Université de Rouen U.F.R. des sciences et techniques

École doctorale sciences physiques, mathématiques et de l'information pour l'ingénieur

Thèse

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE ROUEN

Discipline : informatique, spécialité : combinatoire et bioinformatique

par

Mikaël SALSON

Titre de la thèse :

Structures d'indexation compressées et dynamiques pour le texte

Présentée et soutenue publiquement le 8 décembre 2009 devant le jury composé de :

Julien CLÉMENT	Chargé de recherche CNRS, GREYC	Rapporteur
Éric RIVALS	Directeur de recherche CNRS, LIRMM	Rapporteur
Brigitte VALLÉE	Directrice de recherche CNRS, GREYC	Rapporteuse
Maxime CROCHEMORE	Professeur émérite, université Paris-Est	Président du jury
Thierry LECROQ	Professeur, université de Rouen	Directeur de thèse
Martine LÉONARD	Maîtresse de conférences, université de Rouen	Co-encadrante
Laurent MOUCHARD	Maître de conférences, université de Rouen	Co-encadrant
Thierry PAQUET	Professeur, université de Rouen	Examinateur

Laboratoire d'informatique, de traitement de l'information et des systèmes

Résumé

Les structures d'indexation compressées (SIC) permettent une recherche très rapide dans de grands textes en utilisant un espace inférieur à ceux-ci. L'apparition des SIC en 2000 a autorisé l'indexation de génomes entiers de mammifères. Nous introduisons une méthode qui met à jour une SIC afin de prendre en compte les modifications du texte indexé. À travers des résultats théoriques et pratiques, nous montrons que notre solution est beaucoup plus rapide que la reconstruction complète de la SIC.

Nous proposons aussi une méthode pour la recherche de minimum d'une séquence numérique pour un intervalle donné. Celle-ci est plus économe en espace que les autres méthodes et autorise la mise à jour de la séquence.

Enfin, pour rechercher des millions de courtes séquences au sein d'un génome, nous proposons une méthode qui augmente significativement le pourcentage de séquences localisées et permet d'identifier les mutations génétiques, par exemple.

Mots clés : structures d'indexation, compression de données, structures dynamiques, recherche de minimum, bioinformatique.

Compressed and dynamic full-text indexes.

Abstract

Compressed indexes use less space than the text they index and allow a very fast search into them. With the rise of these structures in 2000, we were able to index complete mammalian genomes. We introduce a method that updates a compressed index according to the modifications in the indexed text. Based on theoretical and practical results, we show that our solution is much faster than a reconstruction from scratch of the index.

Then, we propose a new space-efficient method for range minimum query, on a numeric sequence. We also show how our method can be used to allow updates.

Finally, we consider massive mapping of short sequences on a genome. Our method matches more sequences and allow to identify genetic mutations or sequence errors (due to the sequencing process which is not 100% reliable).

Keywords: indexes, data compression, dynamic structures, minimum query, bioinformatics.

Remerciements

Mes remerciements vont tout d'abord à Martine LÉONARD et Laurent MOUCHARD qui m'ont permis de travailler avec eux sur les structures d'indexation compressées dès mon mémoire de master et qui ont encadré cette thèse. Merci à vous pour le temps que vous m'avez consacré, pour vos conseils avisés, pour m'avoir fait rencontrer des gens brillants à travers le monde, pour la bonne humeur dans laquelle nous avons souvent travaillé. La liste, si elle devait être exhaustive, serait trop longue.

Je remercie aussi Thierry LECROQ qui a accepté d'être mon directeur de thèse. En quelque sorte la boucle est bouclée puisqu'il fût mon premier enseignant en informatique et donc celui par qui j'ai découvert l'informatique.

Julien CLÉMENT, Éric RIVALS et Brigitte VALLÉE ont accepté d'être rapporteurs de cette thèse et m'ont apporté des remarques constructives, dans des délais très courts, qui permettent d'améliorer la qualité de ce manuscrit. Merci à tous les trois d'avoir réalisé ce travail et d'avoir lu ce document avec tant d'attention.

J'ai eu l'occasion de rencontrer à plusieurs reprises Maxime CROCHEMORE, entre autres pour travailler sur la recherche de minimum, et c'est un grand honneur qu'il me fait en acceptant d'être membre de mon jury.

Je remercie aussi Thierry PAQUET qui m'a fait découvrir d'autres aspects de l'informatique à travers un module de l'école doctorale et qui a bien voulu être membre de mon jury.

Merci également à Costas ILIOPOULOS de m'avoir accueilli à de nombreuses reprises au King's College London et de m'avoir permis d'y réaliser mon tout premier séminaire.

Merci à Bill SMYTH pour son accueil à l'université McMaster au Canada. Merci à lui de s'être intéressé de près à nos travaux sur les structures d'indexation dynamiques pour le problème de la LZ-factorisation.

Je remercie aussi Nicolas, doctorant de son état, mais néanmoins ami, pour la discussion que nous avons eue lors d'une certaine soirée et qui nous a amenée à la collaboration évoquée plus loin. Merci également à Camille, non doctorante de son état, mais néanmoins amie, pour sa patience alors que nous travaillions avec Nicolas, lors de mes venues à Montpellier.

J'ai partagé le bureau de Ludovic et Hadrien mais aussi celui d'Arnaud, Élise et Émilie, merci à eux pour l'ambiance qui y régnait.

Merci à tous les membres du département d'informatique et de ex-ABISS. Ils ont, pour la plupart, été des enseignants qui m'ont donné goût à l'informatique mais également de sympathiques collègues.

Merci aux nombreux doctorants rencontrés pendant les formations du CIES pour les échanges enrichissants sur nos différentes façons de vivre nos thèses et la recherche de façon plus générale.

Je remercie aussi les différents membres de ma famille qui m'ont amené à toujours chercher de nouvelles approches pour tenter d'expliquer clairement mon sujet de thèse !

Enfin, *last but not least*, je tiens à remercier très chaleureusement le jury de sélection pour l'entrée en première année à l'INSA, pour m'avoir permis d'aller à l'université. Sans cela je n'aurais très probablement jamais rédigé cette thèse.

Table des matières

Table des figures

Liste des exemples

Liste des tableaux

Liste des algorithmes

Introduction générale

La quantité de données numériques accessibles croît de façon extraordinaire. Par exemple, des projets de numérisation de livres ont déjà traité des millions d'ouvrages (Hathi Trust [1], Europeana [2], Google Livres [3]), de nombreux génomes d'organismes ont été séquencés et un projet prévoit de séquencer ceux d'un millier d'individus humains [4]. L'ensemble de ces données doit pouvoir être stocké de façon à ce que l'accès y soit simple et tout en limitant l'espace mémoire utilisé. Pour que cette quantité impressionnante de données ait une utilité il est indispensable de pouvoir rechercher rapidement parmi celle-ci.

La recherche de données dans un texte est un problème qui a très rapidement connu des solutions efficaces. Ainsi Morris et Pratt (1970) ; Knuth *et al.* (1977) ; Boyer et Moore (1977) proposent des algorithmes permettant une recherche « au vol », pour lesquels l'ensemble du texte est parcouru. Dans ce cas, il n'est pas indispensable que le texte soit connu à l'avance mais le temps de recherche est proportionnel à la longueur du texte. On comprend que lorsque les textes sont très longs (séquences génomiques, banques d'ouvrages numérisés), ce type d'algorithmes est beaucoup trop long pour qu'une telle solution soit acceptable. De plus, lorsqu'il s'agit de documents disponibles à l'avance (tels les exemples cités précédemment), on peut se demander s'il est bien utile de parcourir la totalité du texte à chaque nouvelle recherche. Weiner (1973) apporte une contribution très intéressante et montre qu'il est possible de parcourir une seule fois le texte et de profiter de ce parcours pour construire une *structure d'indexation*, appelée l'*arbre des suffixes*. Une structure d'indexation est une structure de données informatique dans laquelle les informations sont stockées de manière à ce que une recherche ultérieure dans le texte indexé sera rapide avec l'aide de la structure. Ensuite, c'est en utilisant la structure d'indexation que se font toutes les recherches. L'avantage de cette approche est que le temps nécessaire pour la recherche est indépendant de la taille du texte ! Malheureusement, cet avantage a un coût et c'est celui de l'espace mémoire nécessaire pour loger la structure d'indexation. Celle-ci peut nécessiter jusqu'à 20 fois l'espace utilisé par le texte d'origine.

Bien qu'extrêmement rapide pour la recherche de données, la consommation mémoire est le principal inconvénient de l'arbre des suffixes. D'autres techniques sont apparues par la suite, leurs buts étant soit de rendre l'arbre des suffixes un peu plus économe, soit de créer de nouvelles structures d'in-

[1] `http://www.hathitrust.org/`
[2] `http://europeana.eu/`
[3] `http://books.google.com`
[4] `http://www.1000genomes.org`

dexation moins gourmandes en mémoire. Le domaine a connu un regain d'intérêt en l'an 2000 lorsque trois structures d'indexation compressées sont apparues indépendamment (Ferragina et Manzini, 2000 ; Grossi et Vitter, 2000 ; Mäkinen, 2000). Depuis les améliorations se poursuivent, tendant vers un optimum en espace (avoir une structure qui occupe aussi peu d'espace que le texte compressé lui-même) et en temps (la complexité optimale de l'arbre des suffixes).

Les résultats très intéressants de l'indexation en général et de l'indexation compressée en particulier se heurtent à un autre écueil. L'indexation ne peut être réalisée que si le texte est connu à l'avance. Sans cela, il est beaucoup plus avantageux d'utiliser un algorithme de parcours du texte que de construire la structure d'indexation pour ensuite effectuer une seule recherche. Finalement la structure d'indexation n'est « rentable » que si elle est utilisée régulièrement. Mais si le texte est modifié périodiquement, cela brise tout l'intérêt de la structure d'indexation qui devrait être reconstruite après chaque modification. On comprend que reconstruire complètement une structure d'indexation pour des modifications souvent mineures semble démesuré. Dans une telle situation, l'utilisation d'algorithmes de parcours du texte semble plus pertinente.

Lors de mon travail de thèse, nous nous sommes intéressés à la mise à jour de structures d'indexation compressées. Cela permettrait que les structures d'indexation soient aussi utilisées avec des textes susceptibles d'être modifiés. Les résultats précédents soit sont intéressants pour la théorie mais peu adaptés à la mise en pratique (Ferragina et Grossi, 1995) ; soit ne permettent pas tout type de mise à jour, c'est-à-dire insertion, suppression et substitution (Mäkinen et Navarro, 2008 ; Gallé *et al.*, 2008). Dans ce manuscrit, nous présentons un algorithme que nous avons mis au point et qui permet de mettre à jour une structure d'indexation compressée. Cependant cet algorithme peut modifier un grand nombre d'éléments dans la structure par rapport à la longueur de la modification réalisée dans le texte. Ceci n'est pas étonnant car l'insertion d'une seule lettre dans un texte peut engendrer un nombre de modifications proportionnel à la taille du texte dans n'importe laquelle des structures d'indexation compressées connues à ce jour. Nous montrons, avec l'appui de résultats théoriques et d'expériences réalisées à partir de l'implantation de notre algorithme, que, malgré ce fâcheux pire des cas, le nombre d'éléments à modifier est raisonnable par rapport à la taille du texte, ce qui permet de modifier beaucoup plus rapidement une structure d'indexation que de la reconstruire.

Nous étendons ce problème de mise à jour à l'indexation de séquences numériques. Dans cette situation, le problème n'est pas de trouver une suite donnée de nombres mais de retourner la valeur minimale dans un intervalle donné. Ce type de problème a de nombreuses applications parmi lesquelles les structure d'indexation compressées pour le texte, la factorisation d'un texte ou encore la recherche de documents. Nous avons mis au point une nouvelle approche pour résoudre ce type de problème et, de plus, nous apportons la première solution qui permette la mise à jour de la séquence numérique d'origine. Nous avons implanté la partie statique de notre méthode et la comparons à une autre solution proposée par Fischer et Heun (2007).

Ensuite, nous regardons comment il est possible d'utiliser une structure d'indexation autorisant la mise à jour, pour le problème de la LZ-factorisation. Cette LZ-factorisation a été introduite par Ziv et Lempel (1977) et est utilisée

en compression de données pour stocker plus efficacement des textes contenant des répétitions. Le calcul de cette LZ-factorisation se fait par un mécanisme de fenêtres glissantes et a donné lieu à de nombreuses publications. Nous explorons une nouvelle piste en nous intéressant aux structures d'indexation dynamiques pour ce problème. Le principe est d'avoir une structure d'indexation pour la fenêtre glissante et lorsque la fenêtre doit être déplacée nous mettons à jour la structure d'indexation.

Par ailleurs, en biologie, les processus de séquençage des gènes actifs, produisent un grand nombre de données. Ces données ont la forme de courtes séquences (50 à 100 lettres ou paires de bases) et peuvent être cherchées sur le génome afin de savoir à quel gène elles correspondent. Néanmoins, ces données peuvent contenir un certain nombre d'erreurs et qui empêchent cette recherche d'être correctement effectuée. En plus de cela, des raisons biologiques font que les courtes séquences obtenues par le processus de séquençage ne peuvent être facteur du génome. Nous étudions donc, en lien avec des biologistes, la façon de limiter le plus possible ces problèmes. Pour apporter une solution, nous utilisons des structures d'indexation compressées en raison de la taille des génomes en jeu (plusieurs gigaoctets) et de la quantité de données à identifier (plusieurs millions de séquences de longueur 50 à 100). La méthode sur laquelle nous avons travaillé permet à la fois d'identifier plus de données en étant capable de donner rapidement leurs localisations sur le génome. Mais elle permet également de caractériser les phénomènes biologiques (mutation par exemple) ou techniques (erreur lors du processus de séquençage) qui, normalement, empêchent certaines de ces données d'être identifiées. Cette approche est d'utilité pour les biologistes, afin d'identifier les mutations génétiques jouant un rôle dans certaines maladies mais également pour mieux comprendre l'importance de la partie non codante des génomes.

Dans le chapitre 1 page 5, nous présentons les définitions et concepts utiles pour appréhender plus facilement la suite du document. Nous expliquons les mécanismes régissant les principales structures d'indexation, compressées ou non, connues à ce jour. Dans le chapitre 2 page 45, nous détaillons certaines techniques largement utilisées pour le stockage des structures d'indexation compressées. Le chapitre 3 page 57 est consacré à la mise à jour des structures d'indexation. Une première partie permet d'introduire les différentes solutions existantes puis dans une seconde partie nous détaillons notre contribution à ce domaine. Nous nous intéressons ensuite à la complexité de notre algorithme durant le chapitre 4 page 81 et présentons des expériences visant à mesurer le temps de mise à jour ou de recherche ainsi que l'espace mémoire consommé par notre structure d'indexation. Nous montrons ainsi que notre algorithme, à l'inverse de solutions équivalentes, est efficace en pratique. Dans le chapitre 5 page 109, nous présentons des méthodes, en lien avec les structures d'indexation compressées, sur lesquelles nous travaillons en collaboration avec diverses équipes. Enfin dans le chapitre 6 page 133 nous concluons et tirons les perspectives sur les différents sujets abordés dans ce document.

Chapitre 1

Indexation de textes

L'indexation de textes repose, depuis le départ, sur les avancées en algorithmique du texte et, récemment, sur les résultats en compression de données. Nous présentons dans ce chapitre, afin de faciliter la lecture du document, des notions qui seront utilisées tout au long de celui-ci. Dans un premier temps, nous nous intéressons, en section 1.1, aux définitions et notations utilisées en algorithmique du texte pour travailler sur les chaînes de caractères. Nous expliquons également la façon de mesurer la compressibilité d'un texte et détaillons quelques méthodes de compression couramment utilisées pour les structures d'indexation. Dans la section 1.2 page 15, nous rappelons les méthodes classiques d'indexation de texte ainsi que les améliorations qui y ont été apportées plus récemment. Nous expliquons, en section 1.3 page 20, les approches utilisées par les différentes structures d'indexation compressées.

1.1 Définitions et notations

1.1.1 Algorithmique du texte

On appelle *alphabet*, un ensemble fini de lettres ordonnées, noté Σ. Lorsqu'il est précisé qu'on utilise un terminateur, cela signifie que l'on ajoute à l'alphabet Σ une lettre \$ inférieure à tous les éléments de Σ. L'alphabet Σ est de taille σ.

L'opérateur de concaténation est noté $\cdot$, par exemple $c_1 \cdot c_2$, avec $c_1, c_2 \in \Sigma$ représente la concaténation de deux lettres.

Un *mot* w définit sur Σ est une suite de lettres de Σ. La suite peut être vide, dans ce cas le mot vide est noté ϵ. L'ensemble des mots pouvant être générés de la sorte sur l'alphabet Σ est noté Σ^*

Un *texte* ou une *séquence* est un mot T défini sur Σ. Le nombre de lettres de T, sa longueur, est noté $|T|$ et est égal à n. Le mot vide est un texte de longueur 0.

Soit deux mots w et w' définis sur Σ, respectivement de longueurs n et n'. La concaténation de w avec w' est notée $w \cdot w'$ et $|w \cdot w'| = n + n'$. On définit la concaténation des deux mots comme le concaténation des lettres du premier avec les lettres du second : $w \cdot w' = w_0 \cdots w_{n-1} w'_0 \cdots w'_{n'-1}$.

Un texte T possède un *terminateur* si et seulement si on a ajouté à la fin du texte la lettre \$. La longueur du texte T est alors $n + 1$.

La lettre de T qui est en position i est notée $T[i]$ ou T_i. Un *facteur* de T est un mot composé de lettres consécutives dans T. Le facteur qui commence en position i et se termine en position j est noté $T[i\,..\,j]$. Un facteur qui commence à la première position de T est appelé un *préfixe*. Le préfixe qui se termine en position j est noté $T[0\,..\,j]$ ou $T[..\,j]$. Un facteur qui se termine à la dernière position de T est appelé un *suffixe*. Le suffixe qui commence en position i est noté $T[i\,..\,n-1]$ ou $T[i\,..]$. Un facteur, un préfixe ou un suffixe est dit *propre* lorsque celui-ci est différent de T. Le *miroir* de T qui correspond à $T_{n-1}\cdots T_0$ est noté $\overline{T}$.

Exemple. Soit Σ = {A,C,G,T}, $T = \overset{0\ 1\ 2\ 3\ 4\ 5}{\mathrm{AGCGCT}}$, $|T| = 6$. Le préfixe de T se terminant en position 2, $T[..2]$, est AGC. Le suffixe de T commençant en position 3, $T[3\,..]$, est GCT. Le facteur de T commençant en position 2, se terminant en position 3, $T[2\,..\,3]$ = CG. Le miroir de T, $\overline{T}= \overset{0\ 1\ 2\ 3\ 4\ 5}{\mathrm{TCGCGA}}$

Soit $P \in \Sigma^*$, appelé *motif*, de longueur m. Le texte T possède au moins une *occurrence* du mot P si et seulement si P est un facteur de T. Dans la suite, sauf mention contraire, on fera état de recherche exacte d'un motif dans un texte pour laquelle l'occurrence doit être un facteur exact de T. La position d'une occurrence de P correspond à la position de début du facteur P dans T. On note $|T|_P$ le nombre d'occurrences du motif P dans le texte T.

Exemple. Soit $T = \overset{0\ 1\ 2\ 3\ 4\ 5}{\mathrm{AGCGCT}}$ et $P = \overset{0\ 1\ 2}{\mathrm{CGC}}$. Le texte T possède une occurrence de P, en position 2.

Soit Cont_k l'ensemble des facteurs distincts de longueur k dans T, les *contextes*. Soit $c \in \mathsf{Cont}_k$, on note T^c le *texte contextuel* formé de la concaténation de chacune des lettres succédant c dans le texte T.

Exemple. Soit $T = \overset{0\ 1\ 2\ 3\ 4\ 5}{\mathrm{AGCGCT}}$ et $k = 2$.
Cont_k = {AG, GC, CG, CT}, choisissons le contexte c = GC.
On a alors T^c = GT : le contexte GC est suivi une fois par la lettre G, une autre fois par la lettre T.

On appelle *permutation circulaire* de T la concaténation d'un suffixe et d'un préfixe de T, dont la taille totale est celle de T. Dans la littérature le terme *conjugué* peut également être employé. On note $T^{[i]} = T[i\,..]T[..\,i-1]$,avec $0 \leq i < n$, la permutation circulaire d'ordre i, elle commence en position i. Si $i < n-1$, la permutation circulaire suivante est définie et correspond à $T^{[i+1]}$ et si $i > 0$ la permutation précédente est définie et correspond à $T^{[i-1]}$.

Exemple. Soit $T = \overset{0\ 1\ 2\ 3\ 4\ 5}{\mathrm{AGCGCT}}$, on a $T^{[2]} = \overset{0\ 1\ 2\ 3\ 4\ 5}{\mathrm{CGCTAG}}$, la permutation précédente est $T^{[1]} = \overset{0\ 1\ 2\ 3\ 4\ 5}{\mathrm{GCGCTA}}$ et la suivante est $T^{[3]} = \overset{0\ 1\ 2\ 3\ 4\ 5}{\mathrm{GCTAGC}}$. La permutation $T^{[0]} = T = \overset{0\ 1\ 2\ 3\ 4\ 5}{\mathrm{AGCGCT}}$.

1.1.2 Compression de données

Avant compression de données, les informations unitaires (p. ex. une lettre, un pixel) sont stockées sur un nombre fixe d'octets (p. ex. un octet par lettre,

neuf octets pour un pixel). Lorsqu'une image contient des millions de pixels ou qu'un texte est composé de milliards de lettres, ce stockage naïf induit un gaspillage d'espace trop important. A fortiori, lorsqu'il s'agit de transférer de telles données, une réduction de leur taille devient indispensable.

L'objectif de la compression de données est d'utiliser le nombre minimal de bits pour coder une information unitaire. Nous pouvons distinguer deux types de compression : avec perte, principalement appliquée aux images, vidéos et sons, elle permet d'atteindre des compressions très efficaces mais rend impossible la récupération des données d'origine ; ou sans perte, appliquée aux textes (mais aussi, de façon plus marginale, aux images et aux sons) pour lesquels il est indispensable de pouvoir retrouver le texte d'origine, après décompression.

Nous nous intéressons ici à la compression de textes, qui sera également utilisée pour leur indexation. Différents algorithmes existent pour compresser des textes, chacun ayant ses avantages et ses inconvénients. Nous présentons ici les techniques qui sont utilisées en parallèle de l'indexation de textes.

1.1.2.1 Entropie empirique d'un texte

Avant de compresser un texte, il faut savoir s'il est possible, pour le texte considéré, de réduire efficacement sa taille. Ainsi, pour mesurer la compressibilité d'un texte, Shannon (1948) a introduit la notion d'entropie d'un texte. On peut voir celle-ci comme une mesure de la « surprise » : plus on rencontre une lettre inattendue dans un texte, plus l'entropie empirique sera importante. Plus l'entropie empirique est faible, plus la compression sera efficace car il sera plus facile de déduire le contenu des textes peu « surprenants », à partir d'éléments déjà décodés.

On distingue deux notions d'entropies empiriques : celle qui tient compte des caractères du texte indépendamment les uns des autres, il s'agit de l'entropie d'ordre zéro ; et celle qui dépend de la probabilité d'apparition d'un caractère en fonction des k précédents, on parle alors d'entropie d'ordre k. La première est définie ainsi [1] :

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log(\frac{n}{n_c}) \tag{1.1}$$

La seconde est définie en fonction de l'entropie d'ordre zéro de chacun des textes contextuels :

$$H_k(T) = \sum_{c \in \mathsf{Cont}_k} \frac{|T^c|}{n} H_0(T^c) \tag{1.2}$$

L'entropie empirique d'ordre k représente le nombre minimal de bits pour compresser un texte T en considérant des contextes de longueur k. Les algorithmes originaux de Huffman (1952) et le codage arithmétique sont des exemples de méthodes de compression, appelées codages entropiques, qui arrivent à approcher l'entropie lorsque $k = 0$. Plus un texte possède fréquemment le même caractère, plus un codage entropique d'ordre 0 sera efficace.

[1] Dans la suite, sauf précision contraire, log désigne le log à base 2.

1.1.2.2 Terminologie

Dans le domaine de la compression de données, de nombreux qualificatifs permettent de définir la taille occupée par des codages de données. Nous expliquons ici les particularités de chacun d'eux.

succinct Qui utilise l'espace mémoire asymptotiquement minimal pour toutes les entrées possibles. Par exemple, il y a σ^n textes différents dans Σ^n, ils peuvent donc, au minimum, être représentés sur $n \log \sigma$ bits. Dans ce cas, si un codage de tous les textes de longueur n définis sur Σ utilise $n \log \sigma + o(n \log \sigma)$ bits, il est déclaré succinct. De façon plus générale, s'il existe N entrées quelconques distinctes, un codage utilisant $O(\log N)$ bits pour stocker ces entrées sera dit succinct.

compressé À la différence de succincte, une représentation compressée ne prend pas un espace similaire pour toutes les entrées de même taille. L'espace nécessaire dépend alors de l'entrée et de sa compressibilité (mesurée pour un texte, par exemple, par l'entropie). Une structure d'indexation sera dite compressée si elle occupe une place proportionnelle à l'entropie du texte qu'elle indexe.

compact Il n'y a pas de définition précise de « compact » et il peut recouvrir un grand nombre de significations. D'un côté, l'arbre *compact* des suffixes (voir section 1.2.1.1 page 15) n'est pas succinct : il s'agit uniquement d'une représentation réduite en espace. À l'opposé, Mäkinen (2003) définit ainsi sa table *compacte* des suffixes : « *Compact suffix array is a compressed form of the suffix array* ». Dans ce cas, il s'agit de l'utilisation de compact en tant que synonyme de compressé. L'utilisation de l'adjectif « compact » est donc à éviter autant que possible car son utilisation est très large et il ne donne aucune information quant à l'espace mémoire occupé par la structure.

auto-index Un auto-index est une structure d'indexation compressée qui permet de décompresser n'importe quel facteur de la séquence indexée. Ainsi, l'auto-index remplace, avantageusement, la séquence d'origine.

creux Un texte *creux* désigne un texte, le plus souvent binaire, qui possède un grand nombre d'occurrences d'une même lettre, celle-ci est généralement 0. Le nombre d'occurrences de toutes les autres lettres est alors sublinéaire. Dans ce cas, l'entropie est très faible et un tel texte peut très bien se compresser. Mais les méthodes de compression utilisées pour ces textes sont différentes de celles utilisées pour des textes moins atypiques : elles sont plus simples et néanmoins très efficaces.

1.1.2.3 Codages à longueur variable

Les entiers sont habituellement stockés sur un ou plusieurs octets. Cependant, en compression de données, réduire la consommation de l'espace mémoire est très important et il peut être inutile de stocker tous les entiers sur un nombre fixe d'octets. Les codages à longueur variable permettent de n'utiliser qu'un minimum d'espace mémoire. Ceux que nous allons présenter sont mêmes succincts.

Consulter des entiers stockés sur un nombre fixe d'octets est simple et rapide : on sait à quel endroit commence le codage de l'entier ainsi que sa lon-

gueur, fixe. En revanche, dans le cas de codes à longueur variable, pour décoder un nombre, il nous faut connaître l'endroit où l'encodage commence, et sa longueur. La première information n'est pas forcément stockée : dans ce cas il est indispensable de décoder à partir du début du texte, la seconde l'est forcément pour permettre le décodage.

Elias (1975) a proposé trois codages pour des entiers positifs : δ, γ et ω. Nous ne nous intéressons qu'aux deux premiers qui sont les plus utilisés. Dans la suite, nous considérerons un nombre N, x sa représentation en binaire et $\ell = |x|$.

Encodage γ L'encodage consiste à écrire en binaire $(\ell - 1)$ 0, suivis d'un 1, puis à y concaténer x.

Exemple. $N = 13$, $x = 1101$, x contient 4 bits et x est 1101, d'où : $\gamma(N) =$ 00011101

Sachant que $\ell = \lfloor \log N \rfloor + 1$, le codage $\gamma(N)$ utilise $2(\lfloor \log N \rfloor + 1)$ bits.

Encodage δ L'encodage δ consiste à produire $\gamma(\ell)$ suivi de x.

Exemple. $N = 13$, $x = 1101$, $\ell = 4$, $\gamma(\ell) = 001100$, on en déduit $\delta(N) = \gamma(\ell) \cdot 1101 = 0011001101$

Le codage $\gamma(\ell)$ occupe $2(\lfloor \log(1 + \lfloor \log N \rfloor) \rfloor + 1)$ bits soit, au total pour le codage de $\delta(N)$: $\lfloor \log N \rfloor + 1 + 2(\lfloor \log(1 + \lfloor \log N \rfloor) \rfloor + 1)$ bits. Le codage δ est plus économe que le codage γ car il utilise $\log N + o(\log N)$ bits contre $2 \log N + O(1)$ bits.

1.1.2.4 Gap-encoding

Le *gap-encoding*, ou encodage des creux, est très utilisé pour les textes creux. Considérons une séquence T définie sur un alphabet binaire $\Sigma = \{0, 1\}$, avec $|T|_1 = k = o(n)$. T peut s'écrire sous la forme $T = 0^{g_0}10^{g_1}1 \cdots 10^{g_k}$, où $g_i \geq 0$, avec $0 \leq i \leq k$ et $k + \sum_{i=0}^{k} g_i = n$.

Le principe du gap-encoding est de ne coder que les longueurs des suites de zéros, c'est-à-dire uniquement les g_i. Ces g_i sont stockés en utilisant un code à longueur variable, l'encodage δ, afin d'optimiser l'espace mémoire utilisé. On a alors comme codage : $\mathsf{ge}(T) = \delta(g_0)\delta(g_1) \cdots \delta(g_k)$.

Exemple. Soit $T = \underset{5}{\underbrace{0\,0\,0\,0\,0}}\,1\,\underset{2}{\underbrace{0\,0}}\,1\,\underset{0}{1}\,\underset{6}{\underbrace{0\,0\,0\,0\,0\,0}}\,1\,\underset{3}{\underbrace{0\,0\,0}}$.
Le codage sera alors $\mathsf{ge}(T) = \delta(5)\delta(2)\delta(0)\delta(6)\delta(3)$.

1.1.2.5 Run-length encoding

Le *run-length encoding*, ou encodage des suites, est une généralisation du gap-encoding. Ici on s'intéresse à des textes dont les lettres identiques sont fréquemment consécutives. Considérons une séquence T définie sur un alphabet binaire $\Sigma = \{0, 1\}$. T peut s'écrire sous la forme $T = 0^{g_0}1^{g_1}0^{g_2}1^{g_3} \cdots 1^{g_{k-1}}0^{g_k}$, où $g_i \geq 0$, avec $0 \leq i \leq k$ et $\sum_{i=0}^{k} g_i = n$.

Ex. 1.1 — Calcul de la transformée de Burrows-Wheeler. TBW(CTAGTTAG$) = GTT$AATCG

$$T = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}{\mathrm{CTAGTTAG\$}}$$

$T^{[0]}$	C	T	A	G	T	T	A	G	$
$T^{[1]}$	T	A	G	T	T	A	G	$	C
$T^{[2]}$	A	G	T	T	A	G	$	C	T
$T^{[3]}$	G	T	T	A	G	$	C	T	A
$T^{[4]}$	T	T	A	G	$	C	T	A	G
$T^{[5]}$	T	A	G	$	C	T	A	G	T
$T^{[6]}$	A	G	$	C	T	A	G	T	T
$T^{[7]}$	G	$	C	T	A	G	T	T	A
$T^{[8]}$	$	C	T	A	G	T	T	A	G

Permutations circulaires

	F								*L*
$T^{[8]}$	$	C	T	A	G	T	T	A	**G**
$T^{[6]}$	A	G	$	C	T	A	G	T	**T**
$T^{[2]}$	A	G	T	T	A	G	$	C	**T**
$T^{[0]}$	C	T	A	G	T	T	A	G	**$**
$T^{[7]}$	G	$	C	T	A	G	T	T	**A**
$T^{[3]}$	G	T	T	A	G	$	C	T	**A**
$T^{[5]}$	T	A	G	$	C	T	A	G	**T**
$T^{[1]}$	T	A	G	T	T	A	G	$	**C**
$T^{[4]}$	T	T	A	G	$	C	T	A	**G**

Permutations triées

Une façon de stocker une telle suite est de ne coder que T_0 ainsi que l'ensemble des g_i, avec un encodage δ. On a alors comme codage : $\mathsf{rle}(T) = T_0\delta(g_0)\delta(g_1)\cdots\delta(g_k)$. Le passage de g_i à g_{i+1} implique un changement de lettre (passage de 0 à 1 ou inversement). En connaissant la première lettre du texte, il est ainsi possible d'en déduire la totalité du texte d'origine.

Exemple. Soit $T = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\ 16\ 17\ 18\ 19\ 20\ 21\ 22\ 23}{\underbrace{0\,0\,0\,0\,0}_{5}\,\underbrace{1\,1\,1\,1\,1\,1\,1}_{6}\,\underbrace{0\,0}_{2}\,\underbrace{1\,1\,1\,1}_{4}\,\underbrace{0\,0\,0\,0}_{4}\,\underbrace{1\,1\,1}_{3}}$.
Le codage sera alors $\mathsf{rle}(T) = 0\delta(5)\delta(6)\delta(2)\delta(4)\delta(4)\delta(3)$.

1.1.2.6 Transformée de Burrows-Wheeler

La transformation de Burrows-Wheeler (1994) est un algorithme qui permet de réorganiser un texte afin d'obtenir de meilleurs taux de compression. Le texte réorganisé est appelé transformée de Burrows-Wheeler (notée TBW dans ce qui suit). À ce titre, la transformée est utilisée dans l'utilitaire `bzip2`[2] qui est l'un des meilleurs compresseurs à ce jour.

Pour obtenir la TBW du texte T (notée $\mathsf{TBW}(T)$), considérons toutes les permutations circulaires de T et trions-les par ordre alphabétique. La concaténation de la dernière lettre de chacune de ces permutations circulaires triées donne la TBW. L'ensemble des permutations circulaires triées est le plus souvent représenté sous forme d'une matrice M dont la première colonne est appelée F et la dernière, qui correspond à la TBW, L (voir Ex. 1.1).

Ensuite, la compression de la transformée est plus efficace que sur le texte lui-même car cette façon de réordonner le texte tend à rassembler les lettres identiques. Supposons qu'un texte contienne k fois le mot « chasse », il s'en suit que les k permutations circulaires commençant par « hasse » seront très

[2] `http://bzip2.org`

probablement consécutives, car triées. La dernière lettre de ces permutations sera un « c ». On obtiendra donc k « c » consécutifs dans la transformée.

Exemple. Considérons le texte $T =$ que_chaque_chasseur_sachant_chasser_-sache_chasser_sans_son_chien\$, sa transformée de Burrows-Wheeler est $\mathsf{TBW}(T) =$ neteenrrrsssshhhhha____a_uhuisssccccccheoaas\$aueen___sss_aaanqqe.

Le texte T contenant de nombreuses répétitions, nous observons que les lettres identiques ont tendance à être consécutives dans la transformée. En particulier, la lettre « c » est toujours suivi d'un « h » dans le texte, ainsi « ch » est présent à sept reprises dans le texte. C'est pourquoi les sept « c » sont consécutifs dans la transformée de Burrows-Wheeler.

Burrows et Wheeler ont également donné une méthode permettant de compresser efficacement leur transformée. Leur méthode, appelée « *move to front* », attribue d'abord un code (de 0 à $\sigma - 1$) à chaque lettre de l'alphabet. Ensuite, en parcourant la TBW, on produit le code du caractère courant, puis le code 0 est attribué à ce caractère. Les codes correspondants aux autres caractères sont décalés si nécessaire (avant cette affectation, le caractère qui possédait le code 0 possède maintenant le code 1, et ainsi de suite). Cette méthode permet d'avoir de longues suites de 0 et, plus généralement, une grande proportion de petits entiers (voir Ex. 1.2 page suivante). Un codage entropique permet ensuite de tirer parti de cette propriété.

D'autres alternatives sont possibles pour permettre une meilleure compression ce qui a donné lieu à de nombreuses recherches (Arnavut et Magliveras, 1997 ; Chapin et Tate, 1998 ; Balkenhol *et al.*, 1999 ; Deorowicz, 2001 ; Fenwick *et al.*, 2003 ; Ferragina et Manzini, 2004). Néanmoins les résultats permettent généralement un gain de quelques pourcent par rapport à la méthode originelle de Burrows et Wheeler mais pas davantage.

La compression doit être réversible afin de retrouver le texte d'origine. Cela signifie donc qu'à partir de $\mathsf{TBW}(T)$, on doit être capable de retrouver T. Pour ce faire, les propriétés suivantes sont fondamentales. Soit $c \in \Sigma$:

Propriété 1. Soit $T_j = c$, si T_j est le $i^{\text{ème}}$ c dans F alors il est aussi le $i^{\text{ème}}$ c dans L.

Exemple. Dans l'Ex. 1.1, le second G dans F et le second G dans L correspondent tout deux au G qui succède CTA, en position 3 dans T.

Propriété 2. La lettre $F[i]$ suit immédiatement la lettre $L[i]$ dans le texte d'origine, pour $n > i \geq 0$ et i tel que $L[i] \neq \$$.

Exemple. En prenant, dans l'ordre, les lignes de L et F dans l'Ex. 1.1, on a bien dans le texte T : G\$ (position 7), TA (position 5), TA (position 1), AG (position 6), AG (position 2), TT (position 4), CT (position 0) et GT (position 3).

En partant de $F[0] = \$ = T_n$, on obtient T_{n-1}, avec $L[0]$. Ensuite, en utilisant la propriété 1, on localise dans F la lettre correspondant à $L[0]$, soit i cette position. $L[i]$ donne T_{n-2}, et on poursuit le même processus jusqu'à avoir $L[i] = \$$ (voir Ex. 1.3 page 13).

Ex. 1.2 — Exécution de « *move to front* »

$\mathsf{TBW}(T) =$
neteenrrrsssshhhhha____a_uhuisssccccccheoaas$aueen___sss_aaanqqe.
$\Sigma = \{\$, \_, a, c, e, h, i, n, o, q, r, s, t, u\}$, les codes sont attribués dans l'ordre :

0	1	2	3	4	5	6	7	8	9	10	11	12	13
$	_	a	c	e	h	i	n	o	q	r	s	t	u

Parcours de la TBW :

Lettre lue	Code produit	Codes attribués
		0 1 2 3 4 5 6 7 8 9 10 11 12 13
n	7	n $ _ a c e h i o q r s t u
e	5	e n $ _ a c h i o q r s t u
t	12	t e n $ _ a c h i o q r s u
e	1	e t n $ _ a c h i o q r s u
e	0	e t n $ _ a c h i o q r s u
n	2	n e t $ _ a c h i o q r s u
r	11	r n e t $ _ a c h i o q s u
r	0	r n e t $ _ a c h i o q s u
r	0	r n e t $ _ a c h i o q s u
⋮	⋮	⋮

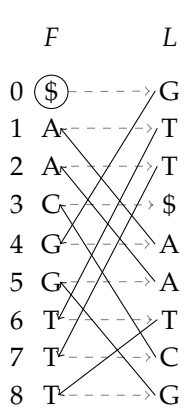
À la fin du parcours, les codes produits par l'algorithme sont : 7 5 12 1 0 2 11 0 0 12 0 0 0 9 0 0 0 0 8 8 0 0 0 1 1 13 3 0 11 5 0 0 11 0 0 0 0 0 0 4 9 12 8 0 5 12 2 8 5 0 11 10 0 0 6 0 0 1 5 0 0 3 13 0 5. Dans cette exemple la proportion de zéros est importante : 31 sur 65.

1.1.2.7 Algorithme LZ-78

Les algorithmes de la famille Lempel et Ziv sont des algorithmes de compression à base de dictionnaires adaptatifs. Le principe de ces derniers consiste à modifier le dictionnaire au fur et à mesure du parcours du texte, afin de tirer parti des répétitions rencontrées. L'algorithme de Lempel et Ziv découpe le texte, de gauche à droite, en facteurs non chevauchants. Ces facteurs sont différents les uns des autres. Ils sont également les plus courts possibles de telle façon qu'en retirant la dernière lettre d'un de ces facteurs, on perd l'unicité de ceux-ci (voir Ex. 1.4 page suivante). Les facteurs sont numérotés dans l'ordre, de gauche à droite. Par « facteur j », on se réfère au facteur dont le numéro attribué est j.

L'intérêt d'un algorithme de compression est de faire gagner de la place. Il est donc nécessaire de tirer parti des propriétés de la factorisation LZ-78 pour stocker le texte, en utilisant le moins de place possible. Pour ce faire, Ziv et Lempel (1978) proposent de stocker chaque facteur j à l'aide d'un couple (i, c), où $i < j$ désigne le numéro d'un facteur et c un caractère. Par construction, on pourra toujours trouver un facteur i permettant de coder le couple correspondant au facteur j. Dans le cas où le facteur j est de longueur 1, en re-

Ex. 1.3 — Récupération du texte d'origine à partir de la TBW



	F	L	Ordre de parcours
0	$	G	1
1	A	T	3
2	A	T	7
3	C	$	9
4	G	A	2
5	G	A	6
6	T	T	4
7	T	C	8
8	T	G	5

Texte récupéré : CTAGTTAG

Ex. 1.4 — Factorisation d'un texte pour la compression LZ-78

$$\text{Soit } T = \underbrace{\overset{0}{\text{a}}}_{0}\underbrace{\overset{1}{\text{n}}}_{1}\underbrace{\overset{2}{\text{a}}\overset{3}{\text{n}}}_{2}\underbrace{\overset{4}{\text{a}}\overset{5}{\text{s}}}_{3}\underbrace{\overset{6}{\text{b}}}_{4}\underbrace{\overset{7}{\text{a}}\overset{8}{\text{n}}\overset{9}{\text{a}}}_{5}\underbrace{\overset{10}{\text{n}}\overset{11}{\text{a}}}_{6}\underbrace{\overset{12}{\text{s}}}_{7}\underbrace{\overset{13}{\$}}_{8}$$

Tous les facteurs LZ-78 sont bien uniques. Par ailleurs, si on retire la dernière lettre du facteur 5, on obtient le facteur « an », et on perd l'unicité puisque « an » est le facteur 2

tirant la dernière lettre, il reste le mot vide ϵ. Pour cela, on suppose qu'il existe un facteur numéroté -1 qui vaut ϵ. Dans la suite, on parlera indifféremment du numéro du couple ou du numéro du facteur. Un couple sert à représenter un facteur, ces deux expressions correspondent donc à la même chose.

Le résultat de la compression est une factorisation du texte, qui produit des couples. Cette factorisation peut être représentée à l'aide d'un trie, appelé LZ-trie. Un trie est une structure arborescente dont chaque branche est étiquetée par une lettre. Chaque nœud d'un trie est donc associé à un mot et la racine au mot vide. Soit une branche du LZ-trie trie étiquetée c et qui va du nœud i au nœud j. Cette branche représente le $j+1$-ème couple, noté (i, c) (voir Ex. 1.5 page suivante).

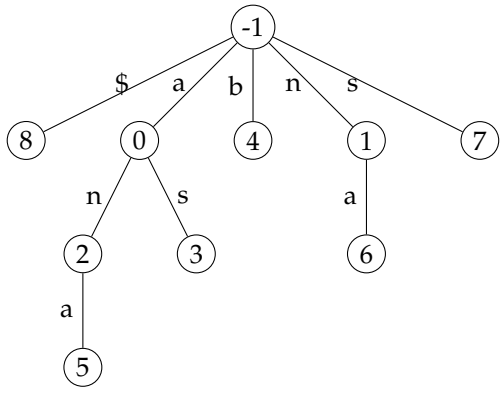
Soit u le nombre de couples obtenus après compression, d'après Bell *et al.* (1990), $u \log u = O(n \log \sigma)$.

Ex. 1.5 — Trie pour la compression LZ-78

$T =$ a n a n a s b a n a n a s \$ (indices 0 1 2 3 4 5 6 7 8 9 10 11 12 13)

Numéro du couple	Facteur	Nouveau couple
0	a= ϵ·a	(-1,a)
1	n= ϵ·n	(-1,n)
2	an=a·n	(0,n)
3	as=a·s	(0,s)
4	b= ϵ·b	(-1,b)
5	ana=an·a	(2,a)
6	na=n·a	(1,a)
7	s= ϵ·s	(-1,s)
8	\$= ϵ·\$	(-1,\$)

La compression de T produit 9 couples différents. Construisons le trie de ces 9 couples.



Le chemin de la racine au nœud 5 est « ana », donc le couple 5 (2,a) représente la chaîne « ana ».

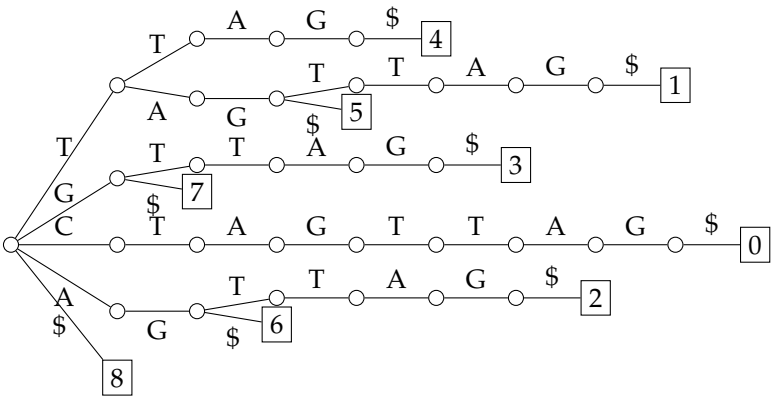
Ce trie est appelé LZ-Trie. Avec les couples, notre texte peut être factorisé comme ceci : T =a·n·an·as·b·ana·na·s·\$

1.1.3 Arbres binaires équilibrés

La notion d'arbre binaire équilibré sera utilisée à différents endroits du document. Nous introduisons maintenant le concept régissant un tel arbre. Un arbre binaire équilibré de n nœuds est un arbre pour lequel la distance de la racine à n'importe quelle feuille est en $O(\log n)$. Ceci garantit ainsi de pouvoir accéder à n'importe quel nœud de l'arbre en temps logarithmique. Des algorithmes existent (Adelson-Velskii et Landis, 1962 ; Bayer, 1972 ; Guibas et Sedgewick, 1978) pour permettre l'insertion et la suppression de nœuds dans ces arbres. De telles modifications risquent de déséquilibrer l'arbre. Les méthodes citées précédemment proposent donc des solutions pour ré-équilibrer l'arbre en temps constant. De cette façon les opérations d'insertions, suppressions et accès sont toutes réalisées en temps logarithmiques dans le pire des cas.

Ex. 1.6 — Arbre des suffixes

$$T = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}{\mathrm{CTAGTTAG\$}}$$



1.2 Structures d'indexation non compressées

Les premiers index à avoir exister indexent une partie des facteurs du texte. Le texte T ayant de l'ordre de n^2 facteurs, tous ne peuvent être directement indexés, faute d'une consommation mémoire qui serait beaucoup trop importante. À l'origine, les suffixes ont été choisis car un texte en possède un nombre linéaire et il est possible d'accéder simplement aux préfixes des suffixes (voir l'arbre des suffixes, section 1.2.1 ou la table des suffixes section 1.2.2 page 18) indexés ce qui permet d'avoir tous les facteurs du texte. Cependant, malgré ces choix, chacune de ces structures d'indexation a une complexité en espace en $O(n \log n)$, ce qui empêche de les construire sur des textes relativement grands.

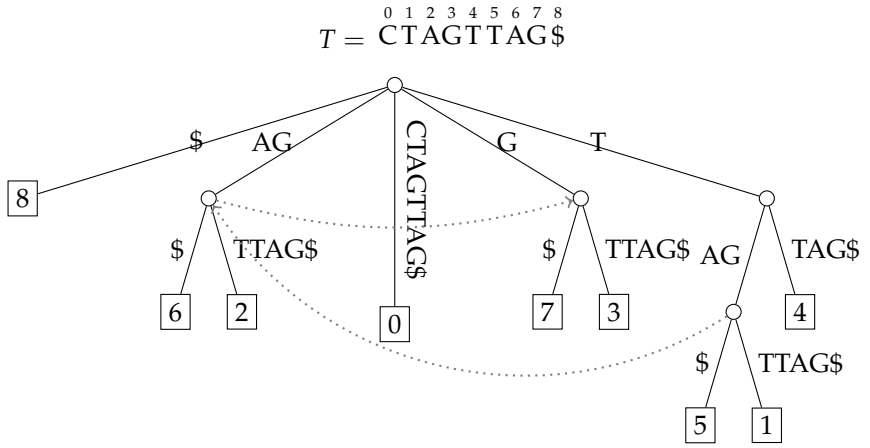
1.2.1 Arbre des suffixes

L'arbre des suffixes est un arbre dont chaque branche est étiquetée par une lettre et chaque chemin de la racine de l'arbre à une feuille correspond à un suffixe du texte T. Le texte T possède un terminateur. Ceci garantit que tous les suffixes se terminent effectivement sur une feuille de l'arbre. Sachant que chaque feuille correspond à un seul suffixe, la position de départ du suffixe qu'elle représente est stockée dans son étiquette. Deux étiquettes de branches sortant d'un même nœud ne peuvent commencer par la même lettre (voir Ex. 1.6).

1.2.1.1 Arbre compact des suffixes

L'arbre compact des suffixes a été introduit par Weiner (1973). L'objectif de l'arbre compact des suffixes est de réduire l'espace mémoire utilisé par l'arbre des suffixes en retirant les nœuds qui ne possèdent qu'un seul descendant. Ainsi, les étiquettes des branches sont maintenant des chaînes de caractères.

Ex. 1.7 — Arbre compact des suffixes



La concaténation de toutes les étiquettes d'une branche allant de la racine à une feuille est égale au suffixe représenté par la feuille (voir Ex. 1.7, ignorer les flèches en pointillés pour l'instant).

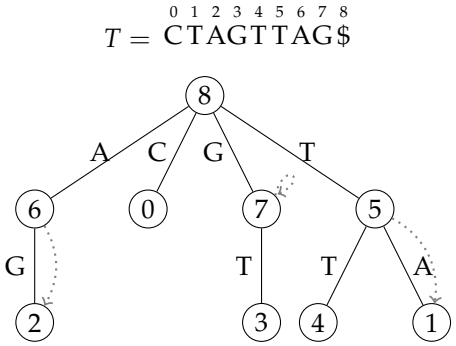
Par la suite McCreight (1976) puis Ukkonen (1992) ont amélioré l'algorithme de construction de l'arbre compact des suffixes. L'algorithme d'Ukkonen, en utilisant les *liens suffixes* introduits par Mc Creight, est incrémental. Les liens suffixes sont des liens qui vont du nœud représentant le facteur a·u, avec $a \in \Sigma$ et $u \in \Sigma^*$, au nœud représentant le facteur u, si il existe. Dans l'exemple 1.7, les liens suffixes sont représentés par des flèches en pointillés. On omet généralement les flèches dont le nœud destination est la racine. Ces liens suffixes permettent, entre autres, d'identifier des éléments communs dans l'arbre compact des suffixes (p. ex. dans l'Ex. 1.7 les liens suffixes permettent d'identifier trois sous-arbres possèdant des branches avec les mêmes étiquettes). Ces liens suffixes permettent à la fois de faciliter la construction de l'arbre compact des suffixes et de réduire l'espace mémoire utilisé par la structure. Dans la suite nous utiliserons « arbre des suffixes » pour désigner l'arbre compact des suffixes.

L'arbre des suffixes permet de rechercher un motif P de longueur m en temps $O(m + |T|_P)$. En partant de la racine, il faut descendre dans l'arbre en fonction des lettres lues dans le motif. Si le préfixe lu dans P ne correspond à aucune branche, P n'a pas d'occurrence dans le texte indexé, sinon les occurrences sont représentées par les feuilles contenues dans le sous-arbre atteint.

L'arbre compact des suffixes sert également à résoudre de nombreux autres problèmes en algorithmique du texte (voir Gusfield (1997)).

En raison de sa structure arborescente et des nombreuses informations à stocker (étiquettes des feuilles et des branches), l'arbre compact des suffixes est consommateur d'espace mémoire. Kurtz (1999), en utilisant des propriétés de l'arbre des suffixes, dont les liens suffixes, a réduit l'espace nécessaire à cette structure. Ainsi, avec son implantation l'arbre des suffixes nécessite $10n$ octets en moyenne et jusqu'à $20n$ dans le pire des cas. Malgré cette amélio-

Ex. 1.8 — Arbre contracté des suffixes



ration importante, l'indexation d'un génome complet, tel que celui de *H. sapiens* qui fait 3 Gbp.[3] à l'aide de l'arbre compact des suffixes devrait prendre environ 45 Go de mémoire. Les ordinateurs de bureau actuels possèdent, au mieux, dix fois moins de mémoire centrale. On comprend alors pourquoi l'utilisation de structures bien plus économes en espace devient nécessaire.

Par ailleurs, on parle d'arbre *généralisé* des suffixes lorsqu'un arbre compact des suffixes indexe plusieurs textes. Soit $T^0, \ldots, T^{j-1}$ des textes. L'arbre généralisé des suffixes sur cet ensemble de textes sera équivalent à l'arbre compact des suffixes de $T^0\$T^1\$\cdots T^{j-1}\$$.

1.2.1.2 Arbre contracté des suffixes

Ehrenfeucht *et al.* (2009) ont introduit l'arbre contracté des suffixes. Au contraire de l'arbre des suffixes, cette structure arborescente ne stocke pas la totalité des suffixes mais seulement un préfixe de chaque suffixe. À ce titre, cette méthode peut être vue comme similaire au LZ-78, dans le sens où on cherche à insérer les plus courts facteurs à ne pas être encore présents dans l'arbre. Cependant il ne s'agit pas d'une méthode de compression : le nombre de nœuds de l'arbre est $n + 1$.

Les suffixes, sauf \$, sont considérés par longueurs croissantes. Pour chaque suffixe, on insère dans l'arbre le plus court préfixe qui n'est pas encore présent. Cette insertion produit dans l'arbre un nouveau nœud dont le numéro correspond à la position de début du suffixe. Ainsi, pour chaque suffixe, un nœud lui est attribué. Afin de permettre une recherche en temps optimal, chaque nœud i ayant un ou plusieurs descendants posséde un lien vers le nœud représentant le plus long préfixe de $T[i\,..]$ présent dans le trie (voir Ex. 1.8).

Cette structure permet un temps de recherche optimal en $O(m + |T|_P)$ et une construction en temps linéaire mais les auteurs ne donnent pas d'information quant à la consommation mémoire. Cette structure supporte les mises à jour comme nous le verrons en section 3.1.4 page 60.

[3] Gbp. : milliards de paires de bases, désigne le nombre de nucléotides (les lettres) dans une séquence génomique.

Ex. 1.9 — Table des suffixes

$$T = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}{\mathrm{CTAGTTAG\$}}$$

TS	8	6	2	0	7	3	5	1	4
	$	AG$	AGTTAG$	CTAGTTAG$	G$	GTTAG$	TAG$	TAGTTAG$	TTAG$

1.2.2 Table des suffixes

Nous avons vu que l'arbre compact des suffixes est trop consommateur d'espace mémoire pour indexer de très longs textes connus, les séquences génomiques de mammifères, par exemple. La table des suffixes a été créée dans le but d'avoir une structure d'indexation plus économe en espace.

1.2.2.1 Présentation

Manber et Myers (1990) d'une part et Gonnet *et al.* (1992) d'autre part ont présenté indépendamment la table des suffixes. La table des suffixes (notée TS) peut être obtenue à partir du tri lexicographique de l'ensemble des suffixes. La table ne stocke pas les suffixes triés mais leurs positions respectives, les suffixes correspondants peuvent ensuite être retrouvés à partir du texte (voir Ex. 1.9). On remarquera que ce tableau de positions correspond au tableau de feuilles obtenu après un parcours préfixe de l'arbre compact des suffixes. L'algorithme de recherche pour cette table est ensuite très simple : sachant que les suffixes sont triés, il suffit de réaliser une recherche par dichotomie sur la table. La recherche par dichotomie se fait au plus en $\log n$ étapes, et à chaque étape on a, dans le pire des cas, m comparaisons. Ce qui donne la complexité finale pour trouver toutes les occurrences d'un motif P : $O(m \log n + |T|_P)$. La table étant uniquement constituée de n entiers, elle peut être stockée sur $n \log n$ bits, soit $4n$ octets si $n < 2^{32}$.

En plus de la table des suffixes, certains algorithmes peuvent considérer la table inverse des suffixes (TIS). Cette table inverse correspond à la permutation inverse de la table des suffixes. Elle permet de connaître le rang lexicographique d'un suffixe à partir de sa position de départ dans le texte. C'est-à-dire que $\mathsf{TIS}[i]$ est le rang lexicographique du suffixe $T[i\,..]$ parmi l'ensemble des suffixes.

1.2.2.2 Algorithmes de construction

Le premier algorithme construisait la table des suffixes en temps $O(n \log n)$, il utilisait la technique de doublement de préfixe (*prefix-doubling*) : les suffixes sont d'abord triés en fonction de leur deux premières lettres puis, en utili-

sant ces résultats on peut les trier en fonction de leur quatre premières lettres, puis les huit premières et ainsi de suite. Cette méthode demande au plus $\log n$ étapes afin que tous les suffixes soient triés et chaque tri est effectué en temps linéaire, d'où la complexité finale.

Les premiers algorithmes linéaires en temps sont apparus simultanément en 2003 (Kärkkäinen et Sanders ; Ko et Aluru ; Kim *et al.*). Ces trois algorithmes sont récursifs et divisent les suffixes en différentes classes (deux ou trois selon les algorithmes). Le tri réalisé sur une classe à un niveau de récursion permet d'induire le tri pour les autres classes puis au niveau de récursion précédent. En pratique, le fait que ces algorithmes soient récursifs les ralentit de façon significative par rapport à des algorithmes ayant de moins bonnes complexités théoriques (voir Puglisi *et al.* (2007) pour les comparaisons entre les différents algorithmes).

D'autres approches (Manzini et Ferragina, 2004 ; Maniscalco et Puglisi, 2006) ont une complexité dans le pire des cas de $O(n^2 \log n)$, ce qui peut sembler rédhibitoire. Malgré tout, d'après les tests menés par Puglisi *et al.* sur un large panel de textes et d'algorithmes de construction, ce sont ces derniers algorithmes qui, dans la pratique, sont les plus rapides et les plus économes en mémoire.

Les algorithmes de Manzini et Ferragina ; Maniscalco et Puglisi utilisent, sur une partie de l'entrée, un algorithme de tri de chaînes de caractères (un quicksort amélioré), ce qui explique la complexité théorique dans le pire des cas.

1.2.2.3 Table PLPC

Nous avons vu que l'algorithme de recherche utilisé pour la table des suffixes est très simple mais n'est cependant pas optimal, contrairement à celui utilisé pour l'arbre des suffixes.

Une table additionnelle, la table PLPC(pour « plus long préfixe commun »), permet d'atteindre cette borne en $O(m + |T|_P)$. Cette table stocke la longueur du plus long préfixe commun entre deux suffixes consécutifs de TS (voir Ex. 1.10 page suivante). L'utilisation de ces informations lors de la phase de recherche évite des comparaisons redondantes et permet donc, d'atteindre la borne optimale.

1.2.2.4 Résultats annexes

Arbre des suffixes virtuels Un résultat de Abouelhoda *et al.* (2004) montre qu'en utilisant la table PLPC, la table des suffixes et, éventuellement, d'autres tables (pour les liens suffixes par exemple) il est possible de remplacer avantageusement l'arbre des suffixes par cette collection de tables. En suivant l'implantation des auteurs, on arrive à un ensemble de tables pour un maximum de $10n$ octets.

Table désordonnée des suffixes Franceschini et Grossi (2008) ont montré qu'il est possible d'avoir un temps de recherche optimal sans réellement stocker la table PLPC. Leur méthode consiste à ne pas laisser la table des suffixes triée mais à la mettre dans un « désordre organisé ». Le fait de désordonner la table des suffixes permet en fait de stocker des informations supplémen-

Ex. 1.10 — Table des suffixes et table PLPC

$$T = \overset{0\,1\,2\,3\,4\,5\,6\,7\,8}{\mathrm{CTAGTTAG\$}}$$

TS	8	6	2	0	7	3	5	1	4
PLPC	0	2	0	0	1	0	3	1	
	$	AG$	AGTTAG$	CTAGTTAG$	G$	GTTAG$	TAG$	TAGTTAG$	TTAG$

taires. Si deux entrées ne sont pas rangées dans le bon ordre, c'est-à-dire que la première est supérieure à la seconde alors que cela devrait être l'inverse, cela permet de stocker un bit 1, un bit 0 sinon. L'espace de stockage ainsi dégagé peut alors être utilisé pour enregistrer des informations pertinentes de la table PLPC. Cette méthode permet d'avoir un temps de recherche optimal sans utiliser plus d'espace mémoire que pour la table des suffixes seule.

1.2.3 Vecteur des suffixes

Le vecteur des suffixes est une structure d'indexation introduite par Monostori *et al.* (2001). Cette structure est équivalente à l'arbre compact des suffixes mais permet d'être stockée en utilisant un espace mémoire moindre. Elle a ensuite été améliorée par Prieur et Lecroq (2008). Leur structure nécessite environ $15n$ octets pour les séquences génomiques mais deux fois moins pour des textes sur des alphabets plus grands (ouvrages, codes sources).

1.3 Structures d'indexation compressées

Nous avons vu dans la section 1.2 page 15 que les structures d'indexation utilisent au mieux $4n$ octets, sans compter le texte. Cette consommation mémoire est rédhibitoire, par exemple, pour l'indexation du génome de référence de *H. sapiens*, d'une banque de séquences génomiques, ou encore d'une collection d'articles de l'encyclopédie collaborative en ligne, Wikipedia.

Les structures d'indexation compressées existant à ce jour peuvent principalement être divisées en deux familles : les index utilisant la compression LZ-78 et les index utilisant un échantillonnage de la table des suffixes. La première famille est la première à être apparue et comporte trois structures d'indexations distinctes. Dans la seconde famille, de nombreuses possibilités existent quant à la façon d'utiliser l'échantillonnage de la table des suffixes, et la manière de récupérer les valeurs non échantillonnées.

1.3.1 LZ-Index

Kärkkäinen et Ukkonen (1996) semblent être les premiers à avoir réalisé une structure d'indexation compressée. Elle est basée sur la factorisation LZ-78 (voir section 1.1.2.7 page 12). Navarro (2002) a ensuite amélioré cette structure d'indexation puis Arroyuelo *et al.* (2006) ; Arroyuelo et Navarro (2007b) ont également contribué à restreindre le temps de recherche et l'espace mémoire de cette structure d'indexation. Russo et Oliveira (2006) ont également présenté une méthode d'indexation basée sur la factorisation LZ-78. Plus tard, ils l'utilisent non plus pour la recherche exacte d'un motif dans un texte pour faire de la recherche approchée de motifs, autorisant des erreurs (Russo *et al.*, 2007). Nous présentons ici une version simplifiée de celle introduite dans Arroyuelo *et al.* (2006).

1.3.1.1 Structures de données

Le seul LZ-trie (cf. Ex. 1.5 page 14) ne suffit pas pour rechercher efficacement des motifs. Avec cet arbre l'accès aux préfixes des facteurs indexés est simple. Néanmoins ce n'est pas suffisant puisque cela ne permet pas un accès à tous les facteurs du texte. Par exemple, il n'est pas possible de connaître simplement la lettre qui suivra le facteur i, pour n'importe quel i.

Un nœud i du trie correspond au facteur i. Le facteur i peut être retrouvé en concaténant les étiquettes de la racine du trie jusqu'au nœud i. En effectuant un parcours préfixe de gauche à droite du trie, on obtient tous les facteurs rangés de façon croissante lexicographiquement (il s'agit d'une propriété des tries). Dans chaque nœud, on enregistre la valeur du plus petit et du plus grand rang lexicographique des facteurs associés aux nœuds du sous-arbre dont il est la racine. Le plus petit rang lexicographique est le rang du facteur associé à la racine : il s'agit du plus petit facteur de son sous-arbre ; le plus grand correspond au rang du facteur du nœud le plus à droite (voir Ex. 1.11 page suivante).

À l'aide de ces rangs, il est possible de savoir en temps constant si un nœud x est dans le sous-arbre dont la racine est un nœud y. x est dans le sous-arbre de y si et seulement si $rang\_min(y) \leq rang\_min(x) \leq rang\_max(y)$. Si un nœud x est dans le sous-arbre du nœud y cela signifie que le facteur associé au nœud y (noté F_y) est un préfixe de F_x.

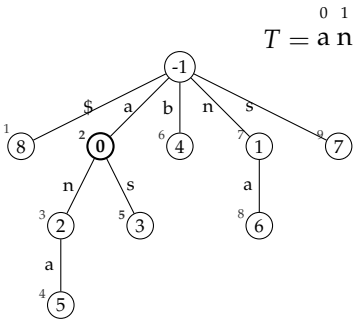
En plus du LZ-trie, nous avons un trie contenant le miroir de tous les facteurs. Ce trie, appelé Rev-trie, permet d'avoir accès aux suffixes de tous les facteurs (cf. Ex. 1.12 page suivante). Enfin, deux structures, Node et RNode, permettent d'accéder directement à un nœud i dans le LZ-trie et dans le Rev-trie, respectivement.

1.3.1.2 Recherche de motifs

La recherche de motifs distingue différents cas indépendants les uns des autres. Une occurrence d'un motif n'est pas forcément préfixe d'un facteur indexé et n'est pas nécessairement contenue dans un seul facteur. Diverses possibilités sont envisageables (cf. FIG. 1.1 page 23). L'algorithme de recherche de motif distingue trois cas :

1. occurrences contenues dans un seul facteur,

Ex. 1.11 — Rangs dans le LZ-trie. Les nombres au dessus des nœuds correspondent aux rangs lexicographiques des facteurs associés.



Nœud	**0**	1	2	3	4	5	6	7	8
rang_min	**2**	7	3	5	6	4	8	9	1
rang_max	**5**	8	4	6	6	4	8	9	1

Le nœud 0 a pour rang 2 : le facteur 0 est « a » et il s'agit du plus petit facteur dans l'ordre lexicographique, après « $ », d'où son rang. Dans le sous-arbre dont la racine est le nœud 0, le plus petit facteur est « a », d'où $rang\_min(0) = 2$.
Dans ce même sous-arbre, le plus grand rang est 5 (nœud 3), d'où $rang\_max(0) = 5$.

Ex. 1.12 — Rev-trie de « ananasbananas$ »

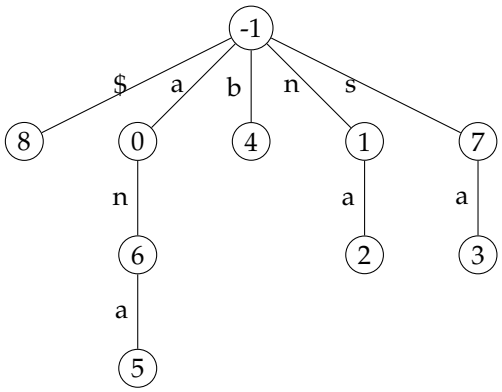


Fig. 1.1 — Distinction des trois cas à explorer pour la recherche de motifs. En gris sont représentées les différentes localisation envisageables d'un motif dans le texte.

2. occurrences à cheval sur deux facteurs consécutifs,
3. occurrences sur au moins trois facteurs consécutifs.

Occurrences dans un seul facteur Si une occurrence du motif est contenue dans un seul facteur indexé, le motif P est alors suffixe d'un autre facteur indexé. Supposons que le motif soit contenu dans un facteur F_i sans être suffixe de celui-ci. Chaque facteur F_i est associé a un couple $(num_i, lettre_i)$. Ce facteur a été construit à partir d'un facteur précédent F_j correspondant au couple $(num_j, lettre_j)$ avec $num_j < num_i$. Ce facteur F_j contient également P, car $F_i = F_j \cdot lettre_i$ mais P n'est pas suffixe de F_i. En remontant ainsi récursivement aux facteurs à partir desquels F_j a été créé, on arrivera à un facteur F_s, $s \leq j$, dont P est suffixe.

La première étape consiste à trouver ce F_s. Nous savons que le Rev-trie indexe le miroir des facteurs. Vérifier si P est suffixe d'un facteur revient donc à vérifier si il existe un chemin partant de la racine étiqueté par $\overline{P}$, miroir de P. Si tel est le cas, le nœud d'arrivée est s. Nous savons maintenant que P est suffixe de F_s. Si s a des descendants dans le LZ-trie cela signifie que P a également une occurrence dans les facteurs correspondant à ces nœuds. Lors de la seconde étape nous récupérons donc tous ces descendants. Chacun de ces nœuds correspond à une occurrence de P. Il faut ensuite recommencer le procédé sur tous les descendants de s mais dans le Rev-trie, cette fois (voir Ex. 1.13 page suivante).

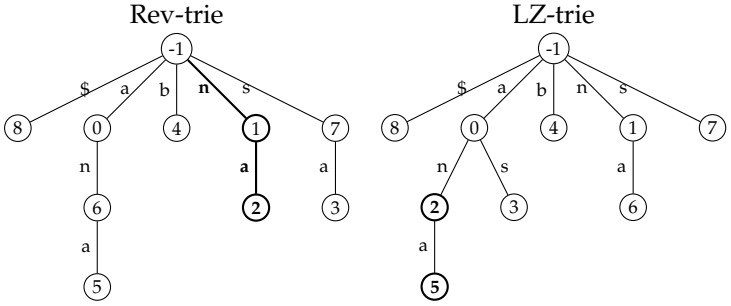
Occurrences à cheval sur deux facteurs Dans la situation où une occurrence du motif est sur deux facteurs consécutifs, un préfixe du motif est suffixe d'un facteur indexé F_i et le suffixe restant du motif est préfixe du facteur indexé suivant F_{i+1}. La méthode de recherche se base sur cette remarque. Nous vérifions, pour chaque préfixe du motif, s'il est suffixe d'un facteur indexé et si le suffixe restant du motif commence le facteur suivant.

Pour vérifier si le préfixe d'un motif est suffixe d'un facteur indexé, nous utilisons le même procédé que précédemment : il faut vérifier si le miroir du préfixe est présent dans le Rev-trie. Soit x le nœud atteint dans le Rev-trie. Ensuite, nous vérifions si le suffixe du motif est présent dans le LZ-trie et notons y le nœud atteint.

Sans perte de généralité, considérons que le nombre de descendants directs et indirects de x est inférieur à celui de y. Pour chaque descendant x' de x, nous vérifions si le nœud ayant le numéro suivant $x' + 1$ est un descendant

Ex. 1.13 — Recherche de « an » contenu dans un seul facteur

Considérons « na », le miroir de « an ».



– Dans le RevTrie, « na » nous amène au nœud 2.
– Considérons le LZTrie, le nœud 2 a un fils : le nœud 5
Les facteurs 2 et 5 possèdent par conséquent tous deux une occurrence du motif « an ». Le motif est suffixe de F_2 et facteur de F_5.

---

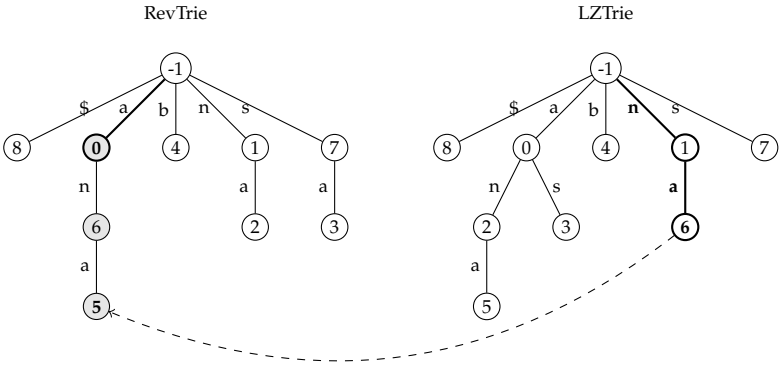
de y. Pour ce faire, nous utilisons la structure Node qui permet d'accéder directement à un nœud. Nous récupérer alors le rang de ce nœud pour vérifier s'il est compris entre les rangs minimaux et maximaux de y. Il faut réaliser ce procédé pour tous les préfixes et suffixes du motif (voir Ex. 1.14 page ci-contre).

Occurrences sur au moins trois facteurs Avoir une occurrence qui s'étend sur au moins trois facteurs indexés signifie qu'un facteur du motif correspond à un facteur indexé entier. Dans un premier temps, nous allons chercher tous les facteurs indexés qui sont facteurs du motif. Les numéros des facteurs sont stockés dans une table A de taille $m \times m$, $A[i,j] = b \iff P[i\,..\,j] = F_b$. Or, plusieurs facteurs consécutifs peuvent être facteurs du motif. Nous allons chercher à étendre les facteurs indexés sur la droite pour les faire correspondre à un facteur plus grand du motif. Ceci revient à vérifier, pour $A[i,j] = b$, que $A[j+1,k] = b+1$. Si tel est le cas, nous avons alors $F_b \cdot F_{b+1}$ qui est un facteur de P. Il faut continuer cette recherche jusqu'à ne plus pouvoir étendre notre facteur davantage. Soit $F_b \cdots F_{b+k}$ une concaténation de facteurs consécutifs correspondant à un facteur $P[i\,..\,j]$ du motif. Il nous reste ensuite à vérifier que le préfixe du motif $P[..\,i-1]$ est un suffixe de F_{b-1}, s'il existe ; et que le suffixe du motif $P[j+1\,..]$ est un préfixe de F_{b+k+1}, s'il existe.

Vérifions si le suffixe $P[j+1\,..\,m-1]$ est préfixe de F_{b+k+1}. Par construction, si un mot est préfixe propre d'un facteur indexé, cela signifie qu'un autre facteur contient exactement le mot. Regardons alors si $P[j+1\,..\,m-1]$ est dans un facteur. Pour cela, nous pouvons utiliser la table A qui contient tous les facteurs indexés correspondant à un facteur du motif. Si $A[j+1,m] = \ell$, F_ℓ contient le suffixe démarrant à la position $j+1$. Il reste à savoir si F_ℓ est un pré-

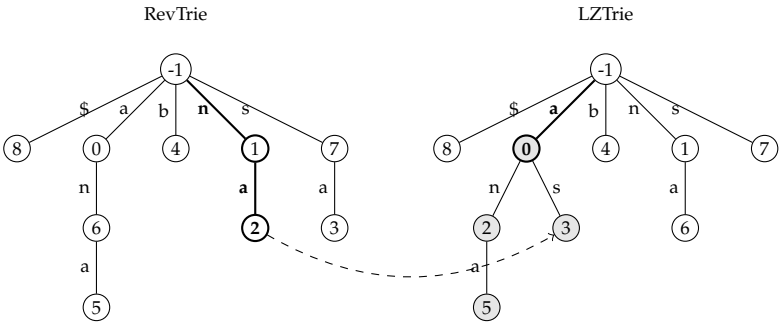
Ex. 1.14 — Recherche de *« ana »* à cheval sur deux facteurs

1. Considérons le préfixe « a » et le suffixe « na ».



   Le facteur **6** est un facteur *candidat*, pour contenir la fin d'une occurrence (nœud 6 dans le LZ-trie), le facteur précédent (le facteur **5**) est un facteur candidat pour contenir le début d'une occurrence. Or, le nœud 5 est bien un descendant du nœud atteint dans le Rev-trie.
   On a donc une occurrence de « ana » qui commence dans F_5 et se termine dans F_6.

2. Considérons le préfixe « an » (dont le miroir est « na ») et le suffixe « a ».



   En suivant le même procédé, on trouve qu'une occurrence de « ana » commence à la fin de F_2 et se termine au début de F_3.

fixe de F_{b+k+1}. Pour cela nous pouvons utiliser les rangs qui nous permettent de savoir si le nœud $b+k+1$ est un descendant de ℓ.

Enfin, il nous faut contrôler que $P[..i-1]$ est suffixe de F_{b-1}. Pour ce faire, nous partons du nœud $b-1$ dans le LZ-trie et remontons vers la racine en vérifiant que les étiquettes des branches correspondent à $\overline{P[..i-1]}$. Si c'est le cas, nous avons alors une occurrence du motif P qui commence dans F_{b-1}, à $i-1$ positions de la fin et qui se termine dans F_{b+k+1} à $m-j-1$ positions du début du facteur (voir Ex. 1.15 page suivante).

1.3.1.3 Complexités

L'espace nécessaire au stockage du LZ-index est de $(2+\mu)nH_k(T)+o(n\log\sigma)$ bits, avec $k=o(\log_\sigma n)$ et avec $\mu>0$. μ correspond à un compromis espace-temps dans une structure de données utilisée par le LZ-index. La recherche d'un motif, en utilisant une telle structure, se fait dans le pire des cas en temps $O(m^2\log m+(m+occ)\log n)$. La complexité du temps de recherche n'est pas surprenante dans le sens où trois cas sont à explorer et ceux-ci nécessitent de considérer tous les préfixes, suffixes et facteurs du motif. Les informations indexées sont le début et la fin des facteurs. Le manque d'information sur les lettres qui suivent immédiatement un facteur donné ou sur les lettres se trouvant au milieu d'un facteur ne permet pas d'avoir une recherche efficace. Dans ces conditions, on doit considérer tous les facteurs du motif pour ne pas oublier d'occurrence.

1.3.2 Table des suffixes

La principale motivation pour obtenir des tables compressées des suffixes tient dans une remarque simple. Différentes chaînes de caractères de même longueur construites sur le même alphabet peuvent avoir la même table des suffixes (voir les travaux de Schürmann et Stoye (2008)) et une chaîne n'a qu'une table des suffixes. Le nombre de chaînes de caractères de longueur n sur un alphabet Σ est donc plus grand que le nombre de table de suffixes différentes issues de ces chaînes. On devrait donc pouvoir stocker les tables des suffixes en utilisant moins d'espace que pour les chaînes de caractères elle-même.

Jusqu'à maintenant toutes les solutions de compression de la table des suffixes se basent sur un échantillonnage. Il faut donc être capable de recalculer les valeurs non échantillonnées. Elles sont retrouvées en partant d'une valeur échantillonnée et en calculant, de proche en proche, les valeurs manquantes. Plusieurs approches existent, toutes introduites en 2000, et seront détaillées dans la suite. On peut d'ors et déjà remarquer que cette méthode d'échantillonnage empêche de récupérer n'importe quelle valeur non échantillonnée en temps constant. Au contraire, ce temps dépendra de la distance d'échantillonnage. C'est la raison pour laquelle aucune solution, basée sur la table compressée des suffixes, n'est optimale pour donner les positions de toutes les occurrences d'un motif.

Ex. 1.15 — Recherche de *« nanas »* dans le texte

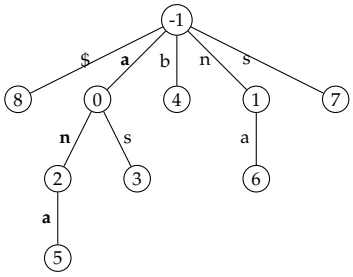
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$T =$	a	n	a	n	a	s	b	a	n	a	n	a	s	$
	0	1	2		3		4	5			6		7	8

$$P = \overset{0\ 1\ 2\ 3\ 4}{\mathrm{n\ a\ n\ a\ s}}$$

Calcul du tableau A

A	0	1	2	**3**	4
0	1	6			
1		0	2	**5**	
2			1	6	
3				0	3
4					7

$P[\mathbf{1}\,..\,\mathbf{3}] = F_5 = \mathbf{ana}$



Extension des facteurs

$A[0,0] = 1$ peut être étendu avec $A[1,2] = 2$ et $A[3,4] = 3$. On sait ainsi que les facteurs 1, 2 et 3 correspondent au facteur $P[0 \cdots 4] = P$. On a alors une occurrence commençant dans F_1 jusqu'à F_3.
L'extension des facteurs nous donne également : $A[1,1]$ et $A[2,2]$; $A[2,3]$ et $A[4,4]$. Notons que si on a $A[i,k] = 0$, avec $i > 0$, on ne peut avoir d'occurrence, il faudrait avoir un facteur avant le facteur 0 et dont $P[0\,..\,i-1]$ est suffixe, ce qui est impossible puisque le facteur -1 est le mot vide. Ceci nous permet donc d'ignorer les candidats $P[1\,..\,2] = F_0F_1$ et $P[3\,..\,3] = F_0$. N'oublions pas que les facteurs non étendus sont tout de même des candidats pour faire partie d'une occurrence. Il nous reste donc trois candidats à une occurrence : $P[0\,..\,1] = F_6$, $P[1\,..\,3] = F_5$ et $P[2\,..\,4] = F_6F_7$.

Test du suffixe

$P[0\,..\,1] = F_6$ **:** $A[2,4]$ n'est pas défini, cela signifie que $P[2\,..\,4] = \mathrm{nas}$ n'est facteur d'aucun $F_i \rightarrow$ pas d'occurrence.

$P[1\,..\,3] = F_5$ **:** le suffixe qui correspond est $A[4,4] = 7$. Il faut tester si le facteur suivant F_5, F_6 donc, a pour préfixe F_7. C'est impossible F_6 précède $F_7 \rightarrow$ pas d'occurrence.

$P[2\,..\,4] = F_6F_7$ **:** rien à tester, déjà suffixe.

Test du préfixe

Sur les candidats restants, il nous reste à vérifier que le préfixe correspond.
$\overline{P[2\,..\,4]} = F_6F_7$: Remontée du nœud 5 dans le LZ-trie, on doit lire $\overline{P[0\,..\,1]} = \overline{\mathrm{na}}$, on lit « an » $\rightarrow$ occurrence.

1.3.2.1 FM-Index

Présentation Sadakane et Imai (1998) ont été les premiers à montrer la proximité entre la table des suffixes et la transformée de Burrows-Wheeler (voir section 1.1.2.6 page 10). Dans leur article, la transformée de Burrows-Wheeler est utilisée pour compresser à la fois le texte et la table des suffixes. Grâce à cette proximité, la table des suffixes peut ensuite être recalculée simplement, lors de la décompression de la transformée. Le FM-Index de Ferragina et Manzini (2000, 2005) pousse cette idée plus loin et utilise la transformée de Burrows-Wheeler pour remplacer, autant que possible, la table des suffixes. Ceci leur permet de se passer du texte. La transformée de Burrows-Wheeler peut être compressée efficacement ce qui leur permet d'aboutir à un auto-index.

Recherche de motif Nous avons préalablement vu que la propriété 1 page 11 est fondamentale pour réussir à retrouver le texte T à partir de $\mathsf{TBW}(T)$. Ferragina et Manzini montrent que cette propriété peut également servir pour rechercher un motif dans un texte. Les auteurs utilisent cette propriété pour définir une fonction notée LFet définie comme suit. Nous avons $\mathsf{LF}(i) = j$ si et seulement si $\mathsf{TS}[i] = \mathsf{TS}[j] + 1$ ou $\mathsf{TS}[j] = n - 1$, si $\mathsf{TS}[i] = 0$, avec $0 \leq i \leq n - 1$. Autrement dit si la position i, dans la TBW, correspond à la permutation circulaire $T^{[k]}$; la position j correspond alors à la permutation circulaire précédente $T^{[k-1]}$ (voir Ex. 1.16 page ci-contre).

Prenons un motif P et deux positions permettant de délimiter un intervalle : $\mathsf{di} \leftarrow 0$ et $\mathsf{fi} \leftarrow n$. Une boucle va consister à parcourir le motif P de droite à gauche et l'intervalle correspondra à l'ensemble des occurrences pour le suffixe du motif lu jusqu'alors. Le motif est lu de droite à gauche car la fonction LF permet de passer d'une permutation circulaire à la précédente et non l'inverse.

Nous commençons avec P_{m-1} et calculons le résultat de la fonction LF pour la première et la dernière permutation circulaire se terminant par cette lettre dans notre intervalle. Avec les résultats obtenus, nous mettons à jour les bornes de l'intervalle. Par définition de la fonction LF, l'intervalle contient désormais toutes, et uniquement, les permutations circulaires commençant par P_{m-1}. Nous renouvelons le procédé pour P_{m-2}. On obtient alors un nouvel intervalle, qui correspond à toutes les permutations circulaires commençant par $P_{m-2}P_{m-1}$. La recherche se termine lorsque l'intervalle obtenu est vide (pas d'occurrence) ou quand toutes les lettres du motif ont été considérées (la taille de l'intervalle donne le nombre d'occurrences) (voir Ex. 1.17 page 30).

Calcul de la fonction LF Intéressons-nous plus particulièrement à la manière dont est calculée la fonction LF. Pour une position i, dans L, $\mathsf{LF}(i)$ donne la position de la lettre correspondante dans F. Soit l'opération $\mathsf{rank}_c(L, i)$ qui donne le nombre d'occurrences de c dans $L[0\,..\,i]$, pour tout $i < |L|$, si $i < 0$ alors rank retourne 0. On suppose que cette opération est réalisée en temps constant, nous verrons comment dans la section 2.2 page 49. Cette opération rank nous permet de savoir que la lettre $L[i]$ est la $j^{\text{ème}}$ dans L. Pour compléter la fonction LF il nous reste à savoir la position du $j^{\text{ème}}$ c dans F. Par définition F est triée, si on connaît la première position de chaque lettre distincte dans F, on en déduit facilement la position du $j^{\text{ème}}$ c. On appelle $\mathsf{First}(c)$ la position de la première occurrence de c dans F. Pour l'exemple 1.17 page 30, First

Ex. 1.16 — Calcul de la fonction LF

$$T = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}{\mathrm{CTAGTTAG\$}}$$

	TS
0	8
1	6
2	2
3	0
4	7
5	3
6	5
7	1
8	4

Nous avons $\mathsf{TS}[0] = 8$, $\mathsf{LF}(0)$ correspond à la position du 7 dans TS. Ainsi $\mathsf{LF}(0) = 4$
Ensuite, $\mathsf{TS}[1] = 6$, $\mathsf{LF}(1)$ correspond à la position du 5 dans TS. Ainsi $\mathsf{LF}(1) = 6$
Ensuite, $\mathsf{TS}[2] = 2$, $\mathsf{LF}(2)$ correspond à la position du 1 dans TS. Ainsi $\mathsf{LF}(2) = 7$.
Ainsi de suite…

On peut utiliser une autre représentation, équivalente, pour le calcul de LF, c'est celle qui est utilisée en pratique :

	F	L
0	\$	G
1	A	T
2	A	T
3	C	\$
4	G	A
5	G	A
6	T	T
7	T	C
8	T	G

$L[0]$ = G, il s'agit du premier G dans L. Recherchons la position du premier G dans F, il est en position 4. Ainsi $\mathsf{LF}(0) = 4$.
$L[1]$ = T, il s'agit du premier T dans L. Recherchons la position du premier T dans F, il est en position 6. Ainsi $\mathsf{LF}(1) = 6$.
$L[2]$ = T, il s'agit du deuxième T dans L. Recherchons la position du deuxième T dans F, il est en position 7. Ainsi $LF(2) = 7$.

Dans les deux cas, on en déduit $\mathsf{LF} = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}{4\ 6\ 7\ 0\ 1\ 2\ 8\ 3\ 5}$

---

prend les valeurs 0, 1, 3, 4, 6 pour \$, A, C, G et T respectivement. Finalement $\mathsf{LF}(i) = \mathsf{rank}_{L_i}(L, i) + \mathsf{First}(L_i) - 1$.

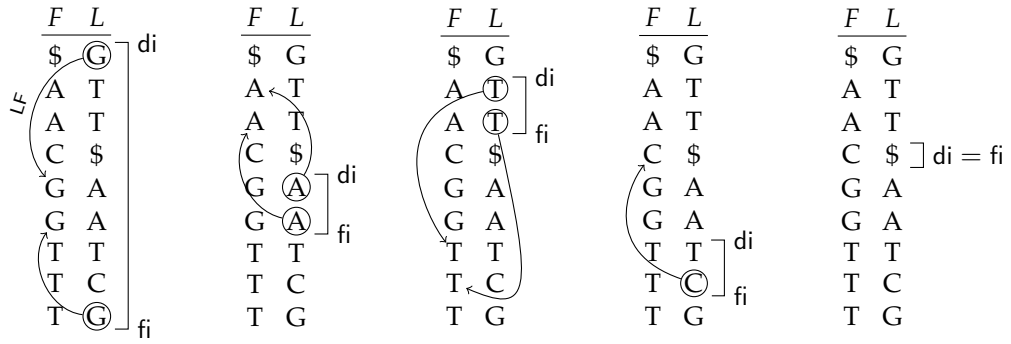
Dans la description de l'algorithme de recherche, nous avons indiqué qu'il faut calculer LF à la position de la première et de la dernière permutation circulaire se terminant par la lettre considérée P_i. Il n'est pas nécessaire pour cela de connaître les positions de ces permutations circulaires. En voyant le détail du calcul de la fonction LF, on comprend qu'il suffit de connaître les valeurs rank associés à ces positions. Ainsi, $\mathsf{rank}_{P_i}(L, \mathsf{di} - 1) + 1$ et $\mathsf{rank}_{P_i}(L, \mathsf{fi})$ donnent le rang pour la première et la dernière permutation se terminant par P_i [4] (pour un pseudo-code de l'algorithme, voir Algo. 1.1 page suivante).

[4] Sauf s'il n'y a pas d'occurrence de P_i dans l'intervalle, dans ce cas la première fonction retourne 1 et la seconde 0.

Ex. 1.17 — Recherche de motifs avec le FM-Index

$$T = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}{\mathrm{CTAGTTAG\$}}$$

Recherche de $P = \overset{0\ 1\ 2\ 3}{\mathrm{CTAG}}$



Après avoir parcouru tout le motif, l'intervalle est de taille 1. On peut en déduire qu'on a une seule occurrence de CTAG dans T.

Algo. 1.1 — Compter le nombre d'occurrences à l'aide d'un FM-index

1: **fonction** COMPTE(P, L, n)
2: di $\leftarrow 0$
3: fi $\leftarrow n - 1$
4: $i \leftarrow |P| - 1$
5: **tant que** di $\leq$ fi et $i \geq 0$ **faire**
6: di $\leftarrow \mathsf{First}(P[i]) + \mathsf{rank}_{P[i]}(L, \mathsf{di} - 1) + 1$
7: fi $\leftarrow \mathsf{First}(P[i]) + \mathsf{rank}_{P[i]}(L, \mathsf{fi})$
8: $i \leftarrow i - 1$
9: **fin tant que**
10: **si** di $\leq$ fi **alors**
11: **retourner** fi $-$ di $+ 1$
12: **sinon**
13: **retourner** 0
14: **fin si**
15: **fin fonction**

Sachant que la fonction LF est calculée en temps $lf(n, \sigma)$; que l'algorithme se déroule en, au plus, m étapes pour déterminer l'intervalle contenant toutes les occurrences de P ; alors déterminer le nombre d'occurrences d'un motif P peut être réalisé en temps $O(m\, lf(n, \sigma))$ avec un FM-Index [5].

Positions des occurrences La transformée de Burrows-Wheeler est uniquement composée de lettres et aucune information ne nous permet de connaître les positions des occurrences dans le texte d'origine. C'est pourquoi il faut échantillonner la table des suffixes. Nous allons échantillonner toutes les valeurs de la table des suffixes de la forme $j \log^{1+\varepsilon} n$, avec $0 \leq j \leq n/\log^{1+\varepsilon} n$ et ε est une constante strictement positive. Pour savoir quelles valeurs sont échantillonnées et lesquelles ne le sont pas, on peut utiliser un champ de bits compressé [6] (cf. section 2.1 page 45). Calculer une valeur $\mathsf{TS}[i]$, si elle n'est pas échantillonnée, se fait également avec la fonction LF. Le principe consiste à appliquer la fonction LF, et donc à parcourir le texte vers la gauche, jusqu'à ce qu'à une position, la valeur de TS soit échantillonnée. Une fois cette valeur $\mathsf{TS}[j]$ échantillonnée trouvée, on peut en déduire la valeur d'origine. La fonction LF a été appliquée $i - j$ fois. Celle-ci permet de se déplacer de droite à gauche dans le texte, la valeur d'origine $\mathsf{TS}[i]$ est donc $\mathsf{TS}[j] + i - j$ (voir Ex. 1.18 page suivante). La complexité en temps dépend de la distance d'échantillonnage, ici elle sera de $O(\log^{1+\varepsilon} n\, lf(n, \sigma))$ dans le pire des cas pour calculer une valeur de la table des suffixes (voir Algo. 1.2 page suivante).

Accès au texte Nous savons que la fonction LF permet de parcourir les lettres du texte de droite à gauche. Ainsi, on peut décompresser le texte en partant de L_0 et en itérant à l'aide de LF. Cependant, qu'en est-il pour la décompression d'un facteur du texte ? Si on souhaite décompresser un facteur $T[i \,..\, i + \ell - 1]$, on a besoin de connaître la position du suffixe (ou de la permutation circulaire) $T[i + \ell \,..]$ afin de récupérer $T[i + \ell - 1]$ dans L et ensuite d'itérer avec LF. La position de ce suffixe peut être connue en utilisant la table inverse des suffixes TIS (voir section 1.2.2.1 page 18). Il nous reste à savoir comment on peut retrouver les valeurs de TIS à partir des échantillons de TS. On souhaite connaître la position du suffixe $T[i + \ell \,..]$, pour cela déterminons la valeur j échantillonnée minimale et supérieure à $i + \ell$. Ensuite $\mathsf{LF}^{j-i-\ell}(j)$ donnera la position de $T[i + \ell \,..]$ et permettra l'accès à $T[i + \ell - 1]$. Enfin, ℓ nouveaux appels à LF permettront de récupérer la totalité du facteur.

Conclusion Le concept du FM-Index, tel que présenté précédemment, fonctionne avec n'importe quelle structure de données stockant $\mathsf{TBW}(T)$, permettant le calcul de $\mathsf{rank}_c(L, i)$, pour $c \in \Sigma$ et $0 \leq i \leq n$, ainsi que l'accès à $L[i]$. De nombreuses structures ont été présentées pour réaliser une telle tâche, nous les détaillerons en section 2.2 page 49.

[5] Nous verrons dans le chapitre 2 page 45 que rank et First peuvent être calculés en temps constant, ainsi $lf(n, \sigma) = O(1)$.

[6] Ferragina et Manzini utilisaient une méthode plus complexe car les champs de bits compressés n'existaient pas lorsque leur article a été publié.

Ex. 1.18 — Échantillonnage de la table des suffixes

$$T = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}{\mathrm{C\,T\,A\,G\,T\,T\,A\,G\,\$}}$$

TS	F	L	
	\$	G	
6	A	T	
	A	T	
0	C	\$	
	G	A	
3	G	A	
	T	T	← occurrence
	T	C	
	T	G	

On veut connaître la position de l'occurrence. On part de celle-ci et on utilise la fonction LF jusqu'à arriver à une valeur échantillonnée.
En appliquant deux fois LF, on arrive sur la valeur échantillonnée 3, nous étions donc partis de la valeur 5. L'occurrence est dans en position 5 dans le texte.

Algo. 1.2 — Retourner, avec le FM-index, tous les éléments d'un intervalle de la table des suffixes

Requiert pos : champ de bits tel que $pos[i] = 1$ ssi $\mathsf{TS}[i]$ est échantillonné.
Requiert TS_e : tableau stockant les valeurs échantillonnées de TS.
Assure que $\mathsf{TS}[d\,..\,f]$ est retourné.

fonction RÉCUPÈREPOSITIONS(L, d, f, pos, TS_e)
  pour $j = d$ à f **faire**
    $i \leftarrow j$
    $cpt \leftarrow 0$
    tant que $pos[i] = 0$ **faire**
      $i \leftarrow \mathsf{LF}(i)$
      $cpt \leftarrow cpt + 1$
    fin tant que
    $valeurs[j - d] \leftarrow SA_e[\mathsf{rank}_1(pos, i)] + cpt$
  fin pour
  retourner $valeurs$
fin fonction

1.3.2.2 Table compressée des suffixes

La table compressée des suffixes a été introduite par Grossi et Vitter (2000, 2005). Une version améliorée a ensuite rapidement été proposée par Sadakane (2000, 2003). Nous commençons par introduire la méthode de Grossi et Vitter puis explicitons les améliorations apportées par Sadakane.

Grossi et Vitter Leur méthode repose principalement sur la définition de la fonction Ψ. Cette fonction est l'inverse de la fonction LF évoquée dans la section 1.3.2.1 page 28. La fonction est donc définie ainsi : $\Psi(i) = j$ tel que $\mathsf{TS}[i] + 1 = \mathsf{TS}[j]$, sauf si $\mathsf{TS}[i] = n$, $\mathsf{TS}[j] = 0$.

Proposition 1. La fonction Ψ peut être découpée en, au plus, σ blocs strictement croissants.

Démonstration. Les σ blocs croissants correspondent aux blocs de suffixes commençant par la même lettre.
Prenons deux suffixes $T[i\,..] = a \cdot u$ et $T[i'\,..] = a \cdot v$, tel que $a \in \Sigma$ et $u, v \in \Sigma^*$. Soit $\mathsf{TS}[k] = i$ et $\mathsf{TS}[k'] = i'$, on suppose sans perte de généralité que $k < k'$.

On a donc $T[i\,..] < T[i'\,..]$, autrement dit $u < v$. Soit $\Psi(k) = j$ et $\Psi(k') = j'$, par définition $T[\mathsf{TS}[j]\,..] = u$ et $T[\mathsf{TS}[j']\,..] = v$. La table des suffixes étant triée en fonction de l'ordre lexicographique des suffixes, on en déduit que $j < j'$. Cela signifie que $k < k' \Rightarrow \Psi(k) < \Psi(k')$, si $T[\mathsf{TS}[k]] = T[\mathsf{TS}[k']]$. □

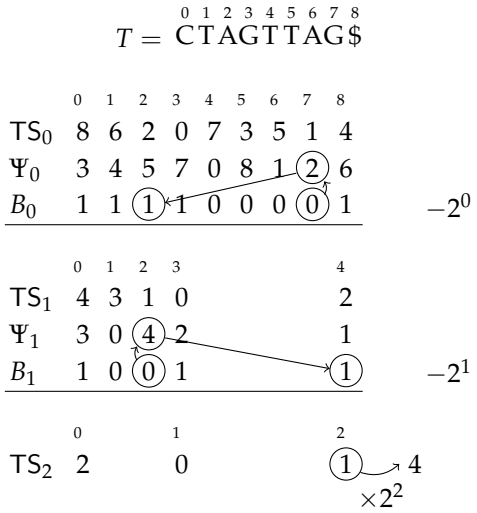
Cette propriété permet de stocker Ψ beaucoup plus efficacement que TS (cf. section 2.1.4 page 48). La fonction Ψ correspond à une liste chaînée permettant de retrouver n'importe quelle valeur de TS. Comme toute liste chaînée l'accès à une valeur peut être réalisé en temps linéaire, ce qui n'est pas acceptable pour l'objectif qui nous concerne. C'est pourquoi Grossi et Vitter ont réalisé une structure hiérarchique dans laquelle ils considèrent des échantillonnages du TS.

On définit TS_k comme étant le TS originel dans lequel on a conservé les seules valeurs multiples de 2^k, avec $k \in \{0, e, 2e, \ldots, \ell\}$, $e = \lceil \varepsilon \log\log n \rceil$, $0 < \varepsilon < 1$ et $\ell = \lceil \log\log n \rceil$. La fonction Ψ_k est également calculée pour chacun des TS_k, TS_k par la suite n'est pas conservé : seuls Ψ_k et un champ de bits B_k, indiquant quelles positions ont été échantillonnées, sont stockés. Au dernier niveau, TS_ℓ est stocké explicitement. Il contient alors $n/\log n$ valeurs et peut être stocké directement en utilisant n bits (voir Ex. 1.19 page suivante).

Ainsi la table compressée des suffixes, telle que présentée par Grossi et Vitter, permet de rechercher un motif en temps $O((m + \log^\varepsilon n)\log n)$ et nécessite un espace en $O(n)$ (contre $n \log n$ pour la table des suffixes). Cette structure d'indexation, d'après la définition donnée en section 1.1.2.2 page 8, n'est pas compressée.

Pour récupérer n'importe quelle valeur de la table des suffixes, il faut parcourir les niveaux. Au sein d'un niveau, l'utilisation de la fonction Ψ permet de d'atteindre une valeur échantillonnée (voir Algo. 1.3 page 35). Le fait de permettre l'accès à n'importe quelle valeur de la table des suffixes autorise la recherche de motifs selon le même algorithme que pour la table des suffixes. La seule différence notable est la complexité en temps : pour la table des suffixes une valeur quelconque peut être récupérée en temps constant contre

Ex. 1.19 — Table compressées des suffixes. Calcul de la valeur $\mathsf{TS}[7]$.



Résultat : $\mathsf{TS}[7] = 4 - 2 - 1 = 1$

Méthode : On part du niveau 0, on regarde si la valeur courante est échantillonnée et comme elle ne l'est pas on utilise la fonction Ψ pour se rendre à la prochaine valeur. Celle-ci est forcément échantillonnée (une valeur sur deux, pour un niveau donné, l'est). Ensuite on peut passer au niveau suivant et itérer le processus, jusqu'à atteindre le dernier niveau.

$O(\log^{\varepsilon} n)$ pour la table compressée. Par ailleurs, on notera qu'il est nécessaire, pour cet algorithme, de conserver le texte d'origine.

Sadakane La version proposée par Sadakane est plus complète (elle permet également d'accéder à la table inverse des suffixes), plus rapide (complexité du temps de recherche identique à la table des suffixes) et plus économe en espace (il s'agit d'un auto-index).

Accès au texte Nous avons vu précédemment que la table compressée des suffixes de Grossi et Vitter permet uniquement d'accéder aux valeurs de la table des suffixes. Sadakane, sur un modèle semblable au FM-Index, propose également d'accéder au texte directement à partir de la structure d'indexation.

Une position i est une *frontière* si et seulement si $T[\mathsf{TS}[i]] \neq T[\mathsf{TS}[i+1]]$. Un champ de bits D_0 enregistre toutes les frontières pour TS_0. De plus, un tableau C stocke toutes les lettres de Σ dans l'ordre lexicographique. Ces deux structures nous permettent de calculer $T[\mathsf{TS}[i]] = C[\mathsf{rank}_1(D_0, i) - 1]$ et de façon plus générale $T[\mathsf{TS}[i] + k] = C[\mathsf{rank}_1(D_0, \Psi^k(i)) - 1]$. Dans ces conditions, une lettre pouvant être récupérée en temps constant, le temps de recherche dans la table compressée des suffixes est en $O(m \log n)$.

Algo. 1.3 — Récupérer une valeur de la table compressée des suffixes

Requiert $k \in \{0, e, 2e, \ldots, \ell\}$ et $0 \leq i \leq n$
1: **fonction** VALEURSA(k, i)
2: **si** $k = \ell$ **alors retourner** $\mathsf{TS}_\ell[i] \times 2^\ell$
3: **sinon**
4: $cpt \leftarrow 0$
5: **si** $\mathsf{TS}_k[i]$ n'est pas échantillonné **alors**
6: $i \leftarrow \Psi_k(i)$
7: $cpt \leftarrow 2^k$
8: **fin si**
9: **retourner** VALEURSA($k+1$,i)$-cpt$
10: **fin si**
11: **fin fonction**

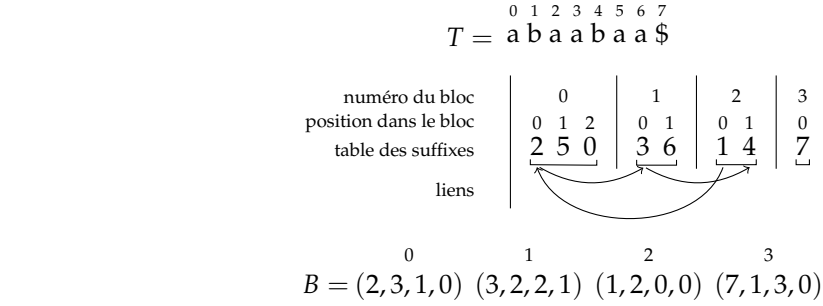
Algo. 1.4 — Accès à une valeur de la table inverse grâce à la table compressée des suffixes

Requiert $0 \leq i \leq n$
1: **fonction** VALEURISA(i)
2: $r \leftarrow i \pmod{2^\ell}$
3: $i \leftarrow \mathsf{TIS}_\ell[\lfloor i/2^\ell \rfloor]$
4: $k \leftarrow \ell - 1$
5: **tant que** $k \geq 0$ **faire**
6: $i \leftarrow \mathsf{select}_1(D_k, i)$
7: **si** $r > 2^k$ **alors**
8: $r \leftarrow r - 2^k$
9: $i \leftarrow \Psi_k(i)$
10: **fin si**
11: $k \leftarrow k - 1$
12: **fin tant que**
13: **retourner** i
14: **fin fonction**

En l'état, l'accès au texte ne peut se faire que via un indice de la table des suffixes mais on ne sait pas récupérer un facteur quelconque $T[g\,..\,d]$. Pour remédier à cela, Sadakane permet l'accès à la table inverse des suffixes. À l'aide de cette table, on peut calculer $\mathsf{TIS}[g]$ ce qui nous permet ensuite de retrouver $T[g]$ à partir de la formule précédente.

Accès à la table inverse des suffixes L'accès à la table inverse des suffixes se fait de façon similaire à l'accès à la table des suffixes. Il nous suffit d'ajouter une table TIS_ℓ correspondant à la table inverse de TS_ℓ. Ensuite, en remontant à travers les niveaux, il est possible de récupérer la valeur (voir Algo. 1.4).

Ex. 1.20 — Construction des quadruplets de la table compacte des suffixes



Les trois premières valeurs 2, 5 et 0 sont liées aux valeurs 3, 6 et 1. Le quadruplet correspondant à ces valeurs est donc $(2,3,1,0)$ car la première valeur du bloc est un 2, le bloc est de longueur 3 et les valeurs liées commencent dans le bloc 1 en position 0.

De même que pour TS, il est possible de récupérer une valeur quelconque de TIS en temps $O(\log^{\varepsilon} n)$ et donc d'extraire un facteur quelconque du texte, de longueur k, en temps $O(k + \log^{\varepsilon} n)$.

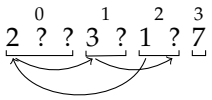
1.3.2.3 Table compacte des suffixes

Le principe à la base de la table compacte de suffixes, introduite par Mäkinen (2000, 2003), repose sur les régularités présentes dans la table des suffixes. Naturellement, ces régularités n'existent que pour des textes compressibles, de la même manière qu'on observe des suites de lettres consécutives dans la TBW seulement pour des textes compressibles. Ces régularités consistent en de grandes zones de la table des suffixes, répétées à plus ou moins 1 (dans Ex. 1.20 on voit que les valeurs consécutives 2, 5 et 0 sont répétées, en ajoutant 1, puisque 3, 6 et 1 sont également consécutifs dans la table des suffixes). Ces zones sont très faciles à repérer lorsqu'on se réfère à la fonction Ψ : les zones correspondent alors à une série de valeurs incrémentées de 1.

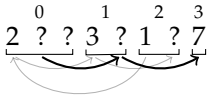
Une fois repérées les différentes zones, on cherche à en tirer parti pour stocker un minimum d'informations. Seules les zones de départ d'un lien sont considérées, elles sont appelées *blocs*. Les blocs sont stockés sous forme de quadruplets (v, ℓ, b, p) où v est la première valeur du bloc, ℓ la longueur de ce bloc, b est le numéro du bloc dans lequel commence la zone d'arrivée, p désigne la position de début de la zone d'arrivée relativement au bloc b (voir Ex. 1.20). Par convention, on lie le dernier bloc à lui-même, forcément de longueur 1. Il correspond au symbole \$.

Recherche de motifs Seule la première valeur de chaque bloc est mémorisée, les autres valeurs de la table non compacte des suffixes peuvent s'en déduire. Pour la recherche de motifs, dans un premier temps, nous utilisons les informations stockées « en clair » : la première valeur de chaque bloc. Avec

Ex. 1.21 — Décompaction d'une valeur de la table compacte



Retrouvons la valeur en deuxième position. Pour cela nous utilisons les liens de la table compacte des suffixes jusqu'à arriver sur une valeur échantillonnée de la table compacte.



La valeur échantillonnée est 7, la valeur visitée précédemment est donc 6 et la valeur de départ, en deuxième position dans la table des suffixes, est 5.

---

ces valeurs, nous suivons le même principe que pour la table des suffixes (recherche dichotomique et tests des suffixes). Une fois localisée la plage de blocs intéressants, il reste alors à connaître les valeurs présentes à l'intérieur de ces blocs pour déterminer précisément les occurrences du motif. Pour connaître les valeurs contenues dans les blocs, nous les décompactons en suivant la méthode expliquée ci-après.

Décompaction Malgré le compactage des blocs, on doit être en mesure de retrouver n'importe quelle valeur de la table non compacte des suffixes. Pour cela, il suffit de suivre les « liens » menant d'un bloc à un autre jusqu'à rencontrer une valeur stockée en clair. En soustrayant le nombre de liens suivis pour atteindre cette valeur on obtient la valeur cherchée (voir Ex. 1.21).

On est sûr, à un moment, de rencontrer une valeur stockée en clair puisqu'il existe au moins un bloc de taille 1 (celui représentant le terminateur). Cependant, pour éviter de suivre trop de liens (et donc de passer trop de temps à chercher une valeur), on peut construire la table compacte de façon à ce que le nombre de plongées à effectuer soit limité à une certaine constante D. De même, on limite la taille des blocs à une longueur C. Enfin, pour borner le temps de décompactage, on ne considère que des zones non chevauchantes lors de la création des blocs (ainsi aucun bloc ne peut pointer sur lui-même, en dehors du bloc du terminateur).

Table compacte compressée La table compacte des suffixes n'est pas un auto-index car elle nécessite le texte pour effectuer la recherche. C'est pourquoi Mäkinen et Navarro (2004) ont introduit la table compacte compressée des suffixes. Cette dernière repose sur le même principe que la table compacte des suffixes mais permet également de stocker le texte de façon efficace, sur un principe similaire à celui développé par Sadakane pour la table compres-

Ex. 1.22 — Stockage efficace de la table PLPC

Nous reprenons l'exemple 1.10 page 20.

i	0	1	2	3	4	5	6	7	8
TS	8	6	2	0	7	3	5	1	4
PLPC	0	2	0	0	1	0	3	1	0
PLPC'	8	8	2	0	8	3	9	2	4

La table PLPC' est la somme de la table TS et PLPC. Nous rangeons maintenant la table PLPC' en fonction des valeurs croissantes de TS.

i	0	1	2	3	4	5	6	7	8
PLPC_S	0	2	2	3	4	8	8	8	8

Pour retrouver $\mathsf{PLPC}[\mathsf{TS}[4]]$, il nous faut connaître $\mathsf{TS}[4] = 7$. On sait que $\mathsf{PLPC}_S[7] = 8$, donc la valeur PLPC est $8 - 7 = 1$.
Pour retrouver une valeur quelconque $\mathsf{PLPC}[i]$, il faut avoir la table inverse des suffixes pour ensuite utiliser la même méthode que précédemment.

---

sée des suffixes. De plus, les quadruplets sont remplacés par des champs de bits et un tableau d'entiers. Ces informations sont suffisantes pour permettre de retrouver les valeurs des quadruplets.

1.3.2.4 Arbre compressé des suffixes

Dès l'introduction de la table compressée des suffixes, Grossi et Vitter (2000, 2005) s'intéressent à l'arbre compressé des suffixes et proposent une solution hybride composée du LZ-Index d'origine, de Kärkkäinen et Ukkonen (1996), d'un arbre de recherche et de leur table compressée des suffixes. D'autres approches sont apparues plus tard (Sadakane, 2007 ; Russo *et al.*, 2008) mais restent basées sur la table compressée des suffixes. Ces approches ont le mérite d'autoriser certaines, voire toutes, les opérations disponibles avec l'arbre des suffixes.

Dans son article, Sadakane présente également une méthode pour stocker la table PLPC en utilisant uniquement $2n + o(n)$ bits. Le principe consiste, dans un premier temps, à faire la somme de la table des suffixes et de la table PLPC. Ensuite les entrées de la table résultante sont triées en fonction des valeurs de TS. Finalement la table obtenue est également triée (voir Ex. 1.22) et peut être stockée très efficacement en utilisant un champ de bits (voir section 2.1.4 page 48).

1.4 Synthèse

Les techniques détaillées dans la section précédente correspondent à l'ensemble des méthodes connues à ce jour pour obtenir des auto-index. Pour des

Tab. 1.1 — Complexités en espace et en temps des structures d'indexation

Nom	Espace	Comptage	Localisation
Arbre des suffixes	$O(n \log n)$	$O(m)$	$O(1)$
Arbre contracté des suffixes	$O(n \log n)$ [a]	$O(m)$	$O(1)$
Table des suffixes	$n \log n$	$O(m \log n)$	$O(1)$
Vecteur des suffixes	$O(n \log n)$	$- - -$	$- - -$
LZ-index	$(3+\varepsilon)H_k(T) + o(n \log \sigma)$	$O(m)$	$O(\log n)$
FM-index	$nH_k(T) + o(n \log \sigma)$	$O(m)$	$O(\log^{1+\varepsilon} n)$
Table compressée des suffixes	$\frac{1}{\varepsilon} nH_k(T) + o(n \log \sigma)$	$O(m \log \sigma + \log^{2+\varepsilon} n)$	$O(\log^{1+\varepsilon} n)$
Table compacte et compressée des suffixes	$O(n(1 + H_k \log n))$	$O(m \log n)$	$O(\log n)$

[a] Aucune indication n'est donnée dans leur article mais c'est la complexité à laquelle on peut s'attendre en raison des pointeurs nécessaires.

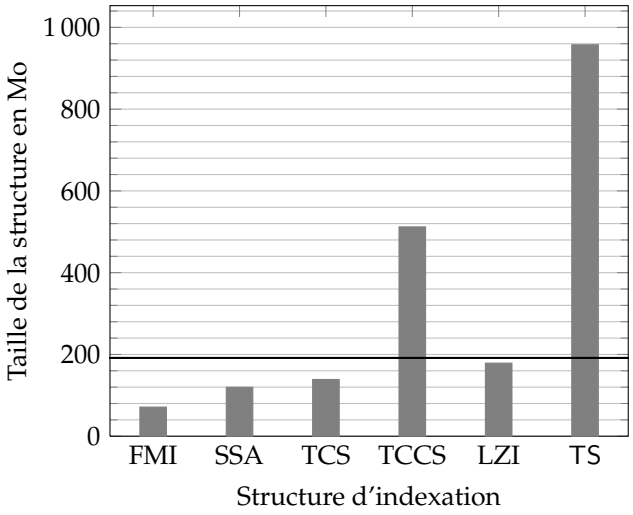
informations complémentaires, il est possible de se référer aux articles d'origine ou à l'article très complet de Navarro et Mäkinen (2007).

Nous résumons dans la TAB. 1.1 les complexités en espace et en temps des différentes structures d'indexation considérées dans ce chapitre. Les versions considérées sont Arroyuelo et Navarro (2007b), pour le LZ-index ; Ferragina *et al.* (2007), pour le FM-index ; Grossi *et al.* (2003), pour la table compressée des suffixes et Mäkinen et Navarro (2004), pour la table compacte et compressée des suffixes. La colonne « comptage » désigne la complexité en temps pour déterminer le nombre d'occurrences d'un motif P, de longueur m et la colonne « localisation » désigne la complexité en temps additionnelle par rapport au comptage pour rapporter la position d'une seule occurrence.

D'un point de vue plus expérimental, nous allons maintenant nous intéresser aux performances en pratique de celles-ci. Le but de cette partie n'est pas de fournir un comparatif exhaustif[7] entre toutes les structures d'indexation mais de donner des éléments permettant la comparaison. Nous choisissons deux textes différents en termes d'alphabets et de répétitions : le chromosome 1 de la souris et la version de Wikipedia en afrikaans. Ces deux textes ont été choisis en raison de leurs différences en termes de répétitions et de taille d'alphabet. Nous donnerons plus de détail sur ces textes et sur la façon de

[7] Ferragina *et al.* (2009) ont mené des expériences plus complètes, sur un panel plus large de données.

Fig. 1.2 — Espace utilisé par les différentes structures pour l'indexation du chromosome 1 de la souris.



La ligne grasse horizontale à 191 Mo représente la taille utilisée par le texte d'origine.

les obtenir en section 4.2.1.1 page 86, lorsque nous mènerons des expériences plus larges sur la structure d'indexation que nous présenterons.

Les structures d'indexation compressées ont été récupérés à partir du site de Pizza & Chili [8]. Ce site se veut être le point central en ce qui concerne les structures d'indexation compressées. Il fournit de nombreuses implantations de structures d'indexation, des données pour tester ces structures ainsi que des outils permettant la manipulation des structures et des données.

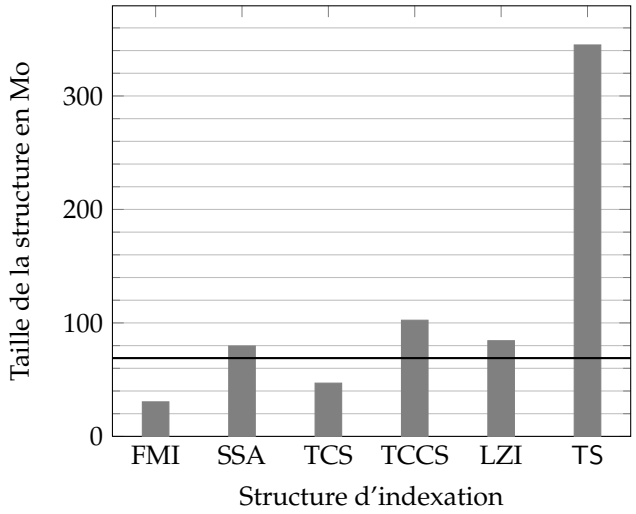
Nous utilisons deux versions du FM-index (notées SSA et FMI), la table compressée des suffixes (TCS, de Sadakane), la table compacte compressée des suffixes (TCCS), une version du LZ-index (LZI) ainsi que la table des suffixes (TS) à titre de comparaison. Tous les codes ont été compilés avec leurs options par défaut et les structures d'indexation construites sur les textes en utilisant les options par défaut.

1.4.1 Espace utilisé par les structures d'indexation

L'espace utilisé par ces structures d'indexation est présenté en FIG. 1.2 pour le chromosome 1 de la souris et en FIG. 1.3 page ci-contre pour la version afrikaans de Wikipedia. L'espace utilisé par la table des suffixes a été fixé à $5n$, soit $4n$ pour la table des suffixes elle-même et n pour le texte, indispensable lors de la recherche.

[8] `http://pizzachili.dcc.uchile.cl`

Fig. 1.3 — Espace utilisé par les différentes structures pour l'indexation de la version afrikaans de Wikipedia.



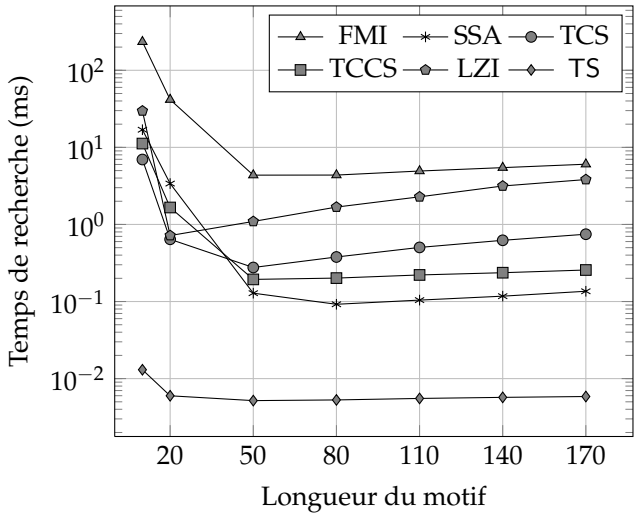
La ligne grasse horizontale à 69 Mo représente la taille utilisée par le texte d'origine.

Dans les deux cas, la version FMI du FM-index apparaît comme la plus économe en espace. Elle utilise de deux à trois fois moins de place que le texte lui même. Les auteurs de cette implantation ont en effet pris un soin particulier à l'occupation mémoire de leur structure ce qui explique les très bons résultats obtenus. Il est important de noter qu'il semble difficile d'obtenir de meilleurs résultats en espace avec une structure basée sur le FM-index. En compressant le chromosome 1 de la souris avec l'utilitaire `bzip2`, basé lui aussi sur la transformée de Burrows-Wheeler, on obtient au mieux un fichier de 49,3 Mo. À cela, dans la structure d'indexation, s'ajoute l'échantillonnage de la table des suffixes, par défaut FMI garde 2 % des valeurs, soit 3,8 millions de valeurs stockée en 15,3 Mo. En associant le résultat de `bzip2` à cet échantillonnage on a un résultat qui occupe plus de 64 Mo mais avec lequel il serait bien compliqué de rechercher des motifs. Avec des éléments équivalents, la structure FMI occupe 71 Mo, soit environ 10 % en plus, et permet la recherche de motifs.

Dans l'ensemble les autres structures d'indexation compressées tirent bien parti du petit alphabet, de taille 4, de la séquence génomique. À l'exception de TCCS, qui se démarque assez largement, elles utilisent toutes un espace inférieur à celui utilisé par le texte d'origine pour le chromosome 1 de la souris.

Pour la version afrikaans de Wikipedia, hormis FMI, toutes les structures d'indexation compressées sont proches de la taille d'origine du texte (légèrement au dessus ou en dessous). La méthode de compression de TCCS repose sur les répétitions présentes dans la table des suffixes. Or ce type de répétitions

Fig. 1.4 — Temps de recherche par motif pour le chromosome 1 de la souris



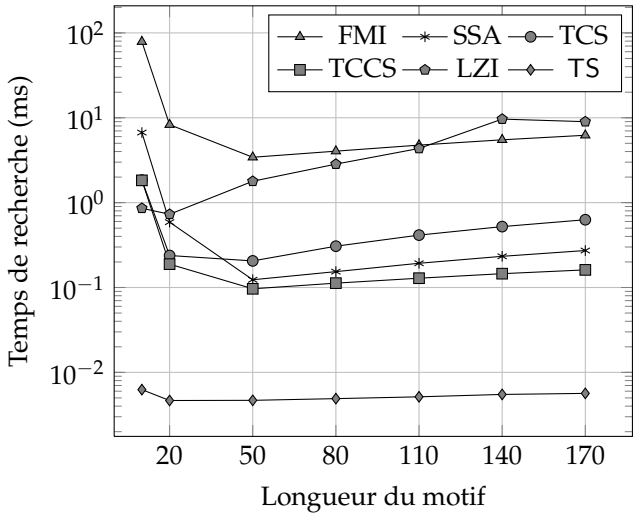
dépend assez faiblement de la taille de l'alphabet mais bien plus du contenu du texte et des répétitions qu'il peut contenir. C'est pourquoi TCCS n'arrive pas forcément à tirer parti du petit alphabet des séquences génomiques.

1.4.2 Temps de recherche

L'utilisation principale des structures d'indexation est la recherche de motifs. Avoir des structures d'indexation économes en espace est important mais il faut qu'elles continuent à donner rapidement la liste des occurrences d'un motif. Pour cela nous extrayons 10 000 facteurs distincts de longueur 200 de chacun des deux textes précédents. Nous allons rechercher, à l'aide des structures d'indexation, toutes les positions des occurrences des préfixes de longueur 10, 20, 50, 80, 110, 140 et 170 de ces facteurs. Dans la FIG. 1.4 nous présentons les temps de recherche pour le chromosome 1 de la souris et dans la FIG. 1.5 page ci-contre nous donnons les temps de recherche pour la version afrikaans de Wikipedia.

Sans surprise l'algorithme de recherche utilisant la table des suffixes est bien plus rapide (de quelques dizaines de fois à plus de mille) qu'avec n'importe quelle structure d'indexation compressée. La version FMI du FM-index s'illustre en étant, quasiment, toujours la plus lente. Cela illustre une nouvelle fois le compromis espace-temps avec lequel il faut souvent composer. Il est certes possible d'avoir des structures d'indexation très efficaces en terme d'espace mémoire mais dans ce cas le temps de recherche devient très long comparé aux autres solutions. Conformément à ce qu'on pouvait attendre en regardant ses complexités théoriques (quadratique en fonction de la longueur du motif, voir section 1.3.1.3 page 26), l'utilisation du LZ-index est de moins

Fig. 1.5 — Temps de recherche par motif pour la version afrikaans de Wikipedia



en moins avantageuse au fur et à mesure que la longueur du motif augmente. Néanmoins, pour de courts motifs ayant un très grand nombre d'occurrences le LZ-index peut être plus rapide que les autres structures, comme on peut le voir dans la FIG. 1.5, pour les motifs de longueur 10. Cela s'explique par le principe du LZ-index qui, contrairement aux méthodes basées sur la table des suffixes, ne repose pas sur un échantillonnage. Ainsi récupérer les positions de toutes les occurrences d'un motif n'est pas plus coûteux pour le LZ-index que de compter uniquement le nombre d'occurrences. Les trois autres méthodes basées sur la table des suffixes (SSA, TCS, TCCS) ne se distinguent pas fondamentalement en termes de temps de recherche. Le remontée rapide du temps de recherche pour les motifs de longueur 10 à 20 est due au nombre d'occurrences qui devient très important : 1 713 en moyenne pour chaque motif de longueur 10, pour le chromosome 1 de la souris et 284 pour le corpus afrikaans. On n'observe pas de grandes différences en termes de temps de recherche entre les deux textes : les ordres de grandeur restent similaires entre les différentes méthodes. Cela corrobore les complexités théoriques que nous avons pour ces structures, elles ne sont pas dépendantes de la taille de l'alphabet pour le temps de recherche.

1.4.3 Conclusion

Ces quelques expériences nous permettent de mieux apprécier l'avancée que constituent les structures d'indexation compressées par rapport à des structures plus anciennes comme la table des suffixes. L'espace mémoire utilisé par ces structures compressées est de l'ordre de la taille du texte lui-même.

Par ailleurs, on a vu que l'implantation joue un rôle important dans le taux de compression et il est possible d'avoir des structures compressées utilisant presque aussi peu de place que le texte compressé. Ces structures d'indexation ont des résultats très variables en termes de temps de recherche et plus la structure va être compressée moins la recherche sera rapide.

On note également que les approches basées sur l'échantillonnage de la table des suffixes offrent de meilleurs résultats, en temps et en espace, que l'approche basée sur la compression LZ-78. Cependant la technique de l'échantillonnage n'est pas parfaite et devient coûteuse lorsque le nombre d'occurrences est grand. Dans ces conditions, le LZ-index peut devenir le plus rapide, comme nous l'avons observé dans les expériences précédentes.

Néanmoins les méthodes basées sur l'échantillonnage de la table des suffixes (que ce soit le FM-index ou la table compressée des suffixes) ont fait l'objet de nombreuses recherches et c'est peut-être une explication à la supériorité de ces solutions par rapport au LZ-index pour lequel il existe moins de résultats.

Malgré ces résultats encourageants on voit les limites actuelles des structures d'indexation compressées :

- une bonne compression empêche d'avoir un temps de recherche compétitif ;
- les méthodes utilisant un échantillonnage sont pénalisées lorsque le motif a un grand nombre d'occurrences ;
- les méthodes utilisant le LZ-index ne sont pas efficaces pour compter le nombre d'occurrences et peu compétitives pour trouver les positions des occurrences.

Chapitre 2

Outils pour les structures d'indexation

Suite à l'apparition des structures d'indexation compressées au début des années 2000, de nombreux outils permettant des calculs sur des champs de bits ou de lettres ont été mis au point. Ces outils, autorisant le calcul d'opérations basiques, ont permis de faciliter les améliorations, tant en temps qu'en espace, des structures d'indexation compressées. Par exemple, de nombreuses variantes du FM-index existent en raison des différentes façons de stocker la transformée de Burrows-Wheeler à l'aide d'outils présentés au sein de ce chapitre.

Dans un premier temps nous considérerons les champs de bits (section 2.1) puis nous nous intéresserons à des alphabets plus grands, avec des champs de lettres (section 2.2 page 49).

2.1 Champs de bits

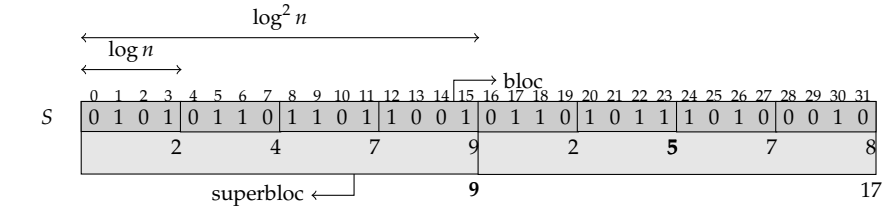
Les champs de bits sont utilisés pour représenter des informations sous forme binaire, en utilisant le moins de place possible. Sur un champ de bits stocké « naïvement » les opérations possibles en temps constant sont : accès à une valeur ; modification d'une valeur.

Des opérations nécessitant le comptage des 0 et des 1 peuvent aussi être très utiles. Par exemple les opérations $\mathsf{rank}_{0/1}(S, i)$ et $\mathsf{select}_{0/1}(S, i)$ permettent respectivement de connaître le nombre de 0 (ou 1) dans la séquence binaire S de la position 0 à la position i, incluse, et de connaître la position du $i^{\text{ème}}$ 0 (ou 1) dans S, cependant n est retourné s'il n'y a pas i 0 (ou 1) dans S. Par la suite, nous considérons que $|S| = n$. Notons que $\mathsf{rank}_0(S, i) = i - \mathsf{rank}_1(S, i)$.

2.1.1 Représentation succincte

Une version asymptotiquement optimale a été présentée par Munro (1996). Pour l'opération rank, elle consiste à diviser le champ de bits en blocs et superblocs de tailles égales. Les valeurs sont pré-calculées pour ces blocs et superblocs. Pour être capable de réaliser le calcul pour n'importe quelle position i, une table additionnelle stocke pour tous les facteurs possibles de longueurs

Ex. 2.1 — Champ de bits succinct avec opérations rank. Pour chaque bloc on stocke le nombre de 1 du début du superbloc jusqu'à la fin du bloc. Pour chaque superbloc, on stocke le nombre de 1 du début de la séquence jusqu'à la fin du superbloc. Pour l'exemple on suppose $\log n = 4$.



Longueur

Demi-blocs	Longueur
00	0 0
01	0 1
10	1 **1**
11	1 2

Le préfixe de longueur 1 du demi-bloc 01 contient zéro 1

$\mathsf{rank}_1(S, 25)$ = **9**+**5**+**1** = 15
9 correspond au nombre de 1 dans le premier superbloc,
5 est le nombre de 1 jusque dans le deuxième bloc du deuxième superbloc,
1 est le nombre de 1 dans 10, obtenu grâce à la table pré-calculée.

inférieures ou égales à $(\log n)/2$ les valeurs rank associées (voir Ex. 2.1 page suivante).

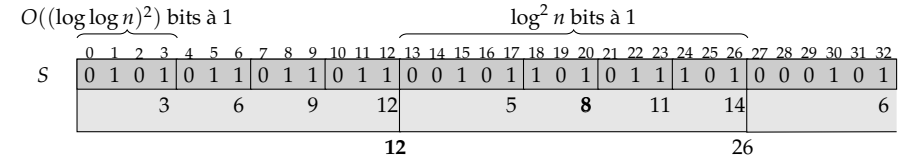
L'opération select est gérée de manière similaire. Cependant, maintenant, les blocs et superblocs contiennent un nombre variable de bits mais toujours le même nombre de bits à 1, respectivement $O((\log \log n)^2)$ et $\log^2 n$. Une table pré-calculée, similaire à celle utilisée pour rank, est également nécessaire. Un exemple est donné dans l'Ex. 2.2 page ci-contre, pour l'opération select_1. Par souci de clarté, l'exemple a été simplifié et ne considère que des superblocs ou blocs « courts ». En réalité, un superbloc ou un bloc est considéré « long » lorsque sa longueur, définie en fonction du nombre de 1 qui le compose, dépasse une longueur prédéfinie. Dans ce cas les positions de tous les 1 sont stockés explicitement.

2.1.2 Compression

2.1.2.1 Représentation par identifiants

Raman *et al.* (2002) ont adapté la méthode précédente afin d'obtenir une version compressée. Cette approche consiste à stocker des identifiants pour chacun des demi-blocs plutôt que la séquence S elle-même. Ainsi un demi bloc B de longueur $(\log n)/2$ contenant b bits à 1 sera codé par un couple (b, o). Dans ce codage, o désigne l'indice du bloc B parmi tous les blocs de longueur $(\log n)/2$ ayant b bits à 1. Par rapport à la méthode succincte, des tables additionnelles sont requises car le codage d'un demi-bloc requiert un nombre variable de bits. Dans ce cas, le champ de bits occupe un espace en $nH_0(S) + o(n)$ bits et toutes les opérations restent en temps constant.

Ex. 2.2 — Champ de bits succinct avec opérations select. Pour l'exemple on suppose $\log^2 n = 8$, $\log n = 4$, et $O((\log\log n)^2) = 2$.



Nb. bits à 1

Demi-blocs	1	2
00	0	0
01	**2**	0
10	1	0
11	1	2

Le deuxième bit du demi-bloc est le premier bit à 1.

Le nombre de bits à 1 par blocs (2) et superblocs (8), est fixe.
$\mathsf{select}_1(S, 13)$ = **12**+**8**+**2** = 22
On veut le treizième 1, c'est le premier 1 dans le troisième bloc du deuxième superbloc ($13 = 8 + 2 \times 2 + 1$).
On somme les positions pour le superbloc et le bloc précédent (**12**+**8**). Il reste à connaître la position du premier 1 dans le bloc, on utilise la table pré-calculée pour cela.

2.1.2.2 Gap-encoding

Il existe un certain nombre de cas où le nombre de bits à 1 est en $o(n)$. Pour ces cas là, d'autres méthodes de compression peuvent être appliquées. Ainsi la séquence $S = 0^{g_0}10^{g_1}1\cdots10^{g_{k-1}}$, avec $g_i \geq 0$, $0 \leq i < k$, peut être stockée uniquement avec le codage des g_i. Ensuite, de même que pour la méthode de Raman *et al.* (2002), il faut des tables additionnelles qui permettront de décoder n'importe quelle partie de la séquence d'origine en temps constant. Le principe pour avoir les opérations rank et select en temps constant reste le même.

Soit ℓ le nombre de bits à 1, cette représentation nécessite $\ell\log(n/\ell) + o(n)$ bits. Or $nH_0 = \ell\log(n/\ell) + (n-\ell)\log(n/(n-\ell))$, on en déduit que la représentation avec gap-encoding utilise moins de $nH_0 + o(n)$ bits.

Par ailleurs d'autres solutions existent pour résoudre ce problème des champs de bits creux (Gupta *et al.*, 2007a ; Mäkinen et Navarro, 2007 ; Okanohara et Sadakane, 2007). Gupta *et al.* montrent que, malgré des complexités similaires, les résultats en espace diffèrent fortement dans la pratique. Ainsi pour des champs de bits de 2^{32} bits et contenant 100 000 bits à 1, leur solution utilise moins de deux millions de bits contre plus de 130 millions pour Mäkinen et Navarro.

2.1.2.3 Run-length encoding

Mäkinen et Navarro (2005) proposent une façon de stocker les champs de bits contenant de longues suites de 0 et de 1. Soit B le champ de bits originel, celui-ci sera stocké à l'aide de deux champs de bits B^1 et B_{rl}. Le premier se contente de marquer avec un 1 la position de début de chaque suite de 1 dans B. Le second stocke ensuite la longueur, en unaire, des suites de 1 contenues

dans B. Ces deux champs sont suffisants pour permettre le calcul des opérations rank et select.

Exemple.

$$
\begin{array}{rl}
 & \begin{array}{cccccccccccccccccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 \end{array} \\
B = & 0\,0\,0\,\underbrace{\mathbf{1}\,1\,1\,1}_{4}\,0\,0\,0\,0\,0\,0\,0\,0\,\underbrace{\mathbf{1}\,1}_{2}\,0\,0\,\underbrace{\mathbf{1}\,1\,1\,1\,1\,1\,1}_{7}\,0\,0\,0 \\
\mathsf{B}^1 = & 0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0 \\
\mathsf{B}_{\mathsf{rl}} = & \underbrace{0\,0\,0\,1}_{4}\,\underbrace{0\,1}_{2}\,\underbrace{0\,0\,0\,0\,0\,0\,1}_{7}
\end{array}
$$

Les deux champs de bits B^1 et B_{rl} sont stockés en utilisant les champs de bits avec gap-encoding.

2.1.3 Mise à jour des champs de bits

Dans le cas où on souhaite ajouter, supprimer, ou modifier des éléments de la séquence S, il faudrait modifier une partie des tables pré-calculées pour tenir compte des changements. Cela serait beaucoup trop coûteux en temps de calcul et c'est une autre approche qui est préférée.

Des arbres binaires équilibrés sont utilisés pour stocker à la fois la séquence et les informations nécessaires pour les opérations rank et select. La séquence est stockée dans les feuilles de l'arbre, chaque feuille correspond à un superbloc. Les nœuds de l'arbre stockent le nombre de bits et le nombre de 1 contenus dans le sous-arbre gauche. Pour les opérations rank et select, en partant de la racine, la consultation des informations contenues dans les nœuds permet d'accéder à la feuille qui nous intéresse. Ensuite un parcours de la feuille, à l'aide d'une table pré-calculée, identique à celle du cas statique, permet de donner la réponse à la requête.

Pour les opérations d'insertion ou suppression de bits, l'insertion ou la suppression d'un bit dans une feuille peut amener à la création ou la fusion de feuilles. Dans ce cas, les algorithmes classiques pour les ajouts ou suppressions de feuilles dans des arbres binaires équilibrés sont utilisés. Ensuite, il suffit de mettre à jour les valeurs dans les nœuds sur le chemin qui va de la feuille modifiée jusqu'à la racine.

Dans les feuilles, la séquence peut également être compressée selon l'une des méthodes vue précédemment. La solution la plus efficace à ce jour est celle de Mäkinen et Navarro (2008), qui permet d'avoir toutes les opérations en temps $O(\log n)$ et qui occupe $nH_0(S) + o(n)$ bits.

2.1.4 Sommes partielles

Une application de ces champs de bits est, par exemple, le calcul de sommes sur des séquences de nombres entiers positifs. Considérons le problème suivant : soit m entiers positifs $N_0, \ldots, N_{m-1}$ tels que $\sum_{i=0}^{m-1} N_i = n$, on veut pouvoir calculer pour tout $0 < k < m$, $S_k = \sum_{i=0}^{k} N_i$. Ce problème peut être résolu en temps constant à l'aide des champs de bits. Soit B une séquence binaire dans laquelle chaque N_i est représenté par N_i zéros suivis d'un seul un. L'utilisation des structures présentées précédemment permet de calculer les opérations rank et select en temps constant sur ce champ de bits B. La somme S_k peut

maintenant être calculée ainsi avec l'opération $\mathsf{select}_1(B, k+1)+1$. Réciproquement, en utilisant la fonction rank, on peut connaître le nombre d'entiers nécessaire pour atteindre une valeur au moins égale à p : $\mathsf{rank}_1(B, p-1)+1$. Le champ de bits sera composé au plus de $2n + o(n)$ bits.

En utilisant une telle solution, il est possible de stocker une séquence de nombres entiers croissants. Un tel stockage offre un bon compromis entre l'espace de stockage utilisé et le temps nécessaire pour récupérer une valeur quelconque de la séquence. Considérons une suite de m entiers strictement positifs $N_0, \ldots, N_{m-1}$ telle que $N_0 \leq N_1 \leq \cdots \leq N_{m-1}$. On stockera alors sous forme de sommes partielles les entiers $N_0, N_1 - N_0, \ldots, N_{m-1} - N_{m-2}$. Ainsi le champ de bits n'est composé que de $N_{m-1} + m$ bits. Le calcul d'une somme partielle permet ensuite de récupérer n'importe quel nombre N_i en temps constant.

Exemple. Soit une suite croissante de nombres : 3, 9, 9, 15, 16, 19.
Les nombres à encoder seront alors : $3, 9-3, 9-9, 15-9, 16-15, 19-16$ soit 3, 6, 0, 6, 1, 3.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Le champ de bits B est :	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	1	0	0	0	1

2.2 Champs de lettres

Les opérations disponibles pour les champs de bits peuvent être également utiles lorsqu'on considère des alphabets plus grands. Nous nous intéressons au cas où la taille de l'alphabet de notre champ de lettres est strictement supérieur à deux. Dans ce cas les opérations rank et select doivent être disponibles pour toutes les lettres de l'alphabet.

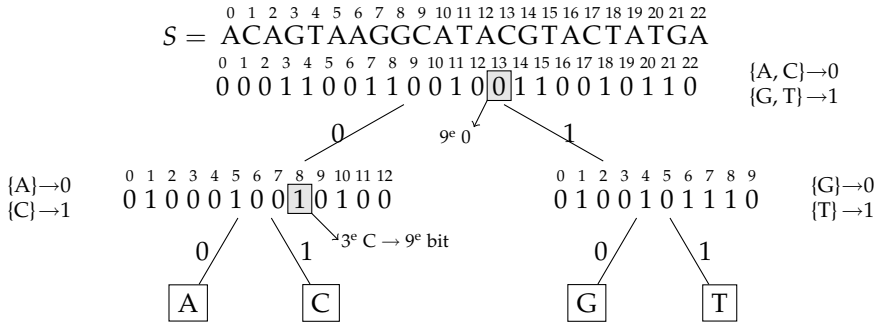
2.2.1 Wavelet tree

2.2.1.1 Approche originelle

Une première solution a été apportée par Grossi *et al.* (2003) et consiste à utiliser un arbre binaire équilibré. L'alphabet est découpé en deux parties égales. On attribue aux lettres de la première partie un 0 et aux lettres de la seconde partie un 1. La séquence d'origine est écrite, à la racine de l'arbre, en utilisant ce codage. Le processus est réitéré, dans le sous-arbre gauche, pour la première partie de l'alphabet et dans le sous-arbre droit, pour la seconde partie. L'alphabet est alors redécoupé en quatre (deux dans le sous-arbre droit, deux dans le gauche), et dans chaque sous-arbre la séquence correspondante est écrite. Le procédé continue jusqu'à ce que chaque sous-ensemble de l'alphabet soit un singleton, une feuille est alors créée, correspondant à une lettre de l'alphabet.

Avec les opérations rank et select sur les champs de bits de chaque nœud de l'arbre, on est capable d'avoir les opérations rank et select pour la séquence d'origine en temps $O(\log \sigma)$. L'opération $\mathsf{rank}_c(S, i)$ est calculée comme suit : on détermine si c est dans la première ou la seconde partie de l'alphabet. On décide, sans perte de généralité, qu'il est dans la première partie. Un 0 lui a donc été attribué. On calcule alors $j = \mathsf{rank}_0(B_0, i)$, où B_0 désigne le champ de bits à la racine. On se déplace dans le sous-arbre gauche, et il faut déterminer si c appartient à la première ou à la seconde partie du sous-alphabet, supposons

Ex. 2.3 — Wavelet tree et exemple de calcul de l'opération select



Calcul de $\mathsf{select}_C(S, 3)$: on part de la feuille C, la branche est étiquetée 1.
On cherche, dans le nœud parent, le troisième 1 → position 8.
La branche menant à ce nœud est étiquetée 0, on cherche le neuvième 0, dans le nœud parent (la racine), il est en position 13.
$\Rightarrow \mathsf{select}_C(S, 3) = 13$.

qu'il appartienne à la seconde. On calcule alors $j = \mathsf{rank}_1(B_1, i)$ où B_1 désigne le champ de bits dans le sous-arbre gauche de la racine. On continue le procédé jusqu'à arriver à une feuille, la valeur j obtenue à ce moment là sera le résultat de l'opération $\mathsf{rank}_c(S, i)$.

L'opération $\mathsf{select}_c(S, i)$ se déroule de manière symétrique : on part de la feuille correspondant à la lettre c, et on remonte dans l'arbre, en utilisant les opérations select sur les champs de bits, jusqu'à la racine (voir Ex. 2.3).

Le stockage de l'arbre étant réalisé avec des champs de bits, la complexité en espace est dépendante de la solution utilisée. Quoi qu'il en soit le nombre de bits présents dans l'ensemble des champs de bits est au plus de $n \log \sigma$. En utilisant des représentations compressées pour les champs de bits, on peut donc stocker un wavelet tree en utilisant $nH_0 + o(n \log \sigma)$ bits et calculer les opérations rank et select en temps $O(\log \sigma)$.

2.2.1.2 Arbres de Huffman

Mäkinen et Navarro (2005) ont eu l'idée d'utiliser, pour le stockage du wavelet tree, un arbre ayant la même structure qu'un arbre de Huffman plutôt qu'un arbre binaire équilibré. Un arbre de Huffman est tel que les caractères les plus présents dans le texte sont les plus proches de la racine. La distance à parcourir de la racine à une feuille est alors moins grande pour un caractère fortement présent que pour un caractère plus rare. Cette astuce permet de transformer le $\log \sigma$ des complexités en temps par un $H_0(S)$ en moyenne.

2.2.1.3 Run-length encoding

Nous avons vu (section 1.1.2.6 page 10) que la transformée de Burrows-Wheeler tend à rassembler les lettres identiques. Dans un wavelet tree, ceci implique des suites de bits identiques à tous les niveaux de l'arbre. C'est pour-

quoi Mäkinen et Navarro (2005) se sont intéressés au run-length encoding au sein des wavelet trees. Le but est de permettre une compression plus efficace de la TBW.

Pour cela, les champs de bits utilisés au sein du wavelet tree sont ceux permettant la compression par run-length encoding (section 2.1.2.3 page 47). Ainsi, nous obtenons immédiatement un wavelet tree compressé par run-length encoding.

2.2.1.4 Généralisation

Ferragina *et al.* (2007) ont proposé une généralisation du wavelet tree. Leur structure n'est plus un arbre binaire mais un arbre k-aire. Les séquences de chaque nœud interne ne sont donc plus définies sur un alphabet binaire mais sur un alphabet de taille k. Le stockage de telles séquences est expliquée dans la partie 2.2.2.

Leur wavelet tree généralisé occupe un espace en $nH_0(S) + o(n \log k)$ et une requête prend un temps $O(\log_k \sigma)$, si $k = \min(\sigma, \sqrt{n})$. En revanche, si on choisit $k = O(\log n/(\log\log n)^2)$, le wavelet tree occupe alors $nH_0(S) + O(\sigma \log n) + o(n \log \sigma)$ bits et les requêtes prennent un temps $O(\log \sigma / \log\log n)$. Par ailleurs, si $\sigma = O(\text{polylog}(n))$, on peut prendre $k = \sigma$ et répondre aux requêtes en temps constant.

2.2.2 Représentation par identifiants

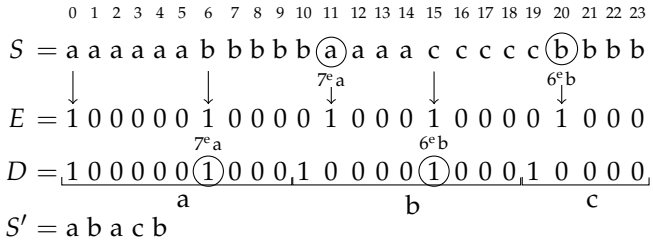
Le wavelet tree n'est pas la seule possibilité pour avoir des opérations rank et select pour des alphabets de taille supérieure à deux. Une autre solution, proposée par Ferragina *et al.* (2007), correspond à la généralisation de la méthode de (Raman *et al.*, 2002). Au lieu d'identifier un demi bloc par un couple (b, o), celui-ci est identifié par un σ-uplet. Ceci permet de stocker la séquence en $nH_0 + O(\sigma n \log\log n / \log_\sigma n)$ bits et d'avoir les opérations rank et select en temps constant. Cette méthode est donc intéressante uniquement lorsque $\sigma = O(\log n/(\log\log n)^2)$, ce qui peut poser des problèmes dans le cas des langages naturels avec des alphabets dépassant la centaine de lettres.

En revanche, coupler l'approche avec le wavelet tree généralisé permet de repousser les limites sur la taille de l'alphabet. Cette méthode est très intéressante lorsqu'elle est utilisée pour stocker la TBW. En effet, Mäkinen et Navarro (2008) ont prouvé que dans ce cas la transformée de Burrows-Wheeler occupe un espace de $nH_k(T) + o(n \log \sigma)$ bits, avec $k \leq (\alpha \log_\sigma n) - 1$ et pour toute constante $0 < \alpha < 1$. Ce résultat permet de stocker la transformée de Burrows-Wheeler de façon optimale asymptotiquement, ce qui n'était possible auparavant qu'en partitionnant la TBW et en compressant ces partitions séparément (Manzini, 2001). Des expériences de Claude et Navarro (2008) ont montré que cette méthode de stockage permet effectivement de réduire l'espace mémoire utilisé pour le stockage de la structure d'indexation.

2.2.3 Run-length encoding

Le run-length encoding peut être utilisé au sein des wavelet trees (section 2.2.1.3 page précédente) mais aussi indépendamment de ceux-ci. Mäkinen *et al.* (2009) introduisent cette technique afin de stocker efficacement des

Ex. 2.4 — Stockage et calcul de l'opération rank avec un champ de lettres compressé par run-length encoding



Calcul de $\mathsf{rank}_{\mathsf{a}}(S, 13)$:

1. $\mathsf{rank}_1(E, 13) = 3$
2. $\mathsf{rank}_{\mathsf{a}}(S', 2) = 2$ et $S'[2] = \mathrm{a} \rightarrow$ il y a deux suites de a, avant la position 13, dont une qui n'est peut être pas complète.
3. On se rend au début de la deuxième suite de a, dans D : $\mathsf{select}_{\mathsf{a}}(D, 2) = 6$. Il y a donc six a dans la première suite.
4. $13 - \mathsf{select}_1(E, 3) = 11 \rightarrow$ il y a trois a du début de la deuxième suite de a jusqu'à la position 13.
5. $\Rightarrow \mathsf{rank}_{\mathsf{a}}(S, 13) = 6 + 3 = 9$.

séquences fortement similaires. Ils s'intéressent au run-length encoding afin de stocker la transformée de Burrows-Wheeler (section 1.1.2.6 page 10) qui, avec des textes fortement répétitifs, produit de longues suites de caractères identiques.

Soit S un texte contenant de longues suites de caractères identiques. Les auteurs introduisent une séquence S' dans laquelle n'est stockée que la première lettre de chaque suite de S. Un champ de bits E permet de connaître les positions des éléments de S qui ont été conservés dans S'. Un autre champ de bits D marque, dans l'ordre alphabétique, le rang dans S des lettres débutant une suite (voir Ex. 2.4). Les champs de bits sont stockés avec la technique du gap-encoding et la séquence S' est stockée dans un wavelet tree.

2.2.4 Gestion des alphabets de grande taille

Nous avons vu que les wavelet trees sont limités par la taille de l'arbre, si l'alphabet devient trop grand, la taille nécessaire pour stocker le seul arbre devient non négligeable par rapport à la séquence elle-même. Golynski *et al.* (2006) proposent une méthode particulièrement utile pour les alphabets de grande taille. Cette méthode repose sur les champs de bits et le stockage de permutations. Elle permet d'accéder à une valeur et de réaliser l'opération rank en temps $O(\log \log \sigma)$ et l'opération select en temps constant. Un autre

compromis est possible, dans ce cas l'accès se fait en temps constant, l'opération select en temps $O(\log\log\sigma)$ et l'opération rank en temps $O(\log\log\sigma\log\log\log\sigma)$. L'espace nécessaire pour le stockage de la structure est de $n\log\sigma + o(n\log\sigma)$ bits.

Leur méthode revient à utiliser σ champ de bits conceptuels (c-à-d. non réellement stockés), $B_0, \ldots, B_{\sigma-1}$, tels que $B_i[j] = 1$ si et seulement si T_j est la $i+1$-ème lettre de l'alphabet. Ces champs de bits sont divisés en blocs de longueur σ. Un bloc contient donc $\sigma \times \sigma$ bits.

Exemple. Soit $T = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7}{\mathrm{T\,T\,A\,G\,T\,C\,G\,C}}$
Note : Afin de garder un exemple suffisant représentatif, nous prenons des blocs de longueur $\sigma - 1$ au lieu de σ.

	C_0			C_1			C_2	
	0	1	2	0	1	2	0	1
A	0	0	1	0	0	0	0	0
C	0	0	0	0	0	1	0	1
G	0	0	0	1	0	0	1	0
T	1	1	0	0	1	0	0	0

Chaque bloc est stocké sous forme d'une permutation π_i et d'un champ de bits X_i. Pour chaque bloc, π_i enregistre la position des 1, de gauche à droite, de haut en bas. Ensuite, X_i donne, pour chaque ligne du bloc, le nombre k de 1 sur cette ligne. Dans X_i, k 1 sont alors stockés, suivis d'un 0.

Exemple.

i	π	X
0	201	1000110
1	201	0101010
2	10	010100

Enfin, ligne par ligne, pour chaque bloc (d'abord la première ligne du bloc 0, du bloc 1, ..., du bloc $\sigma - 1$, ensuite la deuxième ligne du bloc 0, du bloc 1, ...) on stocke le nombre de 1 en unaire, suivis d'un 0.

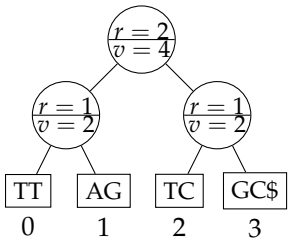
Exemple. $B = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\ 16\ 17\ 18\ 19}{1\,0\,0\,0\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,1\,1\,0\,1\,0\,0}$
$B[0\,..\,1] = 10$: il y a un 1 sur la première ligne du premier bloc.
$B[2] = 0$: il y a zéro 1 sur la première ligne du deuxième bloc.
$B[14\,..\,16] = 110$: il y a deux 1 sur la dernière ligne du premier bloc.

Le calcul des opérations rank et select utilisent les champs de bits et permutations π, X et B. Nous omettons, ici, l'explication pour calculer ces opérations. Cependant nous donnons un exemple du calcul de l'opération rank.

Exemple. Calcul de $\mathsf{rank}_{\mathrm{T}}(T, 4)$, avec $T = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7}{\mathrm{T\,T\,A\,G\,T\,C\,G\,C}}$
Nous avons trois blocs et $\sigma = 4$. La position 5 correspond à la troisième position dans le deuxième bloc.
Début du premier bloc (pour les T) : $d_1 = \mathsf{select}_0(B, 3 \times (\sigma - 1)) = 13$
Début du deuxième bloc (pour les T) : $d_2 = \mathsf{select}_0(B, 3 \times (\sigma - 1) + 1) = 16$
Déterminons le nombre de T du premier bloc : $\mathsf{rank}_1(B, d_2) - \mathsf{rank}_1(B, d_1) = 2$.
Déterminons le nombre de T jusqu'à la position 1 du deuxième bloc :
Début des T dans X_1 : $d_{X_1} = \mathsf{select}_0(X_1, 3) + 1 = 5$.

Ex. 2.5 — Structure dynamique pour les champs de lettres

$$T = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}{\mathrm{TTAGTCGC\$}}$$



r : nombre de feuilles dans le sous-arbre gauche.
v : nombre de feuilles dans le sous-arbre droit.

	Leaves			
	0	1	2	3
$S_\$$	0	0	0	1
S_A	0	1	0	0
S_C	0	0	1	1
S_G	0	1	0	1
S_T	2	0	1	0

Les sommes partielles $S_\$, \ldots, S_\mathrm{T}$ stockent le nombre d'occurrences de chaque lettre dans chaque feuille. Ces sommes partielles ne sont pas indépendantes les unes des autres mais sont gérées collectivement pour plus d'efficacité.

Calcul de $rank_\mathrm{T}(T, 5)$:

La lettre en position 5 dans T est la sixième lettre du texte.
Puisque $v = 4$ à la racine, la sixième lettre est dans le sous-arbre droit.
Dans le sous-arbre droit, $v = 2$. La sixième lettre est donc dans son sous-arbre gauche (feuille 2) et la lettre est la dernière dans cette feuille.
Dans cette feuille, il y a un T avant la sixième lettre du texte. $sum(S_\mathrm{T}, 1) = 2$, ce qui signifie que nous avons deux T dans les deux premières feuilles.
Finalement, $rank_\mathrm{T}(T, 5) = 3$.

Recherche dans $\pi_1[\mathsf{rank}_1(X_1, d_{x_1})\,..\,2]$, par dichotomie, du nombre le plus grand, inférieur ou égal à 1. Dans $\pi[2\,..\,2] = 1$, il y a un nombre qui est inférieur ou égal à 1. On a donc un T dans le deuxième bloc jusqu'à la position 1.
Ainsi $\mathsf{rank}_\mathrm{T}(T, 4) = 3$.

2.2.5 Mise à jour des champs de lettres

En plus de la solution, simple, qui consiste à utiliser un wavelet tree avec des champs de bits dynamiques, il existe d'autres solutions qui généralisent directement le principe des champs de bits dynamiques à un alphabet plus grand. La solution de González et Navarro (2008) est à ce jour la plus efficace. Le principe reste le même que celui décrit en section 2.1.3 page 48. Au lieu de stocker dans les nœuds le nombre de 1 présents dans le sous-arbre gauche c'est le nombre de lettres et de feuilles dans le sous-arbre gauche qui est stocké. De plus une collection de sommes partielles est utilisée afin de connaître le nombre d'occurrences de chacune des lettres dans chacune des feuilles (voir Ex. 2.5).

Dans ces conditions, chaque opération coûte, dans le pire des cas $O(\log n(1 + \log \sigma / \log \log n)$ et la structure occupe $nH_0(S) + o(n \log \sigma)$ bits.

2.2.5.1 Semi-dynamicité

Gupta *et al.* (2007b) proposent une solution originale adaptée aux cas où les modifications sont rares. Leur méthode permet de gérer un champ de lettres dynamique. Leur structure est composée d'un champ de lettres statique et de champs de bits et de lettres dynamiques (utilisant un espace négligeable). Le principe est d'enregistrer les modifications dans les structures dynamiques et de les répercuter sur la structure statique, avant que les structures dynamiques ne deviennent trop grosses. Cette structure permet d'accéder à une lettre et de supporter les opérations rank et select en temps $O(1)$, les insertions et suppressions sont gérées en temps amorti $O(n^\varepsilon)$, où $\varepsilon > 0$ est une constante permettant de déterminer le compromis espace-temps. L'espace utilisé par la structure de données est en $O(\sigma n \log n)$.

2.3 Conclusion

Les méthodes permettant de stocker des champs de bits ou de lettres sont variées. Il existe des solutions pour des cas variés (creux, suites de lettres, alphabets de grande taille) et la plupart sont compressées, permettant de tirer parti au maximum des séquences en entrée. S'il paraît difficile de faire mieux, en théorie, que les champs de bits compressés permettant de répondre aux requêtes en temps constant, il semble possible que d'autres structures soient améliorées. Par exemple les champs de lettres ne répondent pas toujours en temps constant aux requêtes. Par ailleurs dans le domaine des structures dynamiques les avancées sont encore très récentes et on peut probablement espérer encore quelques améliorations dans les prochaines années. Ainsi il existe des champs de lettres statiques offrant les mêmes complexités que les champs de bits (sous certaines conditions sur l'alphabet). On pourrait espérer la même chose pour les champs de lettres dynamiques, et ainsi avoir une structure qui répond aux requêtes en temps $O(\log n)$ au lieu de $O(\log n(1 + \log \sigma / \log \log n)$ pour la structure la plus récente.

Chapitre 3

Mise à jour des structures d'indexation

Les structures d'indexation sont généralement utilisées avec des textes qui ne sont qu'exceptionnellement modifiés. Cela est dû au temps ainsi qu'à l'espace mémoire non négligeables nécessaires pour la construction des ces structures. Il est alors rédhibitoire de reconstruire une structure d'indexation seulement en raison de quelques modifications dans le texte d'origine. C'est pourquoi des travaux, que nous présenterons en section 3.1, ont eu pour but de mettre à jour des structures d'indexation usuelles ou conçues à dessein. Dans la section 3.2.1 page 62, nous expliquons une méthode de mise à jour que nous avons mise au point pour le FM-index et qui vise à contrecarrer les limitations des méthodes existant à ce jour.

3.1 Méthodes existantes

La question de la dynamicité des structures d'indexation n'est pas récente et des solutions ont été proposées avant l'apparition des structures d'indexation compressées. Dans cette section, nous allons voir les méthodes proposées jusqu'à maintenant pour mettre à jour une structure d'indexation. Nous nous intéresserons dans un premier temps à une méthode introduite en 1995 pour l'arbre des suffixes. Cette méthode a par la suite été adaptée à d'autres structures d'indexation. Ensuite, nous verrons les solutions proposées permettant d'ajouter ou de supprimer un texte parmi une collection. Nous expliquerons aussi une technique de mise à jour utilisée pour l'inférence grammaticale. Enfin, nous terminerons avec une méthode très récente, présentée en 2009, utilisant l'arbre contracté des suffixes (cf. section 1.2.1.2 page 17).

3.1.1 Méthode « diviser pour régner »

3.1.1.1 Arbre des suffixes

Cette méthode a été introduite par Ferragina et Grossi (1995) pour les arbres compacts de suffixes. La méthode consiste à découper le texte en facteurs chevauchants de longueur $O(\sqrt{n})$. Un arbre des suffixes est construit sur chacun de ces facteurs. L'intérêt de cette méthode est que la mise à jour est

réalisée, dans tous les cas, en temps sublinéaire et le temps pour rechercher un motif est optimal.

Le texte d'origine est découpé en facteurs I_i de longueurs $\sqrt{n}$ jusqu'à $2\sqrt{n}$, avec $1 \leq i \leq k$ et $k = \Theta(\sqrt{n})$. À ces facteurs sont ajoutés deux facteurs spéciaux I_0 et I_{k+1} de longueurs $14\sqrt{n}$ et uniquement composés de \$. Pour chaque facteur I_i, $1 \leq i \leq k$, les facteurs suivants lui sont concaténés jusqu'à atteindre la longueur maximale, inférieure à $14\sqrt{n}$: $I_i I_{i+1} \cdots I_{i+j}$ tel que $|I_i I_{i+1} \cdots I_{i+j}| \leq 14\sqrt{n}$ et $|I_i I_{i+1} \cdots I_{i+j+1}| > 14\sqrt{n}$. Un arbre généralisé des suffixes est construit sur le résultat de ces concaténations. Le texte indexé est, au plus, de longueur $14k\sqrt{n}$. L'arbre généralisé nécessite, au mieux, 10 octets par lettre du texte. On comprend alors que la méthode est difficilement applicable à des textes de quelques dizaines de millions de caractères.

Ce seul arbre est suffisant pour faire des recherches de motifs de longueurs inférieures à $10\sqrt{n}$. Pour rechercher des motifs plus longs, il faut alors de nouveaux arbres généralisés des suffixes sur des préfixes et suffixes de concaténations de I_i. Il est également nécessaire de construire un arbre compact des suffixes sur le motif.

La mise à jour du texte modifie un ou plusieurs I_i, il est donc nécessaire de supprimer puis de ré-ajouter, dans l'arbre généralisé des suffixes, les concaténations de I_i qui ont été altérées par cette modification. Ainsi, quelle que soit la longueur de la modification, il est nécessaire de supprimer puis ajouter $O(\sqrt{n})$ caractères dans les arbres généralisés des suffixes. Ceci peut être rédhibitoire lorsqu'on effectue uniquement des petites modifications.

3.1.1.2 FM-Index

Lorsqu'ils ont introduit le FM-Index, Ferragina et Manzini (2000) ont également proposé une solution afin de gérer une collection dynamique de textes. Le principe était similaire à celui développé dans la méthode précédente : le texte est découpé en facteurs et ce sont ces facteurs qui sont indexés. Lorsque se produit une modification, les structures impactées par la modification sont à reconstruire, mais généralement la plupart sont conservées.

3.1.1.3 Arbre compressé des suffixes

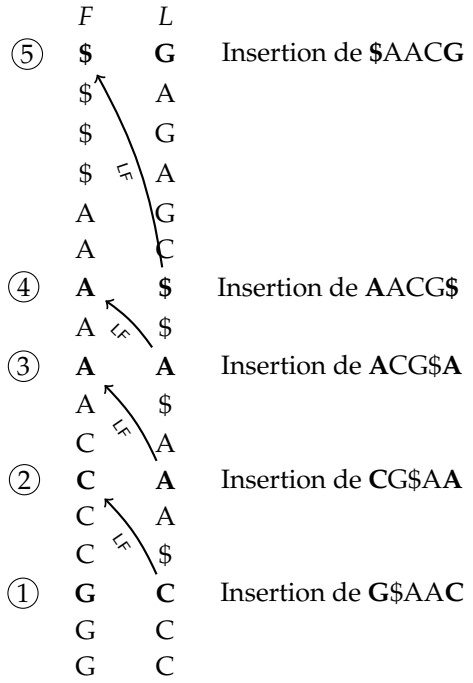
Hon *et al.* (2004) ont adapté la méthode de Ferragina et Grossi à l'arbre compressé des suffixes (cf. 1.3.2.4 page 38). Cette approche est la première à considérer des modifications pour une structure d'indexation compressée. Malgré cela, l'espace mémoire nécessaire pour stocker l'ensemble des structures reste important puisque la longueur de l'ensemble des textes indexés peut aller jusqu'à $28n$. Par ailleurs il n'existe pas d'implantation de leur méthode et on ne peut donc pas juger de son efficacité en pratique.

3.1.2 Pour une collection de textes

Chan *et al.* (2004) ont introduit la première structure d'indexation compressée pour laquelle on peut ajouter et supprimer des textes en temps directement dépendant de la longueur de la modification. Ce résultat a, par la suite, été amélioré par Mäkinen et Navarro (2008)

Ex. 3.1 — Insertion d'un texte dans une collection de textes

$T = \{\$ACA, \$CGA, \$ACG\}$, on insère $u = \$AACG$



Cette méthode repose sur le FM-index et plus particulièrement la fonction LF (cf. 1.3.2.1 page 28). Cette fonction est utilisée, dans le FM-index, pour la recherche de motifs. Ici, elle l'est également pour la mise à jour de la structure. Pour cette approche, le terminateur est placé au début du texte.

Soit T un ensemble de t textes $\{T^0, \ldots, T^{t-1}\}$ et B la transformée utilisée par la structure d'indexation. Notons bien que B n'est pas forcément $\mathsf{TBW}(T)$, nous décrivons donc le procédé pour la calculer.

On souhaite ajouter un texte $u = \$u_0 \cdots u_{n-1}$ à T. Les lettres du texte sont ajoutées à B de droite à gauche. La première permutation circulaire considérée est donc $u_{n-1}\$ \cdots u_{n-2}$ et sa position d'insertion est donnée par $\mathsf{First}(u_{n-1})$. On a alors $B[\mathsf{First}(u_{n-1})] = u_{n-2}$. Ensuite les positions d'insertion des permutations circulaires précédentes sont calculées avec la fonction LF. Ainsi, la permutation circulaire précédente $(u_{n-2}u_{n-1}\$ \cdots u_{n-3})$ est en position $\mathsf{LF}(\mathsf{First}(u_{n-1}))$. Le procédé se poursuit jusqu'à l'insertion de u_{n-1} dans B (voir Ex. 3.1).

La suppression, elle, se fait de la gauche vers la droite. Pour cela, il nous faut connaître la position de la permutation circulaire commençant par \$; c'est-à-dire la position la plus à gauche. Ceci correspond exactement au rang lexicographique du texte parmi l'ensemble des textes. Pour cela, une structure nous permet de connaître le rang lexicographique de chacun des textes. Nous avons deux arbres binaires équilibrés : un contenant l'ensemble des numéros attribués à chacun des textes et un autre stockant ces numéros dans l'ordre

lexicographique des textes (par un parcours préfixe gauche). Chaque nœud du premier arbre possède un pointeur vers un nœud du second arbre afin de retrouver rapidement le rang lexicographique d'un texte.

Les techniques d'échantillonnage restent semblables à celles décrites en section 1.3.2.1 page 31. La différence réside dans les structures utilisées qui doivent être dynamiques afin de permettre des mises à jour.

Les insertions et suppressions de textes sont réalisées en temps $O(n\, lf(|T|, \sigma))$. En ce qui concerne le temps de recherche, l'extraction de texte ou l'espace utilisé, les complexités restent les mêmes que pour le FM-index statique [1].

3.1.3 Pour l'inférence grammaticale

Gallé *et al.* (2008) s'intéressent à l'inférence grammaticale pour la modélisation de séquences génétiques. Dans ce contexte, on cherche à remplacer le facteur le plus répété $w \in \Sigma^*$ par une seule lettre, appartenant à un alphabet distinct Σ'. On considère toutes les lettres de l'alphabet Σ' strictement supérieures, dans l'ordre alphabétique, à celles de Σ. La recherche du facteur le plus répété est réalisé à l'aide de la table des suffixes et de la table PLPC. Après le remplacement, le texte est donc modifié et il faut que la table des suffixes et la table PLPC soit mise à jour.

La table des suffixes est représentée par une liste doublement chaînée afin de gérer efficacement les suppressions. Leur méthode consiste à trouver les occurrences de w dans la table des suffixes (en utilisant l'algorithme classique de recherche) et à les substituer par c, ce qui entraîne un déplacement des lignes représentant les occurrences à la fin de la table des suffixes. Ensuite, en utilisant la table inverse des suffixes, on trouve toutes les lignes correspondant à un suffixe de w, ces lignes doivent être supprimées de la table des suffixes. Enfin les suffixes de la forme vw, avec $v \in \Sigma^*$ doivent éventuellement être déplacés.

Leur algorithme est quadratique dans le pire des cas mais est plus efficace que la reconstruction de la table des suffixes, en pratique.

3.1.4 Arbre contracté des suffixes

En section 1.2.1.2 page 17, nous avons présenté l'arbre contracté des suffixes. Cette structure d'indexation supporte également les mises à jour. Considérons la suppression d'une lettre en position i dans le texte T.

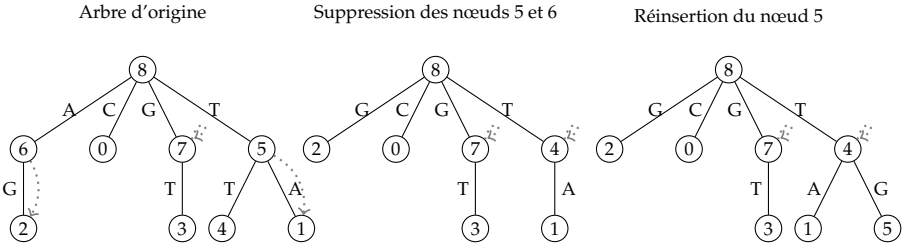
Dans un premier temps nous supprimons tous les nœuds utilisant cette lettre supprimée. Le nœud supprimé est remplacé par un de ses descendants direct. Ensuite, nous réinsérons tous les nœuds (sauf celui commençant par la lettre supprimée) en considérant le plus long facteur non présent dans l'arbre (voir Ex. 3.2 page suivante).

La hauteur d'un arbre contracté des suffixes est au maximum de $h(T)$, où $h(T)$ désigne la longueur de la plus longue répétition de T présente au moins $h(T)$ fois dans le texte. Ainsi le nombre de nœuds supprimés est, au maximum, $h(T)$ et il faut donc réinsérer $h(T) - 1$ nœuds. Chaque réinsertion peut

[1] À la différence près que la fonction LF est calculée en utilisant des structures dynamiques, ainsi les derniers résultats donnent $lf(n, \sigma) = \log n(1 + \log \sigma / \log \log n)$ pour de telles structures, voir section 2.2.5 page 54.

Ex. 3.2 — Mise à jour de l'arbre contracté des suffixes

$$T = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}{\mathrm{CTAGTTAG\$}} \rightarrow T' = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}{\mathrm{CTAGTT\ \ G\$}}$$



prendre un temps $h(T)$. Ainsi l'insertion ou la suppression d'un caractère se fait en temps $O([h(T)]^2)$.

3.2 Mise à jour d'un FM-index

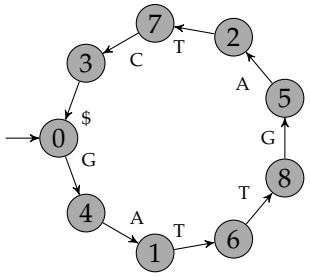
Nous avons vu les différentes solutions existant jusqu'à présent pour la mise à jour d'une structure d'indexation. Les méthodes basées sur les travaux de Ferragina et Grossi (1995) permettant n'importe quel type de modification ne peuvent être mises en œuvre simplement en raison de l'espace mémoire qu'elles consommeraient. De plus, toute modification, même mineure, nécessite une mise à jour en temps au moins $O(\sqrt{n})$, ce qui peut être rédhibitoire. L'arbre contracté des suffixes, quant à lui, semble être une approche intéressante mais leur implantation est en cours de réalisation, ne permettant pas une comparaison facile. Par ailleurs, sur le plan pratique, la consommation mémoire de leur structure risque d'être importante en raison des pointeurs nécessaires pour la recherche de motifs. Les autres solutions présentées n'autorisent pas n'importe quel type de modification sur le texte et, à ce titre, sont limitées dans l'usage qu'on peut faire de ces structures.

Nous présentons maintenant une solution que nous avons mise au point. Celle-ci est économique en espace, puisque basée sur une structure d'indexation compressée et autorise n'importe quel type de mise à jour. Plus précisément, nous nous intéressons à la mise à jour d'un FM-index autorisant la délétion, l'insertion ou la substitution d'un facteur dans un texte indexé par un FM-index. Nos travaux sont fondés sur ceux menés par Mäkinen et Navarro (2008) (cf. section 3.1.2 page 58) et Gallé *et al.* (2008) (cf. section 3.1.3 page précédente). Nous avons précédemment expliqué que le FM-index est composé de la transformée de Burrows-Wheeler et d'un échantillonnage de la table des suffixes. Toute mise à jour de cet index doit donc passer par la mise à jour de ses composantes. Nous allons commencer par expliquer la mise à jour de la TBW (section 3.2.1 page suivante) puis celle de la table des suffixes (section 3.2.2 page 72). Ces deux mises à jour permettent finalement de gérer celle du FM-Index (section 3.2.3 page 77).

Ex. 3.3 — Représentation de la fonction LF

$$T = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}{\mathrm{CTAGTTAG\$}}$$

i	F	L	LF
0	\$	G	4
1	A	T	6
2	A	T	7
3	C	\$	0
4	G	A	1
5	G	A	2
6	T	T	8
7	T	C	3
8	T	G	5



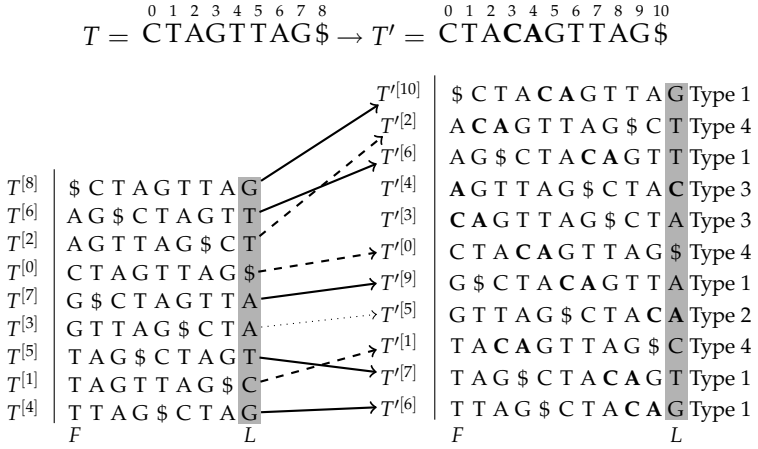
3.2.1 Mise à jour de la transformée de Burrows-Wheeler

Mäkinen et Navarro (2008) ont déjà présenté une façon de mettre à jour la TBW, cependant cette mise à jour n'est pas suffisante pour le cas qui nous intéresse. De plus le résultat qu'ils obtiennent après mise à jour n'est pas forcément la transformée de Burrows-Wheeler de la concaténation des textes qu'ils indexent. La modification d'un texte change l'ordre lexicographique de certaines permutations circulaires qui doivent alors être réordonnées. Ainsi les permutations circulaires du texte ACAG\$ sont, dans l'ordre lexicographique : \$ACAG, ACAG\$, AG\$AC, CAG\$A et G\$ACA ; la TBW est alors G\$CAA. Alors qu'en supprimant uniquement le G, on obtient le texte ACA\$, dont les permutations circulaires sont, dans l'ordre : \$ACA, A\$AC, ACA\$ et CA\$A ; la TBW est alors AC\$A. Le passage de la première TBW à la seconde ne paraît pas trivial au premier abord. Pour mettre à jour la TBW il faut donc être capable de savoir quelles sont les permutations circulaires à réordonner et calculer leurs nouvelles positions.

Pour la clarté de notre propos, nous introduisons une façon différente de représenter la fonction LF et la transformée de Burrows-Wheeler. La représentation est donnée sous forme d'automate dont l'état initial est l'état 0, correspondant à la première permutation circulaire $T^{[0]}$. Les transitions de l'automate sont de la forme $(i, L[i], \mathsf{LF}(i))$ où i et $\mathsf{LF}(i)$ sont les nombres représentant respectivement l'état de départ et l'état d'arrivée et $L[i]$ l'étiquette de la transition de i à $\mathsf{LF}(i)$ (voir Ex. 3.3). Dans la suite, un nombre j pourra aussi bien désigner un état de l'automate numéroté j que la position j dans L, les deux étant équivalents.

Dans la suite, nous considérerons que la mise à jour consiste en une insertion de ℓ lettres consécutives, en position i dans le texte. Soit $T' = T[..i-1]ST[i..]$, et S est le facteur inséré de longueur ℓ. La longueur de T' est $n + \ell + 1$ aussi notée $n' + 1$.

Ex. 3.4 — Caractérisation des différents types de permutations circulaires lors de la mise à jour d'un texte



Nous identifions quatre types exclusifs de permutations circulaires, en fonction de leur position de départ i dans le texte. Ces quatre types sont détaillés dans l'ordre de parcours de l'automate (voir aussi Ex. 3.4) :

Type 1 De $T'^{[n']}$ jusqu'à $T'^{[i+\ell+1]}$ (inclus), la modification apparaît strictement entre le \$ et la fin de la permutation circulaire. Nous démontrons dans le lemme 3.1 que ces permutations circulaires ne sont pas impactées par la modification opérée sur le texte. Cependant leurs positions peuvent changer mais uniquement en raison des déplacements, ajouts ou suppressions réalisés sur les autres permutations circulaires. Notons que plus l'insertion est proche du début du texte et plus ce type de permutations circulaires est prédominant diminuant d'autant le nombre de permutations circulaires susceptibles d'être réordonnées.

Type 2 $T'^{[i+\ell]}$, $S[\ell-1]$ apparaît en dernière position de cette permutation circulaire. Il s'en suit une modification de L (la transformée de Burrows-Wheeler) afin de prendre en compte cette nouvelle lettre. Néanmoins cette permutation circulaire n'est pas, non plus, sujette aux réordonnements.

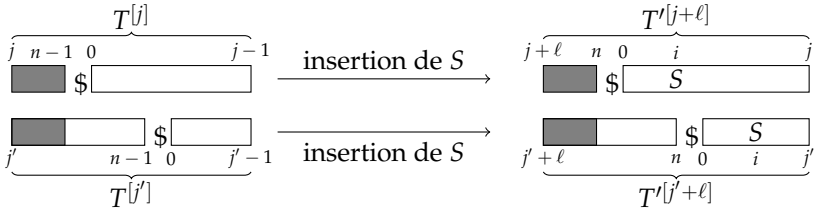
Type 3 $T'^{[i+\ell-1]}$ à $T'^{[i]}$, ces permutations circulaires commencent par une des lettres à insérer, elles n'existent pas encore et doivent donc être ajoutées au bon endroit.

Type 4 De $T'^{[i-1]}$ jusqu'à $T'^{[0]}$, la modification a lieu strictement entre le début de la permutation circulaire et \$. L'ordre de ces permutations est susceptible d'être modifié.

Lemme 3.1. *L'insertion d'une chaîne S de longueur ℓ, en position i dans T n'a pas d'effet sur l'ordre respectif des permutations circulaires suivant $T'^{[i+\ell]}$. C'est-à-dire que pour tout $j \geq i$ et $j' \geq i$, on a $T^{[j]} < T^{[j']} \iff T'^{[j+\ell]} < T'^{[j'+\ell]}$.*

Démonstration. Afin de prouver ce lemme, nous devons prouver que l'ordre lexicographique de deux permutations circulaires suivant $T^{[i]}$ est le même avant ou après l'insertion.

Supposons, sans perte de généralité, que $j > j'$ et $T^{[j]} < T^{[j']}$.
Nous savons que pour chaque $k < |T|$, $T^{[j]}[..k] \leq T^{[j']}[..k]$. C'est en particulier vrai pour le préfixe de $T^{[j]}$ qui se termine une position avant le terminateur \$ (de longueur $n - j < |T|$). Ainsi, $T[j..n-1] \leq T[j'..j'+n-j-1]$ (rectangles gris dans la figure qui suit). De plus, \$, la plus petite lettre de Σ, n'apparaît qu'une seule fois dans T. Le fait que $T[j+n-j] = \$$ induit $T[j'+n-j] \neq \$$, et donc $T[j'+n-j] > \$$. Il s'en suit que $T^{[j]}[0..n-j] < T^{[j']}[0..n-j]$.



Sachant que $T'[j+\ell..n+\ell-1]\$ = T[j..n-1]\$$ et $T'[j'+\ell..n+\ell+j'-j] = T[j'..n+j'-j]$, on a $T'[j+\ell..n+\ell-1]\$ < T'[j'+\ell..n+\ell+j'-j]$. D'où $T'[j+\ell..n+\ell-1]\$u < T'[j'+\ell..n+\ell+j'-j]v$, pour tous les textes u, v définis sur Σ. Finalement, $T^{[j]} < T^{[j']} \implies T'^{[j+\ell]} < T'^{[j'+\ell]}$.

La preuve de $T'^{[j+\ell]} < T'^{[j'+\ell]} \implies T^{[j]} < T^{[j']}$ est similaire. □

La représentation par automate du texte d'origine T et du texte modifié T' permet de bien identifier les différents types de permutations circulaires et l'impact de chacune sur la transformée de Burrows-Wheeler (cf. Ex. 3.5 page ci-contre). Cependant, au premier abord, il semble que les états correspondant aux permutations circulaires de type 1 soient aussi impactés que les autres états. Les changements de numérotations sont en fait dûs aux décalages induits par les changements de numéros des autres états.

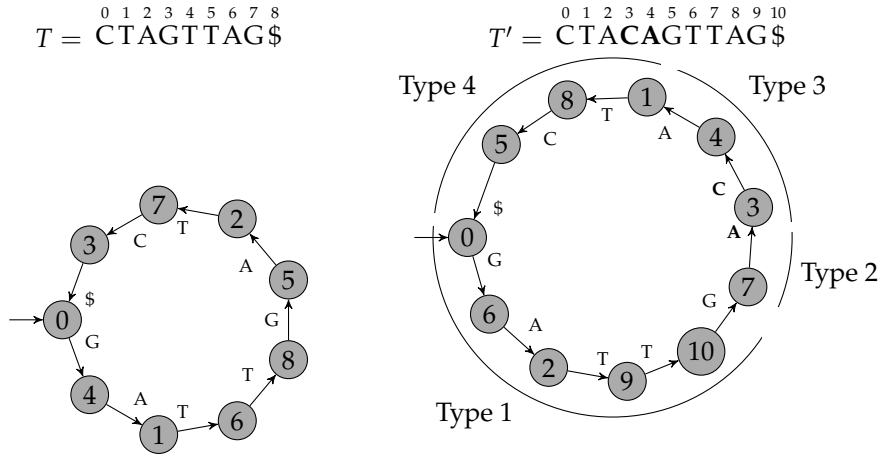
L'automate possède une propriété héritée de la propriété LF. Pour cela nous introduisons l'ensemble Q_c qui correspond à l'ensemble des couples d'états reliés par une transition étiquetée c. On note $Q_c = \{(d_0, f_0), \ldots, (d_{|T|_c-1}, f_{|T|_c-1})\}$.

Propriété 3. On a $\forall (d_j, f_j) \in Q_c, (d_{j'}, f_{j'}) \in Q_c, j \neq j'\ d_j < d_{j'} \Rightarrow f_j < f_{j'}$. Si les couples sont triés en fonction des d_j, ils sont également triés en fonction des f_j. De plus, s'ils sont triés, la différence entre deux f_j consécutifs est toujours de 1.

Démonstration. Par construction, nous savons que, pour un couple $(d_j, f_j)_c$, d_j correspond à une position k dans L et f_j correspond à $\mathsf{LF}(k)$. Si on a $d_j < d_{j'}$, c'est-à-dire $k < k'$, on peut en déduire, par définition de la fonction LF, que $\mathsf{LF}(k) < \mathsf{LF}(k')$, sachant que $L[k] = L[k'] = c$. Ce qui prouve que $d_j < d_{j'} \Rightarrow f_j < f_{j'}$.

De plus, si les couples sont triés et qu'on en prend deux consécutifs $(d_j, f_j)_c$ et $(d_{j'}, f_{j'})_c$, on sait qu'il n'y a aucun c dans L entre les positions d_j et $d_{j'}$.

Ex. 3.5 — Automates correspondant à la transformée du texte d'origine et du texte modifié



Ainsi $f_{j'} = \mathsf{LF}(d_{j'}) = \mathsf{First}(c) + \mathsf{rank}_c(L, d_{j'}) = \mathsf{First}(c) + \mathsf{rank}_c(L, d_j) + 1 = \mathsf{LF}(d_j) + 1 = f_j + 1$. □

Exemple. Dans l'Ex. 3.5, vérifions la propriété sur les transitions par A de l'automate de T' :

d_j	f_j
4	1
6	2
7	3

Les d_j et f_j sont bien triés et la différence entre deux f_j consécutifs est de 1.

Cette représentation par automate permet également de faire le lien entre le FM-index et la table compressée des suffixes.

Propriété 4. La suite donnée par la première composante d_j des couples triés en fonction des f_j, $(d_j, f_j)_c$, pour tous les $c \in \Sigma$ est la fonction Ψ de la table compressée des suffixes.

Exemple. Nous reprenons l'exemple du texte T' dans l'Ex. 3.5.

f_j	0	1	2	3	4	5	6	7	8	9	10
d_j	5	4	6	7	3	8	0	10	1	2	9
$L[d_j]$	$	A			C		G		T		

$\mathsf{TS} = \underset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10}{1\,0\,2\,8\,4\,3\,0\,9\,5\,1\,7\,6}$

$\Psi = \underset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10}{5\,4\,6\,7\,3\,8\,0\,1\,0\,1\,2\,9}$

Démonstration. Appelons $u_0, \ldots, u_n$ une telle suite. Prenons un u_k quelconque, avec $0 \leq k \leq n$. Ce u_k fait partie d'une transition $(u_k, L[u_k], \mathsf{LF}(u_k))$, on a donc un couple $(d_j, f_j) = (u_k, \mathsf{LF}(u_k))$. Or, on a $k = \mathsf{LF}(u_k)$, car les d_j sont triés en

fonction de $L[d_j]$ et en fonction des d_j croissants pour une même lettre. C'est-à-dire qu'on a les couples $(u_0, 0)(u_1, 1) \cdots (u_n, n)$.

$$k = \mathsf{LF}(u_k) \iff \mathsf{LF}^{-1}(k) = u_k \iff \Psi(k) = u_k$$

□

La propriété 3 est utilisée pour la mise à jour de l'automate comme suit :

Type 1 Aucune modification à faire.

Type 2 Aucune modification à faire.

Type 3 Ajout de l'état, afin qu'il soit le $n + 2 - i$-ème en partant de l'état initial. La transition du $n + 2 - i - 1$-ème état au nouvel état est étiquetée par $S[\ell - 1]$. Le numéro du nouvel état est attribué de telle façon à ce que la propriété 3 soit vérifiée. Les états suivants sont ajoutés à la suite de ce nouvel état et les étiquettes attribuées vont de $S[\ell - 2]$ à $S[0]$.

Type 4 À la suite du dernier état ajouté, on vérifie si la propriété est toujours bien respecté avec l'état suivant. Tant que la propriété n'est pas respectée, et donc qu'il y a des renumérotations à faire, on poursuit le parcours de l'automate pour mettre à jour les états.

Pour un exemple, voir Ex. 3.6 page suivante.

Les changements de numérotation réalisés sur les états de l'automate correspondent à des déplacements de permutations circulaires, dûs aux changements d'ordre lexicographique. Nous avons vu le lien entre l'automate et la fonction LF et utilisons cela pour en déduire la façon dont sont modifiés F et L lors d'une mise à jour. La fonction LF est utilisée afin de calculer directement les nouvelles positions des éléments. Les permutations circulaires sont considérées de droite à gauche. Intéressons-nous aux modifications à réaliser en fonction des quatre types de permutations circulaires.

Type 1 Pas de modification.

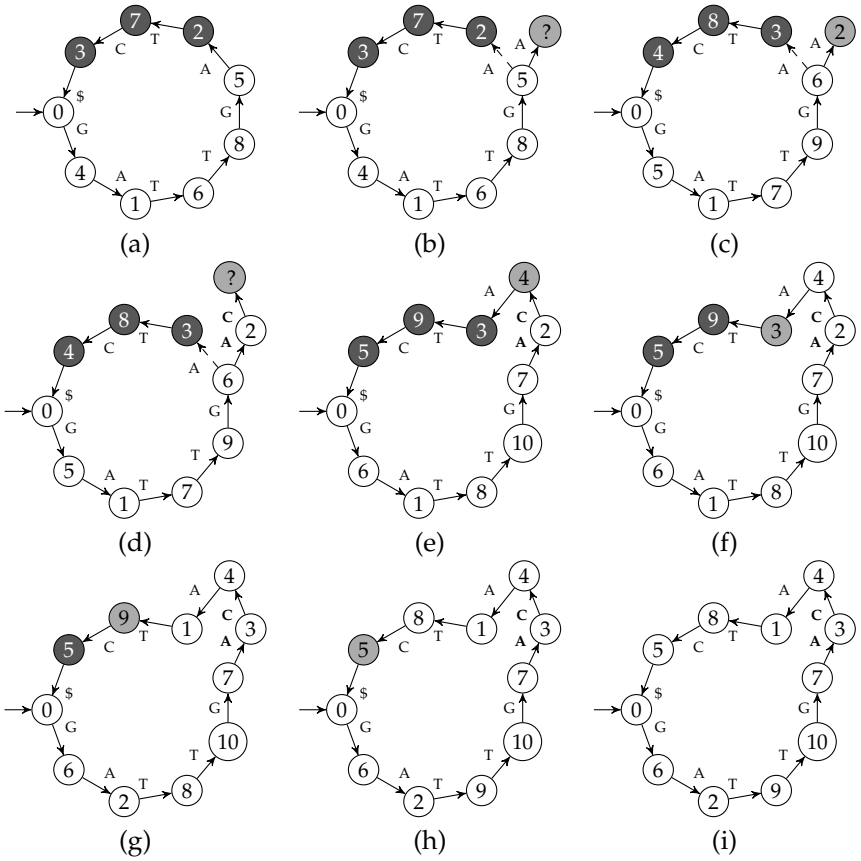
Type 2 $S_{\ell-1}$ remplace la lettre pré-existante, c, dans L à la position $\mathsf{TIS}[i] = k$.

Type 3 Pour $j = 2$ à ℓ : insertion, en position $\mathsf{LF}(k)$, de $S_{\ell-j+1}$ dans F et de $S_{\ell-j}$ dans L ; $k \leftarrow \mathsf{LF}(k)$.
Insertion, en position k, de S_0 dans F et de c, dans L.

Type 4 Calcul de la position de $T^{[i-1]}$ en utilisant la position k_i de $T^{[i]}$ ($i^{\text{ème}}$ permutation circulaire du texte d'origine). La position est calculée avec un $\mathsf{LF}(k_i)$ modifié noté $\widetilde{\mathsf{LF}}_c(k_i) = \mathsf{rank}_c(L, k_i) + \mathsf{First}(c) - 1$, où le $L[k_i]$ présent dans la formule du LF est remplacé par c. Ceci permet de calculer la position de la permutation précédente sans prendre en compte la modification réalisée, c'est-à-dire retrouver la position de $T^{[i-1]}$, notée k_{i-1}. Il reste maintenant à calculer la nouvelle position de $T^{[i-1]}$ c'est-à-dire celle correspondant au texte modifié et donc à $T'^{[i-1]}$. Pour cela, nous utilisons de nouveau la fonction LF (non modifiée cette fois) sur la permutation circulaire $T'^{[i]}$, dont la position est k'_i. Par définition, nous avons $k'_{i-1} = \mathsf{LF}(k'_i)$. Si k_{i-1}, la position réelle de $T^{[i-1]}$, est différente de k'_{i-1}, celle de $T'^{[i-1]}$, il nous faut donc déplacer l'élément dans L de la position k_{i-1} à la position k'_{i-1}.

Ex. 3.6 — Mise à jour de l'automate, étape par étape

$$T' = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10}{\mathrm{CTA\mathbf{CA}GTTAG\$}}$$



(b) La seule autre transition par A est celle de 4 vers 1. Partant de 5, on doit donc avoir un 2. Les autres nombres supérieurs ou égaux à 2 sont incrémentés.

(d) Les transitions par C sont (2, ?) et (8,4). $2 < 8$, le 4 lui est donc attribué, (8,4) devient (8,5). Les autres nombres supérieurs à 4 sont incrémentés.

(f) Les transitions par A sont (4,3), (6,1), (7,2). Le premier couple doit avoir le f_i le plus petit. Les couples mis à jour sont alors (4,1), (6,2) et (7,3).

(g) Les transitions par T sont (1,9), (2,8) et (8,10). Pour que la propriété 3 page 64 soit vérifiée, on échange le 8 et le 9.

(h) Les transitions par C sont (3,4) et (8,5). La propriété est respectée, il n'y a pas de changement à réaliser.

Algo. 3.1 — Réordonnement des permutations circulaires de type 4

1: **procédure** RÉORDONNEMENT(L, k_{i-1}, k'_i)
Requiert L : la transformée de Burrows-Wheeler quoi doit être réordonnée
Requiert k_{i-1} : la position actuelle de $T^{[i-1]}$ dans la matrice conceptuelle, sans que la modification réalisée ne soit encore prise en compte pour son ordre lexicographique.
Requiert k'_i : la position de la dernière permutation circulaire insérée.
Assure que les permutations circulaires de la matrice conceptuelles sont toutes triées dans l'ordre lexicographique.
2: $j \leftarrow k_{i-1}$
3: $j' \leftarrow \mathsf{LF}(k'_i)$
4: **tant que** $j \neq j'$ **faire**
5: nouveau_j $\leftarrow \mathsf{LF}(j)$
6: DÉPLACELIGNE(j, j') ▷ Déplace l'élément dans L de la position j à j'
7: $j \leftarrow$ nouveau_j
8: $j' \leftarrow \mathsf{LF}(j')$
9: **fin tant que**
10: **fin procédure**

Le processus est répété de façon similaire pour les permutations circulaires précédentes jusqu'à ce que la position réelle de la permutation circulaire corresponde à la position calculée. L'algorithme de réordonnement est décrit en Algo. 3.1. L'algorithme de mise à jour pour l'insertion d'un facteur est décrit en Algo. 3.2 page ci-contre.

Nous montrons maintenant que l'algorithme de réordonnement est correct. Pour ce faire, nous introduisons la notation $index(T^{[i]})$, qui correspond à k_i. Cette fonction donne la position d'une permutation circulaire dans la matrice conceptuelle de la transformée de Burrows-Wheeler.

Lemme 3.2. $\forall j < i, \forall j' > j, T'^{[j]} < T'^{[j']} \iff index(T'^{[j]}) < index(T'^{[j']})$, *après l'itération qui considère* $T^{[j]}$, *durant* REORDER.

Démonstration. Nous prouvons le lemme récursivement pour tout $j \leq i + \ell$.

À partir du lemme 3.1 page 63, nous savons que $\forall j' \geq i + \ell$ on a $T'^{[i+\ell]} < T'^{[j']} \iff T^{[i]} < T^{[j'-\ell]}$. Par définition, le lemme qu'on veut prouver est vrai pour tout j, pour le texte T et la TBW originelle. Ainsi, $T'^{[i+\ell]} < T'^{[j']} \iff index(T^{[i]}) < index(T^{[j'-\ell]})$.
Ni $T'^{[i+1]}$ ni $T'^{[j']}$ ne peuvent être déplacés par l'algorithme de réordonnement, on a alors $index(T'^{[i+\ell]}) < index(T'^{[j']}) \iff index(T^{[i]}) < index(T^{[j'-\ell]}) \iff T'^{[i+\ell]} < T'^{[j']}$.

Nous avons montré que le lemme est vrai pour $j = i + \ell$, maintenant prouvons-le récursivement pour $j - 1$.
Par définition, $T'^{[j-1]}_0 = T'^{[j]}_{n+\ell}$, posons $r = \mathsf{rank}_{T'^{[j]}_{n+\ell}}(L, index(T'^{[j]}))$. La position

Algo. 3.2 — Mise à jour de la transformée de Burrows-Wheeler pour l'insertion d'un facteur

1: **procédure** MISEÀJOUR(L, S, ℓ, i)
Requiert L : la transformée de Burrows-Wheeler quoi doit être mise à jour
Requiert On souhaite insérer dans le texte T (dont la TBW est L) une chaîne S de longueur ℓ en position i.
Requiert $\ell \geq 1$ et $0 \leq i < |L| - 1$
2: $k \leftarrow \mathsf{TIS}[i+1]$
3: $p \leftarrow k$
4: $c \leftarrow L[p]$
5: $L[p] \leftarrow S[\ell - 1]$
6: **pour** $j = \ell - 2$ à 0 **faire**
7: $k = \mathsf{LF}(k)$
8: INSÈRE($L, k, S[j]$) ▷ Insère $S[j]$ dans L à la position k
9: **si** $k \leq p$ **alors**
10: $p \leftarrow p + 1$
11: **fin si**
12: **fin pour**
13: $k = \mathsf{LF}(k)$
14: INSÈRE(L, k, c)
15: **si** $k \leq p$ **alors**
16: $p \leftarrow p + 1$
17: **fin si**
18: RÉORDONNEMENT(L, $\widetilde{\mathsf{LF}}_c(p)$, k)
19: **fin procédure**

de $T'^{[j-1]}$ est calculée avec la fonction LF en utilisant la formule suivante : $\mathit{index}(T'^{[j-1]}) = \mathsf{First}(T_0'^{[j-1]}) + r - 1$. Nous distinguons deux cas :

- si la première lettre de $T'^{[j-1]}$ est différente de la première de $T'^{[j']}$, alors $\mathsf{First}(T_0'^{[j-1]}) \neq \mathsf{First}(T_0'^{[j']})$. Sans perte de généralité, considérons que $T_0'^{[j-1]} < T_0'^{[j']}$. Par définition, $r \leq \mathsf{First}(T_0'^{[j']}) - \mathsf{First}(T_0'^{[j-1]})$. Ainsi, $\mathsf{First}(T_0'^{[j-1]}) + r - 1 \leq \mathsf{First}(T_0'^{[j']}) - 1$. Le membre de gauche correspond à $\mathit{index}(T'^{[j-1]})$, le membre de droite, quant à lui, correspond à $\mathit{index}(T'^{[j']})$ moins la valeur du rank lui correspondant. Cependant, ce rank est forcément strictement positif. On en déduit que $T_0'^{[j-1]} < T_0'^{[j']} \implies \mathit{index}(T'^{[j-1]}) < \mathit{index}(T'^{[j']})$.
- sinon, les deux lettres sont égales. On peut alors écrire $T'^{[j-1]} < T'^{[j']} \iff T'^{[j-1]}[1\,..\,n+\ell] < T'^{[j']}[1\,..\,n+\ell] \iff T'^{[j-1]}[1\,..\,n+\ell]T_0'^{[j-1]} < T'^{[j']}[1\,..\,n+\ell]T_0'^{[j']} \iff T'^{[j]} < T'^{[j'+1]}$. On sait que le lemme est vrai pour j, on a donc $T'^{[j]} < T'^{[j'+1]} \iff \mathit{index}(T'^{[j]}) < \mathit{index}(T'^{[j'+1]})$.
  Posons $k = \mathit{index}(T'^{[j]})$, $k' = \mathit{index}(T'^{[j'+1]})$ et $c = T_0'^{[j-1]} = T_0'^{[j']}$.

$$\begin{aligned} index(T'^{[j-1]}) &= \mathsf{First}(c) + \mathsf{rank}_c(L,k) - 1 \\ index(T'^{[j']}) &= \mathsf{First}(c) + \mathsf{rank}_c(L,k') - 1 \end{aligned}$$

Nous savons que $T'^{[j]}_{n+\ell} = L_k = c$, $T'^{[j'+1]}_{n+1} = L_{k'} = c$ et $k < k'$. Donc $\mathsf{rank}_c(L,k) < \mathsf{rank}_c(L,k')$ et finalement $index(T'^{[j-1]}) < index(T'^{[j']})$.
On en conclut que $T'^{[j-1]} < T'^{[j']} \implies index(T'^{[j-1]}) < index(T'^{[j']})$.
On peut prouver que $T'^{[j-1]} < T'^{[j']} \impliedby index(T'^{[j-1]}) < index(T'^{[j']})$ de façon similaire.

Ainsi, si la propriété est vrai pour k, elle est également vraie pour $j-1$. Finalement, quand l'algorithme se termine (avec $j = 0$), on a $\forall j, j', \quad T'^{[j]} < T'^{[j']} \iff index(T'^{[j]}) < index(T'^{[j']})$. Autrement dit, à la fin de l'algorithme, les permutations circulaires sont ordonnées. □

Il nous reste à démontrer qu'il n'est pas forcément indispensable de traiter toutes les permutations circulaires de type 4 jusqu'à $T'^{[0]}$. La condition d'arrêt de la boucle de l'Algo. 3.1 page 68 repose uniquement sur le fait que la position réelle de la permutation et la position calculée coïncident. Cette condition est effectivement suffisante pour que toutes les permutations circulaires précédentes soient également correctement positionnées.

Lemme 3.3. *$index(T^{[k]}) = index(T'^{[k]}) \implies index(T^{[j]}) = index(T'^{[j]})$, pour $j < k < i$.*

Démonstration. On utilise l'hypothèse de départ : $index(T^{[k]}) = index(T'^{[k]})$,

$$\begin{aligned} index(T^{[k-1]}) &= \mathsf{First}(T^{[k]}_n) + \mathsf{rank}_{T^{[k]}_n}(L, index(T^{[k]})) \\ &= \mathsf{First}(T'^{[k]}_{n+\ell}) + \mathsf{rank}_{T'^{[k]}_{n+\ell}}(L, index(T'^{[k]})) = index(T'^{[k-1]}) \end{aligned}$$

Donc, $index(T^{[k]}) = index(T'^{[k]}) \implies index(T^{[k-1]}) = index(T'^{[k-1]})$.
Par induction, on prouve que la propriété est vraie pour chaque $j < k$. □

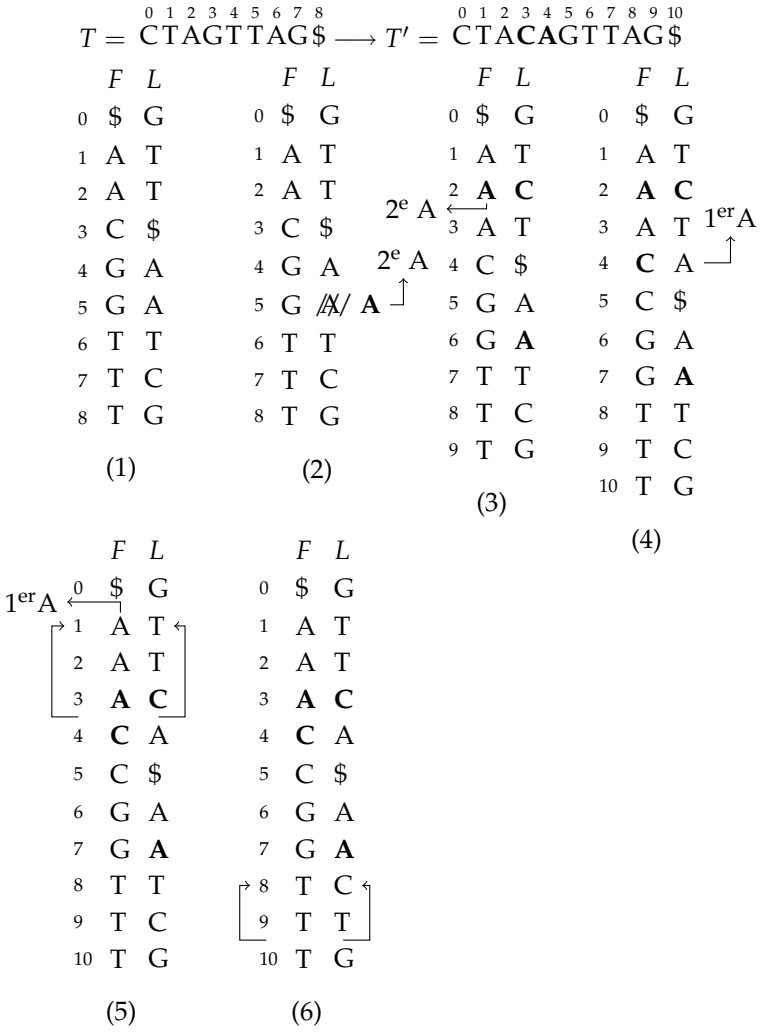
Théorème 3.1. *L'algorithme 3.2 page précédente procède à la mise à jour de la transformée de Burrows-Wheeler lors de l'insertion d'un facteur S de longueur ℓ.*

Démonstration. À partir des lemmes 3.1 page 63, 3.2 page 68 et 3.3, nous avons prouvé que l'algorithme de mise à jour, et plus particulièrement celui de réordonnement est correct. □

Dans l'Ex. 3.7 page suivante nous reprenons l'exemple de mise à jour réalisé précédemment sur l'automate et l'appliquons à la transformée de Burrows-Wheeler, elle-même.

La mise à jour sur la transformée de Burrows-Wheeler nous permet d'envisager la mise à jour sur la table des suffixes, les deux structures étant très proches par construction (tri lexicographique des suffixes d'un côté et des permutations circulaires de l'autre).

Ex. 3.7 — Mise à jour de la transformée de Burrows-Wheeler



(2) On modifie L en substituant la nouvelle lettre à l'ancienne. Il s'agit du 2^{e} A dans L.

(3) La nouvelle permutation circulaire doit être la 2^{e} commençant par un A.

(4) Le A dans L_4 est celui qui a été substitué à l'étape (2). On note qu'il s'agit maintenant du 1er A dans L alors qu'avant modification il était le dernier.

(5) La dernière permutation commençant par A devient la première.

(6) En raison du déplacement à l'étape (5) le deuxième T est devenu le premier, d'où le déplacement réalisé à cette étape (la deuxième permutation commençant par T devient la première).

3.2.2 Mise à jour de la table des suffixes

Nous savons que la table des suffixes et la transformée de Burrows-Wheeler sont conceptuellement proches. Il est alors tentant d'essayer d'adapter l'algorithme de mise à jour de la TBW à la table de suffixes. Pour cela il faudrait avoir un équivalent à la fonction LF pour la table des suffixes. Nous avons vu, en section 1.3.2.2 page 33, qu'il existe la fonction Ψ mais celle-ci permet de se déplacer de gauche à droite dans le texte et non de droite à gauche comme la fonction LF. Pour mettre à jour la table des suffixes, en utilisant le même principe que pour la mise à jour de la TBW, nous avons besoin de la fonction LF et donc de la TBW.

Une fois ce constat fait, la mise à jour de la table des suffixes est relativement directe : il suffit d'appliquer les mêmes changements à la TBW qu'à la table des suffixes. De façon plus détaillée voici les différentes modifications à appliquer à la table de suffixes en considérant les quatre types de permutations circulaires.

Type 1 Aucune modification directe.

Type 2 Aucune modification directe.

Type 3 Pour $j = 1$ à ℓ : à la position d'insertion dans la TBW, insertion dans la table des suffixes de i.

Type 4 Aux mêmes positions que dans TBW les éléments de la table des suffixes sont déplacés.

Notons que l'insertion de i dans la table des suffixes doit incrémenter toutes les valeurs déjà présentes qui sont supérieures ou égales. La table inverse des suffixes pouvant également être utile, nous donnons également les étapes à suivre pour cette modification :

Type 1 Aucune modification directe.

Type 2 Aucune modification directe.

Type 3 Pour $j = 1$ à ℓ : soit k la position d'insertion dans la TBW, insertion en position i dans la table inverse des suffixes de k.

Type 4 Soit un élément de la TBW qui est déplacé de j à j'. Sans perte de généralité, on considère $j < j'$. Toutes les valeurs comprises entre j (exclus) et j' (inclus) sont décrémentées. Dans la table inverse des suffixes, j est modifié en j'.

L'Ex. 3.8 page suivante montre la mise à jour de la table des suffixes, ainsi que son inverse, sur les textes usuels.

3.2.2.1 Mise à jour d'une permutation

Les mises à jour de la table des suffixes ou de sa table inverse impliquent de nombreuses incrémentations. Ces opérations très simples risquent d'accroître significativement le temps d'exécution de l'algorithme si elles deviennent trop nombreuses. Afin de gérer efficacement ces incrémentations, nous introduisons une structure autorisant les mises à jour des valeurs d'une permutation (telle que la table des suffixes).

Considérons une permutation π définie sur $\{1, \ldots, n\}$.

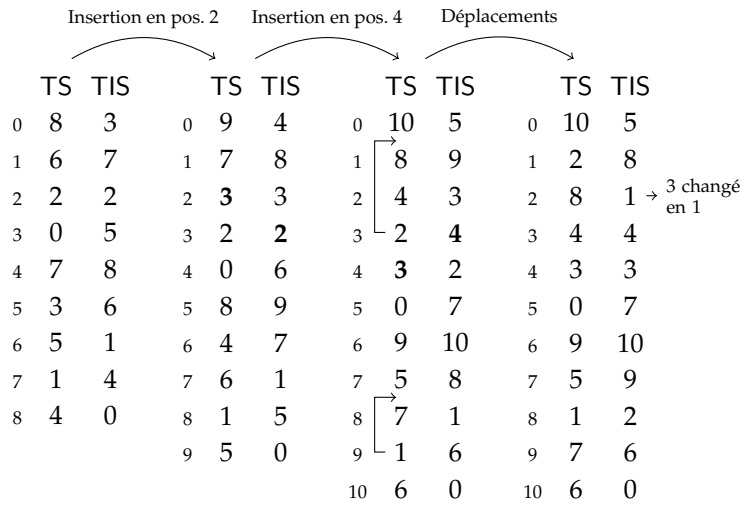
1. Après l'insertion d'une valeur i en position j dans π, la permutation résultante π' est définie sur $\{1, \ldots, n+1\}$ comme suit :

Ex. 3.8 — Exemple de mise à jour de la table des suffixes et de la table inverse

$$T = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}{\mathrm{CTAGTTAG\$}} \longrightarrow T' = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10}{\mathrm{CTA\mathbf{CA}GTTAG\$}}$$

Pour la mise à jour de T en T' les modifications réalisées sur la TBW ont été les suivantes :

1. insertion en position 2 ;
2. insertion en position 4 ;
3. déplacement de la position 3 à la position 1 ;
4. déplacement de la position 9 à la position 8.

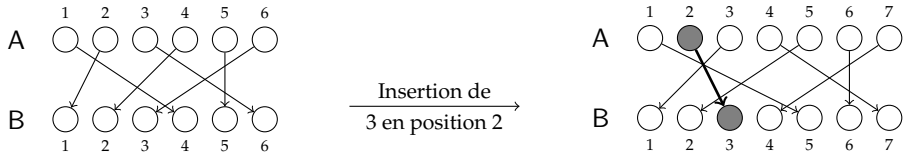


- toutes les valeurs supérieures ou égales à i dans π sont incrémentées d'un dans π' ;
- la valeur i est insérée en position j.

2. Après la suppression d'un élément en position j dans π, dont la valeur est i, la permutation résultante π' est définie sur $\{1, \ldots, n-1\}$ comme suit :
   - l'élément en position j est supprimé,
   - toutes les valeurs supérieures à i sont décrémentées d'un dans π'.

Pour résoudre ce problème, on considère deux structures A et B contenant n nœuds. On représente par $\mathsf{A}[j]$ le $j^{\text{ème}}$ nœud dans A et $\mathsf{B}[i]$ le $i^{\text{ème}}$ nœud dans B. Nous définissons une permutation π de n éléments à partir de A et B comme suit : $\pi(j) = i$ si et seulement si un lien est défini du nœud $\mathsf{A}[j]$ au nœud $\mathsf{B}[i]$.

Ex. 3.9 — Insertion d'un élément dans une permutation



$\pi = [4, 1, 6, 2, 5, 3]$ $\pi' = [5, \mathbf{3}, 1, 7, 2, 6, 4]$

Insertions et suppressions L'insertion de la valeur i en position j est réalisée par l'insertion d'un nœud $\mathsf{A}[j]$, d'un nœud $\mathsf{B}[i]$ et d'un lien du premier au second nœud.

L'ajout de ces nœuds dans A et B décale d'une position tous les nœuds se trouvant à leur droite, comme décrit dans l'Ex. 3.9.

La suppression d'une valeur donnée i est réalisée de manière symétrique : on supprime $\mathsf{B}[i]$ ainsi que $\mathsf{A}[j]$, avec j tel que $\pi(j) = i$.

Accès à une valeur Supposons que l'on souhaite déterminer $\pi(j)$. Tout d'abord, nous accédons au nœud $\mathsf{A}[j]$, suivons le lien et récupérons le nœud N, atteint dans B. Ensuite, il nous faut déterminer la position de N dans B. Cette position correspond à la valeur de $\pi(j)$.

Puisque nous souhaitons accéder aux valeurs dans B en temps logarithmique, nous stockons les nœuds dans un arbre binaire équilibré tel que $\mathsf{B}[i]$ est le $i^{\text{ème}}$ nœud lors d'un parcours préfixe. Pour ce faire, chaque nœud N stocke son rang lors d'un parcours infixe dans le sous-arbre dont N est la racine (voir Ex. 3.10 page suivante).

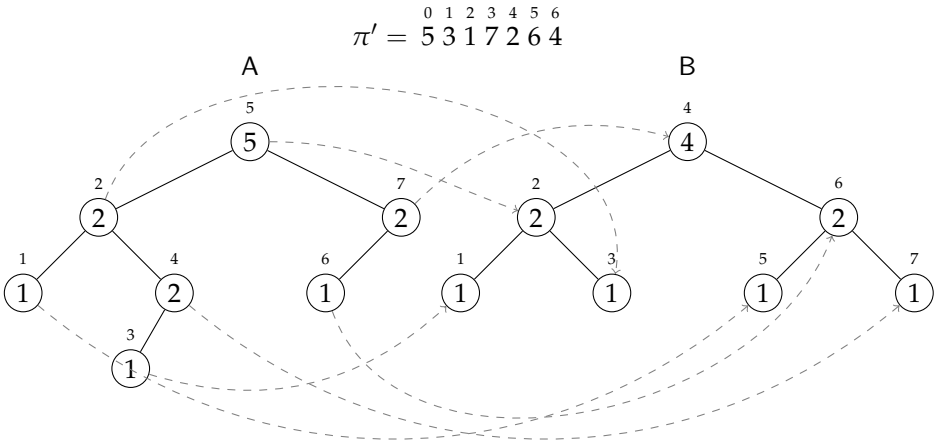
Une telle structure permet de mettre à jour efficacement la table des suffixes. Toute modification dans la table des suffixes est réalisée en temps $O(\log n)$.

3.2.2.2 Mise à jour de la table PLPC

La table des suffixes est souvent accompagnée de la table PLPC additionnelle, utile dans le cadre de certains algorithmes. Il est donc naturel de se poser la question de la mise à jour de cette table. Deux types de modifications peuvent nécessiter une mise à jour de la table PLPC : lorsqu'on supprime une valeur (suppression ou déplacement d'un élément de la TBW) ; ou lorsqu'on insère une nouvelle valeur (insertion ou déplacement d'un élément).

Suppression d'une valeur de la table PLPC Supposons que nous allons supprimer la valeur PLPC en position k, signifiant la suppression (ou le déplacement) de $L[k]$ et $\mathsf{TS}[k]$. La valeur $\mathsf{PLPC}[k-1]$ qui correspondait à la longueur du plus long préfixe commun entre le suffixe $\mathsf{TS}[k-1]$ et $\mathsf{TS}[k]$ n'est plus à jour. Cette valeur $\mathsf{PLPC}[k-1]$ doit maintenant correspondre à la longueur du plus long préfixe commun entre $\mathsf{TS}[k-1]$ et $\mathsf{TS}[k+1]$. La valeur peut être très rapidement mise à jour avec $\mathsf{PLPC}[k-1] \leftarrow \min(\mathsf{PLPC}[k-1], \mathsf{PLPC}[k])$. Après quoi la valeur $\mathsf{PLPC}[k]$ peut être effacée.

Ex. 3.10 — Représentation d'une permutation dynamique



Le nombre figurant au dessus de chaque nœud représente le rang du nœud dans un parcours infixe gauche droite. Le nombre à l'intérieur de chaque nœud représente le rang du nœud dans un parcours infixe dans le sous-arbre dont ce nœud est la racine. Les nombres figurant au dessus de chaque nœud peuvent être retrouvés à partir des nombres réellement stockés dans les nœuds.

Calcul de $\pi'(6)$

À la racine de A, le nombre est 5. Or $5 > 6$ on se déplace donc dans le sous-arbre droit. Il nous faut maintenant aller au premier nœud, dans un parcours infixe, dans ce sous-arbre. À la racine, le nombre est 2, on va donc dans le sous-arbre gauche. Dans ce sous-arbre on souhaite toujours atteindre le premier nœud. À la racine du sous-arbre le numéro est 1, on est donc arrivé au nœud recherché.
On suit alors le lien partant de ce nœud pour arriver dans l'arbre B. Dans B en suivant un principe similaire à celui utilisé dans A, on calcule que notre nœud destination est le nœud numéro 6. Ainsi $\pi(6) = 6$.

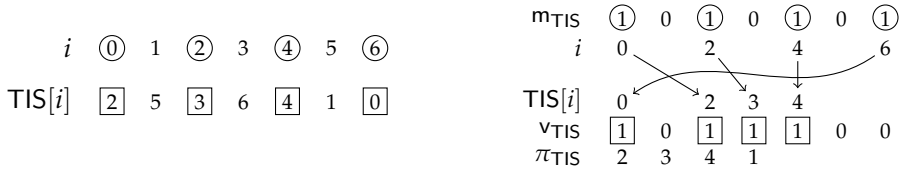
Insertion d'une valeur dans la table PLPC Une valeur est insérée dans la table PLPC lorsqu'un élément est ajouté (ou déplacé, ce qui revient au même ici) dans la table des suffixes. Supposons que l'on insère la valeur s à la position k dans TS. On doit donc recalculer $\mathsf{PLPC}[k-1]$, si elle est définie, et ajouter une valeur $\mathsf{PLPC}[k]$. Pour cela nous devons donc connaître la longueur du plus long préfixe commun entre $T'[\mathsf{TS}[k-1]\,..]$ et $T'[s\,..]$ ainsi qu'entre $T'[s\,..]$ et $T'[\mathsf{TS}[k+1]\,..]$. Les deux calculs sont similaires, nous ne traitons que celui de $\mathsf{PLPC}[k-1]$.

Si $\mathsf{LF}^{-1}(k-1) = \mathsf{LF}^{-1}(k) - 1$, cela signifie que $T'[\mathsf{TS}[k-1]+1\,..]$ et $T'[s+1\,..]$ sont côte à côte dans la table des suffixes. Dans ce cas la valeur PLPC peut être facilement déduite : soit $T'[\mathsf{TS}[k]] = T'[s]$ et alors $\mathsf{PLPC}[k-1] = \mathsf{PLPC}[\mathsf{LF}^{-1}(k-1)] + 1$ ou les deux lettres sont différentes et la valeur PLPC est nulle. Notons que cette situation, d'avoir les deux suffixes suivants côte à côte, est d'autant moins exceptionnelle que le texte est fortement répété. Néanmoins si cela n'est pas le cas il faudra comparer des facteurs du texte. Nous savons que la nouvelle valeur $\mathsf{PLPC}[k-1]$ vaut au moins l'ancienne valeur $\mathsf{PLPC}[k-1]$, sinon le suffixe commençant en position s n'aurait pas été placé ici. Ceci nous permet d'avoir un minorant et donc de savoir que les deux suffixes partagent un préfixe commun de longueur au moins $\mathsf{PLPC}[k-1]$. D'un autre côté, la valeur $\mathsf{PLPC}[\mathsf{LF}^{-1}(k-1)] + 1$, nous permet d'avoir un majorant (Kasai *et al.*, 2001, théorème 1). En utilisant ce minorant ℓ_m et ce majorant ℓ_M nous connaissons la taille minimale des deux facteurs à récupérer pour connaître la longueur du plus long préfixe. Nous récupérons donc $T'[\mathsf{TS}[k-1]+\ell_m\,..\,\mathsf{TS}[k-1]+\ell_M]$ et $T'[s+\ell_m\,..\,s+\ell_M]$ à partir de la transformée de Burrows-Wheeler et calculons la longueur de leur plus long préfixe commun et en déduisons la nouvelle valeur de $\mathsf{PLPC}[k-1]$.

Conclusion Afin de permettre une insertion et une suppression rapide des valeurs PLPC nous devons les stocker dans un arbre binaire équilibré. Cette solution est malheureusement très consommatrice en espace puisqu'il y a n nœuds dans notre arbre pour un texte de taille $n+1$. L'arbre est représenté à l'aide de pointeurs, chaque nœud doit en posséder au moins trois (deux vers les descendants directs, un vers le parent). À cela s'ajoutent les informations nécessaires pour permettre un ré-équilibrage rapide de l'arbre. Au final un stockage des valeurs PLPC avec une telle technique utilise environ $20n$ octets. Un tel espace est rédhibitoire, une de nos hypothèses de départ étant de pouvoir travailler sur des machines « ordinaires ».

De plus, sur des textes courants, $\mathsf{LF}^{-1}(k-1) = \mathsf{LF}^{-1}(k) - 1$ n'est pas vrai tout le temps, ce qui nous oblige à décompresser des facteurs du texte. Cette phase de décompression n'est pas destinée à être réalisée de façon répétée. Or c'est ce qu'il se passe lorsqu'on doit mettre à jour toutes les valeurs PLPC qui ont été impactées par les modifications du texte et le réordonnement des éléments. Cela provoque donc un ralentissement très important, rendant l'algorithme impraticable. Nous avons donc présenté cette solution ici pour l'intérêt théorique qu'elle peut – éventuellement – avoir mais ne l'utiliserons pas dans les expériences menées par la suite.

Ex. 3.11 — Échantillonnage de TIS : deux champs de bits et un tableau d'entiers



3.2.3 Mise à jour du FM-index

Le FM-index est bâti autour de deux piliers : la transformée de Burrows-Wheeler et un échantillonnage de la table des suffixes. La mise à jour d'une telle structure d'indexation nécessite la mise à jour de ces deux composantes. La première ne pose pas problème puisque nous avons décrit sa mise à jour en section 3.2.1 page 62. En revanche, dans la section 3.2.2 page 72, nous nous sommes intéressés à la mise à jour de la table des suffixes et de son inverse mais pas à celle d'un échantillonnage.

3.2.3.1 Mise à jour d'un échantillonnage de la table des suffixes

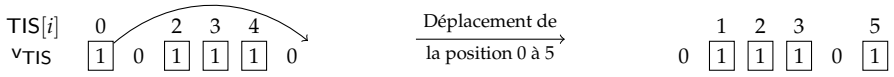
Puisque nous échantillons la table des suffixes et son inverse, nous devons d'abord savoir les positions qui sont échantillonnées. La solution pour gérer un échantillonnage de la table des suffixes a déjà été utilisée avec certaines versions du FM-index, celle de Ferragina *et al.* (2007) par exemple. Nous la présentons et expliquons comment on peut l'utiliser, avec notre structure pour mettre à jour des permutations, afin de gérer des modifications dans l'échantillonnage.

Pour stocker l'échanillonnage, nous utilisons un champ de bits, $\mathsf{m}_{\mathsf{TIS}}$, défini comme suit : $\mathsf{m}_{\mathsf{TIS}}[i] = 1$ si et seulement si la position i de la table inverse des suffixes est échantillonnée. Les valeurs échantillonnées de la table des suffixes sont stockées de manière similaire dans un champ de bits $\mathsf{v}_{\mathsf{TIS}}$: $\mathsf{v}_{\mathsf{TIS}}[i] = 1$ si et seulement si i est une valeur échantillonnée de la table inverse des suffixes. Une permutation π_{TIS} permet de lier une position à la valeur lui correspondant (voir Ex. 3.11). Cette permutation est stockée en utilisant la structure dynamique présentée précédemment.

Pour la table des suffixes nous avons des structures similaires notées m_{TS}, v_{TS} et π_{TS}. Or la table des suffixes étant la permutation inverse de la table inverse des suffixes, nous avons tout simplement : $\mathsf{m}_{\mathsf{TS}} = \mathsf{v}_{\mathsf{TIS}}$, $\mathsf{v}_{\mathsf{TS}} = \mathsf{m}_{\mathsf{TIS}}$ et $\pi_{\mathsf{TS}} = \pi_{\mathsf{TIS}}^{-1}$. Il suffit donc de mettre à jour l'échantillonnage de la table inverse des suffixes puisqu'il peut aussi servir d'échantillonnage de la table des suffixes.

L'utilisation de telles structures permet de simplifier les mises à jour à réaliser dans le cadre de notre algorithme. Ainsi pour mettre à jour TIS, on devait considérer, pour les permutations circulaires de type 4 : « Toutes les valeurs comprises entre j (exclus) et j' (inclus) sont décrémentées. Dans la table des suffixes j est modifié en j'. ». Désormais, il suffit de déplacer un bit dans $\mathsf{v}_{\mathsf{TIS}}$ de la position j à la position j' (voir Ex. 3.12 page suivante).

Ex. 3.12 — Décrémenter les valeurs dans $\mathsf{v_{TIS}}$



Conséquences du déplacement : les valeurs comprises entre 1 et 5 sont décrémentées ; la valeur échantillonnée, à la position 0, est maintenant en position 5.

Récupération d'une valeur Nous expliquons maintenant les mécanismes qui nous permettent d'accéder aux valeurs d'origines de TIS. Notons qu'ici nous nous intéresserons uniquement à la récupération de valeurs échantillonnées. Pour calculer des valeurs non échantillonnées, le principe reste le même que celui détaillé lors de l'explication du concept du FM-index (section 1.3.2.1 page 28).

Pour récupérer une valeur échantillonnée à une position i, nous devons déterminer :

1. le rang de la position échantillonnée i (c.-à-d. $r = \mathsf{rank}_1(\mathsf{m_{TIS}}, i)$) ;
2. le rang de la valeur échantillonnée correspondant à la position i (c.-à-d. $\pi_{\mathsf{TIS}}(r)$) ;
3. la valeur échantillonnée, à partir de son rang (c.-à-d. $\mathsf{select}_1(\mathsf{v_{TIS}}, \pi_{\mathsf{TIS}}(r))$).

Sachant que les opérations rank et select peuvent être effectuées en temps $O(\log n)$ dans le pire des cas (cf. section 2.1.3 page 48), la récupération d'une valeur échantillonnée prend un temps $O(\log n)$.

Ajout ou suppression d'un échantillon Les échantillons doivent être répartis de la façon la plus uniforme possible. Cela signifie que nous ne pouvons pas avoir trop (ou pas assez) d'échantillons dans un intervalle donné. Comme nous autorisons l'ajout ou la suppression de facteurs entiers dans le texte indexé, cela conduit à insérer ou supprimer de nombreuses valeurs consécutives dans $\mathsf{m_{TIS}}$. Lors de mises à jour, il faut vérifier que le nombre d'échantillons n'est pas trop important à l'endroit de la modification (dans le cas d'une suppression), auquel cas il faut en supprimer ou en déplacer. À l'inverse dans le cas d'insertions, il peut manquer des échantillons, auquel cas on peut avoir besoin d'en déplacer ou d'en ajouter.

Pour ajouter un échantillon en position i, nous devons d'abord connaître la valeur $\mathsf{TIS}[i]$ correspondante. Ensuite, nous pouvons mettre à jour nos trois structures :

1. insertion de 1 en position i dans $\mathsf{m_{TIS}}$, on pose $j = \mathsf{rank}_1(\mathsf{m_{TIS}}, i)$;
2. insertion de 1 en position $\mathsf{TIS}[i]$ dans $\mathsf{v_{TIS}}$, on pose $k = \mathsf{rank}_1(\mathsf{v_{TIS}}, \mathsf{TIS}[i])$;
3. insertion de k en position j dans π_{TIS}.

3.3 Généralisation aux suppressions et substitutions

Jusqu'à maintenant nous nous sommes intéressés à l'insertion d'un facteur de longueur ℓ sans considérer les autres types de modifications qui pourraient

avoir lieu sur le texte (suppression ou substitution). Quelle que soit la modification, l'étape de réordonnement reste la même : le but est toujours de trouver la nouvelle position des permutations circulaires commençant avant la position i. En revanche le traitement des permutations circulaires de type 2 et 3 se révèle différent. Nous allons voir en quoi le traitement de ces permutations diffère pour la suppression ou la substitution par rapport à l'insertion. Nous considérons d'abord la suppression d'un facteur de longueur ℓ en position i. Le texte résultant est $T' = T[..i-1]T[i+\ell..]$.

Type 2 **TBW** T'_{i-1} remplace $T_{i+\ell-1} = c$ dans L à la position $\mathsf{TIS}[i+\ell] = k$.

TS Aucune modification.

TIS Aucune modification.

Type 3 **TBW** Pour $j = 2$ à $\ell+1$: suppression dans L en position $k \leftarrow \widetilde{\mathsf{LF}}_c(k)$; $c \leftarrow T_{i+\ell-j}$.

TS Suppression dans TS aux mêmes positions que TBW.

TIS Pour $j = 2$ à $\ell+1$ suppression dans TIS en position $i+\ell-j$.

Pour la substitution nous considérons que le facteur $T[i..i+\ell-1]$ est remplacé par un texte S de longueur ℓ. Le texte résultant est $T' = T[..i-1]ST[i+\ell..]$.

Type2 **TBW** $S_{\ell-1}$ remplace $T_{i+\ell-1} = c$ dans L à la position $\mathsf{TIS}[i+\ell] = k$.

TS Aucune modification.

TIS Aucune modification.

Type 3 **TBW** Pour $j = 2$ à $\ell+1$: déplacement dans L de la position $k' \leftarrow \widetilde{\mathsf{LF}}_c(k)$ à la position $\mathsf{LF}(k)$; si $j \leq \ell$, $S_{\ell-j}$ remplace $T_{i+\ell-j} = c$ dans L à la position $\mathsf{LF}(k)$; $k \leftarrow k'$.

TS Déplacement dans TS aux mêmes positions que TBW.

TIS Pour $j = 2$ à $\ell+1$ substitution dans TIS en position $i+\ell-j$ de k par k', si l'élément dans la TBW a été déplacé de la position k à k'.

Bien que différent, le traitement à réaliser pour les suppressions et substitutions reste très similaire à celui de l'insertion.

3.4 Conclusion

L'algorithme que nous avons présenté nous permet d'avoir une structure d'indexation compressée supportant n'importe quel type de modification (insertion, suppression ou substitution d'un facteur). Cette méthode est la première à permettre n'importe quel type de modification sur la structure d'indexation et à pouvoir être mise en pratique. Il nous reste à vérifier si cette mise à jour est plus efficace que la reconstruction de la structure elle-même. Dans le prochain chapitre nous nous intéressons à la complexité de la mise à jour et nous menons des expériences sur des textes pour lesquels l'indexation représente un intérêt.

Chapitre 4

Complexités et résultats

Les algorithmes de mise à jour que nous avons présenté ne peuvent être intéressants que s'ils sont plus avantageux par rapport à une reconstruction complète. Pour vérifier cela, nous allons dans un premier temps analyser la complexité de l'algorithme de mise à jour du FM-index (section 4.1). Les trois algorithmes de mise à jour sont très proches et leurs complexités similaires, il paraît donc superflu de s'intéresser aux complexités des trois. Nous nous concentrons donc sur la mise à jour du FM-index qui est l'aboutissement des deux autres algorithmes. Par la suite, afin de compléter les analyses de complexités, nous mènerons des expériences sur différents types de textes (section 4.2 page 85). Ces textes sont choisis pour leurs variétés en termes d'alphabet, d'entropie, de taille. Les expériences menées visent à apprécier de façon plus précise les avantages qu'on pourrait retirer d'une mise à jour par rapport à une reconstruction. L'analyse sur différents types de textes permettra de mettre en lumière les forces ou faiblesses de notre algorithme dans différentes situations.

4.1 Complexités

Nous avons vu que notre algorithme de mise à jour peut être amené à réordonner un grand nombre de permutations circulaires. Nous commençons par nous intéresser à la complexité dans le pire des cas (section 4.1.1 page suivante). Cependant, nous savons que le pire des cas ne donne pas forcément une bonne image de la complexité réelle d'un algorithme. À titre d'exemple, les algorithmes de construction de la table des suffixes les plus efficaces sont ceux ayant une complexité dans le pire des cas en $O(n^2 \log n)$ et non ceux en $O(n)$. Afin de déterminer une complexité en moyenne (section 4.1.2 page suivante), nous devons fixer plus précisément le nombre de réordonnements auquel on peut s'attendre. Pour cela, nous détaillerons les éléments qui nous permettent de borner ce nombre.

Dans la suite de ce chapitre, nous utilisons, pour le stockage de la TBW, la structure de González et Navarro (2008) qui est la plus efficace à ce jour. Celle-ci permet de stocker la TBW d'un texte T en $nH_k(T) + o(n \log \sigma)$ bits, d'accéder, insérer, supprimer ou calculer les opérations rank ou select en temps $O(\log n(1 + \log \sigma / \log \log n))$. Par ailleurs, du point de vue de la table des suffixes, une position toutes les $\log^{1+\varepsilon} n$ est échantillonnée, où $\varepsilon > 0$. Il s'en suit

que n'importe quelle valeur de la table des suffixes (ou de l'inverse) peut être récupérée en temps $O(\log^{2+\varepsilon} n(1+\log\sigma/\log\log n))$. La structure permettant de gérer l'échantillonnage est composée de deux arbres binaires équilibrés et de deux champs de bits. Nous savons que dans chaque champ de bits nous avons $n/\log^{1+\varepsilon} n$ bits 1. Il s'en suit qu'en utilisant des champs de bits dynamiques et compressés, ils peuvent être stockés en $o(n)$ bits. Quant aux deux arbres binaires équilibrés, ils contiennent chacun $n/\log^{1+\varepsilon} n$ nœuds. Chaque nœud peut être stocké en utilisant $O(\log n)$ bits. Ainsi le stockage de l'échantillonnage occupe un espace sublinéaire, en $O(n/\log^{\varepsilon} n) = o(n)$ bits. À partir des informations précédentes, il s'en suit le théorème suivant :

Théorème 4.1. *Il existe une structure d'indexation compressée occupant $nH_k(T) + o(n\log\sigma)$ bits, qui permet de compter le nombre d'occurrences en temps $O(m\log n(1+\log\sigma/\log\log n))$, qui permet de connaître la position d'une occurrence d'un motif en temps $O(\log^{2+\varepsilon} n(1+\log\sigma/\log\log n))$ et de récupérer une chaîne de caractères de longueur ℓ en temps $O((\log^{1+\varepsilon} n+\ell)\log n(1+\log\sigma/\log\log n))$. Il est possible de mettre à jour le texte indexé par une telle structure d'indexation.*

Dans la suite, nous considérons une insertion dans le texte T en position i de longueur ℓ. Le texte T' résultant a pour longueur $n'+1$.

4.1.1 Pire des cas

Nous allons étudier, étape par étape, les coûts en temps dans le pire des cas :

Type 1 Coût : 0

Type 2 Récupération d'une valeur de TIS et substitution d'un élément dans L.
Coût : $O(\log^{2+\varepsilon} n(1+\log\sigma/\log\log n))$

Type 3 Calcul des positions et insertion de ℓ permutations circulaires.
Coût : $O(\ell\log n'(1+\log\sigma/\log\log n'))$.

Type 4 Calcul des positions et déplacement de k permutations circulaires.
Coût : $O(k\log n'(1+\log\sigma/\log\log n'))$.

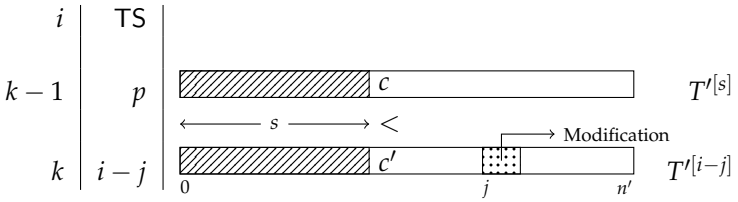
Nous savons que, dans le pire des cas, tous les éléments de type 4 doivent être réordonnés. Ainsi si on a une insertion en position $n+1$ dans le texte, cela nous fait n permutations circulaires à réordonner. Le terme dominant dans la complexité est donc l'étape 4 qui, dans le pire des cas, coûte $O(n\log n'(1+\log\sigma/\log\log n'))$. Cette complexité est donc également celle de l'algorithme.

La complexité que nous obtenons peut paraître rédhibitoire pour un algorithme censé être compétitif comparé à la reconstruction totale de la structure. C'est pourquoi nous nous intéressons maintenant à la complexité en moyenne, ce qui nous permettra d'analyser plus finement la complexité de notre algorithme.

4.1.2 Cas moyen

Afin d'estimer le nombre moyen de réordonnements, nous devons connaître la condition qui nous assure l'arrêt de ces réordonnements. Nous savons que

Fig. 4.1 — La permutation circulaire $T'^{[i-j]}$ n'a pas besoin d'être réordonné car la PLPC est trop petite.



si des éléments sont réordonnés, c'est en raison du changement d'ordre lexicographique provoqué par la modification du texte. Or ceci n'est possible que si des permutations circulaires partagent un préfixe commun et plus il y a de permutations réordonnées, plus ce préfixe commun doit être grand.

4.1.2.1 Étude du nombre de réordonnements

Pour une modification donnée, les permutations circulaires de type 4 réordonnées sont soit toutes déplacées vers le haut soit toutes vers le bas. Sans perte de généralité, on considère le cas où ces permutations circulaires sont déplacées vers le haut. Considérons la $j^{\text{ème}}$ permutation circulaire de type 4, c'est-à-dire $T'^{[i-j]}$. Soit k la position de cette permutation circulaire dans la matrice conceptuelle et p la valeur PLPC correspondante, on a donc $p = \mathsf{PLPC}[k-1]$.

Lemme 4.1. *Si $j > p$ alors $T'^{[i-j]}$ n'a pas besoin d'être déplacée et l'étape de réordonnement se termine.*

Démonstration. Lorsque l'algorithme de réordonnement traite la $j^{\text{ème}}$ permutation circulaire de type 4, $T'^{[i-j]}$, toutes les permutations circulaires $T'^{[r]}$, avec $r > i - j$, ont déjà été réordonnées. Supposons que $j > p$.

Soit $s = \mathsf{TS}[k-1]$, on sait également que $\mathsf{TS}[k] = i - j$. Par souci de clarté, nous notons aussi $c = T'^{[s]}[p]$ et $c' = T'^{[i-j]}[p]$. La modification apparaît à la position j dans $T'^{[i-j]}$, ainsi :

$$T'^{[i-j]}[0\,..\,p] = T^{[i-j]}[0\,..\,p], \text{ puisque } j > p \tag{4.1}$$

Par définition de la PLPC et de la TBW, les préfixes de longueur p de $T^{[i-j]}$ et de $T^{[s]}$ sont égaux et $T^{[i-j]}[p] > c$. En utilisant (4.1), on sait également que la relation précédente est vraie pour $T'^{[i-j]}$: les préfixes de longueur p de $T'^{[i-j]}$ et $T'^{[s]}$ sont égaux et $c' > c$ (voir FIG. 4.1).

Nous avons montré que $T'^{[i-j]}$ est correctement ordonné et n'a donc pas besoin d'être déplacé. Dans le lemme 3.3 page 70, nous avons prouvé que si $T'^{[i-j]}$ est correctement ordonné alors toutes les permutations circulaires $T'^{[q]}$, avec $q < i - j$, sont correctement ordonnées également. Finalement, si $j > \mathsf{PLPC}[k-1]$, alors le réordonnement se termine. □

Notons également que cette condition sur une valeur PLPC est une condition suffisante pour l'arrêt du réordonnement mais n'est pas une condition nécessaire. Le réordonnement peut s'arrêter avant si jamais la modification du texte n'a pas changé l'ordre lexicographique des permutations circulaires.

Les valeurs PLPC peuvent donc nous être utiles pour déterminer une borne supérieure du nombre de réordonnements pour un texte donné. Par la suite :

- le dernier centile des valeurs PLPC, noté $\mathsf{L}_{\mathsf{perc}}$, est une borne supérieure du nombre d'éléments à réordonner dans 99% des cas ;
- la valeur PLPC moyenne, notée $\mathsf{L}_{\mathsf{ave}}$, est une borne supérieure du nombre d'éléments à réordonner en moyenne ;
- la valeur PLPC maximale, notée $\mathsf{L}_{\mathsf{max}}$, est une borne supérieure du nombre d'éléments à réordonner dans le pire des cas.

Lemme 4.2. *L_{ave} est une borne supérieure du nombre de réordonnements à réaliser en moyenne.*

Démonstration. En moyenne, la valeur PLPC qui concerne la permutation circulaire $T^{[i-(\mathsf{L}_{\mathsf{ave}}+1)]}$ vaut $\mathsf{L}_{\mathsf{ave}}$. Or, d'après le lemme 4.1 page précédente, cela signifie que $T^{[i-(\mathsf{L}_{\mathsf{ave}}+1)]}$ n'est pas réordonnée et donc qu'il y a eu $\mathsf{L}_{\mathsf{ave}}$ réordonnements. De plus $\mathsf{L}_{\mathsf{ave}}$ est une borne supérieure car la modification effectuée dans le texte est également un facteur influent et peut amener à diminuer le nombre de réordonnements à effectuer (par exemple pour le texte AAAAA\$, l'insertion d'un A, à n'importe quelle position, n'entraînera jamais de réordonnements malgré des valeurs PLPCqui « autoriseraient » ce réordonnement). □

Théorème 4.2. *La structure d'indexation du théorème 4.1 page 82 peut être mise à jour, en moyenne, en temps $O(\log^{2+\varepsilon} n(1+\log\sigma/\log\log n)+(L_{ave}+\ell)\log n'(1+\log\sigma/\log\log n'))$ pour l'insertion d'un facteur de longueur ℓ dans T.*

Démonstration. Le traitement des permutations circulaires de type 2 se fait en temps $O(\log^{2+\varepsilon} n(1+\log\sigma/\log\log n))$, soit le temps nécessaire à récupérer la valeur dans TIS, permettant ainsi de connaître la position de la première modification. Le traitement des permutations circulaires de type 3 se fait en temps $O(\ell\log n'(1+\log\sigma/\log\log n'))$, ce qui correspond au temps nécessaire à l'insertion de ℓ permutations circulaires dans L ainsi que ℓ appels à la fonction LF. Enfin, il y a en moyenne $\mathsf{L}_{\mathsf{ave}}$ permutations circulaires à réordonner, ce qui correspond à une suppression suivie d'une insertion dans L, ce qui peut être fait en temps $O(\mathsf{L}_{\mathsf{ave}}\log n'(1+\log\sigma/\log\log n'))$. En ajoutant les différentes composantes, nous obtenons la complexité présentée dans le théorème. □

4.1.2.2 Valeur PLPC moyenne

La complexité est maintenant dépendante de $\mathsf{L}_{\mathsf{ave}}$. Or cette valeur peut être très variable en fonction des textes (de $\log_\sigma n$ à $n/2$) et, finalement, nous renseigne peu sur le nombre moyen théorique de réordonnements. Nous cherchons donc, si possible, à définir $\mathsf{L}_{\mathsf{ave}}$ en fonction de notions connues, telles la taille de l'alphabet, l'entropie empirique du texte ou encore la longueur du texte.

C'est pourquoi nous nous intéressons à des travaux de Fayolle et Ward (2005) dont l'objectif est de caractériser la profondeur moyenne d'un arbre

des suffixes en utilisant un modèle de Markov d'ordre 1. Ils définissent la profondeur moyenne d'un arbre des suffixes comme le nombre moyen de lettres permettant de discriminer un suffixe de n'importe quel autre suffixe, ce qui correspond à la définition de la valeur PLPC. Leur résultat est que dans de telles conditions, la valeur moyenne est asymptotiquement en $(\log n)/h + C$, où h est l'entropie de leur modèle de Markov et C est une constante.

Ce résultat est très intéressant puisqu'il nous permet de conclure qu'en moyenne, notre algorithme est polylogarithmique, surpassant donc les algorithmes de reconstruction, linéaires.

Cependant, le résultat de Fayolle et Ward a été obtenu avec un modèle de Markov d'ordre 1 et il n'est pas acquis que les textes habituellement considérés pour l'indexation correspondent à ce modèle. C'est pourquoi nous allons mener des expériences nous permettant d'évaluer, pour différents types de textes, la distribution des valeurs PLPC ainsi que la pertinence du résultat de Fayolle et Ward dans ce cadre.

4.2 Résultats expérimentaux

Consécutivement à la caractérisation de la valeur PLPC moyenne, nous cherchons d'abord à vérifier la cohérence entre la mesure donnée par Fayolle et Ward (2005) et la valeur constatée en pratique. Parallèlement, nous étudions la distribution des valeurs PLPC sur différents types de textes afin d'en déduire le nombre de réordonnements dans la majorité des cas. Ensuite, nous évaluons les performances en temps de notre algorithme de mise à jour comparées à celles des algorithmes de construction du FM-index. Sachant que plus la modification est grande, plus la mise à jour sera longue, nous cherchons à déterminer à partir de quelle longueur de modification il devient plus avantageux de reconstruire la structure que de la mettre à jour. Enfin nous nous intéressons à l'impact de la distance d'échantillonnage sur l'espace mémoire requis par notre structure et sur le temps nécessaire pour récupérer une valeur quelconque de TS.

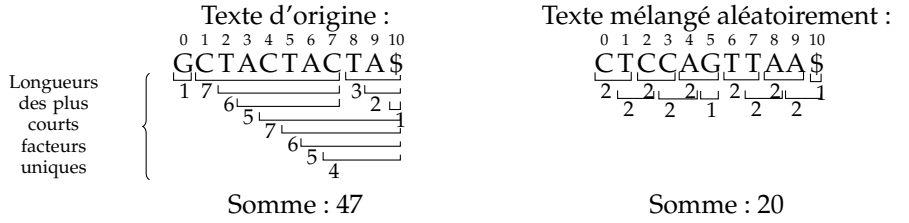
4.2.1 Étude des valeurs PLPC

Pour l'étude des valeurs PLPC nous choisissons deux familles de textes : les séquences génomiques et les textes en langage naturel. Ces textes sont ceux qui sont les pus fréquemment indexés mais ils possèdent également une grande variété pour ce qui concerne le nombre et la longueur de leurs répétitions, leur entropie ou la taille de leur alphabet.

Ces textes peuvent être classés en fonction d'une mesure de la répétition introduite par Haubold et Wiehe (2006). Cette mesure est basée sur la longueur du plus petit facteur unique dans un texte et a été initialement introduite pour les séquences génomiques. Leur indice de répétition, noté $\mathsf{l_r}$, est défini comme le logarithme du ratio entre la somme des longueurs du plus petit facteur unique, pour chaque position du texte et de cette même somme pour le texte mélangé (voir Ex. 4.1 page suivante). Le texte mélangé est un texte contenant le même nombre d'occurrences de chacune des lettres (l'entropie empirique d'ordre 0 est donc identique à celle du texte mélangé) que pour le texte originel.

Nous introduisons les ratios suivants :

Ex. 4.1 — Calcul de l'indice de répétition $\mathsf{I_r}$



$$I_r = \log(47/20) = 0,85$$

- $\mathsf{R_{FW}} = \mathsf{L_{ave}} \times H_1(T)/\log n$, ratio entre la valeur PLPC moyenne en pratique et la valeur théorique de Fayolle et Ward (2005), appliquée à un texte particulier (h qui désignait l'entropie de leur modèle a donc été remplacée par l'entropie empirique d'ordre 1 du texte considéré) ;
- $\mathsf{R_n} = \mathsf{L_{ave}}/n$, ratio entre la borne supérieure du nombre moyen de réordonnements et le nombre de réordonnements dans le pire des cas (n).

Nous les calculerons pour tous les textes que nous considérerons par la suite.

4.2.1.1 Textes sélectionnés

Séquences génomiques Parmi les séquences génomiques disponibles, Haubold et Wiehe (2006) ont choisi un sous-ensemble de trois-cent-trente-six organismes (trois-cent-trente procaryotes et six eucaryotes) et ont calculé la valeur $\mathsf{I_r}$ pour chacune de leurs séquences génomiques. En s'appuyant sur leur étude, nous nous intéressons aux cinq séquences les plus répétées parmi les trois-cent-trente procaryotes (dont les séquences sont proches de 2 Mpb). Les valeurs $\mathsf{I_r}$ associées vont de 6,34 à 3,84 et correspondent aux organismes : *M. flagellatus*, *S. agalactiae*, *D. ethenogenes*, *F. tularensis*, *N. meningitidis*.

Nous considérons également les six organismes eucaryotes de leur article : l'arabette des dames (*A. thaliana*), le nématode (*C. elegans*), la mouche du vinaigre (*D. melanogaster*), la levure (*S. cerevisiae*), l'humain (*H. sapiens*) et la souris (*M. musculus*). Leurs séquences génomiques sont plus longues : le plus grand chromosome (chr. 1) de *H. sapiens* contient plus de 200 Mpb. Ces organismes eucaryotes ont été largement étudiés, sont bien documentés et annotés, car ce sont des organismes modèles. Pour éviter les biais dûs à des régions non séquencées de génomes, nous retirons toutes les suites de N contenues dans les séquences. Ceci explique pourquoi, pour *H. sapiens*, le chromosome 1 est plus court que le chromosome 2 : nous avons retiré plus de N dans le premier que le second.

Textes en langage naturel En plus de ces grandes séquences génomiques, nous avons récupéré des textes en langage naturel du projet Gutenberg [1] (*etexts* 28000 à 28283, 224 textes bruts, sans balise) ainsi que quelques corpus [2] de l'encyclopédie Wikipedia (documents XML, balises Wiki, duplication d'articles,

[1] Téléchargé le 30 mars 2009 depuis `http://www.gutenberg.org`.
[2] Téléchargé le 30 mars 2009 depuis `http://download.wikimedia.org`.

éléments de mise en forme redondants) dans différents langages [3] (afrikaans, basque, bosnien, estonien, latin et occitan).

Après leur avoir retiré les entêtes et pieds de page propres à Gutenberg, nous avons rassemblé les textes du projet Gutenberg en un seul. En raison du nettoyage réalisé, de la taille de l'alphabet et du type de texte auquel nous nous intéressons, on ne s'attend pas à avoir de grandes valeurs PLPC. D'autant plus que les auteurs essaient généralement d'éviter autant que possible de se répéter.

En ce qui concerne Wikipedia, chaque corpus, pour un langage donné, est stocké au format XML. Le contenu lui-même est codé en utilisant le langage Wiki. Comparés aux textes du projet Gutenberg, les documents structurés en XML sont bien plus redondants. De plus, chaque corpus Wikipedia définit de nombreux modèles (éléments de mise en forme, aussi appelés *templates*) permettant de standardiser l'affichage de certaines informations. Certains de ces modèles sont fortement similaires et leur code est donc largement dupliqué, ce qui augmente les valeurs PLPC. À titre d'exemple, dans la version francophone de Wikipedia, deux modèles servent à afficher les dates des élections sous la quatrième ou la cinquième République, en France [4]. Ces deux modèles présentent tous deux, dans un tableau de la même forme, les années des élections présidentielles, législatives, sénatoriales,... Le code du modèle permettant l'affichage d'un tel tableau est donc largement dupliqué. Nous pouvons observer le même genre de phénomène pour d'autres domaines et d'autres langues, comme l'afrikaans par exemple. Pour évaluer l'effet de ces modèles sur les valeurs PLPC, nous considérons deux versions du corpus afrikaans : la version originelle et une version dans laquelle on a retiré le plus grand modèle répété.

De plus Wikipedia est célèbre pour être une encyclopédie collaborative. Ainsi chacun est libre de créer ou modifier des articles (tant que cela n'est pas considéré comme du *vandalisme*). Par conséquent, il peut arriver qu'une partie d'un article soit réutilisée afin d'en compléter un autre. Ce phénomène contribue également à augmenter significativement les valeurs PLPC.

4.2.1.2 Méthodologie

Pour tous les textes, nous calculons les tables de suffixes, en utilisant l'algorithme de Manzini et Ferragina (2004), et les tables PLPC en utilisant l'algorithme de Manzini (2004). L'implantation utilisée est celle proposée par G. Manzini [5]. Ces algorithmes ont été choisis en raison de leur faible consommation mémoire. Pour chaque texte, nous calculons également les valeurs L_{max}, L_{ave}, L_{perc} définies précédemment (voir la section 4.1.2.1 page 83). Nous calculons également les valeurs R_n, R_{FW} et I_r.

Nous traçons des graphiques qui permettent d'étudier le nombre de réordonnements nécessaires pour mettre à jour un index, en fonction de la longueur du texte indexé. Pour ce faire, nous traçons la valeur L_{ave} pour chaque

[3] Les langues ont été choisies en fonction de la taille de leur corpus de façon à ce que, d'un côté, les calculs puissent tenir en mémoire centrale et, de l'autre, le corpus fasse au moins 70 Mo.

[4] Voir `http://fr.wikipedia.org/wiki/Modèle:Élections_en_France_sous_la_Quatrième_République` et `http://fr.wikipedia.org/wiki/Modèle:Élections_en_France_sous_la_Cinquième_République`.

[5] `http://web.unipmn.it/~manzini/lightweight/`

longueur de texte et nous mesurons le nombre réel de réordonnements effectués après l'insertion d'une seule lettre. Les insertions sont répétées 100 000 fois et le nombre moyen de réordonnements est calculé pour toutes ces insertions. Pour des raisons de simplicité de calcul, ce sont des suffixes du texte de plus en plus longs qui sont considérés. Par conséquent, lorsqu'une grande répétition est présente à la fin du texte, un pic apparaîtra sur le graphique, au début. De plus, plus la répétition sera proche de la fin, moins le nombre de valeurs PLPC considérées sera grand et donc plus la valeur L_{ave} sera importante. Ainsi, deux pics de même hauteur, un vers le début du graphique, un autre vers la fin représentent deux répétitions de longueurs très différentes (la deuxième étant bien plus grande que la première). Quand cela s'avère nécessaire, nous fournissons aussi le graphique pour le miroir du texte afin de mieux prendre en compte le biais dû à une répétition en fin de texte.

4.2.1.3 Résultats

Dans cette section, nous présentons les résultats expérimentaux obtenus à partir des ensembles de texte décrits précédemment. À partir des résultats obtenus, nous montrons la forte corrélation entre la valeur PLPC moyenne et le nombre de réordonnements en pratique.

Séquences génomiques Nous présentons maintenant deux analyses qui sont basées sur les valeurs calculées à partir des tables PLPC et sur les graphiques tracés à partir des expériences que nous avons conduites.

Analyse des valeurs PLPC Les caractéristiques des cinq génomes procaryotes sont présentées dans la TAB. 4.1 page ci-contre. Ces courtes séquences génomiques ont été largement étudiées et sont connues pour contenir de nombreuses courtes répétitions ainsi que quelques grandes régions répétées. Il n'est donc pas surprenant d'observer de grande valeurs L_{max} relativement à la taille des séquences. Par exemple, dans *M. flagellatus*, 5 % du génome correspond à une seule répétition. La plus grande région répétée est de 143 kpb tandis que la deuxième plus répétée fait moins d'1 kpb de long. Ainsi, pour cette séquence, la grande valeur de L_{ave} est trompeuse puisqu'elle n'est due qu'à une seule répétition d'une longueur exceptionnelle. La plus grande valeur I_r est obtenue pour *M. flagellatus* alors qu'il ne possède qu'une grande répétition. Ainsi, on peut se demander si un texte ayant une seule répétition gigantesque peut être considéré comme « plus répété » qu'un texte ayant de nombreuses répétitions de longueurs moins exceptionnelles. Puisque la complexité en moyenne de notre algorithme dépend de L_{ave}, et que nous souhaitons tester son efficacité dans les pires conditions, on peut néanmoins considérer qu'il est pertinent de s'intéresser aux textes les plus répétés suivant la définition de I_r.

Bien que nous ayons de grandes valeurs PLPC en moyenne, elles sont encore loin du nombre moyen de réordonnements auquel on peut penser dans le pire des cas, sur le pire des textes : $n/2$. Les valeurs de R_n montrent qu'en moyenne de 0,116 % (*M. flagellatus*) à 0,011 5 % (*N. meningitidis*) éléments doivent être réordonnés, au lieu de 50 %. Les valeurs R_{FW} sont au dessus de 20, signifiant qu'elles sont sensiblement différentes des résultats que nous pouvons attendre de Fayolle et Ward (2005), montrant qu'un modèle

Tab. 4.1 — Valeurs pour les cinq séquences génomiques de procaryotes les plus répétées

Organisme	Lg. séq.	L_{max}	L_{ave}	L_{perc}	R_n	R_{FW}	I_r
M. flagellatus	2 971 519	143 034	3 452	113 320	$1,16 \cdot 10^{-3}$	315,49	6,34
S. agalactiae	2 211 485	47 068	546	24 954	$2,47 \cdot 10^{-4}$	50,02	4,84
D. ethenogenes	1 469 720	21 106	377	11 918	$2,56 \cdot 10^{-4}$	36,41	4,03
F. tularensis	1 892 775	33 912	336	14 984	$1,78 \cdot 10^{-4}$	30,49	3,96
N. meningitidis	2 272 360	32 158	261	9 435	$1,15 \cdot 10^{-4}$	24,22	3,84

Tab. 4.2 — Valeurs pour les génomes eucaryotes

Organisme	Lg. séq.	L_{max}	L_{ave}	L_{perc}	R_n	R_{FW}	I_r
A. thaliana	86 188 477	39 960	53	188	$6,13 \cdot 10^{-7}$	3,9	1,87
C. elegans	100 269 917	38 987	45	196	$4,47 \cdot 10^{-7}$	3,2	1,73
D. melanogaster	120 290 946	30 892	66	1 655	$5,47 \cdot 10^{-7}$	4,8	1,89
S. cerevisiae	12 156 679	8 375	43	922	$3,53 \cdot 10^{-6}$	3,6	1,74

Tab. 4.3 — Valeurs pour certaines séquences chromosomiques de *H. sapiens*.

Chromosome	Longueur	L_{max}	L_{ave}	L_{perc}	R_n	R_{FW}	I_r
1	226 212 984	67 631	40	171	$1,77 \cdot 10^{-7}$	2,8	1,61
2	237 898 220	43 034	26	86	$1,09 \cdot 10^{-7}$	1,8	1,76
9	120 983 611	51 976	76	688	$6,28 \cdot 10^{-7}$	5,4	2,82
10	131 735 771	30 751	32	208	$2,43 \cdot 10^{-7}$	2,3	1,60
14	88 290 585	1 292	16	68	$1,81 \cdot 10^{-7}$	1,2	0,41
15	81 920 097	25 713	41	369	$5 \cdot 10^{-7}$	3	1,88
17	79 601 503	15 692	32	302	$4,02 \cdot 10^{-7}$	2,4	2,41
19	56 037 509	3 412	21	88	$3,75 \cdot 10^{-7}$	1,6	0,67
20	59 505 253	866	15	57	$2,52 \cdot 10^{-7}$	1,2	0,53
22	35 058 629	2 331	19	114	$5,42 \cdot 10^{-7}$	1,5	0,84
X	152 538 530	51 821	52	215	$3,41 \cdot 10^{-7}$	3,7	2,40
Y	25 652 954	11 501	98	2 598	$3,82 \cdot 10^{-6}$	7,7	4,31

de Markov supérieur à 1 devrait être considéré. Puisque nous avons choisi les cinq séquences génomiques les plus répétées, nous ne nous attendons pas à obtenir, pour d'autres séquences, des valeurs I_r ou R_{FW} plus grandes que celles-ci. Ainsi, les valeurs R_{FW}, bien plus grandes que 1, sont les valeurs maximales qu'on est susceptibles de rencontrer. Dans ces conditions, puisque L_{ave} est bien plus proche de $\log n$ que de n, d'après les valeurs R_{FW} et R_n, maintenir le FM-index à jour semble être beaucoup plus avantageux que le reconstruire entièrement.

Les génomes eucaryotes, comme ceux de la TAB. 4.2 (*A. thaliana*, *C. elegans*, *D. melanogaster*, *S. cerevisiae*), TAB. 4.3 (plusieurs chromosomes de *H. sapiens*) et TAB. 4.4 page suivante (plusieurs chromosomes de *M. musculus*), contiennent généralement un pourcentage de répétitions inférieur aux séquences génomiques des procaryotes mais sont bien plus représentatives parmi les espèces

Tab. 4.4 — Valeurs pour certaines séquences chromosomiques de *M. musculus*.

Chromosome	Longueur	L_{max}	L_{ave}	L_{perc}	R_n	R_{FW}	I_r
1	191 477 429	7 279	28	345	$1,49 \cdot 10^{-7}$	2	0,85
2	178 392 072	137 338	154	508	$8,64 \cdot 10^{-7}$	10,8	3,22
7	141 878 210	77 175	167	1 755	$1,18 \cdot 10^{-6}$	11,9	3,67
9	120 720 222	8 705	24	237	$1,97 \cdot 10^{-7}$	1,7	0,74
12	117 459 310	90 253	123	623	$1,05 \cdot 10^{-6}$	8,8	2,92
14	121 635 309	79 256	158	1 539	$1,3 \cdot 10^{-6}$	11,3	3,13
15	100 439 974	6 418	25	267	$2,5 \cdot 10^{-7}$	1,8	0,78
17	91 898 202	26 807	35	310	$3,78 \cdot 10^{-7}$	2,5	2,23
19	58 142 230	4 368	20	167	$3,52 \cdot 10^{-7}$	1,5	0,67
X	162 080 892	82 006	219	2 680	$1,35 \cdot 10^{-6}$	15,4	3,64
Y	2 702 555	16 589	192	3 303	$7,09 \cdot 10^{-5}$	17,2	3,72

végétales et animales. Ainsi, il est assez naturel de constater que, en moyenne, les valeurs PLPC sont largement inférieures par rapport à celles des cinq séquences génomiques de procaryotes les plus répétées. Malgré tout, nous continuons à observer de grandes valeurs L_{max} mais, en raison des séquences qui sont plus longues, les quelques grandes régions répétées sont noyées parmi les autres valeurs : L_{ave} et L_{perc} sont quant à elles plus petites. Nous pouvons effectivement vérifier ceci sur le chromosome 1 de *H. sapiens* : L_{perc}, qui correspond au dernier centile des valeurs PLPC est égal à 171 tandis que L_{max} vaut 67 631. Ainsi, toutes les valeurs PLPC comprises entre 171 et 67 631 représentent seulement 1 % de toutes les valeurs PLPC ! Le reste correspond principalement à de petites répétitions telles que les répétitions courtes en tandem, les micro et mini-satellites ou d'autres structures biologiques.

De nouveau, les grandes valeurs R_{FW} constatées pour plusieurs chromosomes (p. ex. chromosomes 2, 7, 12, 14, X et Y de la souris) confirment qu'un modèle de Markov d'ordre supérieur serait nécessaire afin d'estimer de façon plus fiable les valeurs PLPC moyennes. Malgré tout, pour d'autres chromosomes (p. ex. chromosomes 1, 9, 15 et 19 de la souris), les valeurs R_{FW} sont proches de 1, indiquant qu'en dépit du faible ordre du modèle de Markov utilisé, la valeur théorique est une assez bonne approximation pour les textes les moins répétés.

Notons également que pour la séquence chromosomique la plus répétée de *M. musculus*, seulement 0,007 09 % éléments doivent être réordonnés en moyenne et 0,61 % dans le pire des cas.

Analyse des courbes Pour la FIG. 4.2 page 92, nous choisissons deux génomes de procaryotes : ceux dont la séquence a la valeur R_{FW} la plus grande et la plus faible (*M. flagellatus* et *N. meningitidis*) ; ainsi que deux séquences chromosomiques de *M. musculus* dont les longueurs sont similaires mais ont des valeurs PLPC totalement différentes (chromosomes 1 et 2).

Alors que nous observons dans *M. flagellatus* une grande région répétée, *N. meningitidis* possède deux grandes régions répétées, la plus grande étant néanmoins quatre fois moins longue que celle de *M. flagellatus*.

En raison de la longueur du chromosome 1 de la souris et des valeurs de L_{max} et L_{ave} (voir TAB. 4.4 page précédente), nous n'attendons pas de pentes raides ou de pics sur le graphique. La courbe qui a été tracée confirme nos attentes. À l'opposé, pour le chromosome 2, L_{max} est vingt fois plus grand que celui du chromosome 1 alors que le L_{ave} du chromosome 2 est « seulement » six fois plus grand que celui du chromosome 1. Ceci traduit la présence de très grande(s) répétition(s) au sein d'une séquence contenant peu de répétitions importantes par ailleurs. C'est également ce que nous observons dans les graphiques : nous avons un très grand pic au début (correspondant à une répétition à la fin de la séquence), qui s'amortit au fur et à mesure que les suffixes considérés sont de plus en plus longs. Cet effet d'amortissement masque les valeurs PLPC suivantes, c'est pourquoi nous produisons aussi le graphique pour le miroir du chromosome 2, nous permettant ainsi d'observer ce qui se passe au début de la séquence. Nous obtenons toujours un très grand pic, à la fin cette fois, les valeurs qui précèdent confirment le peu de répétitions hormis un léger sursaut aux alentours de 100 Mpb.

L_{ave} est la borne supérieure du nombre d'éléments à réordonner en moyenne. On s'attend donc logiquement à voir cette courbe au dessus du nombre de réordonnements observé en pratique. C'est presque toujours le cas mais la valeur en pratique étant calculée grâce à une moyenne sur 100 000 insertions choisies aléatoirement, il est normal que, parfois, la valeur moyenne sur 100 000 insertions dépasse légèrement la borne supérieure. En revanche, plus on augmente le nombre d'insertions aléatoires à réaliser, plus ce cas se fera rare. Par ailleurs, les graphiques confirment l'importance de L_{ave} pour estimer finement le nombre de réordonnements.

Textes en langage naturel De façon similaire à l'étude conduite pour les séquences génomiques, nous nous concentrons maintenant sur les valeurs PLPC et l'analyse des courbes pour les textes en langage naturel.

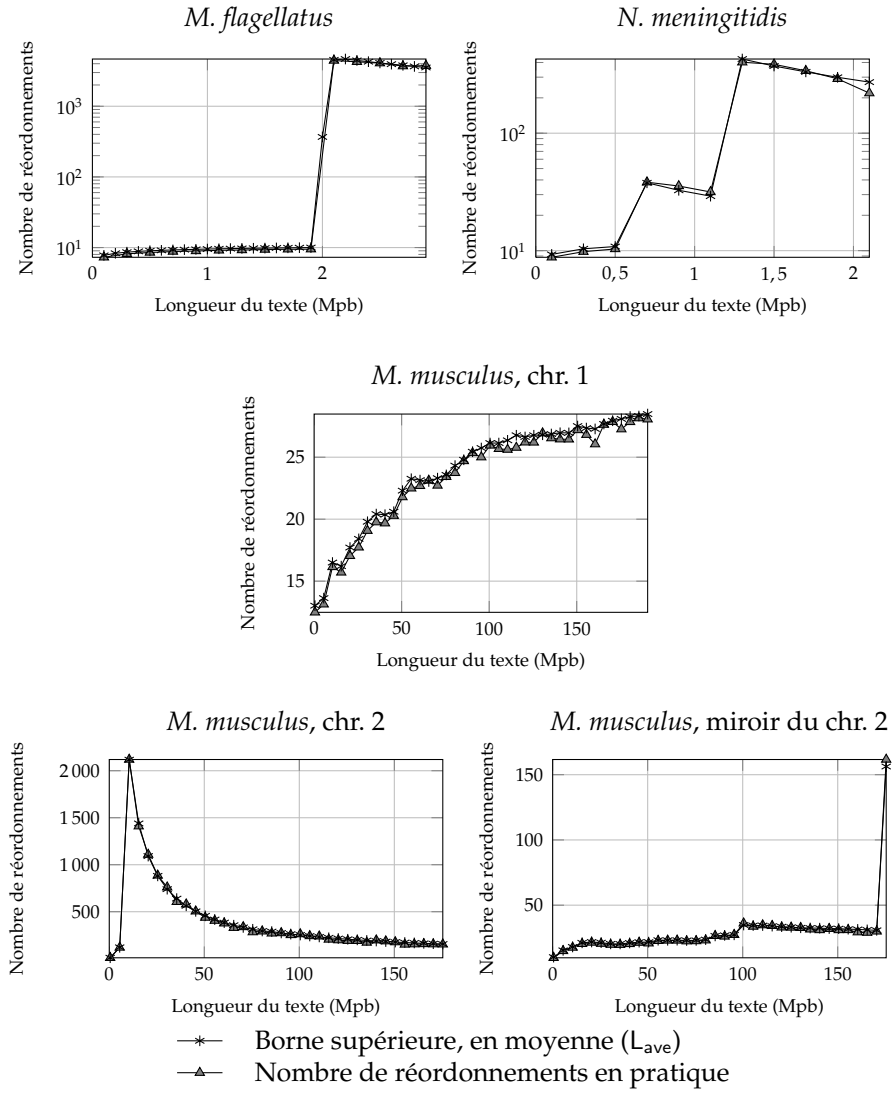
Analyse des valeurs PLPC Pour les *etexts*, nous avons de petites valeurs PLPC comme le montre la TAB. 4.5 page 93. L_{perc} vaut seulement 32, signifiant que la mise à jour du FM-index nécessite au pire 32 réordonnements dans 99 % des cas !

À l'opposé, nous avons des valeurs PLPC plus importantes avec les corpus Wikipedia. Ce n'est pas surprenant, en raison du format de données, et dans ce type de textes nous avons des duplications comme nous l'avons expliqué en section 4.2.1.1 page 86. De plus, retirer un seul *template* dupliqué dans le corpus en afrikaans fait chuter les valeurs L_{max}, L_{ave} et R_{FW}(cf. ligne « af. (nettoyé) »).

Les valeurs L_{ave}, I_r sont largement inférieures pour les *etexts* par rapport aux corpus Wikipedia. Les *templates* redondants présents dans le second étant absents du premier, cela réduit mécaniquement ces valeurs. Néanmoins, même pour le corpus le plus répété, afrikaans, nous observons que moins de 0,000 1 % des éléments devrait être réordonnés en moyenne et au plus 0,05 % dans le pire des cas.

Analyse des courbes Pour les *etexts* nous nous attendons à ce que la valeur L_{ave} monte rapidement jusqu'à se stabiliser. Ainsi, doubler la taille du

Fig. 4.2 — Nombre de réordonnements théoriques et en pratique pour l'insertion d'une seule lettre dans des suffixes de séquences génomiques.



Tab. 4.5 — Valeurs pour différents textes en langage naturel. *etexts* est une concaténation de plusieurs textes du projet Gutenberg. Les autres textes sont des corpus Wikipedia dans le langage associé.

Textes	Longueur	$\mathsf{L}_{\mathsf{max}}$	$\mathsf{L}_{\mathsf{ave}}$	$\mathsf{L}_{\mathsf{perc}}$	R_{n}	R_{FW}	I_{r}
etexts	91 070 340	1 727	12	32	$1,32 \cdot 10^{-7}$	1,7	0.82
afrikaans	68 989 658	34 205	66	383	$9,57 \cdot 10^{-7}$	10,1	3,00
af. (nettoyé)	68 743 934	6 831	31	295	$4,5 \cdot 10^{-7}$	4,7	1,98
basque	139 868 091	13 879	45	406	$3,22 \cdot 10^{-7}$	6,3	2,16
bosnien	122 215 463	18 314	40	318	$3,27 \cdot 10^{-7}$	5,9	2,28
estonien	213 877 916	26 119	39	284	$1,82 \cdot 10^{-7}$	5,4	2,16
latin	85 735 379	8 099	38	346	$4,43 \cdot 10^{-7}$	5,6	2,15
occitan	70 250 160	11 003	64	914	$9,11 \cdot 10^{-7}$	9,4	2,64

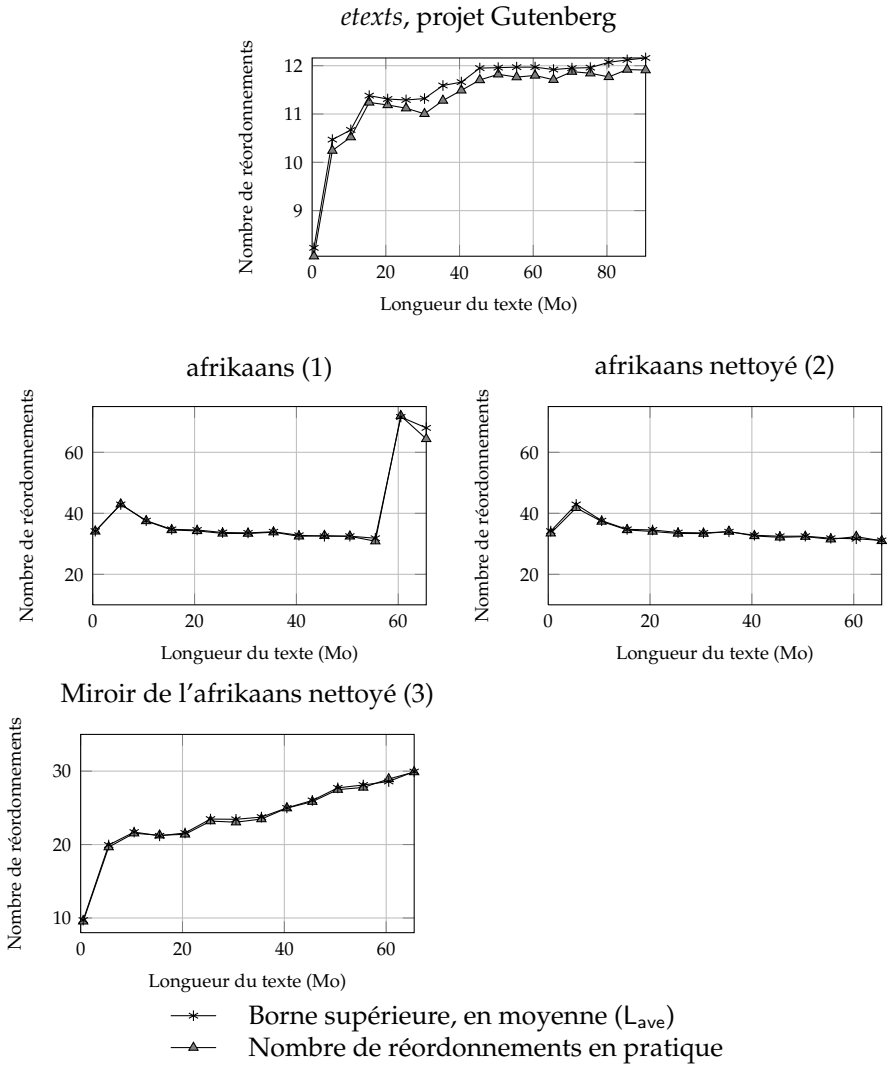
texte n'aura pas un impact significatif sur les valeurs $\mathsf{L}_{\mathsf{ave}}$. La courbe suit une progression similaire à celle d'une courbe logarithmique telle que la valeur théorique de (Fayolle et Ward, 2005), ce qui est confirmé par la valeur R_{FW} indiquant une grande proximité entre la valeur pratique pour les *etexts* et le résultat théorique.

Dans la FIG. 4.3 page suivante, le graphique pour les *etexts* confirme que nous avons uniquement de courtes répétitions dans les textes en langage naturel non formatés. D'un autre côté, le corpus afrikaans de Wikipedia contient quelques grandes répétitions que l'on peut observer par une rapide croissance de $\mathsf{L}_{\mathsf{ave}}$ aux alentours de 60 Mo. Dans la FIG. 4.3 (1), le graphique pour le corpus afrikaans originel est tracé et un pic apparaît clairement sur la droite. Cela correspond à une seule répétition de longueur 34 205 à l'intérieur d'un modèle dont la longueur est de 245 724. Par la suite, nous avons retiré ce modèle particulier du corpus, ce qui supprime le pic correspondant dans la FIG. 4.3 (2). De façon similaire à ce que nous avons fait pour *M. musculus*, nous prenons le miroir du corpus afrikaans nettoyé afin d'atténuer le pic le plus à gauche. Nous observons alors que le graphique de la FIG. 4.3 (3) a un comportement relativement proche du graphique pour les *etexts* et, globalement, ces deux courbes suivent un comportement logarithmique, proche de la formule proposée par Fayolle et Ward.

4.2.1.4 Conclusion

Pour des textes ayant peu de répétitions (dans les textes que nous avons choisi il s'agit de ceux tels que $\mathsf{I}_{\mathsf{r}} < 1$), le résultat de Fayolle et Ward (2005) permet d'approcher de façon satisfaisante $\mathsf{L}_{\mathsf{ave}}$. Ainsi, dans ce cas notre algorithme de mise à jour du FM-index a une complexité en moyenne en $O(\log^{2+\varepsilon} n(1 + \log\sigma/\log\log n) + ((\log n)/H_1(T) + \ell)\log n'(1 + \log\sigma/\log\log n'))$. Exprimer la complexité de cette façon a un avantage certain : celle-ci dépend uniquement de la taille du texte, son alphabet, son entropie et de la taille de la modification. Elle n'est pas exprimée en fonction d'autres éléments, pouvant être difficiles à appréhender, tels que la valeur PLPC moyenne. Cette complexité nous permet de conclure qu'en moyenne notre algorithme est polylogarithmique.

Fig. 4.3 — Nombre de réordonnements en pratique et en théorie pour l'insertion d'une seule lettre dans des suffixes de textes en langage naturel.



Par ailleurs, pour les textes contenant plus de répétitions (ceux dont $l_r \geq 1$), le résultat de Fayolle et Ward peut difficilement s'appliquer. Nous ne sommes donc pas en mesure de tirer le même genre de conclusion pour ces textes. Néanmoins, nous avons montré à travers des exemples variés, que la valeur PLPC moyenne est très faible en comparaison de la longueur du texte, ce qui est également confirmé par les minuscules valeurs R_n observées. Afin de tirer des conclusions définitives, quant à la complexité en moyenne de notre algorithme, pour tout type de texte, il pourrait être utile d'étendre les résultats de Fayolle et Ward à des modèles de Markov d'ordre supérieur. D'après les auteurs cette extension est tout à fait réalisable. Bien qu'une telle extension serait intéressante, elle ne résoudrait pas tous les problèmes car des modèles de Markov d'ordre supérieur ne pourraient pas prendre en compte des répétitions très longues.

4.2.2 Implantation

Avant de nous intéresser aux performances en temps de notre structure d'indexation, nous expliquons les détails de son implantation. Celle-ci est basée sur celle gracieusement fournie par Gerlach (2007). Gerlach a implanté la méthode introduite par Mäkinen et Navarro (2008), ainsi que les outils correspondants (champs de bits et champs de lettres). Néanmoins, avec la méthode de Mäkinen et Navarro, les opérations rank, select ainsi que les insertions ou suppressions sont réalisées en temps $O(\log n \log \sigma)$. Notre implantation est également basée sur cette méthode. Ainsi pour avoir la complexité en pratique de notre méthode, avec cette implantation, il faudra remplacer dans les complexités données précédemment le $\log n(1 + \log \sigma / \log \log n)$ par $\log n \log \sigma$. De plus, l'implantation de Gerlach n'avait pas recours à la compression pour les champs de bits.

Nous avons implanté notre algorithme à partir de cette version et nous avons réorganisé le code, pour plus de modularité. Nous avons également amélioré l'implantation des champs de bits afin qu'ils soient plus économes en espace et ce, sans détérioration des temps de requêtes (voire une légère amélioration).

Nous avons également choisi une distance d'échantillonnage moyenne autour de 90. C'est-à-dire qu'il y a en moyenne 90 valeurs de la table des suffixes entre deux valeurs échantillonnées. Autrement dit, environ 1,1 % de la table des suffixes est réellement stockée. Nous donnons plus d'explication sur les évolutions des performances en temps et en espace, en fonction de la distance d'échantillonnage dans la section 4.2.5 page 103.

4.2.3 Performances en temps

Nous avons montré, expériences à l'appui, que le nombre de réordonnements pour modifier un FM-index est très faible en comparaison de la longueur du texte indexé. Néanmoins, nous souhaitons connaître les avantages de notre structure d'indexation dynamique par rapport à une version statique. Pour cela nous allons donc comparer les temps de mise à jour des structures (c'est-à-dire le temps de reconstruction pour la structure statique) ainsi que les temps pour rechercher les occurrences d'un motif. Parmi les textes que nous

avons utilisé en section 4.1 page 86, nous choisissons ceux ayant des caractéristiques assez éloignées :

- *M. flagellatus*, qui possède une très grande répétition (5 % de sa séquence) ;
- chromosome 1 de la souris, faiblement répété ($L_{max} = 7\ 279$ et $R_{FW} = 2$) ;
- *etexts*, texte en langage naturel très peu répété ;
- afrikaans, corpus Wikipedia le plus répété.

4.2.3.1 Temps de mise à jour

Afin de tester le temps nécessaire à la mise à jour, nous choisissons d'insérer un facteur de longueur 20 dans notre texte à une position aléatoire. Le facteur est choisi aléatoirement à l'intérieur du texte indexé. Le temps d'insertion est mesuré en utilisant la fonction `C gettimeofday`. Ensuite, le facteur inséré est supprimé et le processus renouvelé cinq millions de fois. À partir de toutes ces insertions, le temps moyen d'insertion est calculé.

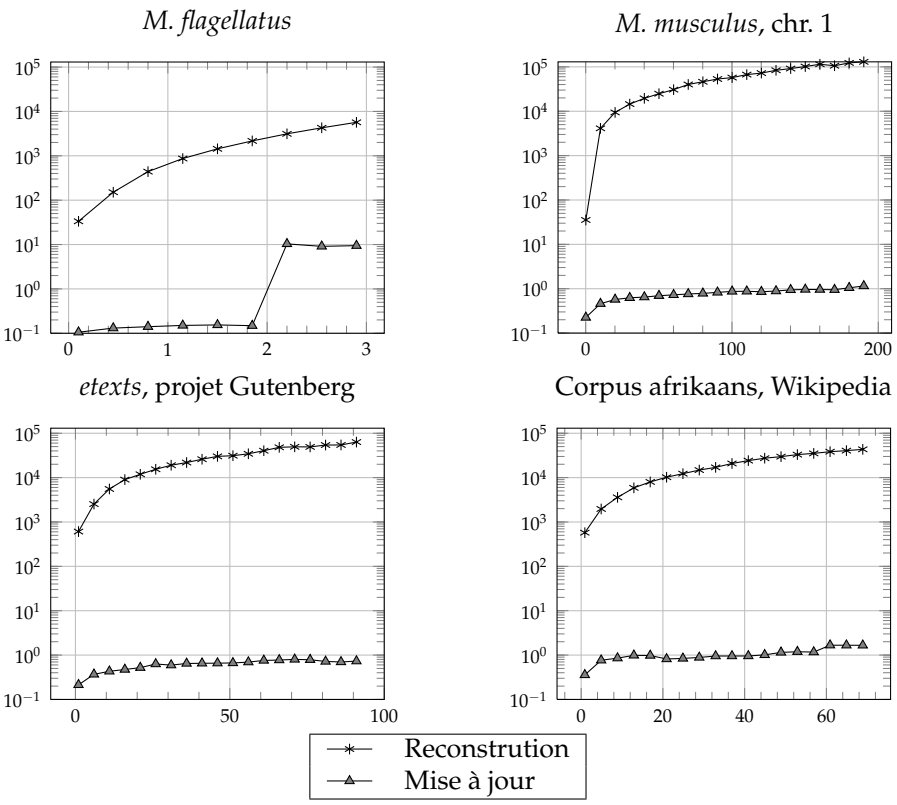
Pour les structures d'indexation statiques, nous choisissons un type de FM-index et mesurons le temps nécessaire à sa reconstruction. Il aurait été plus logique de choisir une structure d'indexation dynamique pour comparer les performances de notre structure avec celles d'autres solutions. Malheureusement les méthodes permettant une mise à jour totale de la structure (comme celles proposées par Ferragina et Grossi (1995) ; Hon *et al.* (2004)) ne sont pas simples et aucune implantation n'existe. De plus ces méthodes consistant à indexer des facteurs chevauchants, la consommation mémoire de la structure serait prohibitive. Par ailleurs la méthode de Ehrenfeucht *et al.* (2009) est récente et leur implantation ne permet pas encore de mises à jour.

Ainsi le FM-index statique que nous avons choisi a été implanté par R. González [6] et est basé sur les résultats de Mäkinen et Navarro (2005). Cette implantation a été choisie car il s'agit d'une des implantations les plus rapides (mais, compromis espace-temps oblige, l'une des moins économe en espace comme nous l'avons vu en section 1.4 page 38). Nous choisissons donc de comparer notre algorithme de mise à jour du FM-index avec l'une des plus rapides implantations du FM-index statique. La reconstruction est effectuée cent fois consécutives et nous prenons la moyenne de ces temps de reconstruction. Les résultats sont présentés dans la FIG. 4.4 page ci-contre. Pour une meilleure lisibilité, nous utilisons une échelle logarithmique en ordonnée.

Les résultats que nous obtenons ne sont pas particulièrement surprenants dans le sens où les expériences précédentes nous avaient montré qu'une faible proportion d'éléments devait être réordonnée pour mettre à jour le FM-index. Il est donc naturel de constater une nette supériorité de notre algorithme de mise à jour par rapport à la reconstruction de la structure. Pour la mise à jour nous observons les effets des valeurs L_{ave} : on voit nettement le temps de mise à jour multiplié par près de 100, pour *M. flagellatus* aux alentours de 2 Mo. Sans surprise la mise à jour pour le chromosome 1 de la souris prend moins de temps que pour *M. flagellatus*, il s'agit également d'une conséquence des L_{ave} constatées. De même, la mise à jour pour les *etexts* est plus rapide que pour le corpus Wikipedia. En revanche, on peut observer des temps de mise à jour similaires pour *etexts* et pour le chromosome 1 de la souris. Cela ne semble pas cohérent avec les résultats obtenus pour les L_{ave} de ces deux textes (res-

[6] `http://pizzachili.dcc.uchile.cl/indexes/Succinct_Suffix_Array/`

Fig. 4.4 — Comparaison du temps de mise à jour et du temps pour la reconstruction, après l'insertion d'une chaîne de 20 lettres.



Abscisse : Longueur du suffixe du texte considéré (en Mo)
Ordonnée : Temps (en ms)

pectivement 12 et 28). Mais un autre élément à prendre en compte est la taille de l'alphabet et celle-ci est bien plus importante pour les *etexts* que pour les séquences génomiques (respectivement 200 et 4). Or la complexité en temps de notre algorithme dépend de la taille de l'alphabet, ce qui explique le léger ralentissement pour les textes en langage naturel par rapport aux séquences génomiques.

Nous pouvons constater que non seulement notre algorithme est plus rapide (jusqu'à 10 000 fois) mais en plus le temps pour la mise à jour croît moins vite que celui pour la reconstruction. Cela signifie donc que plus le texte sera grand plus l'intérêt de mettre à jour sera important. Lorsque l'algorithme de mise à jour est 10 000 fois plus rapide que la reconstruction, cela signifie que l'on pourrait insérer 10 000 facteurs de longueur 20 pour arriver à un temps proche de celui de la reconstruction.

4.2.3.2 Temps de recherche

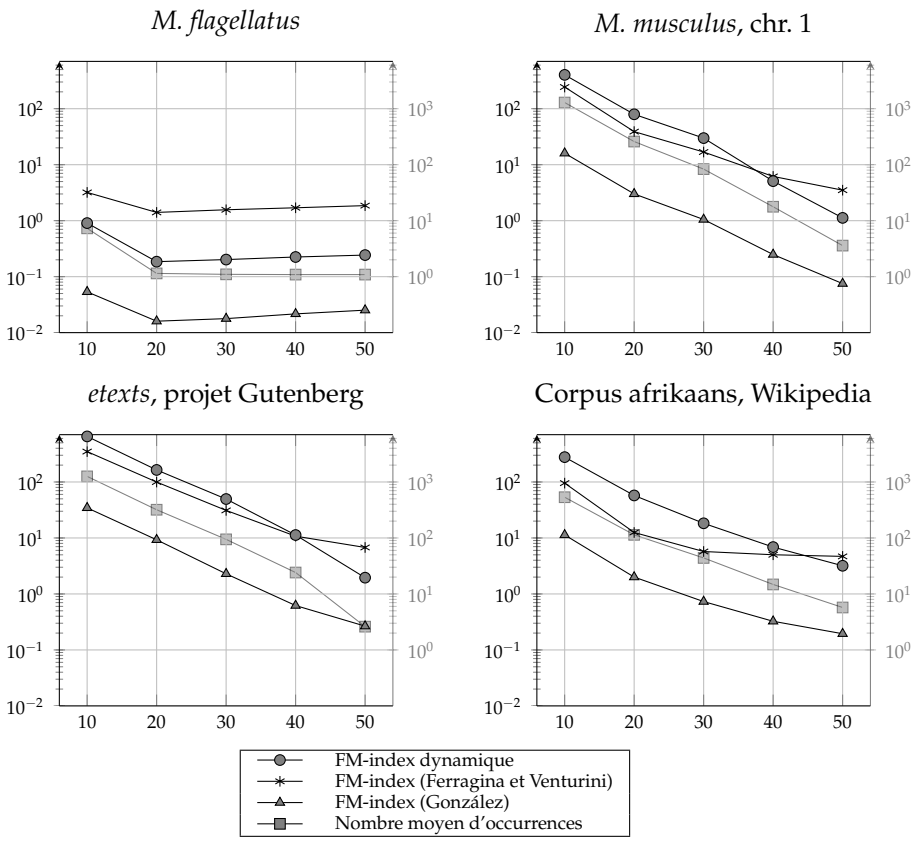
Une structure d'indexation est utilisée pour rechercher des motifs dans un texte. Nous ne proposons rien de nouveau dans ce sens là car nous nous contentons d'utiliser le concept de FM-index avec des structures dynamiques pour les champs de lettres. Néanmoins, se contenter de montrer que notre algorithme de mise à jour est beaucoup plus rapide que la reconstruction de la structure d'indexation n'est pas suffisant pour avoir une bonne appréciation des avantages et inconvénients des deux solutions. Afin d'être le plus complet possible, il faut également prendre en compte le temps nécessaire à la recherche de motifs. Ce temps sera forcément plus important pour la structure dynamique que pour la structure statique, sachant que la fonction LF est calculée en temps $O(\log n(1 + \log \sigma / \log \log n))$ dans le premier cas contre $O(1)$ dans le second.

Nous utilisons toujours les mêmes textes que précédemment et extrayons aléatoirement de ces quatre textes 10 000 motifs distincts de longueur 50. On considère également les préfixes de longueurs 10, 20, 30 et 40 de ces motifs. Nous localisons (c'est-à-dire cherchons la position de toutes les occurrences) chaque facteur ainsi que leurs préfixes mentionnés précédemment. Parmi les structures d'indexation utilisées pour ce test, nous avons celles dont nous nous sommes servis pour les temps de mise à jour. En plus de celles-ci nous ajoutons une autre version du FM-index, très efficace en terme de compression, mais plus lente. Elle a été codée par P. Ferragina et R. Venturini [7]. Les résultats sont présentés en FIG. 4.5 page suivante. Nous utilisons une double échelle en abscisse à la fois pour le temps de recherche (à gauche) et pour le nombre moyen d'occurrences par motif (à droite, grisée). La courbe correspondant à l'échelle de droite est également grisée. Les ordonnées de gauche et de droite utilisent une échelle logarithmique.

Nous commençons par étudier le cas de *M. flagellatus*, atypique par rapport aux comportements avec les autres textes. Le nombre d'occurrences est très faible pour ce texte, cela s'explique par la taille de celui-ci qui est très courte (moins de 3 millions de lettres contre plusieurs dizaines de millions pour les autres). On a en moyenne, un peu plus d'une occurrence par motif pour les longueurs 20 à 50. Cela nous permet d'observer le comportement des structures d'indexation lorsque le nombre d'occurrences est stable mais que la longueur du motif augmente. On observe, assez logiquement une légère augmentation du temps de recherche. Cette augmentation est de 57 %, entre la longueur 20 et la longueur 50, pour le FM-index de González, de 32 % pour le FM-index de Ferragina et Venturini et de 31 % pour le FM-index dynamique. Alors que la longueur du motif est multipliée par 2,5 le temps de recherche augmente relativement peu en comparaison. Pour comprendre ceci il faut revenir au principe de la recherche de motif dans le FM-index. Une première étape consiste à trouver l'intervalle contenant toutes les occurrences du motif. Le temps que prend cette étape est directement proportionnel à la longueur du motif. La seconde étape va rechercher, via l'échantillonnage de la table des suffixes, les positions de chacune des ces occurrences. Le temps que prend cette étape est indépendant de la taille du motif mais proportionnel à la distance d'échantillonnage choisie. Quel que soit le motif la première

[7] `http://pizzachili.dcc.uchile.cl/indexes/FM-indexV2/`

Fig. 4.5 — Temps moyen de recherche pour localiser des motifs dans différents textes avec des variantes de FM-index statiques ou dynamique



Abscisse	Longueur du motif
Ordonnée (gauche)	Temps (en ms) pour localiser toutes les occurrences
Ordonnée (droite)	Nombre d'occurrences

étape est réalisée une seule fois, en revanche la seconde est réalisée autant de fois qu'il y a d'occurrences. Ainsi pour tous les FM-index la première étape prend 2,5 fois plus de temps, en revanche la seconde reste constante : la distance d'échantillonnage n'a pas changée, elle est choisie à la construction, et le nombre d'occurrences évolue très peu entre les longueurs 20 à 50. On comprend donc que plus l'augmentation du temps de recherche est faible entre les facteurs de longueur 20 et 50, plus la seconde étape prend une part importante dans le temps total de la recherche du motif. À partir de cette observation on peut en déduire que environ 60 % du temps de recherche est consacré à la seconde étape, pour trouver une seule occurrence avec le FM-index de González contre près de 80 % pour les deux autres versions. Cela montre l'importance de la seconde étape dans le temps de recherche ainsi que l'inefficacité d'une technique comme l'échantillonnage. C'est malheureusement la seule solution qui existe pour tous les auto-index basés sur la table des suffixes.

Les résultats nous montrent, pour tous les autres textes (chr. 1 de la souris, *etexts*, afrikaans), une forte dépendance du temps de recherche dans le nombre moyen d'occurrences. Cela n'est pas surprenant au vu de l'explication précédente. Le temps pour localiser une seule occurrence (seconde étape) est prédominant par rapport au temps pour trouver l'intervalle contenant toutes les occurrences (première étape). En particulier lorsque le nombre d'occurrences est élevé, le temps pour la première étape devient négligeable par rapport à celui pour la seconde. Cela explique également que le temps de recherche chute alors que le motif devient plus grand : le temps supplémentaire pour réaliser la première étape est largement compensé par la baisse du nombre d'occurrences. La chute du temps de recherche est cependant un peu moins importante que celle du nombre d'occurrences (voir l'évolution des courbes —■— et —▲— par exemple), l'augmentation de la longueur du motif accroissant légèrement le temps de recherche.

De plus nous remarquons que le FM-index de González et notre version dynamique évoluent de la même façon, signifiant qu'ils sont impactés de façon similaire par le nombre d'occurrences. Cela n'est pas le cas du FM-index de Ferragina et Venturini dont le temps de recherche décroît moins rapidement avec le nombre d'occurrences. Dans ce cas, c'est le temps dévolu à la première étape qui prend une part importante du temps de recherche et la chute du nombre d'occurrences ne joue plus qu'un rôle marginal.

Comme nous nous y attendions notre version est plus lente que celle de González (une dizaine de fois), très probablement en raison de l'utilisation de structures dynamiques. En revanche, et c'est une surprise, notre version rivalise avec celle de Ferragina et Venturini. Notre implantation est même plus rapide lorsque le nombre d'occurrences est peu élevé (inférieur à 40), après quoi la distance d'échantillonnage assez importante que nous avons choisie nous ralentit d'autant plus que le nombre d'occurrences est important. Cependant, rappelons que l'implantation de Ferragina et Venturini a été faite dans un souci d'économie d'espace et, sur ce point, notre implantation ne rivalise pas puisque les structures que nous utilisons ne sont pas totalement compressées. Notons que nous pourrions avoir recours au gap-encoding pour améliorer l'espace utilisé par les champs de bits que nous utilisons pour stocker m_{TIS} et v_{TIS}.

4.2.4 Reconstruire ou mettre à jour ?

Lorsqu'on est amené à modifier un texte, il s'agit souvent d'y faire quelques modifications, éventuellement de manière fréquente. C'est dans ce cas qu'une structure d'indexation dynamique révèle son utilité. Nous étudions ici jusqu'à quel point il est intéressant d'avoir recours à notre structure d'indexation et déterminons le seuil à partir duquel la reconstruction s'avère plus rapide que la mise à jour. Pour cela nous étudions différentes tailles de modifications et augmentons le nombre de ces modifications jusqu'à dépasser le temps nécessaire à la reconstruction. Ainsi nous allons étudier trois cas :

- insertions courtes (1 lettre) ;
- insertions moyennes (20 lettres) ;
- insertions longues (400 lettres).

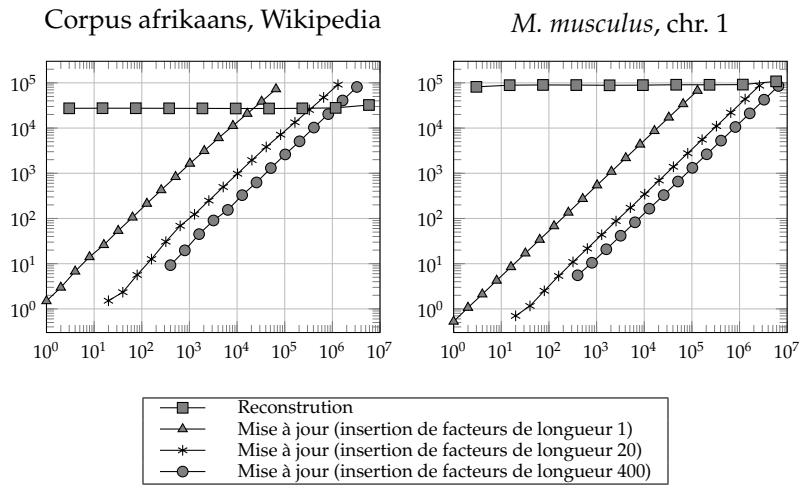
Pour chacune de ces longueurs d'insertions nous allons regarder à partir de quel nombre d'insertions, le temps pour mettre à jour la structure devient plus important que de la reconstruire en utilisant une structure statique. Pour cela nous insérons de plus en plus de facteurs de longueur 1 (ainsi que de longueur 20 et 400) jusqu'à avoir dépasser le temps de reconstruction. Nous exprimons les résultats en fonction de la longueur totale insérée qui est uniquement le résultat de la multiplication entre le nombre de facteurs insérés et la longueur de facteurs que l'on utilise. Pour chaque longueur totale, le calcul est renouvelé 100 fois. Pour cette étude nous utilisons deux textes différents : une séquence génomique (chromosome 1 de la souris) et un texte en langage naturel (version afrikaans du corpus Wikipedia). Les résultats sont présentés dans la FIG. 4.6 page suivante. L'abscisse et l'ordonnée utilisent une échelle logarithmique.

Nous remarquons directement que le temps d'insertion, pour une longueur donnée, est linéaire dans le nombre d'insertions effectué. Ce n'est pas étonnant, le temps aurait pu être calculé en utilisant le temps moyen d'insertion pour un facteur de longueur 1, par exemple, puis multiplié par le nombre de facteurs à insérer. Le résultat aurait été le même.

Nous notons également que pour une même longueur, l'insertion des facteurs de longueur 1 est plus longue que celle des facteurs de longueur 20 qui est plus longue que celle des facteurs de longueur 400. Là non plus, cela ne constitue pas une surprise. Le nombre d'insertions à effectuer, pour arriver à une même longueur, est 400 fois moins important en utilisant des facteurs de longueur 400 qu'avec des facteurs de longueur 1. Dans ce cas, il y a 400 fois moins d'étapes de réordonnements à réaliser. Même si cette étape peut durer plus longtemps pour un facteur de longueur 400 que pour un facteur de longueur 1, le nombre de réordonnements que l'on peut réaliser est de toute façon majoré (voir section 4.1.2.1 page 83) et ne pourra pas continuellement augmenter avec la taille du facteur. Ainsi dans le cas qui nous intéresse l'étape de réordonnement ne prend pas 400 fois plus de temps pour un facteur de longueur 400 que pour un facteur de longueur 1.

Pour le corpus afrikaans (68 989 658 lettres), le temps nécessaire à la reconstruction est atteint autour de 22 000 (soit 0,032 %) avec des facteurs de longueur 1, autour de 360 000 (0,52 %) pour des facteurs de longueur 20 et autour de 1 150 000 (1,7 %) pour la longueur 400. Pour le chromosome 1 de la souris (191 477 429 lettres), le seuil est atteint autour de 180 000 (0,094 %) pour des facteurs de longueur 1, autour de 3 100 000 (1,6 %) pour des facteurs

Fig. 4.6 — Comparaison des temps de reconstruction ou de mise à jour de la structure d'indexation pour le corpus afrikaans de Wikipedia et le chromosome 1 de la souris



de longueur 20 et autour de 8 000 000 (4,2 %) pour des facteurs de longueur 400. On s'aperçoit ainsi qu'avec des facteurs de longueur 1, il n'est intéressant de modifier qu'une infime partie du texte (moins de 0,1 % pour les longueurs considérées). Néanmoins 0,1 % de modifications avec des facteurs de longueur 1, pour un texte de quelques centaines de millions de lettres, cela représente déjà quelques centaines de milliers de modifications indépendantes. Par ailleurs, en utilisant des facteurs de taille plus grande il est possible de modifier une proportion bien plus importante du texte.

De plus, nous savons que le temps de reconstruction de la structure statique croît linéairement avec la taille du texte alors que le temps de mise à jour croît logarithmiquement. Ainsi, en multipliant la taille d'un texte par 10, le temps de reconstruction est multiplié d'autant. Le temps de mise à jour, lui, est multiplié par un nombre bien inférieur. Par exemple, dans la FIG. 4.4 page 97, pour le chromosome 1 de *M. musculus*, le temps de mise à jour est multiplié par 2 quand la taille du texte est multiplié par 10. Donc, quand la taille du texte est multipliée par 10, on a la possibilité d'insérer 5 fois plus de facteurs avant d'atteindre le seuil du temps de reconstruction.

Ces expériences permettent de voir que le nombre de modifications que l'on peut réaliser avant d'atteindre le temps de reconstruction de la structure statique est très important. Pour un texte d'une centaine de millions de lettres il est ainsi possible de modifier des centaines de milliers voire des millions de lettres, tout dépend de la longueur de chaque modification, sans avoir à reconstruire la structure. Néanmoins, en valeur relative à la taille du texte, ces nombres paraissent assez négligeables. D'autant plus que la proportion du

texte qu'il est possible de modifier sans dépasser le temps de reconstruction décroît quand la taille du texte croît. Cependant on peut considérer que le nombre de modifications à réaliser à un moment donné dans un texte n'est généralement pas (ou peu) dépendant de la taille de ce dernier.

4.2.5 Choisir la distance d'échantillonnage

Le FM-index est composé d'un échantillonnage et nous avons précisé qu'en théorie la distance entre deux échantillons est de $\log^{1+\epsilon} n$. Or toutes les méthodes reposant sur une technique d'échantillonnage utilisent, dans leur code, une valeur fixée plus moins arbitrairement définissant la distance entre deux échantillons de la table des suffixes. Nous allons donc procéder à des expériences permettant d'apprécier l'impact de la distance d'échantillonnage sur la structure en termes de consommation mémoire et de temps de modification.

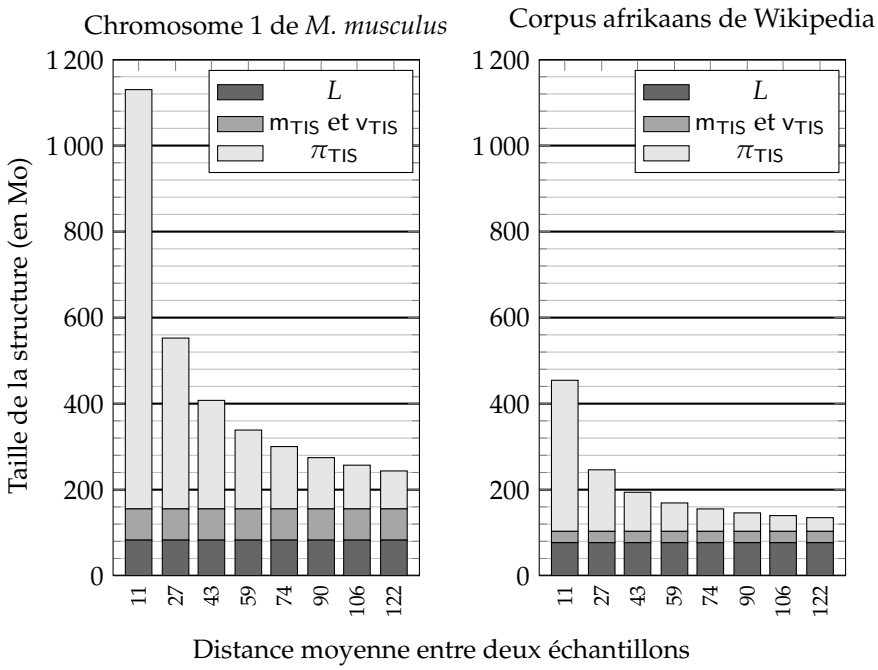
Nous commençons par étudier l'espace mémoire utilisé par notre index et pour cela expliquons quelques détails de l'implantation pour les structures le composant. Nous regardons comment la distance d'échantillonage influe sur l'espace mémoire utilisé par la structure totale. Ensuite nous nous intéressons à la variation du temps de modification en fonction de la distance d'échantillonnage.

4.2.5.1 Impact sur l'espace mémoire

Nous avons précédemment précisé que les champs de bits qui composent notre structure d'indexation pourraient être implantés de façon plus efficace. Après contacts avec G. Navarro, nous savons que de tels champs de bits compressés et dynamiques sont en cours de développement. Malgré ce manque, temporaire, dans notre structure d'indexation nous nous intéressons à sa consommation mémoire et à la façon dont elle est répartie entre les différentes structures. Nous décomposons notre structure d'indexation en trois composantes principales : la transformée de Burrows-Wheeler, stockée à l'aide d'un wavelet tree (section 2.2.1 page 49) ; la permutation π_{TIS} ; les champs de bits servant de masques m_{TIS} et v_{TIS}. Théoriquement c'est le première structure qui occupe le plus d'espace. Néanmoins, les champs de bits ne sont pas encore compressés efficacement ce qui pourrait rendre leur espace mémoire assez significatif. Le stockage de π_{TIS} est réalisé à l'aide de deux arbres binaires équilibrés, chaque nœud de l'un étant relié à un nœud de l'autre. En pratique, le stockage des arbres en utilisant des pointeurs est très consommateur d'espace mémoire (un pointeur vers le fils gauche, le fils droit, le père et, dans notre cas, un pointeur vers un nœud de l'autre arbre). Il est donc à craindre que la consommation mémoire de notre échantillonnage ne soit, en pratique, pas totalement négligeable face à la transformée de Burrows-Wheeler. Nous regardons, pour deux types de texte différents (séquence génomique et texte en langage naturel), la consommation mémoire de notre structure d'indexation en fonction de la distance d'échantillonnage ainsi que la répartition au sein des trois composantes que nous avons définies. Les résultats sont présentés en FIG. 4.7 page suivante.

Nous constatons que les résultats pour les séquences génomiques et les textes en langage naturel sont très différents. La raison tient au fait que les séquences génomiques sont définies sur un alphabet de taille 4 et peuvent donc

Fig. 4.7 — Espace consommé par la structure d'indexation en fonction de la distance d'échantillonnage choisie



être stockées en utilisant assez peu de place. C'est pourquoi L occupe environ 83 Mo pour une séquence génomique de 191 Mo alors que L occupe environ 77 Mo pour un texte en langage naturel de 69 Mo. En revanche les deux autres structures ne sont pas dépendantes du contenu du texte mais uniquement de sa longueur et de la distance d'échantillonnage choisie.

Les deux champs de bits m_{TIS} et v_{TIS} n'utilisent pas un espace mémoire démesuré comparé à L. Ainsi dans le cas de la séquence génomique ils utilisent un espace mémoire correspondant à 88 % de la taille de L et pour le texte en langage naturel, cela descend à 34 %. Nous savons que ces résultats pourraient être améliorés facilement avec l'utilisation de champs de bits utilisant le gap-encoding. Mais ces champs de bits ne sont pas la composante principale de l'espace mémoire utilisé par notre structure d'indexation.

Les résultats confirment nos craintes concernant l'espace utilisé pour stocker π_{TIS}. Stocker un élément sur 11 dans π_{TIS} va nous coûter près de 975 Mo, pour le chromosome 1 de *M. musculus*. Ceci représente une moyenne de 56 octets par valeur stockée dans π_{TIS}. Cette moyenne est bien entendu la même pour d'autres distances d'échantillonnage ou pour d'autres textes. Une moyenne aussi élevée pose évidemment un problème puisqu'elle ne nous permet pas d'utiliser des distances d'échantillonnage plus courtes, pour accélérer le temps de recherche. Par exemple il ne semble pas raisonnable d'utiliser des distances

d'échantillonnage inférieures à 90, la consommation mémoire de π_{TIS} devenant très significative par rapport à L et aux champs de bits.

Il existe des solutions plus compactes pour stocker un arbre binaire dynamique. Par exemple Raman et Rao (2003) permettent de stocker un arbre binaire dynamique en utilisant $(2b+2)n + o(n)$ bits, où $b = O(\log n)$ désigne l'espace nécessaire pour stocker les données que l'on peut associer à chaque nœud. Cette solution permettrait de stocker efficacement la structure d'arbre et d'éviter d'avoir à stocker de nombreux pointeurs vers les parents et descendants directs. Malheureusement il n'existe pas, à l'heure actuelle, d'implantation de cette méthode.

4.2.5.2 Impact sur le temps d'accès à l'échantillonnage

L'échantillonnage est utilisé lorsqu'on souhaite connaître une valeur de TS ou de TIS. Nous avons besoin de ces valeurs à deux occasions : lors de la mise à jour, pour savoir à quel endroit on doit faire la première modification dans la TBW ; lors de la recherche des positions des occurrences, les positions sont obtenues à partir de l'échantillonnage. Pour apprécier l'impact de modifications de la distance d'échantillonnage sur ces opérations, nous procédons à des expériences sur les textes considérés précédemment pour l'évaluation de l'espace mémoire consommé. Pour cela nous allons indexer un même texte à plusieurs reprises en utilisant différentes distances d'échantillonnage et récupérer 1 000 000 de valeurs de TS à des positions aléatoires. Nous présentons les résultats en FIG. 4.8 page suivante. Nous traçons une droite grisée entre le premier et le dernier point de chacune des courbes.

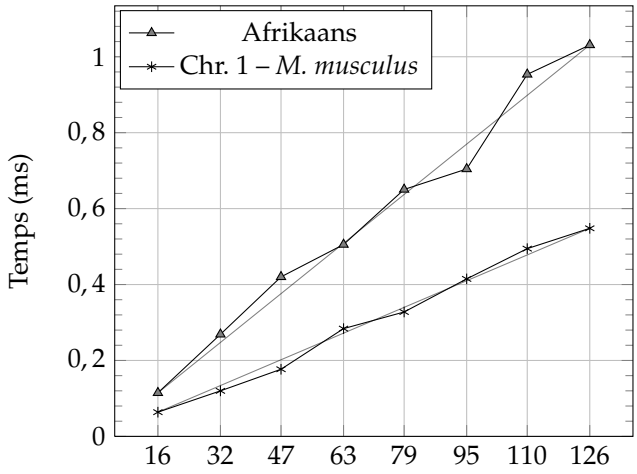
Plus la distance d'échantillonnage est grande plus le temps nécessaire pour récupérer une valeur de TS est important. La distance moyenne d'échantillonnage définit le nombre moyen d'appels à la fonction LF avant d'obtenir la valeur de TS que l'on voulait connaître. Cela explique également pourquoi le temps moyen est plus important pour le texte en langage naturel que pour la séquence génomique, bien que le premier soit plus court que le second. La fonction LF a une complexité, dans notre implantation, en $O(\log n \log \sigma)$. Le ralentissement constaté pour les textes en langage naturel par rapport aux séquences génomiques s'explique donc uniquement par la taille de l'alphabet.

Avec ces expériences on constate que récupérer une seule valeur de la table des suffixes nécessite plus de 0,5 ms pour des textes en langage naturel et des distances d'échantillonnage supérieures à 70. Ce temps est à comparer au temps de recherche du FM-index statique (de González) dans la FIG. 4.5 page 99. Le temps total de recherche est inférieur à 0,2 ms pour les motifs de longueur 50 (qui ont 5,7 occurrences en moyenne) sur le corpus afrikaans de Wikipedia.

4.2.5.3 Conclusion

Les expériences que nous avons menées en faisant varier la distance d'échantillonnage nous montrent qu'un point très important à améliorer dans notre structure d'indexation dynamique est l'échantillonnage. Celui-ci dégrade à la fois les performances en temps de la structure lors des recherches de motifs et les performances en espace, avec un échantillonnage qui prend trop de place. Une piste, que nous évoquée, serait d'utiliser une structure plus compacte

Fig. 4.8 — Temps moyen pour récupérer une valeur de TS en fonction de la distance d'échantillonnage



pour le stockage de l'échantillonnage. Avoir une structure plus économe en espace permet d'utiliser une distance d'échantillonnage plus faible et ainsi d'avoir un temps d'accès à une valeur de TS plus court et donc un temps de recherche plus compétitif.

Par ailleurs, nous l'avons évoqué, une autre piste d'amélioration est le stockage de la permutation π_{TIS} en utilisant une méthode plus économe. Cette modification permettrait à la fois un gain en espace et un gain en temps, en réduisant la distance d'échantillonnage utilisée.

4.3 Conclusion

Nous avons montré à la fois par des expériences et par des aspects théoriques que notre algorithme réordonne uniquement un petit nombre d'éléments par rapport à la taille du texte. Nos expériences ont montré l'avantage net de la mise à jour par rapport à la reconstruction. Notre structure d'indexation dynamique et compressée semble être une bonne alternative pour l'indexation de textes amenés à être modifiés. Néanmoins, la dynamicité est obtenue au prix d'un ralentissement pour le temps de recherche, en raison des structures utilisées. Il semble donc peu avantageux d'utiliser cette structure d'indexation dynamique pour des textes amenés à être modifiés très rarement. Néanmoins, notre structure continuera à bénéficier des améliorations qui pourront être apportées au stockage des champs de bits et champs de lettres. Ainsi, au vu des nombreuses avancées apportées dans le domaine ces dernières années, on peut espérer que les temps d'exécution vont encore

s'améliorer dans un futur proche par rapport aux tests réalisés à ce jour. À ce propos, une piste intéressante pourrait être une adaptation de la structure de Gupta *et al.* (2007b) (voir section 2.2.5.1 page 55). Leur structure n'est pas compressée mais permettrait de significativement améliorer le temps de recherche, la fonction LF pouvant être calculée en temps $O(1)$, dans ce cas. Leur structure étant adaptée aux séquences qui ont peu de modifications, elle ne pourrait être utilisée que si les modifications sur le texte sont minimes.

Chapitre 5

Quelques applications possibles aux structures d'indexation dynamiques ou compressées

Jusqu'à maintenant, nous nous sommes principalement concentrés sur les techniques utilisées au sein des structures d'indexation compressées et à une utilisation très simple de celles-ci : la recherche de motifs. Désormais, nous nous concentrons sur des aspects moins triviaux des structures d'indexation : une extension du principe de l'indexation aux séquences numériques (section 5.1) ; l'utilisation des structures d'indexation pour la compression de données, avec la factorisation d'un texte selon le LZ-77 (section 5.2 page 119) ; l'application à la bioinformatique (section 5.3 page 124).

5.1 Recherche du minimum sur un intervalle

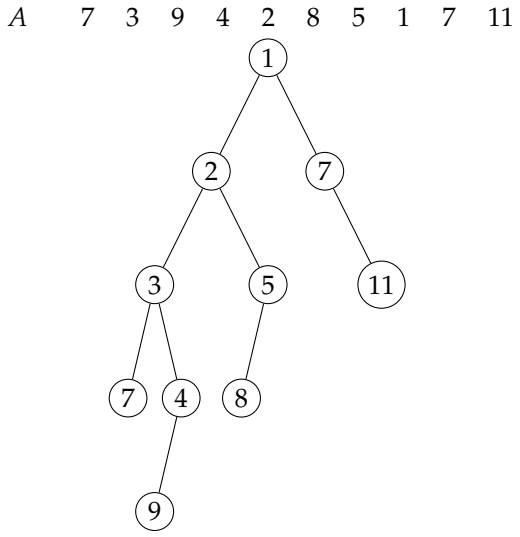
Le problème de la recherche du minimum dans un intervalle (RMI) consiste à trouver la valeur minimale, dans un intervalle donné, parmi un tableau A de n nombres qu'il est possible de pré-traiter. Ce problème est un problème d'indexation de données et il a de nombreuses applications : en indexation de textes (pour connaître la plus petite valeur PLPC, par exemple) ; en compression de données (avec la LZ-factorisation voir les travaux de Okanohara et Sadakane (2008) ; Chen *et al.* (2008) par exemple) ; en recherche de documents.

Nous commençons par présenter le principe utilisé par l'ensemble des solutions existant à ce jour, toutes basées sur les arbres cartésiens (section 5.1.1.1 page suivante). Ensuite, nous introduisons notre méthode, utilisant les minimums locaux, et nous montrons comment nous pouvons l'utiliser afin de fournir une méthode autorisant les mises à jour (section 5.1.2 page 111).

5.1.1 Arbres cartésiens

Un *arbre cartésien* est un arbre binaire construit sur un tableau d'entiers A de longueur n. L'arbre cartésien possède n nœuds, chacun correspondant à un nombre dans le tableau A. La racine de l'arbre cartésien contient la valeur minimale de A. Si cette valeur est en position i, le sous-arbre gauche de l'arbre

Ex. 5.1 — Arbre cartésien



est construit récursivement sur $A[\,.\,.\,i-1]$ et, le sous-arbre droit de l'arbre est construit sur $A[i+1\,.\,.]$ (voir Ex. 5.1).

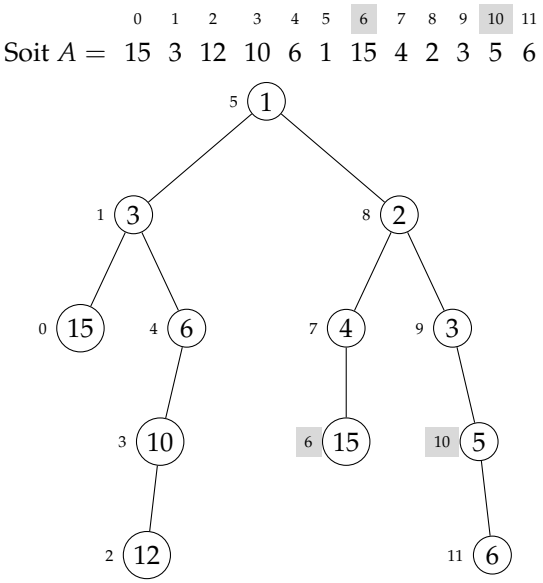
5.1.1.1 Approche originelle

La première solution pour le problème des RMI est due à Gabow *et al.* (1984) et utilise le fait que le problème des RMI peut être ramené au problème du plus petit ancêtre commun (PPAC) dans un arbre. Ce problème consiste, étant donnés deux nœuds d'un arbre, à trouver l'ancêtre commun à ces deux nœuds, le plus proche. Or on sait pré-traiter un arbre afin de répondre aux requêtes PPAC en temps constant (Bender et Farach-Colton, 2004). Ainsi répondre à la requête de RMI peut être réalisé très simplement en stockant le tableau d'entiers à l'aide d'un arbre cartésien pré-traité pour répondre aux requêtes du PPAC (voir Ex. 5.2 page ci-contre). Nous avons donc une solution au problème de RMI en temps constant. Cependant la représentation de l'arbre cartésien et les informations nécessaires au calcul du PPAC en temps constant sont gourmandes en espace mémoire. Les travaux ultérieurs sur le sujet ont donc tenté de réduire celui-ci.

5.1.1.2 Améliorations

Ce problème a fait l'objet de nombreuses publications ces derniers temps Fischer et Heun (2006, 2007, 2008) ; Fischer *et al.* (2008) ; Fischer (2009). Dans les améliorations les plus récentes, le tableau d'entiers est découpé en blocs et superblocs (de façon similaire à ce qui se fait pour les champs de bits). Des résultats de RMI sont pré-calculés pour les requêtes entre superblocs et entre blocs. Il reste ensuite à traîter le cas des requêtes à l'intérieur d'un bloc. Pour ce faire, Fischer et Heun (2007) donnent un type à chaque bloc. Ce type est défini

Ex. 5.2 — Recherche de minimum dans un intervalle, avec les arbres cartésiens



Calcul de RMI entre les positions 6 et 10
Le plus petit ancêtre commun entre les nœuds 6 et 10 est le nœud 8, dont la valeur est 2. Le minimum dans l'intervalle [6, 10] est donc 2.

---

en fonction de l'arbre cartésien qui le représente : deux blocs ayant un arbre cartésien équivalent auront le même type. La position du minimum dépend uniquement de la structure de l'arbre et pas du contenu des nœuds. C'est cet élément qu'utilisent Fischer et Heun pour leur algorithme. Une fois les types stockés, il suffit d'avoir une table donnant la position du minimum, à l'intérieur d'un bloc, pour n'importe quelle position, en fonction du type du bloc.

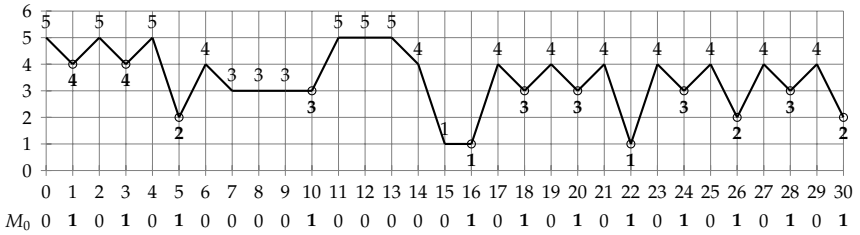
Par la suite Fischer *et al.* (2008) ont amélioré l'algorithme en stockant les types des blocs sous forme compressée. Ceci permet d'obtenir la première structure d'indexation compressée, pour les RMI, qui répond à une requête en temps constant.

5.1.2 Minimums locaux

Nous nous intéressons maintenant à une méthode que nous avons mise au point. Cette méthode a un double intérêt : elle repose sur un principe différent des arbres cartésiens et permet, à ce titre, d'ouvrir de nouvelles perspectives ; elle laisse la possibilité de mettre à jour la structure d'indexation.

Ex. 5.3 — Représentation graphique d'un tableau d'entiers

Soit A= 5 4 5 4 5 2 4 3 3 3 3 5 5 5 4 1 1 4 3 4 3 4 1 4 3 4 2 4 3 4 2



5.1.2.1 Méthode statique

Pour mieux appréhender cette méthode, représentons un tableau A d'entiers sous forme de graphique, où $A[i]$ correspond à l'ordonnée et i à l'abscisse (voir Ex. 5.3). Dans cet exemple, nous avons représenté en gras les minimum locaux. En dessous du graphique, un champ de bits résume, de façon succincte, les positions qui sont des minimums locaux et celles qui ne le sont pas. Soit M_0 le champ de bits et n la longueur du tableau A. Par commodité, nous écrivons $A_0 = A$. Nous définissons les minimums locaux ainsi :

$$\begin{array}{rll} M_0[0] = 1 & \text{si } A_0[0] < A_0[1] \\ M_0[n-1] = 1 & \text{si } A_0[n-2] > A_0[n-1] \\ M_0[k] = 1 & \text{si } A_0[k] < A_0[k+1] \text{ et il existe un } k' \text{ tel que } A_0[k'-1] > A_0[k'], \\ & \text{et } A_0[k''] = A_0[k], \text{avec } k' \leq k'' \leq k. \\ 0 & \text{sinon} \end{array}$$

Considérons maintenant la sous-séquence A_1 issue de A_0 et composée uniquement des valeurs correspondant à des minimums locaux. Pour notre exemple, on a A_1 = 4 4 2 3 1 3 3 1 3 2 3 2, on réitère le processus pour cette séquence. Celui-ci s'achève lorsqu'une séquence contient au maximum un minimum local. Dans l'Ex. 5.4 page ci-contre, nous montrons les graphiques pour les séquences A_1 et A_2.

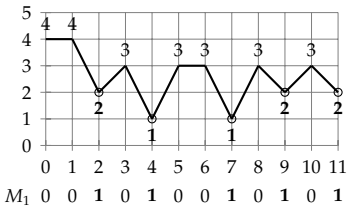
Déterminer un minimum Pour déterminer la valeur minimum dans un intervalle il est indispensable de parcourir les niveaux en partant du premier et en allant vers le dernier. Supposons que l'on souhaite connaître le minimum entre les positions i et j dans A, que l'on notera $rmi_A(i,j)$. On peut définir cette opération récursivement : le résultat est le minimum entre la valeur en position i, en position j et du minimum obtenu récursivement sur A_1 entre les positions qui correspondent à i et j. Autrement dit, avec $0 \leq k < \lceil \log n \rceil$, on a :

$$rmi_{A_k}(i,j) = \begin{cases} A_k[i] & \text{si } i = j \\ \min \begin{pmatrix} A_k[i], A_k[j], \\ rmi_{A_{k+1}}(\mathsf{rank}_1(M_k, i), \mathsf{rank}_1(M_k, j-1) - 1)) \end{pmatrix} & \text{sinon} \end{cases}$$

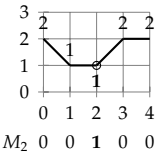
On donne un exemple de recherche de minimum dans l'Ex. 5.5 page suivante.

Ex. 5.4 — Représentation graphique des minimums locaux de A_1 et de A_2

$A_1 = 4\,4\,2\,3\,1\,3\,3\,1\,3\,2\,3\,2$



$A_2 = 2\,1\,1\,2\,2$



Ex. 5.5 — Exemple de recherche de minimum

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Soit A=	5	4	5	4	5	2	4	3	3	3	3	5	5	5	4	1	1	4	3	4	3	4	1	4	3	4	2	4	3	4	2

On veut connaître $rmi_A(2, 15)$:

1. $A[2] = 5$ et $A[15] = 1$, on a $\mathsf{rank}_1(M_0, 2) = 1$ et $\mathsf{rank}_1(M_0, 14) = 4$. On recherche alors $rmi_{A_1}(1, 3)$;
2. $A_1[1] = 4$ et $A_1[3] = 3$, on a $\mathsf{rank}_1(M_1, 1) = 0$ et $\mathsf{rank}_1(M_1, 2) = 1$. On recherche alors $rmi_{A_2}(0, 0)$;
3. $A_2[0] = 2$;
4. le résultat est $\min(5, 1, rmi_{A_1}(1, 3)) = \min(5, 1, 3, 4, rmi_{A_2}(0, 0)) = \min(5, 1, 3, 4, 2) = 1$.

Complexités Nous avons, au maximum, un élément sur deux qui est un minimum local à chaque niveau. Dans ces conditions, nous avons $\log n$ niveaux et l'ensemble des champs de bits totalise $2n - o(n)$ bits. Ces champs de bits doivent être pré-traités afin de répondre efficacement aux requêtes rank et select. Ce qui nous amène donc à une complexité en espace en $2nH_0 + o(n)$ bits. Il est également nécessaire d'accéder aux valeurs des séquences A_1, A_2,...

Espace Pour ce faire, il est possible de stocker certains A_i, par exemple une séquence toutes les $\log \log n$. Si A_i n'est pas échantillonné et que l'on doit récupérer la valeur $A_i[j]$, il est nécessaire de retourner dans les séquences précédentes pour récupérer la valeur correspondante. Sachant qu'on a $A_i[j] = A_{i-1}[\mathsf{select}_1(M_{i-1}, j)]$, n'importe quelle valeur pourra être retrouvée (voir Algo. 5.1 page suivante). En considérant une séquence toutes les $\log \log n$, le nombre to-

Algo. 5.1 — Récupération d'une valeur non échantillonnée pour la RMI

Assure que la valeur de $A_i[j]$ est retournée.

```
1: fonction RECUPEREVALEUR(i, j)
2:     si A_i est échantillonné alors
3:         retourner A_i[j]
4:     fin si
5:     retourner RECUPEREVALEUR(i − 1, select_1(M_{i−1}, j))
6: fin fonction
```

tal d'entiers à stocker serait de :

$$\sum_{i=1}^{\frac{\log n}{\log\log n}} \frac{n}{\log^i n} = \frac{n-1}{\log(n)-1}$$

Sachant que les entiers sont stockés sur $\log n$ bits, le stockage de ces séquences prendrait $n + o(n)$ bits supplémentaires.

Une autre solution, évitant l'échantillonnage, serait de n'avoir que des champs de bits de même longueur. Ceci permet de savoir, pour n'importe quel niveau, à quelle position dans la séquence d'origine, se trouve le minimum local. Ces champs de bits seraient creux et pourrait être efficacement compressés (voir Ex. 5.6 page suivante). Dans le pire des cas, à chaque niveau, une valeur sur deux est un minimum local. Voici l'entropie empirique d'ordre 0 dans ce pire des cas pour la concaténation M' des champs de bits M'_0, …, $M'_{\log(n)-1}$:

$$H_0(M') = \sum_{i=1}^{\log n} \frac{i}{2^i} + (1 - \frac{1}{2^i}) \log\left(\frac{1}{1-\frac{1}{2^i}}\right) < 3,16$$

Ainsi avec un codage entropique optimal, les champs de bits creux occuperaient, au pire, $3,16n$ bits.

Temps En ce qui concerne la complexité en temps, nous devons dans le pire des cas visiter $\log n$ niveaux. À chaque niveau, nous devons récupérer deux valeurs et effectuer des opérations rank ou select. La récupération des valeurs se fait soit en temps $O(\log\log n)$ avec le stockage des séquences ; ou en temps $O(1)$ avec les champs de bits creux.

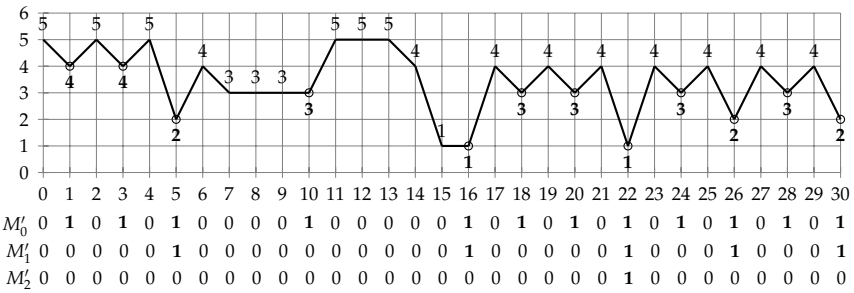
On en arrive donc au compromis espace-temps suivant :

	Espace	Temps
Stockage des séquences	$2nH_0 + n + o(n)$	$O(\log n \log\log n)$
Champs de bits creux	$< 3,16n + o(n)$	$O(\log n)$

Notons que, aussi bien pour la complexité en espace qu'en temps, il s'agit de pire des cas. Ainsi il est extrêmement rare d'avoir $\log n$ niveaux : il faudrait des séquences d'entiers très particulières. Par exemple soit une séquence A, on prend deux entiers consécutifs dans cette séquence et on insère le maximum

Ex. 5.6 — Utilisation des champs de bits creux pour la RMI

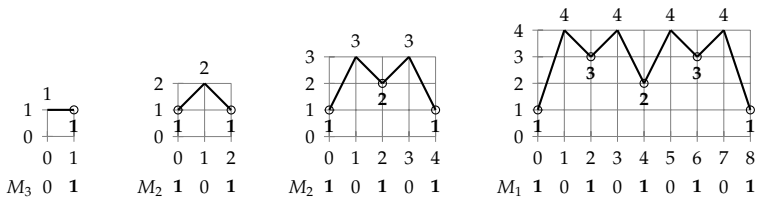
Soit $A = 5\,4\,5\,4\,5\,2\,4\,3\,3\,3\,3\,5\,5\,5\,4\,1\,1\,4\,3\,4\,3\,4\,1\,4\,3\,4\,2\,4\,3\,4\,2$



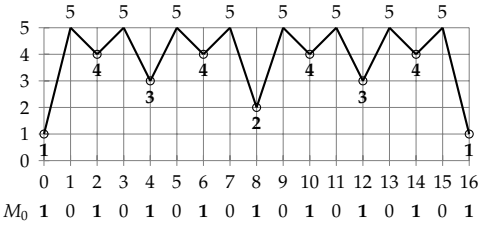
Tous les champs de bits ont la même taille. On peut observer qu'un seul minimum local est présent dans le dernier champ de bits et celui-ci est en position 22, c'est-à-dire qu'il correspond au troisième 1 dans A.

Ex. 5.7 — Cas particulier pour lequel une valeur sur deux est un minimum local, et ceci à tous les niveaux

$A_4 = 1\,1$ $A_3 = 1\,2\,1$ $A_2 = 1\,3\,2\,3\,1$ $A_1 = 1\,4\,3\,4\,2\,4\,3\,4\,1$



$A_0 = 1\,5\,4\,5\,3\,5\,4\,5\,2\,5\,4\,5\,3\,5\,4\,5\,1$



+1 entre ces deux entiers et on renouvelle le processus sur la séquence ainsi obtenue. En partant de la séquence 1 1, on obtient la séquence A_0 de l'Ex. 5.7 au bout de quatre étapes.

À l'opposé, en supposant que l'on ait une séquence croissante d'entiers, il n'y aurait qu'un seul minimum local et donc qu'un seul niveau. Le stockage prendrait, pour une telle séquence, $o(n)$ bits et les requêtes seraient réalisées en temps $O(1)$. Les complexités en espace et en temps dépendent donc de la

séquence, et plus précisément de ses minimums locaux. La complexité en espace de notre structure peut atteindre des espaces inférieurs à une représentation succincte. Cette propriété correspond à celle d'une structure compressée.

Résultats Afin de comparer notre méthode aux méthodes existantes, nous l'avons implantée. Pour notre algorithme, nous avons choisi la méthode consistant à stocker un certain nombre de séquences A_i. Nous avons utilisé les champs de bits conçus par González *et al.* (2005). Malheureusement ces champs de bits ne sont pas compressés et d'autres structures pourraient être plus adéquates comme celle de Gupta *et al.* (2007a) [1] mais leur implantation présente des incohérences. Néanmoins ces derniers temps, d'autres solutions apparaissent telles que Sux, de Vigna (2008) [2] ou SDSL de S. Gog [3]. Il pourrait être intéressant d'utiliser ces structures pour implanter notre méthode avec des champs de bits creux.

Nous avons comparé ces résultats avec l'implantation disponible, la plus récente. Celle-ci est issue de Fischer et Heun (2007) et est due à J. Fischer [4]. Très récemment, une autre implantation a été rendue disponible sur sa page personnelle, correspondant à une publication de Fischer (2009) non parue. En raison de problèmes de compilation de ce projet et de l'apparition récente de ce code, nous ne pouvons l'utiliser pour la comparaison.

Notre code et celui de Fischer sont écrits en C/C++ et ont été compilés avec `gcc` version 4.3.2. Les mesures en temps ont été réalisés avec la fonction `gettimeofday` alors que le programme s'exécutait avec une priorité maximale. Les mesures en espace ont été réalisées à l'aide des fonctions fournies par les deux structures pour connaître leur consommation mémoire. Il a été contrôlé avec `memcheck` que ces fonctions retournaient un résultat correct. Dans la FIG. 5.1 page suivante nous présentons un graphique montrant le résultat pour la structure de Fischer et Heun et le compromis espace-temps que nous obtenons avec notre structure en jouant sur le nombre de niveaux échantillonnés. Les structures sont construites pour une séquence aléatoire de dix millions d'entiers, cent mille requêtes aléatoires sont ensuite exécutées sur ces structures. Un temps moyen est calculé à partir du temps observé pour chacune des requêtes.

Dans le graphique nous observons que la méthode de Fischer et Heun, contrairement à la notre, ne suit pas une courbe mais correspond à un seul point. Ceci s'explique simplement par le fait que leur méthode n'offre pas de compromis espace-temps [5]. La courbe correspondant à notre méthode, au contraire, est typique des courbes de compromis espace-temps : autour de 2,5 Mo on a une dégradation significative du temps de recherche pour un gain modique en espace. À l'opposé entre 7,3 et 6 μs, l'amélioration en temps est faible mais la consommation mémoire double quasiment. Ainsi, pour cette séquence aléatoire, l'optimum semble se situer autour de 3 Mo, une requête prend alors environ 8 μs (quand la méthode de Fischer et Heun utilise 7,5 Mo

[1] `http://www.cs.duke.edu/~agupta/DictionaryCode/`

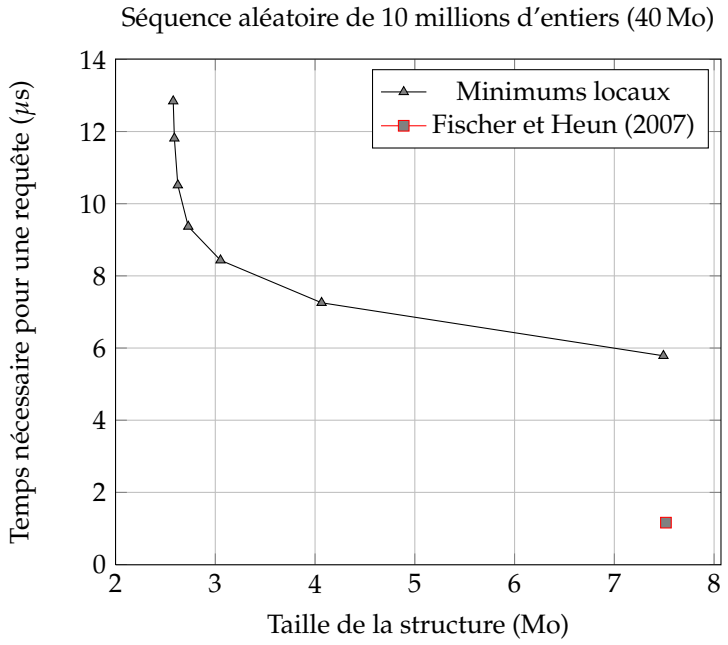
[2] `http://freshmeat.net/projects/succinct`

[3] `http://www.uni-ulm.de/in/theo/research/sdsl.html`

[4] `http://www-ab.informatik.uni-tuebingen.de/people/fischer`

[5] Ce n'est pas tout à fait exact, puisqu'il est théoriquement possible de modifier les paramètres régissant la taille des blocs ou superblocs pour arriver à moduler l'espace mémoire consommé et le temps de recherche. Cependant l'implantation de Fischer ne gère pas de telles modifications.

Fig. 5.1 — Comparaison des solutions pour la recherche de minimum dans un intervalle sur des données alétoires



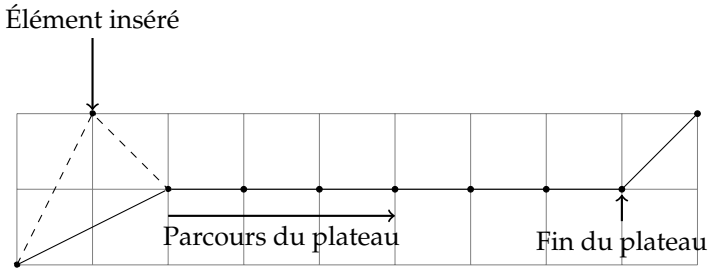
pour une requête en 1,1 μs). Notre méthode constitue donc une avancée certaine en terme d'espace mémoire : elle est plus de deux fois plus économe que la solution de Fischer et Heun, sans subir une dégradation de performances trop importante. De plus nous rappelons que nous utilisons pour l'instant une méthode non compressée pour le stockage des champs de bits. On peut donc penser qu'en utilisant les méthodes décrites par Okanohara et Sadakane (2007) ou celles implantées par Claude et Navarro (2008), la consommation en espace sera moindre, au prix, éventuellement, d'un ralentissement en temps (ce qui donnerait un autre compromis espace-temps).

5.1.2.2 Méthode dynamique

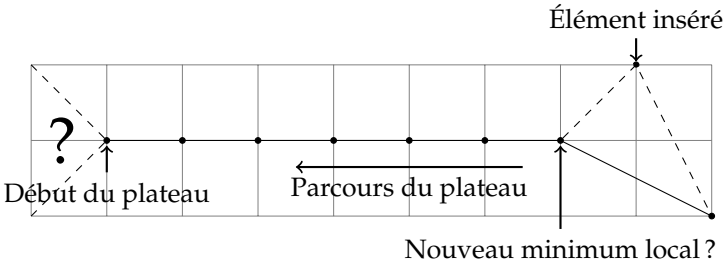
Nous présentons maintenant la façon de mettre à jour notre structure afin de prendre en compte des modifications de la séquence d'origine. L'insertion ou la suppression d'un entier dans la séquence ne peut pas changer un grand nombre de minimums locaux : à chaque niveau il ne peut y avoir qu'un minimum local en plus ou en moins.

Dans le cas d'une insertion, pour chaque niveau il nous faut déterminer si un minimum local doit être ajouté ou si un doit être déplacé. En revanche à partir du moment où il n'y a pas de minimum local à ajouter, aucune modification n'aura lieu dans les niveaux suivants.

Fig. 5.2 — Cas complexes lors de l'insertion d'une valeur dans la séquence d'entiers originelle pour la mise à jour de la structure permettant la RMI



À la fin du plateau, on a une montée, il faut donc ajouter un minimum local à cet endroit.



On se demande si l'élément précédant l'élément inséré doit être un minimum local. Pour cela il faut savoir si avant le plateau on avait une montée ou une descente. Dans le premier cas, on aura bien un nouveau minimum local, pas dans le second.

Au niveau des structures de données, il nous faut utiliser des champs de bits dynamiques afin de gérer efficacement les mises à jour. Ces structures sont plus lentes d'un facteur logarithmique que leurs équivalents statiques pour les requêtes et les accès aux champs de bits. Ainsi les complexités en temps évoquées plus haut doivent être multipliée par $\log n$, dans le cas de la dynamicité.

L'insertion s'avère plus complexe lorsque le point d'insertion est à droite ou à gauche d'un *plateau*. On appelle plateau une suite de valeurs identiques. Dans la FIG. 5.2 nous présentons deux cas, où l'insertion nécessite de parcourir la totalité du plateau afin de savoir si l'on doit ou non ajouter un minimum local.

Même si on ne s'attend pas à trouver de nombreux et longs plateaux dans les séquences à indexer, on peut se demander quel serait l'intérêt d'indexer des séquences ayant une telle régularité, il serait intéressant de mener une étude similaire à celle que nous avons conduite pour notre algorithme de mise à jour du FM-index (voir section 4.2.1).

5.1.3 Conclusion

Nous avons mis au point une méthode novatrice dans le domaine de recherche de minimum qui s'avère compétitive en espace par rapport à des solutions récentes. Notre méthode a également l'avantage de supporter les mises à jour, ce qui n'était pas envisageable jusqu'à maintenant. Par ailleurs, notre méthode bénéficiera de toutes les avancées dans le domaine des champs de bits creux (et dynamiques). Notre approche étant nouvelle pour la recherche de minimum dans un intervalle, il est probable que la publication de ces résultats amène d'autres chercheurs à améliorer notre méthode, la rendant encore plus compétitive par rapport aux solutions alternatives.

5.2 LZ factorisation

5.2.1 Algorithme

Nous commençons par présenter l'algorithme de compression introduit par Ziv et Lempel (1977) puis amélioré par Storer et Szymanski (1982). Le principe de l'algorithme est de stocker à l'aide d'un couple (p, ℓ) chaque facteur répété dans le texte. p correspond au déplacement de p positions vers la gauche qu'il faut effectuer pour retrouver le même facteur et ℓ désigne la longueur de ce facteur.

Soit un texte T et deux fenêtres contiguës g et d de longueurs f_g et f_d. Le principe est toujours le même : trouver le plus long préfixe de d qui est un facteur de g. Supposons que le préfixe trouvé soit de longueur k et commence à i positions de d, il sera alors codé avec un couple (i, k). Ensuite, les deux fenêtres sont décalées vers la droite de k positions. Si aucun préfixe de d n'est un facteur de g, on code alors directement la première lettre de d et les deux fenêtres sont décalées d'une position. Si g est un préfixe du texte, sa longueur peut être inférieure à f_g, pour permettre le codage des premiers caractères. Réciproquement, si d est un suffixe du texte, sa longueur peut être inférieure à f_d pour permettre le codage des derniers caractères. De cette façon les deux fenêtres, au fur et à mesure de leur progression, recouvrent l'ensemble du texte. La suite de couples et caractères obtenus après le parcours complet du texte correspond à la LZ-factorisation (voir Ex. 5.8 page suivante).

Pour trouver le plus long préfixe de d qui est un facteur de g, les structures d'indexation semblent toutes indiquées. Par exemple Chen *et al.* (2008) utilisent une table des suffixes et une structure de RMI sur cette table. Ou encore Crochemore *et al.* (2008) utilisent également une table des suffixes et une table PLPC, pour calculer la LZ-factorisation. Mais la longueur de leurs fenêtres étant illimitée, les structures d'indexation utilisées engendrent une consommation mémoire importante, ce qui peut être problématique pour certaines applications.

Lorsque les fenêtres sont de longueurs limitées (et significativement inférieures à la longueur du texte), leur contenu change après la production de chaque couple ou lettre. Des caractères sont retirés du début de chaque fenêtre et d'autres sont ajoutés à la fin. Les structures d'indexation dynamiques semblent donc beaucoup plus adaptées à cette situation.

Ex. 5.8 — Compression par LZ-77

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$T=$	C	T	A	C	T	A	G	G	T	C	T	C	T	A	$

On prend $f_g = 6$ et $f_d = 3$.

															Sortie
$d \leftarrow$	C	T	A	C	T	A	G	G	T	C	T	C	T	A	$	C
$g \leftarrow$	C	T	A	C	T	A	G	G	T	C	T	C	T	A	$	T
	C	T	A	C	T	A	G	G	T	C	T	C	T	A	$	A
	C	T	A	C	T	A	G	G	T	C	T	C	T	A	$	(3,3)
	C	T	A	C	T	A	G	G	T	C	T	C	T	A	$	G
	C	T	A	C	T	A	G	G	T	C	T	C	T	A	$	(1,1)
	C	T	A	C	T	A	G	G	T	C	T	C	T	A	$	(4,1)
	C	T	A	C	T	A	G	G	T	C	T	C	T	A	$	(6,2)
	C	T	A	C	T	A	G	G	T	C	T	C	T	A	$	(2,2)
	C	T	A	C	T	A	G	G	T	C	T	C	T	A	$	A
	C	T	A	C	T	A	G	G	T	C	T	C	T	A	$	$
	C	T	A	C	T	A	G	G	T	C	T	C	T	A	$	

Lorsque aucun facteur n'est trouvé dans g, le premier caractère de d est produit en sortie. Sinon c'est un couple donc la première composante est la distance entre le début du facteur dans g et le début de d et la seconde la longueur du facteur trouvé.
La LZ-factorisation pour le texte T est CTA(3,3)G(1,1)(4,1)(6,2)(2,2)A$, qui peut aussi être écrite de façon plus explicite (mais moins compacte) C·T·A·CTA·G·G·T·CT·CT·A·$.

5.2.2 Solutions existantes

Ainsi Okanohara et Sadakane (2008) réalisent une construction incrémentale de la table des suffixes et de la transformée de Burrows-Wheeler, en utilisant des champs de bits dynamiques. Or ceci n'est possible qu'en réordonnant les suffixes au fur et à mesure, comme nous l'avons expliqué dans notre algorithme de mise à jour de la table des suffixes. Pour éviter cela, Okanohara et Sadakane utilisent une table des préfixes au lieu d'une table des suffixes. La table des préfixes est définie de façon similaire à une table des suffixes : les miroirs des préfixes sont triés dans l'ordre alphabétique (ba est plus petit que ab). La transformée de Burrows-Wheeler est définie selon la même logique et un champ de lettres dynamique est utilisé sur celle-ci afin de pouvoir calculer facilement la fonction LF. Cela permet de connaître simplement la position d'insertion des nouveaux préfixes. Une table PLPC est également utilisée, qu'il conviendrait d'appeler table PLSC, donnant la longueur des plus longs

suffixes communs. À l'aide de cette table PLSC et d'une version dynamique de la RMI, il est possible de connaître le plus long facteur trouvé. La version dynamique de la RMI qu'ils utilisent semble assez naïve, et nécessite $O(n)$ bits pour des requêtes en temps $O(\log^3 n)$. Mais ils ne s'étendent pas particulièrement sur les détails de leur méthode (trois lignes dans leur article). L'utilisation de ces structures dynamiques leur permet de simuler l'utilisation d'une fenêtre glissante. Cependant la consommation mémoire reste proportionnelle à la longueur du texte. En effet effacer des éléments poserait le problème de réordonner ceux qui restent.

Ferreira *et al.* (2009) proposent trois algorithmes tous basés sur la table des suffixes. Pour le premier, la table des suffixes est construite sur la fenêtre g. Le plus long préfixe de d est recherché dans g, à l'aide de la table des suffixes. Tant que d n'a pas été considéré entièrement, le procédé est recommencé sans que g ne soit décalé, ce qui ne correspond par tout à fait à la définition de la LZ-factorisation. Une fois que d a été complètement factorisé, g est décalé en conséquence, de f_d positions, et la table des suffixes est reconstruite. Cette méthode est très coûteuse puisqu'elle implique de reconstruire de nombreuses fois la table des suffixes.

Le second algorithme fonctionne de façon similaire mais ne reconstruit pas la table des suffixes à chaque fois. Au lieu de cela, elle est mise à jour. Pour ce faire, ils construisent la table des suffixes sur d, retirent de la table des suffixes de g les suffixes commençant aux d premières positions et ensuite fusionnent les deux tables. Leur algorithme de fusion pose problème car elle est réalisée comme s'il s'agissait simplement de la fusion de deux listes croissantes. Or en plus de cela, il doit y avoir des éléments à réordonner, sans quoi la table des suffixes obtenue n'est pas correcte, ce que les auteurs n'évoquent pas.

Le troisième algorithme utilise la table PLPC et procède, comme pour le premier algorithme, par reconstruction complète des deux tables, ce qui est trop coûteux en temps pour être intéressant.

Nous voyons, au travers des solutions récentes que des tentatives de mettre à jour des structures d'indexation existent afin de calculer la LZ-factorisation. Cependant ces solutions ont chacune leurs inconvénients. Soit l'espace utilisé est proportionnel à la longueur du texte, ce qui fait perdre l'intérêt de l'utilisation de fenêtres glissantes, soit les algorithmes procèdent à de coûteuses reconstructions de la table des suffixes, soit ils opèrent une fusion « partielle » de deux tables des suffixes mais n'indiquent pas comment on peut utiliser une telle table, qui n'est plus réellement ordonnée.

5.2.3 Indexation d'une fenêtre glissante avec la table des suffixes

Nous avons vu, en section 4.2.3.1 page 96, que le temps de mise à jour du FM-index dynamique est très largement inférieur à celui de la reconstruction de la structure statique. Cependant plus les textes sont courts et moins la différence est marquée. Par ailleurs nous avons également vu que le temps d'accès à une valeur de l'échantillonnage (section 4.2.5.2 page 105) est très long. D'autant plus si on le compare à un temps d'accès à une valeur quelconque d'un tableau. Sachant que nous travaillons sur des textes courts, les fenêtres glis-

santes sont généralement de taille inférieure à un million de lettres [6] il n'est pas indispensable d'échantillonner la table des suffixes. Il serait donc possible de stocker la table des suffixes en entier. Malgré tout, pour faciliter les insertions et suppressions, il est indispensable de la stocker dans un arbre binaire équilibré, ce qui donne encore un temps d'accès en $O(\log n)$ contre $O(1)$ pour un tableau.

La nécessité de stocker les valeurs de la table des suffixes dans un arbre n'est donc pas satisfaisante. Il faudrait être capable de stocker la table des suffixes dans un tableau sans que cela empêche de la mettre à jour. Pour cela nous allons procéder en deux temps pour la mise à jour de la table des suffixes. Nous commençons par enregistrer dans une liste doublement chaînée les mises à jour qu'il nous faudra réaliser sur la table des suffixes (insertion, suppression ou remplacement de valeurs). Ensuite cette liste est utilisée pour réaliser la mise à jour de la table des suffixes. En raison des insertions et suppressions, certaines valeurs doivent être décalées mais comme nous avons commencé par calculer toutes les modifications qui auront lieu, on va effectuer le nombre minimal de décalages dans la table des suffixes. Afin d'éviter de décrémenter inutilement de nombreuses valeurs, les positions dans la table des suffixes ne sont pas enregistrées de façon relative au début de la fenêtre mais en position absolue par rapport à la première position du texte.

Nous expliquons maintenant plus en détail comment à partir d'une table des suffixes TS_i indexant le facteur $T[i\,..\,i+f_g-1]$, nous pouvons passer à la table des suffixes $T_{i+\ell}$ où ℓ désigne la longueur du facteur trouvé dans g.

Nous commençons par supprimer les éléments qui ne doivent plus figurer dans la table des suffixes. Ainsi tous les nombres compris entre i et $i+\ell-1$ doivent être supprimés de la table des suffixes. Pour tout $k \in [i\,..\,i+\ell-1]$ il faut trouver $\mathsf{TS}_i[j] = k$, c'est-à-dire trouver la position de $T[k\,..]$ dans la table des suffixes.

Ensuite, il faut insérer les suffixes commençant aux position $i+f_g$ à $i+f_g+\ell-1$ dans T. Leur position dans la table des suffixes est trouvée en effectuant une recherche par dichotomie de ces suffixes dans la table.

Enfin il faut réordonner le suffixe commençant à la position $i+f_g-1$ et, éventuellement, les précédents. Pour savoir s'il faut les réordonner, on effectue également une recherche par dichotomie. Supposons que nous voulions réordonner le suffixe commençant en position k dans T, qui est actuellement en position j dans la table des suffixes. S'il doit être déplacé dans la table des suffixes, il le sera forcément à une position strictement supérieure à k [7]. De plus, si dans la table des suffixes, le suffixe considéré précédemment a été déplacé de p positions, le suffixe actuel sera déplacé au plus de p positions. Ainsi, pour ce suffixe, il suffit de faire la recherche dichotomique entre les positions k et $k+p$ dans la table des suffixes.

Tout au long de ces étapes, nous n'avons pas modifié la table des suffixes mais nous avons enregistré dans une liste les modifications à faire. Cette liste contient des couples de la forme (j,i) où j désigne la position de modification dans la table des suffixes et i désigne différents types de modifications selon qu'il soit positif, nul ou négatif. S'il est positif, il faut insérer i en position j

[6] La RFC 1951 (Deutsch, 1996) qui définit le format de compression *deflate*, utilisé par `gzip` et `zip` entre autres, détermine la longueur maximale des fenêtres à 32 768 pour g et 258 pour d.

[7] L'insertion d'éléments à la fin de la fenêtre ne peut qu'augmenter le rang lexicographique des suffixes déjà présents mais pas le diminuer.

dans la table des suffixes. S'il est nul, il faut supprimer l'entrée à la position j dans la table des suffixes. S'il est négatif, il faut substituer l'entrée en position j dans la table des suffixes par $-i$. Dans la liste, les couples sont triés en fonction de leur première composante j.

Maintenant résumons les différentes étapes et les éléments à insérer dans la liste, que nous appellerons Mod.

Suppression Pour chaque position de suppression j, on insère $(j, 0)$ au bon endroit (tel que les couples soient triés) dans Mod.

Insertion Pour l'insertion en position j dans la table des suffixes, du suffixe commençant en position k :
- s'il n'y a aucun couple dont la première composante est j dans Mod, on insère (j, k) ;
- si une entrée $(j, 0)$ existe déjà dans Mod, on la remplace par $(j, -k)$;
- s'il y a $\alpha \geq 1$ entrées de la forme $(j, i_0), (j, i_2), \ldots, (j, i_{\alpha-1})$, elles sont rangées dans l'ordre lexicographique des suffixes qu'elles représentent. Il faut donc insérer (j, k) à la bonne position, de façon à respecter l'ordre lexicographique des suffixes.

Réordonnement Le réordonnement d'un élément dans la table des suffixes correspond uniquement à une suppression suivi d'une insertion. Les éléments à insérer ou supprimer le sont suivant le même principe que pour les deux étapes précédentes.

Une fois toutes les étapes terminées, il reste à réaliser les modifications sur la table elle-même en utilisant les informations contenues dans Mod. La liste peut être divisée en différentes parties qui seront traitées indépendamment. On appelle :
- DécalageG une partie de la liste commençant par un couple dont la seconde composante est nulle. Une telle partie contient autant de couples donc la seconde composante est nulle que strictement positive.
- DécalageD une partie de la liste commençant par un couple dont la seconde composante est strictement positive. Une telle partie contient autant de couples donc la seconde composante est nulle que strictement positive.
- Statique une partie composée d'un seul couple dont la seconde composante est négative.

Nous donnons l'algorithme permettant de traiter une partie DécalageD (voir Algo. 5.2 page suivante). Le cas de la partie DécalageG est similaire quant à la partie Statique elle est triviale à traiter.

Nous donnons un exemple pour un décalage, en Ex. 5.9 page 125 et l'algorithme en Algo. 5.3 page 126.

5.2.4 Conclusion

L'avantage de l'algorithme que nous avons introduit est qu'il utilise très peu d'espace. En effet, il nécessite uniquement la table des suffixes sur la fenêtre g. La complexité en espace de l'algorithme est donc directement proportionnelle à f_g ce qui est très intéressant pour des applications où la mémoire est très limitée. En revanche, en ce qui concerne la complexité en temps, nous savons qu'à chaque étape la liste sera de taille $2s + 2r$, où s désigne la longueur du décalage et r le nombre de réordonnements réalisés. En nous appuyant

Algo. 5.2 — Traiter une partie de type DécalageD de la liste Mod pour mettre à jour la table des suffixes

```
1:  procédure TraitementDécalageDroit(Partie, TS)
2:      (fin_dest, ∗) ← prec(Partie)          ▷ Seconde valeur ignorée (car nulle)
3:                                  ▷ Partie est parcourue de droite à gauche
4:      (deb_src, valeur) ← prec(Partie)
5:      fin_src ← fin_dest − 1
6:      tant que deb_src est défini faire
7:          ℓ ← fin_src − deb_src + 1
8:          si ℓ > 0 alors
9:              TS[fin_dest − ℓ + 1 . . fin_dest] ← TS[deb_src . . fin_src]
10:             fin_src ← deb_src − 1
11:             fin_dest ← fin_dest − ℓ
12:         fin si
13:         si valeur = 0 alors
14:             fin_dest ← fin_dest + 1
15:         sinon si valeur > 0 alors
16:             TS[fin_dest] ← valeur
17:             fin_dest ← fin_dest − 1
18:         sinon
19:             TS[fin_dest + 1] ← −valeur
20:         fin si
21:         (deb_src, valeur) ← prec(Partie)
22:     fin tant que
23: fin procédure
```

sur les résultats de la section 4.1.2 page 82, nous savons que, en moyenne, le nombre de réordonnements est logarithmique par rapport à la taille du texte indexé, pour peu que le texte ne contienne pas trop de répétitions. Soit en moyenne une taille de $2s + O(\log f_g)$ pour la liste Mod. Néanmoins les modifications à réaliser à partir de la liste Mod peuvent être coûteuses (les recopies de valeurs de TS dans l'Algo. 5.2, ligne 9).

L'implantation est en cours de réalisation et des expériences devraient permettre de vérifier si, en plus de sa faible consommation mémoire, cet algorithme permet un calcul rapide de la LZ-factorisation d'un texte.

5.3 Applications à la bioinformatique

Une application naturelle des structures d'indexation est la bioinformatique. Depuis peu de temps, les biologistes ont accès, par exemple, à de grandes quantités de séquences génomiques de très grandes tailles. Afin de tirer parti de ces séquences il faut être capable d'y accéder et de pouvoir y rechercher l'information rapidement. Une seule séquence génomique peut faire 3 Gpb, comme celle de l'être humain, ou jusqu'à 130 Gpb pour *P. aethiopicus* [8]. Il peut

[8] `http://www.genomesize.com/statistics.php`

Ex. 5.9 — Calcul de la liste Mod et mise à jour de la table des suffixes, pour la prise en compte d'un décalage

Pour le texte $T = \overset{1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15}{\mathrm{CTACTAGGTCTCTA\$}}$, nous choisissons la fenêtre contenant $\overset{3\ 4\ 5\ 6\ 7\ 8}{\mathrm{ACTAGG}}$ et qui va être décalée pour contenir $\overset{4\ 5\ 6\ 7\ 8\ 9}{\mathrm{CTAGGT}}$.

Pour ACTAGG, nous avons $\mathsf{TS}_3 = \overset{1\ 2\ 3\ 4\ 5\ 6}{3\ 6\ 4\ 8\ 7\ 5}$.

1. Nous devons supprimer l'élément 3, en positions
   $\Rightarrow \mathsf{Mod} = (1, 0)$.
2. Nous devons trouver la position d'insertion de $\overset{9}{\mathrm{T}}$ parmi les suffixes existants dans la table. $\overset{9}{\mathrm{T}}$ est plus petit que $\overset{5\ 6\ 7\ 8}{\mathrm{TAGG}}$, 9 sera donc inséré en position 6 dans la table.
   $\Rightarrow \mathsf{Mod} = (1, 0) \leftrightarrow (6, 9)$
3. Nous devons vérifier si le suffixe précédent, en position 8 dans le texte et en position 4 dans la table, est bien positionné dans la table ou s'il doit être déplacé vers la droite. $\overset{8\ 9}{\mathrm{GT}} > \overset{7\ 8\ 9}{\mathrm{GGT}}$ mais $\overset{8\ 9}{\mathrm{GT}} < \overset{5\ 6\ 7\ 8\ 9}{\mathrm{TAGGT}}$. Le 8 va donc être déplacé en position 6 dans la table. Or nous avons le couple $(6, 9)$ dans Mod signifiant que $\overset{9}{\mathrm{T}}$ doit déjà être inséré en position 6. Nous devons continuer les comparaisons : $\overset{8\ 9}{\mathrm{GT}} < \overset{9}{\mathrm{T}}$. Dans la liste, nous mettrons donc $(6, 8)$ avant $(6, 9)$.
   $\Rightarrow \mathsf{Mod} = (1, 0) \leftrightarrow (4, 0) \leftrightarrow (6, 8) \leftrightarrow (6, 9)$.
4. Finalement, en supprimant les éléments en positions 1 et 4 et insérant 8 et 9 en position 6, on obtient la nouvelle table $\mathsf{TS}_4 = \overset{1\ 2\ 3\ 4\ 5\ 6}{6\ 4\ 7\ 8\ 9\ 5}$ pour le texte $\overset{4\ 5\ 6\ 7\ 8\ 9}{\mathrm{CTAGGT}}$.

être utile d'indexer plusieurs génomes en même temps afin d'en extraire les similitudes. Dans ce cas la taille totale des textes indexés atteint facilement des dizaines de gigaoctets et peut difficilement tenir en mémoire centrale. A fortiori s'il s'agit d'utiliser des structures d'indexation non compressées. En revanche, en utilisant des structures d'indexation compressées, on obtient un index pour le génome humain qui utilise moins de 2 Go, cela permet de s'en servir sur des machines courantes. Néanmoins ces structures ne résolvent pas tous les problèmes et indexer plusieurs génomes est complexe si on ne veut pas utiliser trop de mémoire. Dans la section 5.3.1 page 127 nous nous intéressons à l'indexation de génomes d'individus d'une même espèce. Dans la section 5.3.2 page 128, nous expliquons une méthode s'appuyant fortement sur les structures d'indexation pour rechercher rapidement des millions de courtes séquences au sein d'un génome.

Algo. 5.3 — Décalage d'une fenêtre de texte pour laquelle une table des suffixes est construite

Requiert i : position de la fenêtre, avant décalage, (de longueur N) dans le texte.
Requiert ℓ : longueur du décalage
Requiert w : contenu de la fenêtre après décalage
Requiert TS : la table des suffixes à modifier

```
1: procédure DecalageFenêtre(i, ℓ, w, TS)
2:     Mod ← ()
3:     pour j ← i à i + ℓ − 1 faire          ▷ Suppression des ℓ caractères de tête
4:         pos ← RechercheDicho(TS, j)              ▷ Recherche TS[pos] = j.
5:         InsèreMod(Mod, w, (pos, 0))   ▷ Insère (pos, 0) dans Mod, pour w.
6:     fin pour
7:     j ← i + ℓ + N
8:     tant que j > i + N faire              ▷ Insertion des ℓ caractères de queue
9:         pos ← RechercheFacteur(TS, w[j..])
10:        InsèreMod(Mod, w, (pos, j))
11:        j − −
12:    fin tant que
13:    pos ← RechercheDicho(TS, j)
14:    pos′ ← RechercheFacteur(SA, w[j..])
15:    tant que pos ≠ pos′ faire                                  ▷ Réordonnement
16:        InsèreMod(Mod, w, (pos, 0)); InsèreMod(Mod, w, (pos′, j))
17:        j − −
18:        pos ← RechercheDicho(TS, j)
19:        pos′ ← RechercheFacteur(SA, w[j..])
20:    fin tant que
21:    n+ ← 0; n− ← 0; n= ← 0; pMod ← 0
22:    tant que ! Mod.fin() faire
23:        (pos, j) ←Mod.courant()
24:        si j ≠ 0 alors
25:            type = (j > 0); prem ← pMod
26:            incValeurs(j, n+, n−, n=)      ▷ Incrémente n+ si j > 0, n− si
   j < 0 et n= sinon
27:            tant que n+ ≠ n− faire
28:                Mod.suivant() ; pMod ← pMod + 1
29:                (pos, j) ←Mod.courant()
30:                incValeurs(j, n+, n−, n=)
31:            fin tant que
32:            TraitementDécalage(type, Mod[prem .. pMod − 1], TS)
33:        sinon
34:            TS[pos] ← −j
35:        fin si
36:        Mod.suivant() ; pMod ← pMod + 1
37:    fin tant que
38: fin procédure
```

5.3.1 Indexation de génomes d'individus

Le seul résultat connu à ce jour présentant une solution spécialement mise au point pour l'indexation de multiples génomes d'individus est celui de Mäkinen *et al.* (2009). Leur solution consiste à améliorer la compression de certaines structures d'indexation compressées (la table compressée des suffixes et le FM-index) en ayant recours au run-length encoding (voir section 1.1.2.5 page 9). Malgré des résultats encourageants, la complexité finale en espace de leur structure d'indexation a une dépendance linéaire sur le nombre de textes indexés. Cela montre que les techniques employées pour compresser les structures d'indexation ne sont pas satisfaisantes pour tirer parti de très grandes répétitions. Les techniques utilisées pour la compression dans le cadre de l'indexation de textes permet d'atteindre l'entropie d'ordre k notée $H_k(T)$, ce qui donne une borne inférieure pour le stockage de l'index du texte T : $nH_k(T)$. Cette mesure de la compression est donc directement proportionnel à la longueur du texte. Celle-ci est valable à la fois pour le FM-index, la table compressée des suffixes ou le LZ-index.

En revanche, comme le soulignent Claude et Navarro (2009), les compressions basées sur la grammaire (telle le LZ-77) sont capables de tirer parti de très grandes répétitions. En fait le LZ-77 code les répétitions, quelle que soit leurs longueurs, toujours de la même façon, à l'aide d'un couple (p, ℓ), où p désigne la position de la répétition et ℓ sa longueur. Ainsi le codage d'un couple est totalement indépendant de la longueur des répétitions mais plus les répétitions seront longues, moins il y aura de couples à coder.

Par exemple prenons $T' = T\$T$, on a $H_k(T') \simeq H_k(T)$ [9]. Un compresseur tels que ceux utilisés pour les structures d'indexation actuelles n'arrivera pas à tirer parti de cette très grande répétition, d'où une entropie qui reste similaire. En conséquence, l'index de T' sera environ deux fois plus gros ($(2n+1)H_k(T)$) que celui de T ($nH_k(T)$). En revanche avec LZ-77, le deuxième T, dans T', pourra être codé avec un seul couple $(0, |T|)$. Ce couple peut être codé en utilisant $O(\log n)$ bits. La taille d'un hypothétique index fondé sur LZ-77 utiliserait de l'ordre de $\log n$ bits en plus pour indexer T' que pour indexer T. À l'opposé les méthodes actuelles ne permettent que d'avoir un index dont la taille est double pour T' par rapport à celle de T. Cet exemple permet d'apprécier la difficulté de tirer parti de très longues répétitions avec les structures d'indexation compressées actuelles. En revanche la compression par LZ-77 ne souffre pas de ce type de problème.

On comprend donc que l'arrivée d'une structure d'indexation utilisant la compression par LZ-77 (ou algorithme équivalent) est très attendue. Une des applications immédiate serait le stockage de génomes individuels, une autre application pourrait être les systèmes de gestion de versions permettant d'archiver et chercher parmi différentes versions d'un même document (comme les wikis, dont Wikipedia par exemple).

En raison de ce vide actuel parmi les solutions pour la gestion des génomes individuels, une solution pourrait être d'utiliser la structure d'indexation dynamique que nous avons mise au point. En connaissant les différences entre le génome de référence, celui indexé par notre structure, et le génome d'un individu, on serait capable d'indexer rapidement le génome de cet individu. Il

[9] On n'a pas égalité à cause du \$ ajouté. L'entropie empirique de T' sera donc légèrement supérieure à celle de T.

suffirait d'appliquer les insertions, suppressions ou substitutions nécessaires au génome de référence indexé afin qu'il corresponde au génome de l'individu qui nous intéresse. Cette solution semble une bonne alternative à la réindexation complète du génome à partir de rien. Sachant que les humains partagent 99,5 % de leur génome en commun (Levy *et al.*, 2007), il serait inutile de recommencer l'indexation à partir de 0 alors qu'une grande partie du travail a déjà été réalisée, avec l'indexation du génome de référence. Néanmoins, en regardant les expériences menées en section 4.2.4 page 101, on peut légitimement s'interroger sur la rapidité d'une telle méthode. Ce dernier point peut cependant être relativisé puisque la méthode de construction classique d'un FM-index statique passe par la construction de la table des suffixes. La table des suffixes occupant $4n$ octets, cela signifie qu'il faudrait au moins 15 Go de mémoire centrale pour construire un FM-index statique avec la méthode classique. Il existe d'autres méthodes de construction, par exemple en passant par une construction en blocs de la transformée de Burrows-Wheeler (Kärkkäinen, 2007) mais elles sont plus lentes.

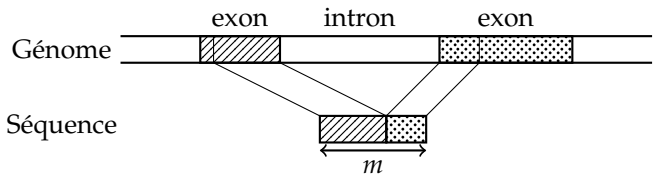
5.3.2 Recherche massive de courtes séquences dans une séquence génomique

5.3.2.1 Problématique

Nous considérons le problème dans lequel on souhaite connaître les positions des occurrences de millions de séquences de petite taille m (aux alentours de 50 pb) dans une séquence génomique. Les structures d'indexation semblent toutes indiquées pour ce problème : on a un très grand nombre d'éléments à chercher dans un texte que l'on peut aisément pré-traiter. Nous nous intéressons plus particulièrement aux structures d'indexation compressées pour que la structure d'indexation du génome puisse tenir dans la mémoire centrale de machines courantes.

En premier lieu, rappelons l'origine des séquences dans lesquelles nous recherchons massivement de courts motifs « répétés ». Dans le génome d'un organisme donné, l'ADN constitue le support de l'information génétique. De manière schématique, l'ADN porte des segments courts portant cette information, les gènes, interrompus par des segments intergéniques plus longs. Chez les eucaryotes supérieurs, les gènes sont eux même constitués de séquences codantes, les exons, « interrompues » par des séquences non codantes, les introns. L'expression d'un gène suit une procédure très précise que nous détaillerons succinctement ci-après. Dans le noyau cellulaire, les séquences d'ADN correspondants à des gènes sont transcrites en ARN. Ces ARN, immatures, subissent différentes modifications post-transcriptionnelles. Outre l'adjonction d'une coiffe et d'une queue polyadénylée à ses deux extrémités, l'ARN est débarrassé de ses séquences introniques par le processus d'épissage. Il en résulte alors un ARN mature, dit ARN messager, qui est par la suite traduit en protéine. Cette vision simple selon laquelle à un gène correspond une protéine s'est en fait révélée plus complexe. En effet, le génome humain compte de 20 000 à 25 000 gènes pour plus de 90 000 protéines, ce qui représente une augmentation de la capacité codante des gènes d'un facteur quatre. L'épissage alternatif constitue un mécanisme majeur par lequel un unique ARN immature peut générer plusieurs ARN messagers différents (rétention d'in-

Fig. 5.3 — Séquence non localisée en raison de l'épissage



tron, exons mutuellement exclusifs, promoteurs alternatifs,...). Chez les eucaryotes supérieurs, près de 95 % des gènes sont régulés par ce processus. Cette régulation est dépendante du type cellulaire et du stade de développement. Certaines pathologies génétiques humaines sont associées à une altération de ce processus. Ainsi une simple mutation ponctuelle, remplacement d'une base (A, T, C ou G) par une autre dans la séquence d'ADN peut avoir des conséquences drastiques sur la nature ou la quantité d'ARN mature produite. C'est notamment le cas dans les cancers. La méthodologie du séquençage à haut-débit offre la possibilité d'identifier les gènes dont l'expression est altérée dans ces pathologies. D'un point de vue pratique, l'ARN d'un type cellulaire donné, par exemple de la tumeur, est extrait. Celui-ci étant instable et rapidement dégradé, il est rétro-transcrit en ADNc (ADN complémentaire). Ce pool d'ADNc, miroir du transcriptome d'un type cellulaire donné à un moment donné pour un individu lambda et alors intégralement séquencé. Ainsi, la confrontation des résultats d'un cas pathologique face à une référence saine permettra l'identification des séquences impliquées dans la maladie (mutations, polymorphismes, déséquilibres d'expression,...). La probabilité qu'une séquence contienne au moins une erreur due au séquenceur varie entre 25 et 45 % selon les techniques (Keime *et al.*, 2007 ; Philippe *et al.*, 2009). En plus de ces erreurs dues au séquenceur, Keime *et al.* décrivent plusieurs autres raisons qui peuvent empêcher les séquences d'être localisées sur le génome :

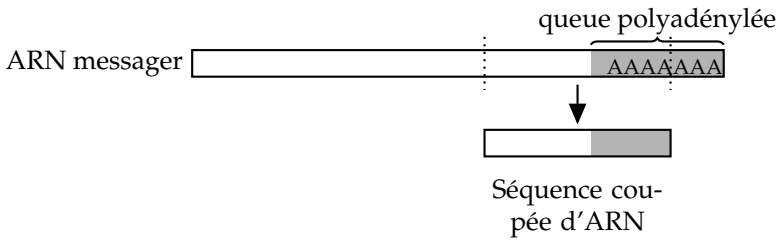
- séquences à cheval sur deux exons (épissage), voir FIG. 5.3 ;
- séquences contenant une partie de la queue polyadénylée, voir FIG. 5.4 page suivante ;
- séquences contenant des SNP (Single Nucleotide Polymorphism, mutation d'une base dans la séquence génomique), voir FIG. 5.5 page suivante ;

L'ensemble de ces phénomènes (biologiques ou techniques) empêchent de localiser sur le génome une partie des séquences obtenues. À titre d'exemple, la recherche exacte de 6 139 825 séquences filtrées (restent les séquences de bonne qualité, avec peu d'erreurs), de longueur 27, sur le génome humain permet d'en localiser à peine 25 %.

5.3.2.2 Solutions

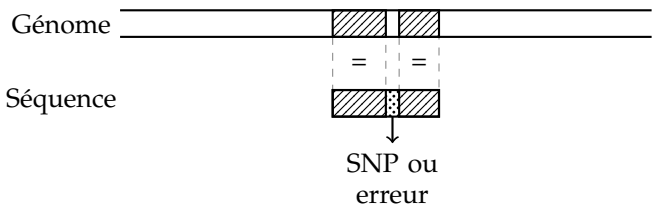
Trapnell et Salzberg (2009) décrivent les nombreuses solutions existant pour tenter de trouver les occurrences d'une séquence contenant des erreurs.

Fig. 5.4 — Séquence non localisée car la séquence d'ARN d'origine a été coupée dans la queue polyadénylée



La queue polyadénylée est spécifique à l'ARN messager et n'existe pas dans l'ADN. Quand on essaiera de localiser cette séquence dans le génome (ADN) on ne pourra pas la retrouver.

Fig. 5.5 — Séquence non localisée car elle contient un SNP (ou une erreur)



Il y a une seule différence entre le génome et la séquence obtenue par la machine, cela peut être dû à une erreur du séquenceur ou à un SNP.

On remarquera, que l'indexation compressée, à l'aide de la transformée de Burrows-Wheeler est assez largement utilisée dans ce cadre là (Lam *et al.*, 2008 ; Langmead *et al.*, 2009 ; Li et Durbin, 2009 ; Li *et al.*, 2009). Ces solutions utilisent toutes la recherche approchée. Contrairement à la recherche exacte, où on cherche les facteurs de T égaux à P, la recherche approchée consiste à trouver des facteurs de T égaux à P, à quelques erreurs près (insertions, suppression ou substitutions). Les algorithmes dédiés à la recherche approchée sont, généralement, génériques.

À l'opposé, nous avons pensé qu'il fallait trouver un algorithme propre aux données que nous avions à traiter. Avec Philippe *et al.* (2009), nous avons utilisé leur résultat pour mettre au point un algorithme efficace. L'idée consiste à rechercher dans le génome, non pas la totalité de la séquence, mais tous les facteurs, d'une longueur ℓ. Quand ℓ est suffisamment grand (20 pour le génome humain), la probabilité de trouver par hasard un facteur de la séquence sur le génome est très faible. L'utilisation de cette technique permet d'évi-

ter l'accumulation d'erreurs biologiques ou techniques (plus la séquence est longue, plus on risque d'en avoir). Sur une séquence suffisamment longue (disons de longueur 50), cela permet d'augmenter fortement les chances qu'un facteur de longueur ℓ soit exempt d'erreur. Ainsi, dans la FIG. 5.3 page 129, si un facteur est contenu exclusivement dans la zone hachurée ou dans la zone en pointillés, celui-ci pourra être localisé sur le génome. Sans cela c'est impossible, les méthodes de recherche approchée ne tolèrent qu'un petit nombre d'erreurs, insuffisant pour localiser ce type de séquences. Notons que cette remarque est également valable pour les séquences contenant une partie de la queue polyadénylée. Cette idée de découpage en facteurs permet d'avoir une recherche de séquences plus pertinente mais également plus simple, car cette méthode ne met en œuvre que de la recherche exacte.

5.3.2.3 Détection des erreurs techniques et biologiques

Par ailleurs ce découpage permet également d'observer d'autres phénomènes. Soit S l'ensemble des séquences uniques obtenues par le séquenceur, $|S| = s$ et soit T le génome de longueur n. Considérons une seule séquence S_i de longueur m et tous ses facteurs de longueur ℓ, $F^{\ell}_{i,0}, \ldots, F^{\ell}_{i,m-\ell+1}$, avec $F^{\ell}_{i,k} = S_i[k \,..\, k+\ell-1]$ et $0 \leq k \leq m-\ell+1$. On note $|T|_{F^{\ell}_{i,k}}$, le nombre d'occurrences de $F^{\ell}_{i,k}$ dans le génome T et $|F^{\ell}|_{F^{\ell}_{i,k}}$ le nombre d'occurrences de $F^{\ell}_{i,k}$ parmi tous les facteurs de longueur ℓ de toutes les séquences S_i, pour $0 \leq i < s$. Nous étudions pour chacun de ces facteurs leur nombre d'occurrences dans le génome, ainsi que parmi l'ensemble des facteurs. Pour ce faire l'ensemble des facteurs est également indexé. L'utilisation d'une structure d'indexation compressée n'est, pour l'instant, pas indispensable au vu du nombre et de la taille des facteurs en jeu. La variation de ces deux nombres d'occurrences tout au long de la séquence S donne diverses informations :

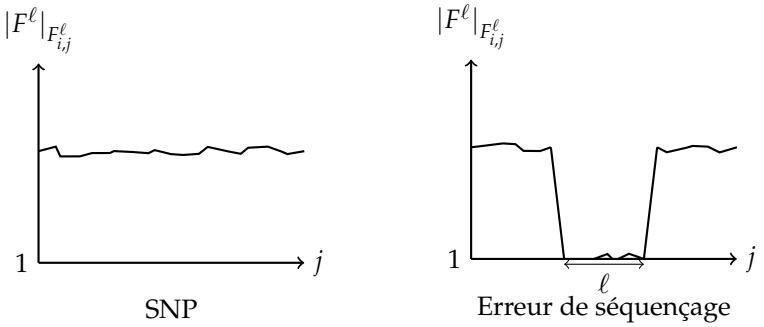
- Si, pendant environ ℓ positions aucun facteur n'est localisé sur le génome alors que les facteurs précédents et suivants le sont, on a un SNP ou une erreur du séquenceur. Il est possible de trancher entre les deux en utilisant le nombre d'occurrences des facteurs concernés parmi l'ensemble des signatures (voir FIG. 5.6).
- Supposons que les facteurs $F^{\ell}_{i,j}$ à $F^{\ell}_{i,k}$ sont localisés côte à côte et de façon unique sur le génome et que les facteurs commençant à partir de la position $k+1$ soient également localisés de façon unique mais à une centaine de bases des précédents facteurs, dans ce cas on sait qu'il s'agit d'un phénomène dû à l'épissage.

5.3.3 Conclusion

Notons que les deux notions abordées dans les précédentes sections ne sont pas exclusives, au contraire. Rechercher des millions de courtes séquences parmi les génomes de plusieurs individus peut se révéler très intéressant au niveau biologique afin de mieux comprendre l'influence de certaines mutations sur le phénotype (ensemble des caractéristiques observables représentant un être vivant). Le séquençage de génomes d'individus est en train de se développer et certaines sociétés proposent déjà de séquencer le génome de

Fig. 5.6 — SNP ou erreur de séquençage ?

On se demande si le tag S_i possède un SNP ou une erreur de séquençage, pour cela on s'intéresse au nombre d'occurrences de ses facteurs parmi l'ensemble F^ℓ.



SNP — Erreur de séquençage

Pour la séquence avec SNP, le nombre d'occurrences est relativement stable pour l'ensemble des facteurs. En revanche, ce qui caractérise une erreur de séquençage c'est qu'environ ℓ facteurs sont quasiment uniques, les autres étant plus fréquents.

n'importe quelle personne intéressée[10] . Mais d'un autre côté, les techniques informatiques permettant de stocker des génomes d'individus, et plus généralement de longues séquences peu différentes, ne sont pas encore au point et n'offrent pas des résultats suffisamment satisfaisant pour permettre le stockage de nombreux génomes individuels sur des machines « raisonnables ».

Étant donnée l'importance du sujet et le peu de résultats sur celui-ci, on peut raisonnablement se demander si une solution efficace (comme celle qui pourrait être issue du LZ-77) est envisageable. Si dans l'avenir c'était le cas, il resterait à réfléchir à la façon de mettre à jour de telles structures d'indexation. Là non plus, la solution n'apparaît pas évidente : le fait de stocker de façon efficace des textes similaires signifie que la technique de stockage induit une dépendance forte entre tous ces textes. La modification d'un seul de ceux-ci risque donc d'avoir des répercussions bien plus fortes, sur la structure d'indexation, que celles observées avec notre algorithme pour le FM-index.

[10]Sous réserve de vouloir dépenser environ 35 000 euros pour ça : `http://www.everygenome.com/genomics101/faqs_dna_sequencing.ilmn`. IBM espère cependant arriver à un coût de 100 à 1 000 dollars : `http://www-03.ibm.com/press/us/en/pressrelease/28558.wss`.

Chapitre 6

Conclusion et perspectives

Le travail présenté dans cette thèse a consisté, dans un premier temps, à étudier les méthodes employées au cœur des différentes structures d'indexation compressées. Ensuite, nous nous sommes rapidement intéressés à la mise à jour de telles structures mais le nombre d'éléments à réordonner, pour la table des suffixes, ou de couples à modifier, pour le LZ-index, nous semblait rédhibitoire. Une présentation préliminaire des travaux de Gallé *et al.* (2008) nous a montré que le nombre de modifications dans la table des suffixes est en pratique bien inférieur à la longueur du texte. La fonction LF est l'outil fondamental pour trouver la position d'une permutation circulaire à partir de la position de la permutation circulaire suivante. Nous avons rapidement pensé que cette fonction pouvait être d'une grande utilité pour permettre de trouver la *nouvelle* position d'une permutation, après modification. L'algorithme qui est né de cette idée (Salson *et al.*, 2008, 2009b) nous permet de modifier la transformée de Burrows-Wheeler pour prendre en compte des insertions, suppressions ou substitutions dans le texte d'origine. L'objectif initial était d'avoir une structure d'indexation dynamique, nous avons donc cherché à étendre notre résultat au FM-index. La mise au point de l'algorithme de mise à jour de l'échantillonnage de la table des suffixes et du stockage de celui-ci ont complété notre premier algorithme. L'ensemble de ces résultats nous a permis alors d'avoir une structure d'indexation compressée autorisant n'importe quel type de modification (Salson *et al.*, 2009a).

Nous avons implanté notre algorithme de mise à jour ainsi que la structure permettant de gérer un échantillonnage dynamique. Grâce à cela, nous avons pu mener des tests sur des textes habituellement utilisés pour l'indexation (séquences génomiques et textes en langage naturel). Ces expériences ont été complétées par des résultats théoriques sur le nombre moyen d'éléments à mettre à jour dans la table des suffixes ou la transformée de Burrows-Wheeler lorsque le texte d'origine est modifié. D'un point de vue théorique, nous avons montré que notre algorithme de mise à jour est polylogarithmique, pour des textes peu répétés. D'un point de vue pratique, les expériences que nous avons menées nous permettent de conclure que notre algorithme de mise à jour est plus rapide, de plusieurs ordres de grandeur, que le plus rapide algorithme de reconstruction, y compris sur des textes très répétés.

Par ailleurs, avec Maxime Crochemore et William Smyth, nous nous sommes intéressés à l'indexation de séquences numériques afin de déterminer le minimum dans un intervalle quelconque. Nous avons présenté un algorithme novateur, et l'avons implanté, dans sa version statique. Les expériences que nous avons menées montrent que cette méthode est compétitive en espace et les résultats pourraient encore être améliorés en utilisant des implantations plus économes des champs de bits. La méthode que nous avons introduite permet également la mise à jour de la structure d'indexation, ce qui autorise les modifications dans la séquence d'origine.

De plus, avec William Smyth, nous nous sommes intéressés à la mise à jour des structures d'indexation dans le but de l'utiliser pour la LZ-factorisation. Notre version dynamique du FM-index n'était pas utilisable en l'état car trop générale pour ce problème particulier, et surtout plus appropriée pour les grands textes. Tout en gardant le principe de base (insertions, suppressions et réordonnements) nous avons mis au point une nouvelle méthode permettant la mise à jour de la table des suffixes à l'aide d'une liste enregistrant les modifications qui doivent être répercutées sur la table. Cette méthode est d'un grand intérêt car elle permet une très faible consommation mémoire, proportionnelle à la taille de la fenêtre utilisée pour la factorisation.

Enfin nous avons réfléchi à l'utilisation qui pouvait être faite des structures d'indexation en bioinformatique. Nous avons pris le parti d'utiliser la signification qu'ont ces données biologiques et de s'en servir pour délivrer une information plus pertinente. Les travaux que nous menons dans ce domaine, avec une équipe du LIRMM [1] et de l'IGH [2] à Montpellier, sont toujours en cours mais les premiers résultats montrent une augmentation significative, par rapport à d'autres méthodes récentes, du pourcentage de séquences localisées. D'un point de vue théorique nous savons que nous pouvons identifier les SNP, splices et erreurs de séquences. Il nous reste à implanter cette méthode et à estimer la qualité des résultats en utilisant des données connues.

Notre technique utilise des structures d'indexation à la fois pour le génome sur lequel on veut rechercher les séquences mais aussi pour les courtes séquences. Pour la première nous utilisons une structure compressée mais pas pour la seconde, afin de permettre un temps de recherche relativement rapide. Cela posera un problème dans un futur relativement proche car les courtes séquences obtenues grâce aux séquenceurs vont être de plus en plus longues et de plus en plus nombreuses. Il y aura donc de plus en plus de données à indexer sans que le temps de calcul ne soit dégradé. Il faut également que les structures continuent à tenir en mémoire centrale. Les problèmes d'espace mémoire pourraient être résolus en utilisant une structure d'indexation compressée mais dans ce cas le temps de recherche serait plus long, comme l'ont montré les expériences en section 1.4 page 38. Finalement il s'agit d'un compromis espace-temps et dans ces conditions le meilleur choix dépend des besoins et des capacités des machines à notre disposition.

[1] Laboratoire d'Informatique Robotique Microélectronique de Montpellier
[2] Institut de Génétique Humaine

Malgré les nombreuses avancées depuis l'an 2000, le domaine des structures d'indexation laisse encore de nombreuses questions sans réponse.

De nombreuses structures d'indexation compressées reposent sur un échantillonnage de la table des suffixes mais nous savons que c'est inefficace lorsqu'on souhaite connaître les positions des occurrences d'un motif. Dans ce cadre là, disposer d'une table compressée des suffixes permettant de récupérer n'importe quelle valeur de la table des suffixes en temps constant rendrait caduque l'échantillonnage de la table des suffixes. De plus cela permettrait de localiser toutes les occurrences d'un motif en temps optimal $O(m + |T|_P)$.

En termes de compression, les structures d'indexation ne savent pas tirer parti des grandes répétitions et donc n'arrivent pas à compresser efficacement de multiples génomes d'individus, par exemple. Il serait intéressant d'avoir une structure d'indexation répondant à ce problème qui, contrairement à la méthode de Mäkinen *et al.* (2009), n'ait pas une complexité linéaire dans le nombre de textes qu'elle indexe. Une telle structure n'aurait pas un intérêt uniquement pour les séquences génomiques mais aussi pour tout ce qui est systèmes de gestion de versions pour lesquels il faut conserver toutes les modifications qui ont été réalisées au sein d'un document.

Le domaine de la mise à jour des structures d'indexation est encore très récent et de nombreuses questions se posent. Avoir une structure d'indexation compressée pour laquelle une seule mise à jour du texte n'entraîne jamais un nombre linéaire de modifications dans la structure permettrait de réduire la complexité de mise à jour dans le pire des cas. Nous savons que l'arbre contracté des suffixes correspond à cette définition mais il n'est pas compressé. Une piste intéressante pourrait être de chercher à compresser cette structure. Néanmoins, on peut se demander si une fois compressé l'arbre contracté ne risque pas de perdre cette intéressante propriété.

Il existe des solutions pour stocker et utiliser efficacement les structures d'indexation, compressées ou non, sur disque (Arroyuelo et Navarro, 2007a ; González et Navarro, 2007 ; Wong *et al.*, 2007). En revanche, aucune solution n'est connue pour les structures d'indexation dynamiques. Une telle solution serait potentiellement révolutionnaire. Elle permettrait d'utiliser ces structures d'indexation, de préférence compressées, comme des systèmes de fichiers, dans lesquels on pourrait rechercher très rapidement et dont l'espace utilisé serait inférieur à un système classique. Mais le stockage et l'utilisation sur disque de structures dynamique pose un gros problème. Pour les structures statiques, les données sont organisées sur le disque afin de minimiser les accès lors de recherches de motifs. Les mises à jour vont avoir tendance à casser cette organisation et donc à éparpiller des données qui étaient auparavant contiguës, rendant beaucoup moins efficace la recherche de motifs.

Enfin, concernant notre méthode de mise à jour, il faudrait s'intéresser à un algorithme plus efficace pour la mise à jour des valeurs PLPC. Par ailleurs pour améliorer le stockage de la table PLPC, on pourrait utiliser la méthode décrite par Sadakane (2007) et utilisant $2n + o(n)$ bits. En ce qui concerne l'espace mémoire, nous savons aussi que notre stockage de l'échantillonnage de la table des suffixes est trop gourmand. En particulier la permutation π_{TIS} stockée à l'aide de deux arbres binaires. Nous pourrions utiliser une autre méthode pour le stockage des arbres, pour diminuer la consommation mémoire. Malgré cela la consommation sera encore importante car nous avons deux arbres plus l'échantillonnage lui-même à stocker. Plutôt que de continuer dans

cette voie, il pourrait être utile de chercher une autre méthode permettant de gérer plus efficacement un échantillonnage dynamique.

La méthode statique de recherche du minimum dans un intervalle ne permet pas de répondre aux requêtes en temps constant, contrairement aux autres méthodes. Une piste à explorer serait d'adapter notre approche, basée sur les minimums locaux, pour répondre aux requêtes de minimum en temps constant plutôt qu'en temps logarithmique. Pour l'instant le verrou réside dans la façon dont nous calculons le minimum : pour chaque niveau nous avons besoin de connaître les valeurs au bord de l'intervalle que nous considérons. Cela est rendu nécessaire par le fait que les minimums locaux nous donnent des informations sur ce qui se passe à l'intérieur de l'intervalle mais pas aux frontières.

Dans le domaine de l'indexation de textes de nombreuses structures sont des auto-index, c'est-à-dire que le texte peut être récupéré à partir de la structure d'indexation. Ce sont d'ailleurs ces structures qui sont les plus efficaces en termes d'espace. En prenant exemple sur ce qui se passe pour l'indexation de textes, il pourrait être intéressant de chercher à indexer et en même temps stocker efficacement la séquence numérique.

Par ailleurs, dans ce document, nous avons fait état de certains travaux toujours en cours. Nous allons implanter la méthode permettant de détecter les SNP, splices et erreurs de séquence. À partir des résultats obtenus, nous pourrons estimer la qualité de détection. Nous prévoyons aussi d'implanter et mener des expériences concernant notre algorithme de LZ-factorisation et de le comparer à des méthodes similaires. Enfin, parmi toutes les pistes que nous évoquées dans ce chapitre, nous envisageons de nous intéresser plus particulièrement au stockage de séquences très similaires et de tirer parti des propriétés de certaines structures (par exemple l'oracle des facteurs (Allauzen *et al.*, 1999)) pour stocker efficacement de telles séquences.

Bibliographie

M. I. ABOUELHODA, S. KURTZ et E. OHLEBUSCH : Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

G. ADELSON-VELSKII et E. M. LANDIS : An algorithm for the organization of information. *In Proceedings of the USSR Academy of Sciences*, volume 146, pages 263–266, 1962.

C. ALLAUZEN, M. CROCHEMORE et M. RAFFINOT : Factor oracle : A new structure for pattern matching. *In SOFSEM'99 : Theory and Practice of Informatics*, pages 295–310, 1999.

Z. ARNAVUT et S. S. MAGLIVERAS : Block sorting and compression. *In Proc. of Data Compression Conference (DCC)*, pages 181–190, 1997.

D. ARROYUELO et G. NAVARRO : A lempel-ziv text index on secondary storage. *In Proc. of Combinatorial Pattern Matching (CPM)*, pages 83–94, 2007a.

D. ARROYUELO et G. NAVARRO : Smaller and faster Lempel-Ziv indices. *In Proc. of International Workshop On Combinatorial Algorithms (IWOCA)*, pages 11–20. King's College Publications, UK, 2007b.

D. ARROYUELO, G. NAVARRO et K. SADAKANE : Reducing the space requirement of LZ-index. *In Proc. of Combinatorial Pattern Matching (CPM)*, volume 4009 de *Lecture Notes in Computer Science*, pages 319–330, 2006.

B. BALKENHOL, S. KURTZ et Y. M. SHTARKOV : Modifications of the Burrows and Wheeler data compression algorithm. *In Proc. of Data Compression Conference (DCC)*, pages 188–197, 1999.

R. BAYER : Symmetric binary B-trees : Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

T. BELL, J. CLEARY et I. WITTEN : *Text compression*. Prentice Hall, 1990.

M. A. BENDER et M. M. FARACH-COLTON : The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.

R. S. BOYER et J. S. MOORE : A fast string searching algorithm. *Commun. ACM*, 20:762–772, 1977.

M. BURROWS et D. J. WHEELER : A block-sorting lossless data compression algorithm. Rapport technique 124, DEC, Palo Alto, California, 1994.

H. L. CHAN, W. K. HON et T. W. LAM : Compressed index for a dynamic collection of texts. *In Proc. of Combinatorial Pattern Matching (CPM)*, pages 445–456, 2004.

B. CHAPIN et S. R. TATE : Higher compression from the Burrows-Wheeler transform by modified sorting. *In Proc. of Data Compression Conference (DCC)*, 1998.

G. CHEN, S. J. PUGLISI et W. F. SMYTH : Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 19:37–52, 2008.

F. CLAUDE et G. NAVARRO : Practical rank/select queries over arbitrary sequences. *In Proc. of String Processing and Information Retrieval (SPIRE)*, pages 176–187, 2008.

F. CLAUDE et G. NAVARRO : Self-indexed text compression using straight-line programs. *In Proc. of Mathematical Foundations of Computer Science (MFCS)*, pages 235–246, 2009.

M. CROCHEMORE, L. ILIE et W. F. SMYTH : A simple algorithm for computing the Lempel-Ziv factorization. *In Proc. of Data Compression Conference (DCC)*, 2008.

S. DEOROWICZ : Second step algorithms in the Burrows-Wheeler compression algorithm. *Softw. Pract. Exper.*, 32(2):99–111, 2001.

P. DEUTSCH : DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), mai 1996.

A. EHRENFEUCHT, R. MCCONNELL et S.-W. WOO : Contracted suffix trees : a simple and dynamic text indexing data structure. *In Proc. of Combinatorial Pattern Matching (CPM)*, pages 41–53, 2009.

P. ELIAS : Universal codeword sets and representation of the integers. *IEEE Trans. on information theory*, 21:194–203, 1975.

J. FAYOLLE et M.D. WARD : Analysis of the average depth in a suffix tree under a Markov model. *In International Conference on the Analysis of Algorithms*, pages 95–104, 2005.

P. FENWICK, M. TITCHENER et M. LORENZ : Burrows Wheeler - alternatives to move to front. *In Proc. of Data Compression Conference (DCC)*, 2003.

P. FERRAGINA, R. GONZÁLEZ, G. NAVARRO et R. VENTURINI : Compressed text indexes : From theory to practice. *Journal on Experimental Algorithmics*, 13:article 12, 2009.

P. FERRAGINA et R. GROSSI : Optimal on-line search and sublinear time update in string matching. *In Proc. of Foundations of Computer Science (FOCS)*, pages 604–612, 1995.

P. FERRAGINA et G. MANZINI : Opportunistic data structures with applications. *In Proc. of Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

P. FERRAGINA et G. MANZINI : Compression boosting in optimal linear time using the Burrows-Wheeler transform. *In Proc. of Symposium on Discrete Algorithms (SODA)*, pages 655–663, 2004.

P. FERRAGINA et G. MANZINI : Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

P. FERRAGINA, G. MANZINI, V. MÄKINEN et G. NAVARRO : Compressed representation of sequences and full-text indexes. *ACM Trans. Alg.*, 3: article 20, 2007.

A. J. FERREIRA, A. L. OLIVEIRA et M. A. T. FIGUEIREDO : On the use of suffix arrays for memory-efficient Lempel-Ziv data compression. *In Proc. of Data Compression Conference (DCC)*, page 444, 2009.

J. FISCHER : Optimal succinctness for range minimum queries, 2009.

J. FISCHER et V. HEUN : Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. *In Proc. of Combinatorial Pattern Matching (CPM)*, pages 36–48, 2006.

J. FISCHER et V. HEUN : A new succinct representation of RMQ-information and improvements in the enhanced suffix array. *In* LNCS, éditeur : *Proc. of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE)*, volume 4614, pages 459–470, 2007.

J. FISCHER et V. HEUN : Range median of minima queries, super-cartesian trees, and text indexing. *In Proc. of International Workshop On Combinatorial Algorithms (IWOCA)*, 2008.

J. FISCHER, V. HEUN et H. M. STÜHLER : Practical entropy-bounded schemes for $o(1)$-range minimum queries. *In Proc. of Data Compression Conference (DCC)*, 2008.

G. FRANCESCHINI et R. GROSSI : No sorting ? Better searching ! *ACM Trans. Alg.*, 4(1):1–13, 2008.

H. N. GABOW, J. L. BENTLEY et R. E. TARJAN : Scaling and related techniques for geometry problems. *In Proc. of the ACM Symposium on Theory Of Computing (STOC)*, pages 135–143, 1984.

M. GALLÉ, P. PETERLONGO et F. COSTE : In-place update of suffix array while recoding words. *In* Jan HOLUB et Jan VALEURZVALEURDÁREK, éditeurs : *Proc. of Prague Stringology Conference (PSC)*, pages 54–67, Czech Technical University in Prague, Czech Republic, 2008. ISBN 978-80-01-04145-1.

W. GERLACH : Dynamic FM-Index for a collection of texts with application to space-efficient construction of the compressed suffix array. Mémoire de Master, Universität Bielefeld, Germany, 2007.

A. GOLYNSKI, J. I. MUNRO et S. S. RAO : Rank/select operations on large alphabets : a tool for text indexing. *In Proc. of Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

G. H. GONNET, R. A. BAEZA-YATES et T. SNIDER : New indices for text : Pat trees and Pat arrays. *Information Retrieval : Data Structures & Algorithms*, pages 66–82, 1992.

R. GONZÁLEZ, Sz. GRABOWSKI, V. MÄKINEN et G. NAVARRO : Practical implementation of rank and select queries. *In Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

R. GONZÁLEZ et G. NAVARRO : A compressed text index on secondary memory. *In Proc. of International Workshop On Combinatorial Algorithms (IWOCA)*, pages 80–91. King's College London Publications, UK, 2007.

R. GONZÁLEZ et G. NAVARRO : Improved dynamic rank-select entropy-bound structures. *In Proc. of the Latin American Theoretical Informatics (LATIN)*, volume 4957 de *Lecture Notes in Computer Science*, pages 374–386, 2008.

R. GROSSI, A. GUPTA et J. S. VITTER : High-order entropy-compressed text indexes. *In Proc. of Symposium on Discrete Algorithms (SODA)*, 2003.

R. GROSSI et J. S. VITTER : Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *In Proc. of the ACM Symposium on Theory Of Computing (STOC)*, 2000.

R. GROSSI et J. S. VITTER : Compressed suffix arrays and suffix trees, with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2): 378–407, 2005.

L. J. GUIBAS et R. SEDGEWICK : A dichromatic framework for balanced trees. *In Proc. of Foundations of Computer Science (FOCS)*, pages 8–21, 1978.

A. GUPTA, W.-K. HON, R. SHAH et J. S. VITTER. : Compressed data structures : Dictionaries and data-aware measures. *Theor. Comput. Sci.*, 387(3):313–331, 2007a.

A. GUPTA, W. K. HON, R. SHAH et J. S. VITTER : A framework for dynamizing succinct data structures. *In Proc. of International Colloquium on Automata, Languages, and Programming (ICALP)*, 2007b.

D. GUSFIELD : *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology*. Cambridge University Press, 1997.

B. HAUBOLD et T. WIEHE : How repetitive are genomes ? *BMC Bioinformatics*, 7:541+, 2006.

W.-K. HON, T.-W. LAM, K. SADAKANE, W.-K. SUNG et S.-M. YIU : Compressed index for dynamic text. *In Proc. of Data Compression Conference (DCC)*, pages 102–111, 2004.

D. A. HUFFMAN : A method for the construction of minimum-redundancy codes. *In Proc. of the Institute Radio Engineers*, 1952.

J. KÄRKKÄINEN : Fast BWT in small space by blockwise suffix sorting. *Theor. Comput. Sci.*, 387(3):249–257, 2007.

J. KÄRKKÄINEN et P. SANDERS : Simple linear work suffix array construction. *In Proc. of International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 943–955, 2003.

J. KÄRKKÄINEN et E. UKKONEN : Lempel-Ziv parsing and sublinear-size index structures for string matching. *In Proc. of SouthAmerican Workshop on String Processing (WSP)*, pages 141–155. Carleton University Press, 1996.

T. KASAI, G. LEE, H. ARIMURA, S. ARIKAWA et K. PARK : Linear-time longest-common-prefix computation in suffix arrays and its applications. *In Proc. of Combinatorial Pattern Matching (CPM)*, volume 2089 de *Lecture Notes in Computer Science*, pages 181–192, 2001.

C. KEIME, M. SÉMON, D. MOUCHIROUD, L. DURET et O. GANDRILLON : Unexpected observations after mapping LongSAGE tags to the human genome. *BMC Bioinformatics*, 8:154, 2007.

D. K. KIM, J. S. SIM, H. PARK et K. PARK : Linear-time construction of suffix arrays. *In Proc. of Combinatorial Pattern Matching (CPM)*, pages 186–199, 2003.

D. E. KNUTH, J. H. MORRIS et V. R. PRATT : Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

P. KO et S. ALURU : Space efficient linear time construction of suffix arrays. *In Proc. of Combinatorial Pattern Matching (CPM)*, pages 200–210, 2003.

S. KURTZ : Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.

T. W. LAM, W. K. SUNG, S. L. TAM, C. K. WONG et S. M. YIU : Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, 2008.

B. LANGMEAD, C. TRAPNELL, M. POP et S. L. SALZBERG : Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

S. LEVY, G. SUTTON, P. C. NG, L. FEUK, A. L. HALPERN, B. P. WALENZ, N. AXELROD, J. HUANG, E. F. KIRKNESS, G. DENISOV, Y. LIN, J. R. MACDONALD, A. Wing Chun PANG, M. SHAGO, T. B. STOCKWELL, A. TSIAMOURI, V. BAFNA, V. BANSAL, S. A. KRAVITZ, D. A. BUSAM, K. Y. BEESON, T. C. MCINTOSH, K. A. REMINGTON, J. F. ABRIL, J. GILL, J. BORMAN, Y.-H. ROGERS, M. E. FRAZIER, S. W. SCHERER, R. L. STRAUSBERG1 et J. C. VENTER : The diploid genome sequence of an individual human. *PLoS Biology*, 5(10):2113–2144, 2007.

H. LI et R. DURBIN : Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 2009.

R. LI, C. YU, Y. LI, T.-W. LAM, S.-M. YIU, K. KRISTIANSEN et J. WANG : SOAP2 : an improved ultrafast tool for short read alignment. *Bioinformatics*, 2009.

V. MÄKINEN : Compact suffix array. *In Proc. of Combinatorial Pattern Matching (CPM)*, 2000.

V. MÄKINEN : Compact suffix array – a space-efficient full-text index. *Fundamenta Informaticae, Special Issue - Computing Patterns in Strings*, 56(1-2):191–210, 2003.

V. MÄKINEN et G. NAVARRO : Compressed compact suffix arrays. *In Proc. of Combinatorial Pattern Matching (CPM)*, volume 3109, pages 420–433, 2004.

V. MÄKINEN et G. NAVARRO : Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing*, 12(1):40–66, 2005.

V. MÄKINEN et G. NAVARRO : Rank and select revisited and extended. *Theor. Comput. Sci.*, 387(3):332–347, 2007.

V. MÄKINEN et G. NAVARRO : Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Alg.*, 4(3):article 32, 2008. 38 pages.

V. MÄKINEN, G. NAVARRO, J. SIRÉN et N. VÄLIMÄKI : Storage and retrieval of highly repetitive sequence collections. *In Journal of Computational Biology*, 2009. To appear.

U. MANBER et G. MYERS : Suffix arrays : a new method for on-line string searches. *In Proc. of Symposium on Discrete Algorithms (SODA)*, pages 319–327, 1990.

M. A. MANISCALCO et S. J. PUGLISI : Faster lightweight suffix array construction. *In Proc. of International Workshop On Combinatorial Algorithms (IWOCA)*, pages 16–29, 2006.

G. MANZINI : An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

G. MANZINI : Two space saving tricks for linear time LCP array computation. *In Proc. 9th Scandinavian Workshop on Algorithm Theory*, volume 3111, pages 372–383, 2004.

G. MANZINI et P. FERRAGINA : Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

E. M. MCCREIGHT : A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

K. MONOSTORI, A. ZASLAVSKY et I. VAJK : Suffix vector : a space efficient suffix tree representation. *In Proceedings of the International Symposium on Algorithms and Computation*, 2001.

J.H. MORRIS et R. PRATT : A linear pattern-matching algorithm. Rapport technique 40, University of California, Berkeley, 1970.

I. MUNRO : Tables. *In Proc. of Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 1180, pages 37–42, 1996.

G. NAVARRO : Indexing text using the Ziv-Lempel trie. *In Proc. of String Processing and Information Retrieval (SPIRE)*, volume 2476, pages 325–336, 2002.

G. NAVARRO et V. MÄKINEN : Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):article 2, 2007.

D. OKANOHARA et K. SADAKANE : Practical entropy-compressed rank/select dictionary. *In Proc. of Algorithm Engineering & Experiments (ALENEX)*, 2007.

D. OKANOHARA et K. SADAKANE : An online algorithm for finding the longest previous factors. *In Proc. of European Symposium on Algorithms (ESA)*, pages 696–707, 2008.

N. PHILIPPE, A. BOUREUX, L. BRÉHÉLIN, J. TARHIO, T. COMMES et E. RIVALS : Using reads to annotate the genome : influence of length, background distribution, and sequence errors on prediction capacity. *Nucleic Acids Research*, 37(15):e104, 2009.

É. PRIEUR et T. LECROQ : On-line construction of compact suffix vectors and maximal repeats. *Theor. Comput. Sci.*, 407(1–3):290–301, 2008.

S. J. PUGLISI, W. F. SMYTH et A. TURPIN : A taxonomy of suffix array construction algorithms. *ACM Comp. Surv.*, 39(2):1–31, 2007.

R. RAMAN, V. RAMAN et S. RAO : Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *In Proc. of Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

R. RAMAN et S. RAO : Succinct dynamic dictionaries and trees. *In Proc. of International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 357–368, 2003.

L. RUSSO, G. NAVARRO et A. OLIVEIRA : Approximate string matching with Lempel-Ziv compressed indexes. *In Proc. of String Processing and Information Retrieval (SPIRE)*, volume 4726 de *Lecture Notes in Computer Science*, pages 265–275. Springer, 2007.

L. RUSSO, G. NAVARRO et A. OLIVEIRA : Fully-compressed suffix trees. *In Proc. of the Latin American Theoretical Informatics (LATIN)*, volume 4957 de *Lecture Notes in Computer Science*, pages 362–373. Springer, 2008.

L. RUSSO et A. OLIVEIRA : A compressed self-index using a Ziv-Lempel dictionary. *In Proc. of String Processing and Information Retrieval (SPIRE)*, volume 4209 de *Lecture Notes in Computer Science*, 2006.

K. SADAKANE : Compressed text databases with efficient query algorithms based on the compressed suffix array. *In Proceedings of the International Conference on Algorithms and Computation*, 2000.

K. SADAKANE : New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.

K. SADAKANE : Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.

K. SADAKANE et H. IMAI : A cooperative distributed text database management method unifying search and compression based on the Burrows-Wheeler transformation. *In Proceedings of the Workshops on Data Warehousing and Data Mining : Advances in Database Technologies*, 1998.

M. SALSON, T. LECROQ, M. LÉONARD et L. MOUCHARD : Dynamic Burrows-Wheeler transform. *In Proc. of Prague Stringology Conference (PSC)*, pages 13–25, 2008.

M. SALSON, T. LECROQ, M. LÉONARD et L. MOUCHARD : Dynamic extended suffix array. *Journal of Discrete Algorithms*, 2009a. Accepted.

M. SALSON, T. LECROQ, M. LÉONARD et L. MOUCHARD : A four-stage algorithm for updating a Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 410 (43):4350–4359, 2009b.

K.-B. SCHÜRMANN et J. STOYE : Counting suffix arrays and strings. *Theor. Comput. Sci.*, 395(2–3):220–234, 2008.

C. E. SHANNON : A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

J. STORER et T. SZYMANSKI : Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.

C. TRAPNELL et L. SALZBERG : How to map billions of short reads onto genomes. *Nature Biotechnology*, 27:455–457, 2009.

E. UKKONEN : Constructing suffix trees on-line in linear time. *In* ELSEVIER, éditeur : *Proc. of Information Processing*, volume 1, pages 484–492. IFIP Transactions A-12, 1992.

S. VIGNA : Broadword implementation of rank/select queries. *In Proc. of Workshop on Efficient and Experimental Algorithms (WEA)*, pages 154–168, 2008.

P. WEINER : Linear pattern matching algorithm. *In Proc. of Symp. on Switching and Automata Theory*, pages 1–11, 1973.

S.S. WONG, W.K. SUNG et L. WONG : CPS-tree : A compact partitioned suffix tree for disk-based indexing on large genome sequences. *In Proc. of International Conference on Data Engineering*, pages 1350–1354, 2007.

J. ZIV et A. LEMPEL : A universal algorithm for sequential data compression. *IEEE Trans. on information theory*, 23:337–343, 1977.

J. ZIV et A. LEMPEL : Compression of individual sequences via variable length coding. *IEEE Trans. on information theory*, 24:530–536, 1978.

Index