Compression et indexation de séquences annotées

Tatiana Rocher

École Doctorale SPI

UMR 9189 CRIStAL

Thèse

Présentée pour l'obtention du DOCTORAT
DE L'UNIVERSITÉ DE LILLE
par

Tatiana ROCHER

Compression et indexation de séquences annotées

Spécialité : Informatique

Soutenue le 12 février 2018 à l'Université de Lille devant un jury composé de :

Rapporteur	**Guillaume BLIN**	(Professeur Univ. Bordeaux, LaBRI)
Rapporteuse	**Lynda TAMINE LECHANI**	(Professeure Univ. Paul Sabatier, IRIT)
Examinatrice	**Laetitia JOURDAN**	(Professeure Univ. Lille, CRIStAL)
Examinateur	**Arnaud LEFEBVRE**	(MdC Univ. Rouen, LITIS)
Directeur de thèse	**Mathieu GIRAUD**	(CR CNRS, CRIStAL)
Co-encadrant de thèse	**Mikaël SALSON**	(MdC Univ. Lille, CRIStAL)

Thèse effectuée au sein de **l'UMR 9189 CRIStAL**

de l'Université de Lille

Bâtiment M3 extension

avenue Carl Gauss

59655 Villeneuve d'Ascq Cedex

France

Résumé

Cette thèse en algorithmique du texte étudie *la compression, l'indexation et les requêtes sur un texte annoté*. Un texte annoté est un texte sur lequel nous ajoutons des informations. Ce peut être par exemple une recombinaison V(D)J, un marqueur de globules blancs, où le texte est une séquence ADN et les annotations sont des noms de gènes. Le système immunitaire d'une personne se représente par un ensemble de recombinaisons V(D)J. Avec le séquençage à haut débit, nous pouvons avoir accès à des millions de recombinaisons V(D)J qui sont stockées et doivent pouvoir être retrouvées et comparées rapidement.

La première contribution de cette thèse est une méthode de compression d'un texte annoté qui repose sur le principe du stockage par références. Le texte est découpé en facteurs pointant vers les séquences annotées déjà connues. La seconde contribution propose deux index pour un texte annoté. Ils utilisent une transformée de Burrows-Wheeler indexant le texte ainsi qu'un Wavelet Tree stockant les annotations. Ces index permettent des requêtes efficaces sur le texte, les annotations ou les deux. Nous souhaitons à terme utiliser l'un de ces index pour indexer des recombinaisons V(D)J obtenues dans des services d'hématologie lors du diagnostic et du suivi de patients atteints de leucémie.

Mots-clés : Algorithme du texte, Structure de données, Bioinformatique, Compression de données, Indexation de texte, Transformée de Burrows-Wheeler, Wavelet Tree, Recombinaisons V(D)J

Compressing and indexing labeled sequences

Abstract

This thesis in text algorithmics studies *the compression, indexation and querying on a labeled text*. A labeled text is a text to which we add information. As an example, in a V(D)J recombination, a marker for lymphocytes, the text is a DNA sequence and the labels are the genes' names. A person's immune system can be represented with a set of V(D)J recombinations. With high-throughput sequencing, we have access to millions of V(D)J recombinations which are stored and need to be recovered and compared quickly.

The first contribution of this thesis is a compression method for a labeled text which uses the concept of storage by references. The text is divided into sections which point to pre-established labeled sequences. The second contribution offers two indexes for a labeled text. Both use a Burrows-Wheeler transform to index the text and a Wavelet Tree to index the labels. These indexes allow efficient queries on text, labels or both. We would like to use one of these indexes on V(D)J recombinations which are obtained with hematology services from the diagnostic or follow-up of patients suffering from leukemia.

Keywords : Stringology, Data structure, Bioinformatic, Data compressing, Text indexing, Burrows-Wheeler transform, Wavelet Tree, V(D)J recombinations

Remerciements

Je souhaite remercier en premier Mathieu Giraud et Mikaël Salson, mes directeurs de thèse. Tout d'abord pour m'avoir permis de réaliser cette thèse, mais aussi pour leurs divers enseignements plus ou moins proches de l'informatique et leur bonne humeur.

Merci à l'Université de Lille et à la région Hauts-de-France qui ont financé cette thèse.

Merci à Lynda Tamine-Lechani et à Guillaume Blin qui ont accepté d'être rapportrice et rapporteur de cette thèse et m'ont fait part de leurs remarques intéressantes. Merci également à Laetitia Jourdan et Arnaud Lefebvre, examinatrice et examinateur dans le jury de soutenance.

Je souhaite remercier Nikos Darzentas qui m'a accueilli une semaine dans son équipe à Brno (République Tchèque). Il m'a expliqué la manière dont fonctionne son logiciel et ces questions et remarques constructives sur mon travail m'ont permis de déterminer ses points forts biologiques. Merci à Tomá Reigl et Andrea Grioni qui ont été des hôtes exemplaires et ont rendus ce séjour inoubliable.

Merci à Maude Pupin, Yann Secq et Philippe Marquet, fondateurs de l'action *L codent, L créent* dans laquelle j'ai enseigné la programmation à des collégiennes. J'ai beaucoup apprécié l'expérience et souhaite à cette action de nombreuses années d'expansion.

Je remercie le consortium EuroClonality-NGS qui m'a invitée à plusieurs de ses réunions biannuelles. Cela m'a permis de rencontrer des utilisateurs de Vidjil et de comprendre l'utilité du logiciel dans les différents thèmes de recherche.

Enfin, merci à mes parents et mes soeurs ainsi que Ryan et sa famille, qui m'ont tous encouragée et m'ont permis de mettre au point les explications de mon travail de thèse.

Table des matières

Introduction

Contexte

Cette thèse, financée par l'université Lille 1 et la région Hauts-de-France, s'est déroulée au sein de l'équipe de bioinformatique Bonsai, commune aux centres de recherche CRIStAL et Inria Lille. L'équipe Bonsai travaille sur des sujets ayant pour point commun l'algorithmique des séquences, qu'elles soient ADN, ARN ou peptidiques.

Avec la collaboration du service d'hématologie et de la plateforme de génomique de Lille, elle a créé le logiciel Vidjil, algorithme d'analyse de répertoire immunologique et plateforme de visualisation de ses résultats. Ce logiciel, utilisé maintenant en routine dans plusieurs hôpitaux, permet de suivre l'évolution de la leucémie d'un patient par un échantillon de ses recombinaisons V(D)J.

Les recombinaisons V(D)J sont des séquences ADN de quelques centaines de nucléotides, constituées de 2 à 3 gènes recombinés. Ce sont des marqueurs efficaces pour estimer l'évolution de la leucémie. Durant le diagnostic et le suivi des patients, de grandes quantités de recombinaisons V(D)J sont analysées.

Face à la génération de plus en plus de données, le but de la thèse était le suivant : *comment indexer ces données ?* Le consortium EuroClonality-NGS [1] réunit les équipes de recherches européennes en hématologie et deux équipes de bioinformatique incluant la nôtre qui propose l'utilisation de Vidjil dans les projets de recherche. Le consortium a émis le souhait de créer une indexation des répertoires immunologiques de patients. Cela s'inscrivait parfaitement dans notre projet d'indexer les recombinaisons V(D)J de patients.

Sur un plan plus formel, nous pouvons voir une recombinaison V(D)J comme une séquence annotée, c'est-à-dire une séquence de caractères ayant des étiquettes sur certaines lettres. Prenons l'exemple de la séquence `AGGGT...GCT` de 95 lettres,

1. http ://www.euroclonalityngs.org/

constituée du gène V_4 aux positions 0 à 42 et du gène J_1 aux positions 67 à 94. Dans cet exemple, les étiquettes sont V_4 et J_1 et les positions 43 à 66 n'ont pas d'étiquette. Nous voulons alors faire des recherches sur les motifs et les annotations : par exemple, y a-t-il un motif `ACT` annoté avec le gène V_1 dans la séquence ?

Répondre à des problèmes liant des objets entre eux pourrait se faire par des recherches en fouille de données, en utilisant des méthodes d'indexation de bases de données. Nos données étant des séquences, nous nous plaçons plutôt ici dans le domaine de *l'algorithmique du texte*, un domaine de recherche sur les algorithmes qui traite les chaînes de caractères. Nous avons des jeux de données avec potentiellement des centaines de millions de caractères. Les travaux en algorithmique du texte cherchent par exemple à compresser un texte, y rechercher un motif, comparer deux textes ... Lorsque ce type de requêtes est demandé régulièrement sur un texte connu à l'avance, il est assez rentable d'indexer ce texte grâce à une structure d'indexation.

Les structures d'indexation proposées par les recherches en algorithmique du texte réorganisent les données pour y avoir un accès dans un temps indépendant de la taille du texte. Les structures plein-texte possèdent toutes les informations se trouvant dans le texte, elles peuvent remplacer le texte et celui-ci peut être retrouvé grâce à une fonction de la structure. Les premières structures d'indexation supportaient la recherche d'occurrences d'un motif dans un texte. Parmi elles se trouve l'arbre des suffixes. Crée en 1973, il est coûteux en mémoire mais permet de compter les occurrences d'un motif en temps proportionnel à la longueur du motif. Des techniques et d'autres structures sont apparues pour utiliser moins de mémoire, allant jusqu'à compresser le texte original, par exemple le FM-index en 2000.

Des algorithmes sur le texte et structures d'indexations sont utilisés lorsque nous voulons stocker un fichier en utilisant le moins de mémoire possible, lors de la détection de plagiat ou lors d'une recherche internet en utilisant une orthographe approximative.

Contenu de la thèse

Cette thèse en algorithmique du texte propose plusieurs manières de compresser, stocker et d'accéder efficacement à un texte annoté.

Ce manuscrit est découpé en quatre chapitres. Le premier chapitre présente le cadre biologique de cette thèse : les recombinaisons V(D)J, quelques techniques hématologiques et bioinformatiques pour les analyser ainsi que notre problématique.

Le deuxième chapitre dresse un état de l'art des structures de données classiques

pour la compression et l'indexation. Nous y présentons la recherche de motifs ainsi que la transformée de Burrows-Wheeler. Ici, tous les algorithmes et structures sont généralement utilisés pour des données unidimensionnelles, d'habitude un seul texte.

Les contributions de la thèse se trouvent dans les deux derniers chapitres. Le troisième chapitre introduit une compression du texte permettant de compresser une grande quantité de séquences annotées. Elle utilise des motifs connus, ici les gènes V, D et J pour faire des références du texte vers les motifs. Nous proposons trois algorithmes de compression, un pour les textes déjà annotés et deux, un optimal et un rapide, pour les textes non annotés et pour lesquels nous devons trouver les annotations. Les tests sur le dernier algorithme indiquent une bonne compression du texte mais cette manière de compresser ne permet pas d'indexer le texte.

Le dernier chapitre propose deux index compressés pour un texte annoté. Les séquences sont stockées dans une transformée de Burrows-Wheeler et les annotations dans un Wavelet Tree. Leur différence réside dans l'ordre dans lequel les annotations sont stockées : dans l'ordre du texte pour le premier index, dans l'ordre de la transformée pour le second. Les index utilisent un espace proportionnel à l'entropie du texte et répondent efficacement à des requêtes sur des annotations, des motifs ou une combinaison annotation/motif.

Autres travaux réalisés

En dehors de mes travaux de recherche, j'ai participé pour près de 10% de mon temps, au développement du logiciel Vidjil et à la vie de l'équipe : réunions, team building, organisation de séminaires ... Cela m'a permis de me familiariser avec le fonctionnement du logiciel, et a abouti à ma participation à plusieurs publications. L'article [10] présente la plateforme Vidjil : l'application web, l'algorithme utilisé ainsi que le serveur et sa base de données. L'article [45] présente le suivi de patients atteints de leucémie dans le service hématologique de Lille en utilisant la méthode classique ainsi que Vidjil.

J'ai ensuite participé au groupe de travail StringMaster en février 2017 à Rouen où j'ai pris plaisir à travailler sur un autre sujet, les mots de Lyndon et leurs liens avec la table des suffixes. Nous avons analysé un article de Baier [2] afin de trouver une construction du tableau de Lyndon plus simple que celle présentée. Pour le moment cette collaboration n'a pas donné lieu à une rédaction d'article.

Enfin, pendant mes deuxième et troisième années, j'ai été doctorante-assistante et

ai encadré des TD à Polytech Lille auprès d'étudiants en troisième et quatrième année de différents cursus. J'ai enseigné plusieurs matières informatiques parmi lesquelles les structures de données, la base de données et la programmation. J'ai participé à la création des sujets, à l'encadrement de projets et la surveillance et correction de devoirs notés.

En plus de ces cours j'ai eu le plaisir de participer à l'atelier *L codent, L créent*[2]. Cette atelier promeut la femme dans l'informatique auprès de jeunes collégiennes de 13 et 14 ans. Il est organisé par le collectif *Ch'ti code*[3] et le groupe de travail *informatique au féminin*[4]. Plusieurs étudiantes et moi-même avons animé des TP pour amener les collégiennes à créer des oeuvres numériques. Cela s'est conclu par une exposition des oeuvres ouverte au familles puis une présentation de l'action via un poster à la conférence internationale womENcourage [43].

2. cours : https ://wikis.univ-lille1.fr/chticode/wiki/ecoles/lclc/2017/home
3. http ://chticode.info
4. http ://femmes.fil.univ-lille1.fr/

Chapitre 1

Immunité adaptative et recombinaisons V(D)J

Ce chapitre présente le contexte biologique de cette thèse. Nous commençons par présenter rapidement le système immunitaire et les recombinaisons V(D)J dans la section 1.1. Puis la section 1.2 présente certains logiciels d'analyse de ces données. Enfin, la section 1.3 indique les problématiques liées à l'indexation de ces séquences, et nos contraintes.

Nous n'expliquerons pas ici les généralités sur les cellules et l'ADN. Ceux-ci, ainsi que les notions d'immunologie de la section 1.1, sont expliqués plus en profondeur dans diverses ressources, notamment dans le livre [27].

1.1 Quelques notions d'immunologie

Le **système immunitaire** est l'ensemble des mécanismes, cellules, tissus et molécules qui défendent le corps face aux diverses infections. Les **lymphocytes B et T**, sous-classes de globules blancs, reconnaissent les corps étrangers ou antigènes. Ils contribuent à l'immunité adaptative, c'est-à-dire que le système s'adapte aux menaces déjà rencontrées afin d'être plus efficace et de les contrer plus rapidement lors d'une nouvelle infection.

Les lymphocytes sont produits dans la moelle osseuse, les lymphocytes B y poursuivent leur maturation pour pouvoir créer des anticorps, ou **immunoglobulines**. Les lymphocytes T se déplacent dans le thymus pour y faire leur maturation et développer

leurs récepteurs de surface, les **récepteurs de cellules T**. Chaque lymphocyte subit une sélection par étapes : ils doivent être fonctionnels mais tolérer les cellules du corps humain. Les lymphocytes quittent ensuite leur lieu de maturation pour voyager dans le corps.

La figure 1.3 montre la reconnaissance d'un antigène par les lymphocytes T et B. Lors d'une infection, les lymphocytes capables de reconnaître l'antigène se multiplient. A la fin de l'infection, ces lymphocytes restent dans le système immunitaire sous forme de lymphocytes mémoire, et permettront une réponse plus rapide à une nouvelle infection présentant le même antigène.

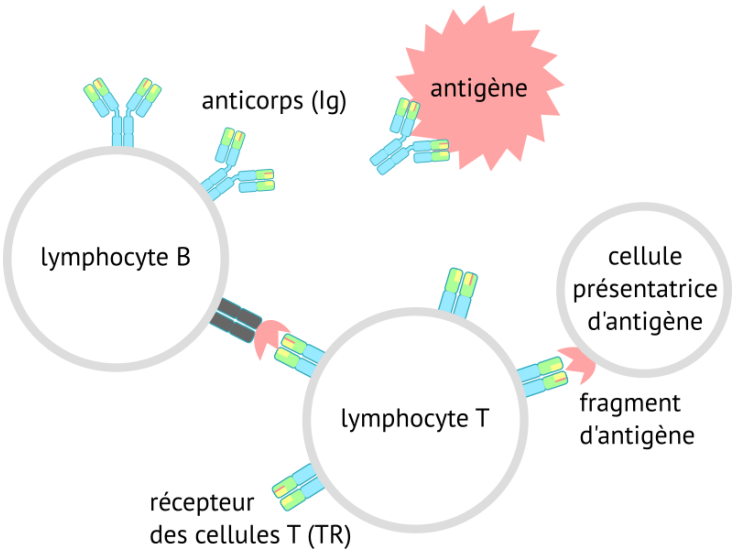


FIGURE 1.1 : Les lymphocytes B produisent des anticorps qui reconnaissent un antigène. Les lymphocytes T reconnaissent un fragment d'antigène présenté par une cellule présentatrice d'antigène [14].

Une grande **diversité immunologique** semi-aléatoire est nécessaire pour combattre la grande variété d'antigènes étrangers. Pour cela, plusieurs mécanismes se succèdent pour créer les séquences d'ADN qui codent pour les immunoglobulines et récepteurs. L'un des mécanismes est la recombinaison V(D)J qui code notamment le CDR3, partie en contact avec l'antigène [48]. Elle rapproche 2 ou 3 gènes parmi les familles de gènes V, D et J pour former une recombinaison VJ ou VDJ (voir figures 1.2 et 1.3). Il y a au total un millier de gènes V, D ou J, répartis en plusieurs groupes, les locus, sur plusieurs chromosomes.

La recombinaison V(D)J, les mutations somatiques (une modification de nucléotide) ainsi qu'un ajout de diversité jonctionnelle (ajout et délétion de nucléotides entre les gènes) provoquent une extrême variabilité de récepteurs. Ces mécanismes créent un

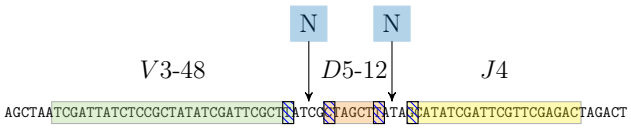


FIGURE 1.2 : Une recombinaison VDJ rapproche un gène V, un gène D et un gène J. Des ajouts de nucléotides (portion N) et délétions d'extrémités de gène (zones hachurées), toutes deux de composition et de taille aléatoires, ajoutent de la diversité.

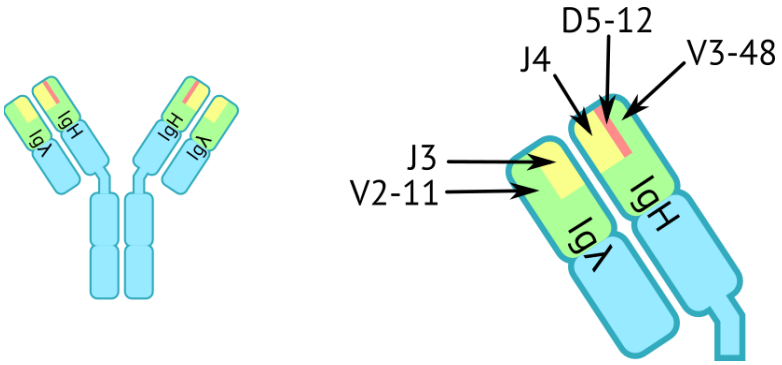


FIGURE 1.3 : Cet anticorps est constitué de deux chaînes lourdes ayant subit une recombinaison VDJ (V3-48 D5-12 J4) et deux chaînes légères ayant subit une recombinaison VJ (V2-11 J3)[14].

nombre d'immunoglobulines et récepteurs de cellules T différents estimé à 10^{12} [50] ou 2^{74} [30].

La **leucémie aiguë lymphoblastique** (LAL), est un cancer de la moelle osseuse touchant le plus souvent les enfants [28]. Elle provoque une multiplication de lymphocytes immatures, des lymphoblastes. Ceux-ci se multiplient continuellement et empêchent la fabrication de cellules normales fonctionnelles. Le lymphoblaste cancéreux peut être identifié au diagnostic et traité. Nous appelons **clone** un ensemble de lymphoblastes identiques. Lors du suivi du patient, la concentration des clones est évaluée afin de déterminer l'évolution du clone cancéreux et éventuellement détecter une rechute de la maladie.

Le suivi hématologique classique identifie la recombinaison VDJ du lymphoblaste et quantifie la présence du clone au cours du temps. La principale limite de cette technique est qu'elle ne suit qu'un clone. Ainsi, on ne pourra identifier une rechute de la maladie provenant d'un clone muté ou non identifié au diagnostic. Plusieurs outils informatiques ont été développés afin d'analyser les recombinaisons V(D)J de plusieurs clones.

1.2 Le séquençage de répertoires

Afin d'avoir un échantillonnage suffisant du système immunitaire d'une personne, nous avons besoin de milliers, voire de millions de recombinaisons VDJ. Elle sont fournies maintenant rapidement grâce aux **séquenceurs à haut débit**. Des algorithmes bioinformatiques sont capables de trouver les gènes composant une recombinaison V(D)J. La figure 1.4 représente des recombinaisons V(D)J et la position probable des ces gènes après analyse bioinformatique. L'un des logiciels de référence est IMGT/V-QUEST [6], développé depuis les années 90 à Montpellier. Il nécessite plusieurs heures de calcul pour un maximum de 500 000 séquences.

```
>TRGV2*02:0-177 TRGJ1*01:192-226
GGAAGGCCCCACAGCGTCTTCAGTACTATGACTCCTACAACTCCAAGGTTGTGTTGGAA (...)

>IGHV3-11*01:0-251 IGHD6-19*01:260-279 IGHJ5*02:285-331
GGAGGTCCCTGAGACTCTCCTGTGCAGCCTCTGGATTCACCTTCAGTGACTACTACATG (...)
```

FIGURE 1.4 : Extrait de deux séquences annotées. La première séquence est annotée avec le gène `TRGV2*02` aux positions 0 à 177 et le gène `TRGJ1*01` aux positions 192 à 226.

Des algorithmes de plus en plus rapides ont été créés pour améliorer le temps d'analyse, par exemple MIXCR [4]. Ils utilisent entre autres des k-mers et la programmation dynamique (expliquée section 2.3) avec k-band. Ces algorithmes analysent les séquences une par une puis les regroupent pour quantifier les clones, voir figure 1.5.

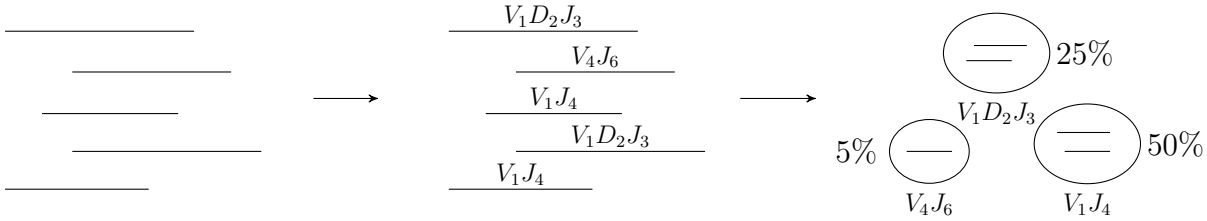


FIGURE 1.5 : Analyse des données séquence par séquence. Chaque séquence est analysée puis elles sont regroupées en clones.

Une autre approche est possible : elle consiste à former les clones à partir des séquences qui se ressemblent, et trouver les gènes constituant les clones ensuite (voir figure 1.6). Le logiciel Vidjil [10], développé par mon équipe, utilise des graines pour évaluer l'appartenance de chaque k-mer d'une séquence au groupe des gènes V ou J. De cette manière il trouve les positions approximatives des gènes V et J, sans déterminer les

gènes exacts, puis positionne une fenêtre de taille fixe sur la position estimée du CDR3. Toutes les séquences ayant la même fenêtre font partie du même clone. Il cherche ensuite les gènes V, D et J de la séquence représentative du clone. Le regroupement est fait très rapidement et la désignation V(D)J, plus longue, est réalisée après le regroupement sur chacun des clones. Cela rend l'analyse extrêmement rapide car aucun alignement n'est réalisé dans la première phase.

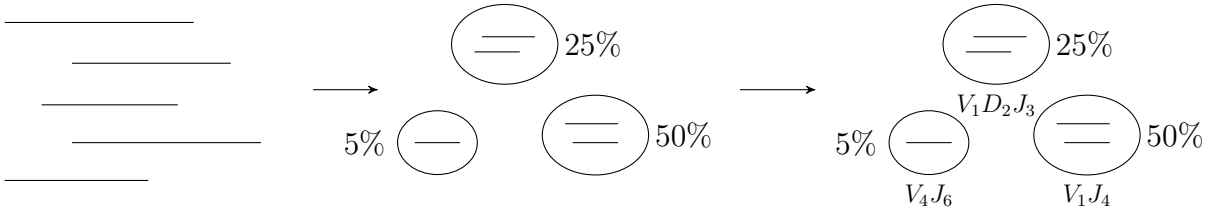


FIGURE 1.6 : Les séquences sont d'abord regroupées en clones par ressemblance. Puis une séquence représentative du clone est analysée.

1.3 Stocker et requêter les recombinaisons V(D)J

De plus en plus de recherches sont effectuées pour comprendre les causes et le fonctionnement de la leucémie. Certaines de ces recherches nécessitent de connaître en profondeur le répertoire immunologique. Par exemple pour faire des comparaisons de répertoire entre plusieurs patients, nous devons identifier un grand nombre de clones afin de savoir lesquels sont partagés entre deux patients et donc pouvoir y faire des recherches autant sur les motifs que les gènes. Nous avons donc besoin de données stockées efficacement, accessibles et requêtables rapidement. Le but de cette thèse est de *compresser et d'adapter ou créer un index pour les recombinaisons V(D)J* (voir figure 1.7).

FIGURE 1.7 : Notre structure d'indexation va indexer les recombinaisons V(D)J. Nous pourrons ensuite effectuer des requêtes qui vont nous donner les séquences recherchées, par exemple trouver les séquences qui ont un facteur `ATTC` et une annotation V_1.

Ces remarques ont permis d'établir un cahier des charges contenant plusieurs points de réflexion. Notre index devrait :

— utiliser le moins de mémoire possible pour stocker une séquence annotée avec des gènes V, D et J (compression, chapitre 3 et indexation compressée, chapitre 4)

— trouver les séquences qui incluent une sous-séquence donnée et/ou sont annotées par un gène donné (indexation compressée, chapitre 4)

Le chapitre 2 présente quelques structures usuelles en algorithmique du texte et détaille pourquoi celles-ci ne conviennent pas ici.

Chapitre 2

Algorithmique du texte

L'algorithmique du texte, aussi nommée stringologie, est le domaine de recherche sur les algorithmes agissant sur un texte. Ce chapitre définit les notions qui seront utilisées tout au long de ce document. La section 2.1 introduit quelques notations. Nous présentons deux algorithmes de compression dans la section 2.2. Parmi les problèmes importants du domaine se trouve la localisation de motif dans un texte, présentée en section 2.3. Les algorithmes font souvent appel à des structures de données, certaines d'entre elles sont détaillées dans la section 2.4. La section 2.5 présente la transformée de Burrows-Wheeler, un algorithme réorganisant les lettres d'un texte, qui sera largement utilisée dans le chapitre 4. Enfin, la section 2.6 introduit formellement notre problématique : comment stocker et requêter des séquences annotées ?

2.1 Définitions

Un **alphabet** est un ensemble non vide d'éléments appelés les **lettres**. Une lettre est un symbole quelconque. Une **séquence** de taille n est une suite finie de lettres $w = w_0 w_1 \dots w_{n-1}$ appartenant à un alphabet $\Sigma - \{\$\}$, de taille σ, sauf le caractère final qui est le symbole \$. Un **mot**, ou **texte**, est composé d'une ou de plusieurs séquences. La longueur du mot w, notée $|w|$, est le nombre de lettres qui le compose. Le **mot vide** ϵ est le mot de longueur nulle. Chaque lettre d'un mot w est placée à une **position** de w. La lettre à la position i du mot w, w_i, est la $i+1$-ème lettre de w.

Le mot y est un **facteur** ou **occurrence** du mot xyz, avec $x, y, z \in \Sigma^*$. Si x est vide, alors y est un **préfixe** de xyz. Si z est vide, alors y est un **suffixe** de xyz. On dit que y **apparaît** à la position i de w si la première lettre de y est placée à la

position i de w. On écrira $T_{i,j} = t_i t_{i+1} \dots t_j$ un facteur du texte T de longueur $j - i + 1$ apparaissant à la position i.

Le mot yx est une **rotation**, ou **permutation circulaire**, du mot xy. Il existe n rotations d'un mot de taille n. La $i+1$-ème rotation du mot $w = w_0 w_1 \dots w_{n-1}$ est le mot $w_i \dots w_{n-1} w_0 \dots w_{i-1}$.

L'**ordre lexicographique** est un ordre sur les mots induit par un ordre sur les lettres. Soient v et w, deux mots sur l'alphabet Σ, $v < w$ si soit v est un préfixe de w, soit $v = xay$ et $w = xbz$ et $a < b$, avec $a, b \in \Sigma$ et $x, y, z \in \Sigma^*$.

Soient v et w, deux mots sur l'alphabet Σ, avec $v \neq w$. La **distance d'édition**, ou distance de Levenshtein, entre ces deux mots est le plus petit nombre d'opérations nécessaires pour transformer v en w. Les opérations possibles sont (voir figure 2.1) :

— la **substitution** d'une lettre par une autre
— la **délétion** d'une lettre
— l'**insertion** d'une lettre

(1)	(2)	(3)	(4)
AABAAA AAAAAA	AABAAA AA-AAA	AAB-AA AABAAA	ABBA AAB-

FIGURE 2.1 : Les différentes opérations de modifications d'une séquence. Le caractère - représente une absence de lettre, il n'est utilisé que pour la représentation d'exemples. Une substitution est représentée en (1), un `B` devient `A`, une délétion est en (2), on supprime la lettre `B`, et une insertion est en (3), on ajoute une lettre `A`. Ici la distance d'édition entre chaque couple est de 1. (4) La distance d'édition entre les mots `ABBA` et `AAB` est 2, en substituant la deuxième lettre et supprimant la dernière.

2.2 Compression

Nous pouvons représenter un texte en utilisant moins d'espace mémoire que sous sa forme originale en utilisant une **compression**. Une compression sans perte signifie que le fichier compressé puis décompressé est strictement identique à l'original, contrairement à la compression avec perte. Le rapport entre le volume avant et après compression est le **quotient de compression**. L'entropie est une mesure de la *surprise* d'un texte, elle indique si un texte peut être bien compressé ou non. Si après la lecture d'une partie du texte, nous pouvons déterminer la lettre suivante, alors l'entropie est faible. L'**entropie** d'un texte T est notée $H_x(T)$ où x est l'**ordre** de l'entropie.

$H_0(T)$ est l'entropie d'ordre zéro de T, où nous considérons la possibilité d'apparition de chaque lettre indépendamment et $H_k(T)$, l'entropie d'ordre k de T qui est fonction de la probabilité d'apparition d'une lettre en fonction des k précédentes. Il est possible de transformer le texte avant de le compresser pour avoir un meilleur taux de compression, par exemple avec une transformée de Burrows-Wheeler, voir la section 2.5.

Nous présentons deux algorithmes de compression : le codage de Huffman et l'algorithme LZ76.

Le **codage de Huffman** [18] est un algorithme de compression de données sans perte. Il consiste à coder chaque élément avec un code à longueur variable déterminé grâce à un arbre binaire (voir figure 2.2). Les doublons élément/poids constituent les feuilles de notre arbre, le poids étant le nombre d'occurrences de l'élément. Puis nous construisons l'arbre de la manière suivante : les deux nœuds de plus petit poids sont descendants d'un nouveau nœud de poids égal à leur somme, le tout jusqu'à obtenir un seul nœud. Pour obtenir le code d'un élément nous suivons le chemin de la racine à la feuille correspondant à cet élément et ajoutons un bit 0 (resp. 1) à chaque fois que nous allons dans le sous-arbre gauche (resp. droit). A partir d'une distribution de probabilité connue, un code de Huffman propose la plus courte longueur de texte pour un codage par symbole. La longueur du code d'un élément est déterminée par la fréquence d'apparition de cet élément dans le texte. La taille d'un texte T codé avec le codage de Huffman est fonction de l'entropie d'ordre 0 ($H_0(T)$).

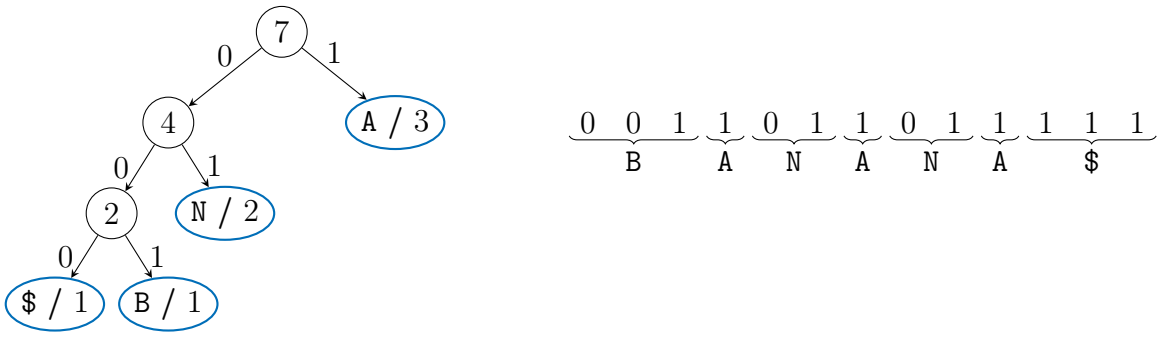


FIGURE 2.2 : Arbre de Huffman pour définir le code du texte de 7 lettres `BANANA$`. Le code est ensuite utilisé pour écrire le texte : 0011011011111, utilisant 13 bits. Le caractère `A` est très fréquent (présent 3 fois), il est donc représenté avec un seul bit : 1. À l'inverse, le caractère `B` n'est que peu présent (1 fois) et est représenté en 3 bits : 001.

LZ76 [20] et ses variantes (LZ77, LZ78 ...) sont des algorithmes de compression de données sans perte, proposés par Lempel et Ziv. Ils utilisent des références vers des

facteurs connus, c'est une compression par dictionnaire. L'algorithme LZ76 découpe un texte T de taille n en $n' \leq n$ facteurs z_i appelés phrases, tels que $z_0 z_1 \dots z_{n'-1} = T$, voir figure 2.3. Ce découpage du texte T sur un alphabet de taille σ produit $n' = O(n/\log_\sigma n)$ phrases [34]. Supposons que nous avons découpé $T_{0,i-1}$ en facteurs $z_0 \dots z_{i'-1}$, le facteur $z_{i'}$ est :

- soit t_i si cette lettre n'est jamais apparue dans $T_{0,i-1}$
- soit le plus long préfixe $z_{i'}$ de $T_{i,n-1}$ tel que $z_{i'}$ est présent dans $T_{0,i-1}$. La position de $z_{i'}$ dans $T_{0,i-1}$ est appelée la source de $z_{i'}$

Le codage utilisant cette découpe est composé de doublons (*source, taille*) et de lettres uniques (exceptionnel car utilisé seulement lors de leur première occurrence).

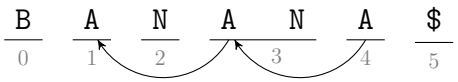


FIGURE 2.3 : Découpage du mot `BANANA$` avec l'algorithme LZ76. Les trois premières lettres sont des premières occurrences, elles provoquent la formation de trois facteurs de taille 1. Les quatrième et cinquième facteurs `AN` et `A` sont déjà présents en position 1 et 3. Le dernier facteur est la première occurrence de `$` et est de taille 1. Le codage de `BANANA$` est $\texttt{B}, \texttt{A}, \texttt{N}, (1,2), (3,1), \texttt{\$}$.

2.3 Recherche de motif

Un problème courant dans l'algorithmique du texte est la **recherche de motif exact ou approché**. Un motif est un mot dont nous cherchons toutes les occurrences dans un texte. Ce problème est très répandu en bioinformatique pour reconstituer un génome à partir de segments issus de séquençage ou pour rechercher un gène particulier dans une séquence.

L'algorithme naïf de recherche de motif exact consiste à tester si le motif est présent à chaque position du texte. Il a une complexité quadratique. Beaucoup d'autres sont plus rapides : KMP [19], Boyer-Moore [5], le parallélisme de bits [1]. Les mots à jokers (ou graines) utilisés sur les mêmes algorithmes que précédemment [13] et l'alignement par programmation dynamique (voir section 2.3) permettent de rechercher des motifs approchés. Les meilleures algorithmes ont une vitesse d'exécution proportionnelle à la taille du motif, ils sont exécutés sur des structures nécessitant un prétraitement dont le temps est lié à la taille du texte.

Les structures d'indexation organisent un texte pour le traiter plus facilement et plus rapidement. Ces structures nécessitent un pré-traitement du texte et permettent au mieux une recherche de motif en temps proportionnel à la longueur du motif et non du texte. Pour arriver à de tels résultats, certaines structures d'indexation nécessitent beaucoup de mémoire. De plus, certaines structures permettent la recherche de motifs exacts comme approchés.

Parmi eux se trouve l'arbre des suffixes qui reconnaît tous les suffixes d'un texte et permet le dénombrement des occurrences d'un motif P en temps $O(|P|)$. Cet arbre est plus amplement défini dans la section 2.4.1.

Programmation dynamique pour la comparaison de mots

La programmation dynamique est une classe d'algorithmes qui consistent à découper un problème en sous-problèmes, les résoudre du plus petit au plus grand en stockant les résultats intermédiaires jusqu'à arriver au résultat du problème original. L'une de ses applications, l'alignement de textes, calcule la plus petite distance d'édition entre deux textes T de taille t et S de taille s.

Soient c_{sub}, c_{del} et c_{ins} des coûts positifs associés aux trois opérations d'édition. La programmation dynamique consiste à remplir une matrice M de taille $(t+1) \times (s+1)$, dont toute case $M[i][j]$ indique le meilleur coût de l'alignement entre $T_{0,i-1}$ et $S_{0,j-1}$.

La matrice est résolue de la manière suivante :

$$M[i][j] = \min \begin{cases} M[i][j-1] + c_{ins} \\ M[i-1][j] + c_{del} \\ M[i-1][j-1] \text{ si } t_i = s_j \\ M[i-1][j-1] + c_{sub} \text{ sinon} \end{cases}$$

L'initialisation de la matrice est fonction du problème à résoudre. Si nous cherchons la meilleure manière d'aligner T et S, nous effectuons un alignement **global**, et utilisons l'algorithme de Needleman-Wunsch [35]. L'initialisation de la matrice est : $M[i][0] = i \times c_{del}$ et $M[0][j] = j \times c_{ins}$, $\forall i \in [0,t], j \in [0,s]$. Le coût de l'alignement entre T et S est la valeur se trouvant dans la case $M[t][s]$. Voir la figure 2.4.

La recherche de motifs P, proches à k erreurs près, dans le texte T revient à effectuer un alignement **semi-global** entre T et P puis à chercher les cases dont les résultats sont inférieurs à k sur la dernière ligne. L'initialisation de la matrice est : $M[i][0] = 0$ et $M[0][j] = j \times c_{ins}$, $\forall i \in [0,t], j \in [0,s]$. Voir la figure 2.5.

Un alignement **local** permet de trouver des facteurs communs à deux textes, il

s'agit de l'algorithme de Smith-Waterman [47], la matrice est initialisée comme suit : $M[i][0] = M[0][j] = 0$, $\forall i \in [0,t], j \in [0,s]$. Toutes les cases de la matrice ayant une valeur inférieure à k reflètent un alignement local avec moins de k erreurs. Nous ne nous intéressons ici qu'aux deux premières méthodes d'alignement.

Afin de trouver l'alignement global, nous commençons à la case $M[t][s]$ et traversons la matrice jusqu'à arriver à la case $M[0][0]$ avec la règle suivante : depuis la case $M[i][j]$, la case suivante est la case, parmi $M[i-1][j]$, $M[i][j-1]$ et $M[i-1][j-1]$, dont la valeur est minimale. En cas d'égalité l'une ou l'autre des cases est choisie. Notons qu'à partir d'une case, plusieurs alignements sont possibles. Voir la figure 2.4.

Dans un alignement semi-global à k erreurs près, nous commençons aux cases $M[i][s]$, telles que $M[i][t] \leq k$, puis nous traversons la matrice de la même manière que précédemment jusqu'à la première case $M[j][0], \forall j \in [1,s]$ trouvée. Voir schéma 2.5.

		A	T	G	G	A
	0	1	2	3	4	5
A	1	0	1	2	3	4
C	2	1	1	2	3	4
G	3	2	2	1	2	3
G	4	3	3	2	1	← (2)

FIGURE 2.4 : Programmation dynamique pour aligner le texte `ATGGA` et le texte `ACGG` avec un alignement global. Ici $c_{ins} = c_{del} = c_{sub} = 1$. La case $M[5][4]$ (entourée) nous indique que l'alignement se fait avec 2 erreurs. Le chemin fléché nous indique les modifications apportées au premier texte : délétion de la lettre `A` en dernière position et substitution du `T` en `C` en deuxième position.

2.4 Quelques structures de données pour l'indexation

Les structures de données permettent d'organiser les données afin de les traiter plus facilement. Nous pouvons rechercher un motif dans un texte très rapidement grâce à un arbre des suffixes (voir section 2.4.1). Les vecteurs de bits et le Wavelet Tree, définis dans la section 2.4.2, accèdent et comptent les éléments d'un texte (bit ou lettre) rapidement.

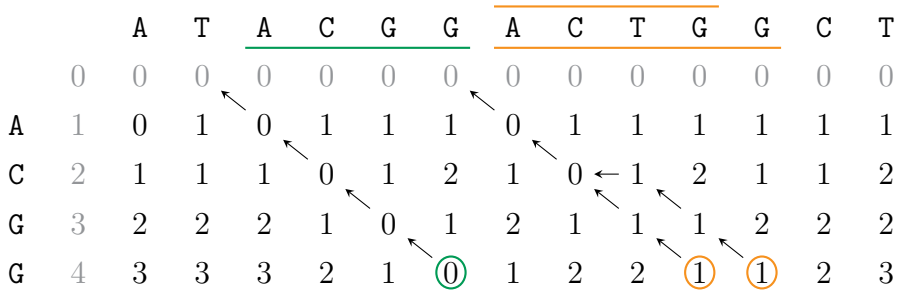


FIGURE 2.5 : Programmation dynamique pour trouver les occurrences du motif `ACGG` dans un texte par alignement semi-global. Nous avons : $c_{ins} = c_{del} = c_{sub} = 1$. Nous autorisons au maximum une erreur. Les positions entourées sont les minimums locaux. Nous trouvons un alignement exact ainsi que deux alignements provoquant une erreur, mais dont les débuts sont à la même position. Il y a donc deux occurrences du motif à une erreur près.

2.4.1 Arbre des suffixes

L'**arbre des suffixes** d'un mot est un automate déterministe acyclique reconnaissant l'ensemble des suffixes du mot, voir figure 2.6. Un chemin est l'ensemble des transitions menant de la racine à une feuille. Un chemin représente le mot issu de la concaténation des étiquettes de ses transitions. Chaque feuille indique la position de début du suffixe représenté par le chemin. L'arbre des suffixes peut être compacté en transformant chaque chemin unaire en une transition ; l'étiquette de la transition est alors un mot, représenté par sa longueur et un pointeur sur le texte. Il peut être construit en temps linéaire [29, 49, 51] tout en utilisant un espace mémoire de $O(n \log n)$ bits.

Il permet de compter les occurrences d'un motif P de taille m en temps optimal $O(m)$. En partant de la racine de l'arbre, il faut lire le motif jusqu'à :

- soit ne plus pouvoir avancer dans l'arbre car la transition par la lettre suivante du motif n'existe pas, et alors le motif n'est pas présent dans l'arbre ;
- soit avoir fini de lire le motif et :
  - soit on se trouve sur une feuille, et le motif se trouve une seule fois dans le texte ;
  - soit on se trouve sur un noeud interne, et le motif est présent autant de fois qu'il y a de feuilles dans le sous-arbre descendant de ce noeud (donnée qui peut être stockée dans le noeud).

La localisation des motifs se fait en temps optimal $O(m + occ)$, avec occ le nombre

d'occurrences du motif.

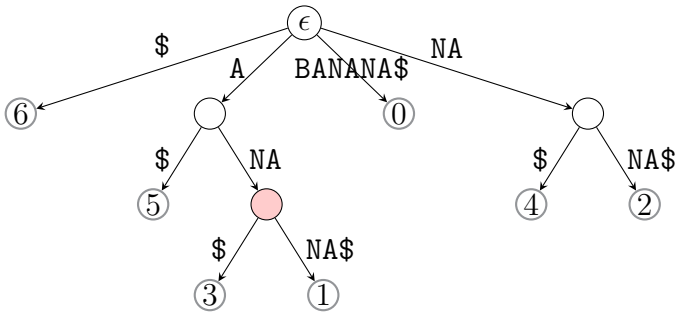
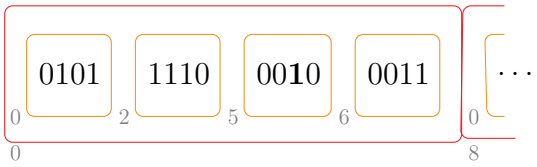


FIGURE 2.6 : Arbre des suffixes compacté du mot `BANANA$`. Le chemin allant de la racine à la feuille étiquetée 3, représente le suffixe `ANA$`, se trouvant à la position 3 du mot. La recherche du motif `ANA`, nous conduit à un noeud interne (nœud plein). Deux feuilles en sont les descendantes, le motif est présent deux fois dans le mot, aux positions 3 et 1.

2.4.2 Vecteur de bits et Wavelet Tree

Dans un **vecteur de bits** B, $B[i]$ est le $i+1$ème élément de B et $B[i,j]$ est le vecteur $b_i, b_{i+1}, \ldots, b_j$. La fonction $rank(b,i,B)$ renvoie le nombre de fois que le bit de valeur b apparaît dans le préfixe $B[0,i]$. La fonction $select(b,j,B)$ renvoie la position i du $j$ème bit de valeur b dans B. Soit le vecteur de bits $B = 0101111000100011$, représenté dans la figure 2.7. Il y a $rank(1,10,B) = 6$ bits 0 dans le facteur $B[0,10]$ et le quatrième bit 1 de B se trouve à la position $select(1,4,B) = 5$. Raman *et al.* [38] et Pagh [37] ont prouvé qu'un vecteur de bits B peut être stocké en utilisant $nH_0(B)+o(n)$ bits tout en supportant les fonctions *rank* et *select* en $O(1)$, où $H_0(B)$ est l'entropie d'ordre 0 de B. La méthode est appelée **RRR vecteurs**.

De manière simplifiée, le fonctionnement de $rank(1,i,B)$ est le suivant : soit un vecteur de bits B de longueur n (en supposant n puissance de 4 pour l'explication). Nous découpons B en superblocs de $\log^2 n$ bits. Chacun de ces superblocs est découpé en blocs de longueur $\log n$. Une table *superblock* indique le nombre de bits 1 se trouvant avant le superbloc courant. Une table *block* indique le nombre de bits 1 se trouvant avant le bloc courant, à l'intérieur du superbloc courant. Nous stockons une table *smallrank* de taille $[0, \sqrt{n}-1][0, \frac{\log n}{2}]$. Cette table indexe chaque vecteur de longueur $\frac{\log n}{2}$ et indique le nombre de bits 1 se trouvant dans chaque préfixe : $smallrank[V][j]$ indique le nombre de bits 1 se trouvant dans $V_{0,j}$. $rank(1,i,B)$ est la somme des valeurs des cases du superbloc, du bloc ainsi que de smallrank courants (voir figure 2.7).



smallrank	0	1
00	0	1
01	0	1
10	1	1
11	1	2

FIGURE 2.7 : Découpage d'un vecteur B pour permettre le fonctionnement de la fonction $rank$ en $O(1)$. Les indices proches des blocs sont les valeurs des tables $superblocks$ et $blocks$. La fonction $rank(1, 10, B)$ est résolue en faisant la somme : $superblock[0] + block[2] + smallrank[00][1] + smallrank[10][0] = 0 + 5 + 0 + 1 = 6$

Afin de permettre la résolution efficace de la fonction *select*, le vecteur de bits n'est plus découpé en blocs de même taille mais en blocs ayant x bits 1, avec x valant $\log^2 n$, ou $\log n$. Pour les blocs trop longs (ayant beaucoup de bits 0 pour peu de bits 1), les solutions sont écrites explicitement.

La solution stockant le vecteur de bit en $nH_0(B) + o(n)$ bits est une optimisation de la solution présentée ci-dessus. Les blocs sont triés par classe, avec la classe 1 étant tous les blocs ayant un seul bit 1, la classe 2, tous les blocs ayant deux bits 1, ... Il y a $\log(n+1)$ classes possibles et dans chaque classe k, il y a $\binom{\log(n+1)}{k}$ vecteurs. Ainsi chaque bloc peut être représenté par son numéro de classe en $\log\log n$ bits et sa position dans la classe k en $\log\binom{\log(n+1)}{k}$ bits.

Le **Wavelet Tree** (WT) permet d'étendre les fonctions rank et select à un alphabet quelconque. Nous pouvons par exemple compter les occurrences d'une lettre dans un facteur ou accéder à une occurrence d'un lettre en temps $O(\log \sigma)$. Défini par Grossi *et al.* [16], le WT est un arbre binaire où chaque symbole de l'alphabet correspond à une feuille et chaque noeud interne est un vecteur de bits. La racine est un vecteur de bits où chaque position correspond à l'élément qu'il indexe. Les positions marquées 0 correspondent aux éléments dont les feuilles se situent dans le sous-arbre gauche. Les feuilles correspondantes à une position marquée 1 se trouvent dans le sous-arbre droit. Ce processus est répété récursivement jusqu'aux feuilles.

Pour un texte de longueur n construit à partir d'un alphabet de taille σ, la construction d'un Wavelet Tree équilibré nécessite un temps de $O(n\lceil \log \sigma / \sqrt{\log n} \rceil)$ [32] et une taille de $n \log \sigma (1 + O(1))$ bits [16]. Les vecteurs de bits d'un WT répondent aux fonctions rank et select. Cela permet à un WT équilibré d'accéder à la lettre $W[i]$ en temps $O(\log \sigma)$ et d'accéder à toutes les positions d'une lettre c en temps $O(occ \times \log \sigma)$,

avec *occ* le nombre d'occurrences de c, voir schéma 2.8. Un WT ayant la forme d'arbre de Huffman, construit en temps $O(nH_0)$ [17], permet d'accéder à une lettre d'un texte T avec un temps moyen de $O(H_0(T))$. Un WT permet d'obtenir une réorganisation d'une séquence de lettres (en considérant les lettres dans l'ordre des feuilles) [33].

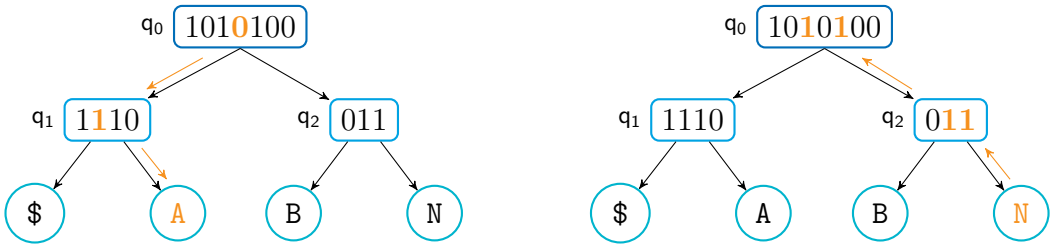


FIGURE 2.8 : Ce Wavelet Tree représente le mot $T =$ `BANANA$`. (gauche) Nous voulons connaître $T[3]$. Le quatrième bit du nœud q_0 est 0 donc la lettre est dans le sous-arbre gauche de q_0. C'est le deuxième 0 de q_0, le deuxième bit du nœud q_1 est 1. La lettre est donc dans le sous-arbre droit, il s'agit d'un `A`. (droite) Nous cherchons toutes les occurrences de la lettre `N` dans le texte. La feuille dont le label est `N` est dans le sous-arbre doit de son parent, donc nous regardons tous les bits 1 du nœud q_2. Celui-ci nous indique qu'il y a 2 occurrences de la lettre `N`. Ces bits sont les 2^e et 3^e bits de q_2 et ce nœud est dans le sous-arbre droit de q_0. Les occurrences de `N` sont les 2^e et 3^e bits 1 de q_0, soient les lettres aux positions 2 et 4.

2.5 La transformée de Burrows-Wheeler

2.5.1 Construction

La **transformée de Burrows-Wheeler** (BWT) [7] est un algorithme réversible qui réorganise les lettres d'un texte, permettant une meilleure compression de celui-ci. Elle est formée de la manière suivante : soit un texte T, dont la lettre finale est \$. Créons toutes les rotations possibles du texte, puis trions-les dans l'ordre lexicographique, ceci forme la matrice M. Nous nommons la première colonne de M, F et la dernière L. L est le texte transformé par la BWT.

Les opérations précédentes sont représentées sur la figure 2.9, où le texte `BANANA$` est transformé en `ANNB$AA`. Sur ce court texte, l'utilité de la BWT n'est pas évidente à voir. Mais sur un texte plus long, par exemple la pièce de théâtre Hamlet, (voir la figure 2.10), la BWT construit beaucoup de zones de lettres consécutives identiques, permettant une meilleure compression.

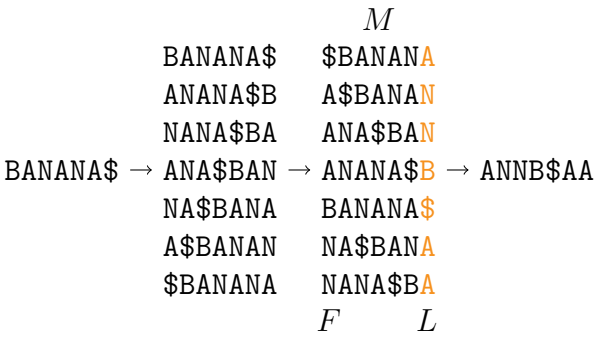


FIGURE 2.9 : Construction de la transformée de Burrows-Wheeler pour le texte `BANANA$`. La première matrice présente toutes les rotations du mot. La seconde, M a les rotations triées dans l'ordre lexicographique.

```
nnnnnnnnnnnnnnnnnnnntnnnnnnnhnnngnnnnnnnnjnnnnnhdnnng
nnnonnNnnnhhNnnnnnnnnnntnnhnnnnnnnnnnnnnnnNnndnnnhnn
19nt7nh3ng8nj5nhd3ng
3no2nN3n2hN9nt2nh14nN2nd3nh2n
```

FIGURE 2.10 : Extrait de la BWT de la pièce de théatre *Hamlet* de Shakespeare en anglais (deux premières lignes) et une manière de représenter cet extrait (deux dernières lignes)

Formellement, le texte transformé est la concaténation de la dernière lettre des rotations du texte, rangées dans l'ordre lexicographique. En pratique, les rotations ne sont pas créées, et nous utilisons des références vers les positions du texte. Nous nommons $T^{bwt} = L$, la transformée de Burrows-Wheeler d'un texte T. Soit t_i une lettre de T se trouvant à la position j dans T^{bwt}, nous disons que t_i correspond à la lettre T^{bwt}_j.

Nous avons annoncé dès la première ligne de cette section que cette transformée est réversible. En effet, depuis T^{bwt} seulement, nous somme capable de trouver T. T^{bwt} est la dernière colonne de M, et nous savons qu'en rangeant les lettres dans l'ordre lexicographique, nous obtenons la colonne F de M. Si nous effectuons une rotation de M, L est à gauche de F. De cette observation, nous retrouvons T de la manière suivante et illustrée dans la figure 2.11 : dans une matrice M', mettre la colonne L. Ranger les lignes dans l'ordre lexicographique. Placer à gauche la colonne L et ranger les lignes lexicographiquement. Recommencer autant de fois qu'il y a de lettres dans le texte. Nous obtenons la matrice M où la première ligne est le texte T précédé du symbole `$`.

La matrice étant imposante, la propriété suivante est utilisée pour retrouver le

1	2	3	4		14
A	$	A$	$B		$BANANA
N	A	NA	A$		A$BANAN
N	A	NA	AN		ANA$BAN
B →	A →	BA →	AN →	... →	ANANA$B
$	B	$B	BA		BANANA$
A	N	AN	NA		NA$BANA
A	N	AN	NA		NANA$BA

FIGURE 2.11 : Retrouver le texte original à partir du texte transformé : `ANNB$AA`. Dans les étapes impaires, nous ajoutons la colonne L à gauche de la matrice. Dans les étapes paires, nous trions la matrice dans l'ordre lexicographique. Le texte original se trouve à la première ligne (précédé du symbole `$`) : `BANANA`.

texte original.

Propriété : Les occurrences d'une même lettre sont dans le même ordre dans L et F.

Regardons la matrice de droite de la figure 2.11 et l'ordre des lettres `N`. Le premier `N` de la colonne F est suivi de `A$`, tandis que le second est suivi de `AN`. Donc le suffixe `NA$...` se trouve avant `NAN...` dans la matrice, en faisant une rotation `A$ ...N` se trouve avant `AN ...N`. Donc les lettres `N` sont dans le même ordre dans les colonnes F et L.

Plus formellement, soient αu et αv deux rotations d'un mot, avec α une lettre. Si $\alpha u < \alpha v$, alors $u < v$, et $u\alpha < v\alpha$. Donc ces deux rotations sont placées dans le même ordre l'une par rapport à l'autre dans M. Cet argument est valable pour toutes les rotations de M.

La propriété précédente nous permet, pour une lettre t_i de la colonne L, de trouver sa position p dans la colonne F : la lettre t_i de valeur α est la $j$ème occurrence de la lettre α dans L et dans F. Maintenant, si nous regardons la lettre se trouvant à la position p dans la colonne L, nous trouvons la lettre t_{i-1}, soit la lettre se trouvant avant t_i dans l'ordre du texte original (avec si $i = 0$, $i - 1 = n - 1$). En utilisant les colonnes F et L, nous pouvons réitérer cette procédure pour naviguer dans T^{bwt} afin de rechercher un motif ou reconstituer le texte original (voir schéma 2.12). Cet algorithme est nommé **la recherche inverse**.

Au lieu de stocker les deux permutations du texte, la colonne F peut être représentée par une table C de taille σ, où $C[\alpha]$ indique la somme des nombres d'occurrences

des lettres lexicographiquement plus petites que α. Une fonction $Occ(\alpha, i)$ indique le nombre de lettres α présentes dans le préfixe $T^{bwt}_{0,i}$. Depuis la lettre α correspondant à la lettre t_i, nous trouvons l'indice de la lettre dans L correspondant à t_{i-1} avec la formule $C[\alpha] + Occ(\alpha, i)$. Par exemple sur la figure 2.12 dans la matrice de gauche, la lettre `A` à la position 1 est la première occurrence des `A`, la position de la lettre précédente dans l'ordre de T est $C[\texttt{A}] + Occ(\texttt{A}, 1) = 1 + 1 = 2$. La lettre précédente est le `N` à la position 2.

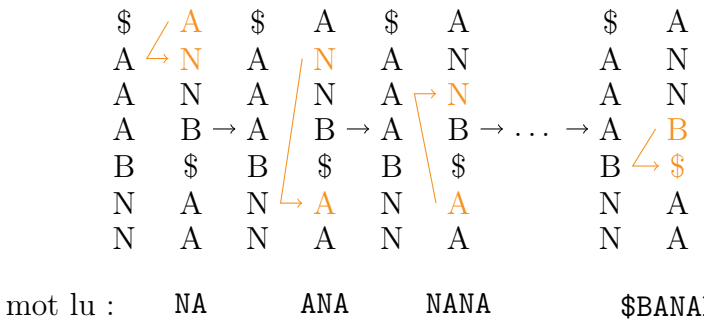


FIGURE 2.12 : Lecture du texte T^{bwt} `ANNB$AA`. La première lettre de T^{bwt} est la dernière lettre de T. Le `A` en première position est le premier `A`, donc sur F nous cherchons le premier `A`. En face de lui se trouve la lettre `N`. Nous avons lu le mot `NA`. Le `N` est le premier `N`, donc nous cherchons le premier `N` de F, qui est face d'un `A` ... Et ainsi de suite jusqu'à trouver le symbole `$`.

Le texte transformé par la BWT est plus facilement compressible que le texte original car des suites de lettres identiques se sont formées. Lorsque nous avons transformé le texte `BANANA$` dans nos exemples (figure 2.11 matrice de droite), les rotations `ANA$BAN` et `ANANA$B` sont placées successivement. Leurs rotations `NA$BANA` et `NANA$BA` commencent de la même manière et sont proches dans M créant une succession de lettres `A` dans L. Le même argument est valable pour tous les mots du texte : toute région sera susceptible de contenir beaucoup de fois peu de lettres différentes.

T^{bwt} peut être stocké en utilisant $nH_k(T) + o(n)$ bits [24], où H_k est l'entropie d'ordre $k \leq \log_\sigma n + \omega(1)$ du texte T de taille n.

Actuellement plusieurs logiciels de compression de documents utilisent la BWT, couplée à un codage de Huffman. Nommée méthode BZIP2 [46], elle est intégrée dans 7zip, Winrar, ... La BWT est aussi utilisée en bioinformatique pour aligner des séquences sur une référence avec la méthode BWA [22].

2.5.2 Le FM-index et ses variantes

Grâce à la recherche inverse, une structure d'indexation, le FM-index et plusieurs de ses variantes ont vu le jour. Ils permettent de compter et localiser des motifs dans le texte indexé. Chaque index est évalué en fonction de sa complexité en taille et en temps d'exécution des fonctions : $Count\_Occ(P)$ qui compte le nombre d'occurrences d'un motif P et $Locate\_Occ(P)$ qui localise les occurrences du motif P dans le texte initial.

Le **FM-index** [11] est la première structure d'indexation à atteindre une taille proche de l'entropie du texte indexé. Elle se construit à partir de la BWT du texte sur laquelle on applique plusieurs encodages : move-to-front, run-length encoding puis un code préfixe. Ferragina et Manzini indiquent que sa taille est de l'ordre de $O(H_k(T)) + o(1)$ et a une borne maximale de $5nH_k(T) + g_k \log_n$.

Cette structure permet de compter le nombre d'occurrences d'un motif de taille p en $O(p)$ (voir Algorithme 1) et de trouver les occ occurrences en $O(p + occ \log^{1+\epsilon} n)$, avec $\epsilon > 0$. Ces requêtes nécessitent l'algorithme de recherche inverse, la table C et la fonction $Occ(\alpha, i)$. Le temps d'exécution de la recherche de motif est directement lié au temps d'exécution de la fonction $Occ(\alpha, i)$. Dans leur article, les auteurs proposent une implémentation de $Occ(\alpha, i)$ en $O(1)$, nécessitant deux tables supplémentaires de taille $O(u/\log u)$ et une de taille $O((u \log u) \log \log u)$.

Algorithm 1 $count\_Occ(P)$: Compte le nombre d'occurrences du motif P de longueur p

```
1 : l = P[p], i = p
2 : sp = C[l] + 1, ep = C[l + 1]
3 : while ((sp ≤ ep) et (i ≥ 2)) do
4 :     l = P[i − 1]
5 :     sp = C[l] + Occ(l, sp − 1) + 1
6 :     ep = C[l] + Occ(l, ep)
7 :     i = i − 1
8 : if ep < sp then
        return 0
9 : else
        return ep − sp + 1
```

Plusieurs implémentations ont été proposées, réduisant la taille de l'index et augmentant la vitesse d'exécution des fonctions de requête.

Une implémentation du FMI a été proposée par Sadakane [44] puis par Ferragina et al. [12] et Mäkinen et Navarro [24] en utilisant un Wavelet Tree. Cette dernière,

nommée **WT-FMI**, permet une exécution de la fonction $Occ(l, i)$ en $O(\log \sigma)$.

Mäkinen et Navarro [23] ont voulu exploiter le grand nombre de lettres identiques consécutives pour pouvoir représenter la BWT de T : Soit T^{bwt} un texte de taille n, il peut être décomposé en n' sous-mots de lettres identiques : $T^{bwt} = l_1^{c_1} l_2^{c_2} \dots l_{n'}^{c_{n'}}$. T peut être représenté par le mot $S = l_1 l_2 \dots l_{n'}$ et le vecteur de bits $B = 10^{l_1-1}10^{l_2-1} \dots 10^{l_{n'}-1}$ Par exemple si $T^{bwt} = aaacbbccc = a^3c^1b^2c^2$, nous avons $S = acbc$ et $B = 1001101000$. Les fonctions rank et select sur S et B nous permettent de retrouver T^{bwt}.

Le **RL-FMI** regroupe les idées précédentes : nous indexons seulement le mot S provenant de T^{bwt} avec un Wavelet Tree. Une optimisation peut être faite en utilisant un Wavelet Tree ayant la forme d'un arbre de Huffman. Le WT a alors une taille de $nH_0(S)$. Les requêtes s'exécutent en $O(H_0(S))$. La taille totale du RL-FMI est de $nH_k \log \sigma(1 + o(1))$ bits.

Le **AF-FMI** [12], puis l'implémentation de Mäkinen et Navarro [25], permettent d'atteindre un index de taille $nH_k + o(n \log \sigma)$ bits. La première implémentation utilise une "boosting compression", la seconde utilise des RRR vecteurs [38]. Leurs fonctions utilisent une table $Occ[s_i, l]$ de taille $\sigma^{k+1} \log n$ bits indiquant le nombre d'occurrences de la lettre l avant le contexte s_i, ainsi qu'un vecteur de bits de taille $O(\sigma^k \log n)$ bits indiquant le début de chaque contexte avec un bit 1.

La table 2.1 résume la taille des index ainsi que les temps d'exécution des fonctions $Count\_Occ()$ et $Locate\_Occ()$ [34].

	FM-Index	WT-FMI	RL-FMI	AF-FMI
Taille totale	$5nH_k + o(n \log \sigma)$	$nH_0 + o(n \log \sigma)$	$nH_k \log \sigma + 2n$ $+o(n \log \sigma)$	$nH_k + o(n \log \sigma)$
$Count\_Occ$	$O(p)$	$O(p \times (1 + \frac{\log \sigma}{\log \log n}))$		
$Locate\_Occ$ par occurrence	$O(\log^{1+\epsilon} n)$	$O(\log^{1+\epsilon} n \frac{\log \sigma}{\log \log n})$		
Conditions	$\sigma = o(\frac{\log n}{\log \log n})$ $k \leq \log_\sigma(\frac{n}{\log n}) - \omega(1)$	$\sigma = o(n)$	$\sigma = o(n)$ $k \leq \log_\sigma n - \omega(1)$	$\sigma = o(n)$ $k \leq \alpha \log_\sigma n$ pour $0 < \alpha < 1$

TABLE 2.1 : Taille et temps de requête du FM-index et de ses variations d'après [34].

2.6 Stocker et requêter les textes annotés

Soit $T = t_0 t_1 ... t_{n-1}$ un texte de longueur n sur un alphabet de taille σ. Le texte peut être composé de plusieurs séquences, chaque séquence finissant par le symbole \$. Une **annotation** est une information ajoutée sur le texte ou une portion du texte. Soit $L = \{L_0, L_1 ... L_{\ell-1}\}$ un ensemble d'annotations. Un **texte annoté** (T, A) est un texte ayant des annotations **non chevauchantes** : une lettre a au maximum une annotation. Chaque position i du texte est annotée par exactement une annotation $a_i \in L \cup \{\varepsilon\}$, où l'annotation ε est placée sur toutes les lettres qui n'avaient pas d'annotation à l'origine. $A = a_0 a_1 ... a_{n-1}$ est la séquence d'annotations (Figure 2.13). Ainsi une séquence représentant une recombinaison V(D)J sera annotée avec les noms des gènes qui la composent, voir la figure 2.14. Chaque nucléotide ne provenant que d'un gène, chaque lettre aura une annotation maximum.

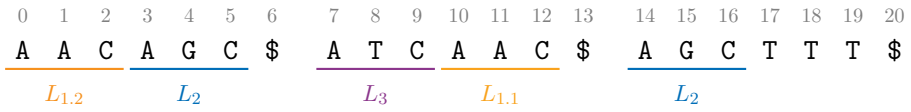


FIGURE 2.13 : Le texte T =AACAGC\$ATCAAC\$AGCTTT\$, composé de trois séquences, est annoté avec la séquence d'annotations $A = L_{1.2}L_{1.2}L_{1.2}L_2L_2L_2\varepsilon L_3L_3L_3L_{1.1}L_{1.1}L_{1.1}\varepsilon L_2L_2L_2\varepsilon\varepsilon\varepsilon\varepsilon$.

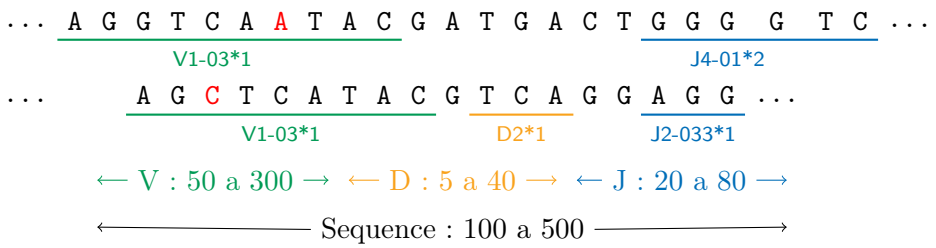


FIGURE 2.14 : Deux recombinaisons V(D)J. La première séquence est une recombinaison VJ avec deux annotations : un gène Vet un gène J.La seconde séquence est une recombinaison VDJ avec trois annotations. Les deux dernières lignes indiquent la longueur des annotations V, D et J ainsi que la longueur des séquences V(D)J.

Nous cherchons à indexer (T, A) tout en permettant des accès rapides aux données (parmi les requêtes de la section 1.3). Les structures présentées dans les sections précédentes ne peuvent être utilisées simplement car elles n'indexent que les textes à une seule dimension : ici soit le texte, soit les annotations. Bien que certaines structures

permettent des requêtes bidimensionnelles [26], aucune ne permet toutes les requêtes voulues. Nous allons donc modifier une structure existante ou créer une nouvelle structure pour compresser et indexer notre texte annoté.

Nous présentons dans le chapitre 3 une manière de représenter le texte T en utilisant des motifs annotés. T peut être annoté ou non, dans le dernier cas, nous ajoutons nous-même les annotations. Le chapitre 4 introduit deux index compressés utilisant une transformée de Burrows-Wheeler ainsi qu'un Wavelet Tree pour le texte annoté (T, A).

Chapitre 3

Compression de séquences annotées par référence

Pour représenter une recombinaison V(D)J, nous pouvons décrire les gènes dont elle est constituée. Par exemple, une séquence peut être constituée du gène V_1 moins les 2 dernières lettres, puis du facteur `GTATT`, puis du gène J_3 ayant une substitution à la cinquième lettre pour la lettre A. Plutôt que de décrire la séquence lettre par lettre, nous utilisons ici des références vers les gènes utilisés, des séquences déjà connues.

Cette manière de procéder est utilisée dans la compression LZ (voir la section 2.2), ainsi que dans les indexations l'utilisant. Dans notre cas, nous voulons que le découpage signifie quelque chose dans notre texte. Ainsi, là où les indexations LZ utilisent des références internes au texte, chaque facteur de texte est déjà présent dans le texte déjà lu, nous utilisons des références externes au texte en utilisant un ensemble de motifs annotés.

Ce chapitre répond à deux problèmes : *comment compresser un texte T non annoté tout en lui donnant les annotations les plus probables* (section 3.2.3), et *comment compresser un texte (T, A) annoté* (section 3.2.2).

3.1 Boites

Soient $L = \{L_0, L_1 ... L_{\ell-1}\}$ un ensemble d'annotations et $M = \{M_0, M_1, \ldots, M_{l-1}\}$, un ensemble de motifs annotés avec L. Chaque motif M_i est lié à une annotation L_i étant donné que l'annotation de chaque lettre de M_i est L_i. Pour plus de clarté, chaque

motif $M_u = (m_0 m_1 \dots m_{m-1}, L_u L_u \dots L_u)$ sera décrit $M_u = m_0 m_1 \dots m_{m-1}$ en sous-entendant que chaque annotation de M_u est L_u. Ces motifs nous permettent d'annoter le texte de la manière suivante : soit le texte annoté (T, A) et le motif annoté $M_u = m_0 m_1 \dots m_{m-1}$. Si $t_i t_{i+1} \dots t_{i+j}$, annoté $a_i a_{i+1} \dots a_{i+j}$, est représenté par M_u aux indices k à $k + j$, alors $t_i = m_k, \dots, t_{i+j} = m_{k+j}$ et $a_i = a_{i+1} = \dots, = a_{i+j} = L_u$ à quelques variations près où $t_{i+x} \neq m_x$ et $a_{i+x} = L_u$. Sur la figure 3.1, la boite B_2 représente le motif J_3 avec une modification : la cinquième lettre du motif est substituée avec la lettre `A` dans le texte.

Soit $T_{i,j} = t_i t_{i+1} \dots t_j$ un facteur du texte (T, A) annoté avec les annotations du motif M_u. Une **boite** B_k est un ensemble d'éléments décrivant $T_{i,j}$ en faisant référence à M_u. On appelle $\varphi(B_k)$, la fonction d'interprétation des boites telle que $\varphi(B_k) = T_{i,j}$. Nous voulons représenter T par un ensemble de boites et de lettres $B = B_0 t_u B_1 t_v \dots t_w B_{b-1}$ tel que $T = \varphi(B) = \varphi(B_0) t_u \varphi(B_1) t_v \dots t_w \varphi(B_{b-1})$.

Prenons l'exemple de l'introduction, représenté dans la figure 3.1. Nous pouvons utiliser des boites pour décrire les facteurs provenant des motifs V_1 et J_3, ainsi que la séquence centrale. La section suivante décrit formellement ces boites.

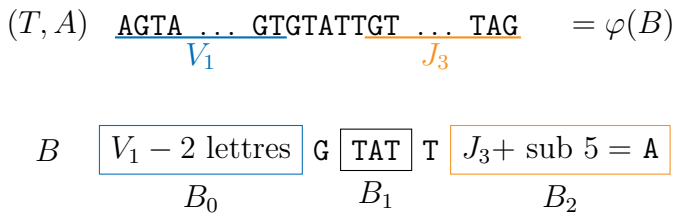


FIGURE 3.1 : La séquence est constituée du gène V_1 moins les 2 dernières lettres, du facteur `GTATT`, puis du gène J_3 ayant une substitution à la cinquième lettre pour la lettre A. Des boites décrivent tous ces facteurs.

Les **éléments** d'une boite sont les données nécessaires pour retrouver la sous-séquence que décrit cette boite. L'élément e_1 de la boite B_i est accessible par $B_i.e_1$. La fonction $\lambda(B_i)$ indique le nombre de bits nécessaires pour coder la boite B_i, $\lambda(B_i.e_1)$ indique le nombre de bits codant l'élément e_1 de la boite B_i. La taille d'une boite B_i est le nombre de lettres décrites par celle-ci. Elle est l'un des éléments de la boite et est accessible par $B_i.t$.

Pour un texte annoté (T, A), un emboitage B, utilisant M, **compatible** avec (T, A) indique que le texte T est représenté par un emboitage $B = B_0 t_u B_1 t_v \dots t_w B_{b-1}$, en utilisant comme séquences références les séquences de M. Lorsque une boite B_l annote $T_{i,j}$ avec le motif M_k, chaque lettre de $T_{i,j}$ est annotée L_k dans (T, A). De cette

manière nous pouvons voir facilement les motifs qui composent un texte : par exemple les gènes qui composent une recombinaison V(D)J.

Nous souhaitons réaliser l'emboitage B de T utilisant le moins de mémoire possible.

Les différentes boites : principe

Les principales boites sont des références vers des motifs existants. Les données réelles que nous utilisons sont des séquences ADN issues de séquençage. Suivant la méthode de séquençage, les deux sens de lectures sont possibles. L'un des éléments d'une boite est le sens de lecture du motif.

Lorsqu'il y a des variations dans le motif, nous ajoutons les informations des modifications à apporter à la sous-séquence représentée. Ces informations étant coûteuses en mémoire, nous décidons de faire deux types de boites : une boite sans variation dans le motif : **boite normale**, **BN** (comme la boite B_0 dans la figure 3.1), ou avec variations dans le motif : **boite à erreurs**, **BE** (comme la boite B_2 dans la figure 3.1).

Lorsque deux boites d'une même séquence décrivent une sous-séquence appartenant au même motif, il est plus efficace que l'une d'elles fasse référence à l'autre, lui permettant d'utiliser ses données, nous introduisons la **boite référence**, **BR**.

Si nous décrivons chaque motif apparaissant dans l'ensemble M, comment décrire le reste de la séquence, n'apparaissant pas dans l'ensemble M ? Les facteurs de séquences se trouvant entre deux motifs sont plus efficacement décrits en toute lettres, dans une **boite explicite**, **BX** (comme la boite B_1 dans la figure 3.1). Nous agissons de la même manière lorsque le motif est petit (inférieur à un seuil défini section 3.2.4).

Les différentes boites : définitions

Nous proposons donc quatre types de boites, avec leur fonction d'interprétation $\varphi()$ et leur taille :

Une **boite normale** est définie par $BN = (i, s, d, t)$, avec i étant l'identifiant du motif, s, son sens de lecture, d, la position de début de la boite sur le motif et t, la longueur de la séquence représentée. Soient le motif $M_i = m_0 m_1 \dots m_{m-1}$ et $sens(s, T)$, la fonction permettant de lire le texte T dans le sens s, alors $\varphi(BN) = sens(s, (m_d, \dots, m_{d+t-1}))$. (Voir figure 3.2.)

$\lambda(BN)$ est égal à la somme des tailles en bits des éléments d'une BN. $\lambda(i) = \log l$,

avec l le nombre de motifs de M. $\lambda(s) = 1$ pour les deux sens de lecture, $\lambda(d) = \lambda(t) = \log m$, avec m la taille maximum d'un motif de M. $\lambda(BN) = \log l + 2\log m + 1$.

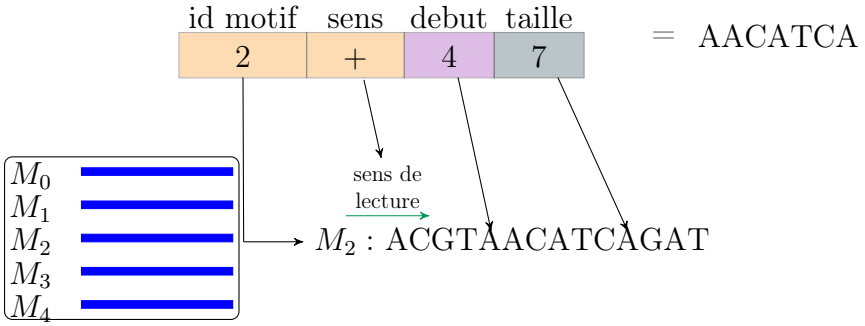


FIGURE 3.2 : Une boite normale $BN(2, +, 4, 7)$ codant la séquence AACATCA. Elle provient du motif M_2 que l'on lit de gauche à droite en commençant à la position 4, et est de taille 7.

Une **boite à erreurs** est définie par $BE = (i, s, d, t, n, P)$, avec P étant une liste de triplets (p, y, l) de taille n. Les quatre premiers éléments d'une boite à erreurs sont similaires à ceux d'une boite normale et sont décodés de la même manière. Les éléments de la liste P représentent les variations qu'a subi la séquence : la lettre de position p dans la séquence que nous venons de trouver a subi la modification y (insertion ("ins"), délétion ("del") ou substitution ("sub") d'une lettre), l est la lettre insérée ou substituée si tel est le cas. Soit $erreurs(P, T)$ la fonction, qui pour chaque triplet (p, y, l) de la liste P, applique la modification y par la lettre l à la position p de T. (Voir la figure 3.3.)

Soit le motif $M_i = m_0, m_1, \ldots, m_{m-1}$, $\varphi(BE) = erreurs(P, \varphi(BN(i, s, d, t))) = erreurs(P, sens(s, (m_d, m_{d+1}, \ldots, m_{d+t})))$.

$\lambda(n) = \log e$ avec e le nombre maximum d'erreurs trouvées dans une BE. $\lambda(p) = \log m$, avec m la taille maximum d'un motif. La taille de stockage de y et l peut être optimisée en utilisant un bit pour décrire le type d'erreur : $\lambda(y) + \lambda(l) = 1 + \log \sigma$, avec par exemple 0 indiquant une insertion et 1 indiquant une délétion ou une substitution. Dans le dernier cas, le type d'erreur est déterminé avec la lettre l : si c'est la lettre présente dans le motif, alors il s'agit d'une délétion, sinon c'est une substitution.

$\lambda(BE) = \lambda(BN) + \log e + w * r$ avec $w = \log m + \log \sigma + 1$ étant le nombre de bits utilisés pour coder une erreur, r le nombre d'erreurs.

Une **boite référence**, figure 3.4, nécessite un pointeur vers la boite référée, une BN, une BE ou encore une autre BR, permettant d'utiliser ces données : le motif de l'ensemble M utilisé ainsi que son sens de lecture. Car si deux boites d'une

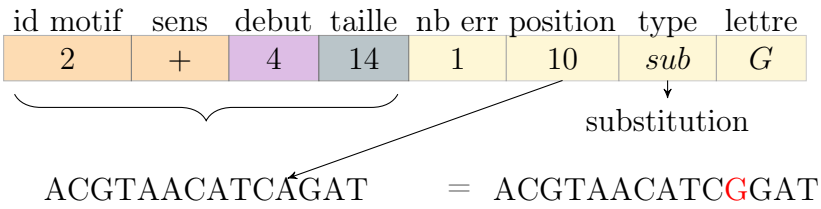


FIGURE 3.3 : Une boite à erreurs $BE(1, +, 4, 14, 1, \{(10, "sub", G)\})$ codant la séquence ACGTAACATCGGAT en ajoutant une variation : la substitution de la onzième lettre par la lettre G.

même séquence font référence au même motif, il est très probable que celui-ci soit présent dans le même sens. Un élément indique la position de début de la séquence sur le motif. Elle est définie par $BR = (r, d, t)$, avec r étant le nombre de boites entre la BR et la boite référée. La valeur est positive si la boite référée se trouve à gauche de la boite actuelle, négative sinon. Soient $B = B_0 t_u B_1 t_v \dots t_w B_{b-1}$ un emboitage, B_j la BR actuelle et le motif $M_i = m_0 m_1 \dots m_{m-1}$. La fonction d'interprétation utilise celle de la boite référée : si la boite référée est une BN alors $\varphi(BR) = \varphi(BN(B_{j-r}.i, B_{j-r}.s, d, t)) = sens(B_{j-r}.s, (m_d, \dots, m_{d+t}))$. Si la boite référée est une BE alors $\varphi(BR) = erreurs(B_{j-r}.P, \varphi(BN(B_{j-r}.i, B_{j-r}.s, d, t)))$
$= erreurs(B_{j-r}.P, sens(B_{j-r}.s, (m_d, m_{d+1}, \dots, m_{d+t})))$.
Sinon, la boite référée est une BR et $\varphi(BR) = \varphi(BR(j - r, d, t))$.

$\lambda(BR)$ est égal à la somme des tailles en bits des éléments d'une BR. $\lambda(r) = \log b$, avec b le nombre total de boites pour T. $\lambda(d) = \lambda(t) = \log m$, avec m la taille maximum d'un motif de M. $\lambda(BR) = \log b + 2 \log m$.

Les BR ne sont incluses dans les algorithmes d'emboitage que si il est rentable de les utiliser, donc si $\lambda(r) < \lambda(i) + 1$. Elle peuvent ne pas être utilisées si le texte est très long et la distance entre une BR et sa boite référée est grande. Nous pouvons aussi poser une borne sur la valeur de r.

Enfin, une **boite explicite** est définie par $BX = (t, O)$, avec O étant la liste de longueur t des lettres qu'elle représente (voir la figure 3.5).

Nous bornons la longueur de la séquence représentée par une BX par $\lambda(BN)/\log\sigma$ pour qu'elle soit toujours plus petite que la plus petite BN. Dès lors : $\varphi(BX) = O$ et $\lambda(BX) = p + \log\sigma \times t$, avec $p = \log(\lambda(BN)/\log\sigma)$ le nombre de bits utilisés pour représenter t.

Les informations stockées dans chaque boite sont décrites dans la figure 3.6.

Lors d'un emboitage $B = B_0 t_u B_1 t_v \dots t_w B_{b-1}$, chaque paire de boites successives

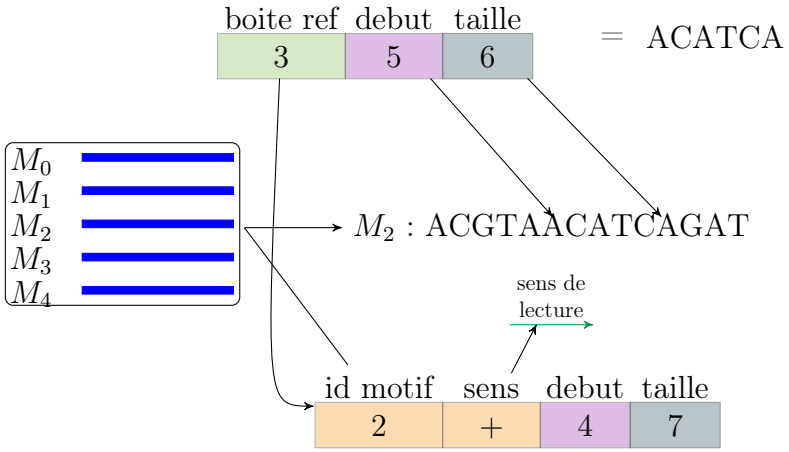


FIGURE 3.4 : Une boite ref $BR(3, 5, 6)$ codant la sous-séquence ACATCA. Nous utilisons des éléments d'une boite normale de la même séquence, se trouvant deux boites avant la boite actuelle, cela nous permet d'avoir accès à l'identifiant de la séquence ainsi qu'au sens de lecture.

FIGURE 3.5 : Une boite explicite $BX(4, \{G, A, T, A\})$ codant la séquence GATA.

B_iB_{i+1} est séparée par une lettre t_x. Le facteur $\varphi(B_i)t_x$ n'existe pas dans l'ensemble M. L'une des raisons possibles est qu'une erreur se soit insérée à la position x. Le fait d'inclure une lettre entre chaque boite nous permet de commencer la boite suivante après l'erreur.

Idéalement, nous souhaitons qu'une séquence soit représentée par autant de BN ou de BE qu'il y a de motifs dans la séquence, chacun séparé par une BX, comme présenté dans la figure 3.7. Mais dans certains cas, par exemple lorsqu'un facteur tiré d'un motif contient beaucoup de variations, il serait moins coûteux d'utiliser une BX au niveau de la zone de forte densité de variation. Nous verrons dans la section suivante les conditions de sélection d'une boite dans différents algorithmes d'emboitage.

3.2 Algorithmes d'emboitage

Un algorithme d'emboitage fournit l'emboitage d'un texte annoté, ou l'annotation et l'emboitage d'un texte non annoté. Nous appelons **optimal** un emboitage dont la taille est minimum. Nous définissons plusieurs algorithmes d'emboitage dans les sections suivantes. Ils répondent à différents problèmes :

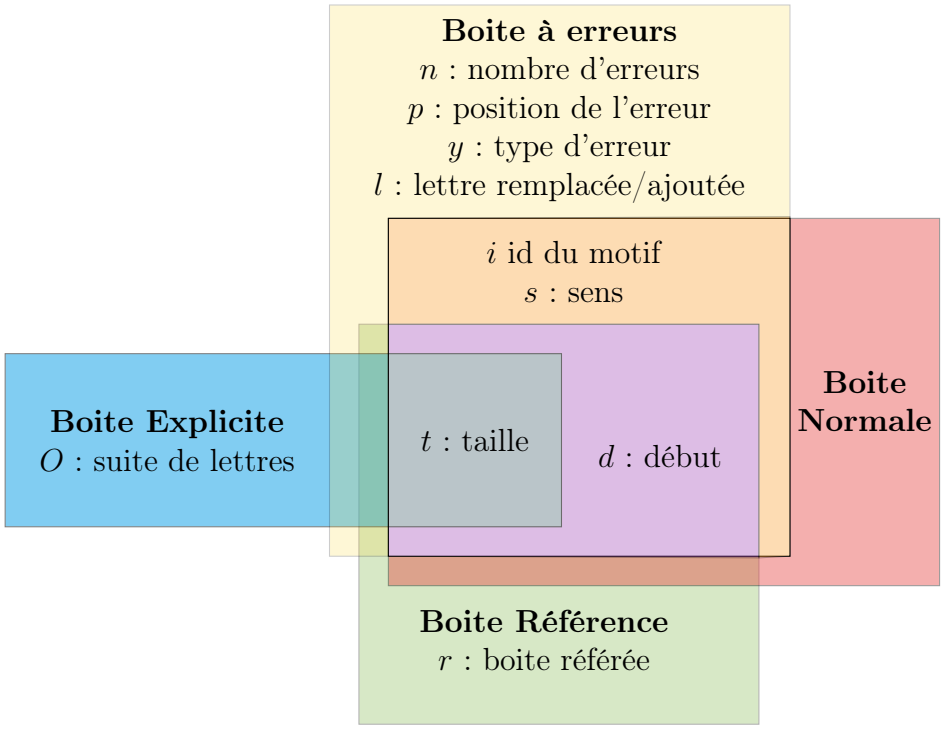


FIGURE 3.6 : Liste des éléments présent dans chaque type de boite.

FIGURE 3.7 : Un texte décrit par trois BX, une BN et une BE contenant une variation.

— Comment emboiter un texte **annoté** de taille n : une programmation dynamique fournit, dans la section 3.2.2, un emboitage non optimal sur le texte mais optimal sur un facteur annoté, avec une complexité de $O(e^4n' + m^2)$, avec e le nombre maximum d'erreurs sur un facteur, n' le nombre de facteurs et m la taille d'un motif de M.
— Comment emboiter un texte **non annoté** : une programmation dynamique fournit un emboitage optimal dans la section 3.2.3.1. Il s'exécute avec une complexité de $O(n^3b)$, avec b le nombre de boites.
— Comment emboiter un texte **non annoté** : cet algorithme glouton emboite de gauche à droite le texte puis optimise chaque boite dans la section 3.2.3.2. L'emboitage final n'est pas optimal mais se calcule avec une complexité de $O(nb \log b)$, avec b le nombre de boites de l'emboitage glouton.

L'emboitage d'un texte annoté proposera, sur les positions annotées L_β, des boites

dont l'identifiant d'annotation sera toujours L_β. L'emboitage d'un texte non annoté proposera des boites issues de tous les motifs possibles et permettra une meilleure compression du texte. Voir figure 3.8.

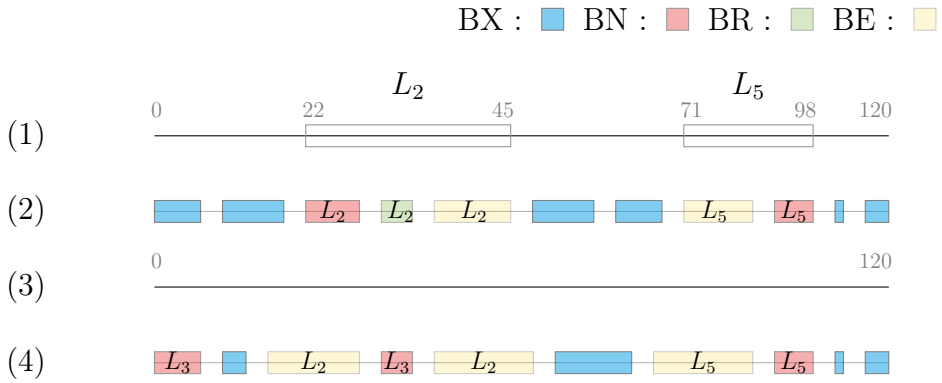


FIGURE 3.8 : (1) Une séquence annotée avec l'annotation L_2 sur les positions 22 à 45 et l'annotation L_5 sur les positions 71 à 98. (2) L'emboitage sur séquence annotée produit des boites dont l'identifiant d'annotation sera L_2 entre les positions 22 à 45, et L_5 sur les positions 71 à 98. Toutes les boites sur des positions non annotées sont des BX. (3) Une séquence non annotée. (4) L'emboitage d'une séquence non annotée produit des boites dont les identifiants d'annotations sont variés, mais les boites peuvent être plus grandes et produire une meilleure compression.

Notons que si le texte T est composé de plusieurs séquences, alors l'emboitage de T se fait séquence après séquence en utilisant le même codage tout le long du texte. Cela permet une décompression plus aisée et une sauvegarde unique du codage.

3.2.1 Opérations sur les emboitages

Nous utilisons plusieurs opérations sur les emboitages dans les sections suivantes. Soient $B = B_0 t_u B_1 \dots B_b$ et $B' = B'_0 t'_u B'_1 \dots B'_{b'}$, deux emboitages ainsi que α une lettre, nous définissons les trois opérateurs :

— la taille en bit de l'emboitage B : $\lambda(B) = \lambda(B_0) + \log\sigma + \lambda(B_1) + \ldots + \lambda(B_b)$,

— la concaténation de deux emboitages et d'une lettre :
$B \oplus \alpha \oplus B' = B_0 t_u B_1 \dots B_b \alpha B'_0 t' u B'_1 \dots B'_{b'}$,

— l'emboitage utilisant le moins de mémoire : $\min(B, B') = \begin{cases} B \text{ si } \lambda(B) \leq \lambda(B') \\ B' \text{ sinon} \end{cases}$

L'opérateur min() sera utilisé pour garder le meilleur emboitage parmi plusieurs concaténations possibles.

La transformation $R_B(B')$ modifie les boites de l'emboitage B' en fonction des boites de B, (voir la figure 3.9 (1)). Pour toute BN (ou BR ou BE) de l'emboitage B, la transformation $R_B(B')$ transforme toutes les BN de B' ayant le même identifiant et sens de lecture du motif en BR pointant vers la BN (ou BR ou BE) de B. La transformation se produit en temps $O(b \log b')$ en triant les boites de B'.

La simplification de la transformation R, $R^0_B(B')$, transforme chaque BN de B' en BR si B possède une BN, dans le cas où l'on connaît déjà les annotations (utilisation dans l'algorithme 3.2.2). Cette transformée est utilisée si toutes les boites d'un emboitage proviennent du même motif de M. Cette transformation se fait en temps $O(b + b')$.

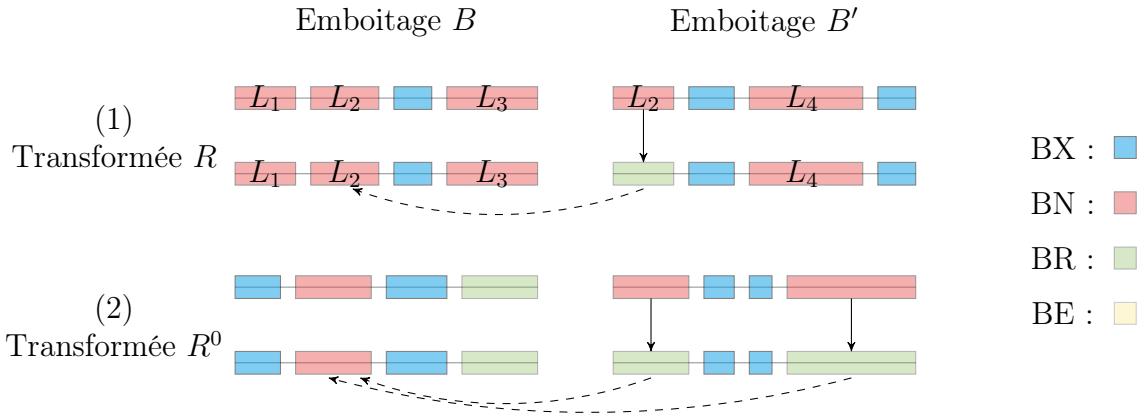


FIGURE 3.9 : (1) La transformée $R_B(B')$ transforme la première BN de B' car il y a une BN ayant l'identifiant L_2 dans B, mais il n'y a pas de BN ayant l'identifiant L_4 dans B donc la seconde BN de B' n'est pas transformée en BR. (2) La transformée $R^0_B(B')$, transforme les BN de B' en BR car il y a une BN dans B.

3.2.2 Emboitage d'un texte annoté

Soit (T, A) un texte annoté avec les motifs de M, quel est le meilleur emboitage $B = B_0 t_u B_1 t_v \dots t_w B_{b-1}$, compatible avec (T, A) et utilisant M, tel que B utilise le moins de mémoire possible ?

Nous examinons chaque facteur annoté du texte indépendamment. Soit $T_{i,j}$ une sous-séquence maximale de T annotée avec une même annotation L_β, avec t_{i-1} et t_{j+1} non annotés L_β. Nous allons emboiter chaque facteur annoté puis chaque facteur non annoté. L'emboitage du texte produit pourrait ne pas être optimal sur l'ensemble de la séquence, mais l'est sur chaque facteur.

Découpage d'un facteur annoté en fonction des variations

Soit $T_{i,j}$ un facteur dont chaque lettre est annotée L_β, quel est le meilleur emboitage $B = B_0 t_u B_1 t_v \dots t_w B_{b-1}$, compatible avec $(T_{i,j}, A)$ et utilisant M_β, tel que B utilise le moins de mémoire possible ?

À partir d'un facteur annoté, nous commençons par trouver les variations entre le facteur et le motif qui a permis son annotation. Nous utilisons un alignement par programmation dynamique entre $T_{i,j}$ et M_β afin de trouver toutes les variations entre eux, le sens de lecture s de M_β ainsi que la position de départ d à laquelle t_i s'aligne. L'alignement se fait en temps $O((j-i) \times m)$, avec m la longueur de M_β.

Soient $E = e_0, e_1, \dots e_{e+1}$, avec $e_0 = i - 1$, $e_{e+1} = j + 1$ et $e_1, \dots e_e$ les positions dans T des e variations trouvées entre $T_{i,j}$ et M_k, illustrées dans la figure 3.10. Nous avons $i - 1 = e_0 < e_1 < e_2 < \dots < e_e < e_{e+1} = j + 1$. Lors de la substitution ou de l'insertion d'une lettre à la position t_i de T, la variation se trouve à la position i. Si il y eu une délétion de la lettre se trouvant entre t_i et t_{i+1}, alors la variation se trouvera à la position $i + 1$.

Soit $r_u = e_{u+1} - e_u - 1$. Nous avons r_0, le nombre de lettres dans T entre t_{i-1} et la première variation, r_e est le nombre de lettres entre la dernière variation et t_{j+1}, et $\forall u \in [1, e]$, r_u est le nombre de positions entre la variation u et la variation $u + 1$.

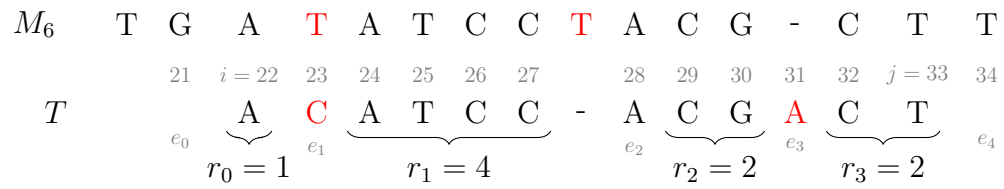


FIGURE 3.10 : Un alignement entre le motif M_6 et la sous-séquence $T_{22,33}$ de T comportant 3 variations. Une substitution se trouve à la position 23 de T, $e_1 = 23$. Une délétion a été faite entre les positions 27 et 28 de T, e_2 doit se trouver sur une position du texte, nous le plaçons après la variation, $e_2 = 28$. Une insertion a été faite à la position 31 de T, $e_3 = 31$.

Emboitage optimal d'un facteur annoté

Chaque portion de texte se trouvant entre deux variations consécutives est représentée par une BX ou une BN. Nous utilisons une programmation dynamique pour combiner ou modifier des boites encodant des portions du texte de plus en plus grandes, afin de trouver le meilleur emboitage du facteur annoté.

Soit S une table de taille $(e+2)^2$, avec $e \leq (j-i+1)$, dont les lignes et colonnes représentent les valeurs de E. Chaque case $S[k,l] = B$ indique un des meilleurs emboitages B de T_{e_k+1,e_l-1}. Nous remplissons cette table par une programmation dynamique, diagonale par diagonale, avec un algorithme qui a des similarités avec l'algorithme de repliement de structure secondaire d'ARN proposé par Nussinov [36]. Le meilleur emboitage de $T_{i,j}$ se trouve dans la case $S[0, e+1]$.

Nous commençons par analyser les facteurs du texte se trouvant entre deux variations, ils sont sur une diagonale de la table S. Nous formons alors une BX ou une BN en fonction de la taille du facteur.

$$\forall i \in [0,q] : S[i,i+1] = \min \begin{cases} BX(e_{i+1} - e_i - 1, \{t_{e_i+1}, t_{e_i+2}, \ldots, t_{e_{i+1}-1}\}) \\ BN(\beta, s, d + e_i, e_{i+1} - e_i - 1) \end{cases}$$

La taille minimum d'une BN est nommée Z_G. Il est utilisé dans les trois algorithmes d'emboitage et est optimisé dans la section 3.2.4.

Nous calculons ensuite diagonale par diagonale avec :

$\forall i, j$ tels que $1 < i+1 < j \leq e+1$:

$$S[i,j] = \min \begin{cases} \min_{k \in [i+1, j-1]}(S[i,k] \oplus t_{e_k} \oplus R^0_{S[i,k]}(S[k,j])), \ (1) : \text{concaténation de deux cases} \\ BE(\beta, s, d + e_i, e_j - e_i - 1, j - i - 1, P) \ (2) : \text{création d'une BE} \end{cases}$$

La figure 3.11 schématise les différents emboitages possibles calculés dans une case.

(1) Meilleure concaténation de deux cases : Nous utilisons la transformation $R^0_{S[i,k]}(S[k,j])$, en $O(e)$ car seul le motif M_β est utilisé. Nous gardons la concaténation $S[i,k] \oplus t_{e_k} \oplus R^0_{S[i,k]}(S[k,j])$, avec $k \in [i+1, j-1]$, utilisant le moins de mémoire, en $O(e^2)$ pour l'ensemble.

(2) Création d'une BE recouvrant toute la case : $S[i,j] = BE(\beta, s, d+e_i, e_j - e_i - 1, j-i-1, P)$, avec P étant les variations de la boite (variations e_u calculées lors de l'alignement).

Les boites du meilleur emboitage $B = B_0 t_u B_1 t_v \ldots t_w B_{b-1}$ de $T_{i,j}$ du facteur sont décrites dans la case $S[0, e+1]$. La programmation dynamique s'exécute avec une complexité de $O(e^4)$.

Emboitage de la séquence complète

Nous utilisons la programmation dynamique pour chaque facteur annoté du texte. Les facteurs dont l'annotation est ε sont représentés par des BX. L'emboitage du texte n'est pas optimal dans certains cas extrêmes, par exemple lorsque deux facteurs sont

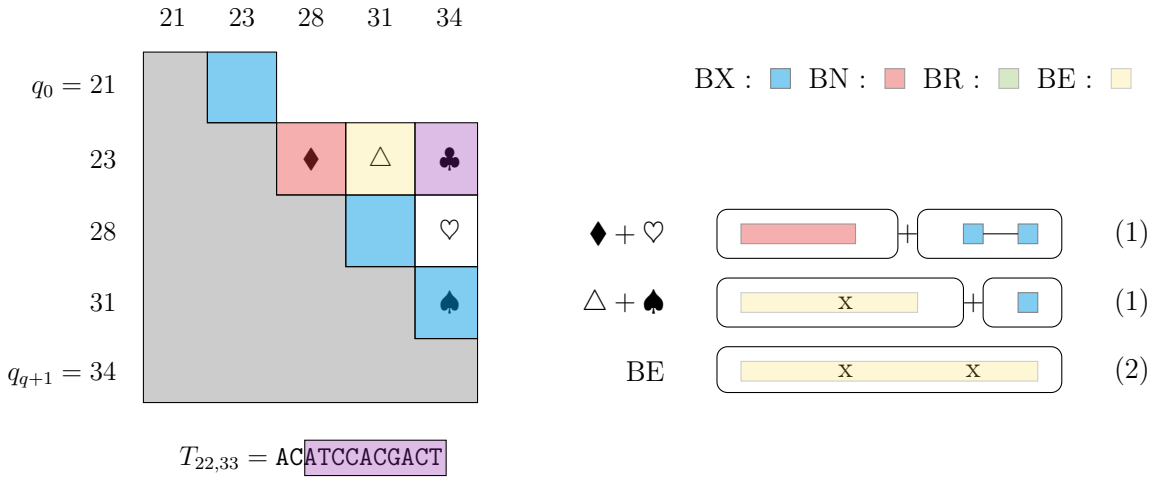


FIGURE 3.11 : Emboitage du facteur $T_{22,33}$ ayant 3 variations avec son motif référence. (Gauche) Sur la première diagonale, le seuil de $Z_G = 4$ a été mis, ainsi la boite représentant $T_{23,28}$ est une GB et celle représentant $T_{21,23}$ est une PB. La case ♣ couvre le facteur entre les positions 23 et 34 (exclues). (Droite) Elle est calculée par le score minimum entre : la concaténation des boites des cases ♦ et ♡ ou la concaténation des boites des cases △ et ♠ ou la formation d'une boite à erreurs.

annotés de la même manière et séparés par une lettre.

Dans le pire des cas, le texte est annoté intégralement avec la même annotation et il y a une variation à chaque position, donc $e = n$. La complexité totale de l'emboitage est alors $O(n^4 + n^2)$. Mais dans le cas général, e est considérablement plus petit que n et l'emboitage a une complexité de $O(e^4 \times n' + m^2)$, où e est le nombre maximum de variations dans un facteur, n' le nombre de facteurs annotés ($en' \leq n$) et m la taille maximale d'un motif de M.

3.2.3 Emboitage d'un texte non annoté

Soit T un texte de taille n non annoté et $M = \{M_0, M_1, \dots M_{m-1}\}$, un ensemble de motifs annotés avec L. Quel est le meilleur emboitage $B = B_0 t_u B_1 t_v \dots t_w B_{b-1}$ de T compatible avec M, tel que B utilise le moins de mémoire possible ?

Cet emboitage crée des boites représentant des facteurs appartenant à plusieurs motifs de M, comme dans la figure 3.8 (cas 4).

Arbre des suffixes N

Commençons par définir l'arbre des suffixes N, qui va être utilisé pour former des boites les plus grandes possibles. Soit N, un arbre des suffixes, voir section 2.4.1, créé à partir de M, un ensemble de motifs annotés. N reconnaît tous les suffixes des séquences de M. Lorsque nous traitons l'ADN, nous ajoutons tous les suffixes des complémentaires inverses des séquences de M.

Un nœud interne de N est défini par $q = (c, \mathit{fils})$ avec c le mot reconnu par le nœud, et la fonction $\mathit{fils}(q, \alpha)$ qui retourne le nœud fils de q par la transition de lettre α si elle existe, sinon retourne $\perp$. Un chemin dans N est une suite de nœuds, allant d'un nœud, à l'un de ses descendants. Chaque mot reconnu par N est représenté par un chemin allant de la racine à une feuille. Chaque feuille indique l'identifiant du ou des motifs dont le chemin est un facteur.

Soit $\mathit{nœud}(c)$, la fonction qui retourne le nœud qui reconnaît la séquence c ou retourne $\perp$ si le chemin n'existe pas. La fonction $\mathit{feuilles}(q)$ retourne la liste des feuilles descendant du nœud q. $feuilles(q)[i]$ permet d'accéder au triplet (i, s, d) de la $i$ème feuille descendant du nœud q.

Étant donné deux nœuds q_0 et q_1 et un caractère β, la fonction $correspond(q_0, \beta, q_1)$ évalue si l'un des nœuds $q = \mathit{nœud}(q_0.c\ \alpha\ q_1.c)$ (substitution), $q = \mathit{nœud}(q_0.c\ \ q_1.c)$ (insertion) ou $q = \mathit{nœud}(q_0.c\ \alpha\beta\ q_1.c)$ (délétion) avec $\alpha \in \Sigma$, existe. Cette fonction retourne $(q, \alpha, \mathit{err})$, avec q, l'un des nœuds trouvé s'il existe, $\perp$ sinon, la lettre α qui permet de trouver le noeud et err le type d'erreur, avec $\mathit{err} = \mathit{ins}$, del ou sub si un noeud a été trouvé, nul sinon. Cette fonction permet de trouver le noeud dont le chemin correspond à une BE.

Pseudo-emboitage

Lors des deux algorithmes suivants, nous allons former des emboitages dont les boites peuvent être modifiées à chaque étape. Nous voulons un moyen de ne pas fixer trop rapidement les boites, au risque de manquer des modifications possibles. Par exemple si nous avons une BN représentant un facteur commun à plusieurs motifs, choisir l'un des motifs de M risque d'empêcher la formation d'une BR faisant référence à la BN actuelle si on avait choisi un autre motif. Nous utilisons, pour chaque boite, un pointeur vers un nœud de N. Le chemin de la racine de N à ce pointeur représente le facteur stocké par la boite. Les feuilles descendant de ce nœud contiennent les valeurs possibles pour les identifiants du motif, sens de lecture et début sur le motif. Ainsi nous

assignerons des valeurs temporaires $(0, 0, 0)$ dans la boite et nous référerons au nœud pour connaître les valeurs possibles. La traduction en emboitage assignera les valeurs finales des éléments des boites.

Soit (B_i, q_i) une pseudo-boite, avec $q_i = noeud(\varphi(B_i))$ un pointeur vers l'arbre N, le pseudo-emboitage de $T_{i,j}$ est défini par $B' = (B_0, q_0)t_u(B_1, q_1)t_v \dots t_w(B_{b-1}, q_{b-1})$

Nous utilisons les mêmes opérations sur les emboitages que celles décrites dans la section précédente avec une variation sur la fonction λ (exprimant la taille en bit d'une boite). Lorsqu'une opération sur les boites n'est pas possible, alors nous formons une boite, non BX, dont le pointeur q est nul, la taille de la boite est alors infiniment grande. Soit (B, q), si $q = \perp$ et B n'est pas une BX, alors la boite est invalide et $\lambda(B) = +\infty$. Une telle taille nous permet d'éliminer les emboitages impossibles grâce à la fonction min.

3.2.3.1 Emboitage optimal par programmation dynamique

Soit S une table de taille $n \times n$ dont les lignes et les colonnes ont pour indice 0 à $n - 1$. La case $S[i, j] = B$ indique l'un des meilleurs pseudo-emboitages de $T_{i,j}$.

Nous remplissons cette table avec une programmation dynamique. Nous commençons par initialiser la table en formant des BX sur tous les facteurs de taille 1. Puis nous examinerons des facteurs de plus en plus grands en modifiant les BX en BN, puis en BR ou BE. Les pointeurs sur l'arbre N seront nuls lorsqu'ils accompagnent une BX (au cas où le facteur ne serait pas présent dans l'arbre), mais seront définis dès qu'il s'agira d'un autre type de boite. L'emboitage du texte T est obtenu par la transformation du pseudo-emboitage de la case $S[0, n - 1]$.

Nous initialisons la table en créant une BX pour chaque facteur de taille 1 :

$S[i, i] = (BX(1, t_i), \perp)$

Nous calculons ensuite diagonale par diagonale avec :

$$S[i,j] = \min \begin{cases} D(S[i, j-1]) \text{ élargissement de l'emboitage vers la droite (1)} \\ \min_{k \in [i+1, j-1]}(F(S[i, k-1] \oplus t_k \oplus R_{S[i,k-1]}(S[k+1, j]))) \\ \text{meilleure concaténation de deux cases (2)} \end{cases}$$

La figure 3.12 représente les emboitages possibles dans une case.

(1) Soit $D(S[i, j-1])$ la transformation de $S[i, j-1]$ étendant sa dernière boite d'une lettre vers la droite. Soit $S[i, j-1] = (B_0, q_0)t_u \dots t_w(B_k, q_k)$. Nous utilisons la pseudo-boite (B_k, q_k) pour créer la nouvelle pseudo-boite (B'_k, q'_k). $D(S[i, j-1]) = (B_0, q_0)t_u \dots t_w(B'_k, q'_k)$ Nous avons $t' = j - w$, la taille du facteur $T_{w+1,j}$. Soit $q =$

$fils(q_k, t_j)$, le nœud représentant $T_{w+1,j}$, s'il existe dans N, sinon $\perp$.

- Si $t' < Z_G$, alors nous formons une BX, $(B'_k, q'_k) = (BX(t', T_{i,j}), \perp)$.
- Si $q = \perp$, alors on ne peut pas augmenter la boite : $(B'_k, q'_k) = (BN(0, 0, 0, t'), \perp)$.
- Sinon, si B_k est une BX, $(B'_k, q'_k) = (BN(0, 0, 0, t'), q)$.
- Sinon, si B_k est une BR faisant référence à la boite (B_l, q_l), si une feuille de q et une feuille de q_l ont les mêmes identifiant et sens de lecture du motif, alors $(B'_k, q'_k) = (B_k, q)$, sinon $(B'_k, q'_k) = (BN(0, 0, 0, t'), \perp)$.
- Sinon, B_k est une BE, $(B'_k, q'_k) = (B_k, q)$.

(2) Pour chaque couple de cases $S[i, k-1]$ et $S[k+1, j]$, avec $i+1 \leq k \leq j-1$, nous testons si nous pouvons transformer des BN de $S[k+1, j]$ en BR (fonction R), puis si la concaténation permet la formation d'une BE à la jonction des deux emboitages (fonction F). Nous gardons la meilleure de ces transformations. La fonction F sur une concaténation $S[i, k-1] \oplus t_k \oplus S[k+1, j]$ appelle la fonction *correspond* sur les noeuds provenant de la dernière boite de $S[i, k-1]$ et de la première pseudo-boite de $S[k+1, j]$ et la lettre t_k. Une BE est formée si dans le retour (q, α, err) de *correspond*, $err \neq$ *"nul"* en ajoutant l'erreur (k, err, α) dans P.

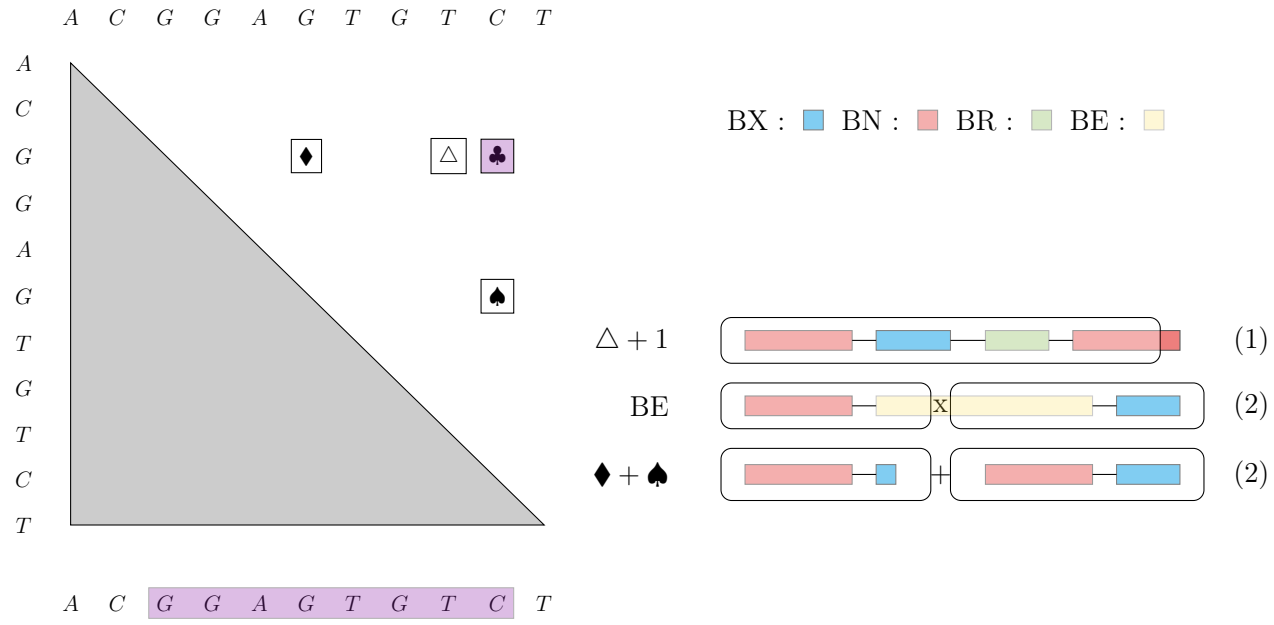


FIGURE 3.12 : Emboitage du texte non annoté $T = ACGGAGTGTCT$ par programmation dynamique. (Gauche) La case ♣, décrit le meilleur emboitage de $T_{2,9}$, facteur encadré en dessous. (Droite) Cette case est le meilleur résultat parmi les cas suivants : (1) l'emboitage de la case △ auquel on ajoute une lettre à droite ou la concaténation de deux cases (par exemple ♦ et ♠) avec (2) ou sans (3) possibilité de création de BE.

Lorsque la table S est complète, le pseudo-emboitage de la case $S[0, n-1]$ est traduit en emboitage. Nous devons commencer par définir les valeurs des éléments de chaque BR ainsi que les BN vers lesquelles elles pointent. Nous trouvons les feuilles, descendant du nœud représentant les boites, ayant identifiant et sens de lecture d'un motif identiques. Toutes les BN et BR ayant un nœud nul ont été éliminées grâce à la fonction min. Nous pouvons ensuite transformer les boites restantes en choisissant par exemple les valeurs se trouvant dans la première feuille descendant du nœud représentant la boite. Puis nous supprimons les pointeurs vers les nœuds de l'arbre N. L'emboitage représente le texte T.

Notre table de programmation dynamique est de taille n^2. Pour chaque case $S[i, j]$, nous allons regarder les $O(n)$ couples de cases de la colonne i et de la ligne j. Pour chaque couple, nous utilisons la fonction $R_B(B')$. Dans le pire des cas, il y a une pseudo-boite toutes les deux lettres, et la complexité de l'emboitage est $O(n^4)$. Mais dans le cas général, il y a sensiblement moins de pseudo-boites que de lettres, la complexité est $O(n^3 \times b)$, avec b étant le nombre de boites données par un emboitage de l'algorithme.

Nous jugeons la complexité de cet algorithme trop importante et décrivons maintenant un autre algorithme d'emboitage, nécessitant moins de temps, mais dont le résultat est approché.

3.2.3.2 Emboitage rapide non optimal, par optimisations successives

Afin d'utiliser moins de temps de calcul que l'algorithme précédent, nous allons pseudo-emboiter T en lisant le texte de gauche à droite, puis nous allons optimiser localement chaque boite en utilisant plusieurs règles successives.

Pseudo-emboitage glouton : Supposons que nous ayons pseudo-emboité $T_{0,i-2}$ en $B' = (B_0, q_0)t_u \dots t_w(B_{k-1}, q_{k-1})$, nous ajoutons la lettre t_{i-1} puis une pseudo-boite (B_k, q_k). Nous cherchons pour cela le plus grand j tel que le motif $T_{i,i+j-1}$ soit présent dans N : $noeud(T_{i,i+j-1}) = q$ et $q \neq \perp$. Si $j < Z_G$, alors $(B_k, q_k) = (BX(j, T_{i,i+j-1}), q)$ sinon $(B_k, q_k) = (BN(0, 0, 0, j), q)$. Nous emboitons tout le texte T de cette manière en $O(n)$.

Optimisation gloutonne locale : Lorsque des BN se forment ainsi, elles ont par construction une erreur à leur droite, sur la lettre t_{i+j}. Nous allons donc essayer de les agrandir vers la gauche pour trouver l'erreur à cette extrémité. De cette manière,

toutes les BN (possiblement transformées en BE ou BR) sont encadrées par des erreurs.

Nous optimisons ce pseudo-emboitage en agrandissant les BN et en intégrant les BE et les BR. Puis nous allons choisir des identifiants de motifs pour les boites ayant plusieurs identifiants possibles en donnant la priorité à des motifs déjà présents dans le texte. Une fois transformée, une boite ne pourra plus être modifiée ce qui conduit à une complexité linéaire de l'algorithme.

Nous formons une liste des positions des BN triées dans l'ordre croissant du nombre de feuilles descendant de chaque nœud de boite en $O(b \log b)$. Nous allons analyser puis traduire chaque BN dans cet ordre. De cette manière, nous pourrons faire des BR avec les dernières boites, en choisissant un identifiant (parmi les nombreux possibles) qui aura déjà été utilisé dans les boites déjà analysées.

Soit (B_k, q_k) la BN actuellement analysée, représentant le facteur $T_{b,c}$ et (B_{k-1}, q_{k-1}) représentant le facteur $T_{a,b-2}$, la boite qui est directement avant B_k dans l'ordre de T. B_k subit plusieurs analyses puis est traduite en boite :

(1) Formation de BE par fusion : Soit $correspond(q_{k-1}, t_{b-1}, q_k) = (q, \alpha, err)$, si $err \neq$ *"nul"*, alors nous pouvons former une BE (si la transformation réduit la taille totale de l'emboitage) : $(B_{k-1}, q_{k-1}) = (BE(0, 0, 0, c-a+1, 1, P), q)$ avec $P = \{(b-a+2, err, \alpha)\}$. Nous recommençons avec les boites $B_{k-2}, B_{k-3}, \ldots, B_0$ jusqu'à ne plus pouvoir agrandir la boite B_k. Puis nous procédons de la même manière avec les boites à droite $B_{k+1}, B_{k+2}, \ldots, B_{b-1}$. Cette étape se fait au maximum en $O(b)$ et est représentée dans la figure 3.13 cas (1).

(2) Agrandissement de la B_k : Si B_{k-1} est une BX ou n'a pas été analysée, nous allons essayer d'augmenter la longueur de B_k. Grâce à la manière dont elle a été construite, la pseudo-boite B_k ne peut être étendue vers la droite, nous allons donc essayer de l'étendre vers la gauche, lettre par lettre, en tronquant ces lettres à la boite B_{k-1}. Pour cela, nous cherchons le plus grand j tel que $noeud(T_{b-j,c}) \neq \perp$, et $j \in [0, a-t'-1] \cup \{a-t'+1\}$ (on ne veut pas deux lettres entre les boites). Nous pouvons donc augmenter la taille de $(B_k, q_k) = (B_k, (noeud(T_{b-j,c}))$, avec $B_k.t$ augmenté de j et diminuer celle de $(B_{k-1}, q_{k-1}) = (B_{k-1}, q)$, avec q, le $j$ème ancêtre de q_{k-1} et $B_{k-1}.t$ diminué de j. Cette étape se fait au maximum en $O(n)$ et est représentée dans la figure 3.13 cas (2).

(3) Vérification de la taille de la boite : Nous regardons la taille de la boite, si $t < Z_G$, alors la boite actuelle a été rognée par la boite se trouvant à sa droite et elle n'a pu être agrandie suffisamment dans les étapes précédentes, alors cette boite est traduite en BX avec $(B_k, q_k) = (BX(c-b+1, T_{b,c}), q_k)$. Cette étape est réalisée en

$O(1)$ et est représentée dans la figure 3.13 cas (4).

(4) Traduction de la pseudo-boite et sélection du motif : lors de la traduction de la pseudo-boite en boite, seuls les identifiants, sens et début sur le motif sont à déterminer parmi les feuilles descendantes de q_k. Nous maintenons une liste K contenant les identifiants des boites déjà traduites. Si plusieurs feuilles descendent de q_k, nous choisissons une feuille dont l'identifiant se trouve déjà dans K en temps $O(|K|)$. $|K|$ est maximisé par le nombre b de boites. S'il n'y en a pas, nous en choisissons une au hasard. La traduction de la boite se fait en temps $O(n \log b)$.

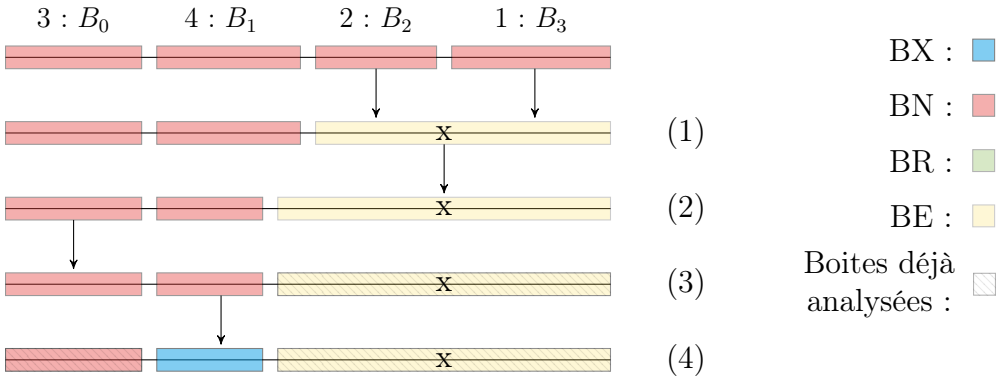


FIGURE 3.13 : Emboitage par optimisation gloutonne sur un texte. Le texte est emboité avec un emboitage glouton de gauche à droite. Il crée quatre BN qui sont triées dans l'ordre B_3, B_2, B_0, B_1. (1) B_3 peut être concaténée à B_2, mais pas à B_1, pour former une BE. (2) B_3 rogne ensuite la boite B_1 de quelques lettres. (3) B_2 n'existe plus, nous examinons B_0 qui ne peut pas être modifiée car c'est la première boite. (4) B_1 ne peut pas être concaténée ni agrandie car B_0 a déjà été analysée et elle est trop petite pour être une BN, elle est transformée en BX.

Dès que toutes les BN sont analysées, les pseudo-boites non analysées sont traduites en BX.

Le tri initial se fait en $O(b \log b)$. L'étape (1) se fait en temps amorti $O(b)$ car chaque boite ne peut être fusionnée qu'avec une seule autre. De la même manière, l'étape (2) se fait en $O(n)$ au total car chaque lettre ne peut être intégrée qu'à une boite. Les étapes (3) et (4) se font en temps $O(1)$ et $O(bn)$ pour chaque boite, soit $O(b)$ et $O(bn \log b)$ au total. L'algorithme a une compléxité totale de $O(b \log b + bn \log b + n + b)$. Il nécessite une mémoire de $O(b)$ bits pour la table, $O(m^2)$ bits pour l'arbre des suffixes et n pour le texte, soit un total de $n + O(b + m^2)$ bits. avec b le nombre de boites formées lors de l'emboitage de départ. Cet algorithme n'est pas optimal car plusieurs cas peuvent être oubliés. Par exemple, si deux BN sont séparées par une BX, l'algorithme ne va

pas tester la formation d'une BE recouvrant le tout, alors qu'il aurait pu économiser de la mémoire.

3.2.4 Optimiser la taille des boites

Lors de l'emboitage d'un texte, nous voulons que le texte emboité prenne le moins d'espace mémoire possible. Nous devons prendre en compte un élément qui sera présent avant chaque boite indiquant le type de la boite et donc la manière dont il faudra la décoder. Cet élément utilise une place de $\log(\beta)$ bits avec β le nombre de type de boites différentes (soit $\log 4 = 2$). Nous utilisons ensuite les optimisations suivantes.

Dans un premier temps, nous utilisons un codage de Huffman pour écrire les différents éléments d'une boite. Pour cela, nous emboitons le texte en sauvegardant dans une table les valeurs de chaque élément de chaque boite, puis nous calculons le code de chaque élément des boites (sauf l'élément O des BX occupant toujours 2 bits par lettres).

Ensuite, nous déterminons le nombre minimal de lettres que peuvent représenter les BN, BR et BE. Ils sont déterminés par la taille en bits qu'occupent ces boites (en utilisant un codage de Huffman ou non). Ainsi nous fixons Z_G tel que $\frac{\lambda(BN)-p}{\log\sigma} \leq Z_G < \frac{\lambda(BN)-p}{\log\sigma} + 1$, le nombre minimal de lettres que peut représenter une BN. Ce seuil détermine le moment à partir duquel nous pouvons choisir de fabriquer une BN plutôt qu'une BX. Nous calculons de manière similaire Z_R et Z_E, le nombre minimal de lettres que peut représenter une BR et une BE.

Ces deux optimisations demandent plusieurs analyses du texte car les codes des éléments des boites demandent à ce que les seuils des boites soient fixés (taille des codes), et le calcul des seuils doit connaître la place mémoire que chaque boite utilise. Ainsi nous pouvons utiliser les deux optimisations en boucle jusqu'à trouver un point d'équilibre, mais cela prendrait une durée indéterminée.

Pour calculer Z_G, Z_R et Z_E, nous emboitons le texte puis nous calculons les codes de chaque élément des boites.

Dans un texte ADN réel de 30M lettres et 1000 annotations différentes constituant 120K séquences V(D)J, nous avons utilisé l'emboitage glouton. Une lettre est codée en 2 bits, une BX est codée en $6 + 2t$ bits, avec t, le nombre de lettres que la boite représente, une BN est codée en 28 bits, une BR en 21 bits et une BE en $30 + 12r$ bits, avec r le nombre d'erreurs dans la boite. Les seuils sont donc $Z_G = 14$ lettres, $Z_R = 10$ lettres et $Z_E = 15 + 6r$ lettres. À partir des ces seuils, nous trouvons les

tailles minimales et maximales en bits des boites en utilisant un codage de Huffman et le théorème du codage sans bruit : une BX est codée en $6+2t$ bits, une BN en 9 à 15 bits, une BR en 9 à 13 bits et une BE en $11+9r$ à $17+9r$ bits. La table 3.1 synthétise ces données et la figure 3.14 les compare visuellement.

	BX	BN	BR	BE
éléments	(t, O)	(i, s, d, t)	(r, d, t)	(i, s, d, t, n, P)
$\lambda(B)$ sans Huffman	$6+2t$	28	21	$30+12r$
$\lambda(B)$ avec Huffman	$6+2t$	9 à 15	9 à 13	$11+9r$ à $17+9r$

TABLE 3.1 : Taille en bits de chaque type de boite avec et sans codage de Huffman.

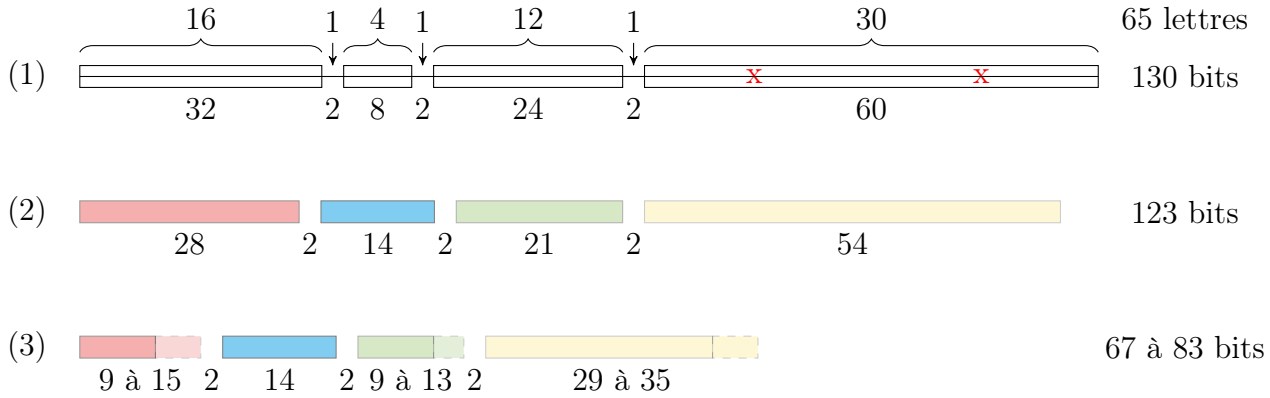


FIGURE 3.14 : Emboitons un texte de 65 lettres. Les trois lignes représentent la taille de la représentation de la séquence en bits. Les rectangles découpent le texte en sous-séquences que nous représentons par des boites. (1) Le texte n'est pas emboité et chaque lettre est écrite avec 2 bits. (2) Le texte est représentés avec des boites : une BN, une BX, une BR puis une BE. (3) Les éléments des boites sont codés avec des codes de Huffman.

3.3 Expérimentations

3.3.1 Méthodologie

Des tests ont été effectués pour connaître le taux de compression d'un emboitage. Nous avons réalisé un emboitage sur des fichiers de données réelles dont les séquences ne sont pas annotées, avec l'algorithme d'emboitage glouton de la section 3.2.3.2.

Procédure de test : Nous avons commencé par simuler un emboitage glouton de chaque séquence pour pouvoir calculer les seuils des tailles des boites. Puis nous avons

calculé l'entropie de chaque élément de chaque boite afin d'estimer l'espace mémoire qu'occupera chaque boite. Enfin nous avons simulé un second emboitage avec l'algorithme glouton.

Cette simulation ne crée pas les boites, mais calcule l'espace mémoire qu'occuperait l'emboitage de chaque séquence. Pour analyser les résultats, nous avons calculé la taille totale qu'occupe l'emboitage de plusieurs fichiers, puis nous nous sommes intéressés de plus près à l'emboitage de plusieurs séquences.

3.3.2 Compression d'un fichier

Les fichiers utilisés pour nos tests sont des données réelles issues de séquençage de répertoire immunologique de patients. Chaque fichier a été obtenu lors du diagnostic de la leucémie d'un patient. Ils proviennent de laboratoires d'analyse différents utilisant plusieurs séquenceurs et techniques de séquençage de répertoire. Parmi les séquences d'un fichier obtenu de cette manière se trouvent des séquences qui ne sont pas des recombinaisons V(D)J, nommées séquences non recombinées. Les divers échantillons tout comme les diverses techniques de séquençage provoquent des pourcentages de séquences recombinées différents. Toutes les séquences d'un même fichier sont emboitées en utilisant le même algorithme dans nos tests. Le fichier 09 a été publié dans l'article [45] et est disponible à l'adresse http://www.vidjil.org/data/#2018-peerjcs. Les fichiers 1 et 2 sont non publiés et proviennent d'un hôpital différent. Dans ces fichiers les variations peuvent être des mutations sur les gènes ou des erreurs de séquençage.

	Fichier 09	Fichier 1	Fichier 2 (extrait)
Recombinaisons V(D)J (%)	87.57	7.24	0.04
Taille fichier (nb lettres*2) (Mbits)	18	19	29.2
Taille emboité (Mbits) (quotient de compression)	8 (2.25)	20 (0.95)	28.4 (1.02)
Taille avec Huffman (Mbits) (quotient de compression)	4.5 (4)	11 (1.72)	24.3 (1.2)

TABLE 3.2 : Fichiers utilisés pour les tests de l'emboitage glouton. La table indique la taille du fichier non emboité, la taille de l'emboitage glouton, puis la taille de l'emboitage glouton en utilisant des codes de Huffman pour les éléments des boites.

Le tableau 3.2 indique la taille des fichiers, leur taux de recombinaison V(D)J ainsi que la taille de leur emboitage en bits. Nous pouvons voir que le pourcentage de recombinaisons V(D)J dans un fichier influe beaucoup sur la compression d'un fichier. Les tailles des emboitages du fichier 1 sont deux fois plus grandes que les tailles des

emboitages du fichier 09, bien que les fichiers de départ soient de même taille. Le fichier 2 est très peu compressé car il y a très peu de recombinaisons V(D)J.

3.3.3 Compression d'une séquence

Nous avons ensuite regardé en détail des emboitages de séquences réelles provenant du même fichier. Nous y voyons qu'à l'intérieur d'un même fichier, les emboitages peuvent être de tailles très différentes. Les figures 3.15 et 3.16 représentent deux séquences emboitées. Dans chacune d'elles, une première ligne représente la séquence, les gènes trouvés par l'analyse de l'algorithme du logiciel Vidjil ainsi que les variations avec ces gènes, la seconde ligne représente l'emboitage de la séquence obtenu par l'algorithme. La première séquence est compressée à 31%, grâce à une bonne reconnaissance des gènes. La seconde séquence possède beaucoup plus de variations et son emboitage utilise plus de mémoire que la première séquence.

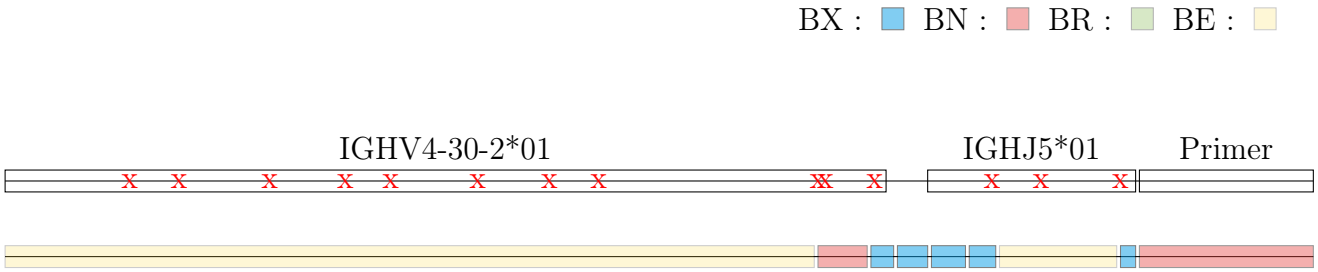


FIGURE 3.15 : Emboitage d'une séquence VJ de 346 lettres. Chaque variation est indiquée par un symbole x. La séquence est composée du gène IGHV4-30-2*01 avec 11 variations et du gène IGHJ5*01 avec 3 variations suivies d'une séquence constante (primer). La seconde ligne nous montre les boites qui ont été trouvées avec l'emboitage glouton. L'emboitage occupe 326 bits (217 avec le codage de Huffman) alors que la séquence brute en aurait occupé 692 en toute lettres. Une BE ayant le bon identifiant couvre quasiment tout le gène V sauf sa fin car les trois dernières variations sont trop proches. Une BE couvre seulement la moitié du gène J. Des BX couvraient le primer avant que nous ne le rajoutions dans l'ensemble de motifs.

L'emboitage des séquences non recombinées est constitué de boites de taille 7 à 10 et utilise beaucoup de mémoire (de 1300 à 1500 bits pour une séquence de 550 lettres). Ils sont considérés de la même manière qu'une séquence aléatoire : un codage de Huffman ne permet que peu d'économie de mémoire. Ces séquences sont problématiques lors de la compression d'un fichier par emboitage mais l'algorithme d'emboitage

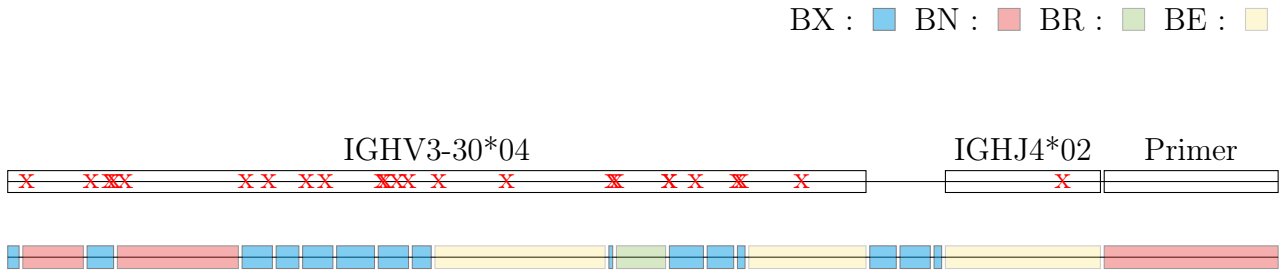


FIGURE 3.16 : Résultat de l'emboitage d'une séquence VJ de 336 lettres. Cette séquence est bien plus bruitée, avec au total 22 variations dans le gène V et une dans le gène J. L'emboitage occupe 595 bits (512 avec un codage de Huffman), alors que la séquence brute en aurait occupé 672. L'emboitage a bien reconnu le gène J mais les nombreuses variations très proches dans le gène V ont rendu l'emboitage difficile.

pourrait nous permettre de les trouver en considérant seulement la taille de l'emboitage de ces séquences.

Dans un fichier, le taux de séquences V(D)J fait beaucoup varier la taille de l'emboitage. Nous pouvons obtenir un fichier 4 fois plus petit lorsque il y a 90% de recombinaisons V(D)J (table 3.2 fichier 09). Dans une séquence, plus il y a de variations, plus les boites seront petites, et la taille de l'emboitage grande. L'espacement entre les variations joue aussi un rôle dans la formation des boites : deux variations proches empêcheront la formation d'une BE.

3.4 Discussion

3.4.1 Entropie et compression

Mora et ali. [30] ont travaillé sur l'entropie des recombinaisons V(D)J (par groupe) en examinant la fréquence d'apparition de chaque recombinaison. Sans proposer de codage, ils présentent par exemple que la recombinaison V(D)J d'un récepteur d'immunoglobuline contient 74 bits d'informations. Ce seuil peut se voir comme une borne inférieure théorique pour la représentation de recombinaisons V(D)J. Nous obtenons une moyenne proche de cette limite avec notre emboitage glouton sur le fichier 09 grâce un codage moyen utilisant 86 bits par séquences.

Nous savons par construction que l'algorithme glouton compresse moins bien que l'algorithme optimal (voir 3.2.3.1). Nous avons donc implémenté l'algorithme glouton

afin d'avoir une idée de la compression minimum d'un emboitage. Il serait intéressant d'implémenter les deux autres algorithmes d'emboitage afin de comparer leur résultat à la limite théorique de Mora et ali. Ces implémentations n'ont pas été effectuées car l'emboitage ne permet pas de requêtes efficaces sur les motifs et/ou les annotations (voir 3.4.2). De plus, nous aurions voulu savoir à quel point l'algorithme glouton peut être proche du résultat de l'algorithme optimal de la section 3.2.3.1.

Le principe de l'emboitage d'une séquence repose sur le fait que plus la séquence ressemble à un motif, plus la taille de son emboitage sera petite. Dans les fichiers représentant le répertoire immunologique d'un patient, un algorithme d'emboitage pourrait être utilisé entre autre pour détecter les séquences non recombinées : les séquences dont l'emboitage est le plus coûteux en mémoire.

3.4.2 Pertinence de la compression pour la recherche dans le texte

A partir de l'emboitage d'un texte, nous pourrions construire un index utilisant les propriétés des boites. Celui-ci pourrait regrouper les boites par identifiant de motif et nous pourrions ainsi faire des recherches sur les séquences ayant une boite provenant d'un certain motif.

Les réflexions sur l'emboitage et une indexation l'utilisant ont été abandonnées car la recherche d'un motif P dans le texte représenté par des boites est difficile. Considérons seulement les motifs internes à une boite.

- BN et BR : la recherche est possible. Nous devons chercher P dans tous les motifs M_x de M puis dans chaque boite représentant un motif M_x ayant P.
- BX : la rechercher est difficile. Les boites étant écrites en toutes lettres, nous ne pouvons pas nous aider de M.
- BE : la recherche est difficile. Il serait trop long de rechercher P dans les BE car nous devrions considérer tous les motifs approchés de P dans M dont une variation d'une BE pourrait produire P.

L'emboitage permet une bonne compression mais pas de recherche efficace parmi les données. Nous nous sommes donc tournés vers une nouvelle indexation utilisant une transformée de Burrows-Wheeler (BWT) car elle permet une bonne compression des données ainsi qu'une recherche de motif rapide. Le chapitre suivant propose plusieurs structures d'indexation.

Chapitre 4

Indexation de séquences annotées

Le chapitre 3 nous a montré comment compresser notre texte annoté, mais nous avions difficilement accès aux données. Soit (T, A) un texte annoté, nous voulons pouvoir rapidement trouver les occurrences d'un motif, d'une annotation, ou d'un motif annoté.

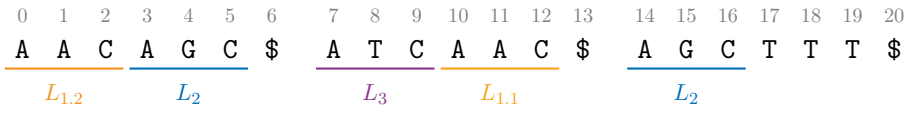


FIGURE 4.1 : Texte annoté (T, A) de taille 21. Il est composé de 3 séquences et 5 annotations issues de 4 annotations différentes, annotant 15 lettres.

Sur le texte présenté dans la figure 4.1, l'annotation de la lettre à la position 4 est $label(4) = L_2$. Les occurrences du motif `GC` sont $findP(\texttt{GC}) = \{4, 15\}$. L'annotation L_2 se trouve sur les positions $findL(L_2) = \{3, 4, 5, 14, 15, 16\}$. Enfin, les occurrences du motif `GC` dont la première lettre (au moins) est annotée L_2 sont aux positions $findPL(\texttt{GC}, L_2) = \{4, 15\}$. Ce texte sera utilisé pour tous les exemples de ce chapitre.

Dans le chapitre précédent il nous était difficile de répondre à ces requêtes car nous devions décoder les boites pour pouvoir rechercher un motif. Nous allons maintenant chercher à indexer ces séquences.

Ce chapitre présente plusieurs index utilisant des structures de données. Nous présentons dans la section 4.1 trois index utilisant une transformée de Burrows-Wheeler (voir section 2.5) : une solution naïve utilisant également une table associative ainsi que

deux index utilisant un Wavelet Tree (voir section 2.4.2). Les constructions détaillées des deux derniers index sont présentées dans la section 4.2. Nous allons comparer tout au long de ce chapitre ces deux structures à la structure naïve en commençant par les requêtes qu'ils autorisent dans la section 4.3. Nous introduisons la notion de famille d'annotations et comment les utiliser dans la section 4.4. Les index ont été testés sur plusieurs jeux de données dont les résultats sont présentés dans la section 4.5.

Les résultats de ce chapitre ont été présentés lors de plusieurs conférences nationales et workshop : JOBIM en juin 2016 à Lyon [40], Seqbio en novembre 2016 à Nantes [39] ainsi qu'à LSD & LAW en février 2017 à Londres [41]. L'article intitulé *Indexing labeled sequences* [42] a finalement été publié dans le journal *PeerJ Computer Science* en mars 2018.

4.1 Trois index pour stocker textes et annotations

Cette section présente le raisonnement, qui depuis un index naïf, nous a permis de créer deux index moins coûteux répondant à toutes nos requêtes. La construction précise de ces deux index sera plus développée dans la section 4.2.

Le texte annoté (T, A) peut être vu comme deux textes différents : le texte T et les annotations A. Une lettre et son annotation sont liées par leur position dans leur texte respectif. Une manière simple de stocker ces textes est de les indexer chacun de leur côté. D'après les requêtes demandées, nous aurons essentiellement besoin de faire des recherches de motif dans le texte T, nous pouvons indexer T avec un FM-index (U) (défini dans la section 2.5) où une recherche du motif P se fait extrêmement rapidement, en temps $O(|P|)$. Dans le texte d'annotations A, nous n'avons pas besoin de rechercher des motifs mais plutôt d'accéder à toutes les occurrences d'une annotation, nous pouvons pour cela regrouper toutes les occurrences d'une annotation dans une table associative : chaque annotation est liée à une liste de doublons (*début*/*fin*) décrivant les positions de chaque facteur annoté avec cette annotation. L'association d'une lettre à son annotation étant difficile, nous stockons de plus les annotations dans l'ordre du texte avec un vecteur d'annotations compressé A' suivant le principe du run-length encoding. Cet index, que nous avons nommé **HT-index** est représenté dans la figure 4.2. La figure 4.5 représente un HT-index indexant l'exemple de la figure 4.1.

Dans le HT-index, les annotations sont stockées sous deux formes différentes pour permettre toutes les requêtes, cela utilise beaucoup de mémoire (la taille du vecteur de bits et deux fois le nombre de facteurs annotés). Nous pouvons n'utiliser qu'une

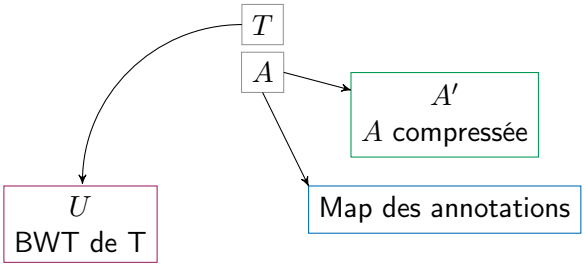


FIGURE 4.2 : Le HT-index : une BWT indexe le texte T. Les annotations A sont compressées grâce à un vecteur de bits. Une table associative indique les positions de chaque facteur annoté.

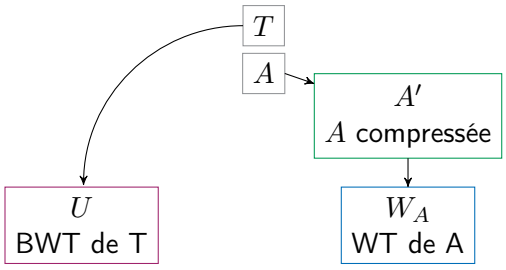


FIGURE 4.3 : TL-index : la BWT U indexe le texte T. Les annotations A sont compressées grâce à un vecteur de bits. Le WT W_A indexe la version compressée de A.

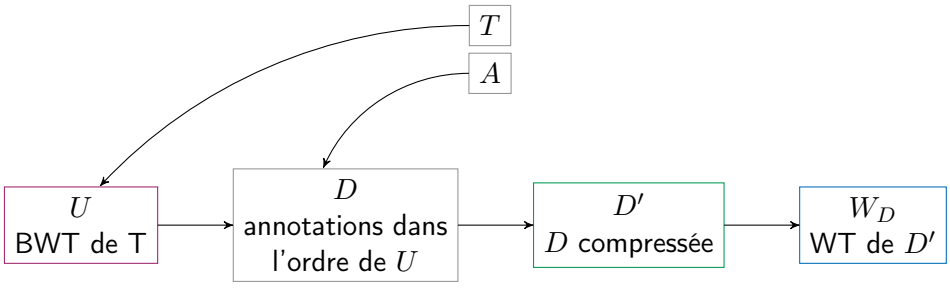


FIGURE 4.4 : TL_{BW}-index : la BWT U indexe le texte T. Les annotations sont ensuite triées dans l'ordre de U, compressées et indexées dans un WT W_D. Notons que D n'est pas dans l'index.

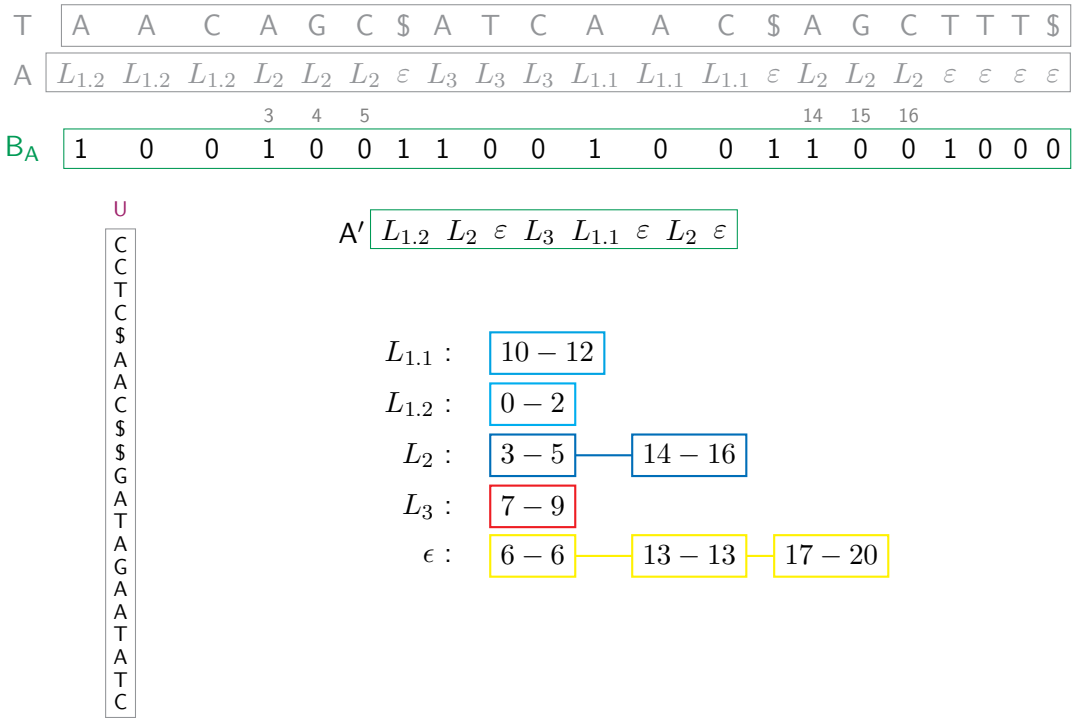


FIGURE 4.5 : Dans le HT-index, la séquence d'annotations A est stockées sous la forme du vecteur A' et du vecteur de bits B_A. Nous avons $B_A[4] = 0$ car $a_4 = a_3 = L_2$, et $B_A[3] = 1$ car $a_3 \neq a_2$. La table d'associations a 5 listes représentant les 5 annotations possibles. L'annotation L_3 est placée sur 3 lettres aux positions 7, 8 et 9. Notons que le texte T et la séquence d'annotations A ne sont pas stockés dans l'index.

seule structure : un WT (W_A) pour indexer et compresser les annotations de A'. Les occurrences d'une même annotation sont regroupées dans une feuille de l'arbre, nous n'avons plus besoin de table associative. Le WT a l'avantage d'autoriser des parcours de l'arbre dans les deux sens : trouver la lettre occupant une position (qui peut être étendu à trouver le nombre d'occurrences d'une lettre dans un intervalle) et trouver les positions d'une lettre. Ce nouvel index, le **TL-index**, schématisé dans la figure 4.3, stocke toujours le texte dans une BWT.

Cet index permet de réaliser des requêtes sur les motifs ou les annotations simplement et rapidement, chaque requête s'exécutant sur une partie de l'index (BWT pour les motifs, WT pour les annotations). Malheureusement, les requêtes sur les motifs annotés sont difficiles car nous avons besoin de passer d'une structure à l'autre. Les annotations et les lettres n'étant liées que par leur position, nous devrons faire des traductions de positions à mi-chemin de la requête.

La BWT rapproche les suffixes identiques et donc augmente le nombre de lettres

identiques successives. Des facteurs identiques auront probablement la même annotation : il serait intéressant de profiter des remarques précédentes pour rapprocher des annotations identiques grâce à la BWT. Nous allons donc stocker les annotations dans l'ordre de la BWT dans un nouvel index, nommé **$\mathbf{TL_{BW}}$-index**, représenté dans la figure 4.4.

Dans cet index, les requêtes sur les motifs annotés seront rapides car lettres et annotations sont rangées dans le même ordre. Mais les requêtes sur les annotations devront passer par le WT puis par la BWT, pour trouver les positions des annotations dans l'ordre du texte T, augmentant leur temps d'exécution.

4.2 Construction des index

Cette section décrit la construction du TL-index et du $\mathrm{TL_{BW}}$-index puis indique leur temps de construction et la mémoire qu'ils occupent.

TL-index

Soit (T, A), un texte annoté, nous définissons le TL-index comme (U, B_A, W_A), utilisant une transformée de Burrows-Wheeler U pour indexer T, un vecteur de bits B_A marquant les positions du texte où les annotations changent, et un Wavelet Tree W_A indexant une séquence d'annotations compressée de la même manière que pour le HT-index. Voir figure 4.6.

BWT U. Soit $U = u_0 u_2 \dots u_{n-1}$ la BWT du texte T. Comme habituellement fait dans un FM-index, nous échantillonnons toutes les $\log^{1+\epsilon}$ valeurs d'une table des suffixes pour retrouver la position dans le texte de toute occurrence [34]. Depuis la position d'une lettre dans U, nous pouvons trouver sa position dans l'ordre du texte T en temps $O(log^{\epsilon+1} n)$.

Vecteur de bits B_A. Soit B_A un vecteur de bit compressé de taille n tel que $B_A[0] = 1$, et pour $i \geq 1$, $B_A[i] = 0$ si $a_i = a_{i-1}$, sinon $B_A[i] = 1$.

Wavelet Tree W_A. Soit $A' = \langle a_i | B_A[i] = 1 \rangle$. $A' = a'_0 a'_1 \dots a'_{a-1}$ est appelé la **séquence d'annotations compressée**. C'est un sous-vecteur de A, de taille a, contenant seulement les annotations successives différentes de A. La séquence d'annotations

compressée A' est stockée dans un Wavelet Tree W_A. Nous expliquons dans la section 4.4 comment la forme de W_A peut être modifiée de manière à dépendre d'une hiérarchie d'annotations.

W_A fait partie de la structure d'indexation. Ce wavelet tree est utilisé pour répondre efficacement à des requêtes bidimensionnelles où le texte et les annotations sont les deux dimensions. Un Wavelet Tree équilibré a une profondeur de $\log \ell$, avec ℓ le nombre de feuilles. L'accesseur $W\langle i \rangle$ renvoie a'_i en temps $O(\log \ell)$. C'est une requête classique dans un wavelet tree. Soit L_x une annotation de L, la fonction $select(L_x, i)$ retourne la position de la $i$ème annotation L_x de A' en temps $O(\log \ell)$. L'accesseur $W^{-1}\langle L_x \rangle$ renvoie la liste de positions de A' où $a'_i = L_x$. Il est effectué en temps $O(\log \ell \times occ)$, où occ est le nombre d'occurrences de L_x dans A'.

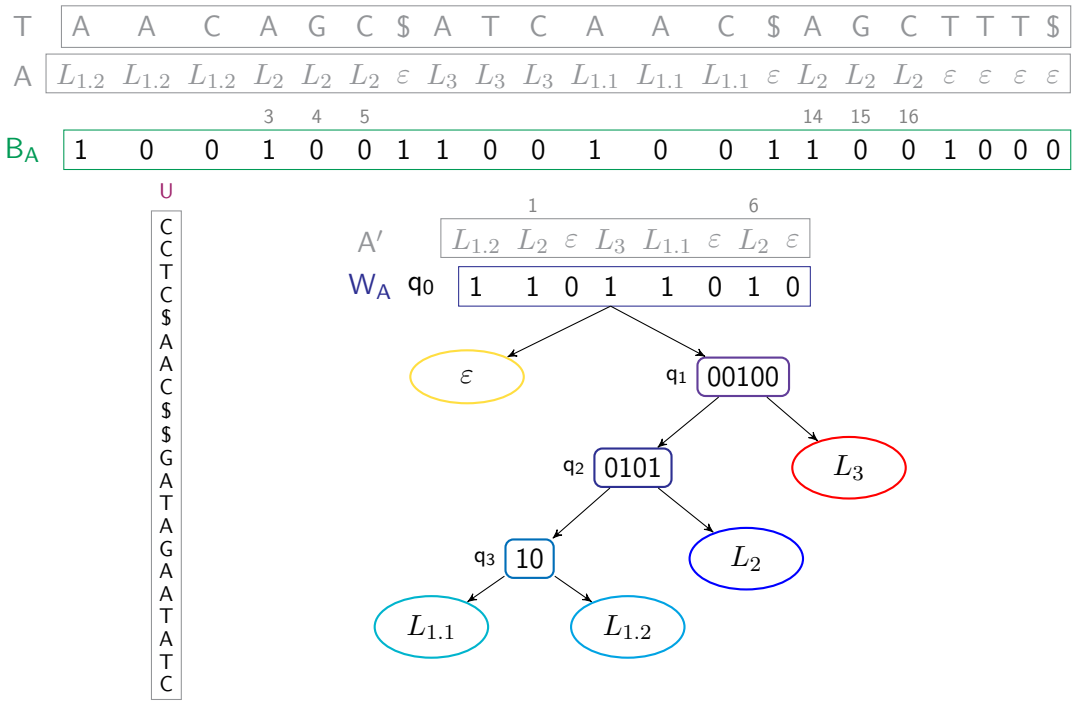


FIGURE 4.6 : Le wavelet tree W_A a 4 nœuds internes et 5 feuilles. L'annotation $a_3 = L_2$ est stockée dans A' à la position 1, donc $W\langle 1 \rangle = a'_1 = L_2$. A' possède deux occurrences de l'annotation L_2 : $W^{-1}\langle L_2 \rangle = \{1, 6\}$, correspondant aux six positions $\{3, 4, 5, 14, 15, 16\}$ dans A. Notons que la séquence d'annotations A et la séquence d'annotations compressée A' ne sont pas stockées dans l'index.

$\mathbf{TL_{BW}}$-index

Soit (T, A) un texte annoté, le $\mathrm{TL_{BW}}$-index est défini par (U, B_D, W_D), voir figure 4.7. La BWT U est construite de la même manière que dans le TL-index. Nous stockons les annotations dans l'ordre de U. Soit $D = d_0 d_1 \dots d_{n-1}$ les annotations de $U = u_0 u_1 \dots u_{n-1}$. Soit B_D, un vecteur de bits de taille n tel que $B_D[0] = 1$, et pour $i \geq 1$, $B_D[i] = 0$ si $d_i = d_{i-1}$, sinon $B_D[i] = 1$. Soit $D' = \langle d_i | B_D[i] = 1 \rangle$. $D' = d'_0, d'_1 \dots d'_{d-1}$ est une séquence d'annotations compressée de taille d, sous-séquence de D. Le wavelet Tree W_D indexe maintenant la séquence d'annotations compressée D'. Le $\mathrm{TL_{BW}}$-index sera plus long à construire que le TL-index car la construction de D nécessite de lire le texte intégralement. D'un autre côté, comme les annotations sont rangées dans l'ordre des lettres de la BWT, les requêtes combinant texte et annotation seront plus rapides.

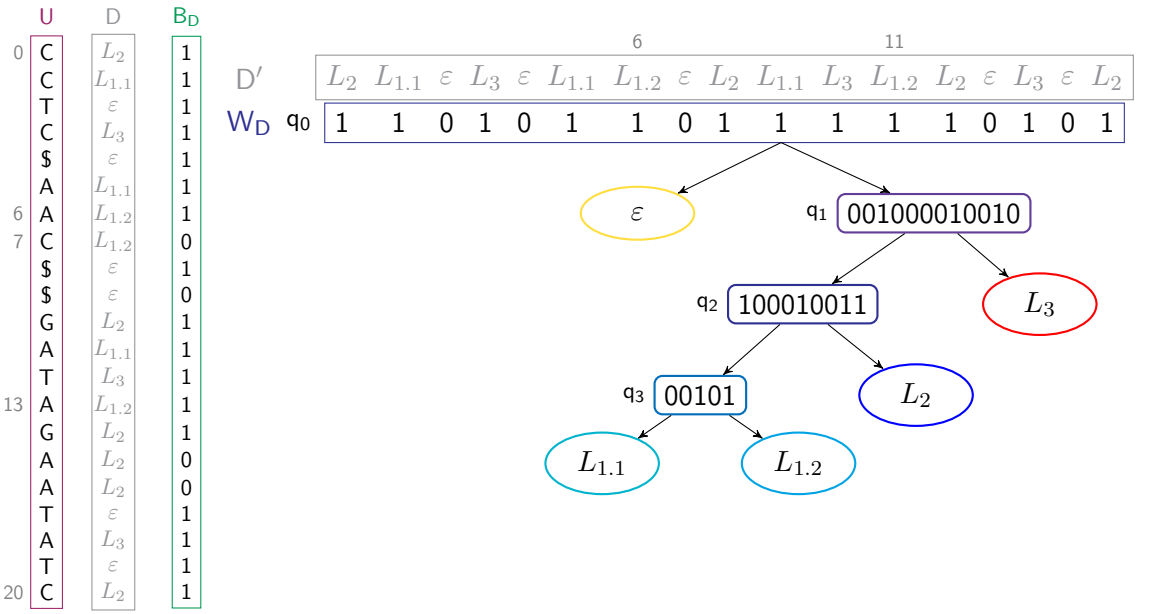


FIGURE 4.7 : La racine D' du wavelet tree est maintenant construite dans l'ordre de la BWT U. Le wavelet tree W_D a 4 nœuds internes et 5 feuilles. L'annotation $d_6 = L_2$ est stockée dans D' à la position 6, donc $W\langle 6 \rangle = d'_6 = L_1.2$. D' a deux occurrences de l'annotation $L_{1.2}$: $W^{-1}\langle L_{1.2} \rangle = \{6, 11\}$ correspondant aux trois positions $\{6, 7, 13\}$ dans U. Notons que la séquence d'annotations D et la séquence d'annotations compressée D' ne sont pas stockées dans l'index.

Temps et mémoire

Rappelons que le texte annoté (T, A) de longueur n, possède ℓ' facteurs annotés provenant de ℓ annotations uniques. Comme défini dans la section 2.5, les index stockent U en $nH_k(T) + o(n)$ bits. Le TL-index stocke le vecteur de bits avec les fonctions *rank* et *select* en $nH_0(B_A) + o(n)$ bits (voir section 2.4.2). La taille de W_A dépend de la séquence d'annotations compressée A', de longueur a. W_A utilise $aH_0(A') + o(a \log \ell)$ bits (voir section 2.4.2). De la même manière, le TL_{BW}-index stocke B_D en $nH_0(B_D) + o(n)$ bits et W_D nécessite $dH_0(D') + o(d \log \ell)$ bits, où d est la longueur de D'. La table d'association est construite en lisant A en temps $O(n)$ et utilise $O(\ell')$ bits. La transformée de Burrows-Wheeler peut être construite en temps linéaire en utilisant peu d'espace [3, 31]. B_A est construit en lisant A en temps $O(n)$. Pour fabriquer B_D, nous avons besoin de lire les annotations dans l'ordre du texte original en temps $O(n)$. Pour construire W_A, nous trouvons les occurrences de chaque annotation correspondant à un bit 1 dans B_A, en temps $O(a)$. Ensuite nous fabriquons la forme de W_A en temps $O(\ell)$. Les annotations correspondant à un bit 1 sont extraites pour faire la racine q_0 du WT. Pour chaque nœud contenant au moins deux annotations, nous les séparons en suivant la forme précédemment calculée, en temps $O(a\lceil \log \ell / \sqrt{\log a} \rceil)$ [32]. Nous construisons W_D de la même manière.

Le HT-index a une taille totale de $nH_k(T) + nH_0(B_A) + a + \ell' + o(n)$ et est construit en temps $O(n \log \ell')$. Le TL-index a une taille totale de $nH_k(T) + nH_0(B_A) + aH_0(A') + o(n \log \ell)$ bits, en supposant que $\sigma = O(\ell)$, et est construit en temps $O(n + \ell + a\lceil \log \ell / \sqrt{\log a} \rceil)$. Le TL_{BW}-index a une taille totale de $nH_k(T) + dH_0(D') + o(n \log \ell)$ bits et est construit en temps $O(n + \ell + d\lceil \log \ell / \sqrt{\log d} \rceil)$.

4.3 Requêtes

Les index permettent les requêtes classiques suivantes.

— *label*(i) – **Quelle annotation est sur la lettre** t_i **?** Cette requête est réalisée en $O(1)$ dans le HT-index, en $O(\log \ell)$ dans le TL-index et en $O(\log^{1+\epsilon} n + \log \ell)$ dans le TL_{BW}-index puisque nous devons traduire la position dans l'ordre de U. Voir figure 4.8.

— *findP*(P) – **Quelles sont les occurrences du motif** P **?**
Cette requête est résolue avec le FM-index seul [11]. Elle est exécutée en temps $O(|P| + occ \times \log^{1+\epsilon} n)$ dans les trois index, où occ est le nombre d'occurrences

de P dans T.

— *findL*(L_x) – **Quelles sont les occurrences de l'annotation L_x ?**
  — HT-index : nous accédons directement à la liste des occurrences grâce à la table associative. Nous lisons la liste et en extrayons les éléments en $O(y)$, avec Y' la liste des doublons (*début*/*fin*) représentant les occurrences de l'annotation et $y = |Y'|$.
  — TL-index : la requête est exécutée dans le WT seul en temps $O(y \times \log \ell)$, avec $Y = W^{-1}\langle L_x \rangle$ et $y = |Y|$. Notons en effet que Y a la même taille que Y', défini pour le HT-index.
  — TL_{BW}-index : nous avons besoin en plus de traduire la position de chaque occurrence dans l'ordre du texte original. La requête s'exécute alors en temps $O(y(\log \ell + \log^{1+\epsilon} n))$, avec $Y = W^{-1}\langle L_x \rangle$ et $y = |Y|$. (Voir figure 4.9.)

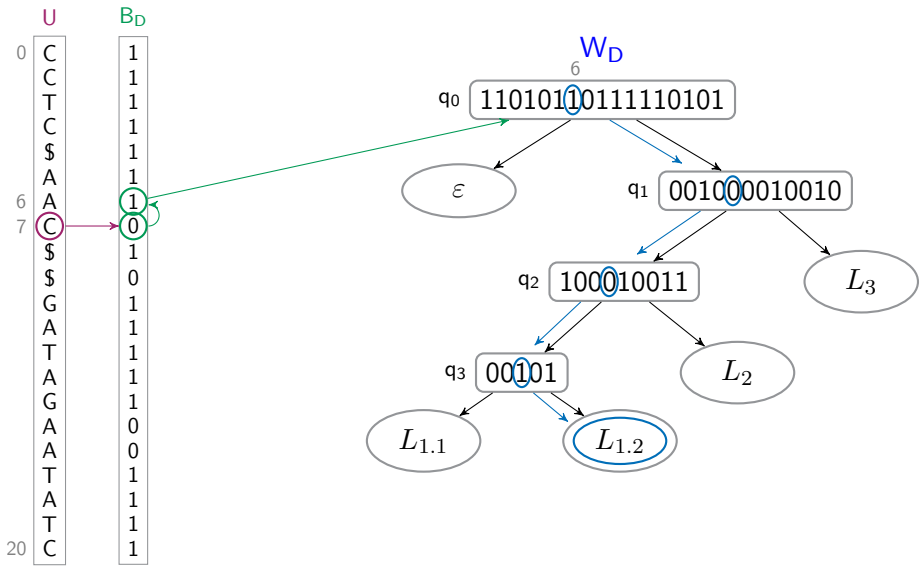


FIGURE 4.8 : Rechercher l'annotation d'une lettre dans un TL_{BW}-index. La lettre u_7 correspond à un bit 0 dans B_D. Le bit 1 précédent dans B_D est le bit à la position 6. C'est le 7ème bit 1 de B_D, il correspond au 7ème bit, à la position 6, dans la racine de W_D dont l'annotation est $W\langle 6 \rangle = L_{1.2}$.

Les trois requêtes précédentes sont des requêtes classiques sur les structures d'indexation. Les deux requêtes suivantes recherchent un motif et une annotation en même temps. Elles sont inédites et propres à ces structures pour texte annoté.

— *countPL*(P, L_x) – **Combien de positions du texte sont annotées L_x et commencent le motif P ?** Comme pour la requête *findL*(L_x), les occurrences du motif P se trouvent dans U aux positions i à j.

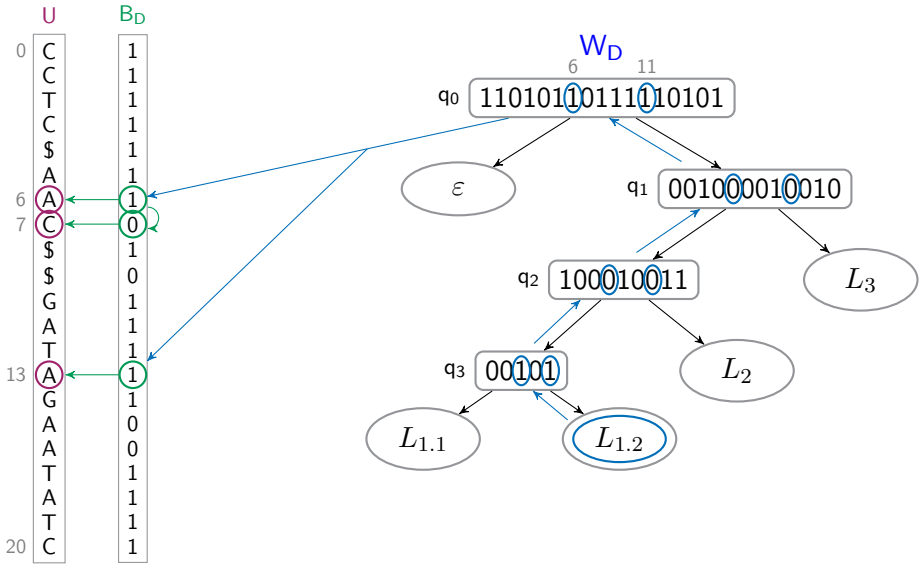


FIGURE 4.9 : Rechercher les occurrences de l'annotation $L_{1.2}$ dans un TL_{BW}-index. $Y = W^{-1}\langle L_{1.2}\rangle = \{6, 11\}$. La feuille ayant l'annotation $L_{1.2}$ est dans le sous-arbre droit de q_3, nous avons les bits 1 de q_3 correspondant à B_D ($\{6, 13\}$). et pour tous les bits 0 dans D qui les suivent ($\{7\}$). Les lettres correspondantes dans U, aux positions $\{6, 7, 13\}$, sont annotées $L_{1.3}$.

— HT-index : Nous traduisons ces $occ_P = j - i + 1$ positions pour avoir les positions correspondantes dans le texte. Pour chacune d'elles, nous regardons la valeur de l'annotation. Cette requête est exécutée en temps $O(|P| + occ_P \times \log^{1+\epsilon} n)$.

— TL-index : Nous traduisons ces occ_P positions pour avoir les positions correspondantes dans le texte. Pour chacune d'elles, nous vérifions si $label(i) = L_x$. Le temps total est $O(|P| + occ_P(\log^{1+\epsilon} n + \log \ell))$.

— TL_{BW}-index : Décrit dans l'algorithme 2. i et j correspondent aux positions $i' = rank(1, i, B_D)$ et $j' = rank(1, j, B_D)$ dans la racine de W_D. Nous utilisons ensuite un accesseur personnalisé depuis l'accesseur *rangeLocate* de [26], simulant une requête bidimensionnelle sur $[L_x, L_x] \times [i', j']$ dans W_D et donnant l'ensemble de positions $Z = \{z \,|\, a'_z = L_x \text{ et } i' \leq z \leq j'\}$ en temps $O(|Z| \times \log \ell)$ (boucle *for* des lignes 6 à 10). Cet accesseur commence par traverser le WT de la racine à la feuille L_x puis de cette feuille à la racine pour trouver les positions de $q_0.val$ qui correspondent à l'annotation dans la zone donnée. Pour toute position trouvée, nous cherchons les positions correspondantes dans B_D et les étendons aux bits 0 suivants dans B_D (boucle *for* des lignes 12 à 16). Cette requête est

exécutée en temps $O(|P| + |Z| \times \log \ell)$.

— *findPL*(P, L_x) – **Quelles séquences ont un motif P annoté A ?**

— TL-index et HT-index : Cette requête est exactement la même que *countPL*(P, L_x) pour ces deux index.

— TL_{BW}-index : Nous utilisons la requête *countPL*(P, L_x) détaillée dans l'algorithme 2, remplaçant le compteur *cnt* avec une liste Y gardant les positions correspondantes dans U. Les positions sont ensuite converties dans l'ordre du texte. Cette requête est exécutée en temps $O(|P| + |Z| \times \log \ell + |Y| \times \log^{1+\varepsilon} n)$.

Algorithm 2 *countPL*(P, L_x) : Compte les positions commençant un motif P et annotées L_x

(i, j) de *findP*(P) ▷ positions de début et fin des occurrences de P dans U
$i' = rank(1, i, B_D)$
$j' = rank(1, j, B_D)$
$C = path(L_x)$ ▷ vecteur de bits représentant le chemin de la racine à la feuille $leaf(L_x)$
$node = q_0$
for p allant de 0 à $|C| - 2$ **do** ▷ boucle correspondant à *rangeLocate* de [26]
    $i' = rank(C[p], i' - 1, node.val)$
    $j' = rank(C[p], j', node.val) - 1$
    $node = (C[p] == 0)?node.left : node.right$
    if $i' > j'$ **then return** 0
$cnt = 0$
for k allant de i' à j' **do**
    $k' = selectW\langle L_x, rank(C[|C| - 1], k, node.val)\rangle$ ▷ $i$ème L_x annoté dans A'
    $i'' = select(1, k', B_D)$
    $j'' = select(1, k' + 1, B_D)$
    $cnt = cnt + j'' - i''$ ▷ positions $[i'', j'' - 1]$ de U prises en compte
return *cnt*

Les temps d'exécution des fonctions sont résumés dans la table 4.1.

Requêtes	HT-index	TL-index	TL_{BW}-index
label(i)	$O(1)$	$O(\log \ell)$	$O(\log^{1+\epsilon} n + \log \ell)$
findP(P)	$O(\|P\| + occ_P \times \log^{1+\epsilon} n)$		
findL(L_x)	$O(y)$	$O(y \times \log \ell)$	$O(y(\log \ell + \log^{1+\epsilon} n))$
countPL(P, L_x)	$O(\|P\| + occ_P \times \log^{1+\epsilon} n)$	$O(\|P\| + occ_p \times (\log^{1+\epsilon} n + \log \ell))$	$O(\|P\| + \|Z\| \times \log \ell)$
findPL(P, L_x)			$O(\|P\| + \|Z\| \times \log \ell + y \times \log^{1+\varepsilon} n)$

TABLE 4.1 : Temps d'exécution des requêtes sur chaque index. Nous avons $|Z| \leq y \leq occ_p$.

Comme $|Z| \leq |Y| \leq occ_p$, les requêtes *countPL*() et *findPL*() peuvent être plus rapides sur le TL_{BW}-index, $|Z|$ dépendant de la compression que B_D peut faire sur Y.

Notons que dans le TL_{BW}-index, la requête $countPL(P, L_x)$ pourrait être plus rapide si le Wavelet Tree était directement construit à partir des annotations, sans le vecteur de bit intermédiaire B_D : la réponse pourrait être connue au moment où nous atteignons la feuille du WT en temps $O(|P| + \log \ell)$. Mais de cette manière le WT est toujours de grande taille, avec n bits à chaque étage complet. Nous avons préféré opter pour une structure de faible taille lorsque une compression de A est possible (économisant les n bits de B).

4.4 Familles d'annotations

Les annotations peuvent être organisées dans une **hiérarchie d'annotations**, étant donné un ensemble $F = \{F_0, ..., F_{f-1}\}$ de **familles d'annotations**. Cette hiérarchie peut être utilisée en hématologie avec les gènes V, D et J. Les allèles d'un même gène seraient regroupés dans la même famille (`V2-01*01`, `V2-01*02`), puis les gènes ayant le même numéro (`V2-01`, `V2-04`), puis les gènes ayant la même fonction (`V1`, `V3`).

Les deux index ayant un WT peuvent être adaptés : le WT W_A (respectivement W_D) aura une forme en concordance avec la hiérarchie d'annotations, et les nœuds internes q de W_A (respectivement W_D) pourront avoir une valeur de $q.label$ non vide appartenant à F. Par exemple, sur la figure 4.10, nous pouvons placer l'annotation $q_1.label = L_1$, où L_1 est le nom de la famille regroupant les annotations $L_{1.1}$, $L_{1.2}$ et $L_{1.3}$.

FIGURE 4.10 : Une hiérarchie d'annotations n-aire (gauche) peut être représentée avec un arbre binaire qui donne la forme du WT (droite). Dans les deux arbres, la famille d'annotations L_1 a 3 descendants, $L_{1.1}$, $L_{1.2}$ et $L_{1.3}$.

Les requêtes *findL*() et *findPL*() peuvent être naturellement utilisées avec les familles d'annotations. À partir de la hiérarchie décrite avec la figure 4.10, la requête

findL(L_1) doit trouver les séquences qui ont une annotation $L_{1.1}$, $L_{1.2}$ ou $L_{1.3}$. Une telle requête n'a pas besoin d'être itérée sur chaque annotation de la famille L_1, mais va plutôt commencer à partir du nœud interne correspondant à la famille : q_1 sur la figure 4.10.

Former l'arbre W en considérant la hiérarchie des annotations peut augmenter la profondeur du WT jusqu'à $O(\ell)$ dans le pire des cas. Pour utiliser moins de mémoire et obtenir un meilleur temps moyen de requête, nous utilisons l'idée de l'arbre de Huffman [18]. Une feuille qui correspond à une annotation fréquemment utilisée sera placée plus haut dans l'arbre que la feuille correspondant à une annotation rarement utilisée. En fonction de la hiérarchie des annotations, la profondeur du WT est $H_0(A')$ (respectivement $H_0(D')$) dans le meilleur des cas, tandis qu'elle sera toujours de $O(\ell)$ dans le pire des cas. Si aucune hiérarchie des annotations n'est donnée, la profondeur du WT sera de $H_0(A')$ (respectivement $H_0(D')$).

Ces deux idées sont regroupées dans l'algorithme 3 indiquant la manière de concevoir la forme du WT à partir d'un arbre n-aire représentant une famille d'annotations. Il fonctionne de manière similaire à un algorithme classique [9] mais utilise un algorithme de Huffman pour chaque nœud. Nous utilisons l'algorithme 3 sur la racine de l'arbre n-aire.

Algorithm 3 *FormeWRec*(n) : Transforme un arbre n-aire en arbre binaire, retourne la racine de l'arbre binaire

```
1 : Paramètre : n : nœud courant de l'arbre n-aire
2 : if estFeuille(n) then
3 :     création feuille' (n.anno, n.frequence) return feuille'
4 : else
5 :     création l                ▷ liste de nœuds, rangés dans l'ordre décroissant des fréquences
6 :     for chaque nœud fils f de n do
7 :         f' = FormeWRec(f)
8 :         insertionDansListe(l, f')
9 : n' = Huffman(l)                ▷ algorithme de Huffman sur les nœuds de l
10 : return n'
```

4.5 Expérimentations

4.5.1 Méthodologie

Les trois index ont été implémentés en C++. Nous avons utilisé la bibliothèque SDSL-Lite [15] pour construire les vecteurs de bits et le WT. Nous avons utilisé la bi-

Algorithm 4 *Huffman*(l_1) : Fabrique un arbre de Huffman à partir des nœuds stockés dans une liste triée

```
1 : Parametres : l_1 : liste de feuilles, triée en fonction de leur fréquence,
2 : l_2 = {}                                  ▷ file de nœuds internes, initialement vide
3 : while l_1.taille + l_2.taille > 1 do
4 :     f_1 = retirerMin(l_1, l_2)            ▷ retire le plus petit élément de l'une des deux listes
5 :     f_2 = retirerMin(l_1, l_2)
6 :     création n'()
7 :     n'.filsGauche = f_1
8 :     n'.filsDroit = f_2
9 :     n'.frequence = f_1.frequence + f_2.frequence
10 :    insérerDansFile(l_2, n')
11 : racine' = retirer(l)
12 : return racine'                           ▷ Complexité : O(ℓ log ℓ + n), avec ℓ nombre de feuilles
```

bliothèque RopeBWT2 [21], qui construit la BWT en temps $O(n \log n)$ sur de courtes séquences ADN. Elle est très efficace pour stocker et indexer des séquences correspondant à notre application [8]. Comme RopeBWT2 n'échantillonne pas la table des suffixes, nous itérons sur le texte jusqu'à trouver un symbole \$. Pour avoir des résultats proches de l'échantillonnage habituel du FM-index en $O(\log^{1+\varepsilon} n)$ pas nous utilisons des séquences de taille 50 dans les fichiers simulés. Les fichiers réels, ayant des séquences plus grandes, ont de ce fait une distance d'échantillonnage plus grande. L'exécution des fonctions utilisant la BWT seront plus longues pour un même résultat et ne pourront être comparées entre jeux de données réelles et simulées. Nous construisons les vecteurs de bits B_A (ou B_D), les compressons en utilisant la classe `rrr_vector` de SDSL-Lite (voir section 2.4.2), enfin nous construisons W_A (ou W_D) en utilisant une forme d'arbre dépendant de la hiérarchie des annotations (voir section 4.4) que nous avons intégrée dans SDSL-Lite. Dans ce prototype d'implémentation, les requêtes ne peuvent pas être utilisées sur les familles d'annotations.

Nous avons évalué le temps de construction, la taille des index, ainsi que le temps d'exécution de trois des requêtes détaillées dans la section 4.3 : *label*(), *findL*() et *findPL*(). En effet, *findP*(P) se comporte de la même manière dans chaque index et *countPL*(P, L_x) est très similaire à la requête *findPL*(P, L_x). Les trois index ont été testés sur plusieurs jeux de données de textes annotés, chacun composés de 100M lettres. La figure 1.4 montre la manière dont un texte annoté est représenté dans les fichiers. Les jeux de données ainsi que le code sont disponibles à l'adresse `http://www.vidjil.org/data/#2018-peerjcs`.

Les jeux de données sont les suivants :

— **fichiers simulés avec séquence et annotations aléatoires.** Les séquences sont aléatoires et ont une longueur allant de 40 à 60 lettres. Pour les annotations, nous lisons chaque lettre en tirant avec une probabilité de 1/2 si cette lettre est le début d'une annotation. Si c'est le cas, nous tirons sa longueur (à partir de 1 et ne dépassant pas la fin de la séquence) et enfin son annotation. La taille du vecteur A' (respectivement D') est $a \sim 0.06n$ (respectivement $d \sim 0.77n$).

— **fichiers simulés avec séquences aléatoires mais annotations fixées.** Ici, une annotation est toujours associée au même motif, avec des variations éventuelles, et nous modifions la proportion de lettres annotées (de 5% à 100%), le pourcentage de variations dans le motif de l'annotation (de 0% à 50%, plus de variation équivaut à avoir des motifs aléatoires), le nombre d'annotations différentes (de 10 à 1000), la longueur des annotations (de 5 à 100 lettres). Le jeu de données est composé de 546 fichiers, deux de ces fichiers sont présentés dans la table 4.2. Le premier fichier a des annotations présentes sur tous les lettres ($a \sim 0.06n$ et $d \sim 0.16n$), tandis que le second a beaucoup de variations dans les annotations ($a \sim 0.08n$ et $d \sim 0.36n$).

— **fichier réel composé de recombinaisons V(D)J.** Ce fichier est constitué de recombinaisons V(D)J, avec une séquence par clone trouvé, ainsi que leur analyse réalisée par Vidjil. Il est composé de la concaténation de trois fichiers patients publiés dans [45] afin d'obtenir un fichier d'environ 100M lettres. Ces fichiers ont été obtenus par séquençage à haut débit lors du diagnostic de patients atteints de leucémie et dont les échantillons sont analysés à l'hôpital de Lille. Le jeu de données est composé d'environ 400 000 séquences dont 838K facteurs annotés issus de 355 annotations différentes (avec $a \sim 0.016n$ et $d \sim 0.26n$). Les longueurs des annotations ont été introduites dans la figure 2.14 de la section 2.6. Ces jeux de données permettent d'envisager des requêtes réalistes, voir la section 4.6.

4.5.2 Résultats

Taille des index. La table 4.2 montre les résultats. Comme attendu, la taille de U, B (B_A ou B_D) et W (W_A ou W_D) augmente linéairement avec le nombre d'éléments à indexer (données non montrées). Le TL-index est l'index le plus petit, et le TL_{BW}-index est en général légèrement plus grand. La compression des index est directement liée à a et d. Les fichiers avec les séquences et annotations aléatoires ($d = 0.77n$)

	aléatoire			fixé #1			fixé #2			réel		
taille des séquences	50			50			50			264		
Anno. (t/u)	3.7M / 1000			4M / 100			4M / 10			852K / 355		
Anno. taille moy.	12.6			25			5			110.3		
Anno. lettres (%)	47			100			20			92		
Variations (%)	100			5			50			??		
$a = ... / d = ...$	$0.06n$ / $0.77n$			$0.06n$ / $0.16n$			$0.08n$ / $0.36n$			$0.016n$ / $0.26n$		
	TL	TL_{BW}	HT	TL	TL_{BW}	HT	TL	TL_{BW}	HT	TL	TL_{BW}	HT
Taille (MB)	**104**	184	217	**33**	38	114	**99**	116	209	**13**	33	35
Temps (s)	**18**	80	**15**	13.2	77	**11**	**14.6**	73.8	**13.2**	**10.2**	65	**10.2**
label(t_i) (μs)	3.84	21.54	**0.73**	4.50	20.02	**0.55**	1.22	21.2	**0.67**	2.98	81.1	**0.37**
findL(L) (μs/l)	**0.4**	34	**0.4**	**0.12**	13.1	0.20	0.40	21.41	**0.21**	**0.04**	52.5	0.17
findPL(P, L) (s)	34.7	**5.13**	29.7	26.0	**3.19**	20.9	30.64	**3.00**	28.91	131.1	**3.83**	127.8

TABLE 4.2 : Taille, temps de construction et temps d'exécution de requêtes des trois index indexant des textes annotés, sur trois fichiers simulés et un fichier réel de recombinaisons V(D)J. Chaque fichier est constitué de $n = 100$M lettres. Les fichiers diffèrent par leur nombre d'annotations ("Anno. (t/u)") et leur longueur ("Anno. taille moy."), par le pourcentage de lettres annotées ("Anno. lettres"), et par le pourcentage de variations dans le motif d'une annotation ("Variations"). La valeur a (respectivement d) représente la taille de A' (resp. D'). Elle est donnée par rapport à la taille du texte n. Les requêtes utilisent un motif P de 3 lettres. Les temps d'exécution des requêtes sont des moyennes calculées sur 1M lancements pour *label*(i) et au moins 5 exécutions pour les autres requêtes. Les temps d'exécution de la requête *findL*(L) sont calculés par lettre. Les meilleurs temps en gras.

sont compliqués à compresser, alors que les fichiers avec un ratio d/n bas donnent un quotient de compression allant de 2 à 7. La figure 4.11 détaille ces variations de tailles. Comme attendu, les index sont plus grands lorsque il y a peu d'annotations consécutives identiques dans T ou dans U, provoquant plus de bits 1 dans B. Notons que lorsqu'il y a plus de lettres annotées, il y a plus de répétitions dans le texte car les annotations sont placées sur des motifs similaires, d'où une diminution de la taille de la BWT (figure 4.11 haut gauche et bas gauche). W augmente lorsque le nombre d'annotations différentes augmente (en bas à droite), la profondeur de W augmente de façon logarithmique avec le nombre d'annotations différentes.

Le TL-index est plus petit que le TL_{BW}-index sur les fichiers réels, car moins d'annotations sont indexées dans le WT du TL-index ($a < d$). Sachant que le fichier réel est constitué de séquences de recombinaisons V(D)J, il y a au maximum 5 facteurs annotés successifs par séquence : `V`, ε, `D`, ε et `J`. Le TL-index compresse très bien les annotations de ces séquences. De son côté, le TL_{BW}-index compresse les annotations plus difficilement. Les recombinaisons étant toutes différentes, la BWT du TL_{BW}-index ne parvient pas à regrouper autant d'annotations successives identiques dans D que le

TL-index avec A. D' est très peu compressé par rapport à A', donc W_D est plus grand que W_A, le TL_{BW}-index est donc plus grand que le TL-index sur ce fichier de séquences réelles.

Temps de construction. La majorité du temps de construction du TL-index est consacrée à la construction de la BWT. Bien qu'un temps identique y soit consacré dans la construction du TL_{BW}-index, la construction de D' est très coûteuse. De ce fait, le TL_{BW}-index nécessite toujours plus de temps de construction que le TL-index.

Requêtes. La requête *label*() est la plus rapide dans le HT-index. Comme attendu, le TL_{BW}-index a besoin de plus de temps pour les requêtes *label*() et *findL*() car il a besoin de traduire les positions dans l'ordre du texte. Notons que localiser les positions dans le texte prend environ autant de temps que la requête $label(i)$ dans le TL-index (données non montrées). Cependant, pour la requête $findPL(P, L)$, le TL_{BW}-index est la solution la plus rapide car la traduction de position ne se fait que sur les lettres qui ont à la fois le motif et l'annotation. Pour le TL-index et HT-index, le temps effectif de la requête $findPL(P, L)$ est plus affecté par le nombre d'occurrences du motif que par le nombre d'occurrences finales (entre 0 et 100K en fonction du fichier).

Sur le fichier réel de recombinaisons V(D)J, les séquences sont plus longues. Le TL_{BW}-index souffre encore plus ici de l'échantillonnage distant dans l'implémentation pour les requêtes *label*() et *findL*(). Cependant, pour la requête $findPL(P, L)$, les autres index sont pénalisés par l'échantillonnage distant, entraînant des requêtes plus de 30 fois plus lentes avec un TL_{BW}-index.

4.6 Discussion

Les TL-index et TL_{BW}-index stockent un texte annoté et permettent des requêtes efficaces. Ils peuvent être construits depuis des fichiers de plusieurs MB en quelques secondes. Les tests confirment que les index restent petits lorsque le texte est redondant (donc un plus petit U), lorsque chaque annotation décrit un motif avec peu de variation (beaucoup de bits 0 dans B, donc un plus petit W_A ou W_D), et lorsque peu de lettres sont annotées (donc un plus petit W_A ou W_D). Toutefois le TL-index et le TL_{BW}-index sont robustes même pour des textes non redondants avec des annotations presque aléatoires. Le TL_{BW}-index a besoin de plus de place que le TL-index mais est plus efficace pour les requêtes combinant motif et annotation.

Nous avons vu dans la section 4.3 que les requêtes classiques ont un bon temps

d'exécution sur le TL-index, mais qu'elles sont plus longues sur le TL_{BW}-index, chaque occurrence devant être traduite dans l'ordre du texte. En revanche les requêtes difficiles sont efficaces et ont un temps d'exécution satisfaisant sur le TL_{BW}-index. Le temps d'exécution de la requête *countPL*()devient indépendant du nombre d'occurrences si le WT du TL_{BW}-index indexe D (les annotations dans l'ordre de la BWT) et non sa version compressée.

Nous ne comparons pas les taux de compression des index du chapitre 4 et de la compression du chapitre 3. L'algorithme de compression testé dans le chapitre 3 utilise en entrée un texte non annoté, nous ne comptons que le texte dans la taille du texte non compressé. De leur côté, les structures d'indexations indexent des textes annotés. Si nous avions voulu faire une comparaison, il nous aurait fallu implémenter le premier algorithme de compression, compressant un texte annoté.

Nous prévoyons d'utiliser l'un de ces index dans une **indexation de recombinaisons V(D)J** d'échantillons de patients suivis pour des maladies hématologiques. Il permettra la recherche de recombinaisons V(D)J entre plusieurs échantillons d'un patient ou plusieurs patients. Nous pourrons suivre l'évolution d'un clone chez un patient ou voir si deux patients partagent des clones. Nous pourrons aussi procéder à des tests de contamination afin de vérifier que le clone majoritaire présent dans un fichier ne se retrouve pas dans un autre. Pour cela nous pouvons exécuter la requête $findP(P)$, avec P étant le clone majoritaire, dans les fichiers. Nous pourrions également faire l'intersection de deux fichiers. Pour cela nous cherchons si l'une des séquences d'un premier fichier apparaît dans un second fichier avec la requête $findP(P)$ sur chaque séquence du premier fichier. Cette solution n'est pas optimale et pourrait être améliorée par des algorithmes comparant deux index.

Grâce à ces structures d'indexation, nous pourrons aussi comparer la situation de plusieurs patients dont la situation médicale est proche. Pour cela, prenons l'exemple d'un patient en attente de traitement ; nous savons quel clone nous devons traiter chez ce patient, les gènes qui le constituent ainsi que ses mutations génétiques. En utilisant la requête $findPL(P, L_x)$ nous trouvons les patients partageant des recombinaisons ayant ces même gènes mutés. Si le patient a été traité sur ce clone, nous connaissons le traitement effectué et la réponse immunitaire du patient pour mettre au point un traitement pour le nouveau patient. D'après les tests effectués, le temps de cette requête est tout à fait convenable pour cette application.

En ce qui concerne la requête $findL(L_x)$, nous pouvons imaginer créer une indexation de patients sains et trouver les recombinaisons contenant un gène spécifique. Cela

permettrait de faire des statistiques sur les taux d'apparition des recombinaisons et en apprendre plus sur le choix des gènes dans le mécanisme de recombinaisons.

Chacune de ces requêtes peut être utilisée sur différents niveaux d'une famille d'annotations. Nous pourrions en effet exécuter la requête *findPL*(P, V_x) sur tous les allèles du gène V_x car ils diffèrent très peu – certains allèles ne variant que d'un seul nucléotide.

Suivant le type de requêtes majoritaires que voudra faire l'utilisateur (motif ou annotation ou motif et annotation), nous choisirons le TL-index ou le TL_{BW}-index.

Dans chaque index, nous rajouterons à l'implémentation les requêtes sur des familles d'annotations mais aussi de nouvelles requêtes propres aux V(D)J. Par exemple, certaines séquences dans le fichier ne sont pas des recombinaisons V(D)J et nous pourrons y accéder et les compter facilement en ajoutant une annotation spéciale sur le symbole `$` de la séquence. Nous pourrons connaître le pourcentage de présence d'une recombinaison en faisant une recherche sur les annotations constituant cette recombinaison. D'autres requêtes, comme calculer la longueur moyenne des séquences, sont plus difficiles à exécuter car demandant une lecture intégrale du texte.

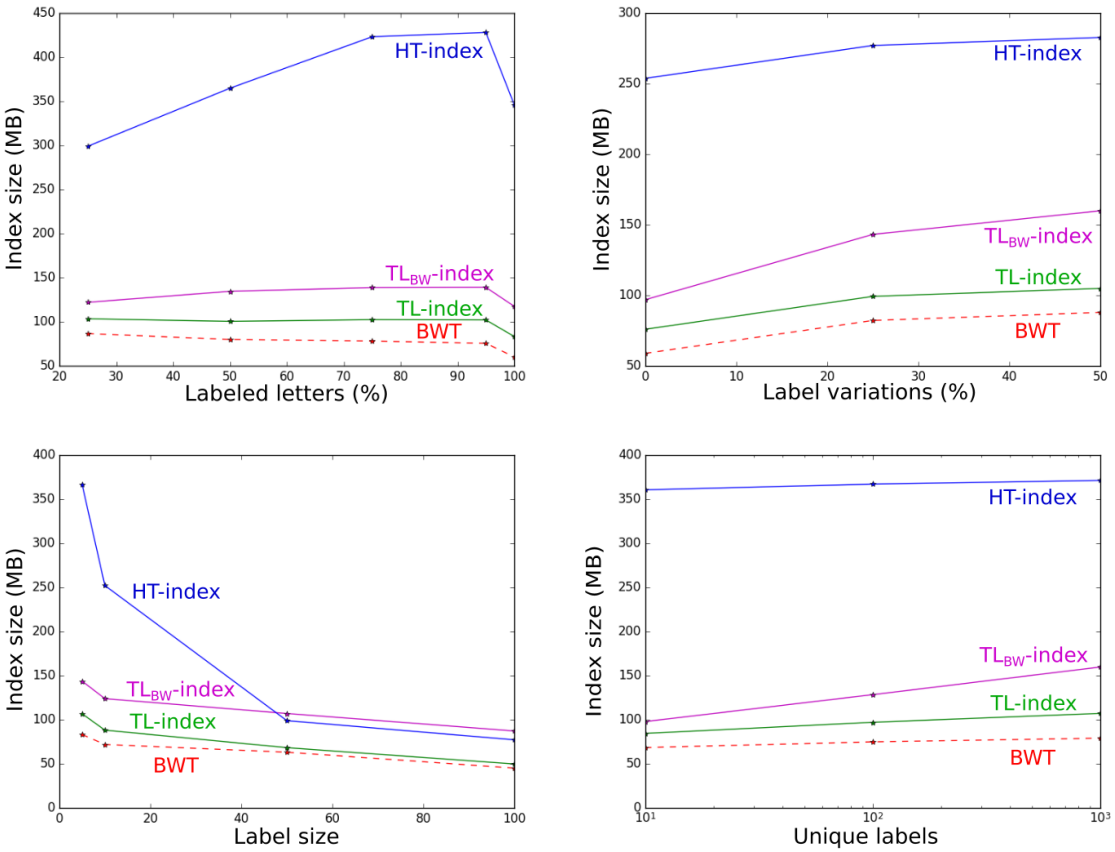


FIGURE 4.11 : Taille des index et de leur BWT sous-jacentes dans les 546 fichiers ayant les annotations fixées. Le HT-index est toujours très grand car A' est stocké en toutes lettres. Les TL- et $\mathrm{TL_{BW}}$-index ont une taille proche de leur BWT. Leur différences dépend surtout de la taille de la séquence d'annotations compressée A' et D'. Les tailles augmentent lorsque le nombre de lettres annotées augmente (haut gauche), lorsqu'il y a plus de variation dans les motifs des annotations (haut droite), ou lorsque le nombre d'annotations différentes augmente (bas droite). Notons que lorsque toutes les lettres sont annotées (haut gauche, 100%), il y a une petite diminution dans la taille des index car il n'y a plus de lettre aléatoire entre les motifs. La taille des index diminue lorsque la longueur des annotations augmente (bas gauche), car il y a un plus grand nombre de suffixes communs dans les séquences.

Conclusions et perspectives

A partir d'une problématique biologique posée par le service d'hématologie de Lille, *comment comparer les répertoires immunologiques de patients* sous forme d'ensemble de recombinaisons V(D)J, nous avons traduit ce problème dans le domaine de l'algorithmique du texte : *comment compresser et indexer un texte annoté tout en permettant des requêtes combinant texte et annotation.* À notre connaissance, il n'existait aucun algorithme ou structure répondant à ce problème. Nous avons donc élaboré une méthode de compression et deux structures d'indexation.

Le chapitre 3 introduit une compression de texte annoté. Nous découpons le texte en facteurs, qui sont représentés par des références vers des séquences connues ou sont décrits en toutes lettres. Ce chapitre propose trois algorithmes de compression. Le premier compresse un texte annoté où chaque section du texte annotée est découpée en facteurs pointant vers la même séquence annotée. Ce découpage n'est pas optimal sur le texte mais l'est dans toute section annotée du texte. Cet algorithme s'exécute au maximum en $O(e^4 \times n)$, avec $e \ll n$ le nombre maximum de variations sur une section annotée. Les deuxième et troisième algorithmes compressent et annotent un texte non annoté ayant besoin d'être annoté. Le deuxième algorithme génère un découpage optimal du texte, impliquant un temps d'exécution global de $O(n^3)$ avec n la taille du texte. Le dernier algorithme est plus rapide car glouton, s'exécute en temps $O(nb \log b)$ avec b le nombre de facteurs d'un découpage glouton, calculé en parcourant le texte. Puis nous essayons de modifier les plus grands facteurs.

Nous avons testé le dernier algorithme sur des données réelles : des fichiers de séquençage de répertoires immunologiques. Ces tests nous ont montré que la compression d'un texte peut être très variable : pour des fichiers de même taille, le facteur de compression peut aller de $0,95$ à 4. Ces variations s'expliquent par des pourcentages très variables de recombinaisons V(D)J dans le texte (les autres étant des séquences non recombinées). Enfin, bien que la compression soit intéressante, nous ne pouvons requêter facilement les données en n'ayant aucun accès à un facteur d'une séquence.

Ainsi, nous n'avons pas continué ces recherches et les algorithmes n'ont pas tous été prototypés. Nous avons choisi de privilégier un accès facile aux données via des requêtes d'intérêt en nous concentrant sur une structure d'indexation.

Le chapitre 4 présente trois structures d'indexation compressées, une naïve et deux utilisant une transformée de Burrows-Wheeler indexant les lettres du texte et un Wavelet Tree indexant les annotations. Ces deux index proposés diffèrent par l'ordre dans lequel les annotations sont organisées. Ils utilisent un espace qui est fonction de l'entropie du texte et des annotations. Ces structures autorisent les requêtes classiques : la recherche de motifs et d'annotations, tout comme un nouveau type de requêtes : la recherche combinant motif et annotation. Par construction, nous savons que le TL-index favorise les requêtes classiques alors que le TL_{BW}-index favorise les requêtes combinées. Les tests réalisés sur des fichiers aléatoires, simulés puis réels nous confirment ces hypothèses. Le TL-index est toujours plus petit que le TL_{BW}-index (jusqu'à $2,5$ fois) et les requêtes classiques sont de 4 à 100 fois plus rapides. En revanche, les requêtes combinées sont de 6 à 30 fois plus rapides dans le TL_{BW}-index.

L'une des structures d'indexation sera utilisée pour indexer des recombinaisons V(D)J, regroupant les analyses de répertoires immunologique d'un même hôpital. Cet index regroupera plusieurs milliers d'échantillons analysés pas Vidjil. Nous choisirons la structure la plus adaptée aux besoins de l'hôpital, suivant les requêtes prioritaires qu'il compte utiliser. Le prototype de test que j'ai réalisé pourrait être intégré pour l'indexation de recombinaisons V(D)J de patients. Nous pourrons ajouter quelques requêtes spécialisées pour ces séquences telles que *filtrer les séquences non recombinées* en ajoutant simplement une annotation spéciale ou *calculer la longueur moyenne des séquences*, requête plus difficile, demandant actuellement la lecture du texte intégral, comme expliqué dans la section 4.6.

La formalisation des recombinaisons V(D)J en texte annoté a permis de créer une compression et des structures de données qui n'utilisent pas les spécificités des recombinaisons. Ainsi nous pouvons utiliser ces méthodes sur tout texte annoté. Par exemple en indiquant la sémantique des mots d'un texte, nous pourrons trouver toutes les phrases contenant un mot et ayant un thème commun. Cela filtrerait les homonymes.

Lors de l'élaboration du sujet, nous avons mis de côté les *annotations chevauchantes*. Il serait intéressant de savoir comment les compresser et les indexer. Prenons l'exemple d'un shiritori (jeu de mot consistant à effectuer une suite de mots, dans laquelle chaque mot commence par la même syllabe qui finit le mot précédent) : `basse-courtisane` est constitué des mots `basse-cour` et `courtisane`. Dans un shiritori, des

annotations chevauchantes pourraient indiquer les mots le composant. Pour indexer ce texte, nous pouvons transformer ces annotations afin de supprimer les chevauchements. Une annotation positionnée sur toutes les lettres d'un facteur devient une annotation sur la première lettre du facteur et une annotation sur la dernière lettre du facteur. Malheureusement cela double le nombre d'annotations à indexer, les requêtes bidimensionnelles deviennent très compliquées, et nous pouvons avoir des cas particuliers lorsque deux annotations chevauchantes commencent ou finissent au même endroit : des annotations sont toujours chevauchantes. Pour compresser ce type de séquences, nous pourrions ajouter un élément indiquant à quel endroit, par rapport à la fin de la séquence déjà emboitée, débute le prochain facteur.

Nous avons proposé des éléments de réponse sur la compression et l'indexation de texte à deux dimensions (le texte et les annotations), mais nous pourrions réfléchir à *comment stocker un texte à trois dimensions* ou plus généralement, *à n dimensions*. Les dimensions deux et supérieures seraient des annotations appartenant à des alphabets différents. Prenons l'exemple d'une séquence ADN. La séquence serait la première dimension, les noms des gènes seraient des annotations sur une deuxième dimension et la qualité de chaque nucléotide serait une troisième dimension. Pour ces textes, nous pourrions ajouter une troisième structure, WT ou autre, indexant la troisième dimension. La conception, l'implémentation et l'évaluation de tels index liant textes et annotations pourraient ainsi constituer de nouvelles pistes de recherche pour des applications en bioinformatique, ou, plus généralement, pour d'autres applications sur des séquences annotées.

Bibliography

[1] R. Baeza-Yates et G. H. Gonnet : A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[2] U. Baier : Linear-time suffix sorting-a new approach for suffix array construction. *In Leibniz International Proceedings in Informatics (LIPIcs 2016)*, vol. 54. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[3] D. Belazzougui, F. Cunial, J. Kärkkäinen et V. Mäkinen : Linear-time string indexing and analysis in small space. *arXiv preprint arXiv:1609.06378*, 2016.

[4] D. A. Bolotin, S. Poslavsky, I. Mitrophanov, M. Shugay, I. Z. Mamedov, E. V. Putintseva et D. M. Chudakov : MiXCR: software for comprehensive adaptive immunity profiling. *Nature Methods*, 12(5):380–381, 2015.

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

[6] X. Brochet, M.-P. Lefranc et V. Giudicelli : IMGT/V-QUEST: the highly customized and integrated system for IG and TR standardized V-J and V-D-J sequence analysis. *Nucleic Acids Research*, 36(S2):W503–W508, 2008.

[7] M. Burrows et D. J. Wheeler : A block-sorting lossless data compression algorithm. *Digital Equipment Corporation*, 1994.

[8] A. J. Cox, M. J. Bauer, T. Jakobi et G. Rosone : Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform. *Bioinformatics*, 28(11):1415–1419, 2012.

[9] R. Dechter : Decomposing an n-ary relation into a tree of binary relations. *In Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, p. 185–189. ACM, 1987.

[10] M. Duez, M. Giraud, R. Herbert, T. Rocher, M. Salson et F. Thonier : Vidjil: A web platform for analysis of high-throughput repertoire sequencing. *PLOS One*, 11(11):e0166126, 2016.

[11] P. Ferragina et G. Manzini : Opportunistic data structures with applications. *In Foundations of Computer Science (FOCS 2000)*, p. 390–398. IEEE, 2000.

[12] P. Ferragina, G. Manzini, V. Mäkinen et G. Navarro : An alphabet-friendly FM-index. *In String Processing and Information Retrieval (SPIRE 2004)*, p. 150–160. Springer, 2004.

[13] M. J. Fischer et M. S. Paterson : String-matching and other products. Rap. tech., Massachusetts Inst Of Tech Cambridge Project Mac, 1974.

[14] M. Giraud : Compter les globules blancs, analyser les partitions, 2016. Habilitation à diriger les recherches.

[15] S. Gog, T. Beller, A. Moffat et M. Petri : From theory to practice: Plug and play with succinct data structures. *In Symposium on Experimental and Efficient Algorithms (SEA2014)*, p. 326–337, 2014.

[16] R. Grossi, A. Gupta et J. Vitter : High-order entropy-compressed text indexes. *In Symposium on Discrete Algorithms (SODA 2003)*, 2003.

[17] R. Grossi, A. Gupta et J. S. Vitter : When indexing equals compression: Experiments with compressing suffix arrays and applications. *In Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, p. 636–645. Society for Industrial and Applied Mathematics, 2004.

[18] D. A. Huffman : A method for the construction of minimum-redundancy codes. *Institute of Radio Engineers (IRE 1952)*, 40(9):1098–1101, 1952.

[19] D. Knuth, J. H. Morris et V. Pratt : Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[20] A. Lempel et J. Ziv : On the complexity of finite sequences. *IEEE Transactions on information theory*, 22(1):75–81, 1976.

[21] H. Li : Fast construction of fm-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014.

[22] H. Li et R. Durbin : Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[23] V. Mäkinen et G. Navarro : Run-length FM-index. *In Proc. DIMACS Workshop:The Burrows-Wheeler Transform: Ten Years Later*, 2004.

[24] V. Mäkinen et G. Navarro : Succinct suffix arrays based on run-length encoding. *In Symposium on Combinatorial Pattern Matching (CPM 2005)*, p. 45–56, 2005.

[25] V. Mäkinen et G. Navarro : Implicit compression boosting with applications to self-indexing. *In String Processing and Information Retrieval (SPIRE 2007)*, p. 229–241. Springer, 2007.

[26] V. Mäkinen et G. Navarro : Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

[27] D. Male : *Immunologie: aide-mémoire illustré*. De Boeck Supérieur, 2005.

[28] G. Mathé, M. Hayat, L. Schwarzenberg, J. Amiel, M. Schneider, A. Cattan, J. Schlumberger et C. Jasmin : Haute fréquence et qualité des rémissions de la leucémie aiguë lymphoblastique chez l'enfant. *Arch. Franc. Ped*, 25:181–188, 1968.

[29] E. M. McCreight : A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

[30] T. Mora et A. M. Walczak : Quantifying lymphocyte receptor diversity. *bioRxiv*, p. 046870, 2016.

[31] J. I. Munro, G. Navarro et Y. Nekrich : Space-efficient construction of compressed indexes in deterministic linear time. *In ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, p. 408–424. SIAM, 2017.

[32] J. I. Munro, Y. Nekrich et J. S. Vitter : Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016.

[33] G. Navarro : Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[34] G. Navarro et V. Mäkinen : Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007.

[35] S. B. Needleman et C. D. Wunsch : A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[36] R. Nussinov, G. Pieczenik, J. R. Griggs et D. J. Kleitman : Algorithms for loop matchings. *SIAM Journal on Applied Mathematics*, 35(1):68–82, 1978.

[37] R. Pagh : Low redundancy in static dictionaries with o (1) worst case lookup time. *In International Colloquium on Automata, Languages, and Programming (ICA2LP 1999)*, p. 595–604. Springer, 1999.

[38] R. Raman, V. Raman et S. S. Rao : Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *In ACM-SIAM Symposium on Discrete algorithms (SODA 2002)*, p. 233–242. Society for Industrial and Applied Mathematics, 2002.

[39] T. Rocher, M. Giraud et M. Salson : Indexer des séquences annotées. *In SeqBio*, 2016. (résumé étendu).

[40] T. Rocher, M. Giraud et M. Salson : Indexer un ensemble de séquences ADN annotées. *In Journées Ouvertes en Biologie, Informatique et Mathématiques (JOBIM 2016)*, 2016. (résumé étendu).

[41] T. Rocher, M. Giraud et M. Salson : Indexing labeled sequences. *In London Stringology Days & London Algorithmic Workshop (LSD & LAW 2017)*, 2017. (résumé).

[42] T. Rocher, M. Giraud et M. Salson : Indexing labeled sequences. *PeerJ Computer Science*, (4:e148), 2018.

[43] T. Rocher, P. Marquet, M. Pupin et Y. Secq : How to make teenage girls love coding? *WomENcourage*, 2017. (présentation de poster).

[44] K. Sadakane : Succinct representations of LCP information and improvements in the compressed suffix arrays. *In ACM-SIAM Symposium on Discrete algorithms (SODA 2002)*, p. 225–232. Society for Industrial and Applied Mathematics, 2002.

[45] M. Salson, M. Giraud, A. Caillault, N. Grardel, N. Duployez, Y. Ferret, M. Duez, R. Herbert, T. Rocher, S. Sebda *et al.* : High-throughput sequencing in acute lymphoblastic leukemia: Follow-up of minimal residual disease and emergence of new clones. *Leukemia research*, 53:1–7, 2017.

[46] J. Seward : bzip2 and libbzip2. *avaliable at http://www. bzip. org*, 1996.

[47] T. F. Smith et M. S. Waterman : Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[48] S. Tonegawa : Somatic generation of antibody diversity. *Nature*, 302(5909): 575–581, 1983.

[49] E. Ukkonen : On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[50] J. Van Dongen, A. Langerak, M. Brüggemann, P. Evans, M. Hummel, F. Lavender, E. Delabesse, F. Davi, E. Schuuring, R. García-Sanz *et al.* : Design and standardization of PCR primers and protocols for detection of clonal immunoglobulin and T-cell receptor gene recombinations in suspect lymphoproliferations: report of the BIOMED-2 concerted action BMH4-CT98-3936. *Leukemia*, 17(12):2257–2317, 2003.

[51] P. Weiner : Linear pattern matching algorithms. *In Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, p. 1–11. IEEE, 1973.