?

Comment et avec quelle complexité peut-on calculer à l'aide des automates $\mathcal{SM}(x)$ et $\mathcal{SM}(y)$, $x, y \in A^*$, un plus court sous-mot distinguant x et y, s'il existe, ou un plus long sous-mot commun à ces deux mots ?

7.9 (Trois mots)
Écrire un algorithme pour aligner trois mots en espace quadratique.

7.10 (Sous-mot restreint)
Soit $x \in A^*$ un mot et soit $u_0u_1 \dots u_{r-1}$ une factorisation de x avec $u_j \in A^*$ pour $j = 0, 1, \dots, r-1$. Un mot z de longueur k est un sous-mot restreint de x muni de sa factorisation $u_0u_1 \dots u_{r-1}$ s'il existe une suite strictement croissante $\langle p_0, p_1, \dots, p_{k-1} \rangle$ de positions sur x telles que :

- $x[p_i] = z[i]$ pour $i = 0, 1, \dots, k-1$;
- si deux positions p_i et $p_{i'}$ sont telles que :
  $|u_0u_1 \dots u_{j-1}| < p_i, p_{i'} \leq |u_0u_1 \dots u_j|$
  pour $j = 1, 2, \dots, r-1$, alors $z[i] \neq z[i']$. Cela signifie que deux lettres égales d'un u_j ne peuvent pas intervenir dans le sous-mot restreint.

Un mot z est un plus long sous-mot restreint d'un mot x factorisé en $u_0u_1 \dots u_{r-1}$ et d'un mot y factorisé en $v_0v_1 \dots v_{s-1}$ si z est un sous-mot restreint de x, z est un sous-mot restreint de y et la longueur de z est maximale.

Donner un algorithme permettant de trouver un plus long sous-mot restreint commun à deux mots factorisés x et y. [*Aide :* voir Andrejková (1998).]

7.11 (Voisins moins fréquentables)
Écrire un algorithme de construction d'un automate déterministe reconnaissant les voisins fréquentables considérés en section 7.6.

Généraliser la notion de voisins fréquentables obtenue en considérant les trois opérations d'édition (et pas seulement la substitution). Écrire un programme d'alignement local associé.

8 Motifs approchés

On s'intéresse dans ce chapitre à la recherche approchée de mots fixes. Plusieurs notions d'approximation sur les mots sont considérées : jokers, différences et inégalités. Les jokers constituent une façon de représenter l'alphabet. Les solutions apportées à ce type de problème font appel à des méthodes spécifiques décrites dans la section 8.1.

De manière plus générale, la recherche approchée consiste à localiser toutes les occurrences des facteurs approchés d'un mot x au sein d'un texte y. Il s'agit de produire les positions des facteurs de y qui sont à distance au plus k de x, pour un entier naturel k donné. On suppose dans la suite que $k < m \leq n$. Nous considérons deux distances pour mesurer l'approximation : la distance d'édition et la distance de Hamming.

La distance d'édition entre deux mots u et v, qui ne sont pas nécessairement de même longueur, est le cout minimal des opérations d'édition élémentaires entre ces deux mots (voir section 7.1). La méthode de base pour la localisation est un prolongement naturel de la méthode d'alignement par programmation dynamique du chapitre 7. Son amélioration s'obtient avec une notion restreinte de distance obtenue en considérant le nombre minimal d'opérations d'édition plutôt que la somme de leurs couts. Avec cette distance, le problème est connu sous le nom de recherche approchée avec k différences. La section 8.2 en présente plusieurs solutions.

La distance de Hamming entre deux mots u et v de même longueur est le nombre de positions en lesquelles les deux mots possèdent des lettres différentes. Avec cette distance, le problème est connu sous le nom de recherche approchée avec k inégalités. Il est traité dans la section 8.3.

On examine ensuite (section 8.4) le cas de la recherche de motifs courts pour lesquels on étend le modèle vecteur-binaire de la section 1.5. Celui-ci donne d'excellents résultats pratiques et est très souple dès lors que les conditions de son utilisation sont remplies.

On aborde pour terminer (section 8.5) une méthode heuristique permettant de localiser rapidement dans un dictionnaire certaines occurrences de facteurs approchés d'un mot fixe.

8.1 Recherche de mots à jokers

Dans cette section, nous supposons que le mot x et le texte y peuvent contenir des occurrences de la lettre § dite ***joker***, lettre spéciale qui n'appartient pas à l'alphabet A. Le joker [1] s'apparie à lui-même ainsi qu'à toutes les lettres de l'alphabet A.

Plus précisément, nous définissons la notion de ***correspondance*** sur $A \cup \{\S\}$ comme suit. Deux lettres a et b de l'alphabet $A \cup \{\S\}$ se correspondent, ce que l'on note :

$$a \approx b \ ,$$

si elles sont égales ou si l'une d'entre elles au moins est le joker. On étend cette notion de correspondance aux mots : deux mots u et v sur l'alphabet $A \cup \{\S\}$ et de même longueur m se correspondent, ce que l'on note :

$$u \approx v \ ,$$

si, à chaque position, leurs lettres respectives se correspondent, c'est-à-dire si, pour $i = 0, 1, \ldots, m-1$, on a :

$$u[i] \approx v[i] \ .$$

La recherche de toutes les occurrences d'un mot à jokers x de longueur m dans un texte y de longueur n consiste à détecter toutes les positions j sur y pour lesquelles $x \approx y[j \,..\, j+m-1]$.

Jokers uniquement dans le mot

Lorsque seul le mot x contient des jokers, il est possible de résoudre le problème de sa localisation dans un texte en utilisant les mêmes techniques que celles utilisées pour la recherche d'un dictionnaire (voir chapitre 2).

Supposons pour la suite que le mot x ne soit pas vide et que l'une des ses lettres au moins soit dans A. Il se décompose alors sous la forme

$$x = \S^{i_0} x_0 \S^{i_1} x_1 \ldots \S^{i_{k-1}} x_{k-1} \S^{i_k}$$

où $k \geq 1$, $i_0 \geq 0$, $i_q > 0$ pour $q = 1, 2, \ldots, k-1$, $i_k \geq 0$ et $x_q \in A^+$ pour $q = 0, 1, \ldots, k-1$. Notons X l'ensemble constitué des mots $x_0, x_1, \ldots, x_{k-1}$ (ces mots ne sont pas nécessairement tous distincts). Puis, soit $M = \mathcal{D}(X)$ l'automate-dictionnaire de X (voir section 2.2) dont les sorties sont définies par : la sortie de l'état u est l'ensemble des positions droites des occurrences dans x des mots x_q qui sont des suffixes de u.

1. Ajoutons que plusieurs jokers distincts peuvent être considérés. Mais l'hypothèse est que, du point de vue de la recherche, ils ne se distinguent pas.

L'algorithme de localisation utilise l'automate M afin d'analyser le texte y. De plus, un compteur est associé à chaque position sur le texte, la valeur initiale du compteur étant nulle. Lorsqu'une occurrence d'un facteur x_q est découverte à une position droite j sur y, les compteurs associés aux positions $j-p$ pour lesquelles p est un élément de la sortie courante sont incrémentés. Lorsqu'un compteur à une position ℓ du texte atteint la valeur k, c'est une indication que x apparait à la position (gauche) ℓ sur y. Le code suivant s'en déduit.

L-JOKER$(M, m, k, i_0, i_k, y, n)$
1 **pour** $j \leftarrow -m+1$ **à** $n-1$ **faire**
2 $\quad C[j] \leftarrow 0$
3 $r \leftarrow \mathit{initial}[M]$
4 **pour** $j \leftarrow i_0$ **à** $n - i_k$ **faire**
5 $\quad r \leftarrow$ CIBLE$(r, y[j])$
6 $\quad$ **pour** chaque $p \in \mathit{sortie}[r]$ **faire**
7 $\quad\quad C[j-p] \leftarrow C[j-p]+1$
8 $\quad\quad$ SIGNALER-SI$(C[j-p] = k)$

On note que les valeurs $C[\ell]$ avec $\ell \leq j-m$ ne sont pas utiles lorsque la position courante sur y est j. Seuls m compteurs sont donc nécessaires au calcul. Cela permet d'énoncer le résultat suivant.

Proposition 8.1
La localisation des occurrences d'un mot à jokers x de longueur m, de la forme $x = \S^{i_0} x_0 \S^{i_1} x_1 \ldots \S^{i_{k-1}} x_{k-1} \S^{i_k}$, dans un texte de longueur n peut se faire en temps $O(k \times n)$ dans un espace $O(m)$, sous l'hypothèse que l'on dispose de l'automate $\mathcal{D}(\{x_0, x_1, \ldots, x_{k-1}\})$ muni de sorties adéquates.

Preuve D'après les résultats du chapitre 2, et si, pour l'instant, on omet la boucle aux lignes 6–8, le temps d'exécution de l'algorithme L-JOKER est $O(k \times n)$ quelle que soit l'implantation retenue pour l'automate M. Maintenant comme le nombre d'éléments de chaque sortie de l'automate est inférieur à k, la boucle aux lignes 6–8 prend un temps $O(k)$, quelle que soit la valeur de j. On obtient donc le temps total $O(k \times n)$ comme annoncé.

L'espace mémoire nécessaire à l'exécution de l'algorithme est $O(m)$, puisqu'il s'agit essentiellement de stocker m valeurs de la table C d'après la remarque qui précède l'énoncé. ∎

La phase préliminaire à l'exécution de l'algorithme L-JOKER consiste à produire l'automate $\mathcal{D}(\{x_0, x_1, \ldots, x_{k-1}\})$ avec ses sorties. Et, pour être conséquent, ce calcul doit se faire en temps $O(k \times m)$ dans un espace $O(m)$. Cela est réalisé par l'implantation de l'automate avec fonction de suppléance (voir section 2.3). Les sorties des états sont générées comme dans la section 2.2.

Jokers dans le texte et dans le mot

Le problème de la localisation de x dans y lorsque les deux mots peuvent contenir des jokers ne se résout pas dans les mêmes termes que pour une recherche classique. Cela est dû au fait que la relation $\approx$ n'est pas transitive : pour $a, b \in A$, les relations $a \approx \S$ et $\S \approx b$ n'impliquent pas nécessairement $a \approx b$. De plus, si les comparaisons de lettres (utilisant la relation $\approx$) constituent le seul accès au texte, il existe une borne minimale quadratique au problème, preuve supplémentaire que ce problème est différent des autres problèmes de recherche de motifs.

Théorème 8.2
Supposons $\operatorname{card} A \geq 2$. *Si les comparaisons de lettres constituent le seul accès au texte y, localiser toutes les occurrences d'un mot à jokers x de longueur m dans un texte à jokers y de longueur n peut demander un temps $\Omega(m \times n)$.*

Preuve La longueur m étant fixée, considérons le cas où $n = 2m$. Supposons que durant son exécution, un algorithme n'effectue pas la comparaison $x[i]$ contre $y[j]$ pour un certain $i = 0, 1, \ldots, m-1$ et un certain $j = i, i+1, \ldots, m+i$. Alors la sortie de cet algorithme est la même dans le cas $x = \S^m$ et $y = \S^{2m}$, que dans le cas $x = \S^i \mathtt{a} \S^{m-i-1}$ et $y = \S^j \mathtt{b} \S^{2m-j-1}$, bien qu'il y ait une occurrence de moins dans le second cas. Ce qui montre qu'un tel algorithme est erroné. Il s'ensuit qu'au moins $m \times (m+1)$ comparaisons doivent être effectuées.

Lorsque $n > 2m$, on factorise y en facteurs de longueur $2m$ (sauf peut-être le dernier). Le raisonnement précédent s'applique à chaque facteur et conduit à la borne de $\Omega(m^2 \times \lfloor \frac{n}{2m} \rfloor)$ comparaisons. ∎

Exposons maintenant une méthode qui permet de localiser toutes les occurrences d'un mot à jokers dans un texte à jokers en faisant appel aux vecteurs binaires. On suppose que $n \geq m \geq 1$.

Pour tous vecteurs binaires p et q d'au moins un bit, on note $p \otimes q$ le produit de p et q qui est le vecteur de $|p| + |q| - 1$ bits défini par :

$$(p \otimes q)[\ell] = \bigvee_{i+j=\ell} p[i] \wedge q[j]$$

pour $\ell = 0, 1, \ldots, |p| + |q| - 2$. Pour tout mot u sur $A \cup \{\S\}$ et toute lettre $a \in A$, on note $\lambda(u, a)$ le vecteur caractéristique des positions de a sur u défini comme le vecteur de $|u|$ bits satisfaisant :

$$\lambda(u, a)[i] = \begin{cases} \mathtt{1} & \text{si } u[i] = a \ , \\ \mathtt{0} & \text{sinon} \ , \end{cases}$$

pour $i = 0, 1, \ldots, |u| - 1$.

Maintenant, si r est le vecteur de $m + n - 1$ bits tel que :

$$r = \bigvee_{a,b \in A \text{ et } a \neq b} \lambda(y, a) \otimes \lambda(x^{\sim}, b) \ ,$$

(a)

				0	0	0	1	1	0	0	0	1	0	$\leftarrow \lambda(y, \mathtt{a})$
									0	0	1	1	0	$\leftarrow \lambda(x^\sim, \mathtt{b})$
				0	0	0	0	0	0	0	0	0	0	
			0	0	0	1	1	0	0	0	1	0		
		0	0	0	1	1	0	0	0	1	0			
	0	0	0	0	0	0	0	0	0	0				
0	0	0	0	0	0	0	0	0	0					
0	0	0	0	0	1	1	1	0	0	1	1	0	0	$\rightarrow \lambda(y, \mathtt{a}) \otimes \lambda(x^\sim, \mathtt{b})$

(b)

				0	1	0	0	0	0	1	0	0	1	$\leftarrow \lambda(y, \mathtt{b})$
									1	0	0	0	1	$\leftarrow \lambda(x^\sim, \mathtt{a})$
				0	1	0	0	0	0	1	0	0	1	
			0	0	0	0	0	0	0	0	0	0		
		0	0	0	0	0	0	0	0	0	0			
	0	0	0	0	0	0	0	0	0	0				
0	1	0	0	0	0	1	0	0	1					
0	1	0	0	0	1	1	0	0	1	1	0	0	1	$\rightarrow \lambda(y, \mathtt{b}) \otimes \lambda(x^\sim, \mathtt{a})$

(c)

0	0	0	0	0	1	1	1	0	0	1	1	0	0	$\leftarrow \lambda(y, \mathtt{a}) \otimes \lambda(x^\sim, \mathtt{b})$
0	1	0	0	0	1	1	0	0	1	1	0	0	1	$\leftarrow \lambda(y, \mathtt{b}) \otimes \lambda(x^\sim, \mathtt{a})$
0	1	0	0	0	1	1	1	0	1	1	1	0	1	$\rightarrow r$

(d)

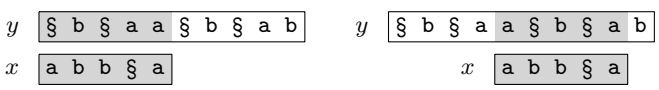


Figure 8.1 Localisation du mot à jokers $x = \mathtt{abb\S a}$ de longueur $m = 5$ dans le texte à jokers $y = \mathtt{\S b\S aa\S b\S ab}$ de longueur $n = 10$. **(a)** Calcul du produit $\lambda(y, \mathtt{a}) \otimes \lambda(x^\sim, \mathtt{b})$. **(b)** Calcul du produit $\lambda(y, \mathtt{b}) \otimes \lambda(x^\sim, \mathtt{a})$. **(c)** Calcul du vecteur binaire r de longueur $m + n - 1 = 14$, disjonction des deux vecteurs précédents. Les positions ℓ sur r comprises entre $m - 1 = 4$ et $n - 1 = 9$ pour lesquelles $r[\ell] = \mathtt{0}$ (indiquées en grisé) sont les positions droites d'une occurrence de x dans y. **(d)** Les deux occurrences de x dans y, aux positions droites 4 et 8.

on a, pour $\ell = m - 1, m, \ldots, n - 1$:

$r[\ell] = \mathtt{0}$

si et seulement si

$x \approx y[\ell - m + 1 \,..\, \ell]$.

Un exemple est montré figure 8.1.

Le temps du calcul du vecteur binaire r est $\Theta((\operatorname{card} A)^2 \times m \times n)$ si le calcul des termes $\lambda(y, a) \otimes \lambda(x^\sim, b)$ s'effectue directement sur les vecteurs binaires. Cette complexité temporelle peut toutefois être sensiblement améliorée si les produits $\otimes$ sont réalisés à l'aide d'entiers. Cette idée est développée dans la preuve du résultat qui suit.

Théorème 8.3
Les occurrences de tout mot à jokers x de longueur m dans tout texte à jokers y de longueur n peuvent être localisées en temps

$$O((\operatorname{card} A)^2 \times n \times (\log m)^2 \times \log\log m) \ .$$

Preuve Commençons par remarquer que si p et q sont deux vecteurs binaires, leur produit $p \otimes q$ peut être réalisé comme un produit de polynômes : il suffit pour cela d'associer $\vee$ à $+$, $\wedge$ à $\times$, le bit `0` à la valeur nulle, et le bit `1` à toute valeur non nulle. Ajoutons que les coefficients du polynôme ainsi associé à $p \otimes q$ sont tous inférieurs à $\min\{|p|, |q|\}$. Mais le produit des polynômes associés à p et q peut lui-même être réalisé comme le produit de deux entiers si l'on prend soin de coder les coefficients sur un nombre suffisant de bits, à savoir sur $t = \lceil \log_2(1 + \min\{|p|, |q|\}) \rceil$ bits.

Il s'ensuit que pour réaliser le produit $s = p \otimes q$, il suffit de disposer de trois entiers P de $t \times |p|$ bits, Q de $t \times |q|$ bits et S de $t \times (|p| + |q| - 1)$ bits, d'initialiser P et Q à zéro, de positionner les bits $P[t \times i]$ à `1` si $p[i] =$ `1`, de positionner les bits $Q[t \times i]$ à `1` si $q[i] =$ `1`, d'effectuer le produit $S = P \times Q$, puis de positionner les bits $s[i]$ à `1` si l'un des bits de $S[t \times i \,..\, t \times (i+1) - 1]$ est non nul et à `0` sinon. Le temps nécessaire afin de réaliser le produit est donc $O(t \times (|p| + |q|))$ pour les initialisations et positionnements, auquel il faut ajouter le temps pour effectuer le produit de deux nombres de $t \times |p|$ et $t \times |q|$ bits.

Or, l'on sait qu'il est possible de multiplier un nombre à M chiffres par un nombre à N chiffres en temps $O(N \times \log M \times \log\log M)$ pour $N \geq M$ (voir notes). Si on pose $p = \lambda(y, a)$ et $q = \lambda(x^\sim, b)$, on a $|p| = m$, $|q| = n$, $t = \lceil \log_2(m+1) \rceil$, $M = t \times m$ et $N = t \times n$. Le temps nécessaire au calcul du produit $\lambda(y, a) \otimes \lambda(x^\sim, b)$ est donc $O(n \times \log m)$ pour les initialisations et positionnements, plus $O(n \times (\log m)^2 \times \log\log m)$ pour la multiplication. Il y a $(\operatorname{card} A - 1)^2$ produits de la sorte à effectuer. Quant au calcul du vecteur binaire r, il peut se faire conjointement à celui des produits ; ce qui nécessite un temps $O((\operatorname{card} A)^2 \times n)$. La complexité totale annoncée s'en déduit. ∎

8.2 Recherche avec différences

On s'intéresse dans cette section à trouver tous les facteurs de y qui sont à une distance maximale k donnée de x. On note $m = |x|$, $n = |y|$ et l'on suppose $k \in \mathbf{N}$ et $k < m \leq n$. La distance entre deux mots est définie ici comme le nombre minimal de différences entre ces deux mots. Une différence peut être l'une des opérations d'édition : substitution, suppression ou insertion (voir section 7.1). Ce problème est connu sous le nom de ***recherche approchée avec k différences***. Il correspond à l'utilisation d'une notion simplifiée de distance d'édition. Les solutions

standard afin de résoudre le problème consistent à utiliser la technique de programmation dynamique introduite au chapitre 7. On décrit trois variations autour de cette technique.

Programmation dynamique

On examine tout d'abord un problème un peu plus général pour lequel le cout des opérations d'édition n'est pas nécessairement unitaire. Il s'agit donc de la distance d'édition ordinaire (voir chapitre 7). Aligner x avec un facteur de y revient à aligner x avec un préfixe de y en considérant que l'insertion d'un nombre quelconque de lettres de y en tête de x n'est pas pénalisant. Avec la table T de la section 7.2 on vérifie que, afin de résoudre la question, il suffit alors d'initialiser à zéro les valeurs de la première ligne de la table. Les positions des occurrences sont alors associées à toutes les valeurs de la dernière ligne de la table qui sont inférieures à k.

Afin d'effectuer la localisation de facteurs approchés, on utilise la table R définie par :

$$R[i,j] = \min\{Lev(x[0\,..\,i], y[\ell\,..\,j]) : \ell = 0, 1, \ldots, j+1\} \ ,$$

pour $i = -1, 0, \ldots, m-1$ et $j = -1, 0, \ldots, n-1$, où Lev est la distance d'édition de la section 7.1. Le calcul des valeurs de la table R utilise les relations de récurrence de la proposition qui suit.

Proposition 8.4
Pour $i = 0, 1, \ldots, m-1$ et $j = 0, 1, \ldots, n-1$, on a :

$$\begin{aligned}
R[-1,-1] &= 0 \ , \\
R[i,-1] &= R[i-1,-1] + Dél(x[i]) \ , \\
R[-1,j] &= 0 \ , \\
R[i,j] &= \min \begin{cases} R[i-1,j-1] + Sub(x[i], y[j]) \ , \\ R[i-1,j] + Dél(x[i]) \ , \\ R[i,j-1] + Ins(y[j]) \ . \end{cases}
\end{aligned}$$

Preuve Analogue à celle de la proposition 7.3. ∎

L'algorithme de localisation L-DIFF-DYN dont le code est donné ci-après et qui traduit la récurrence de la proposition précédente effectue la recherche approchée. Un exemple est donné figure 8.2.

(a)

R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0	0	0	0
0	G	1	1	1	0	1	1	1	1	0	1	0	1	1
1	A	2	2	1	1	0	1	1	1	1	0	1	0	1
2	T	3	3	2	2	1	0	1	2	2	1	1	1	1
3	A	4	4	3	3	2	1	0	1	2	2	2	1	1
4	A	5	5	4	4	3	2	1	0	1	2	3	2	1

(b)

```
(    G A T A A         )
(C A G A T - A A G A G A A)

(    G A T A A         )
(C A G A T A A G A G A A)

(    G A T A A         )
(C A G A T A - A G A G A A)

(  - G A T A A         )
(C A G A T A A G A G A A)

(      G A T A A       )
(C A G - A T A A G A G A A)

(    G A T A A -       )
(C A G A T A A G A G A A)

(              G A T A A)
(C A G A T A A G A G A A)
```

Figure 8.2 Recherche de $x =$ `GATAA` dans $y =$ `CAGATAAGAGAA` avec une différence, en considérant des couts unitaires pour les opérations d'édition. **(a)** Les valeurs de la table R. **(b)** Les sept alignements de x avec des facteurs de y se terminant aux positions 5, 6, 7 et 11 sur y. On note que les quatrième et sixième alignements n'apportent aucune information en supplément du deuxième.

L-DIFF-DYN(x, m, y, n, k)

1 $R[-1,-1] \leftarrow 0$
2 **pour** $i \leftarrow 0$ **à** $m-1$ **faire**
3 $\quad R[i,-1] \leftarrow R[i-1,-1] + \mathit{Dél}(x[i])$
4 **pour** $j \leftarrow 0$ **à** $n-1$ **faire**
5 $\quad R[-1,j] \leftarrow 0$
6 $\quad$ **pour** $i \leftarrow 0$ **à** $m-1$ **faire**
7 $$R[i,j] \leftarrow \min \begin{cases} R[i-1,j-1] + \mathit{Sub}(x[i], y[j]) \\ R[i-1,j] + \mathit{Dél}(x[i]) \\ R[i,j-1] + \mathit{Ins}(y[j]) \end{cases}$$
8 $\quad$ SIGNALER-SI$(R[m-1,j] \leq k)$

On note que l'espace utilisé par l'algorithme L-DIFF-DYN peut être réduit à une seule colonne en reproduisant la technique de la section 7.3. Cette technique est d'ailleurs implantée par l'algorithme L-DIFF-ÉLAG plus loin. En conclusion on obtient le résultat suivant.

Proposition 8.5
L'opération L-DIFF-DYN(x, m, y, n, k) *qui localise les facteurs u de y pour lesquels $Lev(u, x) \leq k$ (Lev distance d'édition avec couts quelconques) s'exécute en temps $O(m \times n)$ et peut être réalisée en espace $O(m)$.* ∎

Monotonie diagonale

Dans la suite de la section, on considère que les couts des opérations d'édition sont unitaires. C'est un exemple simple pour lequel on peut décrire des stratégies de calcul plus efficaces que celle décrite plus haut. La restriction permet d'énoncer une propriété de monotonie sur les diagonales qui est à la base des variations présentées.

Puisque nous supposons que $Sub(a, b) = Dél(a) = Ins(b) = 1$ pour $a, b \in A$, $a \neq b$, la relation de récurrence de la proposition 8.4 se simplifie et devient :

$$\begin{aligned} R[-1,-1] &= 0 \ , \\ R[i,-1] &= i+1 \ , \\ R[-1,j] &= 0 \ , \\ R[i,j] &= \min \begin{cases} R[i-1,j-1] & \text{si } x[i] = y[j] \ , \\ R[i-1,j-1]+1 & \text{si } x[i] \neq y[j] \ , \\ R[i-1,j]+1 \ , \\ R[i,j-1]+1 \ . \end{cases} \end{aligned} \tag{8.1}$$

pour $i = 0, 1, \ldots, m-1$ et $j = 0, 1, \ldots, n-1$.

Une diagonale d de la table R est constituée par les positions $[i, j]$ pour lesquelles $j - i = d$ $(-m \leq d \leq n)$. La propriété de monotonie diagonale exprime que la suite des valeurs sur chaque diagonale de la table R est croissante et que la différence entre deux valeurs consécutives est au plus d'une unité (voir figure 8.2). Avant d'énoncer formellement la propriété, nous montrons des résultats intermédiaires. Le premier signifie que deux valeurs adjacentes sur une colonne du tableau R diffèrent d'au plus une unité. Le second est le symétrique du premier sur les lignes de R.

Lemme 8.6
Pour chaque position j sur le mot y, on a :

$$-1 \leq R[i,j] - R[i-1,j] \leq 1$$

pour $i = 0, 1, \ldots, m-1$.

Preuve De la récurrence sur R énoncée plus haut on déduit, pour $i \geq 0$ et $j \geq 0$:

$$R[i,j] \geq \min \begin{cases} R[i-1,j-1] \\ R[i-1,j]+1 \\ R[i,j-1]+1 \end{cases} \tag{8.2}$$

et $R[i,j] \leq R[i-1,j]+1$. Donc $R[i,j]-R[i-1,j] \leq 1$. Ce qui prouve l'une des inégalités de l'énoncé.

L'inégalité :

$$R[i,j] \leq R[i,j-1]+1 \ , \tag{8.3}$$

qui s'obtient par symétrie, est utilisée dans la suite.

Nous montrons que $R[i,j]-R[i-1,j] \geq -1$ par récurrence sur j, pour $i \geq 0$ et $j \geq 0$. Cette propriété est satisfaite pour $j=-1$ car $R[i,-1]-R[i-1,-1] = i+1-i = 1 \geq -1$.

Supposons l'inégalité satisfaite jusqu'à $j-1$, soit :

$$R[i,j-1]+1 \geq R[i-1,j-1] \ . \tag{8.4}$$

L'équation (8.3) donne, après remplacement de i par $i-1$:

$$R[i-1,j-1] \geq R[i-1,j]-1 \ . \tag{8.5}$$

En combinant les relations (8.2), (8.4) et (8.5), on obtient :

$$R[i,j] \geq \min\{R[i-1,j]+1, R[i-1,j]-1\}$$

c'est-à-dire $R[i,j] \geq R[i-1,j]-1$, et donc $R[i,j]-R[i-1,j] \geq -1$. Cela termine la récurrence et la preuve des inégalités de l'énoncé. ∎

Lemme 8.7
Pour chaque position i sur le mot x, on a :

$$-1 \leq R[i,j]-R[i,j-1] \leq 1$$

pour $j = 0,1,\ldots,n-1$.

Preuve Symétrique de celle du lemme 8.6 en échangeant les rôles de x et y. ∎

Nous pouvons maintenant énoncer la proposition concernant la propriété de monotonie sur les diagonales annoncée plus haut.

Proposition 8.8 (monotonie sur les diagonales)
Pour $i = 0,1,\ldots,m-1$ et $j = 0,1,\ldots,n-1$, on a :

$$R[i-1,j-1] \leq R[i,j] \leq R[i-1,j-1]+1 \ .$$

Preuve D'après la relation (8.1), l'inégalité $R[i-1,j-1] \leq R[i,j]$ est valide si $R[i-1,j-1] \leq R[i-1,j]+1$ et $R[i-1,j-1] \leq R[i,j-1]+1$: c'est une conséquence des lemmes 8.6 et 8.7. De plus, l'équation (8.1) donne $R[i,j] \leq R[i-1,j-1]+1$. Le résultat annoncé s'ensuit. ∎

Calcul partiel

La propriété de monotonie sur les diagonales est exploitée de la manière suivante afin d'éviter de calculer certaines valeurs dans la table R qui sont supérieures à k, le nombre maximal de différences autorisées. Les valeurs sont toujours calculées colonne par colonne, dans l'ordre croissant des positions sur y et pour chaque colonne dans l'ordre croissant des positions sur x, à l'instar de l'algorithme L-DIFF-DYN. Lorsqu'une valeur égale à $k+1$ est trouvée dans une colonne, il est inutile de calculer les valeurs suivantes dans la même diagonale puisque celles-ci sont toutes strictement supérieures à k d'après la proposition 8.8. Pour élaguer le calcul, on garde trace, dans chaque colonne, de la position la plus basse à laquelle se trouve une valeur admissible. Si q_j est cette position, pour une colonne j donnée, seules les valeurs des lignes -1 à q_j+1 sont calculées dans la colonne suivante (d'indice $j+1$).

L'algorithme L-DIFF-ÉLAG ci-dessous réalise cette méthode.

```
L-DIFF-ÉLAG(x, m, y, n, k)
 1  pour i ← −1 à k − 1 faire
 2          C1[i] ← i + 1
 3  p ← k
 4  pour j ← 0 à n − 1 faire
 5          C2[−1] ← 0
 6          pour i ← 0 à p faire
 7                  si x[i] = y[j] alors
 8                          C2[i] ← C1[i − 1]
 9                  sinon C2[i] ← min{C1[i − 1], C2[i − 1], C1[i]} + 1
10          C1 ← C2
11          tantque C1[p] > k faire
12                  p ← p − 1
13          SIGNALER-SI(p = m − 1)
14          p ← min{p + 1, m − 1}
```

La colonne -1 est initialisée jusqu'à la ligne $k-1$ qui correspond à la valeur k. Pour les colonnes suivantes d'indice $j = 0, 1, \ldots, n-1$, les valeurs sont calculées jusqu'à la ligne

$$p_j = \min \begin{cases} 1 + \max\{i : 0 \leq i \leq m-1 \text{ et } R[i, j-1] \leq k\} \ , \\ m - 1 \ . \end{cases}$$

La table R est implantée à l'aide de deux tables C_2 et C_1 qui permettent de mémoriser respectivement les valeurs de la colonne en cours de calcul et les valeurs de la colonne précédente. Le procédé est celui qui est utilisé dans l'algorithme SMC-COLONNE de la section 7.3. À chaque itération de la boucle aux lignes 6–9, on a :

$$C_1[i-1] \quad = \quad R[i-1, j-1] \ ,$$

R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A
-1	$x[i]$	0	0	0	0	0	0	0	0	0	0	0	0	0
0	G	1	1	1	0	1	1	1	1	0	1	0	1	1
1	A		2	1	1	0	1	1	1	1	0	1	0	1
2	T				2	1	0	1	2	2	1	1	1	1
3	A						1	0	1	2	2	2	1	1
4	A							1	0	1	2			1

Figure 8.3 Élagage du calcul de la table de programmation dynamique pour la recherche de $x =$ GATAA dans $y =$ CAGATAAGAGAA avec une différence (voir figure 8.2). On constate que dix-sept valeurs de la table R (celles qui n'apparaissent pas) ne sont pas utiles pour le calcul des occurrences de facteurs approchés de x dans y.

$$\begin{aligned} C_2[i-1] &= R[i-1,j] \ , \\ C_1[i] &= R[i,j-1] \ . \end{aligned}$$

On calcule alors la valeur $C_2[i]$ qui est aussi $R[i,j]$. On retrouve donc à cette ligne une implantation de la relation (8.1). Un exemple de calcul est donné figure 8.3.

On note que l'espace mémoire utilisé par l'algorithme est $O(m)$. En effet, seules deux colonnes sont mémorisées. Cela est rendu possible car le calcul des valeurs d'une colonne n'a besoin que de celles de la colonne précédente.

Calcul diagonal

La variante de recherche avec différences que nous considérons maintenant consiste à calculer les valeurs de la table R suivant les diagonales et en tenant compte de la propriété de monotonie. Les positions intéressantes sur les diagonales sont celles où se produisent des changements de valeurs, qui sont des incrémentations à cause de la distance choisie.

Pour un nombre de différences q et une diagonale d, on note $L[q,d]$ l'indice i de la ligne sur laquelle $R[i,j] = q$ pour la dernière fois sur la diagonale $j-i = d$. L'idée de la définition de $L[q,d]$ est montrée figure 8.4. Formellement, pour $q = 0, 1, \ldots, k$ et $d = -m, -m+1, \ldots, n-m$, on a :

$L[q,d] = i$

si et seulement si i est l'indice maximal, $-1 \leq i < m$, pour lequel il existe un indice j, $-1 \leq j < n$, avec

$R[i,j] \leq q$ et $j - i = d$.

En d'autres termes, à q fixé, les valeurs $L[q,d]$ marquent la frontière inférieure des valeurs inférieures à q dans la table R (valeurs grisées dans la figure 8.5).

R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A
-1	$x[i]$						0							
0	G							1						
1	A								1					
2	T									2				
3	A										2			
4	A											3		

Figure 8.4 Valeurs de la table R sur la diagonale 5 pour la recherche approchée de $x =$ GATAA dans $y =$ CAGATAAGAGAA. Les dernières occurrences de chaque valeur sur la diagonale sont grisées. Les lignes où elles se trouvent sont mémorisées par la table L dans l'algorithme de calcul diagonal. On a ainsi $L[0,5] = -1$, $L[1,5] = 1$, $L[2,5] = 3$, $L[3,5] = 4$.

La définition de $L[q,d]$ implique que q soit le plus petit nombre de différences entre $x[0\,..\,L[q,d]]$ et un facteur du texte se terminant à la position $d+L[q,d]$ sur y. Elle implique en plus que les lettres $x[L[q,d]+1]$ et $y[d+L[q,d]+1]$ soient différentes lorsqu'elles sont définies.

Les valeurs $L[q,d]$ sont calculées par itération sur d pour q variant de 0 à $k+1$. Le principe du calcul repose sur la récurrence (8.1) et les énoncés plus haut. Une simulation du calcul sur la table R est présentée figure 8.5.

Pour le problème de la recherche approchée avec k différences, seules sont nécessaires les valeurs $L[q,d]$ pour lesquelles $q \leq k$. Si $L[q,d] = m-1$, cela signifie qu'il y a une occurrence du mot à la diagonale d avec au plus q différences. L'occurrence se terminant à la position $d+m-1$, ceci n'est valide que si $d+m \leq n$. On obtient d'autres occurrences approchées à la fin de y lorsque $L[q,d] = i$ et $d+i = n-1$; dans ce cas le nombre de différences est de $q+m-1-i$.

L'algorithme L-DIFF-DIAG effectue la recherche approchée de x dans y en calculant les valeurs $L[q,d]$. Remarquons que la première occurrence possible d'un facteur approché de x dans y peut se terminer à la position $m-1-k$ sur y, ce qui correspond à la diagonale $-k$. La dernière occurrence possible commence à la position $n-m+k$ sur y, ce qui correspond à la diagonale $n-m+k$. Donc seules les diagonales allant de $-k$ à $n-m+k$ sont considérées durant le calcul (l'initialisation porte aussi sur les diagonales $-k-1$ et $n-m+k+1$ afin de simplifier l'écriture de l'algorithme). La figure 8.6 montre la table L obtenue sur l'exemple de la figure 8.2.

(a)

R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A
-1	$x[i]$	0	0	0	0	0	0	0	0	0				
0	G				0					0				
1	A					0					0			
2	T						0							
3	A							0						
4	A								0					

(b)

R	j	-1	0	1	2	3	4	5	6	7	8	9	10	11
i		$y[j]$	C	A	G	A	T	A	A	G	A	G	A	A
-1	$x[i]$	0	0	0	0	0	0	0	0	0				
0	G	1	1	1	0	1	1	1	1	0				
1	A			1	1	0	1	1	1	1	0			
2	T					1	0	1			1	1		
3	A						1	0	1				1	
4	A							1	0	1				1

Figure 8.5 Simulation du calcul diagonal pour la recherche de $x =$ GATAA dans $y =$ CAGATAAGAGAA avec une différence (voir figure 8.2). **(a)** Valeurs calculées pendant la première étape (lignes 7–11 pour $q = 0$ de l'algorithme L-DIFF-DIAG) ; elle détectent l'occurrence de x à la position droite 6 sur y (car $R[4, 6] = 0$). **(b)** Valeurs calculées au cours de la seconde étape (lignes 7–11 pour $q = 1$) ; elles indiquent les facteurs approchés de x avec une différence aux positions droites 5, 7 et 11 sur y (car $R[4, 5] = R[4, 7] = R[4, 11] = 1$).

d	-2	-1	0	1	2	3	4	5	6	7	8	9
$q = -1$		-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
$q = 0$	-1	-1	-1	-1	4	-1	-1	-1	-1	1	-1	
$q = 1$		0	1	4	4	4	1	1	2	4		

Figure 8.6 Valeurs de la table L du calcul diagonal lorsque $x =$ GATAA, $y =$ CAGATAAGAGAA et $k = 1$. Les lignes $q = 0$ et $q = 1$ correspondent à un état du calcul tel que simulé sur la table R dans la figure 8.5. Les valeurs $4 = |\text{GATAA}| - 1$ sur la ligne $q = 1$ indiquent la présence d'occurrences de x avec au plus une différence se terminant aux positions $1 + 4$, $2 + 4$, $3 + 4$ et $7 + 4$ sur y.

L-DIFF-DIAG(x, m, y, n, k)

```
 1  pour d ← −1 à n − m + k + 1 faire
 2        L[−1, d] ← −2
 3  pour q ← 0 à k − 1 faire
 4        L[q, −q − 1] ← q − 1
 5        L[q, −q − 2] ← q − 1
 6  pour q ← 0 à k faire
 7        pour d ← −q à n − m + k − q faire
                    ⎧ L[q − 1, d − 1]
 8                ℓ ← max ⎨ L[q − 1, d] + 1
                    ⎩ L[q − 1, d + 1] + 1
 9                ℓ ← min{ℓ, m − 1}
10                L[q, d] ← ℓ
                     + |lpc(x[ℓ + 1 .. m − 1], y[d + ℓ + 1 .. n − 1])|
11                SIGNALER-SI(L[q, d] = m − 1 ou d + L[q, d] = n − 1)
```

Lemme 8.9
L'algorithme L-DIFF-DIAG *calcule la table* L.

Preuve Montrons que $L[q, d]$ est correctement calculé en supposant que toutes les valeurs de la ligne $q - 1$ de L sont exactes. Soient i la valeur de ℓ calculée à la ligne 8 et $j = d + i$.

Il se peut que $i = m$ si $i = L[q-1, d]+1$ ou $i = L[q-1, d+1]+1$. Dans le premier cas, on a $R[i, j] \leq q - 1$ par hypothèse de récurrence et donc aussi $R[i, j] \leq q$, ce qui donne $L[q, d] = i$ comme effectué par l'algorithme après l'instruction à la ligne 9. Dans le second cas, on a aussi $L[q, d] = i$ par le lemme 8.6, et l'algorithme effectue le calcul correctement.

Dans chacun des trois cas qui se présentent avec $i < m$, on remarque que $R[i, j] \geq q$ car la maximalité de i entraine que $R[i, j]$ n'a pas été calculé auparavant. Si $i = L[q-1, d-1]$, le fait que $R[i, j] = q$ résulte du lemme 8.6. Si $i = L[q - 1, d + 1] + 1$, l'égalité provient du lemme 8.7, et enfin si $i = L[q - 1, d] + 1$ elle vient de la monotonie diagonale. L'indice de ligne maximal cherché est obtenu après l'instruction à la ligne 10 en conséquence de la relation de récurrence (8.1) sur R.

On termine la récurrence sur q en vérifiant que la table L est correctement initialisée. ∎

Proposition 8.10
Pour tout mot x de longueur m, tout mot y de longueur n et tout entier k tel que $k < m \leq n$, l'opération L-DIFF-DIAG(x, m, y, n, k) *calcule les occurrences approchées de x dans y avec au plus k différences.*

Preuve D'après le lemme précédent, la table calculée par l'algorithme est la table L.

Si $L[q, d] = m - 1$, par définition de L, $R[m - 1, d + m - 1] \leq q$. Par définition de R, cela signifie que x possède une occurrence approchée

à la diagonale d avec au plus q différences. Les occurrences signalées sous cette condition à la ligne 11 sont donc correctes puisque $q \leq k$. Si $d + L[q, d] = n - 1$, l'algorithme signale une occurrence approchée de x à la diagonale d. Le nombre de différences est inférieur (ou égal) à $q+m-1-L[q, d]$, ce qui est $q+m-1+d-n+1$, soit $q+m+d-n$. Comme $d \leq n-m+k-q$ (ligne 7), on obtient un nombre de différences inférieur à $q+m-n+n-m+k-q = k$ comme souhaité. Les occurrences signalées sous cette seconde condition à la ligne 11 sont donc aussi correctes.

Réciproquement, une occurrence approchée de x dans y avec k différences se repère sur la table R lorsque l'une des conditions $R[m-1, j] \leq k$ ou $R[i, n-1] + m - 1 - i \leq k$ est satisfaite. La première équivaut à $L[k, j - m + 1] = m - 1$, et l'algorithme la signale à la ligne 11. Pour la deuxième, en notant $q = R[i, n-1]$, on a, par définition de L, $L[q, n-1-i] = i$ et donc $n-1-i+L[q, n-1-i] = n-1$. L'occurrence est donc signalée si $q \leq k$, ce qui est immédiat d'après l'inégalité ci-dessus, et si la diagonale est examinée, c'est-à-dire lorsque $n-1-i \leq n-m+k-q$. L'inégalité équivaut à $q + m - 1 - i \leq k$, ce qui montre que la deuxième condition est satisfaite. Cela termine la preuve. ∎

Tel que l'algorithme L-DIFF-DIAG est décrit, l'espace mémoire pour le calcul est utilisé principalement par la table L. On note qu'il suffit d'en mémoriser une seule ligne afin d'effectuer le calcul correctement, ce qui donne une implantation en espace $O(n)$. Il est toutefois possible de réduire l'espace à $O(m)$ pour obtenir un espace comparable à celui de l'algorithme L-DIFF-ÉLAG (voir exercice 8.5).

Temps d'exécution du calcul diagonal

La méthode de calcul diagonal trouve son intérêt dans la mise en évidence d'évaluations de plus longs préfixes communs. Lorsque ceux-ci sont calculés par simples comparaisons de lettres à chaque appel, l'algorithme n'est pas plus rapide que les précédents. C'est le résultat de la proposition qui suit. Mais une préparation des mots x et y permet d'implanter le calcul des plus longs préfixes communs de sorte que chacun s'exécute en temps constant. On obtient alors le résultat énoncé dans le théorème 8.12. De façon schématique sur l'exemple de la figure 8.5, la première implantation prend un temps proportionnel au nombre de valeurs qui figurent dans la seconde table, alors que la deuxième implantation prend un temps proportionnel au nombre de valeurs grisées.

Proposition 8.11
Si le calcul de $lpc(u, v)$ est réalisé en temps $O(|lpc(u, v)|)$, l'algorithme L-DIFF-DIAG *s'exécute en temps $O(m \times n)$.*

Preuve La preuve repose sur la constatation que, si le plus long préfixe commun calculé à la ligne 10 est de longueur $p > 0$, les instructions de la

boucle (aux lignes 8–11) reviennent à définir $p+1$ nouvelles valeurs dans la table R. Le temps cumulé de ces calculs de plus longs préfixes communs est donc $O(m \times n)$. Les autres instructions de la boucle s'exécutent en temps constant (y compris les calculs de *lpc* qui produisent le mot vide). Comme les instructions sont exécutées $(k+1) \times (n-m+k+1)$ fois, elles prennent le temps global $O(k \times n)$. Par conséquent, le calcul complet se fait en temps $O(m \times n)$. ∎

La preuve précédente met en relief le fait que si le calcul de $lpc(u, v)$ peut se faire en temps constant, l'algorithme L-DIFF-DIAG s'exécute en temps $O(k \times n)$. En fait, il est possible de préparer les mots x et y de façon à obtenir cette condition. Pour cela, on utilise l'arbre des suffixes, $\mathcal{A}_{\mathrm{C}}(Suff(z))$, du mot $z = x\$y$ où $\$ \notin alph(y)$ (voir chapitre 5). Le mot

$$w = lpc(x[\ell+1\,..\,m-1], y[d+\ell+1\,..\,n-1])$$

n'est autre que le mot $lpc(x[\ell+1\,..\,m-1]\$y, y[d+\ell+1\,..\,n-1])$ puisque $\$ \notin alph(y)$. Soient f et g les nœuds externes de l'arbre $\mathcal{A}_{\mathrm{C}}(Suff(z))$ associés aux suffixes de $x[\ell+1\,..\,m-1]\$y$ et $y[d+\ell+1\,..\,n-1]$ du mot z. Leur préfixe commun de longueur maximale est alors l'étiquette du chemin conduisant de l'état initial au plus profond nœud qui est un ancêtre commun à f et g. Cela ramène le calcul de w à celui de ce nœud.

Le problème de l'ancêtre commun qui nous intéresse ici est celui pour lequel l'arbre est statique. Un prétraitement linéaire de l'arbre permet d'obtenir une réponse en temps constant aux requêtes concernées (voir notes). La conséquence de ce résultat est le théorème suivant.

Théorème 8.12
Sur un alphabet fixe, après préparation des mots x et y en temps linéaire, il est possible d'exécuter l'algorithme L-DIFF-DIAG *en temps $O(k \times n)$.*

Preuve La préparation consiste d'abord en la construction de l'arbre des suffixes $\mathcal{A}_{\mathrm{C}}(Suff(z))$ du mot $z = x\$y$, puis en la préparation de l'arbre afin de répondre en temps constant à chaque recherche de plus profond ancêtre commun à deux de ses nœuds externes. On associe aussi à chaque nœud de l'arbre la longueur de ce nœud (rappelons que les nœuds de l'arbre sont des facteurs de z). Le temps de préparation total est linéaire puisque l'alphabet est fixe (voir chapitre 5 et notes).

Le calcul de $|lpc(x[\ell+1\,..\,m-1], y[d+\ell+1\,..\,n-1])|$ pendant l'exécution de l'algorithme L-DIFF-DIAG peut alors être réalisé en temps constant. Il s'ensuit, en reprenant la preuve de la proposition précédente, que le temps d'exécution global est $O(k \times n)$. ∎

8.3 Recherche avec inégalités

Dans cette section, on s'intéresse à la localisation de toutes les occurrences d'un mot x de longueur m dans un mot y de longueur n avec au plus k inégalités ($k \in \mathbf{N}$, $k < m \leq n$). On rappelle du chapitre 7 que la distance de Hamming entre deux mots u et v de même longueur, nombre d'inégalités entre u et v, est définie par :

$$Ham(u, v) = \text{card}\{i : u[i] \neq v[i], i = 0, 1, \ldots, |u| - 1\} \ .$$

Le problème de la localisation peut alors s'exprimer comme la recherche de toutes les positions $j = 0, 1, \ldots, n - m$ sur y qui satisfont l'inégalité $Ham(x, y[j \,..\, j + m - 1]) \leq k$.

Automate de recherche

Une solution naturelle au problème de la recherche consiste à utiliser un automate qui reconnait le langage $A^*\{w : Ham(x, w) \leq k\}$. Cela reprend la méthode développée dans le chapitre 2. Pour ce faire, on peut considérer l'automate non déterministe défini comme suit :

- chaque état est une paire (ℓ, i) où ℓ est le niveau de l'état et i est sa profondeur, avec $0 \leq \ell \leq k$, $-1 \leq i \leq m - 1$ et $\ell \leq i + 1$;
- l'état initial est $(0, -1)$;
- les états terminaux sont de la forme $(\ell, m - 1)$ avec $0 \leq \ell \leq k$;
- les flèches sont, pour $0 \leq \ell \leq k$, $0 \leq i < m - 1$ et $a \in A$, soit de la forme $((0, -1), a, (0, -1))$, soit de la forme $((\ell, i), x[i + 1], (\ell, i + 1))$, soit de la forme $((\ell, i), a, (\ell + 1, i + 1))$ si $a \neq x[i + 1]$ et $0 \leq \ell \leq k - 1$.

L'automate possède $k + 1$ niveaux, chacun des niveaux ℓ permettant de reconnaitre les préfixes de x avec ℓ inégalités. Les flèches de la forme $((\ell, i), a, (\ell, i + 1))$ correspondent à l'égalité de lettres alors que celles de la forme $((\ell, i), a, (\ell + 1, i + 1))$ correspondent à l'inégalité de lettres. La boucle sur l'état initial permet de localiser toutes les occurrences des facteurs cherchés. Lors de l'analyse du texte par l'automate, si un état terminal $(\ell, m - 1)$ est atteint, cela indique la présence d'une occurrence de x avec exactement ℓ inégalités.

Il est clair que l'automate possède $(k + 1) \times (m + 1 - \frac{k}{2})$ états et qu'il peut être construit en temps $O(k \times m)$. Un exemple est montré figure 8.7. Malheureusement, le nombre total d'états obtenus en déterminisant l'automate est

$$\Theta(\min\{m^{k+1}, (k + 1)!(k + 2)^{m-k+1}\})$$

(voir notes), et aucune des méthodes indiquées dans le chapitre 2 ne permet de réduire simplement la taille de la représentation de l'automate.

On peut vérifier qu'une simulation directe de l'automate produit un algorithme de recherche dont le temps d'exécution est $O(m \times n)$ en utilisant la programmation dynamique comme dans le chapitre précédent.

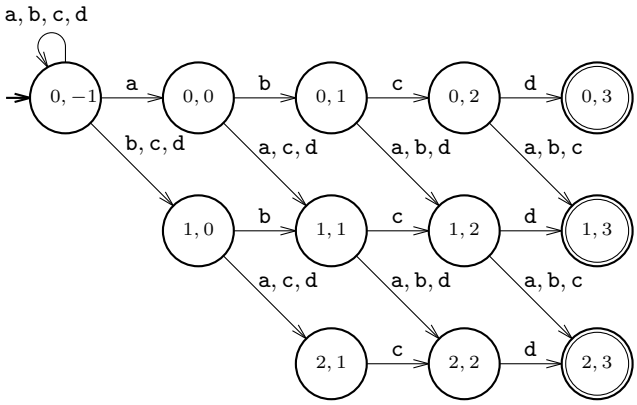


Figure 8.7 L'automate (non déterministe) de recherche approchée avec deux inégalités pour le mot `abcd` sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$.

En fait en utilisant une méthode adaptée au problème on obtient, dans la suite, un algorithme qui effectue la recherche en temps $O(k \times n)$. Cela produit une solution de même complexité que celle fournie par l'algorithme L-DIFF-DIAG qui résout pourtant un problème plus général. Mais la solution qui suit est basée sur une simple gestion de listes sans avoir recours à un algorithme de recherche d'ancêtre commun.

Implantation spécifique

On montre comment réduire le temps d'exécution de la simulation de l'automate précédent. Afin d'obtenir le temps désiré, on utilise durant la recherche une file F de positions qui localisent des inégalités détectées. Sa mise à jour se fait par comparaisons de lettres, mais aussi par fusion avec des files associées au mot x. Les suites qu'elles représentent sont définies comme suit.

Pour un décalage q de x, $1 \leq q \leq m-1$, $G[q]$ est la suite croissante, de longueur maximale $2k+1$, des positions sur x des inégalités les plus à gauche entre $x[q\,..\,m-1]$ et $x[0\,..\,m-q-1]$. Les suites sont déterminées lors d'une phase de prétraitement qui est décrite à la fin de la section.

La phase de recherche consiste à effectuer des tentatives à toutes les positions $j = 0, 1, \ldots, n-m$ sur y. Pendant la tentative à la position j, on examine le facteur $y[j\,..\,j+m-1]$ du texte et la situation générique est la suivante (voir figure 8.8) : le préfixe $y[j\,..\,g]$ de la fenêtre a déjà été examiné lors d'une tentative précédente à la position f, $f < j$, et aucune comparaison n'a encore eu lieu sur le suffixe $y[g+1\,..\,n-1]$ du texte. Le procédé utilisé ici s'apparente à celui réalisé par l'algorithme PRÉFIXES de la section 1.6. La différence intervient ici lors de la comparaison de la partie du texte déjà examinée, $y[j\,..\,g]$, car il n'est plus possible de conclure à l'aide d'un simple test. En effet, environ k tests peuvent

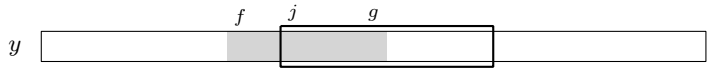


Figure 8.8 Variables de l'algorithme L-INÉGALITÉS. Lors de la tentative à la position j, les variables f et g repèrent une tentative précédente. Les inégalités entre $y[f\,..\,g]$ et $x[0\,..\,g-f]$ sont mémorisées dans la file F.

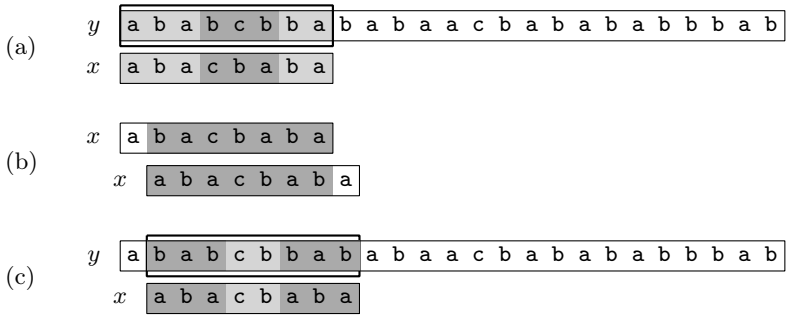


Figure 8.9 Recherche avec inégalités du mot $x =$ `abacbaba` dans le texte $y =$ `ababcbbabababaacbabababbbab`. **(a)** Occurrence du mot avec exactement trois inégalités en position 0 sur y. La file F des inégalités contient les positions 3, 4 et 5 sur x. **(b)** Décalage de longueur 1. Il y a sept inégalités entre $x[0\,..\,6]$ et $x[1\,..\,7]$, ce qui correspond au fait que $G[1]$ contient la suite $\langle 1, 2, 3, 4, 5, 6, 7\rangle$ (voir figure 8.10). **(c)** Tentative à la position 1 : le facteur $y[1\,..\,7]$ a déjà été considéré mais la lettre $y[8] =$ `b` n'a encore jamais été comparée. Les inégalités aux positions 0, 1, 5 et 6 sur x se déduisent de la fusion des files F et $G[1]$. Trois comparaisons de lettres sont nécessaires aux positions 2, 3 et 4 pour trouver l'inégalité en 2 car ces trois positions sont simultanément dans F et $G[1]$. Une comparaison supplémentaire fournit l'inégalité en position 7.

s'avérer nécessaires pour faire la comparaison. La figure 8.9 montre un exemple de calcul.

Les positions des inégalités détectées lors de la tentative à la position f sont mémorisées dans une file F. Leur calcul se fait en examinant les positions en ordre croissant. Pour la recherche avec k inégalités, on n'en retient dans F qu'au plus $k+1$ (les plus à gauche). Considérer une éventuelle $k+1$-ième inégalité revient à mémoriser le plus long préfixe de x qui possède exactement k inégalités avec le facteur aligné de y.

Le code de l'algorithme de recherche avec inégalités, L-INÉGALITÉS, est donné ci-dessous. Le traitement à la position j procède en deux étapes. Il commence tout d'abord par comparer les facteurs $x[0\,..\,g-j]$ et $y[j\,..\,g]$ en utilisant les files F et $G[j-f]$. La comparaison revient à faire une fusion de ces deux files (ligne 7) ; cette dernière est décrite par la suite. La seconde étape n'est appliquée que dans l'éventualité où la suite ainsi obtenue contient moins de k positions. Elle poursuit l'examen de la fenêtre par simples comparaisons de lettres (lignes 10–16). C'est

durant cette étape qu'une occurrence d'un facteur approché peut être détectée.

```
L-INÉGALITÉS(x, m, G, y, n, k)
 1  F ← FILE-VIDE()
 2  (f, g) ← (−1, −1)
 3  pour j ← 0 à n − m faire
 4       si LONGUEUR(F) > 0 et TÊTE(F) = j − f − 1 alors
 5            DÉFILER(F)
 6       si j ≤ g alors
 7            J ← INÉG-FUSION(f, j, g, F, G[j − f])
 8       sinon J ← FILE-VIDE()
 9       si LONGUEUR(J) ≤ k alors
10            F ← J
11            f ← j
12            faire g ← g + 1
13                 si x[g − j] ≠ y[g] alors
14                      ENFILER(F, g − j)
15            tantque LONGUEUR(F) ≤ k et g < j + m − 1
16            SIGNALER-SI(LONGUEUR(F) ≤ k)
```

Un exemple de table G et de valeurs successives de la file F des inégalités est présenté figure 8.10.

Dans l'algorithme L-INÉGALITÉS, les positions mémorisées dans les files F ou J sont des positions sur x. Elles indiquent des inégalités entre x et le facteur aligné à la position f sur y. Ainsi, si p figure dans la file, on a $x[p] \neq y[f + p]$. Lorsque la variable f est mise à jour, l'origine du facteur de y est remplacée par j, et il convient donc d'effectuer une translation en conséquence, c'est-à-dire de diminuer les positions de la quantité $j - f$. Cela est réalisé dans l'algorithme INÉG-FUSION lors de l'ajout d'une position dans la file résultat.

Complexité de la phase de recherche

Avant d'examiner la preuve de l'algorithme L-INÉGALITÉS, nous nous intéressons à sa complexité, c'est-à-dire à celle de la phase de recherche. La complexité dépend de celle de la fonction INÉG-FUSION qui est considérée plus loin (lemme 8.14). Le prétraitement du mot vient à la suite.

Théorème 8.13
Si la fusion réalisée par l'algorithme INÉG-FUSION *s'exécute en temps linéaire, le temps d'exécution de l'algorithme* L-INÉGALITÉS *est* $O(k \times n)$ *dans un espace* $O(k \times m)$.

Preuve À chaque itération de la boucle aux lignes 3–16, le temps d'exécution de l'instruction de fusion à la ligne 7 est $O(k)$ d'après l'hypothèse

i	$x[i]$	$G[i]$
0	a	$\langle\rangle$
1	b	$\langle 1,2,3,4,5,6,7\rangle$
2	a	$\langle 3,4,5\rangle$
3	c	$\langle 3,6,7\rangle$
4	b	$\langle 4,5,6,7\rangle$
5	a	$\langle\rangle$
6	b	$\langle 6,7\rangle$
7	a	$\langle\rangle$

(a)

j	$y[j]$	F
0	a	$\langle 3,4,5\rangle$
1	b	$\langle 0,1,2,5\rangle$
2	a	$\langle 2,3\rangle$
3	b	$\langle 0,1,2,3\rangle$
4	c	$\langle 0,2,3\rangle$
5	b	$\langle 0,3,4,5\rangle$
6	b	$\langle 0,1,2,3\rangle$
7	a	$\langle 3,4,6,7\rangle$
8	b	$\langle 0,1,2,3\rangle$
9	a	$\langle 3,4,5,6\rangle$
10	b	$\langle 0,1\rangle$
11	a	$\langle 1,2,3,4\rangle$
12	a	$\langle 1,2,3\rangle$
13	c	$\langle 3,4,5,7\rangle$
14	b	$\langle 0,1,2,3\rangle$
15	a	$\langle 3,4,5,7\rangle$
16	b	$\langle 0,1,2,3\rangle$
17	a	$\langle 3,5,6,7\rangle$

(b)

Figure 8.10 Files utilisées pour la recherche approchée avec trois inégalités de $x =$ `abacbaba` dans $y =$ `ababcbbababaacbabababbbbab`. **(a)** Valeurs de la table G pour le mot `abacbaba`. La file $G[3]$ par exemple contient 3, 6 et 7, positions sur x des inégalités entre son suffixe `cbaba` et son préfixe `abacb`. **(b)** Valeurs successives de la file F des inégalités calculées par l'algorithme L-INÉGALITÉS. Les valeurs aux positions 0, 2, 4, 10 et 12 sur y possèdent moins de trois éléments, ce qui révèle la présence d'occurrences de x avec au plus trois inégalités à ces positions. À la position 0, par exemple, le facteur `ababcbba` de y possède exactement trois inégalités avec x : elles sont aux positions 3, 4 et 5 sur x.

car la file F contient au plus $k+1$ éléments et $G[j-f]$ en contient au plus $2k+1$. La contribution au temps total est donc $O(k \times n)$.

Les autres opérations de chacune des $n-m+1$ itérations de la boucle aux lignes 3–16, à l'exclusion de la boucle aux lignes 12–15, s'exécutent en temps constant, ce qui contribue pour $O(n)$ au temps global.

Le nombre total d'itérations effectuées par la boucle aux lignes 12–15 est $O(n)$ puisque les instructions font croitre la valeur de la variable g d'une unité à chaque itération sans qu'elle ne diminue jamais.

Il s'ensuit que le temps d'exécution de l'algorithme L-INÉGALITÉS est $O(k \times n)$.

L'espace occupé par la table G est $O(k \times m)$ et celui occupé par les files F et J est $O(k)$, ce qui montre que l'espace utilisé pour le calcul est $O(k \times m)$. ∎

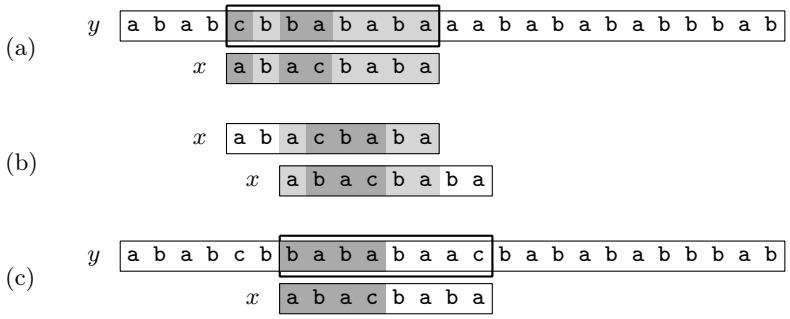


Figure 8.11 Fusion lors de la recherche avec trois inégalités de $x =$ `abacbaba` dans $y =$ `ababcbbababaacbababababbbab`. **(a)** Occurrence de x en position 4 sur y avec trois inégalités aux positions 0, 2 et 3 sur x ; $F = \langle 0, 2, 3\rangle$. **(b)** Il y a trois inégalités entre $x[2\,..\,7]$ et $x[0\,..\,5]$; $G[2] = \langle 3, 4, 5\rangle$. **(c)** Les suites retenues pour la fusion sont $\langle 2, 3\rangle$ et $\langle 3, 4, 5\rangle$, et celle-ci produit la suite $\langle 2, 3, 4, 5\rangle$ des positions des quatre premières inégalités entre x et $y[6\,..\,13]$. Une seule comparaison de lettre est nécessaire à la position 3, entre $x[1]$ et $y[7]$, car les autres positions ne figurent que dans une seule des deux suites.

Fusion

Le but de l'opération INÉG-FUSION$(f, j, g, F, G[j-f])$ (ligne 7 de l'algorithme L-INÉGALITÉS) est de produire la suite des positions des inégalités entre les mots $x[0\,..\,g-j]$ et $y[j\,..\,g]$, en s'appuyant sur la connaissance des inégalités mémorisées dans les files F et $G[j-f]$.

Les positions p dans F marquent les inégalités entre $x[0\,..\,g-f]$ et $y[f\,..\,g]$, mais seules sont utiles au calcul celles qui satisfont l'inégalité $f + p \geq j$ (par définition de F on a déjà $f + p \leq g$). L'objectif du test à la ligne 4 de l'algorithme L-INÉGALITÉS est précisément de supprimer de F les valeurs inutiles. Les positions q de $G[j-f]$ notent les inégalités entre $x[j-f\,..\,m-1]$ et $x[0\,..\,m-j+f-1]$. Celles qui sont utiles doivent satisfaire l'inégalité $f + q \leq g$ (on a déjà $f + q \geq j$). Le test à la ligne 18 de l'algorithme INÉG-FUSION prend en compte cette contrainte. La figure 8.11 illustre la fusion (voir aussi figure 8.9).

Considérons une position p sur x telle que $j \leq f + p \leq g$. Si p figure dans F, cela signifie que $y[f+p] \neq x[p]$. Si p est dans $G[j-f]$, cela signifie que $x[p] \neq x[p-j+f]$. Quatre situations peuvent se présenter pour une position p suivant qu'elle se trouve ou non dans F et $G[j-f]$. (voir figures 8.9 et 8.11) :

1. La position p n'est ni dans F ni dans $G[j-f]$. On a $y[f+p] = x[p]$ et $x[p] = x[p-j+f]$, donc $y[f+p] = x[p-j+f]$.
2. La position p est dans F mais pas dans $G[j-f]$. On a $y[f+p] \neq x[p]$ et $x[p] = x[p-j+f]$, donc $y[f+p] \neq x[p-j+f]$.
3. La position p est dans $G[j-f]$ mais pas dans F. On a $y[f+p] = x[p]$ et $x[p] \neq x[p-j+f]$, donc $y[f+p] \neq x[p-j+f]$.

4. La position p est dans F et dans $G[j-f]$. On a $y[f+p] \neq x[p]$ et $x[p] \neq x[p-j+f]$, ce qui ne permet pas de conclure sur l'égalité entre $y[f+p]$ et $x[p-j+f]$.

Parmi les cas énumérés, seuls les trois derniers peuvent conduire à une inégalité entre les lettres $y[f+p]$ et $x[p-j+f]$. Le dernier cas seulement nécessite une comparaison de lettres supplémentaire. Ils sont traités dans cet ordre respectivement aux lignes 6–7, 9–10 et 11–14 de l'algorithme de fusion.

```
INÉG-FUSION(f, j, g, F, G)
 1  J ← FILE-VIDE()
 2  tantque LONGUEUR(J) ≤ k
      et LONGUEUR(F) > 0 et LONGUEUR(G) > 0 faire
 3        p ← TÊTE(F)
 4        q ← TÊTE(G)
 5        si p < q alors
 6              DÉFILER(F)
 7              ENFILER(J, p − j + f)
 8        sinonsi q < p alors
 9              DÉFILER(G)
10              ENFILER(J, q − j + f)
11        sinon DÉFILER(F)
12              DÉFILER(G)
13              si x[p − j + f] ≠ y[f + p] alors
14                    ENFILER(J, p − j + f)
15  tantque LONGUEUR(J) ≤ k et LONGUEUR(F) > 0 faire
16        p ← DÉFILEMENT(F)
17        ENFILER(J, p − j + f)
18  tantque LONGUEUR(J) ≤ k et LONGUEUR(G) > 0
      et TÊTE(G) ≤ g − f faire
19        q ← DÉFILEMENT(G)
20        ENFILER(J, q − j + f)
21  retourner J
```

Le lemme qui suit fournit le résultat utilisé comme hypothèse dans le théorème 8.13 pour énoncer le temps d'exécution de l'algorithme de recherche approchée avec inégalité.

Lemme 8.14
L'algorithme INÉG-FUSION *s'exécute en temps linéaire.*

Preuve La structure de l'algorithme INÉG-FUSION est formée de trois boucles **tantque**. On constate que l'itération de chacune de ces boucles conduit à ôter un élément des files F ou G (ou des deux). Comme le temps d'exécution d'une itération est constant, on en déduit que le temps total requis par l'algorithme est linéaire en la somme des longueurs des deux files F et G. ∎

Preuve de fonctionnement

La preuve de fonctionnement de l'algorithme L-INÉGALITÉS repose sur celle de la fonction INÉG-FUSION. Un des arguments principaux de la preuve porte sur une propriété de la distance de Hamming qui est énoncée dans le lemme suivant.

Lemme 8.15
Soient u, v et w trois mots de même longueur. Posons $d = Ham(u, v)$, $d' = Ham(v, w)$, et supposons $d' \leq d$. On a alors :

$$d - d' \leq Ham(u, w) \leq d + d' \ .$$

Preuve Les mots étant de même longueur, ils ont même ensemble de positions P. Considérons les ensembles $Q = \{p \in P : u[p] \neq v[p]\}$, $R = \{p \in P : v[p] \neq w[p]\}$ et $S = \{p \in P : u[p] \neq w[p]\}$. Une position $p \in S$ satisfait l'inégalité $u[p] \neq w[p]$ et l'on a donc $u[p] \neq v[p]$ ou $v[p] \neq w[p]$ (ou les deux). Il s'ensuit que $S \subseteq Q \cup R$.

Par ailleurs, $p \in Q \setminus R$ implique $p \in S$ car la condition donne $u[p] \neq v[p]$ et $v[p] = w[p]$; d'où $u[p] \neq w[p]$. De même, par symétrie, $p \in R \setminus Q$ implique $p \in S$.

En conclusion, $Ham(u, w) = \operatorname{card} S$ est majoré par $\operatorname{card}(Q \cup R)$ qui est maximal lorsque Q et R sont disjoints ; donc $Ham(u, w) \leq d + d'$. De plus, $Ham(u, w)$ est minoré par $\operatorname{card}((Q \cup R) \setminus (Q \cap R))$ qui est minimal quand $R \subseteq Q$ (car $d' \leq d$). On a ainsi $Ham(u, w) \geq d - d'$. ∎

Lorsque l'opération INÉG-FUSION$(f, j, g, F, G[j-f])$ est exécutée dans l'algorithme L-INÉGALITÉS, les conditions suivantes sont satisfaites :

1. $f < j \leq g \leq f + m - 1$;
2. $F = \langle p : x[p] \neq y[f + p]$ et $j \leq f + p \leq g) \rangle$;
3. $x[g - f] \neq y[g]$;
4. LONGUEUR$(F) \leq k + 1$;
5. $G = \langle p : x[p] \neq x[p - j + f]$ et $j \leq f + p \leq g' \rangle$ pour un entier g' tel que $j \leq g' \leq f + m - 1$.

De plus, si $g' < f+m-1$, LONGUEUR$(G) = 2k+1$ par définition de G. En prenant ces conditions comme hypothèse on obtient le résultat suivant.

Lemme 8.16
Soit $J =$ INÉG-FUSION$(f, j, g, F, G[j - f])$. *Si* LONGUEUR$(J) \leq k$, *on a :*

$$J = \langle p : x[p] \neq y[j + p] \text{ et } j \leq j + p \leq g \rangle \ ,$$

et, dans le cas contraire :

$$Ham(y[j \,..\, g], x[0 \,..\, g - j]) > k \ .$$

Preuve Posons $u = y[j\,..\,g]$, $v = x[j-f\,..\,g-f]$ et $w = x[0\,..\,g-j]$. Supposons $g' < g$ et posons $v' = x[j-f\,..\,g'-f]$ et $w' = x[0\,..\,g'-j]$. On a LONGUEUR$(G) = 2k+1$, c'est-à-dire $Ham(x[j-f\,..\,g'-f], x[0\,..\,g'-j]) = 2k+1$. Par ailleurs, $Ham(y[j\,..\,g'], x[j-f\,..\,g'-f]) \leq k$ car $g' < g$. D'après le lemme 8.15, on en déduit $Ham(y[j\,..\,g'], x[0\,..\,g'-j]) \geq k+1$.

On déduit de ce résultat que si LONGUEUR$(J) \leq k$, on a nécessairement $g \leq g'$, sinon la fusion effectuée par l'algorithme INÉG-FUSION produirait au moins $k+1$ éléments. Une simple vérification montre alors que l'algorithme fusionne les suites F et $\langle q : q$ dans $G[j-f]$ et $f+q \leq g\rangle$ en une suite S. Et l'algorithme produit la suite $J = \langle q : q+j-f$ dans $S\rangle$ qui satisfait l'égalité de l'énoncé.

Lorsque LONGUEUR$(J) > k$, on a en fait LONGUEUR$(J) = k+1$ puisque l'algorithme de fusion limite la longueur de J à $k+1$. Si $g' < g$, on a vu plus haut que la conclusion est satisfaite. Sinon, l'algorithme trouve effectivement $k+1$ positions q qui satisfont $x[q] \neq y[j+q]$ et $j \leq j+q \leq g$. Ce qui donne la même conclusion et termine la preuve. ∎

La proposition qui suit porte sur le fonctionnement de l'algorithme L-INÉGALITÉS. Elle suppose que les suites $G[q]$ sont calculées conformément à leur définition.

Proposition 8.17
Si $x, y \in A^*$, $m = |x|$, $n = |y|$, $k \in \mathbf{N}$ et $k < m \leq n$, l'algorithme L-INÉGALITÉS *détecte toutes les positions $j = 0, 1, \ldots, n-m$ sur y pour lesquelles $Ham(x, y[j\,..\,j+m-1]) \leq k$.*

Preuve On commence par vérifier qu'après chaque itération dans la boucle principale (aux lignes 3–16) la file F contient la plus longue suite croissante des positions des inégalités entre $y[f\,..\,g]$ et $x[0\,..\,g-f]$ de longueur limitée à $k+1$.

On le vérifie directement pour la première itération grâce aux instructions de la boucle aux lignes 12–15, en remarquant que l'initialisation de la variable g entraine que le test à la ligne 6 n'est pas satisfait, ce qui a pour conséquence une initialisation correcte de J puis de F.

Supposons la condition satisfaite et prouvons qu'elle l'est encore à l'itération suivante. On note que les instructions aux lignes 4–5 ont pour effet de supprimer de F les positions strictement inférieures à $j-f$. Si l'inégalité à la ligne 6 n'est pas satisfaite, la preuve est analogue à celle de la première itération. Dans le cas contraire, la file J est déterminée par la fonction INÉG-FUSION. Si LONGUEUR$(J) > k$, les variables f, g et F sont inchangées donc la condition reste satisfaite. Sinon, la valeur de J ainsi calculée initialise la variable F. D'après le lemme 8.16, la file contient la suite croissante des positions des inégalités entre $y[f\,..\,g]$ et $x[0\,..\,g-f]$. La maximalité de sa longueur est obtenue après exécution des instructions de la boucle aux lignes 12–15. Cela termine l'induction et la preuve de la condition portant sur F.

Soit j une position sur y pour laquelle une occurrence est reportée (ligne 16). La condition à la ligne 15 indique que $g = j + m - 1$. La preuve ci-dessus montre que LONGUEUR$(F) = Ham(x, y[j \,..\, j + m - 1])$, quantité inférieure à k. Il y a donc une occurrence d'un facteur approché à la position j.

Réciproquement, si $Ham(x, y[j \,..\, j + m - 1]) \leq k$, l'instruction à la ligne 16 est exécutée d'après le lemme 8.16. La condition sur F prouvée ci-dessus montre que l'occurrence est détectée. ∎

Prétraitement

Le but de la phase de prétraitement est de calculer les valeurs de la table G qui est requise par l'algorithme L-INÉGALITÉS. Rappelons que pour un décalage q de x, $1 \leq q \leq m - 1$, $G[q]$ est la suite croissante des positions sur x des inégalités les plus à gauche entre $x[q \,..\, m - 1]$ et $x[0 \,..\, m - q - 1]$, et que cette suite est limitée à $2k + 1$ éléments.

Le calcul des suites $G[q]$ est réalisé de façn élémentaire par la fonction dont le code suit.

```
PRÉ-L-INÉGALITÉS(x, m, k)
 1  pour q ← 1 à m − 1 faire
 2        G[q] ← FILE-VIDE()
 3        i ← q
 4        tantque LONGUEUR(G[q]) < 2k + 1 et i < m faire
 5              si x[i] ≠ x[i − q] alors
 6                    ENFILER(G[q], i)
 7              i ← i + 1
 8  retourner G
```

Le temps d'exécution de l'algorithme est $O(m^2)$, mais il est possible de préparer la table en temps $O(k \times m \times \log m)$ (voir exercice 8.6).

8.4 Recherche de motifs courts

L'algorithme présenté dans cette section est une méthode à la fois très rapide en pratique et très simple à mettre en œuvre pour des motifs courts. La méthode permet de résoudre les problèmes présentés dans les sections précédentes dans le modèle vecteur-binaire introduit à la section 1.5. Nous décrivons d'abord la méthode pour la recherche exacte d'un mot dans un texte, puis nous montrons comment on peut l'adapter pour traiter le cas de la recherche avec inégalités et le cas de la recherche avec différences. Le principal avantage de cette méthode est qu'elle peut s'adapter à un large éventail de problèmes.

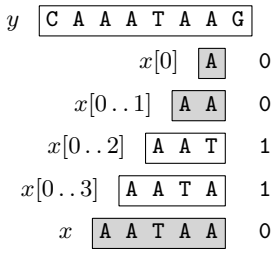


Figure 8.12 Vecteur binaire R_6^0 pour la localisation de $x = \texttt{AATAA}$ dans $y = \texttt{CAAATAAG}$. On a $R_6^0 = \texttt{00110}$. Les seuls préfixes non vides de x qui se terminent à la position 6 sur y sont `A`, `AA` et `AATAA`.

Recherche exacte

Nous allons tout d'abord présenter une technique pour résoudre le problème de la recherche exacte de toutes les occurrences d'un mot x dans un texte y, qui est différente des méthodes déjà rencontrées dans les chapitres 1, 2 et 3.

On considère $n+1$ vecteurs de m bits, $R_{-1}^0, R_0^0, \ldots, R_{n-1}^0$. Le vecteur R_j^0 correspond au traitement de la lettre $y[j]$ du texte. Il contient les informations sur tous les préfixes de x qui se terminent à la position j sur le texte (voir la figure 8.12). Il est défini par :

$$R_j^0[i] = \begin{cases} \texttt{0} & \text{si } x[0\,..\,i] = y[j-i\,..\,j] \ , \\ \texttt{1} & \text{sinon} \ , \end{cases}$$

pour $i = 0, 1, \ldots, m-1$. Lorsque $R_j^0[m-1] = \texttt{0}$, x apparait à la position j. Le vecteur R_{-1}^0 correspond au préfixe de y de longueur nulle ; en conséquence de quoi, toutes ses composantes sont égales à `1` :

$$R_{-1}^0[i] \ = \texttt{1} \ .$$

Pour $j = 0, 1, \ldots, n-1$, le vecteur R_j^0 s'exprime au moyen du vecteur R_{j-1}^0 comme suit :

$$R_j^0[i] = \begin{cases} \texttt{0} & \text{si } R_{j-1}^0[i-1] = \texttt{0} \text{ et } x[i] = y[j] \ , \\ \texttt{1} & \text{sinon} \ , \end{cases}$$

pour $i = 1, 2, \ldots, m-1$, et :

$$R_j^0[0] = \begin{cases} \texttt{0} & \text{si } x[0] = y[j] \ , \\ \texttt{1} & \text{sinon} \ . \end{cases}$$

Le passage du vecteur R_{j-1}^0 au vecteur R_j^0 peut être calculé par l'égalité donnée dans le lemme qui suit, ce qui ramène le calcul à deux opérations sur les vecteurs binaires. On note, pour tout $a \in A$, S_a le vecteur de m bits défini par :

$$S_a[i] = \begin{cases} \texttt{0} & \text{si } x[i] = a \ , \\ \texttt{1} & \text{sinon} \ . \end{cases}$$

(a)

i	$x[i]$	$S_{\mathtt{A}}[i]$	$S_{\mathtt{C}}[i]$	$S_{\mathtt{G}}[i]$	$S_{\mathtt{T}}[i]$
0	A	0	1	1	1
1	A	0	1	1	1
2	T	1	1	1	0
3	A	0	1	1	1
4	A	0	1	1	1

(b)

j	0	1	2	3	4	5	6	7	8	9	10	11
$y[j]$	C	A	A	A	T	A	A	T	A	G	A	A
$R_j^0[0]$	1	0	0	0	1	0	0	1	0	1	0	0
$R_j^0[1]$	1	1	0	0	1	1	0	1	1	1	1	0
$R_j^0[2]$	1	1	1	1	0	1	1	0	1	1	1	1
$R_j^0[3]$	1	1	1	1	1	0	1	1	0	1	1	1
$R_j^0[4]$	1	1	1	1	1	1	0	1	1	1	1	1

Figure 8.13 Illustration de la recherche du mot $x = \mathtt{AATAA}$ dans le texte $y = \mathtt{CAAATAATAGAA}$. **(a)** Les vecteurs S. **(b)** Les vecteurs R^0. Le mot x apparait à la position 6 dans le texte y car $R_6^0[4] = \mathtt{0}$. Il n'apparait qu'à cette position car les autres valeurs $R_j^0[4]$, avec $j \neq 6$, sont égales à $\mathtt{1}$.

Le vecteur S_a est le vecteur caractéristique [2] des positions de la lettre a sur le mot x. Il peut être calculé préalablement à la phase de recherche.

Lemme 8.18
Pour $j = 0, 1, \ldots, n-1$, le calcul de R_j^0 se réduit à deux opérations logiques, un décalage et une disjonction :

$$R_j^0 = (1 \dashv R_{j-1}^0) \vee S_{y[j]} \ .$$

Preuve Pour $i = 0, 1, \ldots, m-1$, $R_j^0[i] = \mathtt{0}$ signifie que $x[0\,..\,i]$ est un suffixe de $y[0\,..\,j]$, ce qui est vrai lorsque les deux conditions suivantes sont vérifiées : $x[0\,..\,i-1]$ est un suffixe de $y[0\,..\,j-1]$, ce qui équivaut à $R_{j-1}^0[i-1] = \mathtt{0}$; $x[i]$ est égale à $y[j]$ ce qui équivaut à $S_{y[j]}[i] = \mathtt{0}$. De plus $R_j^0[0] = \mathtt{0}$ lorsque $S_{y[j]}[0] = \mathtt{0}$. On a bien $R_j^0 = (1 \dashv R_{j-1}^0) \vee S_{y[j]}$ puisque l'opération $1 \dashv R_{j-1}^0$ introduit un $\mathtt{0}$ en première position de R_{j-1}^0. ∎

L'algorithme L-MOTIF-COURT effectue la localisation de x dans y. Une seule variable, notée R^0 dans le code, permet de représenter la suite des vecteurs binaires $R_{-1}^0, R_0^0, \ldots, R_{n-1}^0$. Un exemple d'exécution de l'algorithme L-MOTIF-COURT est donné figure 8.13.

2. Le vecteur caractéristique « opposé » a été introduit section 8.1.

L-MOTIF-COURT(x, m, y, n)
1 **pour** chaque lettre $a \in A$ **faire**
2 $\quad S_a \leftarrow \mathtt{1}^m$
3 **pour** $i \leftarrow 0$ **à** $m-1$ **faire**
4 $\quad S_{x[i]}[i] \leftarrow \mathtt{0}$
5 $R^0 \leftarrow \mathtt{1}^m$
6 **pour** $j \leftarrow 0$ **à** $n-1$ **faire**
7 $\quad R^0 \leftarrow (1 \dashv R^0) \vee S_{y[j]}$
8 $\quad$ SIGNALER-SI$(R^0[m-1] = \mathtt{0})$

Proposition 8.19
L'algorithme L-MOTIF-COURT *localise toutes les occurrences d'un mot x dans un texte y.*

Preuve La preuve est une conséquence du lemme 8.18. ∎

Les opérations sur les vecteurs binaires utilisées dans l'algorithme L-MOTIF-COURT s'effectuent en temps constant lorsque la longueur m du mot x est inférieure au nombre de bits d'un mot machine (modèle vecteur-binaire). D'où le résultat suivant.

Proposition 8.20
Lorsque la longueur m du mot x est inférieure au nombre de bits d'un mot machine, la phase de prétraitement de l'algorithme L-MOTIF-COURT *s'exécute en temps $\Theta(\operatorname{card} A)$ dans un espace mémoire $\Theta(\operatorname{card} A)$. La phase de recherche s'exécute en temps $\Theta(n)$.*

Preuve La phase de prétraitement consiste à calculer les vecteurs S_a, ce qui est effectué par les boucles aux lignes 1–2 et 3–4. La boucle aux lignes 1–2 nécessite un espace $O(\operatorname{card} A)$ et s'exécute en temps $O(\operatorname{card} A)$. La boucle aux lignes 3–4 s'exécute en temps $O(m)$, c'est-à-dire en temps constant d'après l'hypothèse. La phase de recherche effectuée par la boucle aux lignes 6–8 s'exécute en temps $O(n)$ puisque l'examen de chaque lettre du texte y n'implique que deux opérations sur les vecteurs binaires. ∎

Une inégalité

L'algorithme précédent peut facilement être adapté pour résoudre le problème de la recherche approchée avec k inégalités ou substitutions (section 8.3). Pour simplifier nous présentons le cas où une substitution au plus est autorisée.

Nous utilisons les vecteurs $R^0_{-1}, R^0_0, \ldots, R^0_{n-1}$ et les vecteurs de la forme S_a avec $a \in A$ comme précédemment, et nous introduisons les vecteurs de m bits $R^1_{-1}, R^1_0, \ldots, R^1_{n-1}$ pour tenir compte des inégalités.

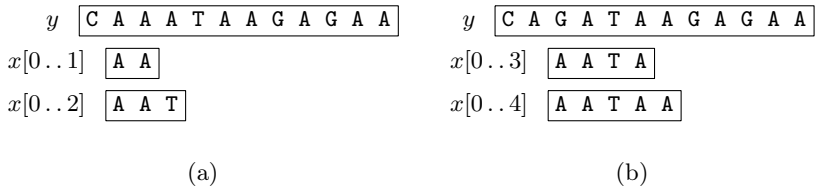


Figure 8.14 Éléments de la preuve du lemme 8.21. **(a)** Le préfixe de longueur 2 du mot x est un suffixe de $y[0\,..\,2]$, ce qui se traduit par $R_2^0[1] = \mathtt{0}$. Donc, substituer `T` à `A` donne une occurrence avec une inégalité du préfixe de longueur 3 de x à la fin de $y[0\,..\,3]$. Ainsi $R_3^1[2] = \mathtt{0}$. **(b)** Le préfixe de longueur 4 de x apparait avec une inégalité à la fin de $y[0\,..\,5]$, ce qui est donné par $R_5^1[3] = \mathtt{0}$. De plus $x[4] = y[6]$: le préfixe de longueur 5 du mot apparait avec une inégalité à la fin de $y[0\,..\,6]$, ce qui correspond à $R_6^1[4] = \mathtt{0}$.

Les vecteurs R_j^1 permettent de détecter toutes les occurrences de x dans y avec au plus une substitution. Ils sont définis par :

$$R_j^1[i] = \begin{cases} \mathtt{0} & \text{si } \mathit{Ham}(x[0\,..\,i], y[j-i\,..\,j]) \leq 1 \ , \\ \mathtt{1} & \text{sinon} \ , \end{cases}$$

pour $i = 0, 1, \ldots, m-1$.

Lemme 8.21
Pour $j = 0, 1, \ldots, n-1$, les vecteurs R_j^1 correspondant à la recherche approchée avec une inégalité vérifient la relation :

$$R_j^1 = ((1 \dashv R_{j-1}^1) \vee S_{y[j]}) \wedge (1 \dashv R_{j-1}^0) \ .$$

Preuve Trois cas peuvent se produire ; ils sont traités séparément.

Cas 1 : les i premières lettres de x sont identiques aux i dernières lettres de $y[0\,..\,j-1]$ (soit $R_{j-1}^0[i-1] = \mathtt{0}$). Dans ce cas, substituer $x[i]$ à $y[j]$ crée une occurrence avec au plus une substitution entre les $i+1$ premières lettres de x et les $i+1$ dernières lettres de $y[0\,..\,j]$ (voir figure 8.14(a)). D'où $R_j^1[i] = \mathtt{0}$ lorsque $R_{j-1}^0[i-1] = \mathtt{0}$.

Cas 2 : il y a une occurrence avec une substitution entre les i premières lettres de x et les i dernières de $y[0\,..\,j-1]$ (soit $R_{j-1}^1[i-1] = \mathtt{0}$). Si $x[i] = y[j]$, alors il y a une occurrence avec une substitution entre les $i+1$ premières lettres de x et les $i+1$ dernières lettres de $y[0\,..\,j]$ (voir figure 8.14(b)). D'où $R_j^1[i] = \mathtt{0}$ lorsque $R_{j-1}^1[i-1] = \mathtt{0}$ et $y[j] = x[i]$.

Cas 3 : Si ni la condition du cas 1 ni celle du cas 2 ne sont satisfaites, on a $R_j^1[i] = \mathtt{1}$.

Il se déduit de l'étude des trois cas que l'expression donnée dans l'énoncé est correcte. ∎

L'algorithme L-INÉG-MOTIF-COURT effectue la recherche approchée avec k inégalités en utilisant la relation du lemme 8.21. Son code est donné ci-dessous. L'algortithme requiert $k+1$ vecteurs binaires, notés

j	0	1	2	3	4	5	6	7	8	9	10	11
$y[j]$	C	A	A	A	T	A	A	T	A	G	A	A
$R^1_j[0]$	0	0	0	0	0	0	0	0	0	0	0	0
$R^1_j[1]$	1	0	0	0	0	0	0	0	0	0	0	0
$R^1_j[2]$	1	1	1	0	0	1	1	0	1	1	1	1
$R^1_j[3]$	1	1	1	1	1	0	1	1	0	1	1	1
$R^1_j[4]$	1	1	1	1	1	1	0	1	1	0	1	1

Figure 8.15 Le mot $x =$ AATAA apparait deux fois, aux positions 6 et 9, avec au plus une inégalité dans le texte $y =$ CAAATAATAGAA. Cela se vérifie sur la table R^1 associée à la recherche car $R^1_6[4] = R^1_9[4] = 0$.

$R^0, R^1, \ldots, R^k$. Les vecteurs R^0_j, pour $j = -1, 0, \ldots, n-1$, sont maintenus comme dans l'algorithme de recherche exacte. Les valeurs des autres vecteurs sont calculées conformément au lemme.

L-INÉG-MOTIF-COURT(x, m, y, n, k)
1 **pour** chaque lettre $a \in A$ **faire**
2 $\quad S_a \leftarrow 1^m$
3 **pour** $i \leftarrow 0$ **à** $m-1$ **faire**
4 $\quad S_{x[i]}[i] \leftarrow 0$
5 $R^0 \leftarrow 1^m$
6 **pour** $\ell \leftarrow 1$ **à** k **faire**
7 $\quad R^\ell \leftarrow (1 \dashv R^{\ell-1})$
8 **pour** $j \leftarrow 0$ **à** $n-1$ **faire**
9 $\quad T \leftarrow R^0$
10 $\quad R^0 \leftarrow (1 \dashv R^0) \vee S_{y[j]}$
11 $\quad$ **pour** $\ell \leftarrow 1$ **à** k **faire**
12 $\quad\quad T' \leftarrow R^\ell$
13 $\quad\quad R^\ell \leftarrow ((1 \dashv R^\ell) \vee S_{y[j]}) \wedge (1 \dashv T)$
14 $\quad\quad T \leftarrow T'$
15 $\quad$ SIGNALER-SI$(R^k[m-1] = 0)$

Un exemple d'exécution de l'algorithme L-INÉG-MOTIF-COURT est donné figure 8.15.

Une insertion

Nous montrons comment adapter la méthode du début de la section au cas où seule une insertion ou seule une suppression est autorisée. La généralisation à k différences et l'algorithme complet sont donnés en fin de section.

On adapte les vecteurs R^1_j au problème. Le vecteur R^1_{j-1} indique ici toutes les occurrences avec une insertion entre un préfixe de x et un suffixe de $y[0\,..\,j-1]$: $R^1_{j-1}[i-1] = 0$ lorsque les i premières lettres de x

j	0	1	2	3	4	5	6	7	8	9	10	11
$y[j]$	C	A	A	A	T	A	A	T	A	G	A	A
$R^1_j[0]$	1	0	0	0	0	0	0	0	0	0	0	0
$R^1_j[1]$	1	1	0	0	0	0	0	0	0	1	0	0
$R^1_j[2]$	1	1	1	1	0	0	1	0	0	1	1	1
$R^1_j[3]$	1	1	1	1	1	0	0	1	0	0	1	1
$R^1_j[4]$	1	1	1	1	1	1	0	0	1	1	0	1

Figure 8.16 Les mots AAATAA, AATAAT et AATAGA sont trois occurrences, aux positions respectives 6, 7 et 10, avec une insertion du mot $x =$ AATAA dans le texte $y =$ CAGATAATAGAA car $R^1_6[4] = R^1_7[4] = R^1_{10}[4] = 0$.

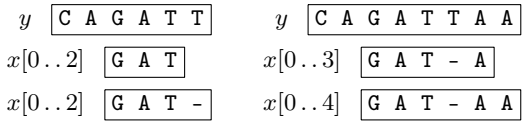


Figure 8.17 Éléments de la preuve du lemme 8.22. **(a)** Le préfixe de longueur 3 du mot apparait à la fin de $y[0\,..\,4]$, ce qui est donné par $R^0_4[2] = 0$. Insérer $y[5]$ donne une occurrence du préfixe de longueur 3 du mot avec une insertion à la fin de $y[0\,..\,5]$, soit $R^1_5[2] = 0$. **(b)** Le préfixe de longueur 4 du mot apparait avec une insertion à la fin de $y[0\,..\,6]$, ce qui est donné par $R^1_6[3] = 0$. De plus, comme $x[4] = y[7]$, le préfixe de longueur 5 du mot apparait avec une insertion à la fin de $y[0\,..\,7]$, soit $R^1_7[4] = 0$.

(le préfixe $x[0\,..\,i-1]$) coïncident avec au moins i des $i+1$ dernières lettres de $y[0\,..\,j-1]$ (le suffixe $y[j-i\,..\,j-1]$). Le vecteur R^0 est mis à jour comme précédemment et nous montrons maintenant comment mettre à jour R^1. Un exemple est donné figure 8.16.

Lemme 8.22
Pour $j = 0, 1, \ldots, n-1$, les vecteurs R^1_j correspondant à la recherche approchée avec une insertion vérifient la relation :

$$R^1_j = ((1 \dashv R^1_{j-1}) \vee S_{y[j]}) \wedge R^0_{j-1} \ .$$

Preuve Les trois cas qui peuvent se produire sont traités séparément.

Cas 1 : les mots $x[0\,..\,i]$ et $y[j-i-1\,..\,j-1]$ sont identiques (soit $R^0_{j-1}[i] = 0$). Alors insérer $y[j]$ crée une occurrence avec une insertion entre $x[0\,..\,i]$ et $y[j-i-1\,..\,j]$ (voir figure 8.17(a)). D'où $R^1_j[i] = 0$ lorsque $R^0_{j-1}[i] = 0$.

Cas 2 : il y a une occurrence avec une insertion entre $x[0\,..\,i-1]$ et $y[j-i-1\,..\,j-1]$ (soit $R^1_{j-1}[i-1] = 0$). Alors, si $y[j] = x[i]$, il y a

j	0	1	2	3	4	5	6	7	8	9	10	11
$y[j]$	C	A	A	A	T	A	A	T	A	G	A	A
$R^1_j[0]$	0	0	0	0	0	0	0	0	0	0	0	0
$R^1_j[1]$	1	0	0	0	1	0	0	1	0	1	0	0
$R^1_j[2]$	1	1	0	0	0	1	0	0	1	1	1	0
$R^1_j[3]$	1	1	1	0	0	0	1	0	0	1	1	1
$R^1_j[4]$	1	1	1	1	1	0	0	1	0	1	1	1

Figure 8.18 Les mots AATA, ATAA et AATA sont trois occurrences, aux positions respectives 5, 6 et 8, avec une suppression du mot $x =$ AATAA dans le texte $y =$ CAGATAATAGAA. Cela est signifié par $R^1_5[4] = R^1_6[4] = R^1_8[4] = 0$.

une occurrence avec une insertion entre $x[0\,..\,i]$ et $y[j-i-1\,..\,j]$ (voir figure 8.17(b)). D'où $R^1_j[i] = 0$ lorsque $R^1_{j-1}[i-1] = 0$ et $y[j] = x[i]$.

Cas 3 : Si ni la condition du cas 1 ni celle du cas 2 ne sont satisfaites, on a $R^1_j[i] = 1$.

L'expression donnée dans l'énoncé s'en déduit. ∎

Une suppression

Nous supposons maintenant que R^1_j signale toutes les occurrences avec au plus une suppression entre des préfixes de x et des suffixes de $y[0\,..\,j]$. Un exemple est donné figure 8.18.

Lemme 8.23
Pour $j = 0, 1, \ldots, n-1$, les vecteurs R^1_j correspondant à la recherche approchée avec une suppression vérifient la relation :

$$R^1_j = ((1 \dashv R^1_{j-1}) \vee S_{y[j]}) \wedge (1 \dashv R^0_j) \ .$$

Preuve Les trois cas qui peuvent se produire sont traités séparément.

Cas 1 : les mots $x[0\,..\,i-1]$ et $y[j-i-1\,..\,j]$ sont identiques (soit $R^0_j[i-1] = 0$). Supprimer $x[i]$ crée une occurrence avec une suppression entre $x[0\,..\,i]$ et $y[j-i-1\,..\,j]$ (voir figure 8.19(a)). D'où $R^1_j[i] = 0$ lorsque $R^0_j[i-1] = 0$.

Cas 2 : il y a une occurrence avec une suppression entre $x[0\,..\,i-1]$ et $y[j-i+1\,..\,j-1]$ (soit $R^1_{j-1}[i-1] = 0$). Alors, si $y[j] = x[i]$, il y a une occurrence avec une suppression entre $x[0\,..\,i]$ et $y[j-i+1\,..\,j]$ (voir figure 8.19(b)). D'où $R^1_j[i] = 0$ lorsque $R^1_{j-1}[i-1] = 0$ et $y[j] = x[i]$.

Cas 3 : Si ni la condition du cas 1 ni celle du cas 2 ne sont satisfaites, on a $R^1_j[i] = 1$.

La correction de l'expression donnée dans l'énoncé s'en déduit. ∎

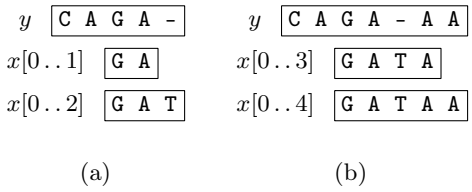


Figure 8.19 Éléments de la preuve du lemme 8.23. **(a)** Le préfixe de longueur 2 du mot apparait à la fin de $y[0\,..\,3]$, ce qui est donné par $R_3^0[1] = \texttt{0}$. Supprimer $x[2]$ donne une occurrence du préfixe de longueur 3 du mot avec une suppression à la fin de $y[0\,..\,3]$ donc $R_3^1[2] = \texttt{0}$. **(b)** Le préfixe de longueur 4 du mot apparait avec une suppression à la fin de $y[0\,..\,4]$, ce qui est donné par $R_4^1[3] = \texttt{0}$. De plus, comme $x[4] = y[5]$, le préfixe de longueur 5 du mot apparait avec une suppression à la fin de $y[0\,..\,5]$ donc $R_5^1[4] = \texttt{0}$.

j	0	1	2	3	4	5	6	7	8	9	10	11
$y[j]$	C	A	A	A	T	A	A	T	A	G	A	A
$R_j^1[0]$	0	0	0	0	0	0	0	0	0	0	0	0
$R_j^1[1]$	1	0	0	0	0	0	0	0	0	0	0	0
$R_j^1[2]$	1	1	0	0	0	0	0	0	0	1	1	0
$R_j^1[3]$	1	1	1	0	0	0	0	0	0	0	1	1
$R_j^1[4]$	1	1	1	1	1	0	0	0	0	0	0	1

Figure 8.20 Les mots de `AATA`, `AATAA`, `ATAA`, `AATAAT`, `AATA`, `AATAG` et `AATAGA` sont sept occurrences avec au plus une différence du mot $x =$ `AATAA` dans le texte $y =$ `CAAATAATAGAA`. Ils apparaissent aux positions respectives 5, 6, 6, 7, 8, 9 et 10, car $R_5^1[4] = R_6^1[4] = R_7^1[4] = R_8^1[4] = R_9^1[4] = R_{10}^1[4] = \texttt{0}$.

Motifs courts avec différences

Nous présentons maintenant un algorithme de recherche approchée de motifs courts avec au plus k différences du type insertion, suppression et substitution. Celui-ci cumule les méthodes utilisées pour le traitement de chaque opération prise séparément. L'algorithme requiert $k+1$ vecteurs binaires $R^0, R^1, \ldots, R^k$. Les vecteurs R_j^0, pour $j = -1, 0, \ldots, m-1$, sont maintenus comme dans l'algorithme de recherche exacte. Les valeurs des autres vecteurs sont calculées d'après la relation de la proposition donnée avant le code de l'algorithme. Un exemple de recherche avec une différence est montré figure 8.20.

Proposition 8.24
Pour $i = 1, 2, \ldots, k$ on a :

$$R_j^i = ((1 \dashv R_{j-1}^i) \vee S_{y[j]}) \wedge (1 \dashv (R_j^{i-1} \wedge R_{j-1}^{i-1})) \wedge R_{j-1}^{i-1} \ .$$

Preuve La preuve de la proposition 8.24 est une conséquence directe des lemmes 8.21, 8.22 et 8.23. La relation

$$R_j^i = ((1 \dashv R_{j-1}^i) \vee S_{y[j]}) \wedge (1 \dashv R_j^{i-1}) \wedge (1 \dashv R_{j-1}^{i-1}) \wedge R_{j-1}^{i-1}$$

se réécrit en celle donnée dans l'énoncé. ∎

L-DIFF-MOTIF-COURT(x, m, y, n, k)

```
 1  pour chaque lettre a ∈ A faire
 2        S_a ← 1^m
 3  pour i ← 0 à m − 1 faire
 4        S_{x[i]}[i] ← 0
 5  R^0 ← 1^m
 6  pour ℓ ← 1 à k faire
 7        R^ℓ ← (1 ⊣ R^{ℓ−1})
 8  pour j ← 0 à n − 1 faire
 9        T ← R^0
10        R^0 ← (1 ⊣ R^0) ∨ S_{y[j]}
11        pour ℓ ← 1 à k faire
12              T' ← R^ℓ
13              R^ℓ ← ((1 ⊣ R^ℓ) ∨ S_{y[j]}) ∧ (1 ⊣ (T ∧ R^{ℓ−1})) ∧ T
14              T ← T'
15        SIGNALER-SI(R^k[m − 1] = 0)
```

Théorème 8.25
Lorsque la longueur m du mot x est inférieure au nombre de bits d'un mot machine, la phase de prétraitement de l'algorithme L-DIFF-MOTIF-COURT *s'exécute en temps $\Theta(k + \operatorname{card} A)$ dans un espace mémoire $\Theta(k + \operatorname{card} A)$. La phase de recherche s'exécute en temps $\Theta(k \times n)$.*

Preuve La preuve du théorème 8.25 est similaire à celle de la proposition 8.20. ∎

8.5 Heuristique pour la recherche avec différences

La méthode heuristique décrite ici permet de localiser avec k différences un préfixe, un suffixe ou la totalité d'un mot x dans un texte y. Elle utilise partiellement des techniques de programmation dynamique.

On fait référence aux diagonales de

$$\{0, 1, \ldots, m-1\} \times \{0, 1, \ldots, n-1\}$$

au moyen d'un entier d. La diagonale d est l'ensemble des couples (i, j) pour lesquels

$$j - i = d \ .$$

La méthode de localisation est paramétrée par deux entiers $\ell, k > 0$. Elle procède en trois phases. Dans la première phase, tous les facteurs de longueur ℓ du mot qui apparaissent dans y sont localisés. Cette phase est réalisée grâce à une technique de hachage. Lors de la deuxième phase, la diagonale d contenant le plus de facteurs de longueur ℓ du mot est sélectionnée. La troisième phase consiste à trouver un alignement par programmation dynamique dans une bande de largeur $2k$ autour de la diagonale d.

Nous décrivons à présent le détail de chaque phase du calcul.

On définit l'ensemble Z_ℓ par :

$$Z_\ell = \{(i,j) : i = 0,1,\ldots,m-\ell \text{ et } j = 0,1,\ldots,n-\ell \\ \text{et } x[i\,..\,i+\ell-1] = y[j\,..\,j+\ell-1]\} \ .$$

Autrement dit, l'ensemble Z_ℓ contient tous les couples (i,j) tels que le facteur de longueur ℓ débutant à la position i sur x est identique au facteur de longueur ℓ débutant à la position j sur y. En reprenant une notation de la section 4.4, on a donc $\mathit{prem}_\ell(x[i\,..\,m-1]) = \mathit{prem}_\ell(y[j\,..\,n-1])$.

Pour chaque diagonale

$$d = -m-1, -m, \ldots, n-1 \ ,$$

on considère le nombre d'éléments de Z_ℓ se trouvant sur cette diagonale :

$$\mathit{compte}[d] = \mathrm{card}\{(i,j) \in Z_\ell : j - i = d\} \ .$$

Pour effectuer le comptage, chaque facteur de longueur ℓ est codé par un entier. Un facteur de longueur ℓ est considéré comme la représentation en base $\mathrm{card}\, A$ d'un entier. Formellement, on associe de manière bijective un rang à chaque lettre a de l'alphabet A. L'entier $\mathit{rang}(a)$ est compris entre 0 et $\mathrm{card}\, A - 1$. On pose :

$$\mathit{code}(w[0\,..\,\ell-1]) = \sum_{i=0}^{\ell-1} \mathit{rang}(w[\ell-i-1]) \times (\mathrm{card}\, A)^i$$

pour tout mot w de longueur supérieure ou égale à ℓ. Ainsi les codes de tous les facteurs de longueur ℓ du mot et du texte peuvent être calculés en temps linéaire en utilisant la relation suivante (pour $i \geq 0$) :

$$\mathit{code}(w[i+1\,..\,i+\ell]) = \\ (\mathit{code}(w[i\,..\,i+\ell-1]) \bmod (\mathrm{card}\, A)^{\ell-1}) \times \mathrm{card}\, A \\ + \mathit{rang}(w[i+\ell]) \ .$$

Les codes des facteurs de longueur ℓ du mot x sont calculés en une passe et l'on accumule les facteurs dans une table *position* de dimension $(\mathrm{card}\, A)^\ell$ éléments. Plus précisément, la valeur de

$$\mathit{position}[c]$$

est l'ensemble des positions droites du facteur de x de longueur ℓ dont le code est c. Le calcul de la table est réalisé par la fonction HACHAGE.

HACHAGE(x, m, ℓ)
1 **pour** $c \leftarrow 0$ **à** $(\operatorname{card} A)^{\ell} - 1$ **faire**
2 $\quad$ *position*$[c] \leftarrow \emptyset$
3 $(exp, code) \leftarrow (1, 0)$
4 **pour** $i \leftarrow 0$ **à** $\ell - 2$ **faire**
5 $\quad$ $exp \leftarrow exp \times \operatorname{card} A$
6 $\quad$ $code \leftarrow code \times \operatorname{card} A + rang(x[i])$
7 **pour** $i \leftarrow \ell - 1$ **à** $m - 1$ **faire**
8 $\quad$ $code \leftarrow (code \bmod exp) \times \operatorname{card} A + rang(x[i])$
9 $\quad$ *position*$[code] \leftarrow$ *position*$[code] \cup \{i\}$
10 **retourner** *position*

Deuxième phase : après l'initialisation de la table *position*, les codes des facteurs du texte y sont calculés. À chaque fois qu'une égalité, entre un facteur de longueur ℓ du mot et un facteur de longueur ℓ du texte, est trouvée sur une diagonale, le compteur de cette diagonale est incrémenté. C'est précisément ce que réalise la fonction DIAGONALE.

DIAGONALE$(x, m, y, n, \ell, position)$
1 **pour** $d \leftarrow -m + 1$ **à** $n - 1$ **faire**
2 $\quad$ *compte*$[d] \leftarrow 0$
3 $(exp, code) \leftarrow (1, 0)$
4 **pour** $j \leftarrow 0$ **à** $\ell - 2$ **faire**
5 $\quad$ $exp \leftarrow exp \times \operatorname{card} A$
6 $\quad$ $code \leftarrow code \times \operatorname{card} A + rang(y[j])$
7 **pour** $j \leftarrow \ell - 1$ **à** $n - 1$ **faire**
8 $\quad$ $code \leftarrow (code \bmod exp) \times \operatorname{card} A + rang(y[j])$
9 $\quad$ **pour** chaque $i \in$ *position*$[code]$ **faire**
10 $\quad\quad$ *compte*$[j - i] \leftarrow$ *compte*$[j - i] + 1$
11 **retourner** *compte*

Pour réaliser la dernière phase de la méthode, il suffit enfin de repérer la diagonale d ayant le compte le plus élevé. On peut alors produire un alignement entre le mot x et le texte y en utilisant l'algorithme de programmation dynamique et en empruntant un chemin éloigné d'au plus k positions de la diagonale d (les insertions et les suppressions sont pénalisées de g). C'est ici aussi qu'il y a approximation. Elle est d'autant plus forte que k est petit. La figure 8.21 montre un exemple de calcul.

(a)

T	j	-1	0	1	2	3	4	5	6	7	8	9	10
i		$y[j]$	L	A	W	Y	Q	Q	K	P	G	K	A
-1	$x[i]$		3	6	9	12	15						
0	Y			5	8	9	12	15					
1	W				5	8	11	14	17				
2	C					7	10	13	16	19			
3	Q						7	10	13	16	19		
4	P							9	12	13	16	19	
5	G								11	14	13	16	19
6	K									13	16	13	16

(b)

$$\begin{pmatrix} \mathtt{Y} & \mathtt{W} & \mathtt{C} & \mathtt{Q} & \mathtt{-} & \mathtt{-} & \mathtt{P} & \mathtt{G} & \mathtt{K} \\ \mathtt{A} & \mathtt{W} & \mathtt{Y} & \mathtt{Q} & \mathtt{Q} & \mathtt{K} & \mathtt{P} & \mathtt{G} & \mathtt{K} \end{pmatrix} \quad \begin{pmatrix} \mathtt{Y} & \mathtt{W} & \mathtt{C} & \mathtt{-} & \mathtt{Q} & \mathtt{-} & \mathtt{P} & \mathtt{G} & \mathtt{K} \\ \mathtt{A} & \mathtt{W} & \mathtt{Y} & \mathtt{Q} & \mathtt{Q} & \mathtt{K} & \mathtt{P} & \mathtt{G} & \mathtt{K} \end{pmatrix}$$

$$\begin{pmatrix} \mathtt{Y} & \mathtt{W} & \mathtt{-} & \mathtt{C} & \mathtt{Q} & \mathtt{-} & \mathtt{P} & \mathtt{G} & \mathtt{K} \\ \mathtt{A} & \mathtt{W} & \mathtt{Y} & \mathtt{Q} & \mathtt{Q} & \mathtt{K} & \mathtt{P} & \mathtt{G} & \mathtt{K} \end{pmatrix}$$

Figure 8.21 Illustration pour la méthode heuristique de recherche avec différences. On considère le cas où $x = \mathtt{YWCQPGK}$, $y = \mathtt{LAWYQQKPGKA}$, $\ell = 2$, $k = 2$, card $A = 20$, et où le rang des lettres qui apparaissent dans x et y est

a	A	C	G	K	L	P	Q	W	Y
$rang(a)$	0	1	5	8	9	12	13	18	19

Pour les codes, on obtient $code(\mathtt{YW}) = 19 \times 20^1 + 18 \times 20^0 = 398$, puis, pour $i = 2$, $code(\mathtt{WC}) = (code(\mathtt{YW}) \bmod 20) \times 20 + 1 = 361$, et ainsi de suite. Ce qui donne les codes suivants pour les facteurs de longueur ℓ de x :

i	0	2	3	4	5	6
$x[i-\ell+1\,..\,i]$	YW	WC	CQ	QP	PG	GK
$code(x[i-\ell+1\,..\,i])$	398	361	33	272	245	108

D'où les valeurs de la table *position*, dont on ne donne que celles qui sont distinctes de l'ensemble vide :

code	33	108	245	272	361	398
position[*code*]	{3}	{6}	{5}	{4}	{2}	{1}

Les codes associés aux facteurs de longueur ℓ de y sont :

j	1	2	3	4	5	6	7	8	9	10
$y[j-\ell+1\,..\,j]$	LA	AW	WY	YQ	QQ	QK	KP	PG	GK	KA
$code(y[j-\ell+1\,..\,j])$	180	18	379	393	273	268	172	245	108	160

Les seuls indices j de code à position non vide sont 8 et 9. Pour ces deux indices, on incrémente les éléments $compte[8-5]$ et $compte[9-6]$, ce qui donne $compte[3] = 2$ après traitement. Il s'ensuit que la diagonale qui possède le compte le plus élevé est la diagonale 3. **(a)** Ensuite, avec les valeurs $g = 3$, $k = 2$, $Sub(a, a) = 0$ et $Sub(a, b) = 2$ pour $a, b \in A$ avec $a \neq b$, on calcule un alignement éloigné d'au plus deux positions de la diagonale 3. **(b)** Les trois alignements correspondants.

```
ALIGNEMENT-BANDE(x, m, y, n, d, k)
 1  (i', i'') ← (max{−1, −d − 1 − k}, min{−d − 1 + k, m − 1})
 2  (j', j'') ← (max{−1, d − 1 − k}, min{d − 1 + k, n − 1})
 3  c ← g
 4  pour i ← i' à i'' faire
 5        T[i, −1] ← c
 6        c ← c + g
 7  c ← g
 8  pour j ← j' à j'' faire
 9        T[−1, j] ← c
10        c ← c + g
11  pour i ← 0 à m − 1 faire
12        pour j ← i + d − k à i + d + k faire
13              si 0 ≤ j ≤ n − 1 alors
14                    T[i, j] ← T[i − 1, j − 1] + Sub(x[i], y[j])
15                    si |j − i − 1 − d| ≤ k alors
16                          T[i, j] ← min{T[i, j], T[i, j − 1] + g}
17                    si |j − i + 1 − d| ≤ k alors
18                          T[i, j] ← min{T[i, j], T[i − 1, j] + g}
19  retourner T
```

Remarquons enfin que l'utilisation d'une technique « diviser pour régner », comme celle utilisée dans la section 7.3, permet une implantation de la fonction ALIGNEMENT-BANDE qui s'exécute en temps $O(m \times k)$ et en espace $O(n)$.

Notes

Le théorème 8.3 est de Fischer et Paterson (1974). Le résultat utilisé dans la preuve du théorème selon lequel il est possible de multiplier un nombre à M chiffres par un nombre à N chiffres en temps $O(N \times \log M \times \log\log M)$ pour $N \geq M$ est de Schnhage et Strassen (1971).

L'algorithme L-DIFF-ÉLAG est dû à Ukkonen (1985). L'algorithme L-DIFF-DIAG avec son implantation à l'aide du calcul d'ancêtres communs a été décrit par Landau et Vishkin (1988). Harel et Tarjan (1984) ont présenté le premier algorithme fonctionnant en temps constant qui résoud le problème de l'ancêtre commun à deux nœuds d'un arbre. Une version sensiblement améliorée est due à Schieber et Vishkin (1988).

Landau et Vishkin (1986) ont conçu l'algorithme L-INÉGALITÉS. La taille de l'automate de la section 8.3 a été établie par Melichar (1995).

La recherche de mots courts à la manière de l'algorithme L-DIFF-MOTIF-COURT est traité par Wu et Manber (1992) ainsi que Baeza-Yates et Gonnet (1992).

Une synthèse et des résultats d'expériences sur la recherche de motifs approchés est présentée par Navarro (2000).

La méthode de comparaison globale avec insertion et suppression est à la base du logiciel FastA (voir Pearson et Lipman, 1988). Le paramètre ℓ introduit dans la section 8.5 correspond au paramètre *KTup* du logiciel ; sa valeur est généralement fixée à 6 pour le traitement des séquences d'acides nucléiques et à 2 pour le traitement des séquences d'acides aminés.

Exercices

8.1 (Moteur !)
Trouver toutes les occurrences du mot à jokers `ab§§b§ab` dans le texte `bababbaabbaba`.

Trouver toutes les occurrences du mot à jokers `ab§§b§a` dans le texte à jokers `bababb§ab§aba`.

8.2
Trouver toutes les occurrences avec au plus deux inégalités du mot `ACGAT` dans le texte `GACGATATATGATAC`.

8.3 (Couts)
Quels couts doit-on attribuer aux opérations d'édition pour réaliser les opérations suivantes ? Pour $x, y \in A^+$ et $\gamma \in \mathbf{N}$:

- localiser le mot x dans le texte y ;
- rechercher les sous-mots de y qui sont égaux à x ;
- rechercher les sous-mots de y de la forme $x_0u_0x_1u_1 \dots u_{k-1}x_{k-1}$ où $x = x_0x_1 \dots x_{k-1}$ et $|u_i| \leq \gamma$ pour $i = 0, 1, \dots, k-1$.

8.4
Trouver toutes les occurrences avec au plus deux différences du mot `ACGAT` dans le texte `GACGATATATGATAC` en utilisant l'algorithme L-DIFF-DYN.

Résoudre la même question en utilisant les algorithmes L-DIFF-ÉLAG et L-DIFF-DIAG.

8.5 (Économie)
Décrire une implantation de l'algorithme L-DIFF-DIAG qui fonctionne en espace $O(m)$. [*Aide :* échanger les boucles sur q et d dans l'écriture de l'algorithme.]

8.6 (Inégalités)
Concevoir un algorithme de préparation des files de la table G (voir section 8.3) qui fonctionne en temps $O(k \times m \times \log m)$. [*Aide :* appliquer la phase de recherche avec inégalités aux blocs d'indices allant de $2^{\ell-1}-1$ à $2^\ell - 2$ pour $\ell = 1, 2, \dots, \lceil \log m \rceil$; voir Landau et Vishkin (1986).]

8.7 (Anagrammes)
Écrire un algorithme linéaire permettant de localiser toutes les permutations d'un mot x dans un texte y. [*Aide :* utiliser un compteur pour chaque lettre de $alph(x)$.]

8.8
Trouver toutes les occurrences avec au plus deux différences du « mot court » $x = \texttt{ACGAT}$ dans le texte $y = \texttt{GACGATATATGATAC}$.

8.9 (La classe)
Proposer une extension de l'algorithme L-DIFF-MOTIF-COURT prenant en entrée une classe de mots. Une classe de mots est une expression de la forme $X[0]X[1]..X[m-1]$ avec $X[i] \subseteq A$ pour $i = 0, 1, \ldots, m-1$.

8.10 (Gamma-delta)
On considère une distance entre lettres d: $A \times A \to \mathbf{R}$, deux réels positifs δ et γ, un mot x de longueur m, et un texte y de longueur n.

Le mot x possède une occurrence δ-approchée dans le texte y s'il existe une position $j = 0, 1, \ldots, n-m$ sur y telle que $d(x[i], y[i+j]) \leq \delta$ pour $i = 0, 1, \ldots, m-1$. Le mot x possède une occurrence γ-approchée dans le texte y s'il existe une position $j = 0, 1, \ldots, n-m$ sur y telle que :

$$\sum_{i=0}^{m-1} d(x[i], y[i+j]) \leq \gamma \ .$$

Le mot x possède une occurrence (δ, γ)-approchée dans le texte y si x possède une occurrence qui est à la fois δ-approchée et γ-approchée, c'est-à-dire, s'il existe une position $j = 0, 1, \ldots, n-m$ sur y telle que $d(x[i], y[i+j]) \leq \delta$ pour $i = 0, 1, \ldots, m-1$ et :

$$\sum_{i=0}^{m-1} d(x[i], y[i+j]) \leq \gamma \ .$$

Écrire un algorithme permettant de localiser toutes les occurrences δ-approchées (respectivement γ-approchées, (δ, γ)-approchées) du mot x dans le texte y. En évaluer la complexité. [*Aide :* voir Cambouropoulos et co-auteurs (1999).]

8.11 (Motifs distribués)
Soient X une liste de k mots de longueur m et Y une liste de ℓ textes de longueur n. On dit que la liste X possède une occurrence distribuée dans la liste Y s'il existe une position $j = 0, 1, \ldots, n-m$ pour laquelle : pour chaque $i = 0, 1, \ldots, m-1$, il existe p et q tels que $0 \leq p \leq k-1$, $0 \leq q \leq \ell - 1$ et $X_p[i] = Y_q[i+j]$.

Écrire un algorithme permettant de localiser toutes les occurrences distribuées de la liste X dans la liste Y. Étudier les cas particuliers pour lesquels X est réduit à un seul mot ($k = 1$) et Y est réduit à un seul texte ($\ell = 1$). [*Aide :* voir Holub et co-auteurs (1999).]

9 Périodes locales

On s'intéresse dans ce chapitre à la localisation des périodicités locales qui peuvent apparaitre au sein d'un mot.

La méthode de détection des périodicités choisie est basée sur un partitionnement des suffixes qui permet également de classer les suffixes en ordre lexicographique. Le procédé est analogue à celui utilisé dans le chapitre 4 pour la préparation de la table des suffixes d'un mot mais présente l'avantage de ne nécessiter qu'un espace mémoire linéaire pour le calcul.

On introduit dans la section 9.1 une méthode de partitionnement simplifiée qui est mieux adaptée et plus efficace pour la seule localisation de périodes locales. La détection des périodes est traitée immédiatement après dans la section 9.2.

Dans la section 9.3, on considère les puissances particulières que sont les carrés. Leur recherche en temps optimal fait appel à des algorithmes qui utilisent des procédés combinatoires mêlés à l'utilisation des structures du chapitre 5. On discute également le nombre maximal de carrés pouvant apparaitre dans un mot, ce qui donne des bornes supérieures au nombre de périodicités locales.

Finalement, dans la section 9.4, on revient sur la question du classement lexicographique des suffixes d'un mot et le calcul de leurs préfixes communs. La solution présentée est une autre adaptation de la méthode de partitionnement ; elle peut être utilisée avec profit pour la préparation d'une table des suffixes (chapitre 4).

9.1 Partitionnement des facteurs

La méthode décrite dans cette section est à la base d'algorithmes de recherche de périodicités locales dans un mot. Elle consiste à partitionner les suffixes du mot par rapport à leurs débuts de longueur k. Les notions d'équivalences utilisées pour le partitionnement sont celles de la section 4.4, mais la méthode de calcul est différente. L'adaptation de la

méthode au classement des suffixes d'un mot est présentée en section 9.4. Le mot de référence est noté y et il est de longueur n.

On commence par rappeler quelques notations introduites dans la section 4.4. Le début d'ordre k, $k > 0$, d'un mot u est défini par :

$$prem_k(u) = \begin{cases} u & \text{si } |u| \leq k \ , \\ u[0\,..\,k-1] & \text{sinon } . \end{cases}$$

La relation d'équivalence $\equiv_k$ sur les positions sur y est définie par :

$i \equiv_k j$

si et seulement si

$prem_k(y[i\,..\,n-1]) = prem_k(y[j\,..\,n-1])$.

L'équivalence $\equiv_k$ induit une partition de l'ensemble des positions en classes d'équivalence qui sont numérotées à partir de 0. Ainsi, on note $E_k[i]$ le numéro de la classe selon $\equiv_k$ qui contient la position i.

Dans la section 4.4, l'équivalence $\equiv_{2k}$ est calculée à partir de $\equiv_k$ en application du lemme de doublement, ce qui donne au plus $\lceil \log_2 n \rceil$ étapes pour le calcul de toutes les équivalences considérées, et produit un temps total $O(n \log n)$. Ici, le calcul des équivalences est incrémental sur les valeurs de k, mais il y a utilisation d'une autre technique pour le calcul des équivalences successives. Elle conduit à traiter chaque position au plus $\lceil \log_2 n \rceil$ fois, ce qui permet de conserver le même temps d'exécution asymptotique $O(n \log n)$.

On décrit maintenant la technique de partitionnement qui travaille sur les partitions associées aux équivalences $\equiv_k$ ($k > 0$). Pour une classe P de la partition on note $P-1$ l'ensemble $\{i-1 : i \in P\}$. Partitionner par rapport à une classe P, c'est remplacer chaque classe d'équivalence C par $C \cap (P-1)$ et $C \setminus (P-1)$ en éliminant les ensembles vides qui résultent de ces opérations. L'algorithme PARTITIONNEMENT ci-dessous calcule les équivalences $\equiv_1, \equiv_2, \ldots$ dans cet ordre. L'étape centrale du calcul consiste à partitionner toutes les classes de l'équivalence courante par rapport à une même classe P. Le lemme qui suit est à la base du principe de fonctionnement de l'algorithme. C'est son raffinement (lemme 9.2) qui est utilisé dans l'algorithme PARTITIONNEMENT. Le lemme qui suit repose essentiellement sur la remarque illustrée par la figure 9.1.

Lemme 9.1
Pour tout entier $k > 0$, les classes de l'équivalence $\equiv_{k+1}$ sont de la forme $G = C \cap (P-1)$ avec $G \neq \emptyset$, où C est une classe de $\equiv_k$, et $P = \{n\}$ ou P est une classe de $\equiv_k$.

Preuve $\Rightarrow$: soient deux positions i et j telles que $i \equiv_{k+1} j$. Par définition :

$prem_{k+1}(y[i\,..\,n-1]) = prem_{k+1}(y[j\,..\,n-1])$.

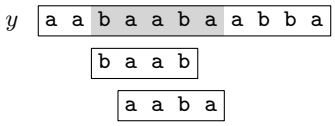


Figure 9.1 Élément de base de la preuve du lemme 9.1. Dans le cas où $y =$ `aabaabaabba`, le mot `baaba` est $prem_5(y[2\,..\,10])$. Il est identifié par ses deux facteurs `baab`, qui est $prem_4(y[2\,..\,10])$, et `aaba`, qui est $prem_4(y[3\,..\,10])$.

On a donc l'égalité :

$$prem_k(y[i\,..\,n-1]) = prem_k(y[j\,..\,n-1]) \ ,$$

ce qui revient à dire que $i, j \in C$ pour une classe C selon $\equiv_k$. Mais on a aussi :

$$prem_k(y[i+1\,..\,n-1]) = prem_k(y[j+1\,..\,n-1])$$

(voir figure 9.1), ce qui signifie que $i+1, j+1 \in P$ pour une classe P selon $\equiv_k$, si $i+1, j+1 < n$. On a alors $i, j \in (P-1)$. Si $i+1 = n$ ou $j+1 = n$, on constate que la seule possibilité est d'avoir $i = j = n-1$. Une classe selon $\equiv_{k+1}$ est donc de la forme $C \cap (P-1)$ comme annoncé.

$\Leftarrow$: considérons un ensemble non vide de la forme $C \cap (P-1)$ où C et P satisfont aux conditions de l'énoncé, et soient $i, j \in C \cap (P-1)$. Si $P = \{n\}$, on a $i = j = n-1$ et donc encore $i \equiv_{k+1} j$. Si $P \neq \{n\}$, C et P sont des classes selon $\equiv_k$ par hypothèse, et l'on a $i+1, j+1 < n$. Par définition de l'équivalence $\equiv_k$, on en déduit l'égalité :

$$prem_k(y[i\,..\,n-1]) = prem_k(y[j\,..\,n-1]) \ .$$

Mais on en déduit aussi l'égalité :

$$prem_k(y[i+1\,..\,n-1]) = prem_k(y[j+1\,..\,n-1]) \ .$$

Ceci implique :

$$prem_{k+1}(y[i\,..\,n-1]) = prem_{k+1}(y[j\,..\,n-1])$$

(voir figure 9.1), c'est-à-dire $i \equiv_{k+1} j$. Ce qui achève la réciproque et la preuve. ∎

Le calcul des équivalences qui se déduit directement du lemme précédent peut être réalisé en temps quadratique ($O(n^2)$) en utilisant un classement par sélection de places comme dans l'algorithme Tri-suffixes de la section 4.4. Un exemple de ce calcul est donné dans la figure 9.2. On reconnait sur le schéma la structure de l'arbre des suffixes du mot de l'exemple. L'algorithme de calcul des équivalences travaille, en quelque sorte, en parcourant l'arbre niveau par niveau depuis sa racine.

Afin d'accélérer le calcul de partitionnement, on considère une notion de différence entre les équivalences $\equiv_k$ et $\equiv_{k-1}$ ($k > 1$). Pour cela, on

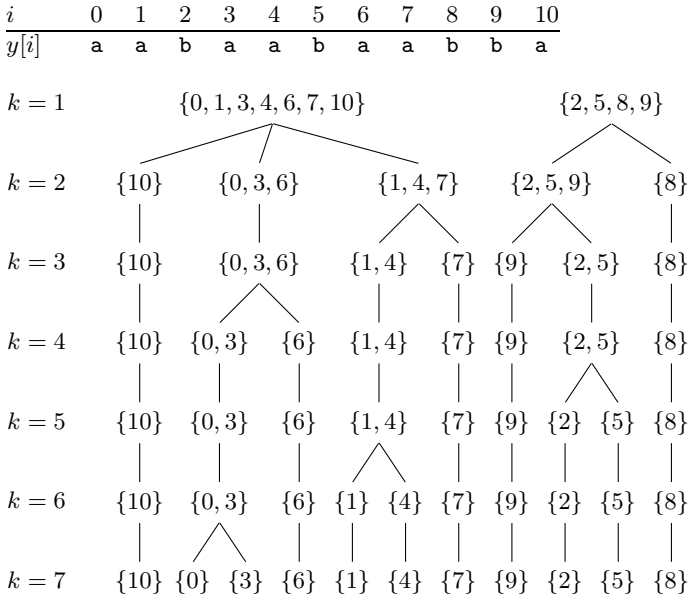


Figure 9.2 Calcul incrémental des partitions associées aux équivalences $\equiv_k$ sur le mot $y =$ `aabaabaabba`. Les classes de positions selon $\equiv_k$ sont notées de gauche à droite en ordre croissant de leur numéro. Ainsi, ligne $k = 2$, $E_2[10] = 0$, $E_2[0] = E_2[3] = E_2[6] = 1$, $E_2[1] = E_2[4] = E_2[7] = 2$, etc.

définit les petites classes de l'équivalence $\equiv_k$. La définition est relative à une fonction de choix de sous-classes, notée c_k, définie sur l'ensemble des classes selon $\equiv_{k-1}$ et à valeur dans l'ensemble des classes selon $\equiv_k$. Si C est une classe relativement à $\equiv_{k-1}$, $c_k(C)$ est une classe de $\equiv_k$ pour laquelle $c_k(C) \subseteq C$, c'est-à-dire que $c_k(C)$ est une sous-classe de C. On appelle ***petite classe*** de $\equiv_k$ relativement à la fonction c_k de choix de sous-classes, toute classe d'équivalence selon $\equiv_k$ qui n'est pas dans l'image de la fonction c_k. Pour $k = 1$, on considère par convention que toutes les classes selon $\equiv_1$ sont des petites classes.

Toujours relativement à c_k on note $\cong_k$ l'équivalence définie sur les positions sur y par :

$i \cong_k j$

si et seulement si

$i, j \in C$ et C petite classe de $\equiv_k$

ou

$i \in c_k(F)$ et $j \in c_k(G)$ pour des classes F, G selon $\equiv_{k-1}$.

La partition des positions induite par $\cong_k$ est constituée des petites classes de $\equiv_k$ et de la classe supplémentaire obtenue par réunion des classes de l'image de la fonction c_k.

On remarque que l'équivalence $\cong_k$ est plus grossière que $\equiv_k$, c'est-à-dire que $i \equiv_k j$ implique $i \cong_k j$, ou que toute classe selon $\equiv_k$ est contenue dans une classe selon $\cong_k$.

Dans l'exemple de la figure 9.2, en définissant c_3 par $c_3(\{10\}) = \{10\}$, $c_3(\{0,3,6\}) = \{0,3,6\}$, $c_3(\{1,4,7\}) = \{1,4\}$, $c_3(\{2,5,9\}) = \{2,5\}$ et $c_3(\{8\}) = \{8\}$, l'équivalence $\cong_3$ partitionne l'ensemble des positions en trois classes : $\{7\}$, $\{9\}$, $\{0,1,2,3,4,5,6,8,10\}$. Les petites classes sont $\{7\}$ et $\{9\}$ (se reporter également à la figure 9.3).

Le lemme suivant a pour conséquence que le calcul de la partition induite par $\equiv_{k+1}$ peut se faire à partir de $\equiv_k$ et de ses seules petites classes, propriété qui est à la base du fonctionnement de l'algorithme PARTITIONNEMENT.

Lemme 9.2
Pour tout entier $k > 0$, les classes de l'équivalence $\equiv_{k+1}$ sont de la forme $G = C \cap (P - 1)$ avec $G \neq \emptyset$, où C est une classe de $\equiv_k$, et $P = \{n\}$ ou P est une classe de $\cong_k$.

Preuve $\Rightarrow$: la première partie de la preuve du lemme 9.1 vaut aussi pour ce lemme car $i \equiv_k j$ implique $i \cong_k j$.

$\Leftarrow$: considérons un ensemble $C \cap (P-1)$ pour lequel C et P satisfont aux conditions de l'énoncé, et soient $i, j \in C \cap (P-1)$. Si $P = \{n\}$ ou si P est une petite classe, donc une classe selon $\equiv_k$, on obtient la conclusion comme dans la preuve du lemme 9.1. Le cas restant est celui pour lequel P est la réunion des $c_k(F)$, F classe selon $\equiv_{k-1}$. Comme $i, j \in C$, on a $i \equiv_k j$. Et comme $i + 1, j + 1 < n$, on en déduit $i + 1 \equiv_{k-1} j + 1$. En résumé, $i + 1$ et $j + 1$ appartiennent à P et à une même classe G selon $\equiv_{k-1}$. Par définition de $\cong_k$, ils appartiennent donc à $c_k(G)$ qui est une classe de $\equiv_k$. Finalement, de $i \equiv_k j$ et $i+1 \equiv_k j+1$, on déduit $i \equiv_{k+1} j$, ce qui termine la réciproque et la preuve. ∎

Le code de l'algorithme PARTITIONNEMENT explicite une grande partie de la méthode de calcul. Il est donné ci-dessous. La variable *Petites* mémorise la liste des petites classes de l'équivalence courante. Celle-ci est représentée par l'ensemble de ses classes, chacune étant implantée par une liste. Les transferts de positions sont effectués vers une classe dite jumelle. Chaque classe jumelle est vide avant l'exécution de la boucle **pour** des lignes 11–18. Il en est de même de l'ensemble de sous-classes associé à chaque classe.

La gestion des classes d'équivalence comme des listes n'est pas un élément essentiel pour le partitionnement proprement dit. Elle est utilisée ici pour permettre une description simple de l'algorithme PUISSANCES de la section suivante, qui, lui, a réellement besoin d'une telle organisation. La figure 9.3 illustre le fonctionnement de l'algorithme PARTITIONNEMENT.

```
PARTITIONNEMENT(y, n)
 1  pour r ← 0 à card alph(y) − 1 faire
 2        C_r ← ⟨⟩
 3  pour i ← 0 à n − 1 faire
 4        r ← rang de y[i] dans la liste classée des lettres de alph(y)
 5        C_r ← C_r · ⟨i⟩
 6  Petites ← {C_r : r = 0, 1, ..., card alph(y) − 1}
 7  k ← 1
 8  tantque Petites ≠ ∅ faire
 9        ▷ Invariant : i, j ∈ C_r ssi i ≡_k j ssi E_k[i] = E_k[j]
10        ▷ Partitionnement
11        pour chaque P ∈ Petites faire
12              pour chaque i ∈ P \ {0}, séquentiellement faire
13                    soit C la classe de i − 1
14                    soit C_P la classe jumelle de C
15                    ôter i − 1 de C
16                    C_P ← C_P · ⟨i − 1⟩
17              pour chaque couple (C, C_P) considéré faire
18                    ajouter C_P aux sous-classes de C
19        ▷ Choix des petites classes
20        Petites ← ∅
21        pour chaque classe C considérée à l'étape précédente faire
22              si C non vide alors
23                    ajouter C aux sous-classes de C
24              remplacer C par ses sous-classes
25              G ← une sous-classe de C de taille maximale
26              Petites ← Petites ∪ ({sous-classes de C} \ {G})
27        k ← k + 1
```

L'analyse du temps d'exécution, qui est $O(n \log n)$, est détaillée dans les trois énoncés qui suivent. Le lemme 9.3 correspond essentiellement à l'étude des lignes 12–18 de l'algorithme.

Une implantation efficace des partitions manipulées consiste à représenter chaque classe d'équivalence par une liste chainée affectée d'un numéro, et simultanément à associer à chaque position le numéro de sa classe. De cette manière, les opérations effectuées sur une position pour partitionner une classe s'exécutent en temps constant. Elles se composent de l'accès à sa classe, son extraction et son ajout.

Lemme 9.3
Le partitionnement par rapport à une classe P peut être réalisé en temps $\Omega(\operatorname{card} P)$.

Preuve Le partitionnement d'une classe C par rapport à P consiste à calculer $C \cap (P-1)$ et $C \setminus C \cap (P-1)$. Cela se réalise au moyen d'une opération de transfert de position d'une classe dans une autre. Avec

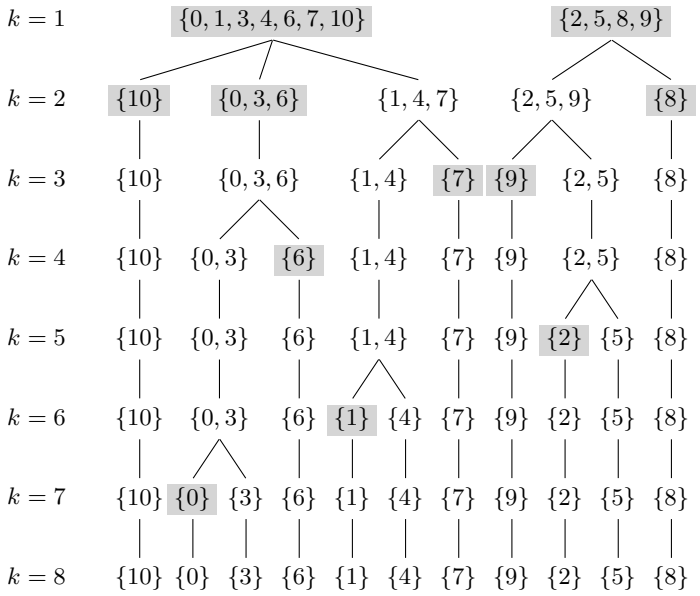


Figure 9.3 Calcul incrémental des partitions associées aux équivalences $\equiv_k$ sur le mot $y =$ `aabaabaabba` comme dans la figure 9.2. Les petites classes sont indiquées par une zone grisée. Le nombre d'opérations effectuées est proportionnel au nombre total de leurs éléments.

l'implantation décrite avant l'énoncé, cette opération prend un temps constant. Les $\operatorname{card} P$ transferts prennent ainsi un temps $\Omega(\operatorname{card} P)$.

Toutes les classes C concernées sont traitées pendant le processus. Les ensembles vides étant éliminés après l'examen de tous les éléments de la classe P. Comme il y a au plus $\operatorname{card} P$ classes C concernées, cette étape prend aussi un temps $O(\operatorname{card} P)$.

Il s'ensuit que le temps total du partitionnement par rapport à P est $\Omega(\operatorname{card} P)$ comme annoncé. ∎

Corollaire 9.4
Pour tout entier $k > 0$, le calcul de $\equiv_{k+1}$ à partir de $\equiv_k$ et de ses petites classes peut être réalisé en temps $\Omega(\sum_{P \text{ petite classe de } \equiv_k} \operatorname{card} P)$.

Preuve Le résultat est une conséquence directe du lemme 9.3. ∎

Considérons l'exemple de la figure 9.3 et le calcul de $\equiv_4$ (ligne $k = 4$). Les petites classes de $\equiv_3$ sont $\{7\}$ et $\{9\}$. Donc le calcul de $\equiv_4$ consiste simplement à extraire 6 et 8 de leurs classes respectives. Cela a pour seul effet de partager la classe $\{0, 3, 6\}$ en $\{0, 3\}$ et $\{6\}$ (8 étant seul dans sa classe) et de produire $\{6\}$ comme petite classe à l'étape suivante.

L'algorithme Partitionnement utilise une fonction de choix spécifique. Celle-ci sélectionne pour chaque C, classe selon l'équivalence $\equiv_{k-1}$,

une sous-classe $c_k(C)$ de taille maximale parmi les classes selon l'équivalence $\equiv_k$ qui sont sous-classes de C. C'est grâce à ce choix particulier que l'on obtient un temps d'exécution $O(n \log n)$.

Théorème 9.5
Soit $K > 0$ le plus petit entier pour lequel les équivalences $\equiv_K$ et $\equiv_{K+1}$ coïncident. L'algorithme Partitionnement *calcule les équivalences $\equiv_1, \equiv_2, \ldots, \equiv_K$, définies sur les positions sur un mot de longueur n, en temps $O(n \log n)$.*

Preuve Les boucles **pour** des lignes 1–2 et 3–5 calculent $\equiv_1$. Les instructions des lignes 11–27 de la boucle **tantque** calculent $\equiv_{k+1}$ à partir de $\equiv_k$ suivant le lemme 9.2, après avoir vérifié que les petites classes sont bien sélectionnées. L'arrêt a lieu dès qu'il n'y a plus de petite classe, c'est-à-dire lorsque les équivalences $\equiv_k$ et $\equiv_{k+1}$ coïncident pour la première fois. Cela intervient pour $k = K$ par définition de K. L'algorithme Partitionnement calcule donc bien la suite d'équivalences de l'énoncé.

Évaluons maintenant son temps d'exécution. Le temps d'exécution de la boucle des lignes 1–2 est $\Omega(\operatorname{card} \mathit{alph}(y))$. Celui de la boucle des lignes 3–5 est $O(n \times \log \operatorname{card} \mathit{alph}(y))$ en utilisant une structure de données efficace afin de mémoriser l'alphabet. Le temps d'exécution de la boucle des lignes 8–27 est proportionnel à la somme des tailles de toutes les petites classes utilisées pendant le partitionnement d'après le corollaire 9.4.

Par le choix particulier des petites classes, une position qui se trouve placée dans une petite classe voit la taille de sa classe diminuer de moitié (au moins) : si $i \in C$, classe selon $\equiv_{k-1}$, et $i \in C'$, C' petite classe de $\equiv_k$ (C' sous-classe de C), on a $\operatorname{card} C' \leq \operatorname{card} C/2$. On en déduit que chaque position appartient à une petite classe au plus $1 + \lfloor \log_2 n \rfloor$ fois. Cela donne le temps $O(n \log n)$ pour la boucle à la ligne 8.

Le temps d'exécution global de l'algorithme est donc $O(n \log n)$. ∎

Lorsqu'on applique l'algorithme Partitionnement à $y = a^n$, le temps d'exécution est $O(n)$. La figure 9.4 illustre le déroulement de l'algorithme sur le mot `aaaaa`. Chaque étape s'exécute en temps constant car, après la phase initiale, il y une seule petite classe qui, de plus, est un singleton.

On rencontre une situation totalement différente quand y est un mot de de Bruijn (voir section 1.2). L'exemple du mot `babbbaaaba` est décrit dans la figure 9.5. Pour ces mots, après la phase initiale, chaque étape prend un temps $O(n)$ car les petites classes contiennent environ $n/2$ éléments. Mais le nombre de ces étapes n'est que de $\lfloor \log_2 n \rfloor$. On obtient ainsi des exemples pour lesquels le nombre d'opérations est $\Omega(n \log n)$.

De façon générale, on vérifie que le nombre K d'étapes exécutées par l'algorithme Partitionnement est aussi $\ell + 1$ où ℓ est la longueur maximale des facteurs qui possèdent au moins deux occurrences dans y.

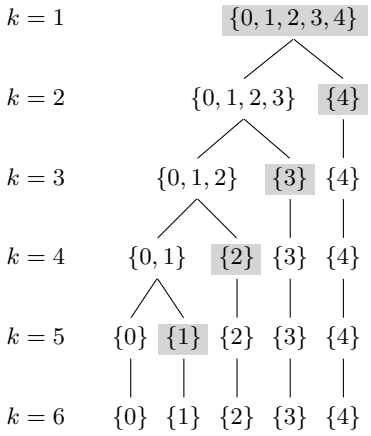


Figure 9.4 L'opération PARTITIONNEMENT appliquée au mot $y =$ `aaaaa`. Après la phase initiale, le calcul se fait en quatre étapes, chacune prenant un temps constant.

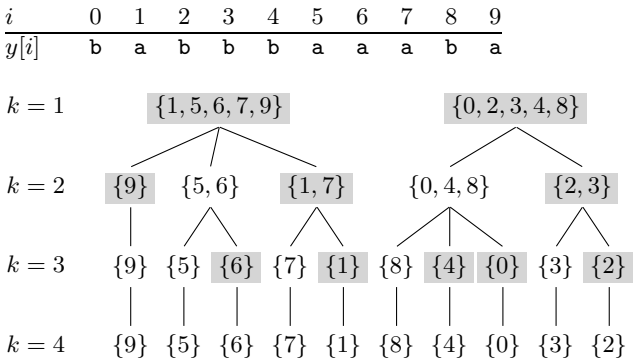


Figure 9.5 L'opération PARTITIONNEMENT appliquée au mot de de Bruijn $y =$ `babbbaaaba`. Après la phase initiale, le calcul se fait en deux étapes, chacune demandant cinq extractions d'éléments.

9.2 Localisation des puissances

Dans cette section, on présente une adaptation assez directe de l'algorithme PARTITIONNEMENT de la section précédente. Elle permet le calcul des facteurs d'un mot qui sont des puissances. On discute ensuite le nombre d'occurrences de puissances qui peuvent exister dans un mot, ce qui conduit à l'optimalité de l'algorithme.

Une ***puissance locale*** d'un mot y de longueur n est un facteur de y de la forme u^e. Plus précisément, u^e est une puissance locale à la position i sur y si u^e est un préfixe de $y[i\,..\,n-1]$ avec $u \in A^+$, u primitif, et e entier, $e > 1$. C'est une ***puissance locale maximale*** (à droite) en i si de plus u^{e+1} n'est pas préfixe de $y[i\,..\,n-1]$. On peut considérer les puissances locales maximales à gauche (en demandant que u ne soit pas suffixe de $y[0\,..\,i-1]$) et maximales bilatères. Leur localisation dans y est une simple adaptation de l'algorithme décrit pour la détection des puissances locales maximales à droite.

Une occurrence d'une puissance locale u^e est identifiée par le triplet (i, p, e) où i est sa position, $p = |u|$ sa période et e son exposant.

Calcul des puissances locales

La détection des puissances locales se fait à l'aide d'une notion de distance associée à l'équivalence $\equiv_k$. Pour toute position i sur y, on note cette distance :

$$D_k[i] = \begin{cases} \min L & \text{si } L \neq \emptyset \ , \\ \infty & \text{sinon} \ , \end{cases}$$

où :

$$L = \{\ell : \ell = 1, 2, \ldots, n-i-1 \text{ et } E_k[i] = E_k[i+\ell]\} \ .$$

Dit autrement, $D_k[i]$ est la distance de i à la plus proche position supérieure de sa classe selon $\equiv_k$, lorsque cette position existe.

Lemme 9.6
Le triplet d'entiers (i, p, e), avec $0 \leq i < n$, $p > 0$ et $e > 1$, identifie l'occurrence d'une puissance locale maximale en i si et seulement si

$$D_p[i] = D_p[i+p] = \cdots = D_p[i+(e-2)p] = p$$

et

$$D_p[i+(e-1)p] \neq p \ .$$

Preuve On pose $u = y[i\,..\,i+p-1]$.

$\Rightarrow$: par définition d'une puissance locale maximale, le mot u apparait aux positions $i, i+p, \ldots, i+(e-1)p$ sur y mais pas à la position $i+ep$. On

en déduit, par définition de D_p, les inégalités $D_p[i] \leq p$, $D_p[i+p] \leq p$, ..., $D_p[i+(e-2)p] \leq p$ ainsi que $D_p[i+(e-1)p] \neq p$. Si l'une des inégalités est stricte, cela implique que u^2 possède une occurrence interne de u lui-même. Mais cela contredit la primitivité de u d'après le lemme de primitivité. Donc les inégalités sont en fait des égalités, ce qui prouve que les conditions de l'énoncé sont satisfaites.

$\Leftarrow$: quand la conclusion de l'énoncé est satisfaite, par définition de D_p, le mot u apparait aux positions $i, i+p, \ldots, i+(e-1)p$ sur y car ces positions sont équivalentes relativement à $\equiv_p$, mais n'apparait pas à la position $i+ep$. Il reste donc à vérifier que u est primitif. Si ce n'est pas le cas, y possède une occurrence de u à une position j, $i < j < i+p$, ce qui implique $D_p[i] \leq j-i$ et contredit l'égalité $D_p[i] = p$. Donc u est primitif et (i,p,e) correspond bien à une puissance locale maximale. ∎

L'algorithme de localisation de toutes les occurrences des puissances locales maximales apparaissant dans y est appelé PUISSANCES. Il est obtenu à partir de l'algorithme PARTITIONNEMENT par ajout d'éléments complémentaires qui sont décrits ici.

On utilise une table D qui réalise la table D_k à chaque étape k. On maintient simultanément la partition des positions associée aux valeurs de la table D. C'est-à-dire que i et j appartiennent à une même classe de cette partition si et seulement si $D[i] = D[j]$. Les classes sont représentées par des listes qui permettent des transferts en temps constant.

Les ajouts à l'algorithme PARTITIONNEMENT portent essentiellement sur le calcul de la table D, ainsi que sur le maintien simultané des listes associées, ce qui ne pose pas de difficulté supplémentaire.

La mise à jour de D intervient lorsqu'il y a transfert d'une position i dans une autre classe d'équivalence. On utilise fortement le fait que les classes d'équivalence selon $\equiv_k$ sont gérées comme des listes, c'est-à-dire que les positions y sont mémorisées en ordre croissant. Si i possède un prédécesseur i' dans sa classe de départ, la nouvelle valeur de $D[i']$ est $D[i'] + D[i]$. Il n'y a pas d'autre changement pour les éléments de la classe car ils sont en ordre croissant. Dans sa classe d'arrivée, i est le dernier élément ajouté, car le partitionnement relatif à une classe P se fait dans l'ordre croissant des éléments de P (voir ligne 12). On définit donc $D[i] = \infty$. De plus, si i a un prédécesseur i'' dans sa nouvelle classe, on définit $D[i''] = i - i''$.

Finalement, à chaque étape k, on obtient les puissances d'exposant k cherchées en examinant la liste des positions i pour lesquelles $D[i] = k$ en application du lemme 9.6. L'algorithme peut alors produire les triplets (i,p,e) concernés. Il faut juste s'assurer, à l'implantation, que les triplets

$$(i,p,e), (i+p,p,e-1), \ldots$$

correspondant à des puissances locales maximales en

$$i, i+p, \ldots$$

sont produits en temps proportionnel à leur nombre, et non en temps quadratique.

La description de PUISSANCES ci-dessus montre que le calcul des puissances locales maximales peut être réalisé dans le même temps que le partitionnement. On constate aussi que les opérations supplémentaires qui produisent les puissances maximales ont un temps d'exécution proportionnel à ce nombre de puissances. En se référant à la proposition 9.8 ci-après, on en déduit alors le résultat suivant.

Théorème 9.7
L'algorithme PUISSANCES *calcule toutes les occurrences des puissances locales maximales d'un mot de longueur n en temps $O(n \log n)$.* ∎

Considérons l'exemple $y = \texttt{aabaabaabba}$ de la figure 9.3. Lorsque la partition associée à $\equiv_3$ est calculée (ligne $k = 3$), les éléments qui ont une distance de 3 avec leur successeur sont 0, 1, 2 et 3 ($D[0] = D[1] = D[2] = D[3] = 3$). Ces éléments correspondent aux puissances maximales $(\texttt{aab})^3$ en 0, $(\texttt{aba})^2$ en 1, $(\texttt{baa})^2$ en 2 et $(\texttt{aab})^2$ en 3.

Nombre d'occurrences de puissances locales

Le temps d'exécution de l'algorithme PUISSANCES dépend de la taille de sa sortie, le nombre de puissances maximales. L'exemple de $y = \texttt{a}^n$ montre qu'un mot peut contenir un nombre quadratique de puissances locales. Mais, avec cet exemple, on n'obtient que $n - 1$ puissances locales maximales. La proposition 9.8 donne une borne supérieure à cette quantité, alors que la proposition 9.9 implique l'optimalité du temps de calcul de l'algorithme PUISSANCES. Optimalité qui vaut même pour la localisation des puissances maximales bilatères.

Proposition 9.8
Il y a moins de $n \log_\Phi n$ occurrences de puissances locales maximales dans un mot de longueur n.

Preuve Le nombre de puissances locales maximales apparaissant à une position i donnée sur y est égal au nombre de carrés de mots primitifs apparaissant en i. Comme cette quantité est majorée par $\log_\Phi n$ d'après le corollaire 9.16 ci-dessous, on obtient le résultat. ∎

Proposition 9.9
Pour tout entier $c \geq 6$, le mot de Fibonacci f_c contient plus de $\frac{1}{6} F_c \log_2 F_c$ occurrences de carrés (de mots primitifs) et de puissances maximales (à droite), et plus de $\frac{1}{12} F_c \log_2 F_c$ occurrences de puissances maximales bilatères.

Preuve Notons $\xi(y)$ le nombre d'occurrences de carrés de mots primitifs qui sont facteurs de y. On montre par récurrence sur c, $c \geq 6$, que $\xi(f_c) \geq \frac{1}{6} F_c \log_2 F_c$.

Pour $c = 6$, on a $f_6 = \texttt{abaababa}$, $\xi(f_6) = 4$ et $\frac{1}{6} \times 8 \times 3 = 4$. Pour $c = 7$, on a $f_7 = \texttt{abaababaabaab}$, $\xi(f_7) = 11$ et $\frac{1}{6} \times 13 \times \log_2 13 < 9$.

Soit $c \geq 8$. Le mot f_c est égal à $f_{c-1}f_{c-2}$. On a l'égalité :

$$\xi(f_c) = \xi(f_{c-1}) + \xi(f_{c-2}) + r_c$$

où r_c est le nombre d'occurrences de carrés de f_c qui ne sont pas comptabilisés par $\xi(f_{c-1})$ ni par $\xi(f_{c-2})$, c'est-à-dire les occurrences de carrés qui chevauchent la séparation entre le préfixe f_{c-1} et le suffixe f_{c-2} de f_c. L'hypothèse de récurrence implique :

$$\xi(f_c) \geq \frac{1}{6}F_{c-1}\log_2 F_{c-1} + \frac{1}{6}F_{c-2}\log_2 F_{c-2} + r_c \ .$$

Pour obtenir le résultat cherché il suffit de montrer :

$$\frac{1}{6}F_{c-1}\log_2 F_{c-1} + \frac{1}{6}F_{c-2}\log_2 F_{c-2} + r_c \geq \frac{1}{6}F_c \log_2 F_c$$

ce qui est équivalent à :

$$r_c \geq \frac{1}{6}F_{c-1}\log_2 \frac{F_c}{F_{c-1}} + \frac{1}{6}F_{c-2}\log_2 \frac{F_c}{F_{c-2}}$$

en utilisant l'égalité $F_c = F_{c-1} + F_{c-2}$. Comme, pour $c > 4$,

$$\frac{F_c}{F_{c-1}} \leq \frac{F_5}{F_4} = \frac{5}{3} \ ,$$

il suffit de montrer :

$$r_c \geq \frac{1}{6}(F_{c-1} + F_{c-2})\log_2 \frac{8}{3}$$

ou encore :

$$r_c \geq \frac{1}{4}F_c \ .$$

On montre d'abord que f_c contient $F_{c-4} + 1$ occurrences de carrés de période F_{c-2} qui contribuent donc à r_c. Par réécriture à partir de la définition des mots de Fibonacci, on obtient $f_c = f_{c-2}f_{c-2}f_{c-5}f_{c-4}$, mais aussi $f_{c-5}f_{c-4} = f_{c-4}f_{c-7}f_{c-6}$, pour $c > 7$. Ainsi le mot $f_{c-2}f_{c-2}$ apparait dans f_c. Mais comme f_{c-4} est un préfixe à la fois de f_{c-2} et de $f_{c-5}f_{c-4}$, on obtient aussi F_{c-4} autres occurrences de carrés de période F_{c-2}.

On montre ensuite que f_c contient $F_{c-4} + 1$ occurrences de carrés de période F_{c-3} qui contribuent à nouveau à r_c. À partir de l'égalité $f_c = f_{c-2}f_{c-3}f_{c-3}f_{c-4}$, on voit que l'occurrence de $f_{c-3}f_{c-3}$ contribue à r_c de même que les F_{c-4} autres occurrences de carrés de période F_{c-3} qui se déduisent du fait que f_{c-4} est un préfixe de f_{c-3}.

En conclusion, pour $c > 7$ on arrive à l'inégalité

$$r_c \geq 2F_{c-4} \ ,$$

soit

$$r_c \geq \frac{1}{4} F_c \ ,$$

ce qui achève la récurrence et la preuve de la minoration du nombre d'occurrences de carrés.

Il y a autant d'occurrences de puissances maximales à droite que d'occurrences de carrés (une puissance maximale d'exposant e, $e > 1$, contient $e - 1$ occurrences de carrés mais aussi $e - 1$ occurrences de puissances maximales qui en sont des suffixes), ce qui donne la même borne pour cette quantité.

La seconde minoration qui porte sur le nombre d'occurrences de puissances maximales bilatères s'obtient au moyen d'une propriété combinatoire des mots de Fibonacci : f_c n'a aucun facteur de la forme u^4 avec $u \neq \varepsilon$ (voir exercice 9.10). Ainsi chaque occurrence de puissance maximale bilatère ne peut contenir que deux occurrences de carrés au plus, ce qui donne la seconde borne de l'énoncé. ∎

9.3 Recherche de carrés

Dans cette section, on considère des puissances d'exposant 2, les carrés. La localisation de toutes les occurrences des carrés dans un mot peut être réalisée avec l'algorithme de la section 9.2. On ne peut espérer trouver un algorithme asymptotiquement plus rapide (avec la représentation des puissances considérée) en raison du résultat de la proposition 9.9 dont une conséquence est que l'algorithme PUISSANCES est encore optimal même si on le restreint à ne produire que les carrés. Néanmoins, cela ne montre pas son optimalité pour la détection de tous carrés (et non de leurs occurrences) car un mot de longueur n en contient moins de $2n$ d'après le lemme 9.17 exposé plus loin. On commence par examiner la question de l'existence d'au moins un carré dans un mot, question qui se teste en temps linéaire lorsque l'alphabet est fixé. On étudie ensuite des bornes relatives au nombre de carrés de mots primitifs qui peuvent être présents dans un mot.

Existence d'un carré

Une des questions essentielle pour la suite est celle de la détection d'un carré dans la concaténation de deux mots sans carrés. Il s'en déduit un algorithme de test d'existence d'un carré dans un mot par la stratégie « diviser pour régner ». Cette méthode est ensuite améliorée par l'utilisation d'une factorisation spéciale du mot à tester.

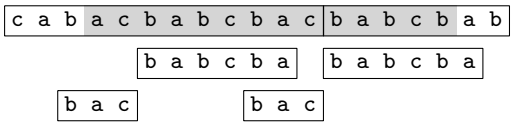


Figure 9.6 Support à la preuve du lemme 9.10. Un carré dans uv dont le centre est dans u est de la forme $ststt$ avec s suffixe de u et t préfixe de v. Ici $u = \texttt{cabacbabcbac}$ et $v = \texttt{babcbab}$. Le carré $(\texttt{acbabcb})^2$ est centré sur u. On a $\mathit{suff}_u[4] = |\texttt{bac}| = 3$, $p_{v,u}[5] = |\texttt{babcba}| = 6$, $i = 5$ et $|u| - i = 7$. Comme l'inégalité $\mathit{suff}_u[i-1] + p_{v,u}[i] \geq |u| - i$ est satisfaite, on en déduit les carrés $(\texttt{bacbabc})^2$, $(\texttt{acbabcb})^2$ et $(\texttt{cbabcba})^2$.

On rappelle de la section 3.3 la définition de la table suff_u, pour tout mot $u \in A^*$:

$$\mathit{suff}_u[i] = |\mathit{lsc}(u, u[0\,..\,i])| = \max\{|s| : s \preceq_{\mathit{suff}} u \text{ et } s \preceq_{\mathit{suff}} u[0\,..\,i]\} \ ,$$

pour $i = 0, 1, \ldots, |u| - 1$. Elle donne la longueur maximale des suffixes de u qui se terminent en chacune des positions sur u lui-même. Pour $u, v \in A^*$, on note $p_{v,u}$ la table définie, pour $j = 0, 1, \ldots, |u| - 1$, par :

$$p_{v,u}[j] = \max\{|t| : t \preceq_{\mathit{préf}} v \text{ et } t \preceq_{\mathit{préf}} u[j\,..\,|u| - 1]\} \ .$$

Cette deuxième table fournit la longueur maximale des préfixes de v qui commencent en chaque position sur u. Lorsque, par exemple, $u = \texttt{cabacbabcbac}$ et $v = \texttt{babcbab}$ (voir figure 9.6) on obtient les tables qui suivent :

i	0	1	2	3	4	5	6	7	8	9	10	11
$u[i]$	c	a	b	a	c	b	a	b	c	b	a	c
$\mathit{suff}_u[i]$	1	0	0	0	3	0	0	0	1	0	0	12
$p_{v,u}[i]$	0	0	2	0	0	6	0	1	0	2	0	0

Considérant deux mots u et v, on dit d'un carré w^2 qui apparait à la position i sur le mot $u \cdot v$ qu'il est un ***carré centré*** sur u lorsque $i + |w| \leq |u|$. Dans le cas contraire, on dit qu'il est centré sur v.

Lemme 9.10
Soient deux mots $u, v \in A^+$. Le mot $u \cdot v$ contient un carré centré sur u si et seulement si pour une position i sur u on a :

$$\mathit{suff}_u[i-1] + p_{v,u}[i] \geq |u| - i \ .$$

Preuve La preuve se construit avec l'aide de la figure 9.6. ∎

Les tables de l'exemple ci-dessus indiquent l'existence d'au moins deux carrés centrés sur u dans $u \cdot v$ car $\mathit{suff}_u[4] + p_{v,u}[5] \geq 7$ et $\mathit{suff}_u[4] + p_{v,u}[5] \geq 3$. En fait, il y a les carrés $(\texttt{bacbabc})^2$, $(\texttt{acbabcb})^2$, $(\texttt{cbabcba})^2$ et $(\texttt{cba})^2$.

Le calcul de la table suff_u est décrit dans la section 3.3, celui de $p_{v,u}$ résulte d'un algorithme de la section 2.6. Le temps de ces deux calculs est $O(|u|)$ en se limitant, pour le second, à ne préparer que le préfixe de v de longueur $|u|$ si jamais $|u| < |v|$. D'où le résultat qui suit.

Corollaire 9.11
Soient deux mots $u, v \in A^+$ sans carré. Tester si $u \cdot v$ contient un carré centré sur u peut se réaliser en temps $O(|u|)$.

Preuve En utilisant le lemme 9.10, il suffit de calculer les tables suff_u et $p_{v,u}$ en se limitant au préfixe de v de longueur $|u|$. Le calcul de ces deux tables se fait en temps $O(|u|)$ comme rappelé plus haut. Le reste du calcul consiste à tester l'inégalité du lemme 9.10, pour chaque position i sur u, ce qui prend encore un temps $O(|u|)$. Le résultat s'en déduit. ∎

On définit les fonctions booléennes *testg* et *testd*, qui prennent pour arguments des mots u et v sans carré, par :

$\mathit{testg}(u, v) = u \cdot v$ contient un carré centré sur u ,

et :

$\mathit{testd}(u, v) = u \cdot v$ contient un carré centré sur v .

Le corollaire 9.11 indique que le calcul de $\mathit{testg}(u, v)$ peut être réalisé en temps $O(|u|)$, et, par symétrie, celui de $\mathit{testd}(u, v)$ en temps $O(|v|)$. Ce résultat est utilisé dans l'analyse du temps d'exécution de l'algorithme RÉC-CARRÉ-DANS dont le code est donné ci-après. Pour un mot $y \in A^*$, l'opération RÉC-CARRÉ-DANS(y, n) retourne VRAI si et seulement si y contient un carré. Le principe du calcul est une stratégie « diviser pour régner » basée sur l'utilisation des fonctions *testg* et *testd*. Celles-ci sont supposées être réalisées par les algorithmes TESTG et TESTD respectivement.

```
RÉC-CARRÉ-DANS(y, n)
 1  si n ≤ 1 alors
 2        retourner FAUX
 3  sinonsi RÉC-CARRÉ-DANS(y[0 .. ⌊n/2⌋ − 1], ⌊n/2⌋) alors
 4        retourner VRAI
 5  sinonsi RÉC-CARRÉ-DANS(y[⌊n/2⌋ .. n − 1], ⌈n/2⌉) alors
 6        retourner VRAI
 7  sinonsi TESTG(y[0 .. ⌊n/2⌋], y[⌊n/2⌋ + 1 .. n − 1]) alors
 8        retourner VRAI
 9  sinonsi TESTD(y[0 .. ⌊n/2⌋], y[⌊n/2⌋ + 1 .. n − 1]) alors
10        retourner VRAI
11  sinon retourner FAUX
```

Proposition 9.12
L'opération RÉC-CARRÉ-DANS(y, n) *retourne* VRAI *si et seulement si* y *contient un carré. Le calcul se fait en temps* $O(n \times \log n)$.

Preuve La preuve de bon fonctionnement se fait par simple récurrence sur la longueur n de y.

En notant $T(n)$ le temps d'exécution de RÉC-CARRÉ-DANS sur un mot de longueur n, on obtient au moyen du corollaire 9.11 les formules de récurrence $T(1) = \alpha$ et, pour $n > 1$, $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \beta n$, où α et β sont des constantes. La résolution de cette récurrence donne le résultat annoncé (voir exercice 1.13). ∎

Il est possible de réduire le temps d'exécution du test d'existence d'un carré dans y en utilisant une stratégie plus fine que la précédente, et qui, tout en étant du genre « diviser pour régner », n'est plus nécessairement équilibrée (en la taille des sous-problèmes). La stratégie est basée sur une factorisation de y appelée sa f-factorisation.

La ***f-factorisation*** de $y \in A^+$ est la suite des facteurs $u_0, u_1, \ldots, u_k$ de y définis itérativement comme suit. On a d'abord $u_0 = y[0]$. Ensuite, en supposant $u_0, u_1, \ldots, u_{j-1}$ déjà définis, avec $u_0 u_1 \ldots u_{j-1} \prec_{préf} y$ et $j > 0$, soient $i = |u_0 u_1 \ldots u_{j-1}|$ (on a $0 < i < n - 1$) et w le plus long préfixe de $y[i \,..\, n-1]$ qui possède au moins deux occurrences dans $y[0 \,..\, i-1] \cdot w$. Alors :

$$u_j = \begin{cases} w & \text{si } w \neq \varepsilon \ , \\ y[i] & \text{sinon} \ . \end{cases}$$

On note que le second cas de la définition se produit lorsque $y[i]$ est une lettre qui n'apparait pas dans $y[0 \,..\, i-1]$. On remarque également que tous les facteurs de la f-factorisation sont non vides.

En prenant l'exemple du mot $y =$ `abaabbaabbaababa`, on obtient pour f-factorisation la suite `a`, `b`, `a`, `ab`, `baabbaab`, `aba`, laquelle est une décomposition de y : $y =$ `a` $\cdot$ `b` $\cdot$ `a` $\cdot$ `ab` $\cdot$ `baabbaab` $\cdot$ `aba`.

Lemme 9.13
Soit $\langle u_0, u_1, \ldots, u_k \rangle$ *la f-factorisation de* $y \in A^+$. *Le mot* y *contient un carré si et seulement si l'une de ces trois conditions suivantes est satisfaite pour un certain indice* j, $0 < j \leq k$:

1. $|u_0 u_1 \ldots u_{j-1}| \leq pos_y(u_j) + |u_j| < |u_0 u_1 \ldots u_j|$;
2. $testg(u_{j-1}, u_j)$ *ou* $testd(u_{j-1}, u_j)$ *est vrai ;*
3. $j > 1$ *et* $testd(u_0 u_1 \ldots u_{j-2}, u_{j-1} u_j)$ *est vrai.*

Preuve On commence par montrer que si l'une des conditions est satisfaite, y contient un carré. Soit j le plus petit indice pour lequel une des conditions est satisfaite. Si la condition 1 est satisfaite, l'occurrence courante de u_j ainsi que sa première occurrence dans y se chevauchent

ou sont adjacentes sans coïncider. On en déduit un carré à la position $pos_y(u_j)$.

Si la condition 1 n'est pas satisfaite, le mot u_j ne contient pas de carré car il est de longueur 1 ou est facteur de $u_0u_1 \dots u_{j-1}$ qui n'en contient pas (ce qui peut se montrer par récurrence sur j en utilisant cette remarque). Par définition des fonctions *testg* et *testd* et puisque u_{j-1} et u_j sont sans carrés, si $testg(u_{j-1}, u_j)$ ou $testd(u_{j-1}, u_j)$ est vrai, le mot $u_{j-1}u_j$ contient un carré qui est donc aussi un carré de y. En revanche, si $testg(u_{j-1}, u_j)$ et $testd(u_{j-1}, u_j)$ sont faux, $u_{j-1}u_j$ ne contient pas de carré ; mais la condition 3 indique l'existence d'un carré dans y car les arguments de *testg* sont des mots sans carré.

Réciproquement, soit j le plus petit entier pour lequel $u_0u_1 \dots u_j$ contient un carré, et soit ww, $w \neq \varepsilon$, ce carré. On a $0 < j < n$ car u_0 est sans carré, et le mot $u_0u_1 \dots u_{j-1}$ est sans carré par définition de l'entier j. Si la condition 1 n'est pas satisfaite, comme dans ce cas u_j est de longueur 1 ou est facteur de $u_0u_1 \dots u_{j-1}$, il est sans carré. Si la condition 2 n'est pas satisfaite $u_{j-1}u_j$ est lui aussi sans carré. Il reste alors à montrer que le carré ww est centré sur $u_{j-1}u_j$. Dans la situation contraire, l'occurrence de la seconde moitié du carré ww recouvre entièrement u_{j-1}, c'est-à-dire que ce mot possède une occurrence qui n'est pas un suffixe dans w. Mais cela contredit la maximalité de la longueur de u_{j-1} dans la définition de la f-factorisation. La condition 3 est donc satisfaite, ce qui termine la preuve. ∎

L'algorithme Carré-dans implante directement le test d'existence d'un carré à partir des conditions énoncées dans le lemme 9.13. Le calcul de la f-factorisation peut être réalisé au moyen de l'arbre compact des suffixes de y (section 5.2) ou de l'automate des suffixes de y (section 5.4). On obtient ainsi un test linéaire lorsque l'alphabet est fixé.

```
Carré-dans(y)
 1  (u_0, u_1, ..., u_k) ← f-factorisation de y
 2  pour j ← 1 à k faire
 3        si |u_0u_1...u_{j-1}| ≤ pos(u_j) + |u_j| < |u_0u_1...u_j| alors
 4              retourner VRAI
 5        sinonsi Testg(u_{j-1}, u_j) alors
 6              retourner VRAI
 7        sinonsi Testd(u_{j-1}, u_j) alors
 8              retourner VRAI
 9        sinonsi j > 1 et Testd(u_0u_1...u_{j-2}, u_{j-1}u_j) alors
10              retourner VRAI
11  retourner FAUX
```

Théorème 9.14
L'opération Carré-dans(y) *retourne* VRAI *si et seulement si le mot y contient un carré. Le calcul se fait en temps $O(|y| \times \log \operatorname{card} A)$.*

Preuve Le bon fonctionnement de l'algorithme est conséquence directe du lemme 9.13.

On peut vérifier qu'on peut calculer la f-factorisation de y au moyen de l'automate des suffixes de y, ou même pendant la construction de celui-ci. On peut d'ailleurs effectuer le test de la ligne 3 pendant ce calcul, sans que cela modifie la majoration asymptotique du temps de la construction. Le temps de cette étape est donc $O(|y| \times \log \operatorname{card} A)$ (section 5.4).

La somme des temps d'exécution des tests effectués aux lignes 5, 7 et 9 est proportionnel à $\sum_{j=1}^{k}(|u_{j-1}| + |u_j| + |u_{j-1}u_j|)$ d'après le corollaire 9.11, ce qui est majoré par $2|y|$.

Le temps total est donc $O(|y| \times \log \operatorname{card} A)$. ∎

Nombre de carrés préfixes ou facteurs

On appelle ***carré préfixe*** un carré d'un mot qui est un préfixe de ce mot.

Le lemme qui suit présente une propriété combinatoire qui est à l'origine du décompte des carrés préfixes du corollaire 9.16. La majoration du nombre de carrés est utilisée dans la section précédente pour majorer le nombre d'occurrences de puissances maximales dans un mot (proposition 9.8) et majorer le temps d'exécution de l'algorithme Puissances.

Lemme 9.15 (lemme des trois carrés préfixes)
Soient trois mots $u, v, w \in A^+$ tels que $u^2 \prec_{préf} v^2 \prec_{préf} w^2$ et u est primitif. Alors $|u| + |v| \leq |w|$.

Preuve On suppose par l'absurde que $|u| + |v| > |w|$, ce qui, avec l'hypothèse, implique $v \prec_{préf} w \prec_{préf} vu \prec_{préf} v^2$. Le mot $t = v^{-1}w$ satisfait alors $t \prec_{préf} u$ et $|t|$ est une période de v (car v apparait aux positions $|v|$ et $|w|$ sur w^2 et que $|w| - |v| = |t| \leq |v|$).

On considère deux cas, suivant que u^2 est un préfixe de v ou non (voir figure 9.7).

Cas 1. Dans cette situation, u^2, qui est un préfixe de v admet deux périodes différentes, $|u|$ et $|t|$, qui satisfont $|u| + |t| < |u^2|$. Le lemme de périodicité s'applique et montre que $\operatorname{pgcd}(|u|, |t|)$ est aussi une période de u^2. Mais, comme $\operatorname{pgcd}(|u|, |t|) \leq |t| < |u|$, cela entraine que u n'est pas primitif, en contradiction avec les hypothèses.

Cas 2. Dans ce cas, v est un préfixe de u^2. Le mot v possède deux périodes distinctes : $|u|$ et $|t|$. Si $|u| + |t| \leq |v|$, le lemme de périodicité s'applique à v et on obtient la même contradiction que dans le cas précédent. On peut donc supposer le contraire, c'est-à-dire que l'inégalité $|u| + |t| > |v|$ est satisfaite.

Le mot $s = u^{-1}v$ est à la fois un préfixe de u et un suffixe de v. Sa longueur satisfait $|s| < |t|$ à cause de l'inégalité précédente et est une période de u (car u apparait aux positions $|u|$ et $|v|$ sur w^2 et que

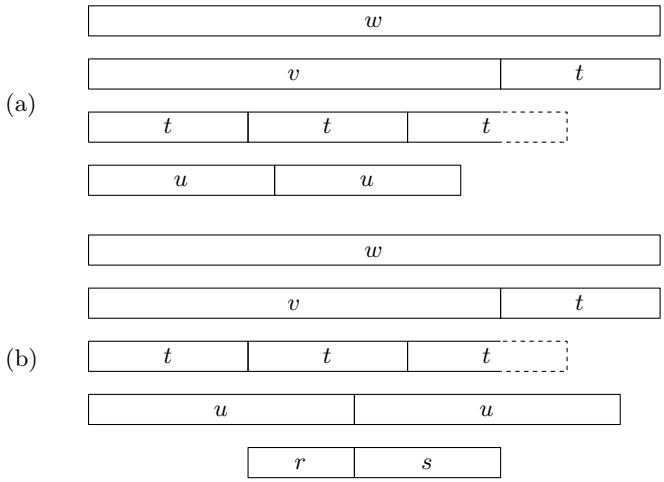


Figure 9.7 Illustration pour les deux situations impossibles envisagées dans la preuve du lemme 9.15. **(a)** Cas 1. Le mot u^2 est un préfixe du mot v. **(b)** Cas 2. Le mot v est un préfixe du mot u^2.

$|s| = |v| - |u| \leq |u|$). Soit finalement $r = t^{-1}u$. On a ainsi $v = t \cdot r \cdot s$. On obtient une contradiction en montrant à nouveau ci-dessous que u possède une période qui divise strictement sa longueur.

Comme $|t|$ est une période de v, le mot $r \cdot s$ est aussi un préfixe de v (proposition 1.4). Et comme $|r \cdot s| < |r| + |t| = |u|$, il est même un préfixe propre de u. Il apparait donc dans w^2 aux positions $|t|$ et $|u|$. Ce qui prouve qu'il a pour période $|u| - |t| = |r|$. Il a aussi pour période $|s|$ qui est une période de u. Le lemme de périodicité s'applique à $r \cdot s$ qui a ainsi pour période $p = \text{pgcd}(|r|, |s|)$. En fait p est aussi une période de u car p divise $|s|$ qui est une période de u.

Considérons maintenant le mot u. Il a pour périodes p et $|t|$ avec l'inégalité $p + |t| \leq |r| + |t| = |u|$. Le lemme de périodicité s'applique à u qui a donc pour période $q = \text{pgcd}(p, |t|)$. Mais q divise $|t|$ et $|r|$, donc également leur somme $|t| + |r| = |u|$. Cela contredit la primitivité de u et termine le cas 2.

Comme les cas 1 et 2 sont impossibles, l'hypothèse $|u| + |v| > |w|$ est absurde, ce qui prouve l'inégalité de l'énoncé. ∎

Considérons, par exemple, le mot `aabaabaaabaabaabaaab` qui a pour préfixes les carrés $\texttt{a}^2$, $(\texttt{aab})^2$, $(\texttt{aabaaba})^2$ et $(\texttt{aabaabaaab})^2$. Les trois mots `a`, `aab` et `aabaaba` satisfont les hypothèses du lemme 9.15, et leurs longueurs satisfont l'inégalité : $1+3 < 7$. Pour les trois mots `aab`, `aabaaba` et `aabaabaaab` qui satisfont aussi les hypothèses du lemme 9.15, on a l'égalité : $3+7 = 10$. Ce qui montre que l'inégalité de l'énoncé du lemme est précise.

Corollaire 9.16
Tout mot y, $|y| > 1$, possède moins de $\log_\Phi |y|$ préfixes qui sont des carrés de mots primitifs, à savoir :

$$\text{card}\{u : u \text{ primitif et } u^2 \preceq_{préf} y\} < \log_\Phi |y| \ .$$

Preuve Notons :

$$\zeta(y) = \text{card}\{u : u \text{ primitif et } u^2 \preceq_{préf} y\} \ .$$

Pour commencer, on montre par récurrence sur c, $c \geq 1$, que :

$$\zeta(y) \geq c \text{ implique } |y| \geq 2F_{c+1} \ .$$

Pour $c = 1$, on a $|y| \geq 2 = 2F_2$. Pour $c = 2$, on vérifie que $|y| \geq 6 > 2F_3 = 4$ (par exemple, on a $\zeta(\texttt{aabaab}) = 6$).

Supposons $\zeta(y) \geq c \geq 3$. Soient $u, v, w \in A^+$ les trois plus longs mots primitifs distincts dont les carrés sont des préfixes de y. On a $u^2 \prec_{préf} v^2 \prec_{préf} w^2$. Les mots u^2 et v^2 satisfont donc respectivement $\zeta(u^2) \geq c-2$ et $\zeta(v^2) \geq c-1$. Par hypothèse de récurrence, on obtient $|u^2| \geq 2F_{c-1}$ et $|v^2| \geq 2F_c$.

Le lemme 9.15 donne l'inégalité $|u| + |v| \leq |w|$, ce qui implique

$$|y| \geq |w^2| \geq |u^2| + |v^2| \geq 2F_{c-1} + 2F_c = 2F_{c+1}$$

et termine la récurrence.

Comme $F_{c+1} \geq \Phi^{c-1}$ et $\Phi < 2$, on obtient $|y| \geq 2\Phi^{c-1} > \Phi^c$, soit $c < \log_\Phi |y|$, ce qui signifie que y possède moins de $\log_\Phi |y|$ préfixes, carrés de mots primitifs, comme annoncé. ∎

Le mot de Fibonacci $f_7 = \texttt{abaababaabaab}$ possède deux carrés préfixes. On peut vérifier, pour $i \geq 5$, que f_i possède $i-5$ carrés préfixes et que ${f_{i-2}}^2$ en est le plus long. L'exercice 9.11 donne une autre suite de mots qui possèdent le maximum possible de carrés préfixes pour une longueur donnée.

Une application directe du lemme précédent montre qu'un mot de longueur n ne peut contenir en facteur plus de $n \log_\Phi n$ carrés de mots primitifs. En fait, cette borne peut être affinée comme le fait apparaitre la proposition suivante.

Proposition 9.17
Tout mot y, $|y| > 4$, ne possède pas plus de $2|y| - 6$ facteurs qui sont carrés de mots primitifs, à savoir :

$$\text{card}\{u : u \text{ primitif et } u^2 \preceq_{fact} y\} \leq 2|y| - 6 \ .$$

Preuve Soit :

$$E = \{u^2 : u \text{ primitif et } u^2 \preceq_{fact} y\} \ .$$

Considérons trois mots u^2, v^2 et w^2 de E, $u^2 \prec_{préf} v^2 \prec_{préf} w^2$. D'après le lemme 9.15 des trois carrés, on a $|u| + |v| \leq |w|$ et donc $2|u| < |w|$, ce qui implique $u^2 \prec_{préf} w$.

Supposons que i soit une position de u^2, v^2 et w^2 sur y. Alors i n'est pas la plus grande position de u^2 sur y. Ainsi, une position i ne peut être la plus grande position que d'au plus deux mots de E. Cela montre que $\operatorname{card} E \leq 2|y|$.

On note ensuite que la position $|y| - 1$ n'est position d'aucun mot de E, et que chacune des positions $|y| - 2$, $|y| - 3$, $|y| - 4$, $|y| - 5$, ne peut être la plus grande position que d'au plus un mot de E. Cela précise la majoration précédente et donne la borne $2|y| - 6$ de l'énoncé. ∎

9.4 Tri des suffixes

Une adaptation de l'algorithme PARTITIONNEMENT (voir section 9.1) permet de classer les suffixes de y en ordre lexicographique. Il permet simultanément de calculer les préfixes communs aux suffixes de y dans le but d'en réaliser une table des suffixes (chapitre 4). Avec cette méthode, le calcul ne demande qu'un espace mémoire linéaire alors que la préparation de la table des suffixes proposée dans la section 4.5 nécessite un espace $O(n \log n)$, avec $n = |y|$.

Calcul incrémental des rangs des suffixes

Rappelons que, pour $k > 0$, on note $R_k[i]$ le rang (compté à partir de la position 0) de $prem_k(y[i \,..\, n-1])$ dans la liste ordonnée des mots de l'ensemble $\{prem_k(u) : u \text{ suffixe non vide de } y\}$ et que $i \equiv_k j$ si et seulement si $R_k[i] = R_k[j]$ (voir section 4.4). Cela est aussi équivalent à $E_k[i] = E_k[j]$ avec la notation de la section 9.1.

Afin de classer les suffixes de y, on transforme l'algorithme PARTITIONNEMENT en l'algorithme RANGS. Le code de ce dernier algorithme est donné ci-après. La modification consiste à maintenir les classes de la partition dans l'ordre lexicographique croissant des débuts de longueur k des suffixes. Pour cela, les classes de la partition sont constituées en une liste, l'ordre de la liste étant un élément essentiel pour obtenir le classement final. Le numéro de la classe d'une position i, noté $E_k[i]$ dans la section 9.1 et dont la valeur peut être choisie relativement librement, est remplacé ici par le rang de la classe dans la liste des classes, $R_k[i]$, qui a une valeur indépendante de l'implantation de l'algorithme.

Un autre point de l'algorithme PARTITIONNEMENT est modifié afin d'obtenir l'algorithme RANGS : il s'agit de la gestion des petite classe. Parmi les sous-classes d'une classe C qui se trouve partagée pendant le partitionnement, il est nécessaire de distinguer les classes qui se trouvent avant la classe de taille maximale retenue, et celles qui se trouvent après

cette même classe, dans la liste des sous-classes de C. Elles sont stockées respectivement dans deux listes appelées *Avant* et *Après*, leur union constituant l'ensemble des petites classes, *Petites*, considéré dans l'algorithme PARTITIONNEMENT.

Dans l'algorithme RANGS les classes de la partition sont des ensembles qu'il n'est plus nécessaire de gérer comme des listes.

Enfin, comme l'algorithme PARTITIONNEMENT, l'algorithme RANGS n'est pas donné dans tous ses détails ; en particulier on sous-entend que les listes de sous-classes et les classes jumelles sont réinitialisées à la liste vide après chaque étape.

RANGS(y, n)

pour $r \leftarrow 0$ **à** $\operatorname{card} alph(y) - 1$ **faire**
    $C_r \leftarrow \emptyset$
pour $i \leftarrow 0$ **à** $n - 1$ **faire**
    $r \leftarrow$ rang de $y[i]$ dans la liste classée des lettres de $alph(y)$
    $C_r \leftarrow C_r \cup \{i\}$
Avant $\leftarrow \langle\rangle$
Après $\leftarrow \langle C_0, C_1, \ldots, C_{\operatorname{card} alph(y)-1}\rangle$
$k \leftarrow 1$
tantque *Avant* $\cdot$ *Après* $\neq \langle\rangle$ **faire**
    ▷ Invariant : $i \in C_r$ si et seulement si $R_k[i] = r$
    pour chaque $P \in$ *Avant* $\cdot$ *Après*, séquentiellement **faire**
        pour chaque $i \in P \setminus \{0\}$ **faire**
            soit C la classe de $i - 1$
            soit C_P la classe jumelle de C
            transférer $i - 1$ de C dans C_P
        pour chaque couple (C, C_P) considéré **faire**
            si $P \in$ *Avant* **alors**
                SsClAv$[C] \leftarrow$ *SsClAv*$[C] \cdot \langle C_P\rangle$
            sinon *SsClAp*$[C] \leftarrow$ *SsClAp*$[C] \cdot \langle C_P\rangle$
    Avant $\leftarrow \langle\rangle$
    Après $\leftarrow \langle\rangle$
    pour chaque classe C considérée à l'étape précédente,
      dans l'ordre de la liste des classes **faire**
        si $C \neq \emptyset$ **alors**
            SsCl$[C] \leftarrow$ *SsClAv*$[C] \cdot \langle C\rangle \cdot$ *SsClAp*$[C]$
        sinon *SsCl*$[C] \leftarrow$ *SsClAv*$[C] \cdot$ *SsClAp*$[C]$
        dans la liste des classes, remplacer C par
          les éléments de *SsCl*$[C]$ dans l'ordre de cette liste
        $G \leftarrow$ une classe de taille maximale dans *SsCl*$[C]$
        Avant $\leftarrow$ *Avant* $\cdot \langle$classes avant G dans *SsCl*$[C]\rangle$
        Après $\leftarrow$ *Après* $\cdot \langle$classes après G dans *SsCl*$[C]\rangle$
    $k \leftarrow k + 1$
retourner la permutation des positions
  associée à la liste des classes

Théorème 9.18
L'algorithme RANGS *classe les suffixes de* $y \in A^*$ *de longueur* n *en ordre lexicographique. Autrement dit, la permutation* $p = \text{RANGS}(y, n)$ *satisfait la condition :*

$$y[p[0]\,..\,n-1] < y[p[1]\,..\,n-1] < \cdots < y[p[n-1]\,..\,n-1] \ .$$

Preuve On commence par montrer que l'équivalence

$i \in C_r$ si et seulement si $R_k[i] = r$

est un invariant de la boucle **tantque**. Cela revient à montrer que la classe de i est (strictement) avant la classe de j dans la liste des classes à l'étape k si et seulement si $R_k[i] < R_k[j]$, à chaque étape. Il suffit de montrer l'implication directe car i et j appartiennent à la même classe à l'étape k si et seulement si $i \equiv_k j$ d'après la preuve de l'algorithme PARTITIONNEMENT qui s'applique ici.

On suppose la condition satisfaite au début de l'étape k et on examine l'effet des instructions de la boucle **tantque**.

Soient i, j deux positions telles que $i \in C_r$, $j \in C_s$ et $r < s$ où C_r et C_s sont des classes selon $\equiv_{k+1}$. Si $i \not\equiv_k j$, l'ordre relatif des classes de i et j étant conservé grâce à l'instruction à la ligne 26, la classe de i précède celle de j à l'étape k. Par hypothèse, on a donc $R_k[i] < R_k[j]$, inégalité qui implique $R_{k+1}[i] < R_{k+1}[j]$ par la définition de R.

On suppose maintenant $i \equiv_k j$. Soit C la classe commune à i et j selon l'équivalence $\equiv_k$.

Supposons que C_r et C_s soient deux sous-classes avant de C (dans $SsClAv[C]$). Alors $i+1$ et $j+1$ appartiennent à deux classes P' et P'' qui sont dans cet ordre dans la liste *Avant*. Par hypothèse on a donc :

$$R_k[i+1] < R_k[j+1]$$

(soit $prem_k(y[i+1\,..\,n-1]) < prem_k(y[j+1\,..\,n-1])$), et donc aussi :

$$R_{k+1}[i] < R_{k+1}[j]$$

(soit $prem_{k+1}(y[i\,..\,n-1]) < prem_{k+1}(y[j\,..\,n-1])$) (voir figure 9.1), d'après la façon de constituer la liste *Avant* à la ligne 28. Le raisonnement est le même si i est placé dans une sous-classe avant de C et j dans une sous-classe après de C, ou si i et j sont placés dans deux sous-classes après de C.

Supposons pour finir que i ne soit pas touché et que j soit placé dans une classe après à l'étape k. Alors, $i+1 \in G$ où G est une sous-classe de taille maximale de sa classe d'origine, ou bien $i+1 = n$. La position $j+1$ appartient à une sous-classe après de la même classe d'origine car $i \equiv_{k-1} j$. Comme la sous-classe de $j+1$ est située après G de par la constitution de *Après* (ligne 29), on a comme précédemment $R_k[i+1] < R_k[j+1]$, puis $R_{k+1}[i] < R_{k+1}[j]$. Le raisonnement est analogue quand i est placé dans une classe avant et que j n'est pas touché.

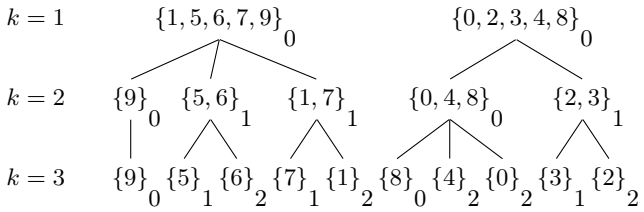


Figure 9.8 L'opération RANGS appliquée à `babbbaaaba` pour le classement de ses suffixes et le calcul des préfixes communs aux suffixes consécutifs. La suite finale $9, 5, 6, \ldots$ donne les suffixes en ordre lexicographique croissant : `a` $<$ `aaaba` $<$ `aaba` $< \cdots$. Pour chaque classe C, la valeur $LPC[C]$ est notée en indice de C. La valeur $LPC[\{1\}] = 2$ indique, par exemple, que le plus long préfixe commun aux suffixes qui sont aux positions 7 et 1, à savoir `aba` et `abbbaaaba`, est de longueur 2.

Cela termine la preuve de l'invariant.

Pour $k = 1$, on constate que la condition est remplie après l'initialisation. L'algorithme s'arrête lorsque *Avant* · *Après* est vide, c'est-à-dire lorsque la partition est stabilisée, ce qui n'intervient que lorsque chaque classe est réduite à un singleton. Dans cette situation il vient de la condition que la permutation des positions obtenue correspond à la suite croissante des valeurs de R, c'est-à-dire la suite croissante des suffixes dans l'ordre lexicographique. ∎

L'exemple de la figure 9.8 reprend celui de la figure 9.5. À la ligne $k = 2$ on a une petite classe avant, $\{9\}$, et deux petites classes après, $\{1, 7\}$ et $\{2, 3\}$. Le partitionnement à cette étape a lieu en prenant les petites classes dans cet ordre. Le partitionnement au moyen de $\{9\}$ a pour effet d'extraire 8 de sa classe $\{0, 4, 8\}$ qui se partage en $\{8\}$ et $\{0, 4\}$ dans cet ordre car $\{9\}$ est une classe avant. Avec $\{1, 7\}$, 0 est extrait de sa classe $\{0, 4\}$ qui se partage en $\{4\}$ et $\{0\}$ dans cet ordre car $\{1, 7\}$ est une classe après. Les positions 7, 2 et 3 sont utilisées de la même façon. Ce qui conduit respectivement au partage de $\{5, 6\}$ en $\{5\}$ et $\{6\}$, de $\{1, 7\}$ en $\{7\}$ et $\{1\}$, et finalement de $\{2, 3\}$ en $\{3\}$ et $\{2\}$. On obtient ainsi la partition de la ligne $k = 3$ qui est la partition finale.

Calcul des préfixes communs

On indique comment étendre l'algorithme RANGS pour obtenir un calcul simultané des plus longs préfixes communs aux suffixes consécutifs dans leur suite classée.

Pour cela, on affecte à chaque classe C une valeur notée $LPC(C)$ qui est la longueur maximale des préfixes communs entre les éléments de C et ceux de la classe précédente dans la liste des classes. Ces valeurs sont toutes initialisées à 0. On sait qu'à l'étape k tous les éléments d'une même classe possèdent un même préfixe de longueur k.

Le calcul de $LPC(C')$ intervient quand C' est une nouvelle classe, sous-classe d'une classe C pour laquelle $LPC(C)$ est défini. La définition de $LPC(C')$ peut se faire au cours de l'instruction à la ligne 26 en utilisant la relation :

$$LPC[C'] = \begin{cases} LPC[C] & \text{si } C' \text{ est la première classe de } SsCl[C] \ , \\ k & \text{pour les autres classes de } SsCl[C] \ . \end{cases}$$

Il est facile de voir que cette règle conduit au calcul correct de LPC.

La figure 9.8 illustre un exemple de calcul des préfixes communs. À l'étape $k = 2$, la classe $\{0, 4, 8\}$ se partage en $\{8\}$, $\{4\}$, $\{0\}$. On obtient ainsi $LPC[\{8\}] = LPC[\{0, 4, 8\}] = 0$ pour la première sous-classe, puis $LPC[\{4\}] = LPC[\{0\}] = k = 2$ pour les deux autres.

À la fin de l'exécution de l'algorithme Rangs, chaque classe contient un seul élément. Si $C = \{i\}$, on a $LPC[C] = LPC[i]$ avec la notation de la section 4.3. Le reste du calcul de la table LPC qui est une des composantes de la table des suffixes peut être conduit comme au chapitre 4.

L'analyse du temps d'exécution effectuée pour l'algorithme Partitionnement vaut aussi pour Rangs. La description précédente montre que le calcul des préfixes communs aux suffixes ne modifie pas la majoration asymptotique du temps d'exécution de l'algorithme. On obtient ainsi le résultat suivant qui conclut le chapitre.

Théorème 9.19
La préparation de la table des suffixes d'un mot de longueur n peut être effectuée en temps $O(n \log n)$ dans un espace linéaire. ∎

Notes

La méthode de partitionnement décrite dans ce chapitre trouve son origine dans un algorithme de minimisation d'automates déterministes dû à Hopcroft (1971). L'algorithme de la section 9.1 en est une variante qui ne s'applique pas seulement aux mots mais aussi aux graphes (voir Cardon et Crochemore, 1982). Des prolongements de la méthode ont été proposés par Paige et Tarjan (1987).

L'utilisation du partitionnement des positions d'un mot pour déterminer les puissances locales est due à Crochemore (1981). Apostolico et Preparata (1983), ont montré que le calcul peut être effectué au moyen de l'arbre de suffixes. Slisenko (1983) a aussi proposé une méthode qui repose sur une structure de données semblable à l'automate des suffixes.

L'algorithme Réc-carré-dans est dû à Main et Lorentz (1979) qui ont donné un algorithme direct afin d'implanter la fonction *testg* (voir exercice 9.8). L'algorithme est aussi à la base d'une méthode de recherche de toutes les occurrences de carrés proposée par les mêmes auteurs en 1984 et dont la complexité est la même que celle des méthodes ci-dessus. Ils ont aussi montré que l'algorithme est optimal parmi ceux qui n'uti-

lisent que les comparaisons du type $=$ et $\neq$ sur les lettres des mots. L'algorithme CARRÉ-DANS (voir Crochemore, 1986) est lui aussi optimal dans la classe des algorithmes qui comparent les lettres au moyen de $<$, $=$ et $>$ en supposant une relation d'ordre sur l'alphabet. Une méthode basée sur le nommage (voir chapitre 4) permet d'atteindre le même temps de calcul (voir Main et Lorentz, 1985).

Pour une utilisation de l'arbre des suffixes à la détection des carrés dans un mot, on peut se référer à Stoye et Gusfield (1998).

Le lemme des trois carrés préfixes est de Crochemore et Rytter (1995). Une autre preuve due à Diekert se trouve dans le chapitre de Mignosi et Restivo (2000) de [97]. Ce chapitre traite de façon approfondie des périodicités dans les mots.

La borne de $2n$ sur le nombre de carrés dans un mot de longueur n (proposition 9.17) a été établie par Fraenkel et Simpson (1998). Le nombre exact de carrés dans le mot de Fibonacci, qui a inspiré les auteurs précédents, a été évalué par Iliopoulos, Moore et Smyth (1997). Kolpakov et Kucherov (1998, 1999) ont étendu le résultat en montrant que le nombre d'occurrences de périodicités maximales bilatères est encore linéaire.

L'algorithme de la section 9.4 est voisin de celui décrit par Manber et Myers (1993) pour la préparation d'une table des suffixes.

Exercices

9.1 (Arbre de carrés)
Indiquer comment transformer l'arbre compact des suffixes d'un mot y pour stocker la totalité des facteurs de y qui sont des carrés d'un mot primitif.

Donner un algorithme linéaire qui effectue la transformation. [*Aide :* voir Stoye et Gusfield (1998).]

9.2 (Puissance fractionnaire)
On appelle exposant fractionnaire d'un mot x non vide la quantité

$$\exp(x) = |x|/p\acute{e}r(x) \ .$$

Montrer, pour tout entier $k > 1$, que $\exp(x) = k$ si et seulement si $x = u^k$ pour un mot u primitif. (Autrement dit la notion d'exposant introduite dans le chapitre 1 et la notion d'exposant fractionnaire coïncident dans ce cas.)

Écrire un algorithme linéaire qui calcule les exposants fractionnaires de tous les préfixes d'un mot y.

Décrire un algorithme fonctionnant en temps $O(n \log n)$ pour le calcul des puissances fractionnaires maximales d'un mot de longueur n. [*Aide :* voir Main (1989).]

9.3 (Puissance maximale)
Montrer qu'un mot de longueur n contient $O(n)$ occurrences de puissances fractionnaires maximales. Donner un algorithme qui les calcule toutes en temps $O(n \times \log \operatorname{card} A)$. [*Aide :* voir Kolpakov et Kucherov (1998).]

9.4 (Morphisme de Thue-Morse)
Un ***chevauchement*** est un mot de la forme $auaua$ avec $a \in A$ et $u \in A^*$. Montrer qu'un mot x contient (en facteur) un chevauchement si et seulement si il possède un facteur non vide v pour lequel $\exp(v) > 2$.

Sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}\}$, on considère le morphisme (voir exercice 1.2) $g\colon A^* \to A^*$ défini par $g(\mathtt{a}) = \mathtt{ab}$ et $g(\mathtt{b}) = \mathtt{ba}$. Montrer que, pour tout entier $k \geq 0$, le mot $g^k(\mathtt{a})$ ne contient aucun chevauchement. [*Aide :* voir Lothaire [96].]

9.5 (Mot sans chevauchement)
Sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}\}$ on considère la substitution g de l'exercice 9.4 ainsi que les ensembles :

$$\begin{aligned} E &= \{\mathtt{aabb}, \mathtt{bbaa}, \mathtt{abaa}, \mathtt{babb}\} \ , \\ F &= \{\mathtt{aabab}, \mathtt{bbaba}\} \ , \\ G &= \{\mathtt{abba}, \mathtt{baab}, \mathtt{baba}, \mathtt{abab}\} \ , \\ H &= \{\mathtt{aabaa}, \mathtt{bbabb}\} \ . \end{aligned}$$

Soit $x \in A^*$ un mot sans chevauchement. Montrer que, si x a un préfixe dans $E \cup F$, $x[j] \neq x[j-1]$ pour chaque entier j impair satisfaisant la condition $3 \leq j \leq |x| - 2$. Montrer que, si x a un préfixe dans $G \cup H$, $x[j] \neq x[j-1]$ pour chaque entier j pair satisfaisant la condition $4 \leq j \leq |x| - 2$.

Montrer que, si $|x| > 6$, x se décompose de façon unique en $d_x \cdot u \cdot f_x$ avec $d_x, f_x \in \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{bb}\}$ et $u \in A^*$.

Montrer que le mot x se décompose de façon unique en

$$d_1 d_2 \dots d_r \cdot g^{r-1}(u) \cdot f_r \dots f_2 f_1$$

avec $|u| < 7$, $r \in \mathbf{N}$ et

$$d_s, f_s \in \{\varepsilon, g^{s-1}(\mathtt{a}), g^{s-1}(\mathtt{b}), g^{s-1}(\mathtt{aa}), g^{s-1}(\mathtt{bb})\} \ .$$

En déduire que le nombre de mots sans chevauchement de longueur n croît de façon polynomiale avec n. [*Aide :* voir Restivo et Salemi (1985).]

9.6 (Test de chevauchement)
Déduire de la décomposition des mots sans chevauchement de l'exercice 9.5 un algorithme linéaire afin de tester si un mot contient un chevauchement. [*Aide :* voir Kfoury (1988).]

9.7 (Pas de carré)
Sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, on considère le morphisme (voir exercice 1.2) $h\colon A^* \to A^*$ défini par $h(\mathtt{a}) = \mathtt{abc}$, $h(\mathtt{b}) = \mathtt{ac}$ et $h(\mathtt{c}) = \mathtt{b}$. Montrer, pour tout entier $k \geq 0$, que le mot $h^k(\mathtt{a})$ ne contient aucun carré. [*Aide :* voir Lothaire [96].]

9.8 (Test gauche)
Détailler la preuve du lemme 9.10.

Donner une implantation de la fonction *testg* qui calcule $testg(u, v)$ en temps $O(|u|)$ au moyen d'un espace supplémentaire constant. [*Aide :* calculer $p_{v,u}[i]$ à la volée pour des valeurs de i bien choisies ; voir Main et Lorentz (1979).]

9.9 (Seulement trois carrés)
Montrer que 3 est le plus petit entier pour lequel il existe des mots arbitrairement longs $y \in \{\mathtt{a}, \mathtt{b}\}^*$ satisfaisant

$$\mathrm{card}\{u : u \neq \varepsilon \text{ et } u^2 \preceq_{fact} y\} = 3 \ .$$

[*Aide :* voir Fraenkel et Simpson (1995).]

9.10 (Puissance quatrième)
Montrer que $\mathtt{b}^2$, $\mathtt{a}^3$, $\mathtt{babab}$ et $\mathtt{aabaabaa}$ ne sont pas facteurs des mots de Fibonacci.

Montrer que si $u^2 \preceq_{fact} f_k$, u est un conjugué d'un mot de Fibonacci. [*Aide :* lorsque $|u| > 2$ on pourra étudier le cas $u \in \mathtt{a}\{\mathtt{a}, \mathtt{b}\}^+$ et vérifier que le cas $u \in \mathtt{b}\{\mathtt{a}, \mathtt{b}\}^+$ se ramène au précédent.]

En déduire qu'aucun mot de Fibonacci ne contient de puissance quatrième (facteur d'exposant 4). [*Aide :* voir Karhumäki (1983).]

9.11 (Carrés préfixes)
On considère la suite $\langle g_i : i \in \mathbf{N}\rangle$ de mots de $\{\mathtt{a}, \mathtt{b}\}^*$ définie par $g_0 = \mathtt{a}$, $g_1 = \mathtt{aab}$, $g_2 = \mathtt{aabaaba}$ et, pour $i \geq 3$, $g_i = g_{i-1}g_{i-2}$.

Vérifier que ${g_i}^2$ possède $i + 1$ carrés préfixes.

Montrer que si $y \in \{\mathtt{a}, \mathtt{b}\}^*$ possède $i + 1$ préfixes qui sont des carrés de mots primitifs, alors $|y| \geq 2|g_i|$.

Pour $i \geq 3$, montrer que si $y \in \{\mathtt{a}, \mathtt{b}\}^*$ possède $i + 1$ préfixes qui sont des carrés de mots primitifs et $|y| = 2|g_i|$, alors, à une permutation près des lettres $\mathtt{a}$ et $\mathtt{b}$, $y = {g_i}^2$. [*Aide :* voir Crochemore et Rytter (1995).]

9.12 (Puissances préfixes)
Soit un entier $k \geq 2$ et soient trois mots $u, v, w \in A^+$ qui satisfont les conditions : $u^k \prec_{préf} v^k \prec_{préf} w^k$ et u est primitif. Montrer que $|u| + (k-1)|v| \leq |w|$. [*Aide :* pour $k \geq 3$ on peut utiliser le lemme de primitivité.]

9.13 (Puissances préfixes, la fin)
Soit un entier $k \geq 2$. Montrer qu'un mot y, $|y| > 1$, possède moins de $\log_{\alpha(k)} |y|$ préfixes qui sont des puissances k-ièmes de mots primitifs, à savoir :

$$\text{card}\{u : u \text{ primitif et } u^k \preceq_{préf} y\} < \log_{\alpha(k)} |y| \ ,$$

où :

$$\alpha(k) = \frac{k - 1 + \sqrt{(k-1)^2 + 4}}{2} \ .$$

9.14 (Beaucoup de carrés)
Donner une famille infinie de mots qui contiennent en facteur le maximum possible de carrés de mots primitifs.

9.15 (Rangs)
Implanter l'algorithme RANGS.

9.16 (Factorisation parfaite)
Soit $x \in A^+$. Montrer qu'il existe une position i sur x qui satisfait les deux propriétés : $i < 2 \times pér(x)$ et au plus un préfixe de $x[i \,..\, |x| - 1]$ est de la forme u^3, u primitif. [*Aide :* voir Galil et Seiferas (1983) ou [5].]

9.17 (Périodicités préfixes)
Soient $u, v \in A^+$ deux mots primitifs tels que $|u| < |v|$.

Montrer que :

$$|lpc(uu, vv)| < |u| + |v| - \text{pgcd}(|u|, |v|) \ .$$

Montrer qu'il existe un conjugué v' de v pour lequel :

$$|lpc(u^\infty, v'v')| \leq \frac{2}{3}(|u| + |v|) \ .$$

Montrer que chaque inégalité est précise. [*Aide :* utiliser le lemme des périodicités et voir Breslauer, Jiang et Jiang (1997) ; voir aussi Mignosi et Restivo (2000) dans [97].]

Bibliographie

Mise à jour en février 2011.

Livres

Ouvrages d'algorithmique du texte.

[1] D. Adjeroh, T. Bell et A. Mukherjee. *The Burrows-Wheeler Transform*. Springer, 2008.

[2] A. Apostolico et Z. Galil, éditeurs. *Pattern Matching Algorithms*. Oxford University Press, 1997.

[3] C. Charras and T. Lecroq. *Handbook of Exact String Matching Algorithms*. King's College London Publications, 2004.

[4] M. Crochemore, C. Hancart et T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[5] M. Crochemore et W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[6] M. Crochemore et W. Rytter. *Jewels of Stringology*. World Scientific Press, 2002.

[7] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[8] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Volume 1(2) of *Foundations and Trends in Theoretical Computer Science*, Now Publishers Inc., Hanover, MA, 2005.

[9] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings—Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[10] W. F. Smyth. *Computing Patterns in Strings*. Addison-Wesley Longman, 2003.

[11] G. A. Stephen. *String Searching Algorithms*. World Scientific Press, 1994.

Compilations d'articles

Recueils d'articles sur l'algorithmique du texte qui, hormis le premier, ont été édités comme numéros spéciaux de revues ou actes de conférences.

[12] J.-I. Aoe, éditeur. *String Pattern Matching Strategies*. IEEE Computer

Society Press, 1994.

[13] A. Apostolico, éditeur. *String Algorithmics and its Applications. Algorithmica*, 12(4/5), 1994.

[14] A. Apostolico et Z. Galil, éditeurs. *Combinatorial Algorithms on Words*, vol. 12. Springer, 1985.

[15] M. Crochemore et L. Gąsieniec, éditeurs. *Matching Patterns. J. Discret. Algorithms*, 1(1), 2000.

[16] M. Crochemore, éditeur. *Proceedings of the 1st Annual Symposium on Combinatorial Pattern Matching. Theoret. Comput. Sci.*, 92(1), 1992.

[17] A. Apostolico, M. Crochemore, Z. Galil et U. Manber, éditeurs. *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, Tucson, Arizona, 1992. Lecture Notes in Computer Science n° 664. Springer.

[18] A. Apostolico, M. Crochemore, Z. Galil et U. Manber, éditeurs. *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, Padoue, Italie, 1993. Lecture Notes in Computer Science n° 684. Springer.

[19] M. Crochemore et D. Gusfield, éditeurs. *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, Asilomar, Californie, 1994. Lecture Notes in Computer Science n° 807. Springer.

[20] Z. Galil et E. Ukkonen, éditeurs. *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, Espoo, Finlande, 1995. Lecture Notes in Computer Science n° 937. Springer.

[21] D. S. Hirschberg et E. W. Myers, éditeurs. *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, Laguna Beach, Californie, 1996. Lecture Notes in Computer Science n° 1075. Springer.

[22] A. Apostolico et J. Hein, éditeurs. *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, Aarhus, Danemark, 1997. Lecture Notes in Computer Science n° 1264. Springer.

[23] M. Farach-Colton. *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, Piscataway, New Jersey, 1998. Lecture Notes in Computer Science n° 1448. Springer.

[24] M. Crochemore et M. Paterson, éditeurs. *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching*, Warwick, Angleterre, 1999. Lecture Notes in Computer Science n° 1645. Springer.

[25] R. Giancarlo et D. Sankoff, éditeurs. *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, Montréal, Canada, 2000. Lecture Notes in Computer Science n° 1848. Springer.

[26] A. Amir et G. M. Landau, éditeurs. *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, Jerusalem, Israel, 2001. Lecture Notes in Computer Science n° 2089. Springer.

[27] A. Apostolico et M. Takeda, éditeurs. *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, Fukuoka, Japan, 2002. Lecture Notes in Computer Science n° 2373. Springer.

[28] R. A. Baeza-Yates, E. Chávez et M. Crochemore, éditeurs. *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, Morelia, Michocán, 2003. Lecture Notes in Computer Science n° 2676. Sprin-

ger.

[29] S. C. Sahinalp, S. Muthukrishnan et U. Dogrusöz, éditeurs. *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching*, Istanbul, Turkey, 2004. Lecture Notes in Computer Science n° 3109. Springer.

[30] A. Apostolico, M. Crochemore et K. Park, éditeurs. *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching*, Jeju Island, Korea, 2005. Lecture Notes in Computer Science n° 3537. Springer.

[31] M. Lewenstein et G. Valiente, éditeurs. *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching*, Barcelona, Spain, 2006. Lecture Notes in Computer Science n° 4009. Springer.

[32] B. Ma et K. Zhang, éditeurs. *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching*, London, Ontario, Canada, 2007. Lecture Notes in Computer Science n° 4580. Springer.

[33] P. Ferragina et G. M. Landau, éditeurs. *Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching*, Pisa, Italy, 2008. Lecture Notes in Computer Science n° 5029. Springer.

[34] G. Kucherov et E. Ukkonen, éditeurs. *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching*, Lille, France, 2009. Lecture Notes in Computer Science n° 5577. Springer.

[35] A. Amir et L. Parida, éditeurs. *Proceedings of the 21th Annual Symposium on Combinatorial Pattern Matching*, New York, NY, USA, 2010. Lecture Notes in Computer Science n° 6129. Springer.

[36] R. Baeza-Yates et N. Ziviani, éditeurs. *Proceedings of the 1st South American Workshop on String Processing*, Minas Gerais, Brésil, 1993. Universidade Federal de Minas Gerais.

[37] R. Baeza-Yates et U. Manber, éditeurs. *Proceedings of the 2nd South American Workshop on String Processing*, Valparaiso, Chili, 1995. University of Chile.

[38] N. Ziviani, R. Baeza-Yates et K. Guimarães, éditeurs. *Proceedings of the 3rd South American Workshop on String Processing*, Recife, Brésil, 1996. Carleton University Press.

[39] R. Baeza-Yates, éditeur. *Proceedings of the 4th South American Workshop on String Processing*, Valparaiso, Chili, 1997. Carleton University Press.

[40] *Proceedings of the 5th International Symposium on String Processing and Information Retrieval*, Santa Cruz de la Sierra Bolivia, 1998. IEEE Computer Society.

[41] *Proceedings of the 6th International Symposium on String Processing and Information Retrieval Symposium & International Workshop on Groupware*, Cancun, Mexico, 1999. IEEE Computer Society.

[42] *Proceedings of the 7th International Symposium on String Processing Information Retrieval*, A Coruña, Spain, 2000. IEEE Computer Society.

[43] *Proceedings of the 8th International Symposium on String Processing Information Retrieval*, Laguna de San Rafael, Chile, 2001. IEEE Computer Society.

[44] A. H. F. Laender et A. L. Oliveira, éditeurs. *Proceedings of the 9th Inter-*

national Symposium on String Processing Information Retrieval, Lisbon, Portugal, 2002. Lecture Notes in Computer Science n° 2476. Springer.

[45] M. A. Nascimento, E. S. de Moura et A. L. Oliveira, éditeurs. *Proceedings of the 10th International Symposium on String Processing Information Retrieval*, Manaus, Brazil, 2003. Lecture Notes in Computer Science n° 2857. Springer.

[46] A. Apostolico et M. Melucci, éditeurs. *Proceedings of the 11th International Symposium on String Processing Information Retrieval*, Padova, Italy, 2004. Lecture Notes in Computer Science n° 3246. Springer.

[47] M. P. Consens et G. Navarro, éditeurs. *Proceedings of the 12th International Symposium on String Processing Information Retrieval*, Buenos Aires, Argentina, 2005. Lecture Notes in Computer Science n° 3772. Springer.

[48] F. Crestani, P. Ferragina et M. Sanderson, éditeurs. *Proceedings of the 13th International Symposium on String Processing Information Retrieval*, Glasgow, UK, 2006. Lecture Notes in Computer Science n° 4209. Springer.

[49] N. Ziviani et R. A. Baeza-Yates, éditeurs. *Proceedings of the 14th International Symposium on String Processing Information Retrieval*, Santiago, Chile, 2007. Lecture Notes in Computer Science n° 4726. Springer.

[50] A. Amir, A. Turpin et A. Moffat, éditeurs. *Proceedings of the 15th International Symposium on String Processing Information Retrieval*, Melbourne, Australia, 2008. Lecture Notes in Computer Science n° 5280. Springer.

[51] J. Karlgren, J. Tarhio et H. Hyyrö, éditeurs. *Proceedings of the 16th International Symposium on String Processing Information Retrieval*, Saariselkä, Finland, 2009. Lecture Notes in Computer Science n° 5721. Springer.

[52] E. Chávez et S. Lonardi, éditeurs. *Proceedings of the 17th International Symposium on String Processing Information Retrieval*, Los Cabos, Mexico, 2010. Lecture Notes in Computer Science n° 6393. Springer.

[53] R. Capocelli, éditeur. *Sequences, Combinatorics, Compression, Security and Transmission*, 1990. Springer.

[54] R. Capocelli, A. De Santis et U. Vaccaro, éditeurs. *Sequences II*, 1993. Springer.

[55] B. Carpentieri, A. De Santis, U. Vaccaro et J. A. Storer, éditeurs. *Compression and Complexity of Sequences*, Los Alamitos, Californie, 1987. IEEE Computer Society.

[56] J. Holub, éditeur. *Proceedings of the Prague Stringology Club Workshop'96*, 1996. Czech Technogical University, Prague.

[57] J. Holub, éditeur. *Proceedings of the Prague Stringology Club Workshop'97*, 1997. Czech Technogical University, Prague.

[58] J. Holub et M. Šimánek, éditeurs. *Proceedings of the Prague Stringology Club Workshop'98*, 1998. Czech Technogical University, Prague.

[59] J. Holub et M. Šimánek, éditeurs. *Proceedings of the Prague Stringology Club Workshop'99*, 1999. Czech Technogical University, Prague.

[60] M. Balík et M. Šimánek, éditeurs. *Proceedings of the Prague Stringology Club Workshop'2000*, Bratislava, Slovaquie, 2000. Czech Technogical University, Prague.

[61] M. Baliík et M. Šimánek, éditeurs. *Proceedings of the Prague Stringology Conference*, Prague, Czech Republic, 2001. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

[62] M. Baliík et M. Šimánek, éditeurs. *Proceedings of the Prague Stringology Conference*, Prague, Czech Republic, 2002. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

[63] M. Šimánek, éditeur. *Proceedings of the Prague Stringology Conference*, Prague, Czech Republic, 2003. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

[64] M. Šimánek et Jan Holub, éditeurs. *Proceedings of the Prague Stringology Conference*, Prague, Czech Republic, 2004. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

[65] J. Holub et M. Šimánek, éditeurs. *Proceedings of the Prague Stringology Conference*, Prague, Czech Republic, 2005. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

[66] J. Holub et J. Žďárek, éditeurs. *Proceedings of the Prague Stringology Conference*, Prague, Czech Republic, 2006. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

[67] J. Holub et J. Žďárek, éditeurs. *Proceedings of the Prague Stringology Conference*, Prague, Czech Republic, 2008. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

[68] J. Holub et J. Žďárek, éditeurs. *Proceedings of the Prague Stringology Conference*, Prague, Czech Republic, 2009. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

[69] J. Holub et J. Žďárek, éditeurs. *Proceedings of the Prague Stringology Conference*, Prague, Czech Republic, 2010. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

Sites sur la Toile

Quelques sites consacrés à l'algorithmique du texte sur la Toile. Ils contiennent des bibliographies régulièrement mises à jour, des animations sur les algorithmes, des pointeurs sur les acteurs du domaine et des informations diverses sur le sujet.

[70] S. Lonardi. *Pattern Matching Pointers*.

http://www.cs.ucr.edu/~stelo/pattern.html

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

[72] C. Charras et T. Lecroq. *Sequence Comparison*. http://www-igm.univ-mlv.fr/~lecroq/seqcomp/

[73] T. Lecroq. *Bibliographie sur la recherche de motifs*. http://www-igm.univ-mlv.fr/~lecroq/

Applications

Quelques références sur deux grands domaines d'application de l'algorithmique du texte que sont la recherche documentaire, y compris le traitement automatique du langage naturel, et l'analyse des séquences génomiques.

[74] T. K. Attwood et D. J. Parry-Smith. *Introduction to Bioinformatics*. Addison-Wesley Longman Limited, 1999.

[75] R. Baeza-Yates et B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[76] R. Durbin, S. Eddy, A. Krogh et G. Mitchison. *Biological Sequence Analysis Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[77] W. B. Frakes et R. Baeza-Yates, éditeurs. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[78] M. Gross et D. Perrin, éditeurs. *Electronic Dictionaries and Automata in Computational Linguistics*. Lecture Notes in Computer Science n° 377. Springer, 1989.

[79] E. W. Myers, éditeur. *Computational Molecular Biology. Algorithmica*, 13(1/2), 1995.

[80] P. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, 2000.

[81] É. Roche et Y. Schabes, éditeurs. *Finite State Language Processing*. The MIT Press, 1997.

[82] G. Salton. *Automatic Text Processing*. Addison-Wesley, 1989.

[83] D. Sankoff et J. Kruskal, éditeurs. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Cambridge University Press, seconde édition, 1999.

[84] J. C. Setubal et J. Meidanis. *Introduction to Computional Molecular Biology*. PWS Publishing Company, 1997.

[85] G. Valiente. *Combinatorial Pattern Matching Algorithms in Computational Biology Using Perl and R*. Chapman & Hall/CRC, 2009.

[86] M. S. Waterman. *Introduction to Computational Biology*. Chapman & Hall, 1995.

Algorithmique et combinatoire

Livres d'algorithmique générale contenant au moins un chapitre sur l'algorithmique du texte, et sélection d'ouvrages présentant des aspects formels en rapport avec le sujet.

[87] A. V. Aho, J. E. Hopcroft et J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[88] A. V. Aho, R. Sethi et J. D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.

[89] M.-P. Béal. *Codage symbolique*. Masson, 1993.

[90] D. Beauquier, J. Berstel et P. Chrétienne. *Éléments d'algorithmique*. Masson, 1992.

[91] J. Berstel. *Transductions and Context-Free Languages*. Teubner, 1979.

[92] T. H. Cormen, C. E. Leiserson et R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[93] P. Flajolet et R. Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

[94] G. H. Gonnet et R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991.

[95] M. T. Goodrich et R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.

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

[97] M. Lothaire, éditeur. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.

[98] M. Lothaire, éditeur. *Applied Combinatorics on Words*. Cambridge University Press, 2005.

[99] D. Perrin et J.-É. Pin. *Infinite Words: Automata, Semigroups, Logic and Games*. Academic Press Inc, 2003.

[100] J.-É. Pin. *Variétés de langages formels*. Masson, 1984.

[101] R. Sedgewick et P. Flajolet. *Introduction à l'analyse des algorithmes*. Vuibert Informatique, 1996.

[102] W. Szpankowski. *Average Case Analysis of Algorithms on Sequences*. Wiley-Interscience Series in Discrete Mathematics, 2001.

Index

Figurent dans l'index qui suit : les auteurs cités, les mots-clefs indiqués en gras dans le texte, les identificateurs des algorithmes, les notations qui ne sont pas des symboles en indiquant la page de leur définition et un choix de termes du texte.