

Université de Rouen

Quelques aspects de l'algorithmique du texte

Mémoire présenté par Thierry Lecroq
pour l'obtention de l'Habilitation à Diriger des Recherches

spécialité : Informatique

soutenue le 8 décembre 2000 devant le jury composé de

Maxime CROCHEMORE	Professeur, Université de Marne-la-Vallée
Serge DULUCQ	Professeur, Université Bordeaux I, rapporteur
Jean-Pierre DUVAL	Professeur, Université de Rouen
Michel HABIB	Professeur, Université Montpellier II, rapporteur
Jean Frédéric MYOUPO	Professeur, Université de Picardie Jules Verne, Amiens
William F. SMYTH	Professeur, McMaster University, Canada, rapporteur

1	Introduction	1
2	Définitions et notations	3
3	Recherche exacte de mots	5
3.1	Les différentes méthodes	6
	Simulation d'un automate déterministe	7
	Algorithmes rapides en pratique	7
	Algorithmes optimaux en espace	8
	Utilisation de vecteurs binaires	8
3.2	Complexité du problème	8
3.3	Algorithme de Boyer-Moore	9
3.4	Algorithme de Galil	11
3.5	Algorithme de Smyth	12
3.6	Algorithme Turbo-BM	12
3.7	Algorithme d'Apostolico-Giancarlo	14
3.8	Calcul de la fonction de décalage	15
3.9	L'algorithme Reverse Factor	17
3.10	L'algorithme Turbo Reverse Factor	20
3.11	Algorithme Pos Tree	22
3.12	Algorithme Skip Search	23
3.13	Algorithme Alpha Skip Search	24
3.14	Recherche d'un ensemble fini de mots	25
	Algorithme RF Multiple	27
	Algorithme Dawg Match	28
3.15	Résultats expérimentaux	32
4	Alignement de séquences	37
4.1	Alignement	37
4.2	Alignement optimal	40
	Calcul de la distance d'édition	41
	Calcul d'un alignement optimal	43
	Calcul de tous les alignements optimaux	43
4.3	Plus long sous-mot commun	45
	Calcul par programmation dynamique	45
4.4	Algorithmes systoliques	48
	Plus long sous-mot commun à deux mots	48
5	Animation d'algorithmes	51
5.1	Exact String Matching Algorithms	53
5.2	Sequence comparison	56
6	Recherche dans les séquences musicales	61
6.1	Recherche $\delta - \gamma$ approchée	61
	δ -, γ - et (δ, γ) -approximation	61

6.2	δ -approximation	62
	Algorithme δ -Tuned-Boyer-Moore	62
	Algorithme δ -Skip Search	63
6.3	(δ, γ) -approximation	63
6.4	Résultats expérimentaux	64
7	Recherche dans les séquences biologiques	67
7.1	Oracle des facteurs	67
7.2	Calcul de répétitions	68
7.3	Applications	69
	Répétitions dans les séquences génomiques	70
	Comparaison de séquences	71
	Compression de données	71
8	Perspectives	73
	Bibliographie	75

Remerciements

Serge Dulucq et Michel Habib ont accepté d'être rapporteur de cette Habilitation. Je les remercie chaleureusement de m'avoir accordé un peu de leur temps.

Durant cette dernière décade j'ai eu la chance de travailler avec des gens formidables.

À tout seigneur tout honneur, j'ai l'immense plaisir de collaborer avec Maxime Crochemore après avoir effectué mon stage de DÉA et ma thèse sous sa direction. Sa gentillesse et sa disponibilité sont connues de tous, il m'a de plus permis de participer à des projets passionnants. Je lui exprime ici toute ma gratitude.

Jean-Pierre Duval m'a accueilli dans son laboratoire et a toujours favoriser au mieux le développement de mes projets scientifiques. Il a de plus accepté de faire partie du jury. Je l'en remercie grandement.

En plus de partager le même directeur de thèse et le même bureau, nous avons avec Christophe Hancart de nombreuses conceptions communes sur beaucoup de sujets. De plus c'est Christophe qui a développé le style \LaTeX pour la rédaction de [CHL00a] et qui m'a permis de l'utiliser pour ce mémoire. Puisseons-nous encore longtemps travailler ensemble.

Joël Alexandre a permis au groupe ABISS d'évoluer dans les meilleures conditions. L'ambiance régnant parmi les membres de ce groupe est un vrai régal.

La constitution de nos deux sites Web avec Christian Charras, et la « lutte » pour leur maintien et survie a toujours été passionnante.

Les membres du Département d'Informatique de Rouen, Martine Léonard en tête, ont toujours rendu le quotidien plus agréable. Maryse Brochet, la fidèle secrétaire du Département, a souvent été d'une aide précieuse.

La gestion du DESS EGOISt avec Hélène Dauchel et Dominique Cellier s'est toujours passée dans la bonne humeur.

Zvi Galil m'a accueilli à Columbia University en 1993.

Wojtek Rytter m'a invité deux fois à Varsovie, il a su rendre les deux séjours agréables.

Leszek Gąsieniec et Wojtek Rytter m'ont très gentiment accueilli à

Liverpool en février 1998.

Costas Iliopoulos m'a accueilli dans les meilleures conditions au King's College London et à Curtin University of Technology.

Les réunions du groupe IREM sont toujours intéressantes.

Gregory Kucherov nous a permis de participer au projet 8 de l'Institut franco-russe A. M. Liapunov en 1998 et 1999.

Nous avons beaucoup voyagé avec Laurent Mouchard et j'espère que nous continuerons.

Jean Frédéric Myoupo m'a permis de travailler, toujours dans la bonne humeur, sur un modèle qui ne m'était pas familier. Il a accepté de participer au jury.

Toutes les rencontres, où je me suis rendu à l'Institut Pasteur, organisées par Marie-France Sagot ont toujours été très enrichissantes.

Nous avons entrepris avec Bill Smyth un échange de nos manuscrits respectifs et communiquons nos corrections et remarques. Il a accepté avec enthousiasme d'être rapporteur de ce mémoire.

Qu'ils soient tous ici remerciés.

Résumé

Nous nous intéressons ici à différents aspects de l'algorithmique du texte. Le premier aspect que nous traitons est la recherche exacte d'un mot fixé dans des textes. Nous présentons deux familles d'algorithmes de recherche exacte de mot : les algorithmes de type « Boyer-Moore » qui reconnaissent des suffixes de plus en plus longs du mot dans le texte et les algorithmes de type « Reverse Factor » qui reconnaissent des facteurs de plus en plus longs du mot dans le texte. Des résultats expérimentaux permettent de déterminer les zones de performances des meilleurs algorithmes en termes de nombres de comparaisons entre lettres du mot et lettres du texte et en termes de temps d'exécution.

Nous nous intéressons ensuite à la notion d'alignement entre deux mots et en particulier au calcul d'un alignement sur un réseau systolique.

La diffusion et l'animation de ces deux thèmes ont été réalisées grâce à deux sites Web dynamiques : nous en détaillons la conception.

Enfin nous nous attarderons sur deux domaines d'applications : la recherche approchée dans les séquences musicales et la recherche de répétitions dans les séquences biologiques.

Abstract

We are interested here in different aspects of text algorithmics. The first aspect that we deal with is exact string matching of a fixed pattern in texts. We present two families of exact string matching algorithms: "Boyer-Moore" type algorithms that match longer and longer suffixes of the pattern in the text and "Reverse Factor" type algorithms that match longer and longer factors of the pattern in the text. Experimental results enable to determinate the performance areas of the best algorithms concerning the number of comparisons between pattern letters and text letters and concerning running times.

We are then interested in the notion of alignment between two words and specially the computation of an alignment on a systolic network.

Distribution and animation of these two themes have been realized in two dynamic Web sites: we detail their conception.

Finally we will look more closely at two applications: approximate string matching in musical sequences and computation of repetitions in biological sequences.

Avant-propos

Le présent document résume nos activités de recherche effectuées pendant les dix dernières années (1990–2000). Les différentes citations bibliographiques apparaissant dans ce document prennent deux formes : nos publications sont du type

« [initiales des auteurs + année de parution] »

alors que les références à des travaux de tiers sont simplement numérotés. Les deux listes de références apparaissent séparément à la fin du document.

1 Introduction

Un état de l'art des différents axes de recherche et des différentes méthodes de l'algorithmique du texte est apparu dans [CL96b].

Bien que les données peuvent être mémorisées à l'aide de divers supports, le texte demeure la forme principale pour échanger l'information. C'est particulièrement évident en littérature ou en linguistique où les données sont constituées de corpus et de dictionnaires énormes. Ceci s'applique aussi à l'informatique où une grande quantité de données est enregistrée dans des fichiers linéaires. Et c'est également le cas en biologie moléculaire où les molécules biologiques peuvent souvent être représentées par des suites de nucléotides ou d'acides aminés. En outre, la quantité de données disponible dans ce domaine tend à doubler tous les dix-huit mois.

C'est pour toutes ces raisons que les algorithmes de recherche d'informations doivent être efficaces même si la vitesse et la capacité de mémoire des ordinateurs augmentent régulièrement.

Le chapitre 2 présente les différents termes et notions communes au domaine.

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

On s'intéressera ici à la recherche d'un mot dans un texte. Les tech-

niques utilisées dans ce type de problème servent généralement de bases aux autres types de recherche.

Le problème de la recherche d'un mot x de longueur m dans un texte y de longueur n possède deux variantes. Lorsque le texte y est connu à l'avance, un prétraitement est alors autorisé sur celui-ci, pour construire une structure d'index en temps et espace $O(n)$ et la localisation de toutes les occurrences de x dans y peut alors s'effectuer en temps et espace $O(m)$. Lorsque le mot x est connu à l'avance, un prétraitement est alors autorisé sur celui-ci, pour construire une structure de données en temps et espace $O(m)$ et la localisation de toutes les occurrences de x dans y peut alors s'effectuer en temps $O(n)$. On s'intéressera dans le chapitre 3 au second cas.

La comparaison entre deux mots est un autre champ d'investigation important. On veut évaluer le degré de similarité, ou de manière duale la distance entre deux mots. Ce problème intervient de manière cruciale en biologie moléculaire où le degré de similarité entre deux séquences est une indication d'une homologie éventuelle entre ces deux séquences. On présentera dans le chapitre 4 les méthodes classiques de résolution de ce problème qui utilise des techniques de programmation dynamique. On en donnera une solution systolique dans le cas restreint de la recherche d'un plus long sous-mot commun à deux mots.

Le chapitre 5 présente deux sites Web dynamiques abordant les notions traitées aux chapitres 3 et 4. Ces notions sont effectivement difficiles et il est important de disposer d'outils permettant de les mieux appréhender.

Les chapitres 6 et 7 présentent des applications de techniques algorithmiques du texte aux séquences musicales et biologiques respectivement. Ces deux domaines demandent en effet d'adapter des méthodes générales à des situations très spécialisées.

Enfin le chapitre 8 décrit les champs d'activité où un travail important reste à effectuer.

2 Définitions et notations

On considère un ensemble fini A de lettres appelé **alphabet**. Un **mot** est une suite de lettres de A . Un mot w de m lettres s'écrit $w = w[0 \dots m-1]$. La **longueur** de w se note $|w|$. On note ε le mot vide de longueur 0. Pour un mot w de longueur m , une **position** i sur w est un entier compris entre 0 et $m-1$. La lettre de w à la position i est $w[i]$.

Un mot u est un **préfixe** d'un mot w si w peut s'écrire $w = uv$, on note $u \preceq_{\text{préf}} w$. Un mot v est un **suffixe** d'un mot w si w peut s'écrire $w = uv$, on note $v \preceq_{\text{suff}} w$. Un mot z est un **facteur** d'un mot w si w peut s'écrire $w = uzv$, on note $z \preceq_{\text{fact}} w$.

Le **renversé** d'un mot $x = x[0 \dots m-1]$ est le mot $x^R = x[m-1]x[m-2] \dots x[0]$.

Un **bord** d'un mot w est un facteur qui est à la fois préfixe et suffixe. Le **bord** d'un mot est le plus long de ses bords (il peut être le mot vide).

Un entier p est une **période** d'un mot w si pour tout i tel que $0 \leq i < m-p$, $w[i] = w[i+p]$. La **période** d'un mot est la plus petite de ses périodes non nulles (elle peut être égale à $|w|$). Un mot est dit **périodique** si sa période est inférieure à la moitié de sa longueur.

Il y a une relation évidente entre les bords d'un mot et ses périodes. La figure 2.1 illustre cette relation avec le mot **abaabaabaab**.

Soit un mot x de longueur m , un mot $w = x[i_1]x[i_2] \dots x[i_k]$ avec $1 \leq i_1 < i_2 < \dots < i_k \leq m$ est appelé un **sous-mot** de x , on note $w \preceq_{\text{smot}} x$.

Soient deux mots x et y , un mot w est un **sous-mot commun** à x et à y s'il est à la fois un sous-mot de x et un sous mot de y . Il est un **plus**

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

Figure 2.1 Relation entre bords et périodes sur le mot **abaabaabaab**.

long sous-mot commun à x et à y s'il est un sous-mot commun à x et à y et s'il est de longueur maximale.

3 Recherche exacte de mots

Les travaux du présents chapitres ont fait l'objet des publications suivantes : [CL96a] présente un panorama, ainsi que des appliquestes pour une trentaine d'algorithmes de recherche exacte de mot. Les algorithmes de type « Boyer-Moore » sont exposés dans [CHL00a]. Quelques méthodes de recherche de mots sont expliquées dans [CL96c] et [Lec00a]. L'algorithme Turbo-BM est présenté dans [CCG⁺91] et [CCG⁺94]. La version de l'algorithme Apostolico-Giancarlo ainsi que son analyse sont exposées dans [CL97] et [CHL00b]. Le calcul de la fonction de décalage de bon suffixe est expliquée et analysée dans [CHL00b]. L'algorithme Reverse Factor est présenté dans [Lec92b]. L'algorithme Turbo Reverse Factor est exposé dans [CCG⁺94]. L'algorithme PosTree est exposé dans [CL00]. Les algorithmes Skip Search et Alpha Skip Search sont présentés dans [CLP98] et [CLP00]. L'algorithme Dawg Match est présenté dans [CCG⁺99]. Des résultats expérimentaux sont exhibés dans [Lec95], [Lec98a] et [Lec00a].

La recherche de mots est un sujet très important dans le domaine plus large du traitement des textes. Les algorithmes de recherche de mots sont les composants de base utilisés dans les réalisations des logiciels pratiques existant sous la plupart des systèmes d'exploitation. D'ailleurs, ils soulignent les méthodes de programmation qui servent de paradigmes dans d'autres domaines de l'informatique (conception de système ou de logiciel). En conclusion, ils jouent également un rôle important en informatique théorique en fournissant des problèmes intéressants.

La recherche de mot consiste à trouver une, ou plus généralement, toutes les occurrences d'un mot (plus généralement appelée un **motif**) dans un **texte**. Le mot est noté $x = x[0..m-1]$; sa longueur est égale à m . Le texte est noté $y = y[0..n-1]$; sa longueur est égale n .

Les applications admettent deux types de solution selon que le mot

ou le texte est donné en premier et est donc fixé. Des algorithmes basés sur l'utilisation d'automates ou des propriétés combinatoires des mots sont généralement mis en œuvre pour prétraiter le mot et résoudre le premier type de problème. La notion d'index réalisés par des arbres ou des automates est utilisée dans le deuxième type de solution. Ce chapitre étudiera seulement des algorithmes de la première sorte.

Les algorithmes de recherche d'un mot fixe dans un texte utilisent un mécanisme dit « de fenêtre glissante ». Une fenêtre de la même longueur que le mot x est utilisée pour accéder au texte y . L'extrémité gauche de cette fenêtre est d'abord alignée avec l'extrémité gauche du texte. Un travail spécifique appelé **tentative** permet de déterminer si le contenu de la fenêtre est identique à celui du mot. Ensuite l'algorithme **décale** la fenêtre vers la droite et le même processus est appliqué jusqu'à ce que l'extrémité droite de la fenêtre dépasse l'extrémité droite du texte. On associe à chaque tentative la position droite de la fenêtre : lors d'une tentative j la fenêtre est positionnée sur le facteur $y[j - m + 1 .. j]$ du texte. Un décalage de longueur d , appliqué après une tentative j , est dit **valide** s'il n'y a pas d'occurrence du mot x aux positions droites comprises entre $j + 1$ et $j + d - 1$. Les algorithmes de recherche de mot diffèrent par les méthodes employées lors des tentatives et des décalages. Un « bon » algorithme de recherche de mot cherche à minimiser le travail effectué lors de chaque tentative et à maximiser la longueur des décalages. L'algorithme naïf localise toutes les occurrences du mot dans le texte en temps $O(m \times n)$. Il est caractérisé par des décalages de longueur exactement égale à 1. Le hachage fournit une méthode simple qui évite le nombre quadratique de comparaisons de lettres dans la plupart des situations pratiques, et qui fonctionne en temps linéaire avec des hypothèses probabilistes raisonnables. Il a été présenté par Harrison [33] et analysé entièrement plus tard par Karp et Rabin [36]. Il existe beaucoup d'autres méthodes linéaires. Nous allons maintenant les passer en revue.

3.1 Les différentes méthodes

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

- Les algorithmes qui simulent le fonctionnement d'un automate déterministe. Ils ont une bonne complexité théorique. Ils examinent généralement les lettres de la fenêtre de la gauche vers la droite lors de chaque tentative.
- Les algorithmes rapides en pratique. Ils examinent généralement les lettres de la fenêtre de la droite vers la gauche lors de chaque tentative.
- Les algorithmes optimaux dans le compromis espace/temps. Ils par-

tionnent les lettres de la fenêtre en deux ensembles et examinent les lettres d'un ensemble dans un sens et les lettres de l'autre dans l'autre sens.

- Les algorithmes qui simulent le comportement d'un automate en représentant l'état de la recherche par un vecteur binaire. Ces algorithmes sont très efficaces pour la recherche de mots courts car alors les mots machines peuvent être utilisés pour représenter l'état de la recherche.

Une trentaine d'algorithmes de recherche exacte de mot sont décrits dans [CL96a].

Simulation d'un automate déterministe

La recherche avec un automate fini déterministe effectue exactement n inspections de lettres du texte mais elle exige un espace supplémentaire en $O(m \times \text{card } A)$ pour mémoriser l'automate.

Le premier algorithme de recherche de mot linéaire, en temps et en espace a été proposé par Morris et Pratt [41]. Il a été amélioré par Knuth, Morris, et Pratt [37]. La recherche se comporte comme un processus d'identification par automate, et une lettre du texte est comparé à une lettre du mot au plus $\log_\phi(m+1)$ fois (ϕ est le nombre d'or $(1 + \sqrt{5})/2$). Hancart [32] a montré que ce délai se réduit à $1 + \log_2 m$ comparaisons par lettre des textes pour un algorithme dû à Simon [47]. Ces algorithmes effectuent au plus $2n - 1$ comparaisons de lettres du texte.

Les algorithmes de Colussi [18] et de Galil-Giancarlo [27] partitionnent l'ensemble des positions du mot en deux sous-ensembles. Ils recherchent d'abord les lettres du mot dont les positions sont dans le premier sous-ensemble de gauche à droite et si aucune inégalité intervient ils recherchent les lettres restantes de gauche à droite. L'algorithme de Colussi effectue au plus $\frac{3}{2}n$ comparaisons de lettres du texte alors que l'algorithme de Galil-Giancarlo en effectue au plus $\frac{4}{3}n$.

L'algorithme d'Apostolico-Crochemore [6] a une complexité en temps en $O(n)$ dans le pire des cas, il effectue au plus $\frac{3}{2}n$ comparaisons de lettres du texte. L'algorithme Pas Si Naïf [31] est un algorithme très simple avec une complexité quadratique dans le pire des cas mais il requiert une phase de prétraitement en temps et espace constants et est légèrement sous-linéaire en moyenne.

Algorithmes rapides en pratique

L'algorithme de Boyer-Moore [14] est considéré comme l'algorithme de recherche de mot le plus efficace dans des applications usuelles. Une version simplifiée (ou l'algorithme entier) est souvent mise en œuvre dans les éditeurs de texte pour les commandes « rechercher » et « remplacer ». Cole [16] a montré que le nombre maximum des comparaisons de lettres

est borné par $3n$, après le prétraitement, pour les mots non-périodiques. L'algorithme de Boyer-Moore a cependant une complexité quadratique dans le pire des cas pour la recherche de mots périodiques. Plusieurs variantes de l'algorithme de Boyer-Moore évitent ce comportement quadratique. Les solutions les plus efficaces pour le nombre de comparaisons de lettres ont été conçues par Apostolico et Giancarlo [9], Crochemore *et alii* (Turbo-BM) [CCG⁺94] et Colussi (Reverse Colussi) [19].

L'algorithme de Horspool [34] est une variante de l'algorithme de Boyer-Moore, il n'utilise qu'une seule de ses deux fonctions de décalage et l'ordre dans laquelle les comparaisons de lettres de la fenêtre sont exécutées n'est pas significatif. Ceci est également vrai pour d'autres variantes telles que l'algorithme Quick Search de Sunday [51], l'algorithme de Smith [48] et celui de Raita [43].

Les résultats empiriques prouvent que la variation de l'algorithme de Boyer-Moore conçue par Sunday (Quick Search) [51], par Hume et Sunday (Tuned Boyer-Moore) [35] et des algorithmes basés sur l'automate des suffixes par Crochemore *et alii* (Reverse Factor et Turbo Reverse Factor) [CCG⁺94] sont les plus efficaces dans la pratique. Il est également possible d'utiliser un oracle des suffixes [3] à la place d'un automate des suffixes. L'utilisation d'un arbre des positions est efficace dans certaines conditions ([CL00] et [Lec00a]). Les algorithmes de Zhu-Takaoka [57] et Berry-Ravindran [12] sont des variantes de l'algorithme de Boyer-Moore qui exige un espace supplémentaire en $O((\text{card } A)^2)$.

Les algorithmes Optimal Mismatch et Maximal Shift de Sunday [51] trient les positions sur le mot selon leur fréquence et la longueur du décalage auxquels ils mènent respectivement.

Algorithmes optimaux en espace

Les deux premiers algorithmes linéaires de recherche de mots optimaux en espace sont dus à Galil et Seiferas [28] et Crochemore et Perrin [23]. Ils découpent le mot en deux parties, ils recherchent d'abord la partie droite du mot de gauche à droite et si aucune inégalité n'intervient ils recherchent la partie gauche.

Utilisation de vecteurs binaires

En supposant que la longueur du mot est plus petite que la taille d'un mot mémoire de la machine, l'algorithme Shift-Or ([11] et [55]) est un algorithme efficace pour résoudre le problème de recherche de mot et il s'adapte facilement à un éventail de problèmes de recherche approchée.

3.2 Complexité du problème

Nous allons maintenant présenter quelques résultats théoriques précisant les complexités de la recherche de mots.

Théorème 3.1 ([28])

La recherche peut être faite en temps $O(n)$ et en espace $O(1)$.

Théorème 3.2 ([56])

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

Théorème 3.3 ([17])

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

3.3 Algorithme de Boyer-Moore

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

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

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

```

BOYER-MOORE( $x, m, y, n$ )
1   $j \leftarrow m - 1$ 
2  tantque  $j < n$  faire
3       $i \leftarrow m - 1$ 
4      tantque  $i \geq 0$  et  $x[i] = y[i + j - m + 1]$  faire
5           $i \leftarrow i - 1$ 
6      si  $i < 0$  alors
7          SIGNALER( $j - m + 1$ )
8           $j \leftarrow j + \text{pér}(x)$ 
9      sinon  $j \leftarrow j + \text{bon-suff}[i]$ 

```

Figure 3.1 L'algorithme de Boyer-Moore.

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

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

Ces trois fonctions de décalages peuvent être calculées en temps et espace $O(m)$. Toutefois la quasi-totalité des présentations de l'algorithme de Boyer-Moore utilise le décalage de bon suffixe. C'est également ce décalage que nous retenons ici. Ses valeurs sont stockées dans une table *bon-suff*. Son calcul est présenté section 3.8. On peut remarquer que $\text{pér}(x) = \text{bon-suff}[0]$. L'algorithme de Boyer-Moore est montré figure 3.1.

Nous allons maintenant en présenter les complexités.

Théorème 3.4 ([14])

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

Théorème 3.5 ([16])

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

```

GALIL( $x, m, y, n$ )
1   $\ell \leftarrow 0$ 
2   $j \leftarrow m - 1$ 
3  tantque  $j < n$  faire
4       $i \leftarrow m - 1$ 
5      tantque  $i \geq \ell$  et  $x[i] = y[i + j - m + 1]$  faire
6           $i \leftarrow i - 1$ 
7      si  $i < \ell$  alors
8          SIGNALER( $j - m + 1$ )
9           $j \leftarrow j + \text{pér}(x)$ 
10          $\ell \leftarrow m - \text{pér}(x)$ 
11     sinon  $j \leftarrow j + \text{bon-suff}[i]$ 
12          $\ell \leftarrow 0$ 

```

Figure 3.2 L'algorithme de Galil.

Lors de la recherche de mots périodiques l'algorithme de Boyer-Moore a un comportement quadratique (exemple recherche de a^m dans a^n). Cela vient du fait que d'une tentative à l'autre l'algorithme oublie toutes les informations précédemment recueillies.

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

3.4 Algorithme de Galil

Pour les mots périodiques il suffit de modifier légèrement l'algorithme de Boyer-Moore pour obtenir un algorithme linéaire. En effet l'indice i , de l'algorithme de la figure 3.1, 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 $m - \text{pér}(x)$. L'algorithme de Galil figure 3.2 implante cette technique, appelée « mémorisation de préfixe ».

Le théorème suivant en donne les complexités.

Théorème 3.6 ([26])

L'algorithme de Galil localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(n)$. Il nécessite un espace supplémentaire $O(m)$. Sa phase de prétraitement peut être effectuée en temps et espace $O(m)$.

```

SMYTH( $x, m, y, n$ )
1   $\ell \leftarrow 0$ 
2   $j \leftarrow m - 1$ 
3  tantque  $j < n$  faire
4       $i \leftarrow m - 1$ 
5      tantque  $i \geq \ell$  et  $x[i] = y[i + j - m + 1]$  faire
6           $i \leftarrow i - 1$ 
7      si  $i < \ell$  alors
8          SIGNALER( $j - m + 1$ )
9           $j \leftarrow j + \text{pér}(x)$ 
10          $\ell \leftarrow m - \text{pér}(x)$ 
11     sinon  $j \leftarrow j + \text{bon-suff}[i]$ 
12         si  $i < \text{bon-suff}[i]$  alors
13              $\ell \leftarrow m - \text{bon-suff}[i]$ 
14     sinon  $\ell \leftarrow 0$ 

```

Figure 3.3 L'algorithme de Smyth.

3.5 Algorithme de Smyth

En fait la technique de mémorisation de préfixe peut s'appliquer à chaque fois qu'un décalage amène un préfixe du mot à être aligné avec le suffixe du mot reconnu lors de la tentative précédente et pas seulement lors de la découverte d'une occurrence de x . L'algorithme de Smyth, figure 3.3, applique cette méthode.

Le théorème qui suit en donne les complexités.

Théorème 3.7 ([49])

L'algorithme de Smyth localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(n)$. Il effectue au plus $4n$ comparaisons de lettres du texte. Il nécessite un espace supplémentaire $O(m)$. Sa phase de prétraitement peut être effectuée en temps et espace $O(m)$.

3.6 Algorithme Turbo-BM

Il est possible d'étendre la technique de mémorisation de préfixe en mémorisant un facteur de la fenêtre. Ainsi lors d'une tentative l'algorithme Turbo-BM mémorise le plus long facteur du texte qui correspondait avec un suffixe du mot lors de la tentative précédente. Cela présente deux avantages :

- pouvoir éventuellement effectuer un saut par dessus ce facteur lors de la tentative courante ;

```

TURBO-BM( $x, m, y, n$ )
1   $déc \leftarrow 0$ 
2   $mém \leftarrow 0$ 
3   $j \leftarrow m - 1$ 
4  tantque  $j < n$  faire
5       $i \leftarrow m - 1$ 
6      tantque  $i \geq 0$  et  $x[i] = y[i + j - m + 1]$  faire
7          si  $i = m - déc$  alors
8               $i \leftarrow i - mém - 1$        $\triangleright$  Saut
9          sinon  $i \leftarrow i - 1$ 
10     si  $i < 0$  alors
11         SIGNALER( $j - m + 1$ )
12          $déc \leftarrow pér(x)$ 
13          $mém \leftarrow m - déc$ 
14     sinon  $turbo \leftarrow mém - m + 1 + i$ 
15         si  $turbo \leq bon-suff[i]$  alors
16              $déc \leftarrow bon-suff[i]$ 
17              $mém \leftarrow \min\{m - déc, m - i\}$ 
18         sinon  $déc \leftarrow \max\{turbo, m - 1 - i\}$ 
19              $mém \leftarrow 0$ 
20      $j \leftarrow j + déc$        $\triangleright$  Décalage

```

Figure 3.4 L'algorithme Turbo-BM.

- pouvoir éventuellement effectuer un **turbo-décalage** après la tentative courante.

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

L'algorithme Turbo-BM est présenté figure 3.4.

À la suite nous en donnons les complexités.

Théorème 3.8 ([CCG⁺94])

L'algorithme Turbo-BM localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(n)$. Il effectue au

plus $2n$ comparaisons entre lettres du mot et lettres du texte. Il nécessite un espace supplémentaire $O(m)$. Sa phase de prétraitement peut être effectuée en temps et espace $O(m)$.

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

3.7 Algorithme d'Apostolico-Giancarlo

La technique de mémorisation de facteur mise en œuvre par l'algorithme Turbo-BM peut se généraliser. L'algorithme d'Apostolico-Giancarlo est une variante de l'algorithme de Boyer-Moore qui mémorise, lors de la phase de recherche, pour chaque position droite de la fenêtre la longueur du plus long suffixe du mot qui se termine à cette position. Après chaque tentative à une position j' dans le texte y , la longueur du plus long suffixe de x reconnu à la position droite j' , $|u|$, est mémorisée dans une table S ($S[j'] = |u|$). Ainsi, lors de la tentative courante à la position j sur le texte y on est amené à examiner une position j' , $j' < j$, (on a $y[j' + 1 \dots j] \preceq_{suff} x$), pour laquelle la valeur $k = S[j']$ est définie, on sait que $y[j' - k + 1 \dots j']$ est un suffixe de x . Soit $i = m - 1 - j + j'$. Il suffit alors de connaître la longueur $s = suff[i]$, du plus long suffixe de x se terminant à la position i sur x , pour conclure la tentative dans la majorité des situations.

Quatre cas peuvent se produire :

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

Seul un cas sur quatre requiert des comparaisons de lettres supplémentaires.

La table *suff* est calculée en temps et espace $O(m)$ pendant le calcul de la fonction de décalage (voir section 3.8). Seules les valeurs de la table S pour les lettres du texte contenues dans la fenêtre sont nécessaires à chaque tentative. L'ensemble de ces valeurs peuvent donc être mémorisées en espace $O(m)$.

L'algorithme de la figure 3.5 met en œuvre cette stratégie.

```

APOSTOLICO-GIANCARLO( $x, m, y, n$ )
1  pour  $j \leftarrow 0$  à  $n - 1$  faire
2       $S[j] \leftarrow 0$ 
3   $j \leftarrow m - 1$ 
4  tantque  $j < n$  faire
5       $i \leftarrow m - 1$ 
6      tantque  $i \geq 0$  faire
7          si  $S[i + j - m + 1] > 0$  alors
8               $k \leftarrow S[i + j - m + 1]$ 
9               $s \leftarrow \text{suff}[i]$ 
10             si  $s \neq k$  alors
11                  $i \leftarrow i - \min\{s, k\}$ 
12             rupture
13             sinon  $i \leftarrow i - k$       ▷ Saut
14             sinon si  $x[i] = y[i + j - m + 1]$  alors
15                  $i \leftarrow i - 1$ 
16             sinon rupture
17     si  $i < 0$  alors
18         SIGNALER( $j - m + 1$ )
19          $j \leftarrow j + \text{pér}(x)$ 
20          $S[j] \leftarrow m$ 
21     sinon  $j \leftarrow j + \text{bon-suff}[i]$ 
22          $S[j] \leftarrow m - 1 - i$ 

```

Figure 3.5 L'algorithme d'Apostolico-Giancarlo.

Le théorème suivant en énonce les complexités.

Théorème 3.9 ([9] et [CL97])

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

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

3.8 Calcul de la fonction de décalage

Nous considérons dans cette section le prétraitement que doit subir le mot x pour effectuer sa localisation au moyen des algorithmes de recherche présentés plus haut. Ce prétraitement consiste en le calcul de la

	i		0	1	2	3	4	5	6	7	8
(a)	$x[i]$		a	a	a	c	a	b	a	b	a
	$suff[i]$		1	1	1	0	1	0	3	0	9
	$bon-suff[i]$		8	8	8	8	8	2	8	4	1

(b)	x	a	a	a	c	a	b	a	b	a
	x	a	a	a	c	a	b	a	b	a

Figure 3.6 On considère le mot $x = \text{aaacababa}$. (a) Les valeurs des tables $suff$ et $bon-suff$. (b) On a $suff[6] = 3$. Ce qui 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 $bon-suff[9 - 1 - 3] = 9 - 1 - 6 = 2$, valeur qui est calculée à la ligne 11 de l'algorithme BON-SUFFIXE.

```

SUFFIXES( $x, m$ )
1   $g \leftarrow m - 1$ 
2   $suff[m - 1] \leftarrow m$ 
3  pour  $i \leftarrow m - 2$  à 0 pas -1 faire
4      si  $i > g$  et  $suff[i + m - 1 - f] < i - g$  alors
5           $suff[i] \leftarrow suff[i + m - 1 - f]$ 
6      sinon  $g \leftarrow \min\{g, i\}$ 
7           $f \leftarrow i$ 
8          tantque  $g \geq 0$  et  $x[g] = x[g + m - 1 - f]$  faire
9               $g \leftarrow g - 1$ 
10          $suff[i] \leftarrow f - g$ 
11 retourner  $suff$ 

```

Figure 3.7 L'algorithme qui calcule les longueurs des suffixes de x se terminant à chaque position sur x .

table du bon suffixe, $bon-suff$, et de la période de x . Ce dernier calcul est contenu dans le premier car on peut remarquer que $pér(x) = bon-suff[0]$.

Pour calculer la table $bon-suff$ on utilise la table $suff$ définie sur le mot x comme suit. Pour $i = 0, 1, \dots, m - 1$,

$suff[i] =$ longueur du plus suffixe commun à x et $x[0..i]$,

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 figure 3.6 donne les deux tables $suff$ et $bon-suff$ pour le mot $x = \text{aaacababa}$.

Le calcul de la table $suff$ est effectué par l'algorithme SUFFIXES de la figure 3.7.

Nous pouvons maintenant formuler l'algorithme BON-SUFFIXE qui calcule la table $bon-suff$ au moyen de la table $suff$ (voir figure 3.8).

La proposition qui suit caractérise cet algorithme.

```

BON-SUFFIXE( $x, m$ )
1   $j \leftarrow 0$ 
2  pour  $i \leftarrow m - 2$  à 0 pas -1 faire
3      si  $\text{suff}[i] = i + 1$  alors
4          tantque  $j < m - 1 - i$  faire
5               $\text{bon-suff}[j] \leftarrow m - 1 - i$ 
6               $j \leftarrow j + 1$ 
7  tantque  $j < m$  faire
8       $\text{bon-suff}[j] \leftarrow m$ 
9       $j \leftarrow j + 1$ 
10 pour  $i \leftarrow 0$  à  $m - 2$  faire
11      $\text{bon-suff}[m - 1 - \text{suff}[i]] \leftarrow m - 1 - i$ 
12 retourner  $\text{bon-suff}$ 

```

Figure 3.8 Calcul des décalages de bon suffixe.

Proposition 3.10 ([CHL00b])

L'algorithme BON-SUFFIXE calcule la table bon-suff du mot x au moyen de la table suff du même mot. Appliqué à un mot de longueur m , il s'exécute en temps $O(m)$ (même en incluant le temps de calcul de la table intermédiaire suff) et nécessite un espace supplémentaire $O(m)$.

3.9 L'algorithme Reverse Factor

Dans les sections qui suivent nous montrons qu'en utilisant l'automate des suffixes du renversé du mot x il est possible d'obtenir des algorithmes optimaux en moyenne. Nous présentons d'abord un algorithme quadratique dans le pire des cas, puis nous en donnons une version linéaire dans le pire des cas.

Les algorithmes du types Boyer Moore reconnaissent des suffixes du mot et calculent leurs décalages en fonction de ces suffixes. Il paraît évident que si on peut de la même manière (en comparant les lettres de droite à gauche) reconnaître des préfixes du mot, on sera alors capable d'effectuer de meilleurs décalages (c'est-à-dire des décalages plus longs). Ceci est possible grâce à l'utilisation de l'automate minimal des suffixes du renversé du mot.

L'automate minimal reconnaissant tous les suffixes d'un mot w est un Automate Fini Déterministe $\mathcal{S}(w) = (Q, q_0, T, \delta)$ où

- Q est un ensemble d'états,
- $q_0 \in Q$ est l'état initial de l'automate,
- $T \subseteq Q$ est l'ensemble des états terminaux de l'automate,
- $\delta : Q \times A \rightarrow Q$ est la fonction de transition de l'automate.

Le langage accepté par $\mathcal{S}(w)$ est : $\mathcal{L}(\mathcal{S}(w)) = \{v \in A^* / \exists u \in A^* \text{ et } uv = w\}$.

Le nombre d'états de $\mathcal{S}(w)$ est borné par $2|w|$ et le nombre de flèches par $3|w|$. La construction de cet automate est linéaire en temps et en espace en la longueur du mot w (voir [13] et [20]).

On pose comme hypothèse dans ce mémoire qu'une transition de l'automate peut être calculée en temps constant ce qui est une hypothèse raisonnable lorsque l'alphabet est fini.

Nous allons maintenant montrer comment utiliser cet automate des suffixes pour construire une première version d'un algorithme de recherche de mot appelé Reverse Factor. Cette version est quadratique dans le pire des cas.

Une des difficultés à comprendre le fonctionnement de l'algorithme utilisant l'automate minimal des suffixes du renversé du mot est qu'il ne cherche pas à faire correspondre une à une les lettres du mot avec celles de la fenêtre. En fait on progresse de la droite vers la gauche dans la fenêtre tant que la partie examinée est un facteur du mot et l'outil permettant de réaliser ceci est l'automate minimal des suffixes du renversé du mot préalablement construit.

Durant une tentative de cet algorithme on commence par examiner la lettre la plus à droite de la fenêtre et on se place dans l'état initial de l'automate. S'il y a une transition depuis l'état initial de l'automate définie pour la lettre la plus à droite de la fenêtre alors on se déplace d'une position vers la gauche dans la fenêtre et la lettre pointée devient ainsi la nouvelle lettre courante et l'état atteint depuis l'état initial par la lettre la plus à droite de la fenêtre devient le nouvel état courant. On progresse ainsi vers la gauche dans la fenêtre tant qu'il existe une transition définie pour la lettre courante depuis l'état courant. Durant tout ce cheminement dans l'automate on retient deux informations :

- (i) la longueur du chemin entre l'état initial et le dernier état terminal rencontré ;
- (ii) la longueur du chemin entre l'état initial et l'avant dernier état terminal rencontré.

À chaque fois qu'on rencontre un état terminal dans l'automate cela signifie qu'on a reconnu un suffixe du renversé du mot lu de la droite vers la gauche donc un préfixe du mot lu naturellement de la gauche vers la droite.

À la fin d'une tentative deux alternatives sont possibles :

- (i) la longueur du chemin entre l'état initial et le dernier état terminal rencontré est égale à la longueur du mot, cela signifie que l'on a localisé une occurrence du mot. Il faut alors décaler la fenêtre et le mot de la longueur de la période du mot, cela nous est fourni par la différence entre la longueur du chemin entre l'état initial et le dernier état terminal rencontré et la longueur du chemin entre l'état initial et l'avant dernier état terminal rencontré ;

```

REVERSE-FACTOR( $x, m, y, n$ )
1   $j \leftarrow m - 1$ 
2  tantque  $j < n$  faire
3       $i \leftarrow m - 1$ 
4       $\ell \leftarrow 0$ 
5       $q \leftarrow q_0$ 
6      tantque  $\delta(q, y[i + j - m + 1])$  est définie faire
7           $q \leftarrow \delta(q, y[i + j - m + 1])$ 
8           $i \leftarrow i - 1$ 
9          si  $i \in T$  alors
10              $\ell' \leftarrow \ell$ 
11              $\ell \leftarrow i$ 
12     si  $\ell = m$  alors
13         SIGNALER( $j - m + 1$ )
14          $j \leftarrow j + \ell'$ 
15     sinon  $j \leftarrow j + \ell$ 

```

Figure 3.9 L'algorithme Reverse Factor.

- (ii) la longueur du chemin entre l'état initial et le dernier état terminal rencontré est inférieure strictement à la longueur du mot, on n'a donc pas localisé d'occurrence du mot. Il faut décaler la fenêtre et le mot de manière à faire correspondre le plus long préfixe du mot reconnu : la longueur de ce préfixe correspond à la longueur du chemin entre l'état initial et le dernier état terminal rencontré.

Une fois le décalage effectué la tentative suivante exécute le même processus. L'algorithme Reverse Factor est montré figure 3.9.

L'algorithme Reverse Factor comme l'algorithme de Boyer-Moore a un comportement sous-linéaire en pratique et on peut même s'attendre à ce qu'il effectue de meilleurs décalages, toutefois sa complexité dans le pire des cas est quadratique.

Théorème 3.11 ([Lec92b] et [CCG⁺94])

L'algorithme Reverse Factor localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(m \times n)$. Il nécessite un espace supplémentaire $O(m)$. Sa phase de prétraitement peut être effectuée en temps et espace $O(m)$.

L'algorithme Reverse Factor atteint la borne théorique démontrée par Yao [56] (théorème 3.2). Il est donc optimal en ce qui concerne le nombre d'inspections de lettres du texte.

Théorème 3.12 ([CCG⁺94])

Le temps moyen de l'algorithme Reverse Factor est $O(\frac{\log m}{m} \times n)$.

La stratégie mise en œuvre par l'algorithme Reverse Factor permet de construire un automate de Boyer-Moore [37] dont le nombre d'états est borné par m^3 . En effet chaque état q de l'automate de Boyer-Moore ainsi construit est de la forme :

- un préfixe du mot (éventuellement vide) ;
- un facteur inconnu (éventuellement vide) ;
- un facteur du mot (éventuellement vide) ;
- un facteur inconnu (éventuellement vide).

$$q = x[1, i] \bullet \dots \bullet x[j, k] \bullet \dots \bullet, \text{ avec } i \leq j \leq k .$$

Il suffit donc de connaître trois positions i , j et k dans un mot de longueur m , il y a donc m^3 mots de cette forme.

3.10 L'algorithme Turbo Reverse Factor

De même que l'algorithme de Boyer-Moore oublie les lettres qu'il a reconnu lors des tentatives précédentes, l'algorithme Reverse Factor oublie le préfixe du mot qu'il a déjà reconnu. Pour remédier à cela il faut mémoriser la longueur du préfixe du mot reconnu à la tentative précédente et alors essayer lors de la tentative suivante de reconnaître la portion du texte qui se trouve à droite de ce préfixe dans la fenêtre. Une fois arrivé à ce point que nous appellerons **position de décision** il est alors possible de montrer qu'il suffit de relire au plus la moitié du préfixe déjà reconnu pour être capable d'effectuer un décalage valide. Supposons que nous soyons dans la situation générale où nous avons reconnu un préfixe u de longueur k du mot lors de la tentative précédente et la fenêtre est positionnée sur $y[j - m + 1 \dots j]$.

Appelons v le facteur $y[j - m + k + 1 \dots j]$. Alors comme précédemment la tentative courante examine le texte depuis la position j et progresse vers la gauche jusqu'à soit qu'il n'y ait pas de transition définie pour la lettre courante depuis l'état courant dans l'automate, soit qu'on ait atteint la position de décision $j - m + k$ avec succès. Si on n'a pas atteint la position $j - m + k$ alors le décalage est déterminé comme dans l'algorithme Reverse Factor, sinon v est un facteur du mot. Dans ce cas il n'est pas utile de relire toutes les lettres de u .

Si v est un facteur d'un mot w de longueur m on note $dépl(v, w)$ le plus petit entier d tel que $v = w[m - d - |v| + 1 \dots m - d]$.

Si v est un facteur du mot alors il suffit de relire au plus la moitié droite de u pour déterminer un décalage valide. Si u est périodique ($pér(u) \leq |u|/2$), soit z le suffixe de u de longueur $pér(u)$ alors la longueur du décalage à effectuer est égale à $dépl(zv, x)$. Si u n'est pas périodique ($pér(u) > |u|/2$) alors il est évident que le mot ne peut réapparaître dans la moitié gauche de u de longueur $pér(u)$ donc il suffit d'examiner la

```

TURBO-REVERSE-FACTOR( $x, m, y, n$ )
1   $j \leftarrow m - 1$ 
2   $d \leftarrow m$ 
3   $u \leftarrow \varepsilon$ 
4  tantque  $j < n$  faire
5       $i \leftarrow m - 1$ 
6       $\ell \leftarrow 0$ 
7       $q \leftarrow q_0$ 
8      tantque  $i > m - d$  et  $\delta(q, y[i + j - m + 1])$  est définie faire
9           $q \leftarrow \delta(q, y[i + j - m + 1])$ 
10          $i \leftarrow i - 1$ 
11         si  $i \in T$  alors
12              $\ell \leftarrow i$ 
13     si  $i \leq m - d$  alors
14         si  $\text{dépl}(y[i + j - m + 2 \dots j], x) = 0$  alors
15             SIGNALER( $j - m + 1$ )
16              $d \leftarrow \text{pér}(x)$ 
17         sinon  $h \leftarrow \min\{\text{pér}(u), \lfloor |u|/2 \rfloor\}$ 
18             tantque  $i > m - d - h$  et
19                  $\delta(q, y[i + j - m + 1])$  est définie faire
20                      $q \leftarrow \delta(q, y[i + j - m + 1])$ 
21                      $i \leftarrow i - 1$ 
22                     si  $i \in T$  alors
23                          $\ell \leftarrow i$ 
24             si  $i \leq m - d - h$  alors
25                  $d \leftarrow \text{dépl}(y[i + j - m + 2 \dots j], x)$ 
26             sinon  $d \leftarrow \ell$ 
27          $j \leftarrow j + d$ 
28      $u \leftarrow x[0 \dots m - d]$ 

```

Figure 3.10 L'algorithme Turbo Reverse Factor.

moitié droite de u de longueur $|u| - \text{pér}(u) < |u|/2$ pour qu'une transition ne soit pas définie pour la lettre courante du texte à partir de l'état courant de l'automate. La fonction *dépl* peut être très facilement calculée et intégrée à l'automate des suffixes du renversé du mot sans altérer sa complexité.

L'algorithme Turbo Reverse Factor est montré figure 3.10.

Pour connaître la valeur de $\text{pér}(u)$ il est possible de calculer à l'avance et de façon linéaire les périodes de tous les préfixes du mot à l'aide de la fonction de suppléance de Morris et Pratt [41], mais il est également possible de les calculer de façon dynamique pendant l'exécution de la phase de recherche.

Théorème 3.13 ([CCG⁺94])

L'algorithme Turbo Reverse Factor localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(n)$. Il effectue au plus $2n$ lectures de lettres du texte. Il nécessite un espace supplémentaire $O(m)$. Sa phase de prétraitement peut être effectuée en temps et espace $O(m)$.

3.11 Algorithme Pos Tree

Les algorithmes Reverse Factor et Turbo Reverse Factor examinent le contenu de la fenêtre de droite à gauche tant qu'il y a une transition définie pour l'état courant par la lettre courante, reconnaissant ainsi des facteurs du mot dans le texte. Dans la plupart des cas, un décalage peut être effectué sans examiner toutes les lettres du facteur car des positions du mot sont déterminées de manière univoque par exactement un seul facteur. Une solution consiste à utiliser un arbre des positions [53] du renversé du mot x^R . Dans la suite nous supposons que le mot est toujours terminé par une lettre spéciale $\mathbf{0}$ qui n'apparaît jamais ailleurs (ce qui est souvent le cas dans les langages de programmation).

Un arbre des positions d'un mot w de longueur m possède exactement $m+1$ feuilles identifiant les $m+1$ facteurs du mot commençant en chacune de ses positions. Donc chaque chemin de l'arbre depuis la racine jusqu'à une feuille identifie une position et chaque branche de l'arbre mène soit directement à une feuille soit à un sous-arbre qui contient une fourche. L'arbre des positions de x^R peut être interprété en termes d'Automate Fini Déterministe (AFD) $\mathcal{P} = (Q, q_0, T, \delta)$ où Q est l'ensemble des nœuds de l'arbre, q_0 est la racine de l'arbre, T est l'ensemble des feuilles et δ est la fonction de transition associée. Cette structure est augmentée par une fonction J de T vers \mathbf{N} qui associe une position sur x^R à chaque feuille. Pour tous les $q \in T$

$$J(q) = i \text{ tel que } \delta(q_0, x[i..i + \text{prof}(q) - 1]) = q ,$$

où $\text{prof}(q)$ est la profondeur du nœud q dans l'arbre. Un exemple est montré figure 3.11.

Nous pouvons maintenant décrire l'algorithme de recherche avec arbre de positions [CL00]. Lors de chaque tentative, l'algorithme examine une partie du texte de droite à gauche jusqu'à ce qu'il n'y ait plus de transition définie depuis le nœud courant par la lettre courante. Le processus commence toujours avec la racine de l'arbre comme nœud courant. Durant ce processus, la valeur par la fonction J du dernier nœud q , tel que $\delta(q, \mathbf{0})$ est définie, est mémorisée. Cette valeur est utilisée pour effectuer un décalage lorsque l'examen ne se termine pas sur une feuille, autrement la valeur par la fonction J de la feuille est utilisée. C'est seulement lorsqu'un suffixe (menant à un décalage de longueur nulle) est reconnu

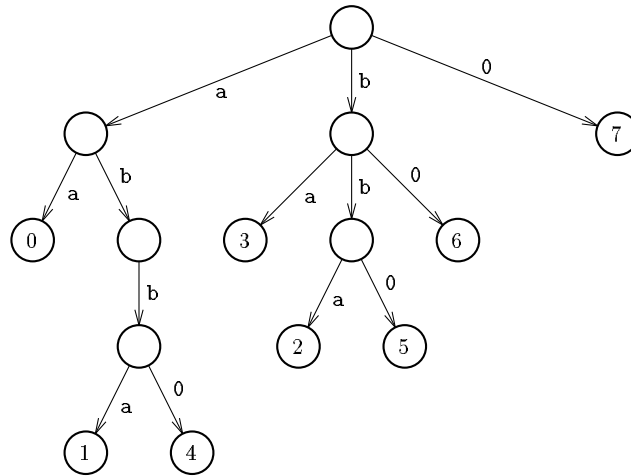


Figure 3.11 Arbre des positions pour $x^R = aabbabb$. Les nœuds contiennent les valeurs de la fonction J . Par exemple, si ab est le facteur le plus à droite de la fenêtre (ba lu de droite à gauche) lors de la recherche de $x = bbabbaa$, un décalage de longueur 3 doit être appliqué.

qu'un test d'une occurrence complète du mot est effectuée de manière naïve. En fait, la longueur du suffixe reconnu est facilement connue et une vérification naïve n'est effectuée que pour le préfixe correspondant. Pour éviter de tester à chaque tentative si la fenêtre a dépassé l'extrémité droite du texte, m occurrences de la lettre nulle 0 sont concaténées à la fin du texte et $J(\delta(q_0, 0))$ est affectée avec la valeur 0. C'est seulement lorsqu'une occurrence du mot est détectée que le test de débordement est effectué [44]. Lorsqu'une occurrence de x est localisée un décalage de longueur $déc = \min\{i : 0 \leq i \leq m - 1 \text{ et } x[m - 1 - i] = x[m - 1]\} \cup \{m\}$ est effectué. L'algorithme de recherche avec arbre de positions est écrit figure 3.12.

La complexité de la phase de prétraitement de cet algorithme est $O(m^2)$ et celle de la phase de recherche est $O(m \times n)$.

3.12 Algorithme Skip Search

L'idée de l'algorithme Skip Search est simple. Pour chaque lettre a de l'alphabet, le compartiment $z[a]$ contient toutes les positions de cette lettre dans x . Cette idée avait déjà été étudié dans [10] mais est utilisée ici d'une façon différente. Quand une lettre est présente k fois dans le mot, il y a les k positions correspondantes dans le compartiment de la lettre. Quand le mot est beaucoup plus court que l'alphabet, beau-

```

PosTREE( $x, m, y, n$ )
1   $j \leftarrow m - 1$ 
2   $s \leftarrow 0$ 
3  tantque  $j < n$  faire
4      répéter
5           $j \leftarrow j + s$ 
6           $i \leftarrow j$ 
7           $s \leftarrow m$ 
8           $q \leftarrow q_0$ 
9          tantque  $\delta(q, y[i])$  est définie faire
10              $q \leftarrow \delta(q, y[i])$ 
11             si  $\delta(q, 0)$  est définie alors
12                  $s \leftarrow J(\delta(q, 0))$ 
13              $i \leftarrow i - 1$ 
14             si  $q \in T$  alors
15                  $s \leftarrow J(q)$ 
16         jusque  $s = 0$ 
17     si  $x = y[j - m + 1 .. j]$  et si  $j < n$  alors
18         SIGNALER( $j - m + 1$ )
19      $s \leftarrow déc$ 

```

Figure 3.12 L'algorithme PosTree.

coup de compartiments sont vides. La boucle principale de la phase de recherche consiste à examiner le texte en effectuant des décalages de longueur exactement m (il y aura donc $\lfloor n/m \rfloor$ tentatives). Lors de l'examen d'une lettre $y[j]$ on utilise les positions du compartiment $z[y[j]]$ pour déterminer les positions de départ possibles d'occurrences de x dans y . Les comparaisons entre lettres de x et lettres de y peuvent alors être effectuées naïvement. L'algorithme Skip Search est montré figure 3.13.

Le théorème suivant en donne les complexités.

Théorème 3.14 ([CLP98])

L'algorithme Skip Search localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(m \times n)$. Il nécessite un espace supplémentaire $O(m + \text{card } A)$. Sa phase de prétraitement peut être effectuée en temps et espace $O(m + \text{card } A)$.

3.13 Algorithme Alpha Skip Search

Au lieu d'avoir un compartiment pour chaque lettre de l'alphabet il est possible d'avoir un compartiment pour chaque mot de longueur $\ell = \log_{\text{card } A} m$. Pour cela nous construisons un arbre $T(x)$ de tous les facteurs de longueur ℓ apparaissant dans le mot x . Les feuilles de $T(x)$ représente tous les facteurs de longueur ℓ de x . Il y a alors un compartiment pour chaque feuille de $T(x)$ dans lequel est stockée la liste des

```

SKIP-SEARCH( $x, m, y, n$ )
1  ▷ Initialisation
2  pour pour toutes les lettres  $a \in A$  faire
3       $z[a] \leftarrow \emptyset$ 
4  ▷ Phase de prétraitement
5  pour  $i \leftarrow 0$  à  $m - 1$  faire
6       $z[x[i]] \leftarrow z[x[i]] \cup \{i\}$ 
7  ▷ Phase de recherche
8   $j \leftarrow m - 1$ 
9  tantque  $j < n$  faire
10     pour tous les  $i \in z[y[j]]$  faire
11         si  $x = y[j - i .. j - i + m - 1]$  alors
12             SIGNALER( $j - i$ )
13      $j \leftarrow j + m$ 

```

Figure 3.13 L'algorithme Skip Search.

positions où le facteur, associé à la feuille, apparaît dans x . La phase de recherche consiste alors à examiner les facteurs du texte $y[j .. j + \ell - 1]$ pour tous les $j = k(m - \ell + 1) - 1$ avec k nombre entier dans l'intervalle $[1, \lfloor (n - \ell)/m \rfloor]$. L'algorithme complet est montré figure 3.14. Il utilise une fonction AJOUR donnée figure 3.15.

Les deux théorèmes suivants en donne les complexités.

Théorème 3.15 ([CLP98])

L'algorithme Alpha Skip Search localise toutes les occurrences d'un mot de longueur m dans un texte de longueur n en temps $O(m \times n)$. Il nécessite un espace supplémentaire $O(m + \text{card } A)$. Sa phase de prétraitement peut être effectuée en temps et espace $O(m + \text{card } A)$.

Théorème 3.16 ([CLP98])

En moyenne l'algorithme Alpha Skip Search effectue moins de $O(\ell(1 + \frac{m-\ell+1}{m})(\frac{n}{m-\ell}))$ inspections de lettres du texte lors de la localisation de toutes les occurrences d'un mot de longueur m dans un texte de longueur n .

3.14 Recherche d'un ensemble fini de mots

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 ensemble fini de k mots $X = \{x_0, x_1, \dots, x_{k-1}\}$ 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 .

```

ALPHA-SKIP-SEARCH( $x, m, y, n$ )
1  ▷ Initialisation
2   $racine \leftarrow$  NOUVEAU-NOEUD()
3   $suffixe[racine] \leftarrow \emptyset$ 
4   $hauteur[racine] \leftarrow 0$ 
5  ▷ Phase de prétraitement
6   $noeud \leftarrow racine$ 
7  pour  $i \leftarrow 0$  à  $m - 1$  faire
8      si  $hauteur[noeud] = \ell$  alors
9           $noeud \leftarrow suffixe[noeud]$ 
10      $noeudSuccesseur \leftarrow successeur[noeud, x[i]]$ 
11     si  $noeudSuccesseur = \emptyset$  alors
12          $noeudSuccesseur \leftarrow$  AJOUT( $noeud, x[i]$ )
13     si  $hauteur[noeudSuccesseur] = \ell$  alors
14          $z[noeud] \leftarrow z[noeud] \cup \{i - \ell - 1\}$ 
15      $noeud \leftarrow noeudSuccesseur$ 
16  ▷ Phase de recherche
17   $j \leftarrow m - \ell$ 
18  tantque  $j < n - \ell$  faire
19      $noeud \leftarrow racine$ 
20     pour  $k \leftarrow 0$  à  $\ell - 1$  faire
21          $noeud \leftarrow successeur[noeud, y[j + k]]$ 
22     si  $noeud \neq \emptyset$  alors
23         pour tous les  $i \in z[noeud]$  faire
24             si  $y[j - i .. j - i + m - 1] = x$  alors
25                 SIGNALER( $j - i$ )
26      $j \leftarrow j + m - \ell + 1$ 

```

Figure 3.14 L'algorithme Alpha Skip Search.

```

AJOUT( $noeud, a$ )
1   $noeudSuccesseur \leftarrow$  NOUVEAU-NOEUD()
2   $successeur[noeud, a] \leftarrow noeudSuccesseur$ 
3   $hauteur[noeudSuccesseur] \leftarrow hauteur[noeud] + 1$ 
4   $noeudSuffixe \leftarrow suffixe[noeud]$ 
5  si  $noeudSuffixe = \emptyset$  alors
6       $suffixe[noeudSuccesseur] \leftarrow noeud$ 
7  sinon  $successeurNoeudSuffixe \leftarrow successeur[noeudSuffixe, a]$ 
8      si  $successeurNoeudSuffixe = \emptyset$  alors
9           $successeurNoeudSuffixe \leftarrow$  AJOUT( $noeudSuffixe, a$ )
10      $suffixe[noeudSuccesseur] \leftarrow successeurNoeudSuffixe$ 
11  retourner  $noeudSuccesseur$ 

```

Figure 3.15 La fonction AJOUT utilisée par l'algorithme Alpha Skip Search.

Algorithme RF Multiple

L'examen du texte pendant une tentative consiste à déterminer le plus long facteur des mots de X qui est un suffixe du contenu de la fenêtre. Pour mettre en œuvre cette méthode, on utilise un automate des suffixes des renversés des mots de X . Pendant l'examen du texte, l'automate contient les informations suffisantes pour produire les positions des occurrences des mots de X .

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^R l'ensemble des renversés des mots de X . On considère l'automate $\mathcal{S}(X^R)$ des suffixes des renversés des mots de X , noté (R, r_0, U, ζ) . Rappelons que cet automate accepte le langage

$$\text{Suff}(X^R) = \{v \in A^* : uv = xR, u \in A^*, x \in X\} .$$

En d'autres termes, $\mathcal{S}(X^R)$ reconnaît de manière déterministe les préfixes des mots de X par lecture de droite à gauche. Notons que tout autre automate déterministe équivalent convient. Pour chaque état terminal $q \in U$ de l'automate atteint par un mot de X^R , on pose

$$\text{sortie}[q] = \{i : 0 \leq i \leq k - 1 \text{ et } \bar{\zeta}(r_0, (x_i)^R) = q\} ,$$

où $\bar{\zeta}$ est l'extension aux mots de la fonction de transition ζ de l'automate $\mathcal{S}(X^R)$.

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 $\mathcal{S}(X^R)$. À chaque fois qu'un état q est atteint par une lettre $y[j']$, on vérifie si $\text{sortie}[q]$ est non vide ; si tel est le cas, le mot x_i apparaît dans le texte y à la position j' quand

$$i \in \text{sortie}[q] \text{ et } j - j' + 1 = |x_i| .$$

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.

L'algorithme de la figure 3.16 implante cette méthode.

Théorème 3.17 ([Lec92a])

L'algorithme RF-MULTIPLE localise toutes les occurrences des mots d'un dictionnaire X dans un texte y . Son temps d'exécution est $O(k \times m' \times n)$.

```

RF-MULTIPLE( $X, m, y, n$ )
1   $j \leftarrow m - 1$ 
2  tantque  $j \leq n$  faire
3       $q \leftarrow r_0$ 
4       $j' \leftarrow j$ 
5       $d \leftarrow m$ 
6      tantque  $j' \geq 0$  et  $\zeta(q, y[j']) \neq \text{NIL}$  faire
7           $q \leftarrow \zeta(q, y[j'])$ 
8          si  $q \in U$  alors
9              si  $\text{sortie}[q] \neq \emptyset$  alors
10                 pour chaque  $i \in \text{sortie}[q]$  faire
11                     si  $|x_i| = j - j' + 1$  alors
12                         SIGNALER( $j'$ )
13                 si  $m - j + j' - 1 > 0$  alors
14                      $d \leftarrow \min\{d, m - j + j' - 1\}$ 
15                 sinon  $d \leftarrow 1$ 
16              $j' \leftarrow j' - 1$ 
17      $j \leftarrow j + d$ 

```

Figure 3.16 L'algorithme RF Multiple.

Algorithme Dawg Match

En combinant les idées du dernier algorithme avec celle de l'algorithme de Aho-Corasick [2], il est possible d'obtenir un algorithme optimal en moyenne qui effectue moins de $2n$ inspections de lettres du texte dans le pire des cas.

On utilise l'automate $M = (Q, q_0, T, \delta)$ qui reconnaît le langage A^*X et un automate (déterministe) $N = \mathcal{S}(X^R) = (R, r_0, U, \zeta)$ qui reconnaît le langage $\text{Suff}(X^R)$. L'automate N agit comme un filtre sur le contenu de la fenêtre.

Rappelons que chaque élément $q \in Q$ est un préfixe d'un mot de X . On définit l'entier $\text{Déca}[q]$ comme suit :

$$\text{Déca}[q] = \min\{|w| : w \in A^+ \text{ et } x \preceq_{\text{suff}} qw \text{ avec } x \in X\}$$

On note que $\text{Déca}[q] \leq m$.

La phase de prétraitement de l'algorithme de localisation consiste, d'une part, à construire l'automate $M = (Q, q_0, T, \delta)$ avec sa fonction de suppléance f et la table Déca , et d'autre part, à construire l'automate des suffixes $N = \mathcal{S}(X^R) = (R, r_0, U, \zeta)$.

Durant la phase de recherche, lors d'une tentative à la position j , la fenêtre de longueur m est partagée en deux parties par une position critique pos-crit . Le préfixe de la fenêtre $u = y[j - m + 1 \dots \text{pos-crit} - 1]$ a déjà été examiné et on connaît l'état $q = \delta(q_0, u)$ de M . Le suffixe $v = y[\text{pos-crit} \dots j]$ de la fenêtre n'a pas encore été examiné.

La tentative à la position j se décompose alors en deux phases.

Première phase On détermine le plus long suffixe de v qui est préfixe d'un mot de X quand v n'est pas facteur d'un mot de X . On considère v lui-même dans le cas contraire. Ceci revient à calculer l'entier j' défini par :

$$j = \begin{cases} pos-crit & \text{si } v \in Fact(X) \text{ ,} \\ \min\{\ell : y[\ell..j] \in Préf(X)\} & \text{sinon ,} \end{cases}$$

où $Fact(X)$ est l'ensemble de tous les facteurs de tous les mots de X .

La seconde phase fait référence à l'état q' de l'automate M à partir duquel poursuivre la recherche avec cet automate :

$$q' = \begin{cases} q & \text{si } v \in Fact(X) \text{ et donc } j' = pos-crit \text{ ,} \\ q_0 & \text{sinon } (j' > pos-crit) \text{ .} \end{cases}$$

Deuxième phase On détermine l'état $\delta(q', y[j'..j])$ de M qui devient la nouvelle valeur de q .

Après la deuxième phase la nouvelle valeur de $pos-crit$ est $j + 1$ et on effectue un décalage de longueur $Déca[q]$ avant de procéder à une nouvelle tentative. La première tentative à lieu à la position $m - 1$ et $pos-crit$ a la valeur 0.

Le calcul de j' dans la première phase se fait en examinant les lettres du mot v de la droite vers la gauche. C'est pour cette raison que l'on utilise l'automate $N = \mathcal{S}(X^R)$.

Le calcul de la nouvelle valeur de l'état q lors de la deuxième phase est réalisé par simple parcours de l'automate M . Ce faisant, on réexamine de gauche à droite le facteur $y[j'..j]$ obtenu durant la première phase pour signaler toutes les occurrences des mots de X dans le texte y .

L'algorithme Dawg Match de la figure 3.17 implante la méthode décrite ci-dessus. Il fait appel à deux fonctions : DE-DROITE-À-GAUCHE (figure 3.18) qui réalise la première phase et DE-GAUCHE-À-DROITE (figure 3.19) qui réalise la seconde.

Il nous faut maintenant expliquer comment calculer la table $Déca$. En fait, $Déca[q]$ est la longueur du plus court chemin dans l'automate M sortant de l'état q et entrant dans un état terminal.

On définit pour tous les q préfixes d'un mot de X la table Ext comme suit :

$$Ext[q] = \min\{|v| : v \in A^* \text{ et } x \preceq_{suff} qv \text{ avec } x \in X\} \text{ .}$$

Il est possible de donner une définition récursive de Ext :

$$Ext[q] = \begin{cases} 0 & \text{si } q \text{ est terminal ,} \\ 1 + \min\{Ext[p] : p \text{ est un successeur de } q\} & \text{sinon .} \end{cases}$$

Il est alors possible de donner une définition formelle de $Déca[q]$:

$$Déca[q_0] = Ext[q_0] = m \text{ ,}$$

```

DAWG-MATCH( $X, y, n$ )
1   $pos-crit \leftarrow 0$ 
2   $j \leftarrow m - 1$ 
3   $q \leftarrow q_0$ 
4  tantque  $j < n$  faire
5       $j' \leftarrow \text{DE-DROITE-À-GAUCHE}(pos-crit, j)$ 
6      si  $j' < pos-crit$  alors
7           $j' \leftarrow pos-crit$ 
8      sinon  $q \leftarrow q_0$ 
9       $q \leftarrow \text{DE-GAUCHE-À-DROITE}(j', j, q)$ 
10      $pos-crit \leftarrow j + 1$  ▷ Mémorisation
11      $j \leftarrow j + \text{Déca}[q]$  ▷ Décalage

```

Figure 3.17 L'algorithme DAWG Match.

```

DE-DROITE-À-GAUCHE( $pos-crit, j$ )
1   $r \leftarrow r_0$ 
2   $j' \leftarrow j$ 
3   $\ell \leftarrow j + 1$ 
4  tantque  $j' \geq pos-crit$  et  $\zeta(r, y[j'])$  est définie faire
5       $r \leftarrow \zeta(r, y[j'])$ 
6      si  $\text{terminal}[r]$  alors
7           $\ell \leftarrow j'$ 
8       $j' \leftarrow j' - 1$ 
9  si  $j' < pos-crit$  alors
10     retourner  $j'$ 
11 sinon retourner  $\ell$ 

```

Figure 3.18 La fonction DE-DROITE-À-GAUCHE utilisée par l'algorithme DAWG-MATCH.

```

DE-GAUCHE-À-DROITE( $j', j, q$ )
1  pour  $i \leftarrow j'$  à  $j$  faire
2       $q \leftarrow \delta(q, y[i])$ 
3      si  $q \in T$  alors
4          SIGNALER( $i$ )
5  retourner  $q$ 

```

Figure 3.19 La fonction DE-GAUCHE-À-DROITE utilisée par l'algorithme DAWG-MATCH.

et pour tous les q préfixes d'un mot de X différents de q_0 :

$$Déca[q] = \begin{cases} Déca[f(q)] & \text{si } Ext[q] = 0 \text{ ,} \\ \min\{Ext[q], Déca[f(q)]\} & \text{sinon .} \end{cases}$$

La procédure CALCUL-EXT permet de calculer les valeurs de la table Ext , elle fait appel à la fonction récursive CALCUL-EXT-RÉC.

CALCUL-EXT(M)

1 $nul \leftarrow \text{CALCUL-EXT-RÉC}(q_0)$

CALCUL-EXT-RÉC(q)

```

1  si  $q \in T$  alors
2      pour chaque couple  $(a, p)$  tel que  $\delta(q, a) = p$  faire
3           $nul \leftarrow \text{CALCUL-EXT-RÉC}(p)$ 
4       $Ext[q] \leftarrow 0$ 
5  sinon  $minim \leftarrow m$ 
6      pour chaque couple  $(a, p)$  tel que  $\delta(q, a) = p$  faire
7           $minim \leftarrow \min\{minim, \text{CALCUL-EXT-RÉC}(p)\}$ 
8       $Ext[q] \leftarrow 1 + minim$ 
9  retourner  $Ext[q]$ 

```

La procédure CALCUL-DÉCAL permet de calculer les valeurs de la table $Déca$.

CALCUL-DÉCAL()

```

1   $Déca[q_0] \leftarrow m$ 
2   $\phi \leftarrow \text{FILE-VIDE}()$ 
3  pour chaque couple  $(a, q)$  tel que  $\delta(q_0, a) = p$  faire
4       $\text{ENFILER}(\phi, q)$ 
5  tantque non  $\text{FILE-EST-VIDE}(\phi)$  faire
6       $\phi \leftarrow \text{DÉFILEMENT}(q)$ 
7      si  $Ext[q] = 0$  alors
8           $Déca[q] \leftarrow Déca[f(q)]$ 
9      sinon  $Déca[q] \leftarrow \min\{Ext[q], Déca[f(q)]\}$ 
10     pour chaque couple  $(a, p)$  tel que  $\delta(q, a) = p$  faire
11          $\text{ENFILER}(\phi, p)$ 

```

Nous allons maintenant montrer que l'algorithme Dawg Match est bien linéaire dans le pire des cas.

Théorème 3.18 ([CCG⁺99])

L'algorithme Dawg Match localise toutes les occurrences des mots d'un ensemble fini $X \subseteq A^+$ dans un texte de longueur n en temps $O(n)$ et espace $O(|X|)$. Il effectue au plus n inspections de lettres du texte.

L'algorithme Dawg Match a un comportement optimal en moyenne.

Théorème 3.19 ([CCG⁺99])

En supposant que les lettres du texte sont équiprobables et indépendantes, le nombre de lettres du texte examinées par l'algorithme *Dawg Match* est $O(\frac{\log m}{m}n)$.

3.15 Résultats expérimentaux

Il y a deux mesures des performances des algorithmes de recherche exacte de mot. L'une est théorique et consiste à compter le nombre de comparaisons entre lettres du mot et lettres du texte de chaque algorithme. La seconde est pratique et consiste à chronométrer le temps d'exécution de chaque algorithme. L'étude publiée dans [Lec95] a permis de comparer sur ces deux critères les algorithmes suivants : naïf, Boyer-Moore, Turbo-BM, Apostolico-Giancarlo, Horspool, Quick Search, Reverse Factor et Turbo Reverse Factor. Les tests ont été menés sur sept types différents de textes. Quatre textes ont été construits aléatoirement sur quatre alphabets différents : binaire, composé de quatre lettres, huit lettres et vingt lettres. Un texte en langage naturel (anglais), un fichier source d'un programme en langage C et une séquence d'ADN ont également servi de support à cette étude.

En ce qui concerne le nombre de comparaisons de lettres, pour des mots longs l'algorithme Turbo Reverse Factor effectue toujours le moins de comparaisons. Pour des mots courts (de plus en plus courts lorsque la taille de l'alphabet diminue) c'est l'algorithme Quick Search qui effectue le moins de comparaisons. Entre les deux c'est l'algorithme Apostolico-Giancarlo qui est le meilleur en fonction de ce critère (voir figure 3.20).

Les résultats montrent également que pour de petits alphabets l'algorithme de Boyer-Moore effectue plus de comparaisons que l'algorithme Turbo-BM qui lui-même en effectue plus que l'algorithme d'Apostolico-Giancarlo. De même l'algorithme Reverse Factor effectue plus de comparaisons que l'algorithme Turbo Reverse Factor. Lorsque la taille de l'alphabet augmente les performances des algorithmes de Boyer-Moore, Turbo-BM et Apostolico-Giancarlo se confondent. Il en est de même pour les performances des algorithmes Reverse Factor et Turbo Reverse Factor. Ceci est dû au fait que ces algorithmes exploitent les régularités dans le texte et le nombre de ces régularités diminue avec l'augmentation de la taille de l'alphabet.

Ces résultats sont confirmés dans [40].

La mesure du temps d'exécution inclut le temps de la phase de prétraitement ainsi que le temps de la phase de recherche. Pour des mots longs l'algorithme Reverse Factor est toujours le plus rapide. Pour des mots très courts c'est l'algorithme Quick Search qui est le plus rapide. Entre les deux pour de petits alphabets il y a une petite zone où l'algorithme de Boyer-Moore est le plus rapide (voir figure 3.21).

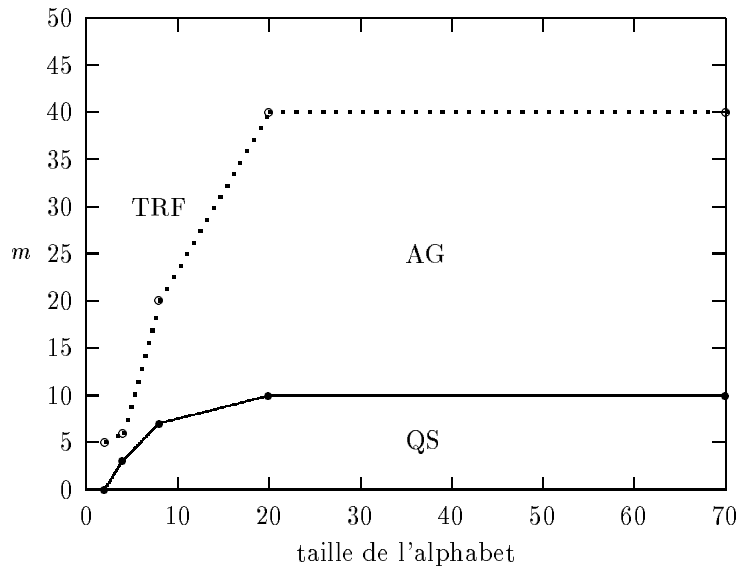


Figure 3.20 Zones de performances des algorithmes pour le nombre de comparaisons de lettres (*AG* = Apostolico-Giancarlo, *QS* = Quick Search, *TRF* = Turbo Reverse Factor).

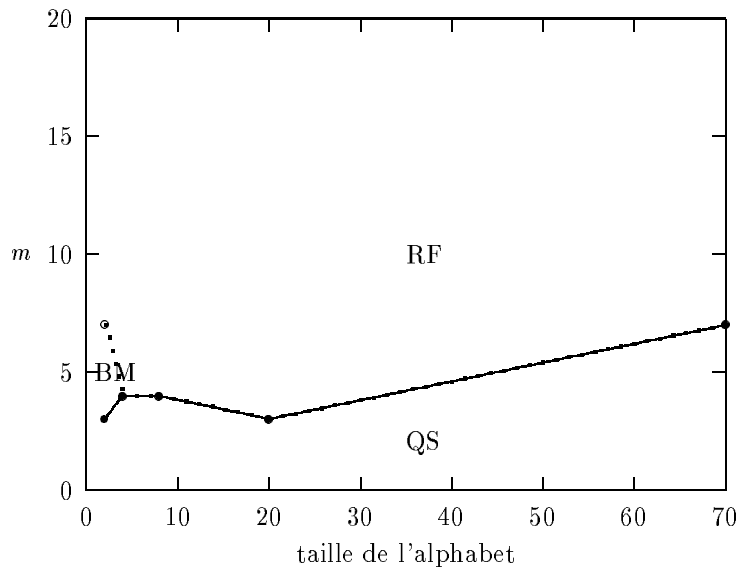


Figure 3.21 Zones de performances des algorithmes pour le temps d'exécution (*BM* = Boyer-Moore, *QS* = Quick Search, *RF* = Reverse Factor).

Les résultats montrent également que l'algorithme de Boyer-Moore est toujours plus rapide que les algorithmes Turbo-BM et d'Apostolico-Giancarlo. De même l'algorithme Reverse Factor est toujours plus rapide que l'algorithme Turbo Reverse Factor. Ceci est dû au fait que le travail nécessaire pour économiser des comparaisons se révèle trop consommateur de temps. En outre les performances de l'algorithme Reverse Factor diminuent pour des mots très longs, ceci étant dû au fait que le temps de construction de l'automate des suffixes, bien qu'étant linéaire, devient trop important au regard du temps de la phase de recherche.

Une étude reportée dans [CLP98] permet de conclure que l'algorithme Alpha Skip Search se révèle très efficace tant en termes de comparaisons de lettres qu'en termes de temps d'exécution pour de longs mots (longueur supérieure à 300) et de petits alphabets (de taille environ 4).

Une étude plus récente [Lec00a] montre qu'en termes de temps d'exécution pour la recherche de mots courts sur de grands alphabets, l'algorithme Tuned Boyer-Moore (voir chapitre 6) est le plus efficace, pour des mots très longs, l'algorithme Backward Oracle Matching est le meilleur, entre les deux l'algorithme Reverse Factor se révèle le plus rapide, sauf sur une petite plage de données où l'algorithme PosTree est le plus efficace. L'algorithme Backward Oracle Matching [3] est une version de l'algorithme Reverse Factor qui utilise l'oracle des suffixes (voir chapitre 7 pour une définition de l'oracle de facteurs) à la place de l'automate des suffixes. La figure 3.22 résume les résultats expérimentaux présentés dans [Lec00a].

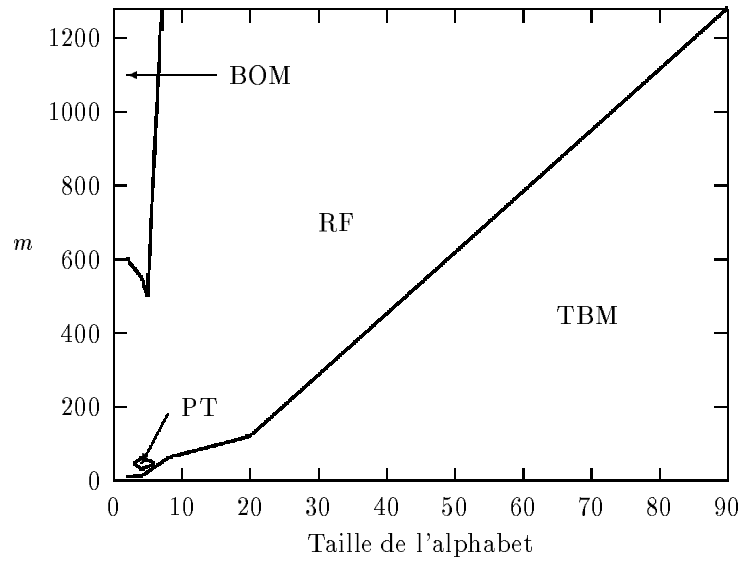


Figure 3.22 Zones de performances des algorithmes de recherche exacte de mot dans des textes de 500000 lettres (*BOM* = Backward Oracle Matching, *PT* = PosTree, *RF* = Reverse Factor, *TBM* = Tuned Boyer-Moore).

4 Alignement de séquences

Différentes méthodes d'alignement de séquences sont présentées dans [CHL00a] ainsi que dans [CL98b] où elles sont en plus accompagnées d'appliquettes permettant d'illustrer leur comportement sur des exemples simples. Une méthode de calcul d'un plus long sous-mot commun à deux mots est présentée dans [LM94] et développée dans [LLM97]. Cette méthode est étendue au calcul de la distance d'édition entre deux mots dans [LMS98].

Un aspect important de l'algorithmique du texte est constitué par la comparaison de deux ou plusieurs mots. Le problème est connu sous le nom d'alignement de séquences. Le problème de base consiste à déterminer les ressemblances, ou de manière duale, les différences entre deux mots x et y . Outre le fait qu'il existe différentes manières d'exprimer ces ressemblances ou ces différences, les solutions peuvent être visualiser sous forme d'alignements.

4.1 Alignement

Les alignements constituent l'un des procédés utilisés pour comparer deux mots. Ils permettent de visualiser la ressemblance entre ces mots. Ils 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.

Dans cette section, nous introduisons les notions d'opérations d'édition et d'alignement.

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.

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 par

$$d_{\text{préf}}(u, v) = |u| + |v| - 2 \times |lpc(u, v)|$$

pour tous $u, v \in A^*$, où $lpc(u, v)$ est le plus long préfixe commun à u et v .

Distance suffixe : distance définie symétriquement à la distance préfixe par

$$d_{\text{suff}}(u, v) = |u| + |v| - 2 \times |lsc(u, v)|$$

pour tous $u, v \in A^*$, 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 par

$$d_{\text{fact}}(u, v) = |u| + |v| - 2 \times LCF(u, v)$$

pour tous $u, v \in A^*$, 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.

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 ;
- 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 coût (de valeur entière positive) est associé à chacune des opérations. Pour $a, b \in A$, on note

- $Sub(a, b)$ le coût de la substitution de la lettre a par la lettre b ,
- $Dél(a)$ le coût de la suppression de la lettre a , et
- $Ins(b)$ le coût de l'insertion de la lettre b .

Ce faisant, on suppose implicitement que ces coûts sont indépendants des positions auxquelles les opérations sont réalisées. À partir des coûts

¹En biologie la délétion dénote la perte d'un fragment de chromosome.

Opération	Mot résultant	Coût
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 T A	1
substituer A par A	A T G C T A	0

Figure 4.1 Exemple illustrant la notion de distance d'édition. On montre dans le tableau ci-dessus une suite d'opérations élémentaires pour passer du mot **ACGA** au mot **ATGCTA**. Si, pour toutes lettres $a, b \in A$, on a les coûts $Sub(a, a) = 0$, $Sub(a, b) = 1$ lorsque $a \neq b$, et $Dél(a) = Ins(a) = 1$, le coût 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 coûts. Autrement dit, la distance d'édition entre les mots, $d(\mathbf{ACGA}, \mathbf{ATGCTA})$, est égale à 3.

élémentaires, on pose

$$d(x, y) = \min\{\text{coût de } \sigma : \sigma \in \Sigma_{x,y}\},$$

où $\Sigma_{x,y}$ est l'ensemble des suites d'opérations d'édition élémentaires permettant de transformer x en y et le coût d'un élément $\sigma \in \Sigma_{x,y}$ est la somme des coûts des opérations d'édition de la suite σ . Dans la suite du chapitre on suppose que Sub est une distance sur A et que $Dél(a) = Ins(a) > 0$ pour tout $a \in A$. La fonction d est alors une distance sur A^* , qui s'appelle la **distance d'édition** ou **distance d'alignement**. La figure 4.1 illustre les notions qui viennent d'être introduites.

Le problème du calcul de $d(x, y)$ consiste à déterminer une suite d'opérations d'édition pour transformer x en y qui minimise le coût 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.

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 4.2. 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

$$\begin{aligned} z &= (\bar{x}_0, \bar{y}_0)(\bar{x}_1, \bar{y}_1) \dots (\bar{x}_{p-1}, \bar{y}_{p-1}), \\ x &= \bar{x}_0 \bar{x}_1 \dots \bar{x}_{p-1}, \\ y &= \bar{y}_0 \bar{y}_1 \dots \bar{y}_{p-1}, \end{aligned}$$

Opération	Paire alignée	Coût
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 4.2 Suite de l'exemple de la figure 4.1. Les paires alignées sont indiquées dans le tableau ci-dessus. L'alignement correspondant est noté

$$\begin{pmatrix} \text{A} & \text{C} & \text{G} & - & - & \text{A} \\ \text{A} & \text{T} & \text{G} & \text{C} & \text{T} & \text{A} \end{pmatrix} .$$

Cet alignement est optimal car son coût, $0 + 1 + 0 + 1 + 1 + 0 = 3$, est la distance d'édition entre les deux mots.

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 coût d'une paire alignée par

$$\text{coût}(a, b) = \text{Sub}(a, b) ,$$

$$\text{coût}(a, \varepsilon) = \text{Dél}(a) ,$$

$$\text{coût}(\varepsilon, b) = \text{Ins}(b) ,$$

pour $a, b \in A$. Le coût d'un alignement est alors défini comme la somme des coûts associés à chacune de ses paires alignées.

4.2 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

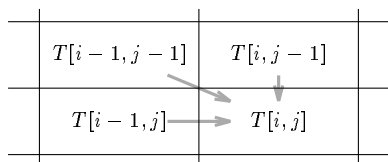


Figure 4.3 La valeur $T[i, j]$ ne dépend que des valeurs aux trois positions voisines : $T[i-1, j-1]$, $T[i-1, j]$ et $T[i, j-1]$ (lorsque $i, j \geq 0$).

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

$$T[i, j] = d(x[0..i], y[0..j])$$

pour $i = -1, 0, \dots, m-1$ et $j = -1, 0, \dots, n-1$.

Pour calculer $T[i, j]$, on utilise la formule de récurrence énoncée dans la proposition qui suit.

Proposition 4.1

Pour $i = 0, 1, \dots, m-1$ et $j = 0, 1, \dots, n-1$, on a

$$\begin{aligned} T[-1, -1] &= 0, \\ T[i, -1] &= T[i-1, -1] + \text{Dél}(x[i]), \\ T[-1, j] &= T[-1, j-1] + \text{Ins}(y[j]), \\ T[i, j] &= \min \begin{cases} T[i-1, j-1] + \text{Sub}(x[i], y[j]), \\ T[i-1, j] + \text{Dél}(x[i]), \\ T[i, j-1] + \text{Ins}(y[j]). \end{cases} \end{aligned}$$

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 4.3). Une illustration du calcul est présentée figure 4.4.

L'algorithme CALCUL-GÉNÉRIQUE, dont le code est donné figure 4.5, effectue le calcul de la distance d'édition en utilisant la table T , la valeur cherchée étant $T[m-1, n-1] = d(x, y)$.

Alors que la programmation directe de la formule de récurrence de la proposition 4.1 mène à un algorithme de complexité exponentielle, il

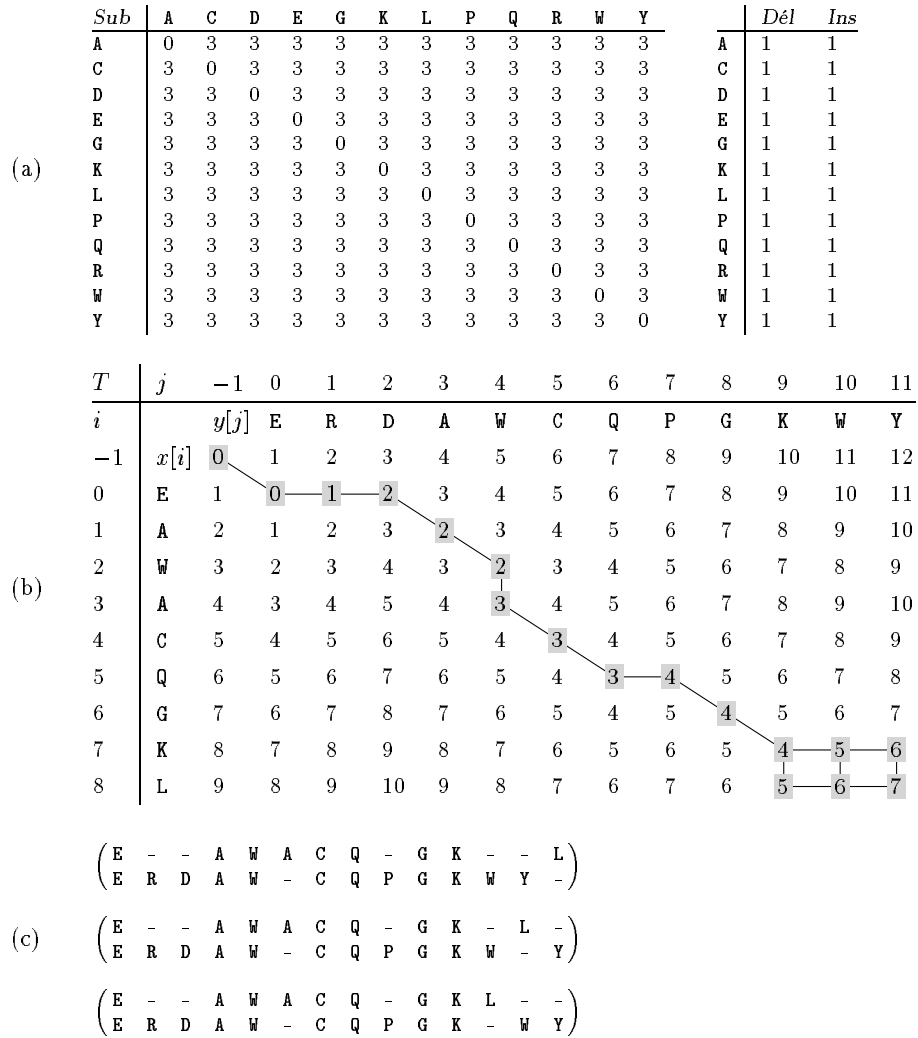


Figure 4.4 Calcul de la distance d'édition entre les deux mots EAWACQGKL et ERDAWCQPGKWWY et alignements correspondants. (a) Matrice d'édition: valeurs des coûts des opérations d'édition avec ici $\text{Sub}(a, b) = 3$ pour $a \neq b$ et $\text{Dél}(a) = \text{Ins}(a) = 1$. (b) Table T , comme calculée lors de l'exécution l'algorithme CALCUL-GÉNÉRIQUE. On obtient $d(\text{EAWACQGKL}, \text{ERDAWCQPGKWWY}) = T[8, 11] = 7$. Sont également figurés les chemins de coût 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 apparaître 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 $|\text{EAWACQGKL}| + |\text{ERDAWCQPGKWWY}| - 2 \times |\text{EAWCQGK}| = 7$.

```

CALCUL-GÉNÉRIQUE( $x, m, y, n$ )
1   $T[-1, -1] \leftarrow 0$ 
2  pour  $i \leftarrow 0$  à  $m - 1$  faire
3       $T[i, -1] \leftarrow T[i - 1, -1] + \text{Dél}(x[i])$ 
4  pour  $j \leftarrow 0$  à  $n - 1$  faire
5       $T[-1, j] \leftarrow T[-1, j - 1] + \text{Ins}(y[j])$ 
6      pour  $i \leftarrow 0$  à  $m - 1$  faire
7           $sub \leftarrow T[i - 1, j - 1] + \text{Sub}(x[i], y[j])$ 
8           $dél \leftarrow T[i - 1, j] + \text{Dél}(x[i])$ 
9           $ins \leftarrow T[i, j - 1] + \text{Ins}(y[j])$ 
10          $T[i, j] \leftarrow \min\{sub, dél, ins\}$ 
11 retourner  $T[m - 1, n - 1]$ 

```

Figure 4.5 Algorithme de calcul de la distance d'édition.

est immédiat de voir que le temps d'exécution de l'opération CALCUL-GÉNÉRIQUE(x, m, y, n) est quadratique. L'espace mémoire nécessaire est linéaire, en effet deux lignes, deux colonnes ou trois antidiagonales consécutives de la table T suffisent pour mener à bien le calcul.

Proposition 4.2

L'algorithme CALCUL-GÉNÉRIQUE produit la distance d'édition entre un mot x de longueur m et un mot y de longueur n . Il s'exécute en temps $O(m \times n)$ dans un espace $O(\min\{m, n\})$.

Calcul d'un alignement optimal

L'algorithme CALCUL-GÉNÉRIQUE ne calcule que le coût de la transformation de x en y . Pour 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 UN-ALIGNEMENT, dont le code est donné figure 4.6, implante cette méthode qui produit un alignement optimal.

Proposition 4.3

L'algorithme UN-ALIGNEMENT produit un alignement optimal entre x et y , c'est-à-dire un alignement de coût $d(x, y)$. Le calcul s'exécute en temps et espace supplémentaire $O(m + n)$.

```

UN-ALIGNEMENT( $x, m, y, n$ )
1   $z \leftarrow (\varepsilon, \varepsilon)$ 
2   $(i, j) \leftarrow (m - 1, n - 1)$ 
3  tantque  $i \neq -1$  et  $j \neq -1$  faire
4      si  $T[i, j] = T[i - 1, j - 1] + \text{Sub}(x[i], y[j])$  alors
5           $z \leftarrow (x[i], y[j]) \cdot z$ 
6           $(i, j) \leftarrow (i - 1, j - 1)$ 
7      sinon si  $T[i, j] = T[i - 1, j] + \text{Dél}(x[i])$  alors
8           $z \leftarrow (x[i], \varepsilon) \cdot z$ 
9           $i \leftarrow i - 1$ 
10     sinon  $z \leftarrow (\varepsilon, y[j]) \cdot z$ 
11          $j \leftarrow j - 1$ 
12 tantque  $i \neq -1$  faire
13      $z \leftarrow (x[i], \varepsilon) \cdot z$ 
14      $i \leftarrow i - 1$ 
15 tantque  $j \neq -1$  faire
16      $z \leftarrow (\varepsilon, y[j]) \cdot z$ 
17      $j \leftarrow j - 1$ 
18 retourner  $z$ 

```

Figure 4.6 Algorithme qui produit un alignement optimal.

```

LES-ALIGNEMENTS( $x, m, y, n$ )
1  LA( $m - 1, n - 1, (\varepsilon, \varepsilon)$ )

```

Figure 4.7 Algorithme qui produit tous les alignements optimaux.

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 4.7. Il fait appel à la procédure récursive LA (voir figure 4.8) pour laquelle les variables x , y et T sont globales.

Proposition 4.4

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.

Produire tous les alignements n'est pas judicieux si ceux-ci sont trop nombreux. Il est préférable de produire un graphe qui contient toute l'information et que l'on peut interroger ensuite.

```

LA( $i, j, z$ )
1  si  $i \neq -1$  et  $j \neq -1$ 
    et  $T[i, j] = T[i - 1, j - 1] + \text{Sub}(x[i], y[j])$  alors
2      LA( $i - 1, j - 1, (x[i], y[j]) \cdot z$ )
3  si  $i \neq -1$ 
    et  $T[i, j] = T[i - 1, j] + \text{Dél}(x[i])$  alors
4      LA( $i - 1, j, (x[i], \varepsilon) \cdot z$ )
5  si  $j \neq -1$ 
    et  $T[i, j] = T[i, j - 1] + \text{Ins}(y[j])$  alors
6      LA( $i, j - 1, (\varepsilon, y[j]) \cdot z$ )
7  si  $i = -1$  et  $j = -1$  alors
8      signaler que  $z$  est un alignement

```

Figure 4.8 Algorithme récursif qui permet de produire tous les alignements optimaux.

4.3 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 des ces mots est noté $\text{Smc}(x, y)$. La longueur (unique) des mots de $\text{Smc}(x, y)$ est notée $\text{smc}(x, y)$.

Si l'on pose $\text{Sub}(a, a) = 0$ et $\text{Dél}(a) = \text{Ins}(a) = 1$ pour $a \in A$, et si l'on suppose $\text{Sub}(a, b) > \text{Dél}(a) + \text{Ins}(b) = 2$ pour $a, b \in A$ et $a \neq b$, la valeur $T[m - 1, n - 1]$ (voir section 4.2) représente ce que l'on appelle la **distance par les sous-mots** entre x et y , distance que l'on note $d_{\text{smot}}(x, y)$. 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 4.4). C'est pourquoi l'on s'intéresse dans cette section à la recherche de plus longs sous-mots communs. La distance par les sous-mots vérifie l'égalité $d_{\text{smot}}(x, y) = |x| + |y| - 2 \times \text{smc}(x, y)$.

Une méthode naïve pour calculer $\text{smc}(x, y)$ consiste à considérer tous les sous-mots de x , vérifier s'ils sont des sous-mots de y et 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 4.2, il est possible de calculer $\text{Smc}(x, y)$ et $\text{smc}(x, y)$ en temps et en espace $O(m \times n)$. La méthode mène naturel-

```

SMC-SIMPLE( $x, m, y, n$ )
1  pour  $i \leftarrow -1$  à  $m - 1$  faire
2       $S[i, -1] \leftarrow 0$ 
3  pour  $j \leftarrow 0$  à  $n - 1$  faire
4       $S[-1, j] \leftarrow 0$ 
5      pour  $i \leftarrow 0$  à  $m - 1$  faire
6          si  $x[i] = y[j]$  alors
7               $S[i, j] \leftarrow S[i - 1, j - 1] + 1$ 
8          sinon  $S[i, j] \leftarrow \max\{S[i - 1, j], S[i, j - 1]\}$ 
9  retourner  $S[m - 1, n - 1]$ 

```

Figure 4.9 Algorithme qui calcule la longueur d'un plus long sous-mot commun à deux mots.

lement 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 $i = -1, 0, \dots, m - 1$ et $j = -1, 0, \dots, n - 1$ par

$$S[i, j] = \begin{cases} 0 & \text{si } i = -1 \text{ ou } j = -1 \text{ ,} \\ \text{smc}(x[0..i], y[0..j]) & \text{sinon .} \end{cases}$$

Calculer $\text{smc}(x, y) = S[m - 1, n - 1]$ repose sur une simple observation qui conduit à la relation de récurrence de l'énoncé suivant (voir aussi figure 4.4).

Proposition 4.5

Pour $i = 0, 1, \dots, m - 1$ et $j = 0, 1, \dots, n - 1$, on a

$$S[i, j] = \begin{cases} S[i - 1, j - 1] + 1 & \text{si } x[i] = y[j] \text{ ,} \\ \max\{S[i - 1, j], S[i, j - 1]\} & \text{sinon .} \end{cases}$$

L'égalité donnée dans l'énoncé précédent est utilisée par l'algorithme SMC-SIMPLE (voir figure 4.9) pour calculer toutes les valeurs de la table S et produire $\text{smc}(x, y) = S[m - 1, n - 1]$.

Un exemple de calcul est montré figure 4.10.

Proposition 4.6

L'algorithme SMC-SIMPLE calcule la longueur maximale des sous-mots communs à x et y . Il s'exécute en temps et espace $O(m \times n)$.

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 4.10), comme effectué dans la section

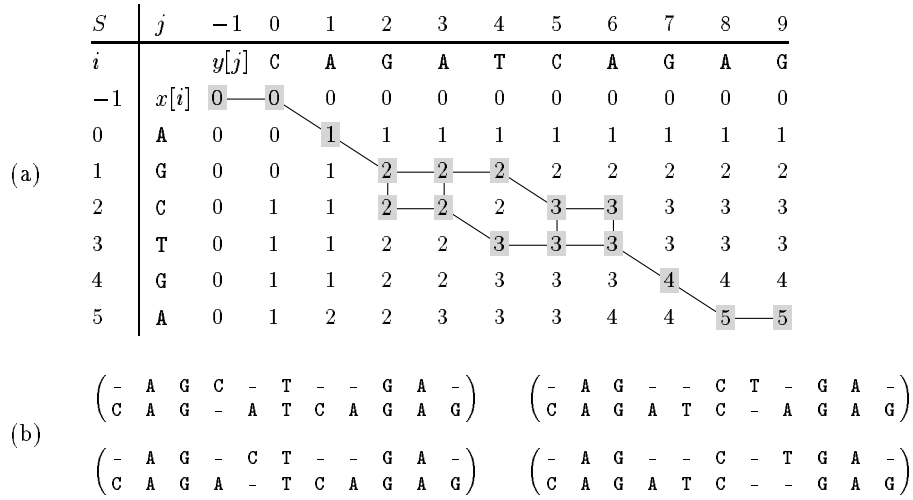


Figure 4.10 Calcul d'un plus long sous-mot commun aux mots $x = AGCTGA$ et $y = CAGATCAGAG$. (a) La table S et les chemins de coût maximal entre les positions $[-1, -1]$ et $[5, 9]$ sur la table. (b) Les quatre alignements associés. Il en résulte que les mots **AGCGA** et **AGTGA** sont les plus long sous-mots communs à x et y .

```

UN-PLUS-LONG-SOUS-MOT-COMMUN( $x, m, y, n, S$ )
1   $z \leftarrow \varepsilon$ 
2   $(i, j) \leftarrow (m - 1, n - 1)$ 
3  tantque  $i \neq -1$  et  $j \neq -1$  faire
4      si  $x[i] = y[j]$  alors
5           $z \leftarrow x[i] \cdot z$ 
6           $(i, j) \leftarrow (i - 1, j - 1)$ 
7      sinon si  $S[i - 1, j] > S[i, j - 1]$  alors
8           $i \leftarrow i - 1$ 
9      sinon  $j \leftarrow j - 1$ 
10 retourner  $z$ 
    
```

Figure 4.11 Algorithme qui produit un plus long sous-mot commun à deux mots.

4.2. Le code de la figure 4.11 effectue ce calcul à la façon de l'algorithme UN-ALIGNEMENT.

Il est bien sûr possible de calculer comme dans la section 4.2 tous les plus longs sous-mots communs à x et y en prolongeant la technique utilisée dans l'algorithme précédent.

4.4 Algorithmes systoliques

Les réseaux systoliques [38] sont des architectures parallèles synchrones qui permettent de traiter efficacement un grand nombre de problèmes. Une architecture systolique est constituée d'un groupe de processeurs élémentaires (PE) simples et tous identiques. Ces PE exécutent tous la même instruction simple à chaque top d'horloge. Ils sont connectés par des canaux uni- ou bidirectionnels. Ils peuvent être disposés dans diverses configurations (ligne, matrice, grille hexagonale, arbre binaire, triangle, ...) Les données sont « impulsées » dans le réseau à un rythme régulier d'où le terme systolique qui provient d'une analogie avec le rythme du cœur humain.

Plus long sous-mot commun à deux mots

Le problème du plus long sous-mot commun consiste à exhiber un plus long sous-mot commun à deux mots x et y de longueurs respectives m et n . On supposera sans perte de généralité que $m \leq n$.

Soit $S[i, j]$ la longueur d'un plus long sous-mot commun entre $x[0..i]$ et $y[0..j]$ et $z(i, j)$ un plus long sous-mot commun à $x[0..i]$ et $y[0..j]$ avec $0 \leq i \leq m-1$ et $0 \leq j \leq n-1$. Le problème consiste donc à déterminer $z(m-1, n-1)$.

Pour cela on utilise les formules de récurrence de la proposition 4.5.

Pour $0 \leq i \leq m-1$ et $0 \leq j \leq n-1$:

$$z(i, j) = \begin{cases} z(i-1, j-1)x[i] & \text{si } x[i] = y[j] \\ z(i-1, j) & \text{si } S[i-1, j] \geq S[i, j-1] \\ z(i, j-1) & \text{sinon} \end{cases}$$

En fait, il suffit pour calculer un plus long sous-mot commun sur un réseau systolique, d'effectuer comme dans l'algorithme de Robert et Tchuente [45], de relier m processeurs avec quatre canaux : un pour faire transiter les caractères de x , deux pour faire transiter les valeurs de $S[i-1, j-1]$ et $S[i, j-1]$ dans le processeur j et un autre pour faire transiter $z(i-1, j-1)$ dans le processeur j . Chaque processeur j devra ainsi posséder trois registres : un pour stocker le caractère $y[j]$ et deux autres pour stocker les valeurs de $S[i-1, j]$ et de $z(i-1, j)$ avant le passage du caractère $x[i]$. Mais les valeurs de z sont des mots, ce qui rend l'algorithme non modulaire. Il suffit d'incrémenter la longueur de

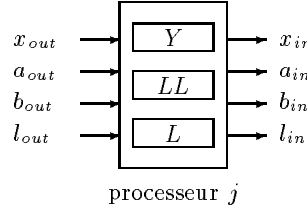
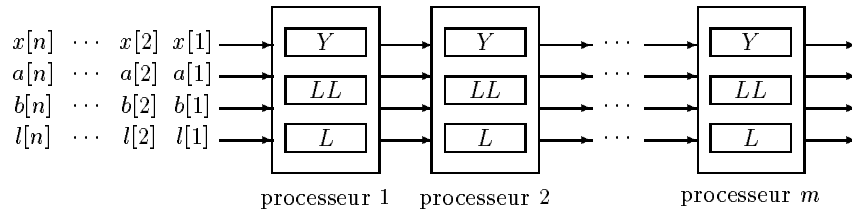


Figure 4.12 Un processeur.

Figure 4.13 Réseau linéaire de m processeurs.

$S[i-1, j-1]$ si $x[i] = y[j]$ uniquement si $S[i-1, j] < 1 + S[i-1, j-1]$. En d'autres termes le processeur j utilise le caractère $x[i] = y[j]$ pour augmenter un plus long sous-mot en construction uniquement si cela permet d'augmenter la longueur du plus long sous-mot construit lors du passage du caractère $x[i]$ dans le processeur $j-1$. Cet algorithme systolique utilise donc une ligne de m processeurs. Chaque processeur j possède trois registres : un registre Y pour stocker $y[j]$, un registre LL pour stocker $S[i, j]$ et un registre logique L pour stocker $z(i, j)$ après le traitement de $x[i]$. Ce registre devant stocker une chaîne de caractères, il est nécessaire de le représenter physiquement par ℓ registres.

Quatre canaux relient ces processeurs (voir la figure 4.12). Un de ces canaux devant faire transiter une chaîne de caractères, il est nécessaire de le représenter par plusieurs canaux physiques. Les lettres $x[i]$ sont envoyés successivement dans la ligne de processeurs comme l'indique la figure 4.13.

Avant le traitement de la lettre $x[i]$ par le processeur j la situation est la suivante :

$$\begin{aligned} Y &= y[j] \\ LL &= S[i-1, j] \\ L &= z(i-1, j) \\ x_{in} &= x[i] \\ a_{in} &= S[i-1, j-1] \end{aligned}$$

```

1   $x_{out} \leftarrow x_{in}$ 
2  si  $b_{in} = \$$  alors
3       $b_{in} \leftarrow b_{out}$ 
4      si  $a_{in} = \$$  alors
5           $Y \leftarrow x_{in}$ 
6           $LL \leftarrow 0$ 
7           $L \leftarrow \varepsilon$ 
8           $a_{out} \leftarrow 0$ 
9      sinon  $a_{out} \leftarrow a_{in}$ 
10  sinon  $a_{out} \leftarrow LL$ 
11      si  $b_{in} > LL$  alors
12           $LL \leftarrow b_{in}$ 
13           $L \leftarrow \ell_{in}$ 
14      sinon si  $x_{in} = Y$  et  $LL < 1 + a_{in}$  alors
15           $LL \leftarrow 1 + a_{in}$ 
16           $L \leftarrow \ell_{in} + x_{in}$ 
17       $b_{out} \leftarrow LL$ 
18       $\ell_{out} \leftarrow L$ 

```

Figure 4.14 L'algorithme de chaque processeur élémentaire.

$$\begin{aligned}
 b_{in} &= S[i, j - 1] \\
 \ell_{in} &= z(i - 1, j - 1)
 \end{aligned}$$

Et après le traitement de la lettre $x[i]$ par le processeur j la situation devient :

$$\begin{aligned}
 Y &= y[j] \\
 LL &= S[i, j] \\
 L &= z(i, j) \\
 x_{out} &= x[i] \\
 a_{out} &= S[i - 1, j] \\
 b_{out} &= S[i, j] \\
 \ell_{out} &= z(i, j)
 \end{aligned}$$

Le chargement des lettres du mot y est effectué de la même manière que dans [45]. Le programme de chaque processeur est montré figure 4.14.

Cette méthode peut être étendue au calcul de la distance d'édition et d'un alignement entre deux mots (voir [LMS98]).

5 Animation d'algorithmes

Deux sites ont été développés ([CL96a],[CL98b]) et ont été présentés dans [CL99]. Le site sur la recherche exacte de mot a été exposé dans [CL98a].

Nous avons développé deux sites Web dynamiques dédiés à la recherche de motifs. Le premier, intitulé « Exact String Matching Algorithms » [CL96a] s'adresse à des informaticiens théoriciens et praticiens et présente des algorithmes de recherche exacte de mots (voir chapitre 3). Le second s'appelle « Sequence Comparison » [CL98b] et peut aider des biologistes et des informaticiens à comprendre les techniques de bases d'alignement de séquences (voir chapitre 4).

L'émergence d'Internet et la multiplication de serveurs Web permet la publication d'innombrables documents sur le réseau. En informatique la plupart des ressources disponibles sont constituées de documents existants accessibles sous forme PostScript ou HTML (Hyper Text Markup Language). Même si ces derniers essaient de tirer avantage des fonctionnalités hypertextes, ils sont souvent très simples. Ainsi les algorithmes présentés ne tirent pas parti des possibilités dynamiques du langage. La conception d'outils pédagogiques est un souci important chez les informaticiens. Malheureusement de nombreux projets ont été lancés avant l'arrivée du Web et la venue de Java. Et donc dans bien des cas, il faudrait faire face à de trop nombreuses difficultés pour adapter ces outils au réseau. De plus ils utilisent souvent des technologies trop « agressives » pour être partagées sauf par des serveurs Intranet.

La généralisation d'HTML qui est maintenant le langage universel pour l'hypertexte depuis l'arrivée des CCS (Cascading Style Sheets) et l'intégration de Java aux clients Web, donne aujourd'hui la meilleure solution technique pour concevoir des documents dynamiques ouverts :

- Java est sécurisé tant au niveau statique qu'au niveau dynamique.

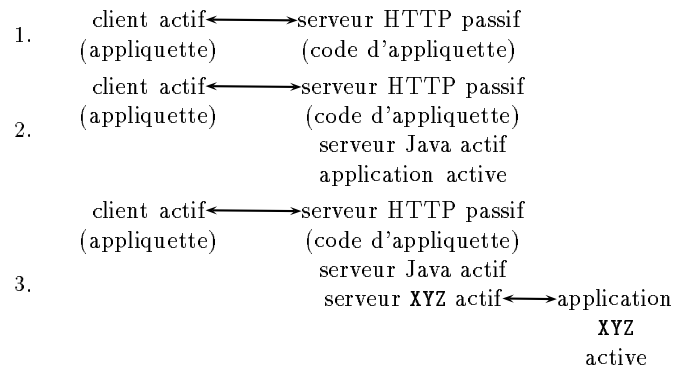


Figure 5.1 Différents niveaux d'intégration d'unités Java dans des documents Web.

Cela permet aux applets Java d'être acceptées sur quasiment tous les sites ;

- Java hérite à la fois des traditions de la programmation modulaire et de la conception orientée objet ;
- Java possède un nombre grandissant d'unités partagées prêtes-à-l'emploi ;
- jusqu'à maintenant les différentes versions de Java possèdent une bonne compatibilité ascendante ;
- la plupart des constructeurs supportent Java.

L'intégration d'unités Java dans des documents Web peut être faite à différents niveaux (voir figure 5.1). Le niveau 3 évite les contraintes pour une applet Java de ne communiquer seulement qu'avec la machine où réside le HTTP (Hyper Text Transport Protocol) qui émet le code. Les niveaux 2 et 3 semblent plus attractifs que le niveau 1 car ils permettent d'incorporer des applications existantes mais ils imposent deux conditions. La première (évidente) est d'avoir les ressources nécessaires pour supporter toutes les connexions. La seconde, souvent cachée, est d'acquiescer toutes les licences pour obtenir les droits d'utiliser les applications sur le Web. Quelques constructeurs ont déjà annoncé que l'utilisation de leurs produits sur le Web n'est pas acceptable avec de simples licences de site. La première solution, même si elle implique d'écrire du code Java, est celle qui impose le moins de contraintes. De plus c'est celle qui tire le plus avantage des liens entre codes actifs et hypertexte.

Coder en Java requiert une bonne préparation de l'architecture générale du système de façon à ce que l'intégration de nouveaux programmes puisse se faire par la dérivation d'un nombre minimal de classes. Aussi l'arborescence des classes entre l'applet et le programme effectif ne doit pas être trop lourde pour que le client soit réellement capable de

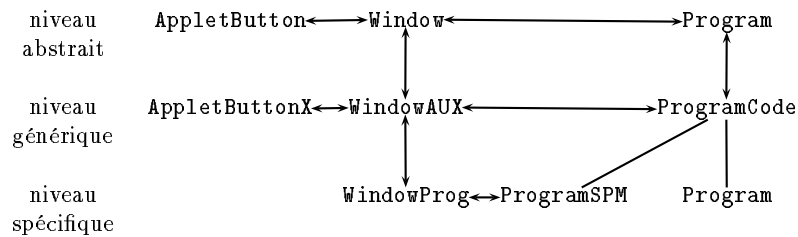


Figure 5.2 Architecture générale des différents systèmes

charger le code source.

L'architecture choisie pour les deux systèmes avaient été préalablement utilisées pour illustrer des algorithmes classiques de tri et des algorithmes sur les graphes. Les appliquestes possèdent un seul bouton qui permet de charger et lancer les programmes. Les programmes sont organisés en deux arborescences, une pour l'Interface Graphique Utilisateur (*Graphic User Interface*) et une pour l'algorithme, qui communiquent virtuellement niveau par niveau (voir figure 5.2). Les interactions dynamiques, lorsque les objets sont créés, sont faites par envois de messages.

D'une application à l'autre, l'arborescence des algorithmes n'a pas besoin d'être modifiée. Pour le site «Exact String Matching Algorithms», une classe `ProgramSPM` est dérivée de `ProgramCode` et les algorithmes classiques sont dérivés de `SPM`. Seuls des algorithmes très spécifiques ont besoin de méthodes spécifiques. Le niveau GUI demande plus de changements en fonction du type d'illustration choisi.

Les deux sites sont organisés comme suit. Les différentes notions sont expliquées et des appliquestes Java permettent à l'utilisateur d'illustrer ces notions avec des valeurs de son choix. Chaque utilisateur est alors capable de choisir l'ensemble des valeurs qui lui semble les plus judicieuses pour comprendre les notions difficiles de chaque algorithme. Nous allons maintenant décrire les deux sites.

5.1 Exact String Matching Algorithms

La recherche exacte de mot consiste à localiser toutes les occurrences d'un mot x dans un texte y (voir chapitre 3). Les différentes solutions à ce problème peuvent différer grandement. Elles utilisent des techniques combinatoires ou des heuristiques. Comprendre une solution ne permet pas systématiquement d'en comprendre une autre. Comprendre toutes ces techniques peut aider à comprendre des algorithmes résolvant des problèmes encore plus complexes. Ils nous a paru intéressant d'offrir un outil convivial pour mieux les appréhender.

Dans notre système l'utilisateur peut choisir parmi environ trente algorithmes. Pour chaque algorithme il peut alors entrer un mot et un

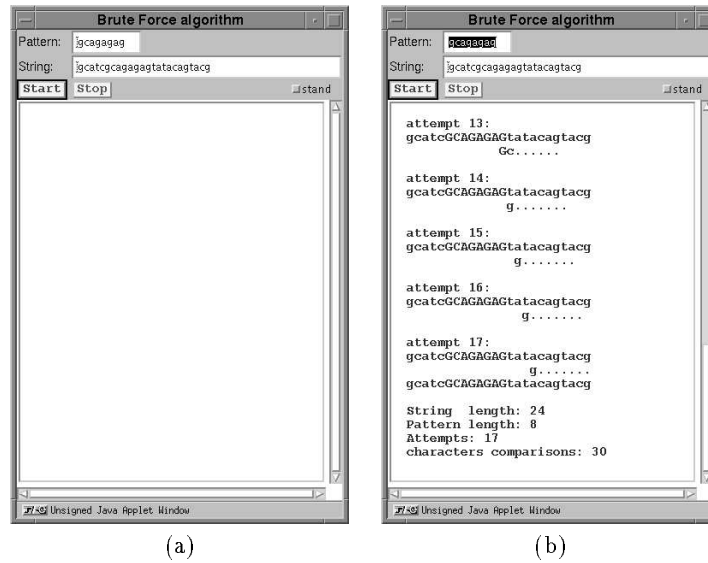


Figure 5.3 La fenêtre correspondant à l'algorithme naïf: (a) avant la recherche; (b) après la recherche.

texte (respectivement `gcagagag` et `gcatcgagagagtatacagtacg` par défaut, voir figure 5.3(a)). Les deux doivent être entrés en minuscules. Un bouton permet de lancer la recherche et un autre permet de la stopper à tout moment. La phase de recherche est alors présentée tentative par tentative. À chaque tentative, le texte apparaît et la fenêtre est matérialisée par des points sous le texte. Chaque comparaison positive fait apparaître la lettre en majuscule dans le mot et chaque comparaison négative la fait apparaître en minuscule. Une occurrence du mot dans le texte est matérialisée par le mot apparaissant en majuscule dans le texte. À la fin de la recherche, le nombre de tentatives et le nombre total de comparaisons est donné (voir figure 5.3(b)). L'utilisateur peut alors visualiser toute la recherche à l'aide d'un ascenseur situé à droite de la fenêtre de recherche.

Les différents boutons pour chaque algorithme de recherche de mot sont traités par une classe appelée `AppletButton2` dont le graphe d'héritage est montré figure 5.4 (`AppletButton1` est une appliquette avec i entrées avec $0 \leq i \leq 3$).

La fenêtre visualisant la phase de recherche est traitée par une classe nommée `ProgramTextWindow2` dont le graphe d'héritage est montré figure 5.5.

Tous les algorithmes de recherche héritent de la classe `ProgramSPM` dont le graphe d'héritage est montré figure 5.6. Cette classe `ProgramSPM` contient les méthodes suivantes :

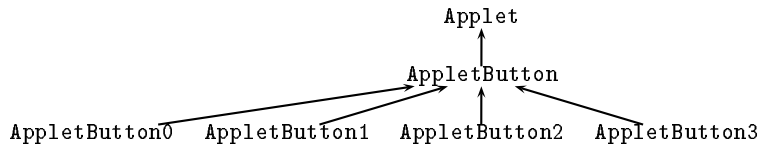


Figure 5.4 Graphe d'héritage de la classe AppletButton.

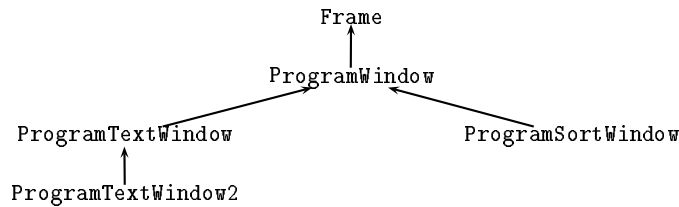


Figure 5.5 Graphe d'héritage de la classe ProgramWindow.

- `showAttemptAt(j)` : affiche m points sous les positions j à $j + m - 1$ sur le texte ;
- `EqCharsAt(j, i)` : teste si $x[i] = y[j]$ et affiche $x[i]$ suivant le résultat de la comparaison ;
- `NotEqCharsAt(j, i)` : teste si $x[i] \neq y[j]$ et affiche $x[i]$ suivant le résultat de la comparaison ;
- `showMatch(j)` : affiche une occurrence du mot à la position j sur le texte ;
- `showComparisons()` : affiche le nombre de tentatives et le nombre de comparaisons effectuées pendant la phase de recherche.

Les mots x et y et leurs longueurs respectives m et n sont des attributs

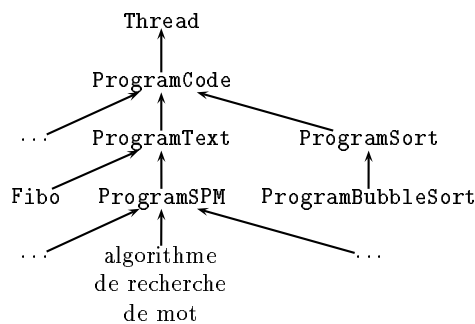


Figure 5.6 Graphe d'héritage des algorithmes de recherche de mot.

```

void BF(char *x, int m, char *y, int n) {
    int i, j;

    for (j = 0; j <= n - m; ++j) {
        i = 0;
        while (i < m && x[i] == y[i + j])
            ++i;
        if (i >= m)
            OUTPUT(j);
    }
}

```

Figure 5.7 Code C de l'algorithme naïf.

de la classe `programSPM`.

L'animation d'un algorithme de recherche de mot est très facile si le début d'une tentative (`showAttemptAt`), la comparaison de lettres (`EqCharsAt` ou `NotEqCharsAt`) et le report d'une occurrence (`showMatch`) sont clairement identifiés et séparés des autres instructions. La traduction de l'algorithme naïf (voir figure 5.7) est immédiate (voir figure 5.8).

Il est tout aussi immédiat de traduire un algorithme plus compliqué, tel l'algorithme de Colussi (figures 5.9 et 5.10). Il est donc très aisé de réaliser l'animation d'un nouvel algorithme de recherche exacte de mot.

5.2 Sequence comparison

La comparaison de deux séquences consiste à trouver le nombre minimal d'opérations d'édition pour transformer une séquence x de longueur m en une séquence y de longueur n (voir chapitre 4). La technique classiquement utilisée pour résoudre ce problème est la programmation dynamique. Elle utilise une matrice de taille $(m + 1) \times (n + 1)$. La solution est généralement donnée sous la forme d'un ou plusieurs alignements. Ces techniques sont essentielles en biologie moléculaire : lorsqu'une nouvelle séquence est déterminée, il est nécessaire de rechercher des similarités avec des séquences connues qui sont stockées dans des bases de séquences. Beaucoup d'outils, comme FastA [42], employés quotidiennement par les biologistes utilisent de la programmation dynamique. La plupart des biologistes utilisent ces outils comme des boîtes noires sans réellement en comprendre le fonctionnement. Il nous a là aussi paru intéressant d'offrir un outil facile pour comprendre les mécanismes mis en œuvre.

L'utilisateur peut choisir parmi différentes méthodes d'alignement. Il peut ensuite entrer ses propres valeurs pour les séquences x et y . Le calcul peut alors être lancé. Il donne les valeurs de la matrice de la programma-

```

import lirdir.aptk.InterruptedException;
import lirdir.progtext.ProgramSPM;

public final class ProgramBruteForce extends ProgramSPM {

    public void MAIN() throws InterruptedException{
        int i, j;
        for (j = 0; j <= n - m; ++j) {
            showAttemptAt(j);
            i = 0;
            while (i < m && EqCharsAt(i+j,i))
                ++i;
            if (i >= m)
                showMatch(j);
        }
        showComparisons();
    }
}

```

Figure 5.8 Code Java de l'algorithme naïf.

```

void COLUSSI(char *x, int m, char *y, int n) {
    int i, j, right, last, nd,
        h[XSIZE], next[XSIZE], shift[XSIZE];

    preColussi(x, m, h, next, shift, &nd);

    /* Searching */
    j = right = 0;
    last = -1;
    while (j <= n - m) {
        i = right;
        while (i < m && last < j + h[i] && x[h[i]] == y[j + h[i]])
            ++i;
        if (i >= m || last >= j + h[i]) {
            OUTPUT(j);
            i = m;
        }
        if (i > nd)
            last = j + m - 11;
        j += shift[i];
        right = next[i];
    }
}

```

Figure 5.9 Code C de l'algorithme de Colussi.

```

import lirdir.aptk.InterruptedException;
import lirdir.progtext.ProgramSPM;

public final class ProgramColussi extends ProgramSPM {

    public void MAIN() throws InterruptedException {
        int i, j, right, last, nd;
        int h[] = new int[m+1];
        int next[] = new int[m+1];
        int shift[] = new int[m+1];
        nd = preColussi(h, next, shift);
        /* Searching */
        j = right = 0;
        last = -1;
        while (j <= n - m) {
            showAttemptAt(j);
            i = right;
            while (i < m && last < j + h[i] &&
                EqCharsAt(j + h[i], h[i]))
                ++i;
            if (i >= m || last >= j + h[i]) {
                showMatch(j);
                i = m;
            }
            if (i > nd) last = j + m - 1;
            j += shift[i];
            right = next[i];
        }
        showComparisons();
    }
}

```

Figure 5.10 Code Java de l'algorithme de Colussi.

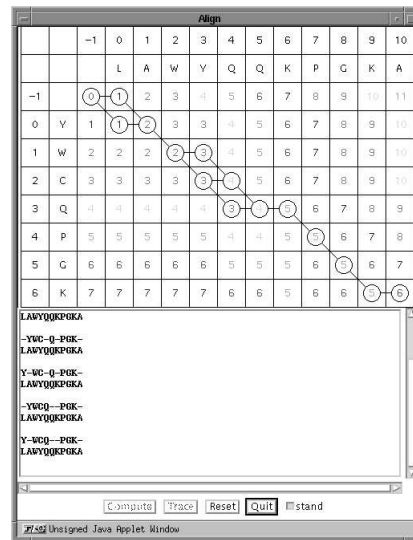


Figure 5.11 La fenêtre correspondante au calcul de la distance de Levenshtein entre deux séquences.

tion dynamique, visualise sur celle-ci les différents chemins constituants des alignements optimaux entre x et y et écrit les alignements correspondants (voir figure 5.11).

Ce site a notamment été utilisé lors de l'École thématique du CNRS intitulée «Traitement de l'information en biologie moléculaire» organisée par le groupe ABISS de l'Université de Rouen à Asnelles-sur-Mer en novembre 1998.

6 Recherche dans les séquences musicales

Les travaux de ce chapitre sont exposés dans [ILMP00] et [ILP00].

À un niveau très rudimentaire, une séquence musicale peut être représentée par un mot. L'alphabet peut être l'ensemble des notes ou l'ensemble des intervalles entre les notes (les hauteurs des notes peuvent être données par des nombres MIDI (Musical Instrument Digital Interface) et les intervalles par des nombres de demi-tons). Il faut bien sûr ajouter à cette représentation un codage pour les durées des notes. La recherche exacte de mot dans ce contexte ne présente guère d'intérêt. Nous présentons dans la suite des méthodes de recherche approchée appropriées aux séquences musicales ainsi représentées.

6.1 Recherche $\delta - \gamma$ approchée

On considère une distance entre lettres $d: A \times A \rightarrow \mathbf{R}$, deux réels positifs δ et γ . Pour deux mots u et v de longueur m on note

- $u \approx_\delta v$ si pour toute les positions i sur u et v avec $0 \leq i \leq m - 1$ on a $d(u[i], v[i]) \leq \delta$;
- $u \approx_\gamma v$ si $\sum_{i=0}^{m-1} d(u[i], v[i]) \leq \gamma$;
- $u \approx_{(\delta, \gamma)} v$ si $u \approx_\delta v$ et $u \approx_\gamma v$.

δ -, γ - et (δ, γ) -approximation

On dit que le mot x possède une occurrence

- δ -approchée dans le texte y s'il existe une position j sur y telle que $0 \leq j \leq n - m$ et $x \approx_\delta y[j..j + m - 1]$;

- γ -approchée dans le texte y s'il existe une position j sur y telle que $0 \leq j \leq n - m$ et $x \approx_\gamma y[j..j + m - 1]$;
- (δ, γ) -approchée dans le texte y s'il existe une position j sur y telle que $0 \leq j \leq n - m$ et $x \approx_{(\delta, \gamma)} y[j..j + m - 1]$.

Une première solution à ces problèmes consiste à construire une machine d'Aho-Corasick [2] de tous les mots qui sont δ -, γ -, ou (δ, γ) -approchés de x et ensuite d'utiliser cet automate pour analyser le texte y . Le temps nécessaire pour construire une telle machine est $O((\text{card } A)^\delta)$, donc cette méthode est inapplicable en pratique. Des méthodes basées sur des techniques de vecteurs de bits sont présentées dans [15]. Elles supposent que la longueur du mot x est inférieure à la taille d'un mot machine.

Il est possible d'adapter des algorithmes de recherche exacte de mots aux recherches δ -, γ - et (δ, γ) -approchées.

6.2 δ -approximation

L'adaptation des algorithmes de recherche exacte de mots Tuned-Boyer-Moore et Skip Search est présentée ci-après.

Algorithme δ -Tuned-Boyer-Moore

L'algorithme Tuned-Boyer-Moore est une implantation particulièrement rapide de l'algorithme de Boyer-Moore. Il n'utilise que le décalage de dernière occurrence définie comme suit, pour chaque lettre a de l'alphabet A

$$\text{dern-occ}[a] = \min\{m - i : x[i] = a\} \cup \{m\} \text{ .}$$

L'algorithme Tuned-Boyer-Moore gagne son efficacité en déroulant trois décalages à la suite en utilisant la lettre la plus à droite de la fenêtre et en fixant à 0 le décalage de la lettre la plus à droite du mot. Lorsqu'une occurrence de $x[m - 1]$ est localisée, un test naïf est effectué pour comparer les autres lettres de la fenêtre. Ensuite un décalage de longueur déc est effectué où déc est la distance entre l'extrémité droite du mot et l'occurrence la plus à droite de $x[m - 1]$ dans $x[0..m - 2]$:

$$\text{déc} = \min\{m - i : x[i] = x[m - 1] \text{ et } i > 1\} \cup \{m\} \text{ .}$$

Pour effectuer une recherche δ -approchée la fonction de dernière occurrence doit être définie pour chaque lettre a comme la distance entre l'extrémité droite du mot et la position de la lettre $x[i]$ la plus à droite dans le mot telle que $d(x[i], a) \leq \delta$:

$$\text{dern-occ}_\delta[a] = \min\{m - i : d(x[i], a) \leq \delta\} \cup \{m\} \text{ .}$$

```

 $\delta$ -TUNED-BOYER-MOORE( $x, m, y, n, \delta$ )
1  ▷ Prétraitement
2  pour toutes les lettres  $a \in A$  faire
3       $derm-occ_\delta[a] \leftarrow \min\{\{m - i : d(x[i], a) \leq \delta\} \cup \{m\}\}$ 
4   $déc_\delta \leftarrow \min\{\{m - i : d(x[i], x[m - 1]) \leq 2\delta\} \cup \{m\}\}$ 
5   $y[n..n + m - 1] \leftarrow x[m - 1]^m$ 
6  ▷ Recherche
7   $j \leftarrow m - 1$ 
8  tantque  $j < n$  faire
9       $k \leftarrow derm-occ_\delta[y[j]]$ 
10     tantque  $k \neq 0$  faire
11          $j \leftarrow j + k$ 
12          $k \leftarrow derm-occ_\delta[y[j]]$ 
13          $j \leftarrow j + k$ 
14          $k \leftarrow derm-occ_\delta[y[j]]$ 
15          $j \leftarrow j + k$ 
16          $k \leftarrow derm-occ_\delta[y[j]]$ 
17     si  $x[0..m - 2] \approx_\delta y[j - m + 1..j - 1]$  et  $j < n$  alors
18         SIGNALER( $j - m + 1$ )
19      $j \leftarrow j + déc_\delta$ 

```

Figure 6.1 Adaptation de l'algorithme Tuned-Boyer-Moore à la δ -approximation.

La longueur $déc$ du décalage à appliquer après la localisation d'une occurrence d'une lettre $y[j]$ telle que $d(x[m - 1], y[j]) \leq \delta$ devient

$$déc_\delta = \min\{\{m - i : d(x[i], x[m - 1]) \leq 2\delta \text{ et } i > 1\} \cup \{m\}\} .$$

L'algorithme δ -Tuned-Boyer-Moore est présenté figure 6.1.

Algorithme δ -Skip Search

Pour adapter l'algorithme Skip Search à la δ -approximation, les compartiments doivent être calculés de la façon suivante :

$$z_\delta[a] = \{i : d(x[i], a) \leq \delta\} .$$

L'algorithme δ -SKIP-SEARCH est montré figure 6.2.

6.3 (δ, γ) -approximation

Pour adapter les algorithmes δ -TUNED-BOYER-MOORE et δ -SKIP-SEARCH à la (δ, γ) -approximation il suffit de modifier légèrement la vérification naïve du contenu de la fenêtre. On obtient des algorithmes nommés (δ, γ) -TUNED-BOYER-MOORE et (δ, γ) -SKIP-SEARCH.

```

 $\delta$ -SKIP-SEARCH( $x, m, y, n, \delta$ )
1  ▷ Prétraitement
2  pour toutes les lettres  $a \in A$  faire
3       $z_\delta[a] \leftarrow \{i : d(x[i], a) \leq \delta\}$ 
4  ▷ Recherche
5   $j \leftarrow m - 1$ 
6  tantque  $j < n$  faire
7      pour toutes les positions  $i \in z_\delta[y[j]]$  faire
8          si  $x \approx_\delta y[j - i .. j - i + m - 1]$  alors
9              SIGNALER( $j - i$ )
10      $j \leftarrow j + m$ 

```

Figure 6.2 Adaptation de l'algorithme Skip-Search à la δ -approximation.

TAB. 6.1 - Temps d'exécution pour la δ -approximation avec $\delta = 5$.

m	SHIFT-AND	δ -TUNED-BOYER-MOORE	δ -SKIP-SEARCH
8	32.98	10.78	18.61
9	32.90	10.55	18.11
10	32.93	10.10	17.65
20	32.86	9.32	15.81

6.4 Résultats expérimentaux

Nous avons implanté en C de manière homogène les algorithmes suivants : SHIFT-AND, δ -TUNED-BOYER-MOORE, δ -SKIP-SEARCH, SHIFT-PLUS, (δ, γ) -TUNED-BOYER-MOORE et (δ, γ) -SKIP-SEARCH. Nous avons cherché 100 mots aléatoirement construits dans un texte, également aléatoirement construit, de 500000 lettres. La taille de l'alphabet est 70. Les temps donnés sont les temps moyens exprimés en centièmes de secondes pour chercher un mot. Ces temps tiennent compte de la phase de prétraitement et de la phase de recherche.

Les résultats pour la δ -approximation sont donnés tables 6.1 à 6.5. Pour les valeurs utilisées dans ces expériences l'algorithme δ -TUNED-BOYER-MOORE est toujours meilleur que l'algorithme δ -SKIP-SEARCH qui lui-même est toujours meilleur que l'algorithme SHIFT-AND.

TAB. 6.2 - Temps d'exécution pour la δ -approximation avec $\delta = 6$.

m	SHIFT-AND	δ -TUNED-BOYER-MOORE	δ -SKIP-SEARCH
8	33.07	13.40	21.66
9	32.90	13.00	20.94
10	32.93	12.64	20.49
20	32.92	11.97	18.81

TAB. 6.3 - *Temps d'exécution pour la δ -approximation avec $\delta = 7$.*

m	SHIFT-AND	δ -TUNED-BOYER-MOORE	δ -SKIP-SEARCH
8	33.65	16.65	24.99
9	33.14	16.05	24.06
10	33.05	15.71	23.62
20	32.93	14.82	21.42

TAB. 6.4 - *Temps d'exécution pour la δ -approximation avec $\delta = 8$.*

m	SHIFT-AND	δ -TUNED-BOYER-MOORE	δ -SKIP-SEARCH
8	34.72	21.18	29.15
9	33.41	20.03	27.64
10	33.07	19.12	26.85
20	32.81	18.20	24.41

Les résultats pour la (δ, γ) -approximation sont donnés tables 6.6 à 6.10. les valeurs utilisées dans ces expériences l'algorithme (δ, γ) -TUNED-BOYER-MOORE est meilleur que l'algorithme (δ, γ) -SKIP-SEARCH qui lui-même est meilleur que l'algorithme SHIFT-PLUS.

TAB. 6.5 - *Temps d'exécution pour la δ -approximation avec $\delta = 9$.*

m	SHIFT-AND	δ -TUNED-BOYER-MOORE	δ -SKIP-SEARCH
8	36.46	26.82	34.64
9	34.46	24.36	31.46
10	33.41	23.61	30.55
20	33.00	22.32	27.54

TAB. 6.6 - Temps d'exécution pour la (δ, γ) -approximation avec $\delta = \min\{m, 10\}$ et $\gamma = 14$.

m	SHIFT-PLUS	(δ, γ) -TUNED-BOYER-MOORE	(δ, γ) -SKIP-SEARCH
8	50.73	23.33	31.93
9	50.32	27.78	35.52
10	51.79	33.76	39.45
20	50.26	32.46	36.91

TAB. 6.7 - Temps d'exécution pour la (δ, γ) -approximation avec $\delta = \min\{m, 10\}$ et $\gamma = 15$.

m	SHIFT-PLUS	(δ, γ) -TUNED-BOYER-MOORE	(δ, γ) -SKIP-SEARCH
8	50.88	23.16	31.99
9	50.86	28.70	36.40
10	51.87	33.74	39.58
20	51.11	32.53	37.38

TAB. 6.8 - Temps d'exécution pour la (δ, γ) -approximation avec $\delta = \min\{m, 10\}$ et $\gamma = 16$.

m	SHIFT-PLUS	(δ, γ) -TUNED-BOYER-MOORE	(δ, γ) -SKIP-SEARCH
8	50.72	23.33	32.02
9	50.70	27.96	35.65
10	51.94	33.88	40.00
20	51.35	33.20	37.03

TAB. 6.9 - Temps d'exécution pour la (δ, γ) -approximation avec $\delta = \min\{m, 10\}$ et $\gamma = 17$.

m	SHIFT-PLUS	(δ, γ) -TUNED-BOYER-MOORE	(δ, γ) -SKIP-SEARCH
8	50.67	23.29	32.20
9	50.83	28.38	35.74
10	51.93	34.41	39.91
20	50.18	32.94	37.10

TAB. 6.10 - Temps d'exécution pour la (δ, γ) -approximation avec $\delta = \min\{m, 10\}$ et $\gamma = 18$.

m	SHIFT-PLUS	(δ, γ) -TUNED-BOYER-MOORE	(δ, γ) -SKIP-SEARCH
8	51.24	23.57	32.22
9	50.31	28.33	35.73
10	51.83	34.36	40.15
20	49.97	32.77	37.03

7 Recherche dans les séquences biologiques

Les résultats de ce chapitre sont parus dans [LL00]. Les méthodes de compression classiques sont présentées dans [CL96b] et [CL98c].

Nous présentons dans ce chapitre une méthode rapide pour détecter de longues répétitions dans les mots ainsi que des applications aux séquences génomiques. Les séquences génomiques peuvent être représentées par des mots sur un alphabet à quatre lettres pour l'ADN et l'ARN (alphabet des acides nucléiques) et sur un alphabet à vingt lettres pour les protéines (alphabet des acides aminés). Les séquences d'acides nucléiques peuvent dépasser les trois milliards de lettres comme c'est le cas pour le génome humain. Il est donc fondamental de trouver des structures de données qui soient à la fois compactes et rapides à manipuler pour les traiter.

7.1 Oracle des facteurs

L'oracle des facteurs d'un mot [3] est un automate qui peut être représenté de manière très compacte et qui reconnaît au moins tous les facteurs du mot. Il reconnaît quelques mots supplémentaires. La caractérisation exacte du langage reconnu par l'oracle des facteurs est toujours une question ouverte.

L'oracle des facteurs d'un mot w est un Automate Fini Déterministe $\mathcal{O}(w) = (Q, q_0, T, \delta)$. Le langage accepté par $\mathcal{O}(w)$ est tel que $\{u \in \Sigma^* : \exists v \in \Sigma^* \text{ tel que } w = vu\} \subseteq \mathcal{L}(\mathcal{O}(w))$. Sa construction est linéaire en temps et en espace en la longueur du mot. Il possède exactement $m + 1$ états et au plus $2m - 1$ transitions. En fait l'oracle consiste en un squelette formé par un chemin partant de l'état initial, parcourant m transitions et les $m + 1$ états et dont les étiquettes forment le mot lui-même. Ces $m + 1$ états et m transitions n'ont pas besoin d'être

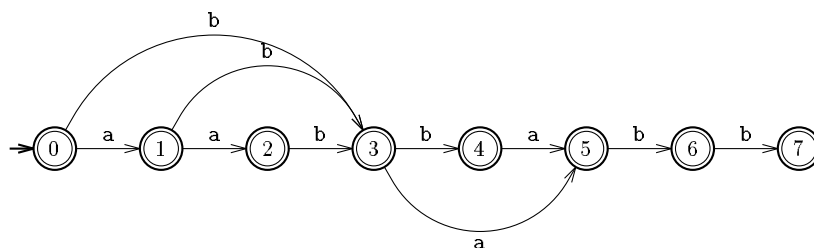


Figure 7.1 Oracle des facteurs $\mathcal{O}(\text{aabbabb})$. Tous les facteurs de aabbabb sont reconnus mais le mot ababb est reconnu bien qu'il ne soit qu'un sous-mot mais pas un facteur de aabbabb . Seuls les couples $(0, 3)$, $(1, 3)$ et $(3, 5)$ doivent être mémorisés en plus du mot aabbabb pour constituer l'oracle.

représentés puisqu'ils peuvent être déduits du mot. Donc chaque préfixe du mot (y compris le mot vide) est associé de manière univoque à un état de l'oracle. En plus de ces m transitions constituant le squelette de l'oracle, il peut y avoir quelques transitions supplémentaires (au plus $m - 1$). Toutes les transitions arrivant dans un état sont étiquetées par la dernière lettre du préfixe associé à l'état. Par conséquent ces transitions supplémentaires peuvent être représentées par des couples (état d'origine, état de destination), les étiquettes étant omises puisqu'elles peuvent être déduites du mot. La représentation de l'oracle est donc constituée du mot et d'un ensemble de transitions supplémentaires sans leurs étiquettes (voir figure 7.1).

7.2 Calcul de répétitions

Il est possible de calculer en chaque position du mot x la longueur du plus long suffixe répété en temps et en espace linéaire en utilisant des structures d'index tel un automate des suffixes ou un arbre des suffixes. Néanmoins lorsque l'on manipule des données volumineuses, ces structures de données peuvent être d'une taille prohibitive. L'oracle est une structure d'index très économique en espace qui permet de calculer une approximation de ces valeurs en ligne.

La structure particulière de l'oracle des facteurs permet de calculer une approximation de ces valeurs en temps et espace très performants. Les résultats sont suffisamment bons pour être exploités en pratique.

Dans cette section une occurrence d'un facteur u de x est repérée par sa position de fin i : $x[i - |u| + 1 .. i] = u$.

Dans l'oracle des facteurs $\mathcal{O}(x)$, pour un mot x de longueur m on définit $\text{RÉPÉT}_x(i)$ comme étant le plus long suffixe répété de $x[0 .. i]$. Le lien suffixe de l'état i , noté $S_x[i]$, est égal à l'état où $\text{RÉPÉT}_x(i)$ est reconnu. Pour $0 \leq i \leq m - 1$, $CS(i) = (k_0 = i, \dots, k_t = -1)$ est le

chemin suffixe de l'état i dans $\mathcal{O}(x)$ tel que pour tout r , $1 \leq r \leq t$, $k_r = S_x[k_{r-1}]$. Pour $0 \leq i \leq m-1$, $\text{REP}_p(i+1)$, est égal à $ux[i+1]$ où u est le plus long suffixe répété de $x[0..i]$ se terminant en une position égale au plus grand élément de $CS(S_x[i+1]-1) \cap CS(i)$. La longueur de $\text{REP}_p(i+1)$, pour $0 \leq i \leq m-1$, est égale à la longueur du plus long suffixe de $x[0..S_x[i+1]-1]$ et $x[0..i]$ plus 1.

Pendant la construction de $\mathcal{O}(x[1..i+1])$ à partir de $\mathcal{O}(x[1..i])$ et $x[i+1]$, le parcours du chemin suffixe de l'état i se termine lorsque qu'un état j , pour lequel $\delta(j, x[i+1])$ est défini, est rencontré. Le fait que $\delta(j, x[i+1])$ soit défini assure (par construction de $\mathcal{O}(x[1..i+1])$) qu'il existe un état de $CS(S_x[i+1]-1)$ dont le lien suffixe est égal à l'état j ou que $S_x[i+1]-1 = j$.

On note π_1 l'état de $CS(i)$ tel que $S_x[\pi_1] = j$. On note π_2 l'état de $CS(S_x[i+1]-1)$ tel que $S_x[\pi_2] = j$, si $S_x[i+1]-1 \neq j$ ou j si $S_x[i+1]-1 = j$.

Une occurrence d'un suffixe commun à $x[0..S_x[i+1]-1]$ et $x[0..i]$ est égal à la position de $S_x[\pi_1] = \pi_2 = j$ si $S_x[i+1]-1 = j$, et à la position $S_x[\pi_1] = S_x[\pi_2] = j$ sinon.

Soit lrs un tableau de $m+1$ entiers tel que $\forall i, 0 \leq i < m$:

$$lrs[i+1] = \begin{cases} 0 & \text{si } S_x[i+1] = 0 \text{ ,} \\ lrs[\pi_1] + 1 & \text{si } \pi_2 = S_x[\pi_1] \text{ ,} \\ \min(lrs[\pi_1], lrs[\pi_2]) + 1 & \text{sinon .} \end{cases}$$

$lrs[0]$ est affecté à 0.

On a donc $\forall i, 0 \leq i < m$, $lrs[i+1] = |\text{REP}_p(i+1)|$. On peut alors énoncer le théorème qui suit.

Théorème 7.1 (LL2000)

Les valeurs de la table lrs pour un mot x de longueur m peuvent être calculées en ligne en temps et espace $O(m)$ lors de la construction de $\mathcal{O}(x)$.

7.3 Applications

Nous présentons maintenant des applications où le calcul des répétitions est d'intérêt. La table lrs donne la longueur de plus long suffixes répétés dans un mot. Le lien suffixe en donne la position. Donc ces deux structures donnent accès à de longues répétitions dans les mots. Lorsqu'on manipule des données aussi longues que les séquences génomiques on veut avoir une représentation compacte de tous ces facteurs ce qui est le cas avec l'oracle. La table lrs peut aussi aider à calculer les répétitions à l'intérieur d'un mot ou les répétitions communes à deux mots. Cette méthode suggère assez naturellement un schéma de compression.

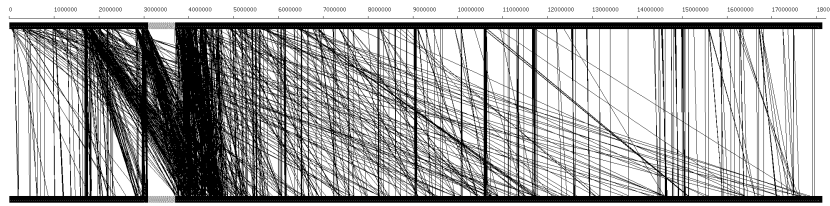


Figure 7.2 Répétitions approchées sur le chromosome IV d'*Arabidopsis thaliana*. Ce résultat a été obtenu en effectuant approximativement 12000 BLASTs [4] entre le chromosome et des fragments de celui-ci. Le centromère (partie grisée), réputé pour contenir de nombreuses répétitions n'a pas été pris en compte.

Répétitions dans les séquences génomiques

Les séquences génomiques sont un bon support pour tester notre méthode puisque :

- durant l'évolution, elles traversent une série de mutations et spécialement des duplications de gènes et des translocations qui font agir de grands blocs ;
- quelques régions contiennent de petites mais rapprochées et nombreuses répétitions (micro-satellites, . . .) ;

Nous avons appliqué notre méthode à la détection de longues répétitions dans de longues séquences et spécialement les chromosomes II et IV de la plante *Arabidopsis thaliana*. Ces deux séquences comportent approximativement 20×10^9 et 17×10^9 lettres respectivement. Il a fallu 28 secondes pour trouver toutes les occurrences de répétitions sur chaque chromosome (sur un PC sous Linux avec un processeur à 500 MHz et 1 Go de RAM). Après étude de ces répétitions, il a été possible de localiser sur le chromosome IV d'*Arabidopsis thaliana*, une répétition de longueur 30318 correspondant à une concaténation de dix régions identiques. Après vérification dans GenBank¹, cette région correspondant à un gène putatif.

Un autre résultat intéressant a été obtenu sur ce chromosome. La figure 7.2, tirée de [52] (disponible sur le site Web du MIPS²) représente les répétitions approchées sur le chromosome IV. Il a fallu quelques heures de calcul pour obtenir ce résultat alors qu'il n'a fallu qu'une minute pour obtenir la figure 7.3 avec notre méthode.

¹<http://www.ncbi.nlm.nih.gov>

²Munich Information Center for Protein Sequences, <http://websvr.mips.biochem.mpg.de>

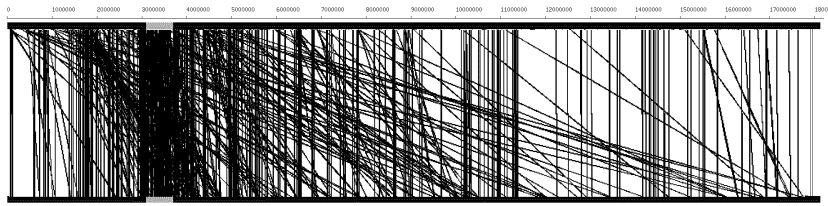


Figure 7.3 Répétitions exactes de longueur supérieure à 75 sur le chromosome IV d'*Arabidopsis thaliana*. Le centromère est ici pris en compte.

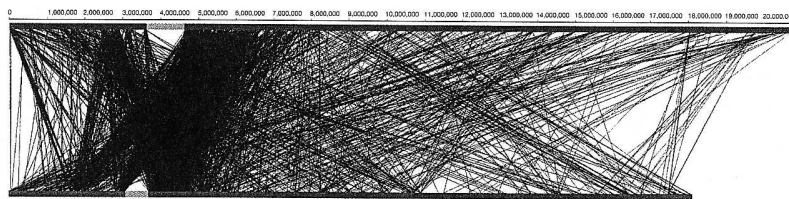


Figure 7.4 Comparaisons entre les chromosomes II et IV d'*Arabidopsis thaliana*. Les répétitions ici détectées sont approchées.

Comparaison de séquences

La recherche de similarités entre deux séquences est un domaine important de l'étude des séquences génomiques. Notre approche peut permettre de détecter des répétitions exactes communes à deux séquences. Pour comparer deux séquences x et y il suffit de construire l'oracle de $x\$y$ où $\$ \notin A$. Les liens suffixes passant de y à x permettent de détecter des répétitions communes à x et y en temps et espace linéaires. Les figures 7.4 et 7.5 présentent respectivement les résultats du MIPS et les nôtres. Le rapport entre les temps d'exécution des deux méthodes est le même que pour le calcul des répétitions sur une même séquence évoqué dans la section précédente.

Compression de données

Localiser des répétitions dans un texte mène naturellement à considérer des questions de compression de données. De nombreuses méthodes de compression de données ont été développées ([58], [59], [29], [54], [25]). Puisque notre méthode permet de détecter des répétitions exactes nous en avons dérivé un schéma de compression. Son principe est simple. Le texte à compresser est lu séquentiellement et son oracle est construit en même temps qu'il est compressé. À chaque fois qu'une nouvelle lettre

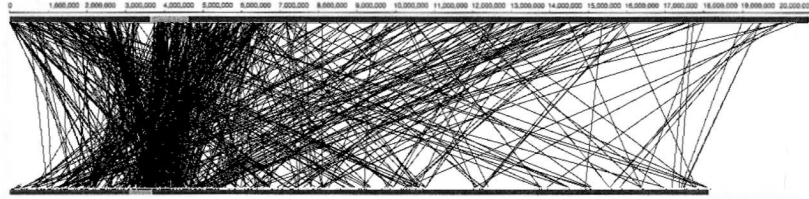


Figure 7.5 Répétitions exactes de longueur supérieure à 75 sur le chromosome IV d'*Arabidopsis thaliana*. Le centromère est ici pris en compte.

file	gzip			notre méthode			bzip2		
	%	tc	td	%	tc	td	%	tc	td
ChrIV	27	230	2	43	40	10	26	34	10
ChrII	27	261	2	43	46	11	27	40	12
bib	31	0.14	0.01	43	0.18	0.08	24	0.17	0.04
book1	40	1.18	0.08	56	2.08	0.55	30	1.47	0.47
book2	33	0.68	0.06	48	1.32	0.04	25	1.1	0.33
progc	33	0.04	0.01	52	0.07	0.04	31	0.07	0.02
trans	20	0.08	0.01	52	0.07	0.04	19	0.14	0.03

Figure 7.6 Résultats de compression entre notre méthode et gzip et bzip2. Les fichiers ChrII et ChrIV contiennent les séquences d'ADN des chromosomes II et IV d'*Arabidopsis thaliana*. Les fichiers bib, book1, book2, progc et trans sont tirés du Calgary Corpus. Pour chaque méthode, sont donnés le taux de compression (%), le temps de compression (tc) et de décompression (td) en secondes.

$x[i + 1]$ est lue, $S_p(i + 1)$ et $lrs[i + 1]$ sont calculés. Si $S_p(i + 1) = 0$, cela signifie que c'est la première occurrence de $x[i + 1]$: elle est codée comme une simple lettre. Si $S_p(i + 1) \neq 0$, cela signifie qu'un suffixe répété de longueur $lrs[i + 1]$ est localisé et une de ses occurrences est donnée par $S_p(i + 1)$. Tant que $lrs[i + 1]$ est plus grand ou égal à la différence entre la position $i + 1$ et la position de la dernière lettre codée on progresse dans le texte. Lorsque $lrs[i + 1]$ est plus petit que la différence, la région répétée est codée par un couple (longueur, position).

Des résultats expérimentaux sont présentés figure 7.6.

8 Perspectives

Toutes les techniques développées dans le cadre de la recherche exacte d'un seul mot doivent pouvoir être développées et popularisées pour la recherche d'un mot dans des dimensions supérieures, pour la recherche approchée, pour la recherche distribuée, pour la recherche d'un ensemble fini de mots et la recherche d'un ensemble infini. Il reste encore beaucoup d'efforts à faire pour obtenir des solutions distribuées réellement utilisables pour les méthodes de recherche de motifs ou d'alignement de séquences. Aucune méthode satisfaisante et efficace d'alignement de séquences d'acides nucléiques n'existe qui tient réellement compte de la traduction éventuelle des codons (triplet d'acides nucléiques) en acides aminés (à l'aide du code génétique). L'apport de techniques combinatoires et algorithmiques devrait aboutir à l'élaboration d'outils efficaces d'analyse des séquences musicales, intégrant toutes les facettes de ces données. Enfin, la cadence infernale des programmes de séquençage des génomes d'organismes de toute provenance demande des outils d'analyse qui soient à la fois performants et qui soient capables de gérer de grosses masses de données. En particulier la comparaison de deux génomes entiers est une question qui commence à se poser.

Bibliographie

Références de l'auteur

Les résultats présentés dans ce mémoire ont fait l'objet des publications suivantes :

- [CL96a] C. Charras and T. Lecroq. Exact string matching algorithms. URL:<http://www-igm.univ-mlv.fr/~lecroq/string/>, 1996.
- [CL98a] C. Charras and T. Lecroq. Exact string matching animation in Java. In J. Holub and M. Šimánek, editors, *Proceedings of the Prague Stringology Club Workshop '98*, pages 36–43, Prague, République Tchèque, 1998. Collaborative Report DC-98-06 – également rapport LIR 97.10.
- [CL98b] C. Charras and T. Lecroq. Sequence comparison. URL:<http://www-igm.univ-mlv.fr/~lecroq/seqcomp/>, 1998.
- [CL99] C. Charras and T. Lecroq. Java tools to help understanding pattern matching techniques. In D.C. Dimitrov, editor, *Proceedings of the 5th International Conference on Computer Aided Engineering Education*, pages 105–111, Sofia, Bulgarie, 1999.
- [CL00] C. Charras and T. Lecroq. Fast string matching with position tree. manuscrit, 2000.
- [CLP98] C. Charras, T. Lecroq, and J.D. Pehoushek. A very fast string matching algorithm for small alphabets and long patterns. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 55–64, Piscataway, NJ, 1998. Springer-Verlag, Berlin.

- [CLP00] C. Charras, T. Lecroq, and J.D. Pehoushek. Fast practical string matching. In *Actes du séminaire de février 1999 du projet Algorithmique et combinatoire des structures discrètes de l'Institut franco-russe A.M.Liapunov d'informatique et de mathématiques appliqués*, 2000. À paraître.
- [CCG⁺91] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Deux méthodes pour accélérer l'algorithme de Boyer-Moore. In D. Krob, editor, *Actes des 2èmes Journées franco-belges: Théories des Automates et Applications*, number 176 in Publications de l'Université de Rouen, pages 45–63, 1991.
- [CCG⁺94] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994. Également proceedings du STACS'92 et rapport LITP 92.21.
- [CCG⁺99] M. Crochemore, A. Czumaj, L. Gąsieniec, T. Lecroq, W. Plandowski, and W. Rytter. Fast practical multi-pattern matching. *Inf. Process. Lett.*, 71((3–4)):107–113, 1999. Également rapport IGM 93-3.
- [CHL00a] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithmique du texte*. Vuibert, 2000. À paraître.
- [CHL00b] M. Crochemore, C. Hancart, and T. Lecroq. A unifying look at the Apostolico-Giancarlo string matching algorithm. *J. Disc. Algo.*, 2000. Accepté – également actes d'un séminaire de l'Institut franco-russe A.M.Liapunov.
- [CL96b] M. Crochemore and T. Lecroq. Pattern matching and text data compression algorithms. *ACM Comp. Surv.*, 28(1):39–41, 1996.
- [CL96c] M. Crochemore and T. Lecroq. Pattern matching and text data compression algorithms. In Allen B. Tucker Jr., editor, *The Computer Science and Engineering Handbook*, chapter 8, pages 162–202. CRC Press Inc., Boca Raton, FL, 1996. Également rapports IGM 95-21 et LIR 95.11.
- [CL97] M. Crochemore and T. Lecroq. Tight bounds on the complexity of the Apostolico-Giancarlo algorithm. *Inf. Process. Lett.*, 63(4):195–203, 1997. Également proceedings de WSP'96 et rapport LIR 96.15.
- [CL98c] M. Crochemore and T. Lecroq. Text data compression algorithms. In Mikhail J. Atallah, editor, *Algorithms and Theory*

- of Computation Handbook*, chapter 12. CRC Press Inc., Boca Raton, FL, 1998.
- [ILMP00] C. S. Iliopoulos, T. Lecroq, L. Mouchard, and Y. J. Pinzon. Computing repetitions in musical sequences. In M. Balík and M. Šimánek, editors, *Proceedings of the Prague Stringology Club Workshop '2000*, pages 49–59, Bratislava, Slovaquie, 2000. Collaborative Report DC-2000-03.
- [ILP00] C. S. Iliopoulos, T. Lecroq, and Y. J. Pinzon. Approximate string matching in musical sequences. manuscrit, 2000.
- [Lec92a] T. Lecroq. *Recherches de mot*. Thèse de doctorat, Université d'Orléans, France, 1992. Également rapport LITP 92.15.
- [Lec92b] T. Lecroq. A variation on the Boyer-Moore algorithm. *Theor. Comput. Sci.*, 92(1):119–144, 1992.
- [Lec95] T. Lecroq. Experimental results on string matching algorithms. *Softw. Pract. Exp.*, 25(7):727–765, 1995. Également rapport LITP 94.12.
- [Lec98a] T. Lecroq. Experiments on string matching in memory structures. *Softw. Pract. Exp.*, 28(5):561–568, 1998. Également rapport LIR 97.01.
- [Lec98b] T. Lecroq. Theoretical and practical aspects of string matching. In *Abstracts of invited lectures and short communications delivered at the 7th International Colloquium on Numerical Analysis and Computer Science with Applications*, page 75, Plovdiv, Bulgarie, 1998.
- [Lec99] T. Lecroq. Theoretical and practical string matching. In R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop on Combinatorial Algorithms*, page 2, Perth, Australie, 1999.
- [Lec00a] T. Lecroq. New experimental results on exact string-matching. Rapport LIFAR 2000.03, Université de Rouen, France, 2000.
- [Lec00b] T. Lecroq. Une introduction à la recherche de mot. Actes de l'École Jeunes Chercheurs Algorithmique et Calcul Formel du GDR ALP, Caen, France, 2000.
- [LLM97] T. Lecroq, G. Luce, and J.-F. Myoupo. A faster linear systolic algorithm for recovering a longest common subsequence. *Inf. Process. Lett.*, 61(3):129–136, 1997. Également rapport LaRIA 96-07.

- [LM94] T. Lecroq and J.-F. Myoupo. Détermination d'un plus long sous-mot commun à deux mots sur un réseau linéaire. In L. Bougé, M. Cosnard, and P. Fraigniaud, editors, *Actes des 6èmes Rencontres Francophones du Parallélisme*, page 301, Lyon, France, 1994. Également rapport LIR 95.08.
- [LMS98] T. Lecroq, J.-F. Myoupo, and D. Semé. A one-phase parallel algorithm for the sequence alignment problem. *Parallel Process. Lett.*, 8(4):515–526, 1998. Également rapport LaRIA 97–16.
- [LL00] A. Lefebvre and T. Lecroq. Computing repeated factors with a factor oracle. In L. Brankovic and J. Ryan, editors, *Proceedings of the 11th Australasian Workshop on Combinatorial Algorithms*, pages 145–158, Hunter Valley, Australie, 2000.

Autres références

- [1] A. V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Algorithms and complexity*, volume A, chapter 5, pages 255–300. Elsevier, Amsterdam, The Netherlands, 1990.
- [2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.
- [3] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: a new structure for pattern matching. In J. Pavelka, G. Tel, and M. Bartosek, editors, *SOFSEM'99, Theory and Practice of Informatics*, number 1725 in Lecture Notes in Computer Science, pages 291–306, Milovy, Czech Republic, 1999. Springer-Verlag, Berlin.
- [4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. A basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.
- [5] J.-I. Aoe, editor. *String pattern matching strategies*. IEEE Computer Society Press, 1994.
- [6] A. Apostolico and M. Crochemore. Optimal canonization of all substrings of a string. *Inf. Comput.*, 95(1):76–95, 1991.
- [7] A. Apostolico and Z. Galil, editors. *Combinatorial algorithms on words*. Number 12. Springer-Verlag, Berlin, nato advanced science institutes, serie f edition, 1985.
- [8] A. Apostolico and Z. Galil, editors. *Pattern matching algorithms*. Oxford University Press, 1997.

- [9] A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, 1986.
- [10] R. A. Baeza-Yates and C. H. Perleberg. Fast and practical approximate string matching. *Inf. Process. Lett.*, 59(1):21–27, 1996.
- [11] R.A. Baeza-Yates and G.H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
- [12] T. Berry and S. Ravindran. A fast string matching algorithm and experimental results. In J. Holub and M. Šimánek, editors, *Proceedings of the Prague Stringology Club Workshop '99*, pages 16–28, Czech Technical University, Prague, Czech Republic, 1999. Collaborative Report DC–99–05.
- [13] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40(1):31–55, 1985.
- [14] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
- [15] E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. In R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, pages 129–144, Perth, WA, Australia, 1999.
- [16] R. Cole. Tight bounds on the complexity of the Boyer-Moore string matching algorithm. *SIAM J. Comput.*, 23(5):1075–1091, 1994.
- [17] R. Cole and R. Hariharan. Tighter upper bounds on the exact complexity of string matching. *SIAM J. Comput.*, 26(3):803–856, 1997.
- [18] L. Colussi. Correctness and efficiency of the pattern matching algorithms. *Inf. Comput.*, 95(2):225–251, 1991.
- [19] L. Colussi. Fastest pattern matching in strings. *J. Algorithms*, 16(2):163–189, 1994.
- [20] M. Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.
- [21] M. Crochemore and C. Hancart. Automata for matching patterns. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 2 Linear Modeling: Background and Application, chapter 9, pages 399–462. Springer-Verlag, Berlin, 1997.

- [22] M. Crochemore and C. Hancart. Pattern matching in strings. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, pages 11.1–11.28. CRC Press Inc., Boca Raton, FL, 1998.
- [23] M. Crochemore and D. Perrin. Two-way string-matching. *J. Assoc. Comput. Mach.*, 38(3):651–675, 1991.
- [24] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.
- [25] P. Fenwick. The Burrows-Wheeler transform for block sorting text compression – principles and improvements. *Comput. J.*, 39(9):731–740, 1996.
- [26] Z. Galil. On improving the worst case running time of the Boyer-Moore string searching algorithm. *Commun. ACM*, 22(9):505–508, 1979.
- [27] Z. Galil and R. Giancarlo. On the exact complexity of string matching: upper bounds. *SIAM J. Comput.*, 21(3):407–437, 1992.
- [28] Z. Galil and J. Seiferas. Time-space optimal string matching. *J. Comput. Syst. Sci.*, 26(3):280–294, 1983.
- [29] R. Gallager. Variations on a theme by Huffman. *IEEE Trans. Inf. Theory*, 24(6):668–674, 1978.
- [30] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, Cambridge, 1997.
- [31] C. Hancart. *Analyse exacte et en moyenne d’algorithmes de recherche d’un motif dans un texte*. Thèse de doctorat, Université Paris 7, 1993.
- [32] C. Hancart. On Simon’s string searching algorithm. *Inf. Process. Lett.*, 47(2):95–99, 1993.
- [33] M. C. Harrison. Implementation of the substring test by hashing. *Commun. ACM*, 14(12):777–779, 1971.
- [34] R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.
- [35] A. Hume and D. M. Sunday. Fast string searching. *Softw. Pract. Exp.*, 21(11):1221–1248, 1991.
- [36] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

- [37] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.
- [38] H. T. Kung. Why systolic architectures. *IEEE Comput.*, 15(1):37–46, 1980.
- [39] M. Lothaire, editor. *Combinatorics on Words*. Cambridge University Press, second edition, 1997.
- [40] P.D. Michailidis and K.G. Margaritis. String matching algorithms: Survey and experimental results. *Int. J. Comput. Math.*, 2000. to appear.
- [41] J. H. Morris, Jr and V. R. Pratt. A linear pattern-matching algorithm. Report 40, University of California, Berkeley, 1970.
- [42] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. U.S.A.*, 85:2444–2448, 1988.
- [43] T. Raita. Tuning the Boyer-Moore-Horspool string searching algorithm. *Softw. Pract. Exp.*, 22(10):879–884, 1992.
- [44] T. Raita. On guards and symbol dependencies in substring search. *Softw. Pract. Exp.*, 29(11):931–941, 1999.
- [45] Y. Robert and M. Tchuente. A systolic array for the longest common subsequence problem. *Inf. Process. Lett.*, 21(4):191–198, 1985.
- [46] W. Rytter. A correct preprocessing algorithm for Boyer-Moore string searching. *SIAM J. Comput.*, 9(3):509–512, 1980.
- [47] I. Simon. String matching algorithms and automata. In R. Baeza-Yates and N. Ziviani, editors, *Proceedings of the 1st South American Workshop on String Processing*, pages 151–157, Universidade Federal de Minas Gerais, Brazil, 1993.
- [48] P. D. Smith. Experiments with a very fast substring search algorithm. *Softw. Pract. Exp.*, 21(10):1065–1074, 1991.
- [49] W. F. Smyth. *Computing patterns in strings*. Addison Wesley Longman, 2001. à paraître.
- [50] G. A. Stephen. *String searching algorithms*. World Scientific Press, 1994.
- [51] D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.

- [52] Wahington University in St Louis The European Union Arabidopsis Genome Sequencing Consortium, The Cold Spring Harbor and PE Biosystems Arabidopsis Sequencing Consortium. Sequence analysis of chromosome 4 of the plant *Arabidopsis thaliana*. *Nature*, 402:769–777, 1999.
- [53] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.
- [54] I. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, 1987.
- [55] S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.
- [56] A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.
- [57] R. F. Zhu and T. Takaoka. On improving the average case of the Boyer-Moore string matching algorithm. *J. Inform. Process.*, 10(3):173–177, 1987.
- [58] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.
- [59] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inf. Theory*, 24:530–536, 1978.

Index

- acides
 - aminés, 65
 - nucléiques, 65
- ADN, 65
- algorithme systolique, 45
- alignement, 37
- alphabet, 3
- appliquettes, 50
- arbre des positions, 22
- ARN, 65
- automate
 - de Boyer-Moore, 20
 - des suffixes, 17
- bord, 3
- chemin suffixe, 67
- comparaison de séquences, 69
- compression, 69
- décalage, 6
 - de bon suffixe, 9, 15
 - de faible suffixe, 9
 - de meilleur suffixe, 9
 - valide, 6
- délétion, 36
- distance
 - d'édition, 37
 - d'alignement, 37
 - de Hamming, 36
 - par les sous-mots, 43
- facteur, 3
- fenêtre glissante, 6
- génomome, 65
- insertion, 36
- Java, 49
- lettre, 3
- lien suffixe, 66
- longueur, 3
- mémorisation de préfixe, 11, 12
- mot, 3
 - vide, 3
- motif, 5
- opération d'édition, 36
- oracle des facteurs, 65
- période, 3
- périodique, 3
- paire alignée, 38
- plus long sous-mot commun, 3, 43, 46
- position, 3
 - de décision, 20
- préfixe, 3
- processeurs élémentaires, 45
- protéines, 65
- répétitions, 66
- réseau systolique, 45
- recherche approchée, 59
- renversé, 3
- séquence
 - génomique, 65
 - musicale, 59
- sous-mot, 3
 - commun, 3
- substitution, 36
- suffixe, 3
- suppression, 36
- systolique, 45

tentative, 6
texte, 5
trou, 38

turbo-décalage, 13

Web, 49