Algorithme de recherche incrémentale d'un motif dans un ensemble de séquences d'ADN issues de séquençages à haut débit

Nadia Ben Nsira

▶ **To cite this version:**

Nadia Ben Nsira. Algorithme de recherche incrémentale d'un motif dans un ensemble de séquences d'ADN issues de séquençages à haut débit. Autre [cs.OH]. Normandie Université; Université de Tunis El Manar, 2017. Français. NNT : 2017NORMR143 . tel-01818085

HAL Id: tel-01818085
https://theses.hal.science/tel-01818085

Submitted on 18 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Normandie Université

THÈSE EN CO- TUTELLE INTERNATIONALE

Pour obtenir le diplôme de doctorat

Spécialité INFORMATIQUE

Préparée au sein de Université de Rouen Normandie
et de Université de Tunis El Manar

Titre de la thèse

Algorithmes de recherche incrémentale d'un motif dans un ensemble de données issues de séquençage à haut débit

Présentée et soutenue par
Nadia BEN NSIRA

Thèse soutenue publiquement le 05/12/2017 devant le jury composé de		
Mme / Annie CHATEAU	Maître de conférences, HDR, Université de Montpellier 2	Rapporteur
M / Yahya SLIMANI	Professeur, Université de la Manouba	Rapporteur
M / Mourad ELLOUMI	Professeur, Université de Tunis El Manar	Directeur de thèse
M / Thierry LECROQ	Professeur, Université de Rouen Normandie	Directeur de thèse
Mme/ Élise PRIEUR	Maître de conférences, Université de Rouen Normandie	Examinateur

Thèse dirigée par

M. **Thierry LECROQ** Université de Rouen Normandie
M. **Mourad ELLOUMI** Université de Tunis El Manar

UNIVERSITÉ DE ROUEN NORMANDIE

MIIS

Laboratoire d'Informatique, de Traitement de l'Information et des Systèmes
litis

كلية العلوم بتونس
FACULTE DES SCIENCES DE TUNIS

Résumé

Dans cette thèse, nous nous intéressons au problème de *recherche incrémentale de motifs* dans des séquences fortement similaires (*On-line Pattern Matching on Highly Similar Sequences*), issues de technologies de séquençage à haut débit (SHD). Ces séquences ne diffèrent que par de très petites quantités de variations et présentent un niveau de similarité très élevé. Il y a donc un fort besoin d'algorithmes efficaces pour effectuer la recherche rapide de motifs dans de tels ensembles de séquences spécifiques. Nous développons de nouveaux algorithmes pour traiter ce problème. Cette thèse est répartie en cinq parties. Dans la première partie, nous présentons un état de l'art sur les algorithmes les plus connus du problème de recherche de motifs et les index associés. Puis, dans les trois parties suivantes, nous développons trois algorithmes directement dédiés à la recherche incrémentale de motifs dans un ensemble de séquences fortement similaires. Enfin, dans la cinquième partie, nous effectuons une étude expérimentale sur ces algorithmes. Cette étude a montré que nos algorithmes sont efficaces en pratique en terme de temps de calcul.

Mots-clés : Algorithmes, structure d'indexation, recherche incrémentale, séquençage à haut débit, séquences d'ADN, compression selon la référence, complexités.

Algorithms of on-line pattern matching in a set of highly sequences outcoming from next sequencing generation

---

Abstract

In this thesis, we are interested in the problem of *on-line pattern matching* in highly similar sequences, *On-line Pattern Matching on Highly Similar Sequences*, outcoming from Next Generation Sequencing technologies (NGS). These sequences only differ by a very small amount. There is thus a strong need for efficient algorithms for performing fast pattern matching in such specific sets of sequences. We develop new algorithms to process this problem. This thesis is partitioned into five parts. In the first part, we present a state of the art on the most popular algorithms of finding problem and the related indexes. Then, in the three following parts, we develop three algorithms directly dedicated to the on-line search for patterns in a set of highly similar sequences. Finally, in the fifth part, we conduct an experimental study on these algorithms. This study shows that our algorithms are efficient in practice in terms of computation time.

Keywords : Algorithms, indexes, on-line search, next generation Sequencing, DNA sequences, based-reference-compression, complexities.

Remerciements

Tout d'abord, je tiens à adresser toute ma gratitude à mon directeur de thèse M. Thierry Lecroq, Professeur à l'Université de Rouen. Je le remercie de m'avoir conseillée, orienté et aidé à faire avancer mes travaux de recherche. Sa disponibilité, ses conseils et ses réponses bien précises m'ont permis de me familiariser avec le domaine de la Bioinformatique. Il m'a appris à bien concevoir, développer mes idées et à rédiger une publication scientifique judicieuse. Il n'a fait que nourrir mes réflexions.

Je suis également reconnaissante à M. Mourad Elloumi, Professeur à la Faculté des Sciences économiques et de Gestion de Tunis pour la confiance qu'il m'a accordée en acceptant d'encadrer ma thèse. Je le remercie pour sa disponibilité même les soirs et pendant ses vacances. Il m'a guidé dans la rédaction de mon manuscrit de thèse. Sa patience et ses critiques m'ont été d'un grand apport.

Je voudrais exprimer mes remerciements les plus cordiaux à M. Yahya Slimani, Professeur à l'Université de Tunis El Manar et à Mme. Annie Chateau, Maître de conférences HDR à l'Université de Montpellier, pour l'honneur qu'ils m'ont accordée de rapporter cette thèse et pour le temps qu'ils ont consacré pour lire et juger mon manuscrit de thèse.

Mes vifs remerciements s'adressent à M. Thierry Pacquet, Professeur à l'Université de Rouen, pour m'avoir acceptée dans le laboratoire de Recherche LITIS, qu'il dirige. Je remercie M. Mohammed Jemmni, Professeur à l'Université de Tunis, l'ancien directeur du laboratoire LATICE et également la nouvelle directrice Mme. Souad Chebbi, Professeur à l'Université de Tunis.

Je souhaite remercier particulièrement Martine Léonard, Maître de conférences à l'Université de Rouen, de m'avoir donnée ma première chance d'enseigner en tant qu' enseignante vacataire. Je n'oublierai jamais son soutien et ses conseils en matière d'enseignement et le temps qu'on a passé ensemble à préparer les cours.

Je remercie aussi toute l'équipe TIBS avec qui j'ai passé toutes ces années en thèse, notamment Laurent Mouchard, Arnaud Lefebvre, Élise Prieur, Hélène Dauchel et Caroline Berard.

Je désire aussi remercier tous les enseignants du département d'informatique de l'Université de Rouen, Christophe Hancart, Yannich Guesnet, Philippe Andarry, Olivier Mallet, Jean-Philippe Dubernard et les autres.

Je remercie Fabienne Boquet, secrétaire au LITIS, pour son aide dans les démarches administratives. Et également, je remercie Brigitte Diarra, et toutes les secrétaires qui ont travaillé sur le dossier de ma soutenance.

Aucun remerciement n'est assez éloquent pour exprimer ce que mérite ma chère mère Fatma pour ses sacrifices, son support moral, ses encouragements et ses prières pour moi. Je rends hommage à mon cher père Abdel Majid qui est parti avant de partager avec moi

mon succès et ma joie. Ce modeste travail est un témoignage de mon amour, mes pensées et de reconnaissance des sacrifices qu'il a toujours consentis depuis ma naissance.

Puis, je remercie mes sœurs Imen, Ines et Khaoula pour tout ce qu'elles ont fait pour moi. Je n'oublierai pas de remercier ma grande-mère, mes oncles, mes tantes, mes cousins et mes cousines. Je remercie très spécialement mon oncle Ridha qui a été toujours là pour moi pendant les moments les plus difficiles.

Je remercie tous mes amis que j'aime tant pour leur confiance en moi.

Et enfin, Iyed, mon petit fils, je te remercie tout simplement d'être là et d'avoir donné sens à ma vie.

Pour tous ces intervenants et pour toute personne qui m'a soutenue pendant ma thèse, je présente mes sincères remerciements, mon respect et ma gratitude.

Table des matières

Résumé / Abstract **i**

Liste des figures **ix**

Liste des tables **xiii**

Liste des algorithmes **xv**

Abréviations **xvii**

Introduction Générale **1**

1 Préliminaires **5**
1.1 Introduction 5
1.2 Définitions et Notations 5
1.3 Recherche de Motifs 6
1.3.1 Problème de recherche exacte 6
1.3.2 Problème de recherche distribuée 8
1.4 Compressibilité 8
1.5 Recherche du Minimum dans un Intervalle 9
9
10
11
1.6 Formalisation du Problème de Recherche Incrémentale d'un motif dans un ensemble de séquences fortement similaires 12

2 État de l'Art sur les Algorithmes de Recherche de Motifs dans un Index Représentant un Ensemble de Séquences **13**
2.1 Algorithmes de Recherche de Motifs dans un Index Classique 13
2.1.1 Préliminaires 14
2.1.2 Algorithmes de Recherche de Motifs dans un Trie 14
2.1.3 Algorithmes de Recherche de Motifs dans un Trie de Suffixes 15
2.1.4 Algorithmes de Recherche de Motifs dans un Arbre de Suffixes . . . 16

2.1.5 Algorithmes de Recherche de Motifs dans un Automate de Suffixes . 18
2.1.6 Algorithme de Recherche de Motifs dans un Automate Compact de Suffixes . . . 19
2.1.7 Algorithmes de Recherche de Motifs dans un Tableau de Suffixes . . 20
2.2 Algorithmes de Recherche de Motifs dans un Index Compressé . . . 22
2.2.1 Algorithmes de Recherche de Motifs dans un FM-index . . . 22
2.2.2 Algorithmes de Recherche de Motifs dans un Tableau de Suffixes Compressé . . . 25
2.2.3 Algorithmes de Recherche de Motifs dans un LZ-index . . . 27
2.3 Algorithmes de Recherche de Motifs dans un Index Avancé . . . 28
2.3.1 Algorithmes de Recherche de Motifs dans un Index Basé sur les Segments Communs et les Segments Différenciés . . . 28
29
32
2.3.2 Algorithmes de Recherche de Motifs dans un Auto-index Basé sur la Compression Lempel-Ziv Relative . . . 33
2.3.3 Algorithmes de Recherche de Motifs dans l'Index BIO-FMI . . . 37
2.3.4 Algorithmes de Recherche de Motifs dans un Arbre de Suffixes d'alignement . . . 39
2.3.5 Algorithme de Recherche de Motifs dans un Tableau de Suffixes d'Alignement . . . 41
2.4 Algorithme de Recherche Incrémentale de Motifs dans une Structure Arborescente . . . 42
2.5 Conclusion . . . 44

3 MPE : Algorithme de Morris-Pratt Étendu 47
3.1 MP : Algorithme de Morris-Pratt . . . 47
3.1.1 Préliminaires . . . 47
3.1.2 Description . . . 48
3.2 MPE : Algorithme de Morris-Pratt Étendu . . . 48
3.2.1 Préliminaires . . . 49
3.2.2 Description . . . 49
3.3 Complexités . . . 58
3.4 Exemple Illustratif . . . 59
3.5 Conclusion . . . 61

4 KMPE : Algorithme de Knuth-Morris-Pratt Étendu 63
4.1 KMP : Algorithme de Knuth-Morris-Pratt . . . 63
4.1.1 Préliminaires . . . 63
4.1.2 Description . . . 64
4.2 KMPE : Algorithme de Knuth-Morris-Pratt Étendu . . . 65
4.2.1 Préliminaires . . . 65
4.2.2 Description . . . 66
4.3 Complexités . . . 71

4.4 Exemple Illustratif 72
4.5 Conclusion 75

5 FSE : Algorithme de Recherche Rapide Étendu 77
5.1 FS : Algorithme de Recherche Rapide 77
5.1.1 Préliminaires 77
5.1.2 Description 79
5.2 FSE : Algorithme de Recherche Rapide Étendu 80
5.2.1 Préliminaires 80
5.2.2 Description 82
5.3 Complexités 86
5.4 Exemple Illustratif 87
5.5 Conclusion 88

6 Étude Expérimentale 89
6.1 Résultats de Comparaison avec un Algorithme de Recherche Incrémentale Classique 90
6.1.1 Données pseudo-aléatoires 90
6.1.2 Données réelles 90
6.1.3 Méthodologie 91
6.1.4 Évaluation en fonction du nombre de séquences 91
6.2 Évaluation de l'algorithme FSE en fonction des longueurs des motifs 94
6.3 Résultats de Comparaison avec un Algorithme de Recherche Incrémentale dédiées aux Séquences Fortement Similaires 96
6.3.1 Méthodologie et jeu de données 96
6.4 Résultats de Comparaison entre des Méthodes De Calcul de Tableax De Préfixes 98
6.5 Conclusion 100

Conclusion Générale 103

Liste des publications 105

Références 107

Liste des figures

1.1 Principe d'utilisation d'une fenêtre glissante pour faire de la recherche exacte d'un motif x dans une séquence y. 8

2.1 Trie associé à l'ensemble de séquences $Y = \{\texttt{C\$, CTAG\$, GTTG\$, GTAGTTAG\$}\}$. Afin d'éviter que certaines séquences de Y correspondent aux préfixes d'autres séquences de Y, on ajoute à la fin des séquences un caractère spécial (qui n'appartient pas à l'alphabet), appelé *terminateur*, noté \$. Ceci permet d'éviter que certaines séquences se terminent dans des nœuds internes du trie. Ainsi, chaque feuille représente une séquence distincte de Y et $\mathit{Trie}(Y)$ contient exactement r feuilles. En général le nombre de nœuds issus d'un nœud interne peut être aussi grand que la taille de l'alphabet. 15

2.2 Arbre des suffixes de $y = \texttt{CTAGTTAG\$}$. La racine représente ε. Les cercles représentent les nœuds internes. Ces derniers représentent les facteurs de y. Les carrés représentent les feuilles. Elles sont numérotées par les positions des débuts des suffixes qu'elles représentent. 17

2.3 Relations entre trie de suffixes, arbre de suffixes, DAWG et CDAWG. 19

2.4 Tableau de suffixes de la séquence $y = \texttt{CTAGTTAG\$}$. Comme nous supposons que $y[n-1] = \texttt{\$}$ est le plus petit caractère, nous avons toujours $SA_y[0] = n-1$. 21

2.6 Étapes de construction de $SA_{k_{\min}(y)=SA_2(y)}$. 26

2.7 Constructions de $\mathit{HRBI}(Y)$ pour le modèle 1. 30

2.8 Factorisation RLZ de $Y' = \{y_1, y_2, y_3\}$ selon y_0 (en dessus). Les facteurs distincts (les plus longs) sont mémorisés suivant l'ordre lexicographique dans le tableau y_F (en dessous) comme couple de positions de début et de fin dans y_0. 34

3.1 Décalage de la fenêtre glissante avec l'algorithme MP. 48

3.2 Relation de calcul de $\mathit{pref}_x^{\sim}[i+1]$ à partir de $\mathit{pref}_x^{\sim}[i]$: $(i_0, \ell_0), (i_1, \ell_1) \in \mathit{pref}_x^{\sim}[i] \mid x[i_0+1] = x[i+1]$ et $x[i_1+1] \neq x[i+1]$, alors $(i_0+1, \ell_0+1) \in \mathit{pref}_x^{\sim}[i+1]$. 51

3.3 Les tableaux *last* et *prev* pour $x = \texttt{ATAATAGACAATAC}$ et $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. 51

3.4 Situation de la Proposition 1 : v est le plus long bord à distance de Hamming 0 de $x[0\,..\,i]$ et w est le plus long bord à distance de Hamming 1. 52

3.5 Exemple illustrant le résultat de la Proposition 1 : ATA est le plus long bord à distance de Hamming 0 de ATAATAGACAATA et ATAATA est le plus long bord à distance de Hamming 1. 53

3.6 53

3.7 Décalage dans le Cas 1. Le préfixe $u = x[0\,..\,i]$ est reconnu dans y_0 et y_1 (n'ayant pas une variation dans la fenêtre en cours par rapport à y_0). Une inégalité est trouvée à la position $i + j + 1$, $b \neq a$. Après décalage de la fenêtre en utilisant *mpNext*, on reprend les comparaisons entre $x[mpNext[i]]$ et $y_0[i + j + 1]$. 55

3.8 Décalage dans le Cas 2. Le préfixe $u = x[0\,..\,i]$ est reconnu dans y_0 et non pas dans y_1 (ayant une variation b dans la fenêtre en cours par rapport à y_0). On a $b = y_0[i + j + 1] = y_1[i + j + 1] \neq x[i] = a$. Alors un décalage doit être effectué en utilisant B. On passe au Cas 1 si après décalage la fenêtre dépasse la variation b dans y_1. On reste dans le Cas 2 si après décalage la fenêtre contient la variation b dans y_1 mais le préfixe aligné est reconnu dans y_0. On passe au Cas 3 si après décalage la fenêtre contient la variation b dans y_1 et celle-ci permet de reconnaitre le préfixe dans y_1. 56

3.9 Décalage dans le Cas 3. Le préfixe $u = x[0\,..\,i]$ est reconnu dans y_1 et non pas dans y_0. On a $b = y_1[i + j + 1] = x[i] \neq y_0[i + j + 1]$. Alors un décalage doit être effectué en utilisant B. On passe au Cas 1 si après décalage la fenêtre dépasse la variation b dans y_1. On passe au Cas 2 si après décalage la fenêtre contient la variation b dans y_1 mais le préfixe aligné est reconnu dans y_0. On reste dans le Cas 3 si après décalage la fenêtre contient la variation b dans y_1 et celle-ci permet de reconnaitre le préfixe dans y_1. 56

4.1 $kmp_x^0[i] = \max\{j \mid x[0\,..\,j-1]$ est un bord de $x[0\,..\,i-1]$ et $x[j] \neq x[i]\} \cup \{-1\}$. 64

4.2 $kmp_x^0[i + pref_x^0[i]] \geq pref_x^0[i]$. 64

4.3 Décalage avec l'algorithme KMP (v est un bord de u et $c \neq b$). 65

4.4 $kmp_x^1[i] = \{(k, j) \mid x[0\,..\,j-1]$ est un bord à distance de Hamming 1 de $x[0\,..\,i-1]$, $x[k] \neq x[i - j + k]$ et $x[j] \neq x[i]\}$. 65

4.5 $(pref_x^0[i], pref_x^0[i] + 1 + \ell) \in kmp_x^1[i + pref_x^0[i] + 1 + \ell]$ où $\ell = |lcp(x[pref_x^0[i] + 1], x[i + pref_x^0[i] + 1])|$. 66

4.6 Décalage dans le Cas 1 avec $c \neq b$. Le préfixe $u = x[0\,..\,i]$ est reconnu dans y_0 et y_1 (n'ayant pas de variation dans la fenêtre en cours par rapport à y_0). Une inégalité est trouvée à la position $i + j + 1$, $b \neq a$. Un décalage est effectué alors en utilisant $kmp_x^0[i]$ qui correspond à la longueur du plus long bord v de $x[0\,..\,i]$ suivi par $c \neq b$. 70

4.7 Décalage dans le Cas 2 (v est un bord de u et $c \neq b$). Le préfixe $u = x[0\,..\,i]$ est reconnu dans y_0 et non pas dans y_1 (ayant une variation e dans la fenêtre en cours par rapport à y_0). On a $b = y_0[i + j + 1] = y_1[i + j + 1] \neq x[i] = a$. Alors un décalage doit être effectué en utilisant kmp_x^1. On passe au Cas 1 si après décalage la fenêtre dépasse la variation e dans y_1. On reste dans le Cas 2 si après décalage la fenêtre contient la variation e dans y_1 mais le préfixe aligné est reconnu dans y_0. On passe au Cas 3 si après décalage la fenêtre contient la variation e dans y_1 et celle-ci permet de reconnaitre le préfixe dans y_1. 71

4.8 Décalage dans le Cas 3 (v est un bord de u et $c \neq b$). Le préfixe $u = x[0\,..\,i]$ est reconnu dans y_1 et non pas dans y_0. On a $b = y_1[i+j+1] = y_0[i+j+1] \neq x[i] = a$. Alors un décalage doit être effectué en utilisant kmp_x^1. On passe au Cas 1 si après décalage la fenêtre dépasse la variation e dans y_1. On passe au Cas 2 si après décalage la fenêtre contient la variation e dans y_1 mais le préfixe aligné est reconnu dans y_0. On reste dans le Cas 3 si après décalage la fenêtre contient la variation e dans y_1 et celle-ci permet de reconnaitre le préfixe dans y_1. 72

5.1 Si $0 < s \leq i+1$ alors $x[i+1-s\,..\,m-1-s] = x[i+1\,..\,m-1]$ et si $0 < s \leq i$ alors $x[i-s]] \neq x[i]$. 78
5.2 Si $i+1 < s$ alors $x[0\,..\,m-1-s] = x[s\,..\,m-1]$. 78
5.3 $\mathit{suff}_x^1[i] = \ell + 1 + \mathit{lcsuff}(x[0\,..\,i-\ell-1], x[0\,..\,m-\ell-2])$ où $\ell = \mathit{lcsuff}(x[0\,..\,i], x])$. 80
5.4 Si $0 < s \leq i+1$ alors $\mathit{Ham}(x[i+1-s\,..\,m-1-s], x[i+1\,..\,m-1]) = 1$ et si $0 < s \leq i$ alors $x[i-s]] \neq x[i]$. 81
5.5 Si $i+1 < s$ alors $x[0\,..\,m-1-s] = x[s\,..\,m-1]$. 81

6.1 Résultats expérimentaux avec 100 motifs de longueur 8. 92
6.2 Résultats expérimentaux avec 100 motifs de longueur 16. 93
6.3 Résultats expérimentaux avec 100 motifs de longueur 64. 94
6.4 Résultats expérimentaux avec 100 motifs de longueur 128. 95
6.5 Résultats éxpérimentaux de l'algorithme FSE avec 6 séquences pseudo-aléatoires. 96
6.6 100

Liste des tables

2.1 Table illustrant certaines complexités (consommations mémoires et temps de recherches) : n est la longueur des séquences d'entrée, n' et N' sont respectivement le nombre des segments communs et différenciés et N_{psc} est le nombre des préfixes d'un motif donné qui sont des suffixes de certaines parties communes (Huang et al.), N_F est le nombre des facteurs de la séquence référence utilisés pour compresser les autres séquences de l'ensemble d'entrée (RLZ). Le nombre d'occurrences du motif est noté par *occ.* 45

6.1 Résultats obtenus avec 100 motifs de longueur $m = 32$ 97
6.2 Résultats obtenus avec 100 motifs de longueur $m = 64$ 97
6.3 Résultats obtenus avec 100 motifs de longueur $m = 128$ 97
6.4 Résultats obtenus avec 100 motifs de longueur $m = 265$ 98

Liste des algorithmes

3.1 *ComputePrev* 52
3.2 *ComputeB* 54
3.3 *MPE_Search* 57
3.4 *Shift* 58
4.1 *PrefixesBis* 68
4.2 *PreKmpBis* 69
4.3 *KMPE_Search* 73
4.4 *Shift* 74
5.1 *SuffixesBis* 83
5.2 *PreBmGsBis* 84
5.3 *FSE_Search* 86
6.1 *PrefixesRmq* 99

Abréviations

ADN : *Acide DésoxyriboNucléique*
ARN : *Acide RiboNucléique*
FS : *Fast Search*
FSE : *Fast Search Extended*
ISA : *Inverse Suffix Array*
KMP : *Knuth-Morris-Pratt*
KMPE : *Knuth-Morris-Pratt Extended*
LCP : *Longest Common Prefix*
MP : *Morris-Pratt*
MPE : *Morris-Pratt Extended*
NGS : *Next Generation Sequences*
RMQ : *Range Minimum Query*
SA : *Suffix Array*

Introduction Générale

Contexte et Présentation du Sujet de la Thèse

Au cours des dernières années, il y a eu un développement rapide des nouvelles technologies de *Séquençage à Haut Débit* (SHD) (en anglais, *Next Generation Sequencing* (NGS)) [Shendure and Ji, 2008; von Bubnoff, 2008; Reuter *et al.*, 2015] produisant de plus en plus de *séquences biologiques*, i.e., des séquences codant des macromolécules biologiques (ADN, ARN et protéines). Ces technologies permettent de produire des séquences biologiques de manière plus rapide, et à des coûts plus abordables, que la technique de séquençage traditionnelle de Sanger [Sanger *et al.*, 1977]. Ainsi, il est devenu possible de produire des quantités gigantesques de ces données en un temps assez court, même pour des laboratoires de petites tailles et de budgets réduits.

Une conséquence directe des NGS est la production de milliards de séquences biologiques *fortement similaires* ayant un taux de similarité très élevé, i.e., des séquences biologiques identiques à plus de 99% à un génome de référence. À titre d'exemple, le projet *1000 Genomes* [Consortium and others, 2010; Zheng-Bradley and Flicek, 2016] vise à produire un catalogue de variations génétiques humaines via le séquençage de génomes d'un grand nombre de personnes. Un autre exemple, le projet *The Cancer Genome Atlas* (TCGA) [Weinstein *et al.*, 2013] vise à identifier les variations génétiques humaines générant le cancer.

À partir de là se révèle un fort besoin de manipuler efficacement des ensembles de séquences fortement similaires. Parmi les problèmes rencontrés lors de la manipulation de ce type de séquences, on a le problème de *recherche incrémentale* d'un mot dans un ensemble de séquences fortement similaires (*On-line String Matching in Highly Similar Sequences*) [Nsira *et al.*, 2014a; Nsira *et al.*, 2014b; Nsira *et al.*, 2017].

Dans cette thèse, nous développons de nouveaux algorithmes, appelés respectivement MPE [Nsira *et al.*, 2014b], KMPE [Nsira *et al.*, 2017] et FSE [Nsira *et al.*, 2014a], pour traiter le problème de recherche incrémentale d'un mot dans un ensemble de séquences fortement similaires (*On-line String Matching in Highly Similar Sequences*).

Contributions

Dans le cadre de cette thèse, nous apportons les contributions suivantes, nous développons **trois nouveaux algorithmes** de recherche incrémentale d'un seul motif dans un ensemble de séquences fortement similaires :

- Le premier algorithme, appelé MPE [Nsira *et al.*, 2014b], est une extension de l'algorithme de Morris-Pratt classique [Morris and Pratt, 1970] à la recherche dans un ensemble de séquences fortement similaires ;
- Le deuxième algorithme, appelé KMPE [Nsira *et al.*, 2017], est une extension de l'algorithme de de Knuth-Morris-Pratt classique [Knuth *et al.*, 1977] à la recherche dans un ensemble de séquences fortement similaires ;

— Le troisième algorithme, appelé FSE [Nsira *et al.*, 2014a], est une extension de l'algorithme de recherche rapide qui est une variante de l'algorithme de Boyer-Moore [Boyer and Moore, 1977] ;

Dans ce qui suit, nous détaillons comment est organisé ce mémoire de thèse.

Organisation du Mémoire

Ce mémoire est organisé de la manière suivante :

Dans le **chapitre 1**, nous établissons les notions et concepts utiles pour la lecture du reste du mémoire. Nous définissons les problèmes de l'algorithmique du texte qui nous intéressent.

Dans le **chapitre 2**, nous présentons une étude de l'art sur les différentes solutions de recherche de motifs dans un ensemble de séquences fortement similaires. Nous commençons par présenter la recherche de motifs dans la famille des index basiques. Puis dans une partie suivante, nous décrivons la recherche de motifs dans une deuxième famille encore plus avancée se basant sur les répétitions régulières dans les séquences pour créer l'index. Nous retraçons toutes les complexités temporelles et spatiales offertes par ces index. Nous allons montrer que ces deux familles ne sont pas appropriées directement à l'indexation de séquences de similarité élevée. Dans une troisième partie, nous présentons la recherche de motifs dans une famille récente d'index permettant de tenir profit de la grande similarité entre les données d'entrée. Nous allons montrer que ces index ont montré leurs intérêt. La dernière partie est consacrée pour décrire la seule solution, selon nos connaissances, qui est partie sur la recherche incrémentale avec l'indexation des séquences fortement similaires en entrée. Nous présentons une conclusion à ce chapitre.

Dans le **chapitre 3**, nous présentons notre premier algorithme, intitulé MPE. Dans la première partie de ce chapitre, nous détaillons l'algorithme de Morris-Pratt séquentiel. Dans la seconde partie, nous donnons une présentation détaillée de notre algorithme MPE. Dans la troisième partie, nous calculons la complexité en temps de calcul et celle en espace mémoire de l'algorithme. Dans la quatrième partie, nous donnons un exemple illustratif pour l'algorithme MPE. Enfin, dans la dernière partie, nous présentons une conclusion à ce chapitre.

Dans le **chapitre 4**, nous présentons notre deuxième algorithme, intitulé KMPE. Dans la première partie de ce chapitre, nous faisons une présentation détaillée des différentes phases de l'algorithme de Knuth-Morris-Pratt classique. Dans la seconde partie, nous décrivons notre algorithme KMPE. Dans la troisième partie, nous calculons la complexité en temps de calcul et celle en espace mémoire de l'algorithme. Dans la quatrième partie, nous donnons un exemple illustratif pour l'algorithme KMPE. Enfin, dans la dernière partie, nous présentons une conclusion à ce chapitre.

Dans le **chapitre 5**, nous présentons le dernier algorithme intitulé FSE. Dans la

première partie de ce chapitre, nous détaillons l'algorithme rapide classique. Dans la seconde partie, nous expliquons le fonctionnement de notre algorithme FSE. Dans la troisième partie, nous discutons la complexité en temps de calcul et celle en espace mémoire de l'algorithme. Dans la quatrième partie, nous donnons un exemple illustratif pour l'algorithme FSE. Enfin, dans la dernière partie, nous terminons par une conclusion à ce chapitre.

Le **chapitre 6**, nous le consacrons aux résultats expérimentaux. Nous donnons des résultats obtenus en pratique à l'issue d'une étude expérimentale menée sur des jeux de données synthétiques et sur des jeux de données biologiques réelles. Nous effectuons une étude comparative de nos algorithmes avec un des plus efficace algorithmes séquentiels de recherche incrémentale. Puis, nous faisons une étude comparative entre nos différents algorithmes. Finalement, nous comparons notre algorithme rapide FSE avec l'unique approche de recherche incrémentale qui s'intéresse au même problème que nous. Enfin, nous donnons une conclusion.

Finalement, nous terminons ce mémoire par une conclusion générale qui résume l'ensemble de nos travaux et présente nos perspectives futures de recherche.

Chapitre 1

Préliminaires

1.1 Introduction

Dans ce chapitre, nous présentons des définitions, des notations et des notions clés utilisées dans la suite de ce mémoire.

Ce chapitre est organisé comme suit :

Dans la section 1.2, nous introduisons des propriétés combinatoires sur les alphabets. Dans la section 1.3, nous détaillons les différentes méthodes pour la recherche d'un mot dans un autre mot de longueur supérieure. Dans la section 1.4, nous introduisons la notion de *compressibilité* de mots. Enfin, la dernière section 1.5, nous l'avons consacrée pour présenter le problème de recherche du minimum dans un intervalle, appelée en anglais *Range Minimum Query* (RMQ).

1.2 Définitions et Notations

Soit Σ un alphabet fini, un *mot* est un élément de Σ^*, c'est la concaténation d'éléments de Σ. La longueur d'un mot y, notée $|y|$, est le nombre de caractères constituant ce mot. Par convention, le mot de longueur nulle sera notée ε et $\Sigma^+ = \Sigma^* \cup \{\varepsilon\}$. Le i-ème caractère d'un mot y, $0 \leq i \leq |y| - 1$, sera noté $y[i]$. Une portion d'un mot y qui commence à une position i et se termine à une position j, $0 \leq i \leq j \leq |y| - 1$, est appelée *facteur* de y et sera noté $y[i\,..\,j]$. Quand $i = 0$ et $0 \leq j \leq |y| - 1$, le facteur correspondant est appelé *préfixe* de y et quand $0 \leq i \leq |y| - 1$ et $j = |y| - 1$, le facteur correspondant est appelé *suffixe* de y. Un préfixe (ou un suffixe) x' de x est dit *propre* si $x' \neq x$.

Le *miroir* d'un mot y, noté $\tilde{y}$, est le mot $y[|y| - 1]y[|y| - 2] \cdots y[i + 1]y[i] \cdots y[0]$.

L'*ordre lexicographique*, noté $\ll \leq \gg$, est un ordre sur les mots induit par un ordre sur les caractères noté de la même façon. Il est défini comme suit. Pour y et $y' \in \Sigma^*$, $y \leq y'$ si et seulement si, soit y est un préfixe de y', soit y et y' se décomposent de la forme uav et ubw avec $u, v, w \in \Sigma^*$, $a, b \in \Sigma$ et $a < b$ [Crochemore *et al.*, 2007].

Deux mots x et x' sont *conjugués* s'il existe deux mots u et v tels que $x = uv$ et $x' = vu$. Le mot x est aussi dit *décalage circulaire* de x'.

1.3 Recherche de Motifs

Le problème de *recherche d'un mot dans un texte* (*string matching*) [Stephen, 1994; Crochemore and Rytter, 1994; Navarro and Raffinot, 2002; Crochemore *et al.*, 2007; Pandiselvam *et al.*, 2009] est l'un des plus vieux problèmes dans le domaine de l'*Algorithmique du Texte* (*Stringology*). Il s'agit de localiser toutes les occurrences, exactes ou approchées, d'un mot x (*motif*) dans un mot plus long y (*texte* ou *séquence*). Dans le reste de ce mémoire nous utilisons le terme *motif* pour désigner le mot à chercher et le terme *séquence* pour désigner le mot où se fait la recherche.

Dans cette thèse, nous nous intéressons particulièrement au problème de *recherche exacte d'un motif dans une/ des séquence(s)* (*exact string matching*) [Stephen, 1994; Crochemore and Rytter, 2003; Crochemore *et al.*, 2007; Kalsi *et al.*, 2008].

1.3.1 Problème de recherche exacte

Commençons d'abord par définir ce problème.

Le *problème de recherche exacte d'un motif dans une séquence* est défini comme suit : Soient x un motif et y une séquence, $|x| \leq |y|$, trouver un facteur x' de y tel que $x = x'$.

On peut regrouper les algorithmes qui traitent le problème de recherche exacte en deux classes principales [Charras and Lecroq, 2004a; Melichar *et al.*, 2005; Crochemore *et al.*, 2007] :

1. Algorithmes adoptant l'approche de *recherche globale* [Crochemore and Rytter, 2003; Crochemore *et al.*, 2007]
2. Algorithmes adoptant l'approche de *recherche incrémentale* [Charras and Lecroq, 2004a; Faro and Lecroq, 2013].

Approche de recherche globale

Un algorithme de *recherche globale* (*offline*) est une approche où le traitement des séquences se fait en utilisant un *index*, i.e. structure de données, construit sur les facteurs de ces séquences. On a alors besoin d'avoir la totalité des séquences pour commencer le traitement.

On distingue plusieurs types d'index [Prieur, 2007; Salson, 2009a] :

— *Index succincts* : Ce sont des index qui utilisent un espace mémoire proportionnel aux longueurs des séquences indexées et qui permettent une recherche rapide.

— *Index compressés* : Ce sont des index qui se basent sur les *régularités*, i.e. répétitions, des séquences. Ils utilisent un espace mémoire qui dépend de la *compressibilité* (mesurée par l'*entropie*) des séquences indexées.

— *Auto-index* : Ce sont des index compressés qui permettent de maintenir l'intégralité de séquences avec un taux négligeable d'informations, i.e. capables de reproduire n'importe quel facteur sans stocker les séquences explicitement.

En général, lorsque les séquences sont de longueurs importantes, les index deviennent coûteux en espace mémoire [Navarro and Raffinot, 2002; Firdose and Nilay, 2014]. C'est la raison pour laquelle, il est préférable d'adopter une *approche de recherche incrémentale* pour ce type de séquences.

Approche de recherche incrémentale

Une approche de *recherche incrémentale* (*on-line*) est une approche où le traitement des séquences se fait au fur et à mesure de la lecture de ces séquences, sans avoir besoin d'avoir la totalité des séquences pour commencer le traitement.

On distingue quatre types d'approches incrémentales de recherche exacte d'un motif dans une séquence [Faro and Lecroq, 2010; Faro, 2016] :

1. Approche incrémentale par *utilisation d'automates* [Crochemore and Rytter, 2003; Charras and Lecroq, 2004a; Faro and Lecroq, 2012]
2. Approche incrémentale par *applications d'algorithmes temps réel* [Crochemore and Rytter, 2003; Breslauer *et al.*, 2011; Breslauer and Galil, 2011]
3. Approche incrémentale par *applications d'algorithmes à espace constant* [Crochemore and Rytter, 2003; Breslauer *et al.*, 2011; Breslauer and Galil, 2011]
4. Approche incrémentale par *comparaisons de caractères* [Crochemore and Rytter, 2003; Charras and Lecroq, 2004a; Crochemore *et al.*, 2007]

Dans cette thèse, nous nous intéressons à l'approche incrémentale par comparaisons de caractères. En effet, une répartition temporelle des algorithmes proposés au cours des 26 dernières années montre que la classe d'algorithmes basée sur la comparaison des caractères est la classe plus large et se compose de près de 50% de tous les algorithmes [Faro *et al.*, 2016]. En outre, des résultats expérimentaux sur différents ensembles de données montrent que les algorithmes de cette classe permettent en moyenne de réaliser de bonnes performances, notamment pour des longs motifs [Faro *et al.*, 2016].

En adoptant cette approche, la recherche exacte d'un motif x dans une séquence y se fait comme suit :

Durant la première étape, on fait un prétraitement du motif x. Cette étape permet d'avoir des informations nécessaires à la recherche exacte de x dans y.

Durant la seconde étape, on examine la séquence y en utilisant une fenêtre glissante de longueur $|x|$. La fenêtre est d'abord positionnée sur $y[0\,..\,|x|-1]$. Elle glisse tout au long de la séquence y. Lorsque la fenêtre est sur un facteur $y[j\,..\,j+|x|-1]$, $0 \leq j \leq |y|-1$, on vérifie si ce facteur est égal à x. Ceci se fait par comparaisons des caractères de x à ceux en face dans y. Cette opération de comparaison est appelée *tentative* à la position j. Chaque tentative est suivie d'un décalage à droite de la fenêtre. Un décalage de longueur s, à partir d'une position j, est dit *valide* s'il ne peut y avoir d'occurrences du motif x commençant à des positions entre $j+1$ et $j+s-1$.

Le principe d'utilisation d'une fenêtre glissante pour faire de la recherche exacte d'un motif x dans une séquence y est illustré par la figure 1.1.

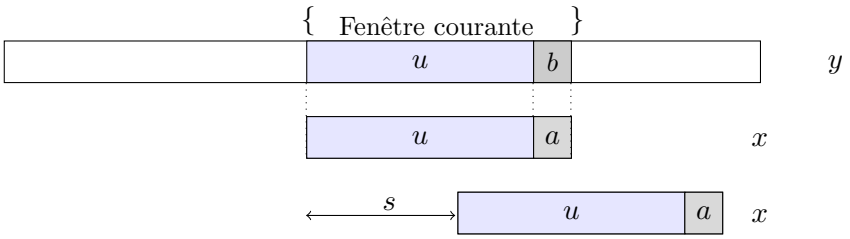


FIGURE 1.1 – Principe d'utilisation d'une fenêtre glissante pour faire de la recherche exacte d'un motif x dans une séquence y.

1.3.2 Problème de recherche distribuée

Le *problème de recherche distribuée* d'un motif dans un ensemble de séquences est défini comme suit : Soient un ensemble de r séquences de même longueur $Y = \{y_0, y_1, \ldots, y_{r-1}\}$ et un motif x, trouver un facteur x' dans les séquences tel que $x = x'$ de façon à ce que les facteurs de x soient localisés à des **positions consécutives** dans différentes séquences de Y.

Formellement, trouver toutes les positions $0 \leq j \leq |y_g| - |x| + 1$, tel que pour $0 \leq i \leq |x| - 1$ nous avons $x[i] = y_g[j + i]$ pour tout $g \in [0, r - 1]$.

Ce problème a été d'abord introduit par [Crawford *et al.*, 1998] et traité par la suite par [Holub *et al.*, 2001]. L'approche de [Holub *et al.*, 2001] se base sur la construction d'automates fini non déterministes (NFA) et l'utilisation de l'algorithme *Shift-Or* pour simuler les NFA construits.

1.4 Compressibilité

Dans cette partie, nous présentons des mesures de quantité d'informations dans un mot. Notamment, nous introduisons l'*entropie empirique* qui est une mesure commune de la compressibilité d'un mot. Cette information est utile pour l'indexation basée sur la compression.

Définition 1.1 *L'entropie empirique d'ordre zéro d'un mot y de longueur n, $H_0(y)$, est le nombre moyen de bits nécessaires pour représenter un caractère de y si chaque caractère reçoit toujours le même code. Formellement elle est définie comme suit :*

$$H_0(y) = \sum_{c \in \Sigma} \frac{n_c}{n} \log_2 \frac{n}{n_c} = -\sum_{c \in \Sigma} \frac{n_c}{n} \log_2 \frac{n_c}{n} \tag{1.1}$$

tel que n_c est le nombre d'occurrences du caractère c dans y.

Si l'on veut obtenir de meilleurs ratios de compressibilité, on peut coder chaque caractère selon la probabilité d'apparition des k caractères précédents. Ainsi, on obtient le k-ème ordre empirique défini comme suit :

$$H_k(y) = \sum_{s \in \Sigma^k} \frac{|y^s|}{n} H_0(y^s) \tag{1.2}$$

tel que y^s est la suite de caractères précédés par le facteur s dans y.

L'objectif principal des index compressés est de représenter efficacement l'information de la manière la plus réduite, sans risque de perte, de sorte que l'on peut facilement récupérer le mot d'origine après la décompression. Les structures d'index compressés d'un mot requièrent un espace de stockage proportionnel à l'**entropie** du mot.

La plupart des structures de données compressées permettent de répondre à des requêtes de base telles que :

— $rank_c(y, i)$: elle compte le nombre d'occurrences d'un caractère donné c jusqu'à une position i dans le mot y.

— $select_c(y, i)$: elle trouve la position de la i-ème occurrence d'un caractère donné c dans le mot y.

Ces structures de données sont aussi capables de répondre à d'autres requêtes sur les facteurs d'un mot y telle que l'opération $access(y, i)$ permettant de retourner le i-ème caractère $y[i]$ dans le mot y [Munro and Nekrich, 2015].

1.5 Recherche du Minimum dans un Intervalle

Commençons par une définition :

Définition 1.2 *Étant donné un tableau A à n entiers et deux indices i et j, $1 \leq i \leq j \leq n$, une recherche du minimum sur un intervalle $[i, j]$, en anglais Range Minimum Query (RMQ), retourne un indice k tel que :*

$$A[k] = \min\{A[h] | i \leq h \leq j\} \tag{1.3}$$

Une telle requête sera notée $rmq_A(i, j)$.

Plusieurs algorithmes ont été proposés pour déterminer $rmq_A(i, j)$ [Bender and Farach-Colton, 2000; Bender *et al.*, 2005; Fischer and Heun, 2006; Ohlebusch, 2013].

Dans ce qui suit nous décrivons des algorithmes de répondre à des requêtes RMQ avec prétraitement. En fait, la phase de prétraitement permet de mémoriser des réponses à toutes les requêtes possibles de tous les sous-tableaux de A dans une matrice M.

Algorithme avec temps constant et espace linéarithmique [Ohlebusch, 2013]

En adoptant cet algorithme on opère en deux étapes :

(i) Durant la première étape, on remplit la matrice M. Chaque case $M[i, j]$ contient la position du minimum dans l'intervalle $A[i, i + 2^j - 1]$ et est calculée selon la récurrence suivante :

Pour $1 \leq i \leq n$ et $0 \leq j \leq \lfloor \log n \rfloor$ on a :

$$\begin{array}{lll} M[i,0] & = i & \\ M[i,j] & = & \begin{cases} M[\text{i, j-1}] & \text{si } A[M[i,j-1]] \leq A[M[i+2^{j-1}, j-1]] \\ M[\text{i}+2^{j-1}, j-1] & \text{sinon .} \end{cases} \end{array} \quad (1.4)$$

La valeur de $M[i,j]$ est définie pour $i+2^j-1 \leq n$ et elle est indéfinie pour $i+2^j-1 > n$.

(ii) Enfin durant la seconde étape, le calcul de $rmq_A(i,j)$ se fait en deux sous-étapes :

D'abord on sélectionne deux sous- intervalles chevauchants qui couvrent exactement l'intervalle $[i,j]$:

(a) Le premier sous-intervalle commence à la position i et se termine à l'intérieur de l'intervalle $[i,j]$. Il a une taille 2^ℓ, où $\ell = \lfloor \log(j-i+1) \rfloor$. Le minimum dans ce sous-intervalle est donné par l'équation suivante :

$$A[M[i,\ell]] = \min\{A[i, i+2^{\lfloor \log(j-i+1) \rfloor} - 1]\} \quad (1.5)$$

(b) Le second sous-intervalle commence à l'intérieur du premier sous-intervalle et se termine à la position j. Le minimum dans ce sous-intervalle est donné par l'équation suivante :

$$A[M[j-2^\ell+1, \ell]] = \min\{A[j-2^{\lfloor \log(j-i+1) \rfloor}+1, j]\} \quad (1.6)$$

Ainsi $rmq_A(i,j)$ est le minimum des minima des sous-intervalles précédemment décrits. Il est donné par l'équation suivante :

$$\min\{A[M[i,\ell]], A[M[j-2^\ell+1, \ell]]\} \quad (1.7)$$

Notons que pour cet algorithme on traite seulement les intervalles dont les longueurs sont des puissances de 2, i.e., tous les intervalles de tailles 2^j, $0 \leq j \leq \log n$.

Cet algorithme est de complexité $O(n \log n)$ en espace mémoire et $O(1)$ en temps de calcul.

Algorithme avec temps logarithmique et espace linéaire [Ohlebusch, 2013]

En adoptant cet algorithme on opère en deux étapes :

(i) Durant la première étape, on divise le tableau A en sous-tableaux non-chevauchants $B_1 \ldots B_{n/s}$, chacun de taille $s = \frac{\log n}{4}$.

(ii) Durant la seconde étape, on calcule deux nouveaux tableaux A' et B' de taille $\frac{n}{s}$, où $A'[i]$ contient le minimum dans le sous-tableau B_i et $B'[i]$ contient la position de ce minimum dans B_i.

(iii) Ensuite, durant la troisième étape, on effectue le prétraitement de A' en utilisant l'algorithme linéarithmique précédant afin de répondre à $rmq_{A'}(i,j)$ avec l'équation 1.7, i.e., basé sur la matrice M' correspondante.

Ainsi, la réponse à la requête $rmq_A(i, j)$ se fait en trois étapes comme suit :

(i) On calcule les positions relatives des minimums dans les sous-tableaux contenant les bornes de l'intervalle i et j. En fait, on commence la recherche dans le premier sous-tableau par la position i et on termine la recherche dans le deuxième sous-tableau à la position j, i.e., les éléments en dehors des bornes ne sont pas considérés. On peut ainsi calculer facilement les positions absolues des minimums obtenus dans A.

(ii) On calcule la position relative du minimum de tous les sous-tableaux $B_q, \ldots, B_q$, où $p \leq q$, entièrement contenus entre les deux sous-tableaux de l'étape précédente. Pour ce faire, on calcule d'abord $rmq_{A'}(p, q)$ (en utilisant l'algorithme linéarithmique) et on obtient la position k du minimum dans $A'[p, q]$. Ensuite, on utilise $B'[k]$ pour trouver la position absolue du minimum obtenu dans A.

(iii) Finalement, on compare les trois minimum obtenus.

Cet algorithme est de complexité $O(n/\log n \log(n/\log n)) = O(n)$ en temps de calcul et en espace mémoire.

Approche avec temps constant et espace linéaire [Ohlebusch, 2013]

Commençons d'abord par donner une définition d' *arbre cartésien.*

Définition 1.3 *[Vuillemin, 1980] L'arbre cartésien d'une séquence de nombres distincts est un arbre binaire vérifiant les propriétés suivantes :*

(a) Chaque nombre de la séquence est représenté par un seul nœud. Chaque nœud est associé à une seule valeur de la séquence.

(b) Une traversée symétrique de l'arbre donne les valeurs dans le même ordre de leurs apparitions dans la séquence d'entrée.

(c) Chaque nœud interne possède une valeur supérieure à celle de son parent.

En adoptant cet algorithme, on opère en deux étapes :

(i) Durant la première étape, on construit les *arbres cartésiens* pour tous les sous-tableaux possibles du tableau A. Ensuite, on détermine pour chaque arbre construit le résultat possible pour toutes les *rmq*. On mémorise les résultats de tous les atbres dans une table T de taille $O(n)$.

(ii) Enfin durant la seconde étape, on consulte l'arbre cartésien d'un sous-tableau spécifique pour repérer la valeur de $rmq_A(i, j)$.

Notons que durant la première étape, on doit trouver un moyen pour savoir pour chaque arbre binaire l'entier qu'il représente de façon unique dans la table T. Pour ce faire, on utilise un algorithme de recherche de parcours en largeur à travers l'arbre et on ajoute des feuilles de telle sorte que chaque nœud possède exactement deux descendants. L'entier est ensuite généré en représentant chaque nœud interne par 0 et chaque feuille par 1.

Enfin, notons que cet algorithme vérifie les propriétés suivantes :

(a) Pour deux tableaux A et B on a $rmq_A(i, j) = rmq_B(i, j)$ si et seulement leurs *arbres cartésiens* sont isomorphes.

(b) Si le nombre de différents arbres cartésiens de s nœuds, où $s = \frac{\log n}{4}$ est C_s alors ce nombre correspond au s-ème *nombre de Catalan* [Roman, 2015].

(c) Le nombre de différents arbres cartésiens de s nœuds pour tous les sous-tableaux est de l'ordre de 4^s.

Cet algorithme est de complexité $O(n)$ en temps de calcul et en espace mémoire.

1.6 Formalisation du Problème de Recherche Incrémentale d'un motif dans un ensemble de séquences fortement similaires

Un ensemble de séquence fortement similaires est défini comme suit :

Définition 1.4 *[Nsira* et al.*, 2014b]*

Un ensemble Y de r séquences $\{y_0, y_1, \ldots, y_{r-1}\}$ de même longueur n sur un alphabet Σ est un ensemble de ***séquence fortement similaires*** *si :*

(a) y_0 est une séquence de référence,

(b) Les séquences $y_1, y_2, \ldots, y_{r-1}$ sont représentées par une liste de k positions de variations par rapport à y_0 : $Z = ((\mathcal{G}_0, j_0, c_0), (\mathcal{G}_1, j_1, c_1), \ldots, (\mathcal{G}_{k-1}, j_{k-1}, c_{k-1}))$, où $c_i \in \Sigma$, $c_i = y_g[j_i] \neq y_0[j_i])$, $0 \leq j_i \leq n-1$, $\mathcal{G}_i = \{g \mid 1 \leq g \leq r-1, c_i = y_g[j_i] \neq y_0[j_i]\} \neq \emptyset$ pour $0 \leq i \leq k-1$,

(c) La liste Z est triée selon l'ordre croissant du deuxième composant du triplet, i.e., la position : $\forall\, (\mathcal{G}_i, j_i, c_i), (\mathcal{G}_{i+1}, j_{i+1}, c_{i+1}) \in Z$ on a $j_i < j_{i+1}$,

(d) Pour chaque couple de triplet dans Z il existe, au moins, M positions d'écart : $\forall\, (\mathcal{G}_i, j_i, c_i), (\mathcal{G}_{i+1}, j_{i+1}, c_{i+1}) \in Z$ on a $j_{i+1} - j_i > M$.

L'ensemble de séquences Y est représenté par y_0 et Z. Notons que plusieurs séquences peuvent avoir la même variation à la même position. De plus, il existe une restriction sur le nombre de variations ce qui signifie qu'il peut y avoir au plus une variation dans une fenêtre de taille M.

Notre problème consiste à trouver toutes les occurrences d'un motif x donné de longueur $m \leq M$ dans les séquences de l'ensemble de séquences fortement similaires Y représenté par y_0 et Z.

Ce problème peut être vu comme un hybride entre la recherche de motif distribuée et la recherche de motif approchée avec k différences [Amihood *et al.*, 2004].

Chapitre 2

État de l'Art sur les Algorithmes de Recherche de Motifs dans un Index Représentant un Ensemble de Séquences

Introduction

Dans ce chapitre, nous dressons un état de l'art sur les algorithmes de recherche de motifs dans un ensemble de séquences fortement similaires. La majorité des algorithmes présentés sont basés sur un *index*, i.e. structure de données, construit sur les facteurs de séquences d'entrée. Ces algorithmes sont présentés selon l'ordre décroissant de consommation d'espace mémoire.

Le reste du chapitre est organisé comme suit :

Dans la section 2.1, nous présentons des *algorithmes de recherche de motifs dans un index classique*. Dans la seconde section 2.2, nous présentons des *algorithmes de recherche de motifs dans un index compressé*. Dans la section 2.3, nous présentons des *algorithmes de recherche de motifs dans un index avancé*, i.e., un index directement dédié à l'indexation des séquences fortement similaires.

2.1 Algorithmes de Recherche de Motifs dans un Index Classique

Les *index classiques*, i.e. *trie*, *trie de suffixes*, *arbre de suffixes*, *automate de suffixes*, *automate compact de suffixes*, *tableau de suffixes*, définis ci-dessous permettent de mémoriser les facteurs d'une séquence. Une séquence possède un nombre quadratique de facteurs et un nombre linéaire de suffixes. La représentation des suffixes parait judicieuse, puisque tout facteur est un préfixe d'un suffixe.

2.1.1 Préliminaires

Dans cette partie, nous donnons quelques définitions qui seront utiles tout au long de ce chapitre.

Définition 2.1 *Un arbre est une structure de données composée de nœuds, connectés par des branches. À l'exception d'un nœud particulier appelé racine, chaque nœud a exactement un parent. Les nœud peuvent avoir zéro enfant ou plus. Les nœuds n'ayant pas d'enfants sont appelés feuilles, alors que les autres nœud sont appelés nœud internes.*

Définition 2.2 *Un automate fini déterministe M est défini comme un quintuplet : $M = (Q, \Sigma, \delta, I, F)$, tel que :*

- *Q est un ensemble fini d'états ;*
- *Σ est l'alphabet ;*
- *$i \in Q$ est l'état initial ;*
- *$F \subseteq Q$ est l'ensemble des états terminaux ;*
- *$\delta : Q \times \Sigma \to Q$ est la fonction transition permettant de passer d'un état à un autre par un caractère de l'alphabet.*

2.1.2 Algorithmes de Recherche de Motifs dans un Trie

Donnons une définition d'un *trie*.

Définition 2.3 *[Sedgewick and Flajolet, 1996; Ervin, 1998; Crochemore and Lecroq, 2009] Un trie associé à un ensemble de séquences $Y = \{y_0, y_1, \ldots, y_{r-1}\}$ est un arbre où chaque branche est étiquetée par un seul caractère. Les préfixes communs à plusieurs séquences sont mis en facteurs et la concaténation des étiquettes d'un chemin de la racine vers un nœud interne donne un préfixe d'une séquence de l'ensemble des séquences Y. Un trie peut être vu comme un automate fini déterministe où la racine représente l'état initial et une feuille représente un état final.*

Ci-dessous, un trie associé à l'ensemble $Y = \{$`C$, CTAG$, GTTG$, GTAGTTAG$`$\}$ (voir la figure 2.1).

Construction

Soit un ensemble de séquences $Y = \{y_0, y_1, \ldots, y_{r-1}\}$, la construction [Crochemore and Lecroq, 2009] du trie associé à Y, *Trie*(Y), se fait en insérant des chemins représentant les séquences de Y. Pour insérer une séquence y_g, $g \in [0, r-1]$, dans *Trie*(Y), on localise d'abord le nœud représentant le plus long préfixe commun à y_g et les séquences déjà insérées et on termine le chemin par des nœuds représentant le suffixe restant de y_g. Le chemin partant de la racine du *Trie*(Y) vers la feuille créée lors de l'insertion de y_g épelle alors la séquence y_g.

Cet algorithme est de complexité $O(N \log \sigma)$ en temps de calcul, où N est la somme des longueurs des séquences de Y et σ est le cardinal de l'alphabet.

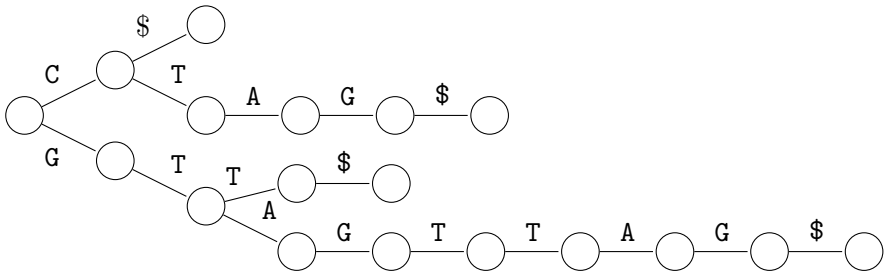


FIGURE 2.1 – Trie associé à l'ensemble de séquences Y = {`C$, CTAG$, GTTG$, GTAGTTAG$`}. Afin d'éviter que certaines séquences de Y correspondent aux préfixes d'autres séquences de Y, on ajoute à la fin des séquences un caractère spécial (qui n'appartient pas à l'alphabet), appelé *terminateur*, noté \$. Ceci permet d'éviter que certaines séquences se terminent dans des nœuds internes du trie. Ainsi, chaque feuille représente une séquence distincte de Y et *Trie*(Y) contient exactement r feuilles. En général le nombre de nœuds issus d'un nœud interne peut être aussi grand que la taille de l'alphabet.

Recherche d'un motif

La recherche d'un motif x dans Y [Crochemore and Lecroq, 2009] consiste à identifier le plus long préfixe de x qui apparaît dans des séquences de Y. Pour ce faire, on repère un chemin étiqueté par x dans *Trie*(Y). Si un tel chemin existe, alors il y a deux cas possibles :

(a) Le chemin se termine à un nœud interne, alors x est un préfixe de certaines séquences de Y représentées par *Trie*(Y),

(b) Le chemin se termine à une feuille, alors x est une séquence de Y.

Si un tel chemin n'existe pas alors x ne correspond à aucun préfixe de séquences de Y.

Cet algorithme est de complexité $O(m \log \sigma)$ en temps de calcul, où m est la longueur du motif.

Un *trie* peut être aussi utilisé pour représenter une seule séquence. Ainsi, un index, appelé *trie de suffixes*, dont la taille est linéaire à la longueur de la séquence d'entrée est construit. Décrivons *trie de suffixes*.

2.1.3 Algorithmes de Recherche de Motifs dans un Trie de Suffixes

Donnons une définition d'un *trie de suffixes*.

Définition 2.4 *[Crochemore and Lecroq, 2009] Un trie de suffixes d'une séquence y est un trie qui représente tous les suffixes de cette séquence.*

Construction

Soit une séquence y, la construction [Crochemore and Lecroq, 2009] du trie de suffixes associé à y se fait en insérant des chemins représentant les suffixes de y. Pour insérer un

suffixe y' de y dans le trie de y, on localise d'abord le nœud représentant le plus long préfixe commun à y' et les suffixes déjà insérés et on termine le chemin par des nœuds représentant le suffixe restant de y'. Le chemin partant de la racine du trie et arrivant à la feuille créée lors de l'insertion de y' représente alors le suffixe y'.

Cet algorithme est de complexité $O(n^2)$ en temps de calcul, où n est la longueur de la séquence y.

Recherche d'un motif

La recherche d'un motif x dans y [Crochemore and Lecroq, 2009] consiste à identifier le plus long préfixe de x qui apparaît comme suffixe de y. Pour ce faire, on repère un chemin dans le trie étiqueté par les caractères du motif x. Lors de la recherche du motif x, il y a deux possibilités :

(a) Il existe un chemin étiqueté par tous les caractères du motif x. Un nœud interne est rencontré et on obtient tous les suffixes de y préfixés par x. Les positions de ces suffixes sont mémorisées dans les feuilles contenues dans le sous-arbre enraciné par le nœud interne atteint. Ainsi, le nombre d'occurrences de x correspond au nombre de feuilles dans ce sous-arbre ;

(b) Il n'existe pas de chemin étiqueté par tous les caractères du motif x. Aucun suffixe de y n'est alors préfixé par le motif x. Ainsi x n'apparait pas dans y.

Cet algorithme est de complexité $O(m + occ)$ en temps de calcul, où m est la longueur du motif x et occ est le nombre d'occurrences de x dans y.

Il est évident que le trie permet de résoudre plusieurs problèmes dans le domaine de l'algorithmique de texte [Crochemore *et al.*, 2007]. Cependant, en pratique, le stockage de nœuds nécessite un espace mémoire important. Un trie peut être représenté sous forme d'une liste [Dundas, 1991]. Bien que l'utilisation d'une liste permet de compresser le trie, l'accès aux éléments de cette liste reste lent vu sa nature séquentielle. Une meilleure représentation d'un trie a été proposée par [Aoe *et al.*, 1992] sous forme de tableau multidimensionnel. Cependant, cette représentation n'est pas très utilisée.

2.1.4 Algorithmes de Recherche de Motifs dans un Arbre de Suffixes

Commençons par une définition.

Définition 2.5 *Soit une séquence y de longueur n, l'arbre des suffixes associé, $ST(y)$, et un arbre tel que :*

(a) Il a exactement n feuilles numérotées de 0 à $n-1$ (positions de début des suffixes) ;

(b) Chaque nœud interne possède au moins deux descendants ;

(c) Chaque branche est étiquetée par un facteur non-vide de y ;

(d) Les branches sortantes d'un nœud ont des étiquettes qui commencent par des caractères différents ;

(e) La séquence obtenue par concaténation de toutes les étiquettes sur le chemin de la racine à la feuille i représente le suffixe $y[i\,..\,n-1]$, pour $i = 0, \ldots, n-1$.

Un exemple de $ST(y)$ associé à la séquence $y =$ `CTAGTTAG$` est donné par la figure 2.2.

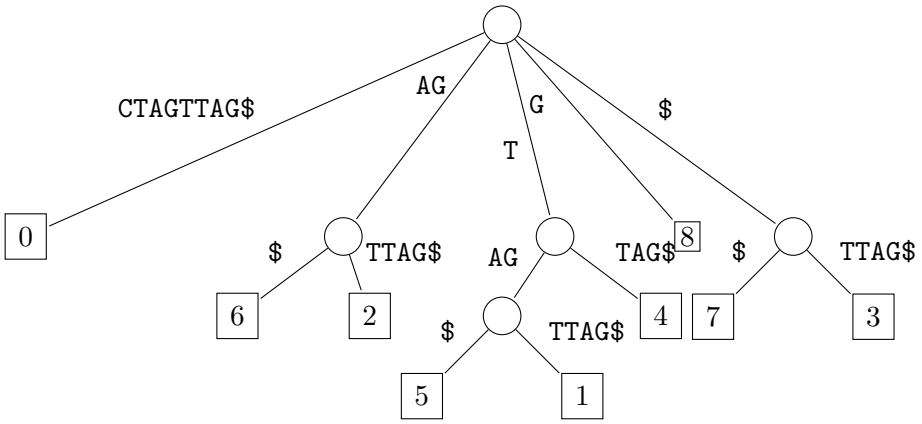


FIGURE 2.2 – Arbre des suffixes de $y =$ `CTAGTTAG$`. La racine représente ε. Les cercles représentent les nœuds internes. Ces derniers représentent les facteurs de y. Les carrés représentent les feuilles. Elles sont numérotées par les positions des débuts des suffixes qu'elles représentent.

Construction

Ils existent plusieurs algorithmes de construction d'arbres de suffixes [Weiner, 1973; McCreight, 1976; Ukkonen, 1995; Farach, 1997]. Dans cette partie, nous choisissons de présenter l'algorithme proposé par [Ukkonen, 1995]. En effet, l'algorithme d'Ukkonen est un algorithme incrémentale et cette forte propriété lui rend très utile.

Soit une séquence y, la construction [Ukkonen, 1995] de l'arbre de suffixes $ST(y)$ se fait en insérant ses caractères dans $ST(y)$ de façon incrémentale, un par un, du premier au dernier. Ainsi $ST(y)$ augmente progressivement au fur et à mesure avec l'ajout de nouveaux caractères. Chaque nœud interne représente le mot formant l'étiquette associée au chemin allant de la racine à ce nœud interne. L'algorithme de Ukkonen se base sur l'utilisation des *liens suffixes* [Ukkonen, 1995]. Ainsi, lors de l'insertion dans $ST(y)$ d'un nouveau nœud interne représentant le mot $c \cdot u$, où $c \in \Sigma$ et $u \in \Sigma^*$, on localise le nœud interne représentant u. Si ce nœud existe alors on utilise un *lien suffixe* permettant de relier le nœud interne représentant $c \cdot u$ avec celui représentant u. Si ce nœud n'existe pas alors on le crée.

Cet algorithme est de complexité $O(n)$ en temps de calcul et en espace mémoire, où n est la longueur de la séquence y.

Recherche d'un motif

La recherche d'un motif x dans une séquence y en utilisant $ST(y)$ est similaire à celle utilisant $Trie(y)$.

En pratique, en utilisant les arbres de suffixes, la consommation d'espace mémoire risque d'être très importante quand il s'agit de très longues séquences, comme les séquences génomiques. Concrètement, la taille de l'arbre de suffixes est d'environ $10 - 20n$ octets. À titre d'exemple, le génome humain contient environ $3 \cdot 10^9$ nucléotides et son arbre de suffixes occupe environ 45 Go d'espace mémoire [Kurtz, 1999]. Ainsi, cette contrainte empêche de gérer d'énormes ensembles de données telles que les données issues de *Séquençage à Haut Débit* (SHD) (en anglais, *Next Generation Sequencing* (NGS)) [Shendure and Ji, 2008; von Bubnoff, 2008; Reuter *et al.*, 2015].

2.1.5 Algorithmes de Recherche de Motifs dans un Automate de Suffixes

Dans cette partie, nous présentons un index appelé *Automate de suffixes*, en anglais *Directed Acyclic Word Graph* (DAWG). Donnons sa définition.

Définition 2.6 *[Blumer* et al.*, 1985] Étant donnée une séquence y, le $DAWG(y)$ est le plus petit automate déterministe qui accepte tous les suffixes de la séquence y.*

Construction

Soit une séquence y, la construction de $DAWG(y)$ [Blumer *et al.*, 1985] se fait en insérant ses caractères, un par un. À chaque étape de la construction, le DAWG doit représenter les suffixes du préfixe courant u de y.

Lors de l'insertion d'un nouveau caractère a, on construit $DAWG(u \cdot a)$ à partir de $DAWG(u)$. Ceci se fait de la manière suivante :

(i) D'abord, on crée un nouveau nœud v et on ajoute de nouveaux arcs étiquetés par a à partir des nœuds associés aux suffixes de u ;

(ii) Ensuite, on considère un nœud v' associé à un suffixe de u et un nœud v'' dans $DAWG(u)$ tel qu'il existe un arc entre v' et v" étiqueté par a et qui correspond à un chemin raccourci par rapport au plus long chemin passant par v' et v''. Puis, on considère u_1 le facteur représenté par ce chemin raccourci. On vérifie si u_1 est un suffixe de $u \cdot a$. Si ce n'est pas le cas alors on ajoute un nouveau nœud v''_1 ayant les mêmes arcs sortant que v'' ;

(iii) Puis, on supprime l'arc étiqueté par a dans le chemin raccourci entre v' à v'' et on le redirige vers v''_1 ;

(iv) Enfin, on vérifie si d'autres nœuds doivent rediriger leurs arcs sortant vers v''_1.

On réitère ce processus jusqu'à ce que la totalité de la séquence $y\$$ soit traitée, i.e., jusqu'à ce que $DAWG(y)$ soit construit.

Cet algorithme est de complexité $O(n)$ en temps de calcul et en espace mémoire, où n est la longueur de la séquence y.

Recherche d'un motif

L'algorithme de recherche d'un motif x dans une séquence y en utilisant $DAWG(y)$ est similaire à celui utilisant $Trie(y)$.

2.1.6 Algorithme de Recherche de Motifs dans un Automate Compact de Suffixes

Dans cette partie, nous présentons un index appelé *Automate Compact de Suffixes*, en anglais *Compact Direct Acyclic Word Graph* (CDAWG). Il est défini comme suit.

Définition 2.7 *[Crochemore and Vérin, 1997] Étant donnée une séquence y, le $CDAWG(y)$ est l'automate minimal de suffixes de y obtenu par compactage de $DAWG(y)$ ou par minimisation de $ST(y)$.*

En fait, la construction de $CDAWG(y)$ par compactage de $DAWG(y)$ se fait en supprimant tous les états non-finaux avec une seule transition sortante. En outre, le $CDAWG(y)$ peut être obtenu par minimisation de $ST(y)$ associé en identifiant les sous-arbres qui reconnaissent les mêmes mots. Ainsi, toutes les feuilles sont fusionnées à un seul nœud final, celui qui représente y lui-même, et tous les nœuds sauf le nœud final sont en correspondance biunivoque avec les répétitions maximales de y.

Dans [Crochemore and Vérin, 1997], Crochemore et Vérin expliquent les relations entre trie de suffixes, arbre de suffixes, DAWG et CDAWG . Ces relations sont illustrées dans la figure 2.3.

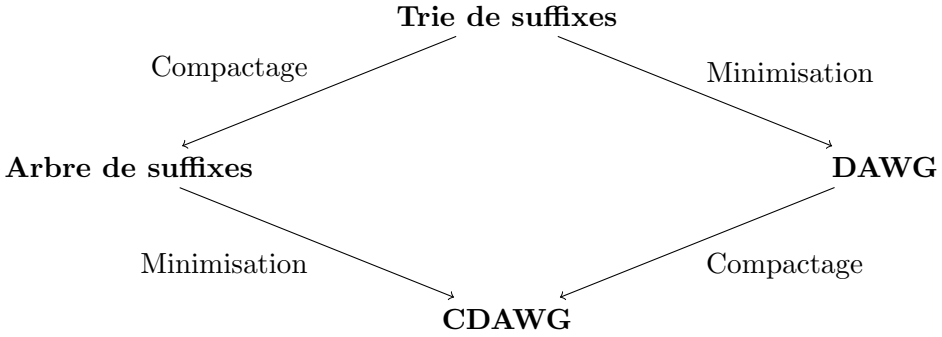


FIGURE 2.3 – Relations entre trie de suffixes, arbre de suffixes, DAWG et CDAWG.

Construction

Plusieurs algorithmes de construction de CDAWG ont été proposés [Blumer *et al.*, 1987; Crochemore and Vérin, 1997; Inenaga *et al.*, 2006]. L'algorithme proposé par [Inenaga *et al.*, 2006] est le plus récent.

Soit une séquence y, la construction directe de $CDAWG(y)$ [Inenaga *et al.*, 2006] se base sur l'algorithme de Ukkonen de construction d'arbres de suffixes [Ukkonen, 1995]. En effet,

Cet algorithme est de complexité $O(n)$ en temps de calcul et en espace mémoire, où n est la longueur de la séquence y.

Recherche d'un motif

L'algorithme de recherche d'un motif x dans une séquence y en utilisant $CDAWG(y)$ est similaire à celui utilisant $ST(y)$.

Le CDAWG pour un grand nombre de séquences, lorsqu'on considère des séquences d'ADN par exemple, nécessite un espace de stockage considérable [Blumer *et al.*, 1987]. Ceci dépend essentiellement de l'implémentation en pratique [Blumer *et al.*, 1987].

Une implémentation efficace du CDAWG donnée par [Holub and Crochemore, 2003] requiert de 1.7n à 5n octets dans le pire des cas pour une séquence de longueur n.

Les index mentionnés dans cette partie se basent sur la construction de structures arborescentes. Ces structures s'avèrent efficaces pour résoudre le problème de recherche d'un motif donné dans une séquence indexée. Cependant, en pratique, le stockage des nœuds ainsi que des liens suffixes est coûteux en espace mémoire. D'autres alternatives à ces structures existent, elles consistent à utiliser des tableaux au lieu des arbres tout en représentant les mêmes informations. Le premier tableau de ce type est le *tableau de suffixes* proposé par [Manber and Myers, 1993].

2.1.7 Algorithmes de Recherche de Motifs dans un Tableau de Suffixes

Dans cette partie, nous présentons un autre index appelé *tableau de suffixes*. Donnons sa définition.

Définition 2.8 *[Manber and Myers, 1993]*
Soit une permutation π sur les positions $\{0, 1, \ldots, n-1\}$ dans une séquence y, définie suivant l'ordre lexicographique des suffixes $y[i \,..\, n-1]$, pour $0 \leq i \leq n-1$. Le tableau de suffixes SA_y associé à y est un tableau à une seule dimension, indexé de 0 à $n-1$, tel que :

$$SA_y[i] = \pi_y(i)$$

$\forall i, 0 \leq i \leq n-1$.

La figure 2.4 montre un exemple d'un tableau de suffixes pour $y =$ `CTAGTTAG$`.

Construction

Plusieurs algorithmes de construction de tableaux de suffixes ont été proposés [Manber and Myers, 1993; Kim *et al.*, 2003; Ko and Aluru, 2003; Kärkkäinen and Sanders, 2003]. Les algorithmes proposés par [Kim *et al.*, 2003; Ko and Aluru, 2003; Kärkkäinen and Sanders, 2003] sont les plus sophistiqués.

La construction du tableau de suffixes SA_y sur les caractères de la séquence $y\$$ se fait récursivement comme suit [Kim *et al.*, 2003; Ko and Aluru, 2003; Kärkkäinen and Sanders, 2003] :

Suffixes	i	$SA_y[i]$	Suffixes triés
CTAGTTAG$	0	8	$
TAGTTAG$	1	6	AG$
AGTTAG$	2	2	AGTTAG$
GTTAG$	3	0	CTAGTTAG$
TTAG$	4	7	G$
TAG$	5	3	GTTAG$
AG$	6	5	TAG$
G$	7	1	TAGTTAG$
$	8	4	TTAG$

FIGURE 2.4 – Tableau de suffixes de la séquence $y =$ CTAGTTAG$. Comme nous supposons que $y[n-1] =$ $ est le plus petit caractère, nous avons toujours $SA_y[0] = n - 1$.

(*i*) D'abord, on divise les suffixes de y\$ en deux classes selon les valeurs de leurs positions ;

(*ii*) Ensuite, on effectue un tri sur les positions de chaque classe ;

(*iii*) Enfin, on fusionne les résultats obtenus lors des deux premières étapes pour obtenir SA_y.

Ces algorithmes sont de complexité $O(n)$ en temps de calcul et en espace mémoire, où n est la longueur de la séquence y.

Recherche d'un motif

La recherche d'un motif x dans une séquence y se fait en appliquant une recherche dichotomique sur SA_y (ceci est possible parce que les suffixes sont triés par ordre croissant de leurs positions dans y). On obtient ainsi un rang $SA_y[s, e]$ de tous les suffixes de la séquence y préfixés par x. De ce fait, les occurrences de x sont localisées dans les positions $SA_y[s], SA_y[s+1], \ldots, SA_y[e]$ de y. Le nombre d'occurrences de x est alors égal à $occ = e - s + 1$.

Cet algorithme est de complexité $O(m \log n)$ en temps de calcul, où m est la longueur du motif x.

Notons que la complexité de la recherche peut être réduite à $O(m + \log n)$ avec utilisation de tableaux supplémentaires [Manber and Myers, 1993]. Ainsi, le tableau de suffixes peut être augmenté avec le *tableau des plus longs préfixes communs*, en anglais *Longest Commun Prefixes*, noté *LCP*. Ce tableau permet de mémoriser la longueur de plus long préfixe commun à deux suffixes consécutifs. Bien que cette technique nécessite un espace mémoire additionnel, elle permet d'éviter de faire des comparaisons redondantes en particulier s'il existe de longs facteurs répétés dans la séquence indexée.

Puglisi et *al.* [Puglisi *et al.*, 2007] ont effectué des comparaisons entre deux types d'algorithmes de construction de tableaux de suffixes. Le premier type est formé par des

algorithmes de complexités théoriques non-linéaires [Itoh and Tanaka, 1999; Manzini and Ferragina, 2004; Schürmann and Stoye, 2005; Maniscalco and Puglisi, 2006; Larsson and Sadakane, 2007] et le second type est formé par des algorithmes de complexité théorique linéaire [Kim *et al.*, 2003; Ko and Aluru, 2003; Kärkkäinen and Sanders, 2003]. En pratique, les algorithmes du premier type sontt plus rapides que les algorithmes du second type. Ceci est dû au fait que les algorithmes du second type sont des algorithmes récursifs.

En plus du tableaux de suffixes, certains algorithmes peuvent considérer le *tableau de suffixes renversé*, en anglais *inverse suffix array*, noté ISA_y. Il s'agit de la permutation inverse de SA_y, ainsi $ISA_y[j]$ mémorise le *rank* lexicographique du j-iéme suffixe. Par exemple, le tableau de la figure 2.4 donne $ISA_y[0] = 3$.

Tous les index présentés dans cette partie permettent de représenter la séquence d'entrée, par le biais de ses suffixes, et répondent à des questions fondamentales en algorithmique du texte, dont la recherche d'un motif donné dans une séquence indexée.

Cependant, ces index sont coûteux en espace mémoire quand il s'agit d'indexer des séquences de grandes tailles comme les séquences issues des données NGS.

Ils existent d'autres index qui sont plus compressés. Ces index sont basés sur l'entropie de mots. On peut partitionner ces index en deux classes [Salson, 2009b] :

(a) Index basés sur l'échantillonnage des tableaux de suffixes ;

(b Index basés sur les algorithmes de compression sans perte.

2.2 Algorithmes de Recherche de Motifs dans un Index Compressé

Nous présentons dans cette section, des index compressés de type auto-index. Ils donnent la possibilité de récupérer la séquence originale après compression et permettent de lire son contenu sans avoir recours à la décompression.

Dans cette thèse, nous choisissons de présenter le *FM-index* [Ferragina and Manzini, 2005] et le *tableau de suffixes compressé* [Sadakane, 2003; Grossi *et al.*, 2003]. Ces index permettent, après échantillonnage, de récupérer la valeur $SA_y[i]$ pour tout indice i.

2.2.1 Algorithmes de Recherche de Motifs dans un FM-index

Le *FM-index* a été introduit par Ferragina et Manzini [Ferragina and Manzini, 2000; Ferragina and Manzini, 2005]. Il est défini comme suit :

Définition 2.9 *[Ferragina and Manzini, 2000; Ferragina and Manzini, 2005] Le FM-index est un index composé d'une représentation compressée d'une séquence à partir de la Transformée de Burrows-Wheeler, en anglais Burrows-Wheeler Transform (BWT) [Burrows and Wheeler, 1994], et du tableau de suffixes échantillonné.*

Construction

La construction de la BWT de y, notée y^{bwt}, se fait de la manière suivante [Ferragina and Manzini, 2000; Ferragina and Manzini, 2005] :

(i) Durant la première étape, on construit une matrice M_y, de dimension $n \times n$, où la i-ème ligne de M_y, $0 \leq i \leq n-1$, contient le décalage circulaire d'ordre i de la séquence y ;

(ii) Durant la seconde étape, les lignes de la matrice sont triées selon l'ordre lexicographique des séquences stockées dans ces lignes ;

(iii) Enfin, durant la dernière étape, la dernière colonne de la matrice M_y ainsi obtenue représente y^{bwt}.

La première et la dernière colonne de M_y sont respectivement notées F_y et L_y.

La matrice M_y et le tableau de suffixes SA_y se ressemblent. Cette relation étroite est donnée par l'équation suivante :

$$y^{bwt}[i] = \begin{cases} \text{y}[SA_y[i]-1] & \text{si } SA_y[i] \geq 1 \\ \text{y[n-1]= \$} & \text{si } SA_y[i] = 0. \end{cases} \tag{2.1}$$

La construction de la BWT de y se fait ainsi à partir de SA_y comme suit [Ferragina and Manzini, 2000; Ferragina and Manzini, 2005] :

(i) D'abord, on construit SA_y ;

(ii) Enfin, on considère les caractères qui juste précèdent les suffixes de SA_y dans y.

Exemple :

Considérons la séquence $y =$ `CTAGTTAG$`.

`C T A G T T A G $`	$\$\ C_0\ T_0\ A_0\ G_0\ T_1\ T_2\ A_1\ G_1$
`T A G T T A G $ C`	$A_1\ G_1\ \$\ C_0\ T_0\ A_0\ G_0\ T_1\ T_2$
`A G T T A G $ C T`	$A_0\ G_0\ T_1\ T_2\ A_1\ G_1\ \$\ C_0\ T_0$
`G T T A G $ C T A`	$C_0\ T_0\ A_0\ G_0\ T_1\ T_2\ A_1\ G_1\ \$$
`T T A G $ C T A G`	$G_1\ \$\ C_0\ T_0\ A_0\ G_0\ T_1\ T_2\ A_1$
`T A G $ C T A G T`	$G_0\ T_1\ T_2\ A_1\ G_1\ \$\ C_0\ T_0\ A_0$
`A G $ C T A G T T`	$T_2\ A_1\ G_1\ \$\ C_0\ T_0\ A_0\ G_0\ T_1$
`G $ C T A G T T A`	$T_0\ A_0\ G_0\ T_1\ T_2\ A_1\ G_1\ \$\ C_0$
`$ C T A G T T A G`	$T_1\ T_2\ A_1\ G_1\ \$\ C_0\ T_0\ A_0\ G_0$

(a) M_y non triée lexicographiquement

$\boldsymbol{F_y}$								$\boldsymbol{L_y}$
$\$$	C_0	T_0	A_0	G_0	T_1	T_2	A_1	G_1
A_1	G_1	$\$$	C_0	T_0	A_0	G_0	T_1	T_2
A_0	G_0	T_1	T_2	A_1	G_1	$\$$	C_0	T_0
C_0	T_0	A_0	G_0	T_1	T_2	A_1	G_1	$\$$
G_1	$\$$	C_0	T_0	A_0	G_0	T_1	T_2	A_1
G_0	T_1	T_2	A_1	G_1	$\$$	C_0	T_0	A_0
T_2	A_1	G_1	$\$$	C_0	T_0	A_0	G_0	T_1
T_0	A_0	G_0	T_1	T_2	A_1	G_1	$\$$	C_0
T_1	T_2	A_1	G_1	$\$$	C_0	T_0	A_0	G_0

(b) M_y triée lexicographiquement

La BWT est réversible, i.e., on peut récupérer la séquence originale y à partir de y^{bwt}. Ceci peut se faire grâce à la propriété fondamentale de la BWT, i.e., l'ordre d'apparition des caractères dans les colonnes F_y est L_y est préservé. En effet, la i-ème occurrence d'un caractère c dans L_y correspond à la la i-ème occurrence de c dans F_y.

Cette propriété constitue la base d'une fonction appelée *Fonction LF* [Burrows and Wheeler, 1994]. On utilise la fonction LF afin d'associer les caractères mémorisés dans L_y à ceux mémorisés dans F_y. On définit cette fonction comme suit.

Définition 2.10 *[Burrows and Wheeler, 1994; Ferragina and Manzini, 2005] Soit* $c = y^{bwt}[i]$*, la fonction LF est définie comme suit :*

$$LF(i) = Count[c] + rank_c(y^{bwt}, i) - 1 \quad (2.2)$$

où $Count[c]$ *est le nombre total de caractères dont l'ordre lexicographique est inférieur à celui de* c *et* $rank_c(y^{bwt}, i)$ *est le nombre total d'occurrences de* c *dans* y^{bwt} *apparaissant avant la position* i*.*

Recherche d'un motif

La recherche d'un motif x dans une séquence y se fait en utilisant un algorithme de recherche appelé *Algorithme de Recherche en Arrière*, en anglais *Backward Search* [Burrows and Wheeler, 1994]. On procède comme suit :

(i) D'abord, on calcule le rang $M_y[s_{m-1}, e_{m-1}]$ de tous décalages circulaires préfixés par $x[m-1]$;

(ii) Puis, tenant en compte du caractère $x[m-2]$, on calcule le rang $M[s_{m-2}, e_{m-2}]$ de tous les décalages circulaires préfixés par $x[m-2\,..\,m-1]$;

(iii) Ensuite, on réitère inductivement de façon à obtenir à l'itération i, $0 \leq i \leq n-1$, le rang $M_y[s_i, e_i]$ de tous les décalages circulaires préfixés par $x[i\,..\,m-1]$. On utilise la fonction LF pour mettre à jour le rang $M_y[s_{i+1}, e_{i+1}]$ obtenu à l'itération précédente, i.e., toutes les occurrences de $x[i]$ dans $y^{bwt}[s_{i+1}, e_{i+1}]$ apparaissent de façon contiguë dans F_y. Un déplacement en arrière avec LF permet d'obtenir $M_y[s_i, e_i]$;

(iv) Enfin, lors de la dernière itération on a deux situations possibles :

(a) Soit, on obtient le rang $M_y[s_0, e_0]$ de tous les décalages circulaires préfixés par x. Le nombre d'occurrences de x correspond alors à la longueur du rang résultant ;

(b) Soit, on obtient un rang vide. Aucune ligne de M_y n'est alors préfixée par x.

Rappelons que M_y et SA_y sont étroitement liés, ainsi, le résultat obtenu consiste à trouver tous les suffixes de y préfixés par x, s'il en existe.

Cet algorithme est de complexité en $O(mn)$ en temps de calcul, où m est la longueur du motif x.

Comme on peut le remarquer, la BWT ne donne aucune information sur la localisation d'un motif x dans une séquence y. L'idée de base pour récupérer les positions des suffixes préfixés par x consiste à sauvegarder alors un échantillon de SA_y, ensuite à appliquer la fonction LF pour calculer les valeurs non échantillonnées, i.e., toutes les valeurs des positions multiples de $\log^{1+\epsilon}(n)$, où $\epsilon > 0$ est une constante, sont échantillonnées. Supposons qu'on souhaite récupérer la valeur $SA_y[i]$ on a $SA_y[i] = SA_y[LF^k(i)] + k$ où k le nombre de fois où la fonction LF a été appliquée [Navarro and Mäkinen, 2007].

Ferragina et Manzini [Ferragina and Manzini, 2000] montrent que le FM-index compresse y^{bwt} en, au plus, $5nH_k(y)+o(n)$ bits, où H_k est le k-ème ordre de l'entropie empirique

de la séquence y, pour une valeur fixée $k \geq 0$ (négligeable). La recherche d'occurrences est résolue en $O(m + \log^{\epsilon} n)$ pour une constante $\epsilon > 0$.

En pratique, Ferragina et Manzini modifient légèrement l'implémentation théorique pour réduire l'espace requis[Ferragina and Manzini, 2001]. Il est ainsi possible de stocker le génome humain *hg19* avec un espace mémoire de moins de 2 GO [Ferragina *et al.*, 2009]. Une alternative de FM-index qui utilise les *Arbres à Ondelettes*, en anglais *Wavelet trees* [Navarro and Mäkinen, 2007], a été plus tard proposée par [Grossi *et al.*, 2003; Ferragina *et al.*, 2004]. Sa complexité théorique en espace mémoire est $nH_0(y) + O(n \log \sigma)$ bits. En pratique, l'utilisation des *Arbres d'Huffman en forme d'Ondelettes*, en anglais *Huffman-shaped wavelet trees* [Mäkinen and Navarro, 2005], nécessite $n(H_0(y) + 1) + o(n \log \sigma)$ bits en espace mémoire et $O(H_0(y) + 1)$ en temps de calcul. Ferragina et Manzini sont allés plus loin et ont donné une version surprenante du FM-index [Ferragina *et al.*, 2007]. Cette nouvelle version nécessite $nH_k(y) + o(n \log \sigma)$ bits en espace mémoire, pour $k \leq \alpha \log_{\sigma} n$, pour une constante $0 < \alpha < 1$, et $\sigma = O(\text{polylog}(n))$.

2.2.2 Algorithmes de Recherche de Motifs dans un Tableau de Suffixes Compressé

Dans cette partie, nous présentons un index appelé *Tableau de Suffixes Compressé*, en anglais *Compressed Suffix Array* (CSA).

La définition de *CSA(y)* se base sur celle d'une fonction appelée ψ. Donnons alors la définition de la fonction ψ

Définition 2.11 *[Grossi and Vitter, 2000] Soient deux entiers i et j , tels que $0 \leq i,\ j \leq n-1$, on a $\psi(i) = j$ si $SA(y)[j] = SA(y)[i] + 1 \bmod n$.*

Maintenant, donnons une définition de *CSA(y)*.

Définition 2.12 *[Grossi and Vitter, 2000] Étant donnée une séquence y, le CSA(y) est un auto-index permettant une représentation plus économique en espace mémoire des informations contenues dans $SA(y)$.*

Le CSA(y) supporte les opérations basiques suivantes :

(a) Compresser $SA(y)$ de façon à ce que ce dernier soit mis en rebut tandis que la séquence y est retenue,

(b) Pour tout $0 \leq i \leq n-1$, trouver $SA(y)[i]$ à partir de CSA(y).

Construction

Plusieurs algorithmes de construction d'un CSA ont été élaborés [Grossi and Vitter, 2000; Sadakane, 2003; Grossi *et al.*, 2003; Grossi *et al.*, 2004; Grossi and Vitter, 2005]. Dans cette thèse, nous choisissons de décrire l'algorithme *Grossi-CSA* [Grossi and Vitter, 2000].

Soit une séquence y, en adoptant l'algorithme *Grossi-CSA* [Grossi and Vitter, 2000] la construction de *CSA(y)* se fait en quarte étapes principales :

(*i*) D'abord, on crée un vecteur de bits $B_k[0, n_k]$, où $n_k = \lceil n/2^k \rceil$, tel que $B_k[i] = 1$ si $SA_k(y)[i]$ est pair, 0 sinon ;

(*ii*) Ensuite, on calcule la fonction ψ_k pour les valeurs impaires de $SA_k(y)$;

(*iii*) Puis, on utilise $rank_1(B_k, i)$ pour calculer le nombre de 1 dans B_k jusqu'à l'indice i ;

(*iv*) Enfin, on calcule $SA_{k+1}(y)$ en divisant les valeurs paires mémorisées dans $SA_k(y)$ par 2.

On répète ce processus jusqu'à obtenir le plus petit SA, $SA_{k_{\min}}(y)$, et tous les ψ_k et B_k pour $0 \leq k \leq k_{\min} - 1$.

Cet algorithme est de complexité $O(n)$ en temps de calcul et en espace mémoire, où n est la longueur de la séquence y.

Pour accéder à $SA_k(y)[i]$, on vérifie si $B_k[i] = 1$. Deux cas sont possibles :

(*a*) Soit $B_k[i] = 1$ alors $SA_k(y)[i]$ est paire et échantillonnée dans $SA_{k+1}(y)$. On a $SA_k(y)[i] = 2SA_{k+1}(y)[rank_1(B, i) - 1]$;

(*b*) Soit $B_k[i] = 0$ alors $SA_k(y)[i]$ est impaire et $\psi(i)$ est paire et échantillonnée dans $SA_{k+1}(y)$. On a $SA_k(y)[i] = SA_k(y)[\psi(i)] - 1$.

Exemple :

Soit $y =$ `CTAGTTAG$`.

	0	1	2	3	4	5	6	7	8
$\boldsymbol{SA}_0(y)$	8	6	2	0	7	3	5	1	4
B_0	1	1	1	1	0	0	0	0	1
					0	1	2	3	
ψ_0					0	8	1	2	
	0	1	2	3					4
$\boldsymbol{SA}_1(y)$	4	3	1	0					2
B_1	1	0	0	1					1
		0	1						
ψ_1		0	4						
	0			1					2
$\boldsymbol{SA}_2(y)$	2			0					1

FIGURE 2.6 – Étapes de construction de $SA_{k_{\min}(y)=SA_2(y)}$.

Recherche d'un motif

La recherche d'un motif x dans une séquence y en utilisant *CSA(y)* [Grossi and Vitter, 2000] est similaire à la recherche en utilisant $SA(y)$.

Cet algorithme est de complexité $O((m+\log^{\epsilon} n)\log n))$, où m est la longueur du motif x et $0 < \epsilon \leq 1$ est une valeur fixée.

2.2.3 Algorithmes de Recherche de Motifs dans un LZ-index

Dans cette partie, nous présentons un index appelé *LZ-index* (LZI). Mais avant de définir le *LZI(y)*, nous décrivons la compression LZ78 [Ziv and Lempel, 1978] :

La compression LZ78 [Ziv and Lempel, 1978] d'une séquence y permet de découper y de gauche à droite en facteurs appelés *phrases*. Ces phrases sont distinctes, non-chevauchantes et les plus longues possible, i.e., si on retire le dernier caractère d'une phrase, celle-ci perd son unicité. Chaque phrase possède un numéro. Le mot ε correspond à la phrase w_{-1}. La séquence y s'écrit alors $y = w_0 \cdots w_{n'-1}$, tel que :

(a) $w_i \neq w_j$, pour $0 \leq i < n'$ et $-1 \leq j < i$,

(b) Pour $0 \leq i < n'$ et $0 \leq j < i$, on a chaque préfixe propre de w_i correspond à une phrase w_j.

Maintenant, donnons la définition de *LZI(y)*.

Définition 2.13 *[Kärkkäinen and Ukkonen, 1996] Soit une séquence y, le LZI(y) est un auto-index basé sur la compression LZ78 de y. Il utilise un arbre de suffixes qui indexe seulement le début des phrases obtenues par la compression LZ78 de y.*

Construction

Plusieurs algorithmes de construction de LZI ont été proposés [Kärkkäinen and Ukkonen, 1996; Navarro, 2002; Ferragina and Manzini, 2005; Arroyuelo *et al.*, 2006]. L'algorithme proposé par Arroyuelo et *al.* [Arroyuelo *et al.*, 2006] est le plus économique en espace mémoire et le LZI construit permet une recherche plus rapide d'un motif. C'est ce dernier que nous présentons dans ce qui suit.

Soit une séquence y de longueur n, en adoptant l'algorithme de [Arroyuelo *et al.*, 2006] la construction de *LZI(y)* se fait de la manière suivante :

(i) D'abord, on construit un trie, appelé *LZ-trie*, sur les phrases $\{w_0, \ldots, w_{n-1}\}$;

(ii) Ensuite, on construit un trie, appelé *RevTrie*, sur les phrases renversées $(\{w^{\sim}{}_0 \ \ldots, w^{\sim}_{n-1}\})$;

(iii) Puis, on construit une structure, appelée *Node*, permettant d'associer l'identifiant de chaque w_i au nœud qui lui correspond dans le *LZ-trie* ;

(iv) Enfin, on construit une matrice de taille $(n+1) \times (n+1)$, appelée *Range*, pour mémoriser $\{(revpos(\tilde{w}_k), pos(\tilde{w}_{k+1})\}, k \in \{0, \ldots, n-1\})$, où *revpos* et *pos* sont les positions lexicographiques dans respectivement $\{w_0 \ldots w_n\}$ et $\{w_0^{\sim} \ldots w_n^{\sim}\}$.

Cet algorithme est de complexité $O(n \log n(1 + o(1)))$ en espace mémoire. Au total, *LZI(y)* nécessite $4nH_k(y) + o(n \log \sigma)$ bits pour $k = o(\log_\sigma n)$.

Recherche d'un motif

En adoptant l'algorithme proposé par [Arroyuelo *et al.*, 2006], la recherche d'un motif x dans une séquence y en utilisant *LZI(y)* se fait en distinguant trois types d'occurrences

de x :

(a) Soit x est contenu dans une seule phrase w_k et x n'est pas un suffixe de w_k. Dans ce cas, il existe w_ℓ qui contient x tel que $w_k = w_\ell \cdot c$. Parmi les phrases qui sont préfixes de w_ℓ, il en existe une, notée w_j, qui se termine par x. On opère alors comme suit :

(i) Durant la première étape, on cherche $x^\sim$ dans le *RevTrie*. Si on trouve un chemin partant de la racine étiqueté par les caractères de $x^\sim$ alors on atteint le nœud j qui représente w_j ;

(ii) Durant la seconde étape, on récupère tous les descendants du nœud j à partir du *LZ-trie*. On réitère ce processus pour tous ces descendants en utilisant le *RevTrie* ;

(b) Soit x apparait sur deux phrases consécutives w_k et w_{k+1} tel que $x[0\,..\,i]$ est un suffixe de w_k et $x[i+1\,..\,m-1]$ est un préfixe de w_{k+1}. On opère alors comme suit :

(i) D'abord, on cherche $x[0\,..\,i]^\sim$ en utilisant le *RevTrie*. On obtient ainsi le rang des phrases renversées se terminant par $x[0\,..\,i]$;

(ii) Puis, on cherche $x[i\,..\,m-1]$ en utilisant le *LZ-trie*. On obtient ainsi le rang des phrases commençant par $x[i\,..\,m-1]$;

(iii) Enfin, on utilise la structure *Range* pour trouver le couple $(k, k+1)$,

(c) Soit x est contenu dans la plus longue concaténation $w_k \cdots w_\ell$ tel que $x[i\,..\,j] = w_{k+1} \cdots w_{\ell-1}$, $x[0\,..\,i-1]$ est un suffixe de w_k et $x[j\,..\,m-1]$ est un préfixe de w_ℓ. On opère alors comme suit :

(i) D'abord, on cherche chaque facteur $x[i\,..\,j]$ dans le *LZ-trie* pour obtenir la plus longue concaténation $w_k \cdots w_\ell$ qui est égale à $x[i\,..\,j]$;

(ii) Puis, pour chaque concaténation obtenue on utilise une opération, appelée *parent*, pour w_k permettant de vérifier si w_{k-1} se termine par $x[0\,..\,i-1]$ et si $w_{\ell+1}$ commence par $x[j+1\,..\,m-1]$. Si c'est le cas alors on a détecté une occurrence de x.

Cet algorithme est de complexité $O(m^3 \log \sigma + (m + occ) \log n)$ en temps de calcul dans le pire des cas, où m est la longueur du motif x et occ est le nombre totale de ses occurrences.

2.3 Algorithmes de Recherche de Motifs dans un Index Avancé

Dans cette partie, nous présentons des index *avancés*. Ces index sont directement dédiés à l'indexation des ensembles de séquences d'ADN fortement similaires. Pour représenter un ensemble de séquences d'ADN fortement similaires, ces index se basent sur une compression des séquences selon une séquence de référence, notée y_0, arbitrairement choisie.

2.3.1 Algorithmes de Recherche de Motifs dans un Index Basé sur les Segments Communs et les Segments Différenciés

Nous choisissons de représenter deux index proposés par [Huang *et al.*, 2010; Alatabbi *et al.*, 2012]. Ces index permettent de représenter un ensemble de séquences d'ADN fortement similaires, en construisant une structure sur tous les facteurs de y_0 et une autre structure pour les facteurs non communs à y_0 contenus dans les autres séquences.

Algorithme de [Huang *et al.*, 2010]

Dans cette partie, nous présentons l'index proposé par [Huang *et al.*, 2010] que nous désignons par HRBI. Avant de définir $HRBI(Y)$, nous présentons les deux modèles de similarité des séquences d'entrée qu'il considère :

En fait, $HRBI(Y)$ représente Y selon deux modèles différents [Huang *et al.*, 2010] :

(a) **Modèle1** : Soit $Y = \{y_0, y_1, \ldots, y_{r-1}\}$ un ensemble de r séquences non nécessairement de même longueur. Chaque y_i est de la forme $R_{i,0}C_0R_{i,1}C_1 \cdots C_{k-1}R_{i,k}$, où $R_{i,j}$ $(0 \leq j \leq k)$ est un *segment différencié* appelé aussi un *R-segment*, i.e., un segment qui diffère du segment correspondant dans y_0, et C_j $(0 \leq j \leq k-1)$ est un *segment commun*, i.e., un segment qui appartient à toutes les séquences de Y. Chaque segment $R_{i,j}$ possède un rang dans R. Un segment $R_{i,j}$ peut être vide. Pour tout $i \neq i'$, $R_{i,j}$ et $R_{i',j}$ n'ont pas nécessairement la même longueur. Un *suffixe différencié* est un suffixe commençant dans un R-segment. Il est représenté par une paire (w, s) tel que w est le numéro du R-segment et s est sa position de début.

(b) **Modèle2** : Soit $Y = \{y_0, y_1, \ldots, y_{r-1}\}$ un ensemble de r séquences de même longueur n. Chaque y_i $(i \neq 0)$ diffère de y_0 dans p_i positions, où $p_i << n$. Ces positions ne sont pas forcement les mêmes pour $y_{i'}$, où $i \neq i'$. Un *suffixe différencié* est un suffixe qui commence à une des positions p_i.

Définissons alors *HRBIY(Y)*.

Définition 2.14 *[Huang* et al.*, 2010] Soit un ensemble de séquences fortement similaires $Y = \{y_0, y_1, \ldots, y_{r-1}\}$, le HRBI(Y) est un index compact qui permet d'indexer séparément les segments communs et les segments différenciés. Le HRBI(Y) se base sur la BWT et le tableau de suffixes.*

Construction

Commençons par une construction de $HRBI(Y)$ suivant le modèle 1 [Huang *et al.*, 2010].

Modèle 1 : Soit $C = C_0\$C_1\$\cdots\$C_{k-1}\$$ la concaténation des segments communs séparés par un marqueur de fin \$, où $|C| = n' + k$ où $n' = \sum_{j=0}^{k-1} |C_j|$. En adoptant l'algorithme de [Huang *et al.*, 2010], la construction de $HRBI(Y)$ sur C se fait comme suit :

(i) Durant la première étape, on crée un tableau $Start_C$ tel que $Start_C[j]$ est la position de début du segment C_j dans C;

(ii) Durant la seconde étape, on construit la BWT de $C^{\sim}$.

Soit $R = R_{0,0} \cdots R_{0,k}R_{1,0} \cdots R_{1,k} \cdots R_{r-1,0} \ldots R_{r-1,k}$, la concaténation des R-segments de toutes les séquences y_i, où $|R| = N'$. En adoptant l'algorithme de [Huang *et al.*, 2010], la construction de $HRBI(Y)$ sur R se fait comme suit :

(i) D'abord, on construit un tableau $Start_R$ tel que $Start_R[j]$, $0 \leq j \leq k$, est la position de début du j-ème R-segment dans R;

(ii) Puis, on construit un tableau de suffixes SAR tel que $SAR[i]$, $0 \leq i \leq r-1$, est le plus petit suffixe différencié d'ordre lexicographique i;

(*iii*) Enfin, on crée un tableau SAR_0 qui mémorise les entrées de SAR dont la position de début est 0. De plus, SAR_0 mémorise un champ *c-rank* tel que pour $R_{i,j}$ le *c-rank* correspondant est égale au rang du suffixe $\$C^{\sim}_{j-1}\$\cdots\$C^{\sim}_0$ dans $C^{\sim}$. Si $j = 0$ alors *c-rank* $= -1$.

Cet algorithme est de complexité $O(n' + N' \log rk + rk(\log n' + \log N'))$ en espace mémoire.

Exemple :

(a) Les segments communs et les R-segments (soulignés).

$$\begin{array}{lcl} y_0 & = & \underline{\texttt{A}}\texttt{ACG}\underline{\texttt{CG}}\texttt{CCG}\underline{\texttt{G}} \\ y_1 & = & \underline{\texttt{C}}\texttt{ACG}\underline{\texttt{AG}}\texttt{CCG}\underline{\texttt{G}} \\ y_2 & = & \underline{\texttt{T}}\texttt{ACG}\underline{\texttt{AT}}\texttt{CCG}\underline{\texttt{C}} \end{array} \qquad \begin{array}{lcl} C_0 & = & \texttt{ACG} \\ C_1 & = & \texttt{CCG} \end{array} \qquad \begin{array}{lcl} R_{0,0} & = & \texttt{A} \\ R_{0,1} & = & \texttt{CG} \\ \vdots & & \\ R_{2,2} & = & \texttt{C} \end{array}$$

(b) Construction des séquences R, C, des tableaux $Start_C$ et $Start_R$.

$R =$ A CG G C AG G T AT C $\qquad Start_R :$

0	1	3	4	5	6	7	8	9
0	1	3	4	5	7	8	9	11

$C =$ ACGCCG

$C^{\sim} =$ GCCGCA

$Start_C :$

0	1
0	4

(c) Tous les suffixes différenciés triés dans l'ordre lexicographique et construction des tableaux SAR, SAR_0 et du champ *c-rank*.

ordre	suffixe	SAR
0	AACGCGCCGG	(0,0)
1	AGCCGG	(4,0)
2	ATCCGC	(7,0)
3	C	(8,0)
4	CACGAGCCGG	(3,0)
5	CGCCGG	(1,0)
6	G	(2,0)
7	G	(5,0)
8	GCCGG	(1,1)
9	GCCGG	(4,1)
10	TACGATCCGC	(6,0)
11	TCCGC	(7,1)

SAR_0	*c-rank*
(0,0)	-1
(4,0)	0
(7,0)	0
(8,0)	1
(3,0)	-1
(1,0)	0
(2,0)	1
(5,0)	1
(6,0)	-1

FIGURE 2.7 – Constructions de $HRBI(Y)$ pour le modèle 1.

Passons maintenant à une construction de $HRBI(Y)$ suivant le modèle 2 [Huang *et*

al., 2010].

Modèle 2 : Soit un ensemble Y de séquences fortement similaires, la construction de $HRBI(Y)$ suivant le modèle 2 [Huang *et al.*, 2010] est similaire à celle avec le modèle 1. Ce qu'on doit distinguer c'est que, durant la première étape, on construit la BWT de $y_0^{\sim}$. En outre, durant la dernière étape, on construit un tableau B tel que pour v et v' les rangs de respectivement $y_i[j\,..\,n-1]$ dans SAR_0 et de $(y_0[0\,..\,j-1])^{\sim}$ dans $y_0^{\sim}$, on a $B[v] = v'$,

Cet algorithme est de complexité $O(n+N'\log N'+s(\log n+\log N'))$ en espace mémoire, où s est le nombre des R-segments.

Recherche d'un motif

Commençons par la recherche d'un motif x selon le modèle 1.

Modèle 1 : En adoptant l'algorithme proposé par [Huang *et al.*, 2010], la recherche d'un motif x dans un ensemble Y de séquences fortement similaires représenté par le modèle 1 [Huang *et al.*, 2010] se fait en distinguant trois types d'occurrences :

(a) Soit x est contenu entièrement dans un segment commun. On cherche alors $x^{\sim}$ le miroir de x en utilisant la BWT de $C^{\sim}$ le miroir de C;

(b) Soit x un préfixe d'un R-segment : On cherche alors x en utilisant SAR;

(c) Soit $x = x_1 \cdot x_2$, tel que x_1 est un suffixe d'un segment C_{j-1} et x_2 est un préfixe du segment $R_{i,j}$ qui suit dans y_i. On opère alors comme suit :

(i) D'abord, on cherche $(x_1\$)^{\sim}$ en utilisant la BWT de $C^{\sim}$ pour trouver les segments communs dont x_1 est suffixe. On obtient alors un rang LR_1 ;

(ii) Ensuite, on cherche x_2 en utilisant SAR_0, pour trouver les occurrences de x_2 commençant par le premier caractère de $R_{i,j}$. Obtient alors un rang LR_2 ;

(iii) Enfin, on vérifie pour chaque valeur dans LR_1 qui se réfère à C_{j-1} s'il existe une valeur dans LR_2 qui pointe sur le segment $R_{i,j}$ suivant dans y_i. Pour ce faire, on calcule *c-rank* pour chaque valeur de LR_2 en utilisant un tableau à deux dimensions [Nekrich, 2007].

Cet algorithme est de complexité $O(m + PSC(x)(m\log(rk)) + occ_3\log n)$ en temps de calcul, où $PSC(x)$ est le nombre de préfixes de x qui sont suffixes de certains segments communs.

Modèle 2 : La recherche d'un motif x dans un ensemble Y de séquences fortement similaires est similaire à celle utilisant le modèle 1 [Huang *et al.*, 2010].

Cet algorithme est de complexité $O(m+m\log N'+PSC'(x)(m\log s+s)+occ)$ en temps de calcul, où $N' = |R|$ et $PSC'(x)$ est le nombre des préfixes de x qui apparaissent dans y_0.

Algorithme de [Alatabbi *et al.*, 2012]

Dans cette partie, nous présentons l'index proposé par [Alatabbi *et al.*, 2012] que nous désignons par *ARBI*. Avant de définir $ARBI(Y)$, nous décrivons le modèle de séquences fortement similaires d'entrée que considèrent [Alatabbi *et al.*, 2012] :

Soit $Y = \{y_0, \ldots, y_{r-1}\}$ un ensemble de séquences, chaque y_i est représentée sous forme de concaténation de *permutations de blocs identiques* séparées par des différences de longueur bornée. Formellement, $y_i = e_{i_1} C_{\sigma_1} e_{i_2} C_{\sigma_2} \cdots e_{i_k} C_{\sigma_k}$, où $|e_j| \leq \omega/2$ et ω est la longueur d'un mot machine.

Définissons maintenant l'index $ARBI(Y)$:

Définition 2.15 *[Alatabbi* et al.*, 2012] Étant donné un ensemble Y de séquences fortement similaires, ARBI(Y) est un index contient :*

(a) Tous les blocs identiques $C = C_1 C_2 \cdots C_k$,

(b) Pour chaque y_i, la permutation $\tau_i = \sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_k}$ et tous les segments différenciés $e_i = e_{i_1}\cdots e_{i_k}$.

ARBI(Y) se base sur la construction de tableaux de suffixes et de l'automate d'Aho-Corasick [Aho and Corasick, 1975] pour représenter les blocs identiques et sur un codage en 2 bits pour représenter les segments différenciés.

Construction

Soit un ensemble de séquences fortement similaires Y, en adoptant l'algorithme de [Alatabbi *et al.*, 2012], la construction de $ARBI(Y)$ se fait de la manière suivante :

(i) D'abord, on construit SA_C et l'automate d'Aho-Corasick des blocs identiques $C_1, \ldots, C_k$;

(ii) Puis, on utilise un codage binaire pour représenter chaque segment différencié par des entiers.

Recherche d'un motif

En adoptant l'algorithme de [Alatabbi *et al.*, 2012], la recherche d'un motif x dans l'ensemble Y, se fait selon les quatre types d'occurrences de x suivants :

(a) **Occurrence simple** :

(a_1) Soit x apparait entièrement dans un bloc identique. On cherche alors x dans SA_C,

(a_2) Soit x apparait entièrement dans un segment différencié. On utilise alors une opération qui retourne la distance de Hamming entre deux mots en temps constant.

(b) **Occurrence bornée** :

Soit x est de la forme $C'_{x_{j-1}} e_j C''_{x_j}$, où $C'_{x_{j-1}}$ est un suffixe de $C_{x_{j-1}}$ et C''_{x_j} est un préfixe de C_{x_j}, ou bien x est de la forme $C'_{x_{j-1}} e''_j$ où $C'_{x_{j-1}}$ est un suffixe de $C_{x_{j-1}}$ et e''_j est un préfixe de e_j.

Dans ce cas, on opère de façon similaire que le type d'occurrences simples.

(c) **Occurrence complexe** :

soit x est de la forme $e'_1 C_{x_1} e'_2 C_{x_2} \cdots e'_n C_{x_n}$, où e'_j est un suffixe de e_j, où $1 \leq j \leq n$. Soit une *factorisation valide* de x de la forme $C'_{x_0} e_1 C_{x_1} e_2 \cdots e_v C_{x_v} e''_{v+1}$, où C'_{x_0} est un suffixe d'un bloc identique et e''_{v+1} est un préfixe d'un segment différencié. Dans ce cas, on opère comme suit :

(i) D'abord, on cherche toutes les factorisations valides de x à l'aide de l'automate d'Aho-Corasick de $C_1, \ldots, C_k$. On obtient alors toutes les occurrences des C_i dans x. Ensuite, on vérifie si une factorisation valide de x commence dans un bloc identique ou dans un segment différencié

(ii) Puis, on cherche les séquences y_i, $0 \leq i \leq r-1$, qui contiennent les permutations $\tau_x = x_1 \cdots x_v$ qui correspond à l'ordre d'apparition des blocs identiques dans x. Ainsi, on construit un automate d'Aho-Corasick sur les factorisations valides et on l'alimente par les τ_i

(iii) Ensuite, pour chaque occurrence de τ_x, on vérifie les différences entre les C_{x_j}, où $1 \leq j \leq v$. Ceci se fait de la même façon que la recherche d'occurrences simple dans un segment différencié.

(iv) Enfin, on valide le début et la fin de chaque séquence y_i. Ceci se fait en utilisant le même processus que la recherche bornée. En outre, on utilise un vecteur binaire pour représenter le motif x et le vérifier par des opérations binaires.

(d) **Occurrence complexe bornée** :

Soit x est de la forme $x = C'_{x_0} e_1 C_{x_1} \cdots e_v C_{x_v} e''_{v+1}$, où C'_{x_0} est un suffixe de C_{x_0} et e''_{v+1} est un préfixe de e_{v+1}. Dans ce cas, on opère de la même façon que le type d'occurrences complexes.

L'algorithme de [Alatabbi *et al.*, 2012] est de complexité $O(m + vk \log k + occ_v(m/\omega + v) + (PSC(p)mr)/\omega + \log n)$ en temps de calcul et $O(n' \log n' + kh + vh \log vh)$ en espace mémoire, où v est le nombre de factorisations valides de longueur h, k est le nombre de différences, occ_v est le nombre d'occurrences de factorisations valides, $PSC(x)$ est le nombre de préfixes de x qui sont suffixes de blocs identiques et n' est la longueur totale des blocs identiques.

2.3.2 Algorithmes de Recherche de Motifs dans un Auto-index Basé sur la Compression Lempel-Ziv Relative

Dans cette partie, nous présentons un index proposé par Do et *al.* [Do *et al.*, 2012] que nous désignons par RLZ-index. Cet index se base sur une nouvelle alternative de factorisation de séquences, appelée *Relative Lempel-Ziv scheme* (RLZ) [Kuruppu *et al.*, 2010]. Avant de définir *RLZ-index*, décrivons la compression RLZ [Kuruppu *et al.*, 2010].

Soit un ensemble de séquences $Y = \{y_0, \ldots, y_{r-1}\}$, la factorisation RLZ de l'ensemble $Y' = \{y_1, \ldots, y_{r-1}\}$ selon la séquence de référence y_0, notée $\mathrm{LZ}(Y'/y_0)$, est une représentation de Y' par des phrases, non nécessairement distinctes et maximales, égales à des facteurs de y_0. La factorisation $LZ(Y'/y_0)$ utilise le nombre minimal possible de phrases pour représenter Y'. Ainsi, chaque séquence y_i est de la forme $y_i = w_{(i,0)} w_{(i,1)} \cdots w_{(i,c_i)}$, où :

(a) La première phrase $w_{(i,-1)}$ correspond à ε ;

(b) La phrase $w_{(i,j)}$, $j \geq 0$, est le plus long préfixe de $y_i[(|w_{(i,-1)} \dots w_{(i,j-1)}|+1) \,..\, |y_i|-1]$ qui apparait dans y_0.

La RLZ génère des couples (p_i, ℓ_i), où p_i est la position de début de w_i dans y_0 et ℓ est sa longueur.

Soit y_F un tableau qui mémorise les f facteurs distincts apparaissant dans la factorisation de Y' selon l'ordre lexicographique. Ainsi, $y_F[j] = (s_j, e_j)$, où $y_0[s_j \,..\, e_j]$ est le j-ème facteur dans y_F.

La factorisation RLZ de Y est de complexité $O(n_F \log f) = O(n_F \log n)$ en espace mémoire, où n est la longueur de y_0 et n_F est le nombre minimal de facteurs nécessaires pour représenter Y' .

Exemple :

$$
\begin{array}{lcccc}
y_0 & = & \texttt{ACGTGACATAGT} & & \\
y_1 & = & \texttt{GATAGAC} & = & \texttt{GA, TAG, AC} \\
y_2 & = & \texttt{TGCA} & = & \texttt{TG, CA} \\
y_3 & = & \texttt{TGACGT} & = & \texttt{TGAC, GT}
\end{array}
$$

$y_F[id]$	Facteur	Pos. dans y_0
0	AC	$(0, 1)$
1	CA	$(6, 7)$
2	GA	$(4, 5)$
3	GT	$(2, 3)$
4	TAG	$(8, 10)$
5	TG	$(3, 4)$
6	TGAC	$(3, 6)$

FIGURE 2.8 – Factorisation RLZ de $Y' = \{y_1, y_2, y_3\}$ selon y_0 (en dessus). Les facteurs distincts (les plus longs) sont mémorisés suivant l'ordre lexicographique dans le tableau y_F (en dessous) comme couple de positions de début et de fin dans y_0.

Maintenant, donnons un définition de *RLZ-index(Y)*.

Définition 2.16 *[Do* et al.*, 2012] Soit un ensemble de séquences $Y = \{y_0, \dots, y_{r-1}\}$, RLZ-index(Y) est un auto-index basé sur la factorisation RLZ. Cet index est composé de structures de données basées sur $SA(y_0)$ et y_F.*

Construction

Soient Y un ensemble de séquences fortement similaires, les tableaux $SA(y_0)$ et y_F, en adoptant l'algorithme de [Do *et al.*, 2012], la construction de *RLZ-index* se fait en construisant les structures suivantes :

(a) **Structure $\mathcal{I}(y_F)$** :

Avant de décrire l'algorithme de construction, donnons d'abord des définitions utiles. Le facteur $y_F[j] = (s_j, e_j)$ *couvre* une position p si $s_j \leq p \leq e_j$. Le facteur $y_F[j]$ est

situé à *gauche* de $y_F[j']$ si $s_j \leq s_{j'}$ et $e_j < e_{j'}$. On définit le tableau G tel que $G[i] = j$ si $y_F[j]$ est le i-ème facteur gauche de y_F. Si $G[i] = j$ alors on définit $I_s[j] = s_{G[i]}$ et $I_e[j] = e_{G[i]}$. L'entrée $I_s[0]$ mémorise la position du facteur le plus à gauche. Les valeurs de I_s sont croissantes. Pour chaque $p \in \{0, \ldots, n-1\}$, on calcule le tableau D tel que $D[p] = \max_{j=1..f}\{I_e[j] - p + 1 \mid I_s[j] \leq p\}$, i.e., $D[p]$ mémorise la distance entre p et la position de fin maximale de tous les facteurs couvrant p. Soit D' tel que $D'[p] = D[SA(y_0)[p]]$. Ainsi, chaque entrée de D' mémorise la longueur du plus long rang dont la position de début est $SA(y_0)[p]$.

En adoptant l'algorithme de [Do *et al.*, 2012], la construction de $\mathcal{I}(y_F)$ se fait comme suit :

(i) D'abord, on construit le tableau G;

(ii) Ensuite, on construit une structure appelée *y-fast trie* [Willard, 1983] pour I_s ;

(iii) Puis, on construit la structure proposée par [Fischer and Heun, 2007], permettant de calculer la valeur maximale dans un rang donné dans un tableau d'entiers, pour I_e et D'.

Les tableaux D, D, I_s et I_e ne sont pas mémorisés explicitement. C'est la raison pour laquelle :

(α) Pour tout p, $0 \leq p \leq n-1$, on calcule $D[p]$ et $D'[p]$. Ce calcul se fait en, respectivement, $O(1)$ et $O(\log n)$ en temps de calcul.

(β) Pour tout i, $0 \leq i \leq f-1$, on utilise le tableau G pour calculer $I_s[i]$ et $I_e[i]$ en un temps constant.

L'algorithme de [Do *et al.*, 2012] est de complexité $O(2n + o(n)) + O(f \log n)$ en espace mémoire.

(b) **Structure $\mathcal{X}(y_F)$** :

Pour tout i, $0 \leq i \leq n-1$, on définit $\Gamma(i) = \{|y_F[j]| \mid s_j = i$ et $[s_j, e_j]$ le rang de $y_F[j]$ dans $SA(y_0)\}$, i.e., $\Gamma(i)$ mémorise les longueurs des facteurs dont les rangs dans $SA(y_0)$ commencent à i. On utilise par la suite $\Gamma(i)$ pour calculer un rang d'un facteur $y_F[j]$ dans y_F à partir de son rang dans $SA(y_0)$. Soient B et C deux vecteurs de bits tels que, respectivement, $B = 1$ si $\Gamma(i)$ est non-vide et $C[\sum_{i=1}^{r} |\Gamma(i)|] = 1$, où $\Gamma(i)$ est le r-ème ensemble non-vide.

En adoptant l'algorithme de [Do *et al.*, 2012], la construction de $\mathcal{X}(y_F)$ se fait comme suit :

(i) D'abord, on construit pour B la structure de données proposée par [Patrascu, 2008] permettant de répondre à l'opération *rank*;

(ii) Puis, on construit pour C la structure de données proposée par [Patrascu, 2008], permettant de répondre à l'opération *select*;

(iii) Enfin, on construit *y-fast trie* pour $\Gamma(i)$.

Cet algorithme est de complexité $O(f \log n) + o(n)$ en espace mémoire.

(c) **Structure $\mathcal{Y}(\boldsymbol{F}, y_{\boldsymbol{F}})$** :

Soit un tableau F mémorise les suffixes non-vides de Y' triés selon l'ordre lexicographique, i.e., chaque élément de F est de la forme $w_{(i,p)} w_{(i,p+1)} \cdots w_{(i,c_i)}$. Soit $\Im$ la

concaténation des facteurs des séquences dans Y'.

En adoptant l'algorithme de [Do *et al.*, 2012], la construction de $\mathcal{Y}(F, y_F)$ se fait comme suit :

(i) D'abord, on construit la BWT de y_0 et de $y_0^{\sim}$ le miroir de y_0 ;

(ii) Ensuite, on utilise la structure $\mathcal{X}(y_F)$;

(iii) Puis, on construit une structure de données de [Patrascu, 2008] afin de répondre à l'opération *select* dans un vecteur de bits X, où $X = 1$ si le premier facteur de $F[i]$ est différent de celui de $F[i+1]$;

(iv) Enfin, on construit la BWT de la séquence $\Im$.

Cet algorithme est de complexité $O(2.55n + 2nH_k(y_0) + n_F \log n)$ en espace mémoire.

(d) **Structure $\mathcal{M}$** :

Soit une matrice M telle que $M[i, j] = 1$ si ${y_F}^{\sim}[i]$ est le facteur qui précède $F[j]$, i.e., $F[j] = w_{(i,p)} \cdots w_{(i,c_i)}$, où $w^{\sim}_{(i,p-1)} = {y_F}^{\sim}[i]$ est le i-ème facteur dans ${y_F}^{\sim}$. La construction de $\mathcal{M}$ peut se faire en adoptant les algorithmes de [Chan *et al.*, 2011; Nekrich, 2009]. Ces algorithmes sont de complexités $O(n_F \log f \log\log f)$ en espace mémoire.

Recherche d'un motif

En adoptant l'algorithme de [Do *et al.*, 2012], la recherche d'un motif x dans un ensemble Y de séquences fortement similaires se fait selon les deux types suivants d'occurrences de x :

(a) Soit x apparait dans un seul facteur $w_{(i,p)}$. En utilisant la structure $\mathcal{I}(y_F)$ et le rang $[s_x, e_x]$ de x dans $SA(y_0)$, la recherche d'occurrences de x dans les facteurs mémorisés dans y_F se fait comme suit :

(i) Durant la première étape, on calcule récursivement chaque indice q tel que $s_x \leq q \leq e_x$ et $D'[q] \geq |x|$. De ce fait, on a $SA(y_0)[q], \ldots, SA(y_0)[q] + |x| - 1$ couvertes par au moins un facteur y_F et $y_0[SA(y_0)[q], \ldots, SA(y_0)[q] + |x| - 1]$ est une occurrence de x.

(ii) Enfin, durant la seconde étape, étant données une position p d'occurrence de x dans y_0 et q calculé à l'étape précédente, on repère l'ensemble des facteurs $\{y_F[G[i]] \mid I_s[i] \leq p \text{ et } q \leq I_{e[i]}\}$.

Cet algorithme est de complexité $O(m + occ_1 \log n)$ en temps de calcul, où m est la longueur de x et occ_1 est le nombre d'occurrences de x.

(b) Soit $x = X_{(i,p-1)} w_{(i,p)} \cdots w_{(i,q)} X'_{(i,q+1)}$, où $X_{(i,p-1)}$ est un suffixe de $w_{(i,p-1)}$ et $X'_{(i,q+1)}$ est un préfixe de $w_{(i,q+1)}$. Dans ce cas, on partitionne les occurrences de x en deux sous-ensembles :

(b_1) **Premier sous-ensemble** : Ce sous-ensemble est formé par les préfixes propres non-vides de x égaux à un suffixe $X_{(i,p-1)}$ de $w_{(i,p-1)}$. Pour ce sous-ensemble, en adoptant l'algorithme de [Do *et al.*, 2012] et en utilisant la structure $\mathcal{X}(y_F)$, on opère comme suit :

(i) D'abord, pour chaque préfixe x' de x, on détermine le rang $[s_{x'}, e_{x'}]$ dans $SA(y_0)$;

(ii) Puis, on considère $[s_x, e_x]$ le rang de x dans $SA(y_0)$. On détermine le rang maximal $[p, q]$ dans y_F de tous les facteurs $y_F[p], \ldots, y_F[q]$ préfixés par x en utilisant $\Gamma(i)$,

$0 \leq i \leq n-1$, où $p = 1 + \sum_{i=0}^{s_x-1} |\Gamma(i)| + |\{x \in \Gamma(s_x) \mid x < |x|\}|$ et $q = \sum_{i=0}^{e_x} |\Gamma(i)|$.

(b_2) **Deuxième sous-ensemble** : Ce sous-ensemble est formé par les suffixes de x égaux à des préfixes d'un suffixe de F. Pour ce sous-ensemble, en adoptant l'algorithme de [Do *et al.*, 2012] et en utilisant la structure $\mathcal{Y}(F, y_F)$, on opère en deux étapes principales :

(i) D'abord, on calcule les positions relatives dans Y' d'un facteur de F : Pour un indice p donné de x dans F, on retourne i et j tel que $F[p]$ commence à $w_{(i,j)}$ dans Y' ;

(ii) Puis, on convertit les positions relatives dans Y' en positions exactes dans Y' : Pour i et j, on retourne $1 + \sum_{q=1}^{j-1} |w_{(i,q)}|$. Ceci donne les positions de $w_{(i,j)}$ dans la séquence $y_i \in Y'$.

Soit Q un tableau tel que $Q[i]$ mémorise le rang dans F de $x[i\,..\,m-1]$, $0 \leq i \leq m-1$. On définit $A[i] = x[i\,..\,j]$, où j est l'indice maximal tel que $x[i\,..\,j]$ est un facteur de Y', si $x[i\,..\,j]$ existe et $A[i] = \varepsilon$ sinon. Soit $P[i]$ le rang $[p, q]$ dans F tel que $x[i\,..\,m-1]$ est un préfixe de toutes les entêtes des facteurs $F[p], \ldots, F[q]$, si $x[i\,..\,m-1]$ existe et $A[i] = \varepsilon$ sinon. En se basant sur $\mathcal{Y}(F, y_F)$, on calcule Q comme suit :

$$Q[i] = \begin{cases} P[i] & \text{si } P[i] \neq nil \\ \text{BackwardSearch}(A[i], Q[i + |A[i]|]) & \text{si } P[i] = nil \text{ et } A[i] \neq nil \\ \varepsilon & \text{sinon} \end{cases}$$

Enfin, on utilise la structure $\mathcal{M}$ pour repérer des combinaisons de $X_{(i,p-1)}$ et $w_{i_p} \cdots w_{(i,q)} X'_{(i,q+1)}$ adjacentes dans $y_i \in Y'$, $1 \leq i \leq r-1$. En effet, toutes les occurrences de x dans le deuxième sous-ensemble sont trouvées en listant les entrées de M égales à 1.

L'algorithme de [Do *et al.*, 2012] est de complexité $O(m(\log \sigma + \log\log n) + occ_2 \times (\log n \frac{log n_F}{\log n}))$ en temps de calcul, où m est la longueur de x et occ_2 est le nombre d'occurrences de x.

2.3.3 Algorithmes de Recherche de Motifs dans l'Index BIO-FMI

Dan cette partie, nous décrivons un index proposé par [Procházka and Holub, 2014], appelé *BIO-FMI*. Commençons par détailler le modèle de séquences utilisé par *BIO-FMI*.

Soit un ensemble $Y = \{y_0, \ldots, y_{r-1}\}$ de séquences fortement similaires, où chaque séquence y_i, $1 \leq i \leq r-1$, est de la forme $y_i = R_{i,0} C_{i,0} R_{i,1} C_{i,1} \cdots C_{i,k_{i-1}} R_{i,k_i}$, telle que $C_{i,j}$, $0 \leq j < k_i$, est un segment commun et $R_{i,j}$, $0 \leq j \leq k_i$, est un segment différencié.

Soit d une séquence mémorisant la concaténation de toutes les variations dans les y_i, $1 \leq i \leq r-1$, de la forme $C''_{i,j-1} R_{i,j} C'_{i,j}$, où $C''_{i,j-1}$ est un suffixe de $C_{i,j-1}$ et $C'_{i,j}$ est un préfixe de $C_{i,j}$ de longueur $\ell_c - 1$, oú ℓ_c est un paramètre donné. On appelle $C''_{i,j-1}$ *contexte gauche* de $R_{i,j}$ et $C'_{i,j}$ *contexte droit* de $R_{i,j}$. Le segment $R_{i,j}$ peut être une suppression ou une substitution. Pour chaque y_i, $1 \leq i \leq r-1$, on définit le quadruplet (Δ, p, ℓ, o), où Δ est le type de la variation mémorisée dans la position p dans y_0, de longueur ℓ et dont la différence o entre la position p et la position q de la variation dans y_i est $o = q - p$.

Définition 2.17 *[Procházka and Holub, 2014] Soit $Y = \{y_0, \ldots, y_{r-1}\}$ un ensemble de séquences fortement similaires, BIO-FMI est un index qui mémorise l'alignement, i.e., le quadruplet (Δ, p, ℓ, o), entre chaque séquence y_i, $1 \leq i \leq r-1$ et la séquence de référence y_0. BIO-FMI contient deux FM-index dont un est construit à partir de y_0 et l'autre est construit à partir de la séquence d. On note ces index, respectivement, I_0 et I_d.*

Construction

Soit $Y = \{y_0, \ldots, y_{r-1}\}$ un ensemble de séquences fortement similaires, en adoptant l'algorithme de [Procházka and Holub, 2014], la construction de I_0 et I_d se fait comme suit :

(a) **Construction de I_0** :

(i) D'abord, on construit un *arbre à Oondelettes* [Navarro and Mäkinen, 2007] sur y_0^{bwt} (rappelons que y_0^{bwt} est la Transformée de Burrows-Wheleer de y_0) ;

(ii) Puis, on construit un tableau Loc_0 dans lequel on mémorise les positions des caractères dans y_0 ;

(iii) Enfin, on construit un vecteur de bits $IsLoc_0$ sur l'*arbre à ondelettes* obtenu à la première étape. On utilise le vecteur $IsLoc_0$ et la fonction *rank* pour repérer les caractères de y_0 stockés à des positions échantillonnées (on rappelle qu'on mémorise des positions échantillonnées régulièrement sur y_0 avec un taux de $\log n$).

Cette construction se fait en utilisant un espace mémoire de complexité $O(2n+n\log\sigma+\log n+4n\log\log n)$, où n est la longueur de la séquence y_0.

(b) **Construction de I_d** :

(i) D'abord, on construit un *arbre à ondelettes* pour d ;

(ii) Puis, on construit $IsLoc_d$ permettant de reconnaitre les positions des variations dans y_i, $1 \leq i \leq r-1$, et le tableau Loc_d. Pour une variation donnée, Loc_d permet de repérer les séquences y_i qui la contiennent et ainsi de repérer le quadruplet (Δ, p, ℓ, o). Pour ce faire, Loc_d pointe sur un tableau *aIndex* dans lequel on mémorise, pour chaque y_i, $1 \leq i \leq r-1$, les positions de début des variations et leur ordre. Les variations dans *aIndex* sont stockées dans l'ordre croissant et sont aussi mémorisées dans des tableaux additionnels : Le tableau *aBasePos* dans lequel on stocke la position de début de la variation dans y_0, le tableau *aOffset* dans lequel on stocke la différence entre la position de la variation dans y_i et celle dans y_0, le tableau *aOp* dans lequel on stocke le type et la longueur de la variation. En outre, on utilise un tableau auxiliaire *SampleStart* pour accélérer l'extraction de la i-ème séquence. On mémorise dans *SampleStart* des pointeurs sur les positions de début des variations.

Cette construction se fait en utilisant un espace mémoire de complexité $O(N+N'(\log r+\log\frac{N'}{r})+N\log\sigma+\frac{N}{\log N}+4N\frac{\log\log N}{\log N}+r\log N'+N'\log n+N'\log n+N'+r\log N)$, où $N' = O(\Sigma_{i=1}^{i<r}(k_i+1))$ et $N = \Sigma_{i=1}^{i<r}\Sigma_{j=0}^{j<k_i}|R_{i,j}|$, sont respectivement, le nombre et la longueur totale de tous les segments différenciés.

Recherche d'un motif

Soient x un motif de longueur m et Y un ensemble de séquences fortement similaires, avant de commencer la recherche de x dans Y, on partitionne x en facteurs de longueur ℓ_c, i.e., $x = x_{1,\ell_c} \cdots x_{|\frac{m}{\ell_c}|\ell_c,m}$, puis on cherche séparément chacun de ces facteurs.

En adoptant l'algorithme de [Procházka and Holub, 2014], la recherche de x dans Y se fait comme suit :

(*i*) Durant la première étape, on utilise I_d pour chercher les facteurs de x. Pour chaque occurrence trouvée, notée occ_j, on calcule sa position $aBasePos_j + aOffset_j$ et on la mémorise dans une table de hachage ;

(*ii*) Durant la seconde étape, on utilise I_0 pour chercher les facteurs de x. On mémorise la position d'occurrence trouvée, en utilisant $aBasePos_j$, dans la table de hachage utilisée durant l'étape précédent ;

(*iii*) Durant la troisième étape, on évalue les occurrences comme suit : pour chaque occurrence occ_j on calcule $aOffset_{j,i}$ pour repérer la variation précédente dans y_i, $1 \leq i \leq r-1$, et ainsi pour valider occ_j. Puis, on stocke dans la table de hachage sa position en utilisant $aBasePos_j + aOffset_{j,i}$;

(*iv*) Durant la dernière étape, on repère toutes les positions d'occurrences de x.

Cet algorithme est de complexité $O(r \log \frac{N'}{r})$ en temps de calcul.

2.3.4 Algorithmes de Recherche de Motifs dans un Arbre de Suffixes d'alignement

Dans cette partie, nous présentons un index proposé par Na *et al.* [Na *et al.*, 2013a], appelé *Arbre de Suffixes d'Alignement*, en anglais *Suffix Tree of Alignment* (STA). Cet index est basé sur une nouvelle représentation de l'alignement des séquences défini comme suit :

Définition 2.18 *[Na* et al.*, 2013a] Soient deux séquences $y_0 = \alpha_1\beta_1 \cdots \beta_k\alpha_{k+1}$ et $y_1 = \alpha_1\delta_1 \cdots \delta_k\alpha_{k+1}$, $k \geq 1$, l'alignement de y_0 et y_1, noté $\alpha_1(\beta_1/\delta_1) \cdots \alpha_k(\beta_k/\delta_k)\alpha_{k+1}$, consiste à remplacer un facteur β_i par un facteur δ_i, $1 \leq i \leq k+1$.*

Cet alignement vérifie les conditions suivantes :

(a) Pour tout i, $i > 1$, le facteur α_i doit être différent du mot vide ;

(b) Soit le facteur β_i ou le facteur δ_i peut être le mot vide ;

(c) Les premiers caractères de $\beta_i\alpha_{i+1}$ et $\delta_i\alpha_{i+1}$ sont distincts.

Soient y_0 et y_1 deux séquences de la forme $y_0 = \alpha\beta\gamma$, $y_1 = \alpha\delta\gamma$ et l'alignement de ces séquences est $\alpha(\beta/\delta)\gamma$, où β/δ exprime le fait de remplacer le facteur β par le facteur δ. Soit α^i le plus long suffixe de α qui apparait au moins deux fois dans y_i, $0 \leq i \leq 1$. Soit α^*, tel que $\alpha^* = \alpha^0$ si $|\alpha^0| > |\alpha^1|7$ et $\alpha^* = \alpha^1$, sinon. On définit de nouveaux suffixes d'alignement, appelés *a-suffixes*. Il existe quatre type de a-suffixes :

(*a*) **Type 1 :** Un suffixe de γ ;

(*b*) **Type 2 :** Un suffixe de $\alpha^*\beta\gamma$ plus long que γ ;

(*c*) **Type 3 :** Un suffixe de $\alpha^*\delta\gamma$ plus long que γ ;

(d) **Type 4 :** Un suffixe de la forme $\alpha'(\beta/\delta)\gamma$, où α' est un suffixe de α plus long que α^*. Ce type de a-suffixes représente deux suffixes dérivés de y_0 et y_1.

Maintenant donnons une définition de *STA*.

Définition 2.19 *[Na* et al.*, 2013a] Soit Y un ensemble de séquences fortement similaires, STA(Y) est un trie compact représentant tous les a-suffixes de Y. Il satisfait les propriétés suivantes :*

(a) Les suffixes communs à y_0 et y_1, i.e., a-suffixes de type 1 sont représentés par les mêmes feuilles ;

(b) Les a-suffixes de type 4 sont liés par une feuille étiquetée par l'alignement ;

(c) Les a-suffixes de type 2 et 3 sont traités séparément comme suit : Soit un a-suffixe de la forme $\alpha''(\beta/\delta)\gamma$, où α'' est un suffixe de α^*. Selon la définition de α^*, α'' apparait deux fois dans y_0 ou dans y_1, donc α'' n'est pas toujours suivi par (β/δ). Ainsi, les a-suffixes $\alpha''\beta\gamma$ de type 2 et $\alpha''\delta\gamma$ de type 3 sont représentés par des feuilles distinctes.

Construction

Soit $Y = \{y_0, y_1\}$ un ensemble de séquences fortement similaires, en adoptant l'algorithme de [Na *et al.*, 2013a], la construction de *STA(Y)* se fait de la manière suivante :

(i) Durant la première étape, on construit $ST(y_0)$;

(ii) Durant la seconde étape, on transforme les suffixes de y_1 en a-suffixes selon les types 1, 3 et 4 et on les insère dans $ST(y_0)$ en adoptant la notion suivante : les a-suffixes de type 1 existent déjà dans l'arbre. L'insertion des a-suffixes plus longs que γ se fait en trois étapes. Durant la première et la deuxième étape, on insère les a-suffixes de type 3. Durant la troisième étape, on insère les a-suffixes de type 4. Pour ce faire, on doit :

(a) Repérer α^0 et insérer les suffixes de $\alpha^0\delta\gamma$ plus longs que γ. Ainsi, on effectue une recherche dans $ST(y_0)$ en vérifiant pour certains suffixes de α s'il existe deux feuilles pour les représenter. Pour $\alpha^{(i)}$ le suffixe de α de longueur i, on vérifie s'il est représenté par au moins deux feuilles en utilisant une technique de doublement décrite par [McCreight, 1976]. Soit h le plus petit i dans l'ensemble précédent tel que $\alpha^{(h)}$ est représenté par une seule feuille. Pour $\alpha^{(k)} = \alpha^0$ le plus long suffixe de α représenté par deux feuilles, $h/2 \leq |\alpha^{(k)}| < h$. On cherche $\alpha^{(k)}$ en parcourant $\alpha^{(h-1)}, \alpha^{(h-2)}, \ldots$ par utilisation de liens suffixes. Une fois α^0 est trouvé, on insère les suffixes du plus long $\alpha^0\delta\gamma$ au plus court $d\gamma$, où d est le dernier caractère de δ, sont insérés. On obtient ainsi l'arbre $ST'(Y)$;

(b Repérer α^* et insérer les suffixes de $\alpha^*\delta\gamma$ plus longs que $\alpha^0\delta\gamma$;

(c) Insérer les suffixes de $\alpha\delta\gamma$ plus longs que $\alpha^*\delta\gamma$. Ainsi, on insère les a-suffixes de type 4 de la forme $\alpha'\delta\gamma$, où $|\alpha'| > |\alpha^*|$. Pour ce faire, on remplace tout simplement chaque occurrence de $\beta\gamma$ par $(\beta/\delta)\gamma$.

L'algorithme de [Na *et al.*, 2013a] est de complexité $O(|\alpha| + |\beta| + |\delta| + |\gamma|)$ en espace mémoire.

Recherche d'un motif

La recherche d'un motif x dans un ensemble Y en utilisant $STA(Y)$ est similaire à celle utilisant $ST(Y)$.

2.3.5 Algorithme de Recherche de Motifs dans un Tableau de Suffixes d'Alignement

Dans cette partie, nous présentons un index proposé par Na *et al.* [Na *et al.*, 2013b], appelé *Tableau de Suffixes d'Alignement*, en anglais *Suffix Array of Alignment* (SAA). Cet index se base sur la même représentation de l'alignement décrite à la partie précédente.

Soient y_0, y_1 et y_2 trois séquences, telles que $y_0 = \alpha\beta\gamma$, $y_1 = \alpha\delta\gamma$ et et $y_2 = \alpha\vartheta\gamma$. L'alignement de ces trois séquences est de la forme $\alpha(\beta/\delta/\vartheta)\gamma$. Soient α^i le plus long suffixe de α qui apparait au moins deux fois dans y_i, $0 \leq i \leq 2$, et α^* tel que :

$$\alpha^* = \begin{cases} \alpha^0 & \text{si } |\alpha^0| > \max(|\alpha^1|, |\alpha^2|) \\ \alpha^1 & \text{si } |\alpha^1| > \max(|\alpha^0|, |\alpha^2|) \\ \alpha^2 & \text{sinon.} \end{cases}$$

On appelle a-suffixes :
(a) Les suffixes de γ ;
(b) Les suffixes $\omega^0\gamma$, où ω^0 est un suffixe non-vide de $\alpha^*\beta$;
(c) Les suffixes $\omega^1\gamma$, où ω^1 est un suffixe non-vide de $\alpha^*\gamma$;
(d) Les suffixes $\omega^2\gamma$, où ω^2 est un suffixe non-vide de $\alpha^*\vartheta$;
(e) Les suffixes de la forme $\alpha'(\beta/\delta/\vartheta)\gamma$, où α' est un suffixe de α plus long que α^*. Ces suffixes sont dérivés de y_0, y_1 et y_2.

Définition 2.20 *[Na* et al.*, 2013b] Soit $Y = \{y_0, y_1, y_2\}$ un ensemble de trois séquences fortement similaires, dont l'alignement est de la forme $\alpha(\beta/\delta/\vartheta)\gamma$. SAA ($\alpha(\beta/\delta/\vartheta)\gamma$) est une liste triée selon l'ordre lexicographique de tous les a-suffixes de Y.*

L'ordre de tri est bien défini pour les a-suffixes de types 1, 2, 3, 4 puisqu'ils représentent une seule séquence. Pour les a-suffixes de type 5 de la forme $\alpha'(\beta/\delta/\vartheta)\gamma$, on sait que $|\alpha'| > |\alpha^*|$ et que α' est un préfixe qui apparait une seule fois dans $\alpha'\beta\gamma$, $\alpha'\delta\gamma$ et $\alpha'\vartheta\gamma$, i.e., α' est un facteur de y_0, y_1 et y_2. De ce fait, l'ordre de $\alpha'(\beta/\delta/\vartheta)\gamma$ est donné par α'. Ainsi, le plus long préfixe commun entre les a-suffixes de $\alpha(\beta/\delta/\vartheta)\gamma$ est bien défini.

Construction

Soit $Y = \{y_0, y_1, y_2\}$ un ensemble de trois séquences fortement similaires. Soient γ^0, γ^1 et γ^2 les plus long préfixes de γ, apparaissant au moins deux fois dans, respectivement, y_0, y_1 et y_2. Soit γ^* tel que :

$$\gamma^* = \begin{cases} \gamma^0 & \text{si } |\gamma^0| > \max(|\gamma^1|, |\gamma^2|) \\ \gamma^1 & \text{si } |\gamma^1| > \max(|\gamma^0|, |\gamma^2|) \\ \gamma^2 & \text{sinon.} \end{cases}$$

En adoptant l'algorithme de [Na *et al.*, 2013b], la construction de $SSA(\alpha(\beta/\delta/\vartheta)\gamma)$ se fait comme suit :

(i) Durant la première étape, on repère $|\alpha^*|$ et $|\gamma^*|$. Pour ce faire, on procède en deux étapes :

(a) On cherche α^0 en utilisant $SA(y_0^{\sim})$ ($y_0^{\sim}$ est le miroir de y_0). Une fois trouvée, on cherche le plus long suffixe $\alpha^{(k)}$, $k = 1, 2$, de longueur supérieure à celle de α^0. Puisque $\alpha^{(k)}$ est plus long que α^0 alors il est facile de démontrer qu'il apparait dans un suffixe préfixé par α^0 dans y_k. De ce fait, on utilise les tableaux de suffixes de $\alpha^0\delta\gamma^0$ et $\alpha^0\vartheta\gamma^0$ pour déduire $|\alpha^*|$;

(b) On cherche γ^* de façon symétrique à la recherche de α^0 . Ainsi, on utilise $SA(y_0)$, $SA(\alpha^0\delta\gamma^0)$ et $SA(\alpha^0\vartheta\gamma^0)$. On obtient γ^0, γ^1 et γ^2 et on retient celui de longueur maximale ;

(ii) Durant la seconde étape, on construit le tableau de suffixes généralisé GSA de y_0, $\alpha^*\delta\gamma^*d$ $\alpha^*\vartheta\gamma^*d$, où d est le caractère qui suit γ^* dans γ. L'ordre des a-suffixes est déterminé par $\omega\gamma^*d$;

(iii)Enfin, on supprime les suffixes de γ^*d dérivés de $\alpha^*\delta\gamma^*d$ et $\alpha^*\vartheta\gamma^*d$.

L'algorithme de construction de SAA de [Na *et al.*, 2013b] est de complexité $O(|y_0| + |\alpha^*\delta\gamma^*| + |\alpha^*\vartheta\gamma^*|)$ en espace mémoire.

Recherche d'un motif

L'algorithme de recherche d'un motif x dans un ensemble Y de séquences fortement similaires en utilisant $SAA(Y)$ [Na *et al.*, 2013b] est similaire à celui utilisant $SA(Y)$.

2.4 Algorithme de Recherche Incrémentale de Motifs dans une Structure Arborescente

Dans cette partie, nous présentons un index proposé par René *et al.* [Rahn *et al.*, 2014], appelé *Journaled String Tree* (JST).

Avant de décrire formellement JST, donnons des définitions utiles. Commençons par définir la notion de *séquences journalisées.*

Définition 2.21 *[Rahn* et al.*, 2014] Soit $Y = \{y_0, \dots y_{r-1}\}$ un ensemble de séquences fortement similaires, chaque séquence y_i, $1 \leq i \leq r-1$, est représentée par une nouvelle version, appelée séquence journalisée et notée λ_i. Celle-ci est une version compressée de y_i contenant :*

(a) Un pointeur sur y_0 ;

(b) Un tampon ib dans lequel on mémorise les variations de y_i de type insertion ou substitution ;

(c)Un arbre binaire de recherche, appelé arbre journal et noté $J(\lambda_i)$. Cet arbre est construit à partir des facteurs en provenance de y_0 ou de ib. Il mémorise chaque facteur sous forme d'une entrée $e = (pv, pp, \ell, \delta)$, où :

(c_1) $\delta = 0$ si le facteur provient de y_0, sinon $\delta = 1$ si le facteur provient de ib ;

(c_2) pv est la position de début du facteur dans y_i, appelée position virtuelle ;

(c_3) pp est la position de début du facteur dans y_i, appelée position physique ;
(c_4) ℓ est la longueur de e.

Pour deux entrées e et e′ on a $e < e'$ si et seulement si $pv < pv'$.

Définissons maintenant la notion de *nœud branchant.*

Définition 2.22 *[Rahn* et al.*, 2014] Soit Y un ensemble de séquences fortement similaires, on appelle nœud branchant, noté u, une position dans y_0 dans laquelle au moins une séquence y_i contient une variation.*

Un nœud u à la position j dans y_0, dont s est le facteur associé, appartenant à l'ensemble de séquences $C \subseteq \{1, \ldots, r-1\}$ et dont la variation est de type Δ, i.e., suppression, insertion ou substitution, est représenté par le quadruplet (j, s, C, Δ).

On note $j = pos(u)$, $s = label(u)$, $C \subseteq \{1, \ldots, r-1\} = cov(u)$.

Maintenant, donnons une définition de JST :

Définition 2.23 *[Rahn* et al.*, 2014] Soit $Y = \{y_0, \ldots, y_{r-1}\}$ un ensemble de séquences fortement similaires, JST(Y) est une structure de données arborescente qui utilise y_0 comme coordonnée d'ancrage avec des nœuds branchants.*

JST(Y) est composée de :

(a) La séquence de référence y_0 ;

(b) Un tableau de nœuds branchants, noté $V(T)$, trié selon l'ordre croissant des positions des nœuds dans y_0 ;

(c) Les séquences journalisées $\lambda_1, \ldots, \lambda_{r-1}$.

Construction

Soit $Y = \{y_0, \ldots, y_{r-1}\}$ un ensemble de séquences fortement similaires, en adoptant l'algorithme de [Rahn *et al.*, 2014], on génère les séquences λ_i, $1 \leq i \leq r-1$, comme suit :

(i) D'abord, on insère dans $J(\lambda_i)$ les entrées associées aux facteurs en provenance de y_0 ;

(ii) Ensuite, on insère dans $J(\lambda_i)$ les entrées associées aux facteurs en provenance de y_i dont les variations sont de type insertion ou substitution. On ajoute ces facteurs dans ib ;

L'algorithme de génération d'une séquence journalisée λ_i [Rahn *et al.*, 2014], $1 \leq i \leq r-1$, est de complexité $O(nbvar)$ en temps de calcul, où $nbvar$ est le nombre des variations dans y_i.

Pour accéder directement à une position j dans une séquence λ_i en utilisant $J(\lambda_i)$, en adoptant l'algorithme de [Rahn *et al.*, 2014], on opère de la manière suivante :

(i) D'abord, on cherche l'entrée (pv, pp, ℓ, δ), où $pv \leq j \leq pv + \ell$;

(ii) Puis, on récupère directement le caractère mémorisé à la position $pp + (j - pv)$.

L'accès direct à une position j, $0 \leq j \leq n-1$, dans une séquence λ_i est de complexité $O(\log |J(\lambda_i)|)$ en temps de calcul.

Recherche d'un motif

Soient un motif x et un ensemble Y, en adoptant l'algorithme de [Rahn *et al.*, 2014], la recherche de x dans JST(Y) se fait en utilisant un algorithme de recherche incrémentale exacte ou approchée. On opère comme suit :

(i) D'abord, on positionne la fenêtre glissante sur y_0 et on compare les caractères de x avec ceux en face dans y_0. On utilise un vecteur de bits pour détecter les variations dans la fenêtre en cours et on fournit à l'algorithme les séquences λ_i, $0 \leq i \leq r-1$, qui la contiennent.

(ii) Ensuite, si lors du parcours de y_0 on rencontre un nœud branchant u alors on positionne la fenêtre sur le sous-arbre enraciné par u. Ainsi, on termine la recherche de x dans l'une des séquences λ_i, où $i \in cov(u)$. Ce branchement modifie l'état de l'algorithme, on doit alors mémoriser son état pour faciliter à l'algorithme de restaurer son état précédent. Pour ce faire, on utilise une pile permettant de mémoriser l'état courant du parcours.

2.5 Conclusion

Dans ce chapitre, nous avons dressé un état de l'art sur les algorithmes de recherche de motifs dans un ensemble de séquences fortement similaires. La majorité des algorithmes que nous avons présentés sont basés sur un index construit sur les facteurs de séquences d'entrée. Nous avons présentés ces algorithmes selon l'ordre décroissant de consommation d'espace mémoire.

La conclusion qu'on peut tirer est que les algorithmes de recherche dans un index classique n'exploitent pas la propriété de la haute similarité entre les séquences et leur représentation des séquences est redondante. Récemment, des algorithmes de recherche dans un index directement dédiés aux séquences fortement similaires sont apparus. Ces algorithmes se basent sur une représentation compressée de séquences selon une séquence de référence choisie arbitrairement. Ils permettent ainsi d'aboutir à des index compacts et permettent la recherche rapide et efficace d'un motif donné dans l'ensemble de séquences fortement similaires. En d'autres mots, les algorithmes les plus récents que nous avons présentés sont de complexités spatiales dépendant de la longueur d'une seule séquence en plus de nombre de variations.

Dans la table 2.1, nous donnons quelques complexités des algorithmes de recherche dans les index que nous avons décrits dans les parties précédentes. Nous avons choisi de présenter les meilleurs résultats atteints dans chaque famille d'algorithmes.

En conclusion, quoiqu'il en soit chaque algorithme peut défendre son intérêt. Ceci dépend essentiellement des fins d'utilisation : gain en espace, accès et recherche rapide, etc.

Structure	Epace	Temps de recherche
$ST(T)$	$O(n)$	$O(m)$
$SA(T)$	$O(n)$	$O(m + \log n)$
FM-index	$nH_k + o(n \log \sigma)$	$O(m)$
CSA	$nH_k(T)$	$O(\log \log n + \log \sigma)$
LZ-index	$4nH_k(T) + o(n \log \sigma)$	$O((occ + 1) \log n)$
Huang et al.	$O(n' + N' \log rk + rk(\log n' + \log N'))$	$O(m + N_{psc}(m \log(rk)) + occ_3 \log n)$
RLZ	$2nH_k(T_0) + 5.55n + O(N_F \log n \log \log n)$	$O(m(\log \sigma + \log \log n) + occ \cdot (\log n \frac{\log N_F}{\log n}))$
SAA	$O(n + \lvert\alpha^* \delta \gamma^*\rvert + \lvert\alpha^* \vartheta \gamma^*\rvert)$	$O(m + occ)$

TABLE 2.1 – Table illustrant certaines complexités (consommations mémoires et temps de recherches) : n est la longueur des séquences d'entrée, n' et N' sont respectivement le nombre des segments communs et différenciés et N_{psc} est le nombre des préfixes d'un motif donné qui sont des suffixes de certaines parties communes (Huang et al.), N_F est le nombre des facteurs de la séquence référence utilisés pour compresser les autres séquences de l'ensemble d'entrée (RLZ). Le nombre d'occurrences du motif est noté par *occ*.

Chapitre 3

MPE : Algorithme de Morris-Pratt Étendu

Introduction

Dans ce chapitre, nous proposons un nouvel algorithme, appelé MPE [Nsira *et al.*, 2014b], permettant de faire de la recherche incrémentale exacte d'un motif dans un ensemble de séquences fortement similaires.

En fait, l'algorithme MPE est une extension de l'algorithme de *Morris-Pratt* (MP) [Morris and Pratt, 1970] qui permet de faire de la recherche incrémentale exacte d'un motif dans une seule séquence.

Le reste du chapitre est organisé comme suit :

Dans la première section 3.1, nous décrivons l'algorithme de Morris-Pratt classique. Dans la deuxième section 3.2, nous donnons une présentation détaillée de l'algorithme MPE. Dans la troisième section 3.3, nous calculons les complexités théoriques de l'algorithme MPE. Dans la quatrième section 3.4, nous présentons un exemple illustratif. Enfin, dans la dernière section 3.5, nous terminons par une conclusion à ce chapitre.

3.1 MP : Algorithme de Morris-Pratt

Commençons d'abord par présenter les notions utilisées par l'algorithme MP.

3.1.1 Préliminaires

Définition 3.1 *[Crochemore* et al.*, 2007] Soit x un motif de longueur $m \geq 0$, un bord de x est un facteur qui est à la fois un préfixe et un suffixe de x.*

Définition 3.2 *[Crochemore* et al.*, 2007] Soit x un motif de longueur $m \geq 0$, le tableau mpNext associé à x est défini comme suit :*

Pour tout i, $0 \leq i \leq m$, mpNext$[i]$ mémorise la longueur du plus long bord de $x[0\,..\,i-1]$.

3.1.2 Description

L'algorithme MP utilise la notion de la fenêtre glissante. L'algorithme MP se décompose en deux phases principales :

(a) Une phase de prétraitement : au cours de laquelle on construit un tableau *mpNext* mémorisant les décalages à effectuer, à la fin de chaque itération, sur une fenêtre balayant une séquence y d'entrée.

(b) Une phase de recherche : au cours de laquelle on localise toutes les occurrences d'un motif x dans la séquence y.

Phase de prétraitement

La phase de prétraitement de l'algorithme MP est assurée par la procédure *Calcul_mpNext*. Cette procédure se décompose en deux étapes :

(i) D'abord, on initialise *mpNext*[0] par -1, la longueur du plus long bord du mot vide ε ;

(ii) Ensuite, pour tout i, $1 \leq i \leq m$, on mémorise la longueur de plus long bord de $x[0\,..\,i-1]$ dans *mpNext*[i].

Phase de recherche

La phase de recherche de l'algorithme MP se décompose en deux étapes :

(i) D'abord, à une tentative j, $0 \leq j \leq n-1$, on compare les caractères de x avec ceux de y contenus dans la fenêtre positionnée à j sur y. Pour i, $0 \leq i \leq m-1$, si $x[0\,..\,i] = y[j\,..\,i+j]$ est le préfixe reconnu et $x[i+1] \neq y[i+j+1]$ alors on utilise *mpNext*[i] pour décaler la fenêtre à droite ;

(ii) Puis, à la tentative suivante, on poursuit les comparaisons à partir des caractères $x[mpNext[i]]$ et $y[i+j+1]$.

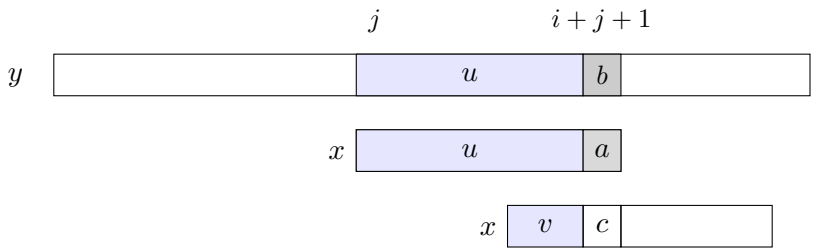


FIGURE 3.1 – Décalage de la fenêtre glissante avec l'algorithme MP.

3.2 MPE : Algorithme de Morris-Pratt Étendu

D'abord, nous commençons par présenter des notions utilisées par l'algorithme MPE.

3.2.1 Préliminaires

Définition 3.3 *[Crochemore* et al.*, 2007] Soient u et v deux mots de même longueur, la distance de Hamming entre u et v, notée $Ham(u, v)$, est le nombre de positions en lesquelles les deux mots possèdent des caractères différents, i.e., $Ham(u, v)$ est le nombre de substitutions nécessaires pour transformer u en v. Formellement :*

$$Ham(u, v) = card\{k \mid 0 \leq k \leq |u| - 1 \text{ et } u[k] \neq v[k]\} \tag{3.1}$$

Définition 3.4 *[Crochemore* et al.*, 2007] Soit x un motif de longueur m, le tableau $pref_x^0$ associé à x est défini comme suit :*

Pour tout i, $0 \leq i \leq m-1$, $pref_x^0[i]$ mémorise la longueur de plus long préfixe commun entre x et $x[i\,..\,m-1]$, i.e., $pref_x^0[i] = |lcp(x, x[i\,..\,m-1])|$.

Définition 3.5 *[Nsira* et al.*, 2014b] Soit x un motif de longueur m, le tableau $pref\,\widetilde{}_x$ associé à x est défini comme suit :*

Pour tout i, $0 \leq i \leq m-1$, $pref\,\widetilde{}_x[i]$ mémorise la longueur du plus long préfixe commun entre le miroir du préfixe $x[0\,..\,i]$ et celui du préfixe $x[0\,..\,i']$, où $i' < i$, si $|lcp(x[0\,..\,i]\widetilde{}, x[0\,..\,i']\widetilde{})| \neq 0$. Formellement :

$$pref\,\widetilde{}_x[i] = \{(i', \ell) \mid i' < i, x[i] = x[i'] \text{ et } 0 < \ell = |lcp(x[0\,..\,i]\widetilde{},\ x[0\,..\,i']\widetilde{})|\}. \tag{3.2}$$

Définition 3.6 *[Nsira* et al.*, 2014b] Soit x un motif de longueur m, le tableau B associé à x est défini comme suit :*

Pour tout i, $0 \leq i \leq m-1$, $B[i]$ mémorise les positions de début des bords de $x[0\,..\,i]$ avec une erreur.

Formellement :

$$B[i] = \{i' \mid Ham(x[0\,..\,i-i'], x[i'\,..\,i]) = 1\} \tag{3.3}$$

3.2.2 Description

À l'instar de l'algorithme MP, cet algorithme utilise une fenêtre glissante pour parcourir, simultanément, les séquences d'entrée. L'algorithme MPE se décompose en deux phases principales :

(a) Une phase de prétraitement : au cours de laquelle on construit un tableau B mémorisant de nouveaux décalages à effectuer, à la fin de chaque itération, sur la fenêtre positionnée couramment sur les séquences d'un ensemble $Y = \{y_0, y_1, \ldots, y_{r-1}\}$ d'entrée.

(b) Une phase de recherche : au cours de laquelle on localise toutes les occurrences d'un motif x dans l'ensemble Y.

L'algorithme de recherche MPE considère la totalité de la séquence de référence y_0 et une liste Z représentant les séquences y_g, $1 \leq g \leq r-1$ par les variations qu'elles contiennent par rapport à y_0.

Pour tout préfixe $x[0\,..\,i]$, $0 \leq i \leq m-1$, reconnu en parcourant les r séquences à une position j, $0 \leq j \leq n-1$, l'algorithme de recherche MPE considère trois cas :

(a) **Cas 1** : $x[0\,..\,i] = y_0[j\,..\,j+i]$ et $\nexists(\mathcal{G}, k, c) \in Z$ tel que $j \leq k \leq j+i$. De ce fait, une correspondance de $x[0\,..\,i]$ est trouvée dans y_0 et la fenêtre en cours ne contient aucune variation dans les autres séquences. Ainsi, $x[0\,..\,i]$ est reconnu également dans toutes les séquences.

(b) **Cas 2** : $x[0\,..\,i] = y_0[j\,..\,j+i]$ et $\exists(\mathcal{G}, k, c) \in Z$ tel que $j \leq k \leq j+i$. De ce fait, une correspondance de $x[0\,..\,i-1]$ est trouvée dans toutes les séquences sauf dans y_g.

(c) **Cas 3** : $x[0\,..\,i] = y_g[j\,..\,j+i]$ et $\exists(\mathcal{G}, k, c) \in Z$ tel que $j \leq k \leq j+i$ et $g \in \mathcal{G}$. Alors une correspondance de $x[0\,..\,i-1]$ est trouvée uniquement dans y_g.

Phase de Prétraitement

Au cours de cette phase, on construit tous les tableaux utiles pour évaluer des décalages à la fin de chaque tentative de l'algorithme *MPE_Search*. On utilise le tableau *mpNext* de MP pour les bords ayant une distance de Hamming 0 et on construit un nouveau tableau, appelé B, pour les bords ayant une distance de Hamming 1.

Dans les sous-paragraphes suivants, nous développons les algorithmes permettant de construire les tableaux $pref_x^0$, $pref\,\widetilde{}_x$, B.

Construction de $pref_x^0$

En fait, la construction de $pref_x^0$ se fait en utilisant un algorithme linéaire, appelé *Préfixes* [Crochemore *et al.*, 2007].

Un pseudo code de construction de $pref_x^0$ est illustré dans la section 4.2.2.

Construction de $pref\,\widetilde{}_x$

Notre algorithme de construction de $pref\,\widetilde{}_x$ est un algorithme incrémental. Il localise les occurrences de $x[i]$, $1 \leq i \leq m-1$, dans des positions gauches à i sur x afin d'alimenter $pref\,\widetilde{}_x[i]$ par des couples (i', ℓ), où $i' < i$, $x[i'] = x[i]$ et $\ell = |lcp(x[0\,..\,i]\widetilde{}, x[0\,..\,i']\widetilde{})|$.

Selon la définition de $pref\,\widetilde{}_x$, on peut obtenir $pref\,\widetilde{}_x[i+1]$ à partir de $pref\,\widetilde{}_x[i]$ en se basant sur la relation suivante :

$$pref\widetilde{}_x[i{+}1] = \{(i',\, \ell{+}1) \mid x[i'] = x[i{+}1] \text{ and } \exists(i'{-}1) \in pref\widetilde{}_x[i]\} \cup \{(i', 1) \mid \nexists(i'{-}1) \in pref\widetilde{}_x[i]\}.$$

À partir de la définition de $pref\,\widetilde{}_x$, on remarque que l'algorithme de construction de $pref\,\widetilde{}_x$ se base sur les occurrences des caractères. De ce fait, on a besoin de tableaux intermédiaires comme suit :

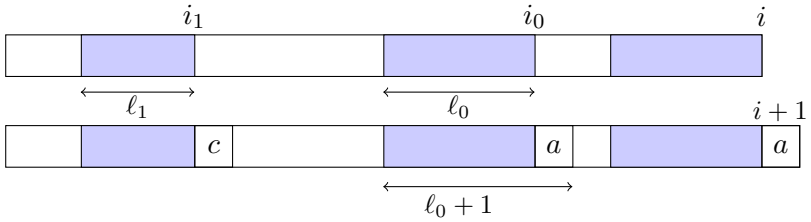


FIGURE 3.2 – Relation de calcul de $pref_x^{\sim}[i+1]$ à partir de $pref_x^{\sim}[i] : (i_0, \ell_0), (i_1, \ell_1) \in pref_x^{\sim}[i] \mid x[i_0+1] = x[i+1]$ et $x[i_1+1] \neq x[i+1]$, alors $(i_0+1, \ell_0+1) \in pref_x^{\sim}[i+1]$.

(*a*) *last* : *last*[*c*], $\forall c \in \Sigma$, mémorise la position de l'occurrence la plus à droite de c dans x. Formellement :

$$last[c] = \max\{i \mid x[i] = c \text{ et } 0 \leq i < m\} \tag{3.4}$$

(*b*) *prev* : *prev*[*k*], $1 \leq k \leq m-1$, mémorise la position de l'occurrence précédente de $x[k]$ sur la gauche. Formellement :

$$prev[k] = \{-1\} \cup \{i \mid x[i] = x[k] \text{ et } 0 \leq i < k\}. \tag{3.5}$$

Exemple :

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$x[k]$	A	T	A	A	T	A	G	A	C	A	A	T	A	C
$prev[k]$	−1	−1	0	2	1	3	−1	5	−1	7	9	4	10	8

c	A	C	G	T
$last[c]$	12	13	6	11

FIGURE 3.3 – Les tableaux *last* et *prev* pour $x =$ ATAATAGACAATAC et $\Sigma = \{$A, C, G, T$\}$.

La construction des tableaux *last* et *prev* se fait à l'aide d'un algorithme appelé *ComputePrev*. Cet algorithme peut être représenté par le pseudo code **Algorithme 3.1**, où on reçoit en entrée un motif x de longueur m et on renvoie en sortie le tableau *prev*.

La construction de $pref_x^{\sim}$ est assurée par un algorithme appelé *ComputeB*. Cet algorithme est présenté par le pseudo code donné **Algorithme 3.2** détaillé dans le sous-paragraphe qui suit. Les éléments de $pref_x^{\sim}$ sont donnés par les lignes 6-14 et 23.

Construction de *B*

Notre algorithme de construction du tableau B, appelé *ComputeB* se base sur l'utilisation des tableaux $pref_x^0$ et $pref_x^{\sim}$ pour remplir chaque ensemble $B[i]$, $0 \leq i \leq m$, par les positions de début des suffixes des bords de $x[0\,..\,i]$ avec une substitution.

Ceci est montré par la proposition suivante :

Algorithme 3.1 : *ComputePrev*

```
1: Données : x, m
2: Résultats : prev
3: prev[0] = -1
4: last[c] = -1, ∀c ∈ Σ
5: last[x[0]] = 0
6:     pour i de 1 à m − 1 faire
7:         prev[i] = last[x[i]]
8:         last[x[i]] = i
9:     fin pour
```

Proposition 1 *Pour* $1 \leq i \leq m-1$, *s'il* $\exists (i-i', \ell) \in pref_x^{\sim}[i]$ *et* $i - i' = pref_x^0[i'] + \ell$ *et* $i - i' < i' + pref_x^0[i']$ *alors* $x[0\,..\,i-i']$ *est un bord de* $x[0\,..\,i' + pref_x^0[i'] - 1]x[pref_x^0[i']]x[i' + pref_x^0[i'] + 1\,..\,i]$.

Puisque $i - i' < i' + pref_x^0[i']$ alors $x[0\,..\,i-i']$ est un préfixe de $x[0\,..\,i' + pref_x^0[i'] - 1]$ et ainsi est un préfixe de $x[0\,..\,i' + pref_x^0[i'] - 1]x[pref_x^0[i']]x[i' + pref_x^0[i'] + 1\,..\,i]$. Par définition de $pref_x^0[i']$, on a $x[i'\,..\,i' + pref_x^0[i'] - 1] = x[0\,..\,pref_x^0[i'] - 1]$ et par définition de $pref_x^{\sim}[i]$ et par le fait que $\ell = i - i' - pref_x^0[i']$ on a $x[i' + pref_x^0[i'] + 1\,..\,i] = x[pref_x^0[i'] + 1\,..\,i - i']$. Ainsi $x[0\,..\,i-i']$ est un suffixe de $x[0\,..\,i' + pref_x^0[i'] - 1]x[pref_x^0[i']]x[i' + pref_x^0[i'] + 1\,..\,i]$. Il en résulte $x[0\,..\,i-i']$ est un bord de $x[0\,..\,i' + pref_x^0[i'] - 1]x[pref_x^0[i']]x[i' + pref_x^0[i'] + 1\,..\,i]$.

La figure 3.4 illustre la situation donnée par la Proposition 1. La figure 3.5 donne un exemple.

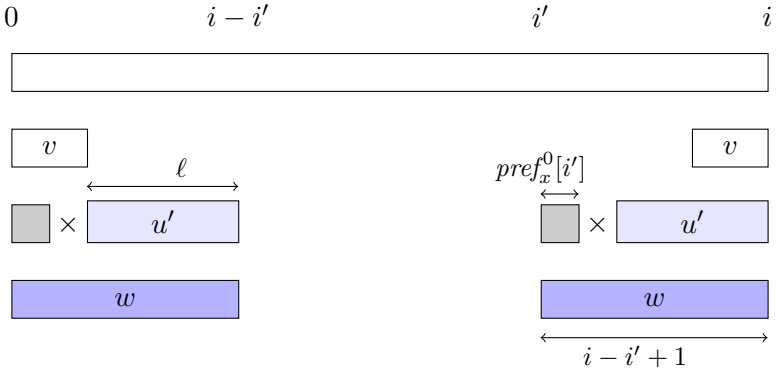


FIGURE 3.4 – Situation de la Proposition 1 : v est le plus long bord à distance de Hamming 0 de $x[0\,..\,i]$ et w est le plus long bord à distance de Hamming 1.

Par conséquence, on a $B[i] = \{i' \mid \exists (i-i', \ell) \in pref_x^{\sim}[i]$ et $i - i' = pref_x^0[i'] + \ell\} \cup \{i \mid pref_x^0[i] = 0\}$. Un exemple est décrit par la figure 3.6.

La construction de B est réalisée par l'algorithme *ComputeB*. En adoptant cet algorithme, on opère comme suit :

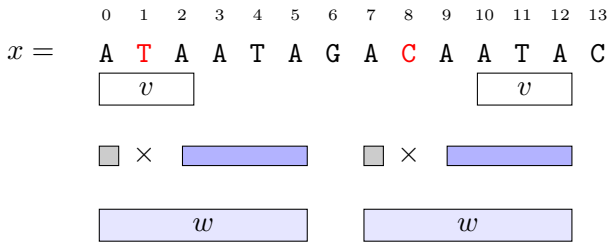


FIGURE 3.5 – Exemple illustrant le résultat de la Proposition 1 : `ATA` est le plus long bord à distance de Hamming 0 de `ATAATAGACAATA` et `ATAATA` est le plus long bord à distance de Hamming 1.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$x[i]$	A	T	A	A	T	A	G	A	C	A	A	T	A	C
$pref_x^0[i]$	13	0	1	3	0	1	0	1	0	1	3	0	1	0
$pref_{\sim x}[i]$	$\emptyset$	$\emptyset$	L_2	L_3	L_4	L_5	$\emptyset$	L_7	$\emptyset$	L_9	L_{10}	L_{11}	L_{12}	L_{13}
$B[i]$	$\emptyset$	$\{1\}$	$\emptyset$	$\emptyset$	$\{4\}$	$\emptyset$	$\{6\}$	$\{5\}$	$\{8\}$	$\{7\}$	$\{7\}$	$\{7, 11\}$	$\{7\}$	$\{13\}$

$L_2 = \{(0,1)\}$, $L_3 = \{(2,1)(0,1)\}$, $L_4 = \{(1,2)\}$, $L_5 = \{(3,1),(2,3),(0,1)\}$,
$L_7 = \{(5,1)(3,1),(2,1),(0,1)\}$, $L_9 = \{(7,1),(5,1)(3,1),(2,1),(0,1)\}$,
$L_{10} = \{(9,1),(7,1),(5,1)(3,2),(2,1),(0,1)\}$, $L_{11} = \{(4,3),(1,2)\}$,
$L_{12} = \{(10,1),(9,1),(7,1),(5,4)(3,1),(2,3),(0,1)\}$, $L_{13} = \{(8,2)\}$.

FIGURE 3.6

(i) Durant la première étape, on calcule incrémentallement les éléments du tableau $pref_{\sim x}$. En effet, pour tout i, $1 \leq i \leq m-1$, on obtient $pref_{\sim x}[i]$ en fonction de $pref_{\sim x}[i-1]$. À chaque itération i, on ne sauvegarde que $pref_{\sim x}[i-1]$ et $pref_{\sim x}[i]$: Puisque $pref_{\sim x}[i]$ est obtenue en fonction de $pref_{\sim x}[i-1]$ et $B[i]$ dépend uniquement de $pref_{\sim x}[i]$ et non de $pref_{\sim x}[j]$, $j < i$, on omet les autres éléments de $pref_{\sim x}$. De ce fait, on utilise deux variables L_1 et L_2 pour mémoriser respectivement $pref_{\sim x}[i-1]$ et $pref_{\sim x}[i]$. L_1 et L_2 sont des listes de couples : $p.pos$ donne le premier élément d'un couple p, $p.length$ donne le deuxième composant, $first(L)$ retourne le premier élément de la liste L, $next(L)$ supprime le premier élément d'une liste L et $L.append(p)$ ajoute le couple p à la fin de la liste L ;

(ii) Durant la seconde étape, on construit le tableau B. On vérifie d'abord si la condition de la Proposition 1 est satisfaite pour la position i. S'il existe une position $i-j$ vérifiant cette condition alors on ajoute $i-j$ à $B[i]$. Si $pref_x^0[i] = 0$ alors on ajoute i à $B[i]$ car il est trivial que lorsqu'on substitut $x[i]$ à $x[0]$ on obtient un nouveau bord de $x[0\,..\,i]$ à distance de Hamming 1 de longueur $1 > pref_x^0[i] = 0$.

L'algorithme *ComputeB* reçoit en entrée un motif x de longueur m, les tableaux $prev$ et $pref_x^0$. Il construit dans un premier temps $pref_{\sim x}$ et fournit en sortie le tableau B. Un pseudo code de *ComputeB* est comme suit :

Algorithme 3.2 : *ComputeB*

Données : x, m, *prev*, $pref_x^0$
Résultats : B
  pour i de 1 à $m-1$ **faire**
    $j = prev[i]$
    $L_1 = \emptyset$
    $L_2 = \emptyset$
    tant que $j \neq -1$
      tant que $L_1 \neq \emptyset$ et $first(L_1).pos >= j$
        $L_1 = next(L_1)$
      fin tant que
      si $L_1 \neq \emptyset$ et $first(L_1).pos = j-1$
        $\ell = first(L_1).length + 1$
      sinon
        $\ell = 1$
      fin si
      $L_2 = L_2.append((j, \ell))$
      si $pref_x^0[i-j] = j - \ell$ et $2j < i + pref_x^0[i-j]$
        $B[i] = B[i] \cup \{i-j\}$
      fin si
      si $pref_x^0[i] = 0$
        $B[i] = B[i] \cup \{i\}$
      fin si
      $j = prev[j]$
    fin tant que
    $L_1 = L_2$
  fin pour

Phase de recherche

La phase de recherche de l'algorithme MPE est assurée par l'algorithme *MPE_Search*. Cet algorithme permet d'examiner les facteurs communs à toutes les séquences d'entrée en analysant uniquement la séquence de référence.

Étant donné $Y = \{y_0, \ldots, y_{r-1}\}$ un ensemble de séquences fortement similaires de même longueur et un motif x de longueur m, l'algorithme *MPE_Search* fonctionne sous l'hypothèse que toutes les variations dans les séquences y_g, $1 \leq g \leq r-1$, sont de type substitution et que deux variations sont distantes par au moins $M \geq m$.

En adoptant l'algorithme *MPE_Search*, on opère comme suit :

À chaque itération j, $j < n$, on parcourt toutes les séquences à l'aide d'une fenêtre glissante de longueur m. D'abord, à la première itération ($j = 0$), on commence par comparer les caractères de x avec ceux de y_0. On vérifie au fur et à mesure s'il existe $(\mathcal{G}_p, j_p, c_p) \in Z$, $Z = ((\mathcal{G}_0, j_0, c_0), (\mathcal{G}_1, j_1, c_1), \ldots, (\mathcal{G}_{k-1}, j_{k-1}, c_{k-1}))$, où $c_p \in \Sigma$, $c_p = y_g[j_p] \neq y_0[j_p])$, $0 \leq j_p \leq n-1$, $\mathcal{G}_p = \{g \mid 1 \leq g \leq r-1, c_p = y_g[j_p] \neq y_0[j_p]\} \neq \emptyset$ pour $0 \leq p \leq k-1$ et

$0 \leq j_p \leq j + m - 1$.

Si c'est le cas, alors on a deux situations possibles :

(a) Soit $x[i] = y_0[i + j]$, $0 \leq i \leq m - 1$, dans ce cas on poursuit la recherche dans y_0 et on mémorise la position j_k. Celle-ci sera utilisée à l'itération suivante.

(b) Soit $x[i] = c_p$, dans ce cas on poursuit la recherche dans les séquences $y_g \in \mathcal{G}_k$, $1 \leq g \leq r - 1$.

De façon itérative, on effectue une transition entre les différents cas (décrits ci-dessus) en fonction de l'existence ou pas de la variation dans la fenêtre de l'itération j en cours. À la fin de chaque itération, on décale la fenêtre à droite selon le cas courant mémorisé.

Pour effectuer des transitions entre les cas, et ainsi les décalages de la fenêtre glissante, on opère comme suit :

Si une correspondance du préfixe $x[0\,..\,i]$ est trouvée dans au moins une séquence de l'ensemble Y à une position gauche j. Alors la position $i + j + 1$ est examinée et une décision doit être prise pour les 3 cas :

(a) **Cas 1** : Si $x[i + 1] = y_0[i + j + 1]$ et s'il n'existe pas de variation à la position $i+j+1$ dans les autres séquences alors on reste dans le Cas 1, sinon si une variation existe alors on passe au Cas 2. Si $x[i + 1] \neq y_0[i + j + 1]$ et s'il n'existe pas de variation dans les autres séquences à la position $i + j + 1$ alors on utilise *mpNext* pour décaler la fenêtre. Sinon s'il existe une variation et $c_k = x[i + 1]$ on passe au Cas 3, sinon si $c_k \neq x[i + 1]$ un décalage doit être effectué en utilisant *mpNext* et B (voir la figure 3.7).

(b) **Cas 2** : Si $x[i + 1] = y_0[j + 1]$ alors on reste dans le Cas 2. Si $x[i + 1] \neq y_0[j + 1]$ alors un décalage doit être effectué en utilisant$B$ (voir la figure 3.8).

(c) **Cas 3** : Si $x[i + 1] = y_0[j + 1]$ alors on reste dans le Cas 3. Si $x[i + 1] \neq y_0[j + 1]$ alors un décalage doit être effectué en utilisant$B$ (voir la figure 3.9).

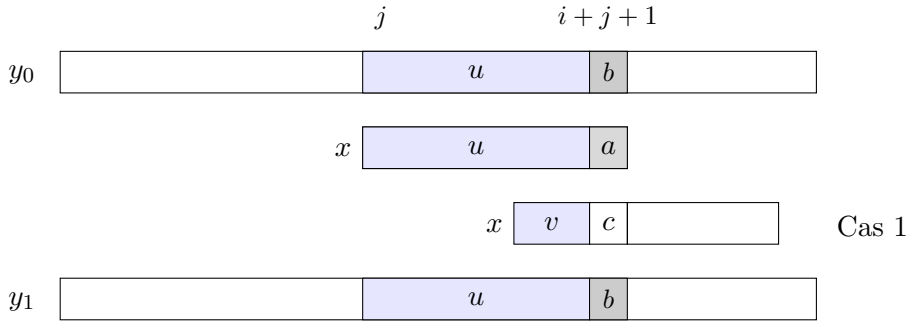


FIGURE 3.7 – Décalage dans le Cas 1. Le préfixe $u = x[0\,..\,i]$ est reconnu dans y_0 et y_1 (n'ayant pas une variation dans la fenêtre en cours par rapport à y_0). Une inégalité est trouvée à la position $i + j + 1$, $b \neq a$. Après décalage de la fenêtre en utilisant *mpNext*, on reprend les comparaisons entre $x[mpNext[i]]$ et $y_0[i + j + 1]$.

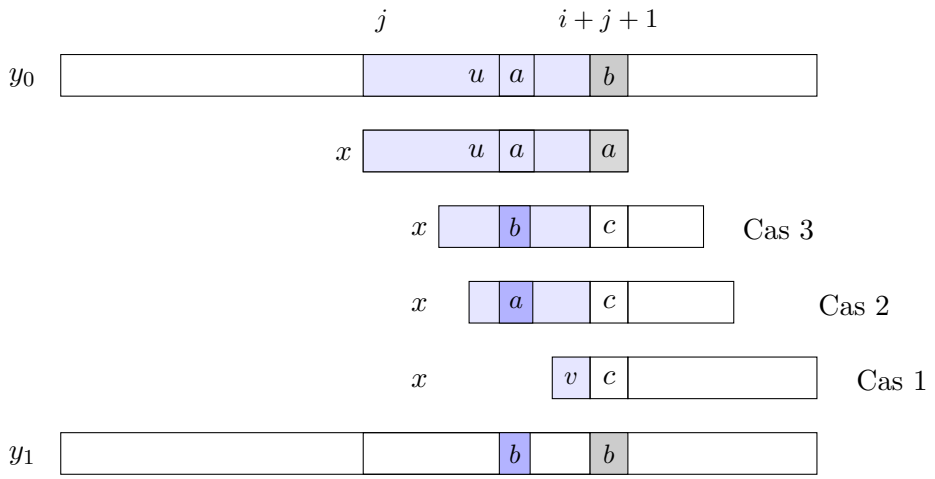


FIGURE 3.8 – Décalage dans le Cas 2. Le préfixe $u = x[0\,..\,i]$ est reconnu dans y_0 et non pas dans y_1 (ayant une variation b dans la fenêtre en cours par rapport à y_0). On a $b = y_0[i + j + 1] = y_1[i + j + 1] \neq x[i] = a$. Alors un décalage doit être effectué en utilisant B. On passe au Cas 1 si après décalage la fenêtre dépasse la variation b dans y_1. On reste dans le Cas 2 si après décalage la fenêtre contient la variation b dans y_1 mais le préfixe aligné est reconnu dans y_0. On passe au Cas 3 si après décalage la fenêtre contient la variation b dans y_1 et celle-ci permet de reconnaitre le préfixe dans y_1.

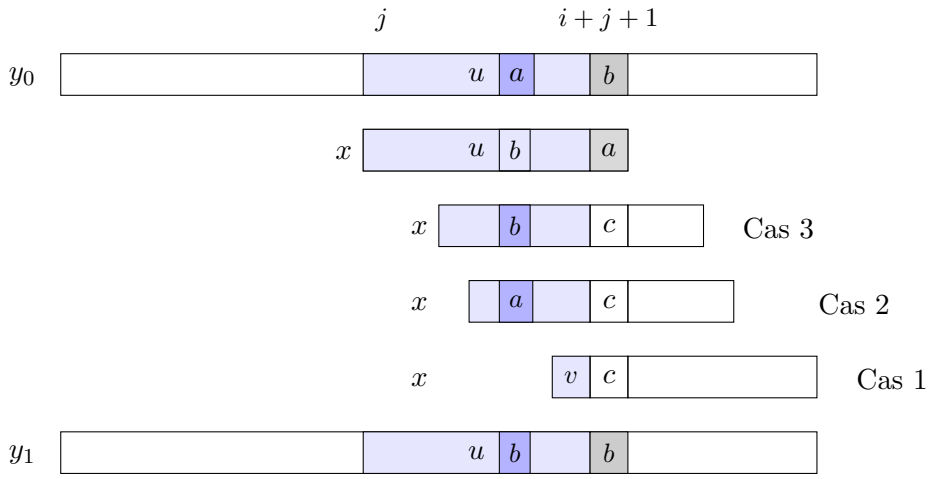


FIGURE 3.9 – Décalage dans le Cas 3. Le préfixe $u = x[0\,..\,i]$ est reconnu dans y_1 et non pas dans y_0. On a $b = y_1[i+j+1] = x[i] \neq y_0[i+j+1]$. Alors un décalage doit être effectué en utilisant B. On passe au Cas 1 si après décalage la fenêtre dépasse la variation b dans y_1. On passe au Cas 2 si après décalage la fenêtre contient la variation b dans y_1 mais le préfixe aligné est reconnu dans y_0. On reste dans le Cas 3 si après décalage la fenêtre contient la variation b dans y_1 et celle-ci permet de reconnaitre le préfixe dans y_1.

L'algorithme *MPE_Search* utilise un algorithme appelé *Shift* pour calculer des décalages de la fenêtre glissante positionnée sur les séquences d'entrée après chaque itération.

L'algorithme *MPE_Search* reçoit en entrée un motif x de longueur m, une séquence de référence y_0 de longueur n et l'ensemble Z représentant les séquences y_g, $1 \leq g \leq r-1$. Il renvoie en sortie les positions d'occurrences gauches de x dans toutes les séquences d'entrée.

L'algorithme *MPE_Search* peut être représenté par le pseudo code suivant :

Algorithme 3.3 : *MPE_Search*

1: **Données** : x, m, y_0, n, Z, $pref_x^0$, *mpNext*, B
2: **Résultats** : Positions j d'occurrences gauches de x dans toutes les séquences d'entrées
3: **Auxiliaires** : $case = 1$, $i = 0$, $j = 0$
4: **tant que** $j < n$ **faire**
5: **si** $case = 1$ **alors**
6: **si** $\exists\{(\mathcal{G}_p, j_p, c_p)\} \in Z \mid j_p = j$ **alors**
7: **tant que** $i > -1$ et $y_0[j] \neq x[i]$ et $c_p \neq x[i]$ **faire**
8: $i = mpNext[i]$
9: **fin tant que**
10: **si** $i > -1$ **alors**
11: $i_p = i$
12: **si** $x[i] = y_0[j]$ **alors**
13: $case = 2$
14: **sinon**
15: $case = 3$
16: **fin si**
17: **fin si**
18: **sinon tant que** $i > -1$ et $y_0[j] \neq x[i]$ **faire**
19: $i = mpNext$
20: **fin tant que**
21: **fin si**
22: **sinon tant que** $i > -1$ et $y_0[j] \neq x[i]$ **faire**
23: $(i, case) = Shift(x, i, j, case, mpNext, pref_x^0, B, i_p, c_p, j_p)$
24: **fin tant que**
25: **fin si**
26: $i = i + 1$
27: **si** $i = m$ **alors**
28: **Output (j - i)**
29: $(i, case) = Shift\ (x, i, j, case, mpNext, pref_x^0, B, i_p, c_p, j_p)$
30: **fin si**
31: $j = j + 1$
32: **fin tant que**

L'algorithme *Shift* reçoit en entrée le motif x, les dernières positions i et j dans respectivement x et y_0, la variable *case*, les tableaux $mpNext, pref_x^0, B$, les positions i_p, j_p de la dernière variation rencontrée c_p dans respectivement x et y_0. Il renvoie en sortie les nouvelles valeurs de i et *case*. L'algorithme *Shift* peut être représenté par le pseudo code qui suit :

Algorithme 3.4 : *Shift*

Données : $x, i, j, case, mpNext, pref_x^0, B, i_p, c_p, j_p$
Résultats : $(i, case)$
  si $case = 1$ **alors**
    $i = mpNext[i]$
  sinon si $\exists i' \in B[i-1] \mid i' + pref_x^0[i'] = i_p \text{et} x[pref_x^0[i']] = c_p$ **alors**
    si $i - i' > mpNext[i]$ **alors**
      $case = 3$
      $i = i - i'$
    sinon $i = mpNext[i]$
      si $j - i > j_p$ **alors**
        $case = 1$
      fin si
    fin si
  sinon $i = mpNext[i]$
    si $j - i > j_p$ **alors**
      $case = 1$
    fin si
  fin si
  fin si

3.3 Complexités

La proposition suivante donne une borne de nombre d'éléments dans le tableau B.

Proposition 2 *Le nombre des éléments dans B est $O(m)$.*

Le tableau B mémorise les éléments de chaque position dans v où $x = uavcwubvdw'$ ou $x = uav'ubvdw$ avec $v = v'v"$ où $v"$ est un préfixe de u et $v"d$ ne l'est pas, pour $u, v, w, w' \in \Sigma^*$ et $a, b, c, d \in \Sigma$ tel que $a \neq b$ et $c \neq d$. Puisque les chevauchements ne sont pas permis entre les occurrences de v dans de telles factorisations de x il ne peut y avoir qu'un nombre linéaire d'éléments dans B.

Proposition 3 *L'algorithme ComputeB calcule B avec une compléxité $O(m^2)$ en temps de calcul.*

Les boucles dans les lignes 7–23 parcourent tous les éléments de $pref_x^{\sim}$. Puisque le nombre total des éléments dans $pref_x^{\sim}$ est $O(m^2)$ la complexité en temps de calcul de l'algorithme *ComputeB* est également $O(m^2)$.

Theorème 4 *L'algorithme MPE_Search nécessite un temps $O(n)$.*

Supposons que les éléments de Z sont mémorisés selon l'ordre croissant des seconds composants qui sont des positions dans les séquences et supposons que chaque décalage est effectué en temps constant, la complexité de l'algorithme *MPE_Search* est dominée par les boucles des lignes 7, 18 et 22 qui effectuent les comparaisons des caractères. Le long de la même ligne comme pour l'algorithme MP, lorsqu'un symbole à une position $0 \leq j \leq n$ des séquences est traité il est soit une égalité ou une différence. Lorsqu'il s'agit d'une égalité il n'est jamais traité de nouveau. Il peut y avoir au plus n égalités. Lorsqu'il s'agit d'une inégalité un décalage est effectué. Il peut y avoir au plus n décalages et ainsi au plus n inégalités. L'ensemble en tout donne une borne linéaire sur le nombre de comparaisons.

3.4 Exemple Illustratif

$x =$ `AACATACA`, $m = 8$

a	A	C	G	T
$last[a]$	7	6	-1	4

i	0	1	2	3	4	5	6	7
$x[i]$	A	A	C	A	T	A	C	A
$prev[i]$	-1	0	-1	1	-1	3	2	5

i	0	1	2	3	4	5	6	7
$x[i]$	A	A	C	A	T	A	C	A
$mpNext[i]$	-1	0	1	0	1	0	1	0
$pref_x^0[i]$	7	1	0	1	0	1	0	1
$pref_x^{\sim}[i]$	$\emptyset$	L1	$\emptyset$	L3	$\emptyset$	L5	L6	L7
$B[i]$	$\emptyset$	$\emptyset$	{2}	{1}	{4}	{3}	{6}	{3,5}

$Y = \{y_0, y_1\}$, $r = 2$, $n = 17$
$y_0 =$ `ATGCTAGCAAGATACAG`, $y_1 =$ `ATGCTAGCAACATACAG`
$Z = (\{1\}, 10, \mathtt{C})$

Première tentative

		↓														
y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
		≠														
x	A	A	C	A	T	A	C	A								
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$ Décalage de longueur 1 donné par $i - mpNext[i] = 1 - 0$

Deuxième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
		$\neq$														
x		A	A	C	A	T	A	C	A								
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$ Décalage de longueur 1 donné par $i - mpNext[i] = 0 - -1$

Troisième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
			$\neq$													
x			A	A	C	A	T	A	C	A							
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$ Décalage de longueur 1 donné par $i - mpNext[i] = 0 - -1$

Quatrième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
				$\neq$												
x				A	A	C	A	T	A	C	A						
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$ Décalage de longueur 1 donné par $i - mpNext[i] = 0 - -1$

Cinquième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
					$\neq$											
x					A	A	C	A	T	A	C	A					
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$ Décalage de longueur 1 donné par $i - mpNext[i] = 0 - -1$

Sixième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
						$\neq$										
x						A	A	C	A	T	A	C	A				
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$ Décalage de longueur 1 donné par $i - mpNext[i] = 1 - 0$

Septième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
							$\neq$									
x							A	A	C	A	T	A	C	A			
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$ Décalage de longueur 1 donné par $i - mpNext[i] = 0 - -1$

Huitième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
								$\neq$								
x								A	A	C	A	T	A	C	A	
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$ Décalage de longueur 1 donné par $i - mpNext[i] = 0 - -1$

Neuvième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
											$\neq$					
x									A	A	C	A	T	A	C	A	
											=					
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 3$ Une occurrence de x à la position 8, Décalage de longueur 7 donné par $i - mpNext[i] = 7 - 0$

3.5 Conclusion

Dans ce chapitre, nous avons proposé un nouvel algorithme, appelé MPE [Nsira *et al.*, 2014b] permettant de traiter le problème de recherche exacte d'un seul motif dans un ensemble de séquences fortement similaires. L'algorithme MPE étend l'algorithme de Morris-Pratt [Morris and Pratt, 1970] classique de recherche d'un seul motif dans une seule séquence. Nous avons décider de proposer MPE dans l'objectif de permettre de :

(a) Exploiter la similarité entre les séquences pour éviter de ré-examiner les parties communes.

(b) Utiliser un algorithme de recherche simple et rapide qui permet de garder trace de ce qui a été fait avant la recherche actuelle. Ce qui permettra de préciser à partir de quelle position la recherche devra recommencer.

À l'issue de l'algorithme MP, MPE fait appel à la notion de répétition dans un motif. Grâce à cette notion, MPE permet de calculer des décalage en utilisant des bords à distance de Hamming 0 utilisés par l'algorithme MP classique et de nouveaux bords à distance de Hamming 1 que nous précalculons. En bref, les propriétés de MPE sont comme suit :

($P1$) Les facteurs communs entre toutes les séquences sont examinés une seule fois dans la séquence de référence. Ceci permet de tenir profit de la propriété de similarité entre les séquences. Le temps de calcul est alors réduit.

($P2$) Un bord à distance de Hamming 1 peut être plus long que le bord à distance de Hamming 0. L'algorithme MPE décide d'effectuer un décalage par le plus long des deux.

Rappelons que l'algorithme MPE utilise un modèle particulier des données. En effet, il fonctionne sous l'hypothèse que les séquences fortement similaires considérées ne contiennent que des variations de type substitutions par rapport à une séquence de référence. De plus, il suppose que ces variations sont distantes par au moins la longueur du motif. Ainsi, le modèle de données utilisé permet de considérer la totalité de la séquence de

référence avec les variations de type substitutions pour les autres séquences. De ce fait, la phase de recherche de l'algorithme MPE dépend uniquement de la longueur de la séquence de référence. Ainsi, la complexité de MPE dépend de la longueur de la séquence de référence et elle est de $O(n)$.

L'algorithme MPE est une étude préliminaire permettant d'ouvrir la porte sur la recherche incrémentale dans un ensemble de séquences fortement similaires.

Nous avons décidé d'étendre l'algorithme de Knuth-Morris-Pratt [Knuth *et al.*, 1977] classique. Ceci fait l'objet du chapitre suivant.

Chapitre 4

KMPE : Algorithme de Knuth-Morris-Pratt Étendu

Introduction

Dans ce chapitre, nous proposons un nouvel algorithme, appelé KMPE [Nsira *et al.*, 2017], permettant de faire la recherche exacte d'un motif dans un ensemble de séquences fortement similaires d'entrée.

L'algorithme KMPE est une extension de l'algorithme de Knuth-Morris-Pratt séquentiel [Knuth *et al.*, 1977] permettant de faire la recherche exacte d'un seul motif dans une seule séquence.

Le reste du chapitre est organisé comme suit :

Dans la première section 4.1, nous décrivons l'algorithme de Knuth-Morris-Pratt classique. Dans la deuxième section 4.2, nous donnons une présentation détaillée de l'algorithme KMPE. Dans la troisième section 4.3, nous calculons les complexités théoriques de l'algorithme KMPE. Dans la quatrième section 4.4, nous donnons un exemple illustratif. Enfin, dans la dernière section 4.5, nous terminons par une conclusion à ce chapitre.

4.1 KMP : Algorithme de Knuth-Morris-Pratt

Commençons d'abord par présenter les notions utilisées par l'algorithme KMP.

4.1.1 Préliminaires

Définition 4.1 *[Knuth* et al.*, 1977] Soit x un motif de longueur $m \geq 0$, le tableau kmp_x^0 associé à x est défini comme suit :*

Pour tout i, $0 \leq i \leq m$, $kmp_x^0[i]$ mémorise la longueur du plus long bord de $x[0\,..\,i-1]$ suivi par un a caractère c différent de $x[i]$ et -1 *si un tel bord n'existe pas.*

(voir la figure 4.1).

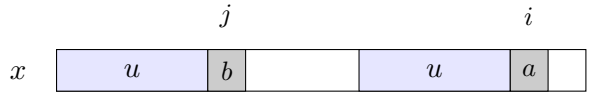


FIGURE 4.1 – $kmp_x^0[i] = \max\{j \mid x[0\,..\,j-1]$ est un bord de $x[0\,..\,i-1]$ et $x[j] \neq x[i]\} \cup \{-1\}$.

4.1.2 Description

L'algorithme KMP utilise la notion de fenêtre glissante. L'algorithme KMP se décompose en deux phases principales :

(a) Une phase de prétraitement : au cours de laquelle on construit un tableau kmp_x^0 mémorisant les décalages à effectuer, à la fin de chaque itération, sur une fenêtre balayant une séquence y d'entrée.

(b) Une phase de recherche : au cours de laquelle on localise toutes les occurrences exactes d'un motif x dans la séquence y.

Phase de prétraitement

La phase de prétraitement de l'algorithme KMP est assurée par la procédure *Calcul_kmp*$_x^0$. Cette procédure se décompose en deux étapes :

(i) D'abord, on initialise $kmp_x^0[0]$ par -1, la longueur du plus long bord du mot vide ε ;

(ii) Ensuite, pour tout i, $1 \leq i \leq m$, on mémorise la longueur de plus long bord de $x[0\,..\,i-1]$ suivi par un caractère différent de $x[i]$ dans $kmp_x^0[i]$. On calcule les valeurs de kmp_x^0 en utilisant le tableau $pref_x^0$ selon la relation suivante :

$$kmp_x^0[i + pref_x^0[i]] \geq pref_x^0[i]$$

(voir la figure 4.2).

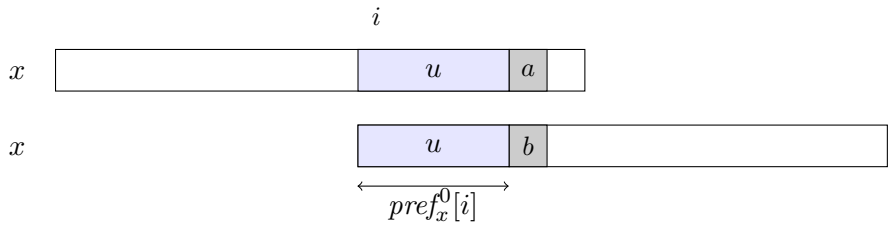


FIGURE 4.2 – $kmp_x^0[i + pref_x^0[i]] \geq pref_x^0[i]$.

Phase de recherche

La phase de recherche de l'algorithme KMP se décompose en deux étapes :

(i) D'abord, à une tentative j, $0 \leq j \leq n-1$, on compare les caractères de x avec ceux de y contenus dans la fenêtre positionnée à j sur y. Pour i, $0 \leq i \leq m-1$, si

$x[0\,..\,i] = y[j\,..\,i+j]$ est le préfixe reconnu et $x[i+1] \neq y[i+j+1]$ alors on utilise $kmp_x^0[i]$ pour décaler la fenêtre à droite. ;

(*ii*) Puis, à la tentative suivante, on poursuit les comparaisons à partir des caractères $x[kmp_x^0[i]]$ et $y[i+j+1]$. (voir la figure 4.3).

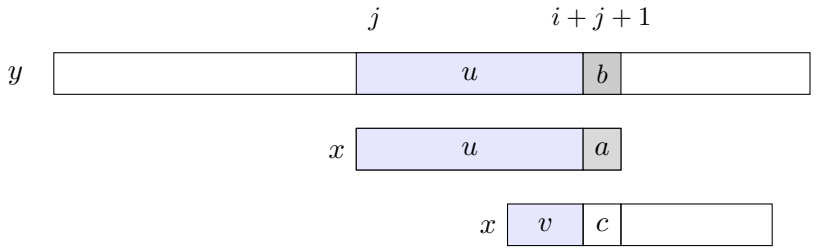


FIGURE 4.3 – Décalage avec l'algorithme KMP (v est un bord de u et $c \neq b$).

4.2 KMPE : Algorithme de Knuth-Morris-Pratt Étendu

D'abord, nous commençons par présenter des notions utilisées par l'algorithme KMPE.

4.2.1 Préliminaires

Définition 4.2 *[Nsira* et al.*, 2017] Soit x un motif de longueur m, le tableau kmp_x^1 associé à x est défini comme suit :*

Pour tout i, $0 \leq i \leq m-1$, $kmp_x^1[i]$ mémorise des bords d'un motif x à distance de Hamming 1.

Formellement :

$$kmp_x^1[i] = \{(k,j) \mid Ham(x[0\,..\,j-1], x[0\,..\,i-1]) = 1, x[k] \neq x[i-j+k] \text{ et } x[j] \neq x[i]\} \quad (4.1)$$

(voir la figure 4.4).

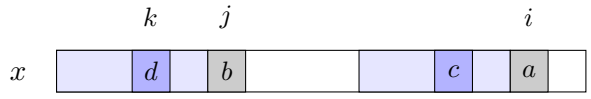


FIGURE 4.4 – $kmp_x^1[i] = \{(k,j) \mid x[0\,..\,j-1]$ est un bord à distance de Hamming 1 de $x[0\,..\,i-1]$, $x[k] \neq x[i-j+k]$ et $x[j] \neq x[i]\}$.

Les valeurs du tableau kmp_x^1 peuvent être obtenues en utilisant la relation suivante :

$$(pref_x^0[i], pref_x^0[i] + 1 + \ell) \in kmp_x^1[i + pref_x^0[i] + 1 + \ell]$$

où $\ell = |lcp(x[pref_x^0[i] + 1], x[i + pref_x^0[i] + 1])|$ (voir la figure 4.5).

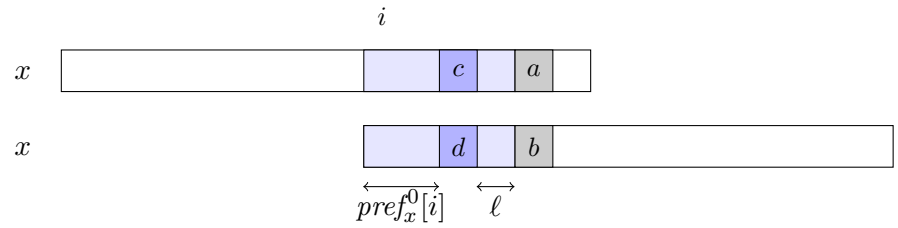


FIGURE 4.5 – $(pref_x^0[i], pref_x^0[i]+1+\ell) \in kmp_x^1[i+pref_x^0[i]+1+\ell]$ où $\ell = |lcp(x[pref_x^0[i]+1], x[i+pref_x^0[i]+1])|$.

Selon la relation décrite ci-dessus par la figure 4.5, nous avons besoin de répondre aux requêtes de plus long préfixe commun (LCP). Ces requêtes peuvent être calculées en temps constant en utilisant :

(a) Un arbre de suffixes de x et des requêtes de plus *Proche Ancêtre Commun* (en anglais *Lowest Common Ancestor queries*) ;

(b) Un tableau de suffixes de x et des RMQ (voir la section 1.5).

Pour notre propos, nous construisons un tableau $pref_x^1$ défini comme suit :

Définition 4.3 *Soit x un motif de longueur m, le tableau $pref_x^1$ associé à x est défini comme suit :*

Pour tout i, $1 \leq i \leq m-1$, $pref_x^1[i]$ mémorise la longueur de plus long préfixe commun à distance de Hamming 1 commençant à chaque position du motif x.

Formellement :

$$pref_x^1[i] = \max\{\ell \mid Ham(x[0\,..\,\ell-1], x[i\,..\,i+\ell-1]) = 1\} \tag{4.2}$$

ou de manière équivalente,

$$pref_x^1[i] = pref_x^0[i] + 1 + |lcp(x[pref_x^0[i]+1], x[i+pref_x^0[i]+1])|. \tag{4.3}$$

4.2.2 Description

L'analyse de l'algorithme KMPE est similaire à celle de l'algorithme MPE. Il utilise une fenêtre glissante pour parcourir, simultanément, les séquences d'entrée et fonctionne sous les mêmes restrictions, i.e., des variations de type substitution et sont distantes par au moins la longueur du motif. L'algorithme KMPE préserve la propriété fondamentale de l'algorithme KMP séquentiel tel qu'il permet d'améliorer encore plus les longueurs des décalages. En fait, il utilise des informations tirées de correspondances partielles entre les séquences et le motif afin de sauter des décalages garantis à l'avance qu'ils n'aboutissent pas à une reconnaissance du motif.

L'algorithme KMPE se décompose en deux phases principales :

(a) Une phase de prétraitement : au cours de laquelle on construit le tableau kmp_x^1 mémorisant de nouveaux décalages à effectuer, à la fin de chaque itération, sur la fenêtre positionnée couramment sur les séquences d'un ensemble $Y = \{y_0, y_1, \ldots, y_{r-1}\}$ d'entrée.

(b) Une phase de recherche : au cours de laquelle on localise toutes les occurrences d'un motif x dans l'ensemble Y.

L'algorithme de recherche KMPE utilise la représentation de Y par la totalité de la séquence de référence y_0 et la liste Z représentant les séquences y_g, $1 \leq g \leq r-1$ par les variations qu'elles contiennent par rapport à y_0.

Pour tout préfixe $x[0\,..\,i]$, $0 \leq i \leq m-1$, reconnu en parcourant les r séquences à une position j, $0 \leq j \leq n-1$, l'algorithme de recherche KMPE doit prendre sa décision pour les 3 cas suivants :

(a) **Cas 1** : Soit $x[i+1] = y_0[i+j+1]$. S'il n'existe pas de variation à la position $j+i+1$ dans le reste des séquences, on reste dans le Cas 1 sinon on passe au Cas 2. Si $x[i+1] \neq y_0[i+j+1]$ alors s'il n'existe pas de variation dans les autres séquences à la position $i+j+1$ un décalage est effectué avec $kmp_x^0[i]$ sinon si le caractère de variation c est égal à $x[i+1]$ on passe au Cas 3 sinon un décalage doit être effectué en utilisant kmp_x^0 et kmp_x^1.

(b) **Cas 2** : Soit $x[i+1] = y_0[i+j+1]$ alors on reste dans le Cas 2. Si $x[i+1] \neq y_0[i+j+1]$ alors un décalage doit être effectué en utilisant kmp_x^1.

(c) **Cas 3** : Soit $x[i+1] = y_0[i+j+1]$ alors on reste dans le Cas 3. Si $x[i+1] \neq y_0[i+j+1]$ alors un décalage doit être effectué en utilisant kmp_x^1.

Phase de Prétraitement

Au cours de cette phase, on construit tous les tableaux permettant de fournir les informations nécessaires à l'algorithme de recherche de KMPE pour effectuer des décalages valides à la fin de chaque tentative. On utilise le tableau kmp_x^0 de KMP pour les bords ayant une distance de Hamming 0 et on construit le tableau kmp_x^1, pour les bords ayant une distance de Hamming 1.

Dans ce qui suit nous développons les algorithmes permettant de calculer tous les tableaux utilisés.

Construction de $pref_x^1$

La construction du tableau $pref_x^1$ se fait en fonction des informations du tableau $pref_x^0$. Cette construction est assurée par un algorithme appelé *PréfixesBis*. Il s'agit d'une adaptation de l'algorithme *Préfixes* [Crochemore *et al.*, 2007].

En fait, en adoptant l'algorithme *PréfixesBis*, on opère comme suit :

(i) Durant la première étape, on construit le tableau $pref_x^0$ par utilisation des instructions de l'algorithme *Préfixes* ;

(ii) Durant la seconde étape, à chaque itération, on calcule un préfixe à distance de Hamming 1 à la position i, où $1 \leq i \leq m-1$. Puisque $pref_x^0[i]$ est la longueur de plus long préfixe commun à $x[0\,..\,i]$ et x, il est alors trivial que $x[pref_x^0[i]] \neq x[i + pref_x^0[i]]$. À partir de là, on peut estimer que l'écart entre les positions $pref_x^0[i]$ et $i + pref_x^0[i]$ vaut 1. Ainsi,

on utilise une variable j initialisée à $pref_x^0[i]+1$ puis on compare $x[j]$ et $x[i+j]$ et on incrémente la valeur de j à chaque fois où on trouve une égalité. À la fin des comparaisons, si on a $j \neq pref_x^0[i]+1$ alors $lcp(pref_x^0[i]+1, i+pref_x^0[i]+1) \neq 0$. De ce fait, il existe un préfixe à distance de Hamming 1 à la position i existe. On mémorise alors dans $pref_x^1[i]$ la valeur de j. Dans le cas échéant, si $j = pref_x^0[i]+1$ alors on mémorise dans $pref_x^1[i]$ la valeur 0.

Ceci nous donne le pseudo code suivant, où en entrée on reçoit un motif x de longueur m et en sortie on renvoie les tableaux $pref_x^0$ et $pref_x^1$. La construction de $pref_x^1$, se fait en insérant à la fin de l'algorithme *Préfixes* les instructions des lignes 14–24.

Algorithme 4.1 : *PrefixesBis*

1: **Données** : x, m
2: **Résultats** : $pref_x^0, pref_x^1$
3: $pref_x^0[0] = m$
4: $g = 0$
5: **pour** i de 1 à $m-1$ **faire**
6: **si** $i < g$ et $pref_x^0[i-f] \neq g-i$
7: $pref_x^0[i] = \min(pref_x^0[i-f], g-i)$
8: **sinon si** $i > g$
9: $g = i$
10: **fin si**
11: $f = i$
12: **tant que** $x[g] = x[g-f]$ **faire**
13: $g = g+1$
14: **fin tant que**
15: $pref_x^0[i] = g-f$
16: **Instructions ajoutées**
17: $j = pref_x^0[i]+1$
18: **tant que** $i+j \leq m$ et $x[j] = x[i+j]$ **faire**
19: $j = j+1$
20: **fin tant que**
21: **si** $j \neq pref_x^0[i]+1$ **alors**
22: $pref_x^1[i] = j$
23: **sinon**
24: $pref_x^1[i] = 0$
25: **fin si**
26: **fin si**
27: **fin pour**

Construction de kmp_x^1

L'algorithme de construction du tableau kmp_x^1 se base sur l'utilisation des tableaux $pref_x^0$ et $pref_x^1$ à l'aide du Lemme 5 suivant.

Lemme 5 *Pour i, $1 \leq i \leq m$, on a $(pref_x^0[i], pref_x^1[i]) \in kmp_x^1[i + pref_x^1[i]]$ (voir la figure 4.5).*

Pour $1 \leq i \leq m$, par définition de $pref_x^0$ nous avons $x[pref_x^0[i]] \neq x[i + pref_x^0[i]]$ et par définition de $pref_x^1[i]$ nous avons $Ham(x[0\,..\,pref_x^1[i] - 1], x[i\,..\,i + pref_x^1[i] - 1]) = 1$ et $x[pref_x^1[i]] \neq x[i + pref_x^1[i]]$ ainsi $(pref_x^0[i], pref_x^1[i]) \in kmp_x^1[i + pref_x^1[i]]$.

Notre algorithme de construction est appelé *PreKmpBis*. En adoptant cet algorithme, on opère comme suit :

(i) D'abord, on initialise les valeurs de kmp_x^1 par $\emptyset$;

(ii) Ensuite, par la boucle des lignes 7–9, on donne une formulation algorithmique du lemme 5.

L'algorithme reçoit en paramètre un motif x de longueur m et les tableaux $pref_x^0$ et $pref_x^1$ associés. Ceci donne le pseudo code qui suit :

Algorithme 4.2 : *PreKmpBis*

1: **Données** : $x, m, pref_x^0, pref_x^1$
2: **Résultats** : kmp_x^1
3: **Initialisation**
4: **pour** i de 0 à m **faire**
5: $\quad kmp_x^1[i] = \emptyset$
6: **fin pour**
7: **pour** i de $m - 1$ à 1 **faire**
8: $\quad kmp_x^1[i + pref_x^1[i]] = kmp_x^1[i + pref_x^1[i]] \cup (pref_x^0[i], pref_x^1[i])$
9: **fin pour**

Phase de recherche

La phase de recherche de l'algorithme KMPE est réalisée par un algorithme appelé *KMPE_Search*. Cet algorithme permet d'examiner les facteurs communs à toutes les séquences d'entrée en analysant uniquement la séquence de référence.

Étant donné $Y = \{y_0, \ldots, y_{r-1}\}$ un ensemble de séquences fortement similaires de même longueur et un motif x de longueur m, l'algorithme *KMPE_Search* fonctionne sous l'hypothèse que toutes les variations dans les séquences y_g, $1 \leq g \leq r - 1$, sont de type substitution et que deux variations sont distantes par au moins $M \geq m$.

En adoptant l'algorithme *KMPE_Search*, on opère comme suit :

À chaque itération j, on utilise une fenêtre glissante de longueur m pour parcourir toutes les séquences simultanément. D'abord, on commence initialement par comparer les caractères de x avec ceux en face de y_0. On vérifie lors du parcours s'il existe une variation dans les autres séquences représentées par le triplet $(\mathcal{G}_p, j_p, c_p) \in Z$, $Z = ((\mathcal{G}_0, j_0, c_0), (\mathcal{G}_1, j_1, c_1), \ldots, (\mathcal{G}_{k-1}, j_{k-1}, c_{k-1}))$, où $c_p \in \Sigma$, $c_p = y_g[j_p] \neq y_0[j_p])$, $0 \leq j_p \leq n - 1$, $\mathcal{G}_p = \{g \mid 1 \leq g \leq r - 1, c_p = y_g[j_p] \neq y_0[j_p]\} \neq \emptyset$ pour $0 \leq p \leq k - 1$ et $0 \leq j_p \leq j + m - 1$. Si un telle variation existe, alors on se trouve face à deux situations possibles :

(a) Soit $x[i] = y_0[i+j]$, $0 \leq i \leq m-1$, dans ce cas on la mémorise et on poursuit la recherche de x dans y_0. À l'itération suivante, on tient en compte de l'existence de cette variation si après calcule de décalage la nouvelle fenêtre la contiendra.

(b) Soit $x[i] = c_p$, dans ce cas on poursuit la recherche de x dans toutes les séquences qui englobent c_p.

Pour tout préfixe $x[0\,..\,i]$, $0 \leq i \leq m-1$, reconnu en parcourant les r séquences à une position j, $0 \leq j \leq n-1$, l'algorithme *KMPE_Search* doit effectuer les transitions d'un cas à un autre et calculer ses décalages comme suit :

(a) **Cas 1** : Soit $x[i+1] = y_0[i+j+1]$. S'il n'existe pas de variation à la position $j+i+1$ dans le reste des séquences, on reste dans le Cas 1 sinon on passe au Cas 2. Si $x[i+1] \neq y_0[i+j+1]$ alors s'il n'existe pas de variation dans les autres séquences à la position $i+j+1$ un décalage est effectué avec $kmp_x^0[i]$ sinon si le caractère de variation c est égal à $x[i+1]$ on passe au Cas 3 sinon un décalage doit être effectué en utilisant kmp_x^0 et kmp_x^1 (voir la figure 4.6).

(b) **Cas 2** : Soit $x[i+1] = y_0[i+j+1]$ alors on reste dans le Cas 2. Si $x[i+1] \neq y_0[i+j+1]$ alors un décalage doit être effectué en utilisant kmp_x^1 (voir la figure 4.7).

(c) **Cas 3** : Soit $x[i+1] = y_0[i+j+1]$ alors on reste dans le Cas 3. Si $x[i+1] \neq y_0[i+j+1]$ alors un décalage doit être effectué en utilisant kmp_x^1 (voir la figure 4.8).

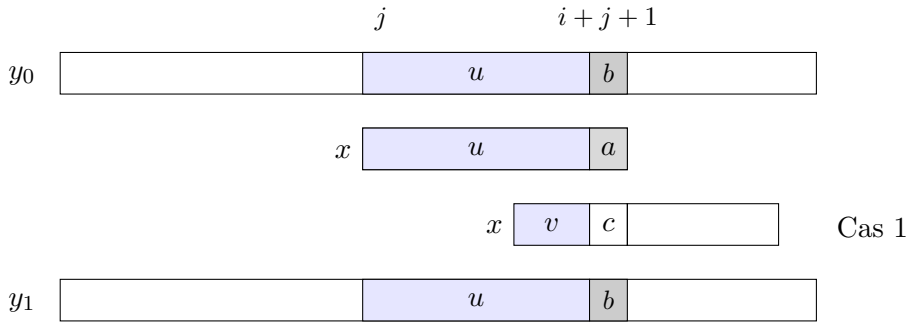


FIGURE 4.6 – Décalage dans le Cas 1 avec $c \neq b$. Le préfixe $u = x[0\,..\,i]$ est reconnu dans y_0 et y_1 (n'ayant pas de variation dans la fenêtre en cours par rapport à y_0). Une inégalité est trouvée à la position $i+j+1$, $b \neq a$. Un décalage est effectué alors en utilisant $kmp_x^0[i]$ qui correspond à la longueur du plus long bord v de $x[0\,..\,i]$ suivi par $c \neq b$.

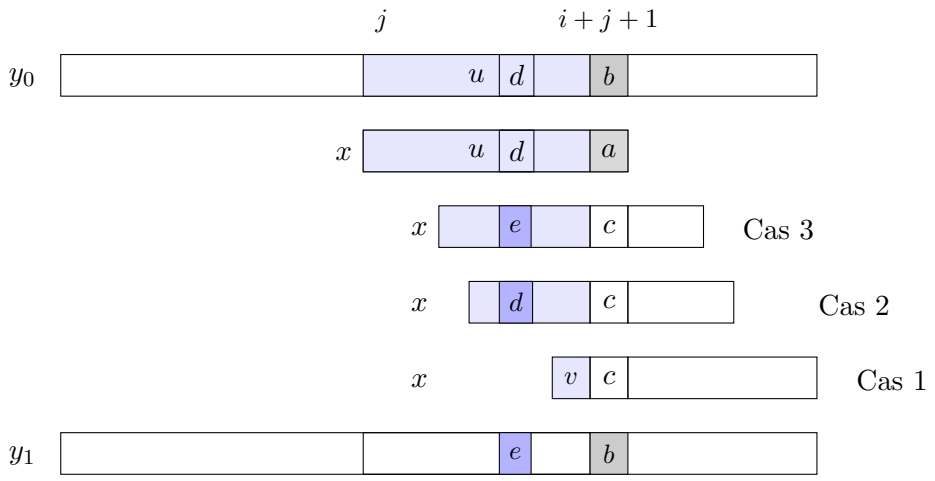


FIGURE 4.7 – Décalage dans le Cas 2 (v est un bord de u et $c \neq b$). Le préfixe $u = x[0\,..\,i]$ est reconnu dans y_0 et non pas dans y_1 (ayant une variation e dans la fenêtre en cours par rapport à y_0). On a $b = y_0[i+j+1] = y_1[i+j+1] \neq x[i] = a$. Alors un décalage doit être effectué en utilisant kmp_x^1. On passe au Cas 1 si après décalage la fenêtre dépasse la variation e dans y_1. On reste dans le Cas 2 si après décalage la fenêtre contient la variation e dans y_1 mais le préfixe aligné est reconnu dans y_0. On passe au Cas 3 si après décalage la fenêtre contient la variation e dans y_1 et celle-ci permet de reconnaitre le préfixe dans y_1.

L'algorithme *KMPE_Search* utilise un algorithme appelé *Shift* pour calculer des décalages de la fenêtre glissante positionnée sur les séquences d'entrée après chaque itération en fonction de kmp_x^0 ou kmp_x^1.

L'algorithme *KMPE_Search* reçoit en entrée un motif x de longueur m, une séquence de référence y_0 de longueur n, l'ensemble Z représentant les séquences y_g, $1 \leq g \leq r-1$ et les tableaux kmp_x^0 et kmp_x^1 précalculés lors de la phase de prétraitement. Il renvoie en sortie les positions d'occurrences gauches de x dans toutes les séquences d'entrée. Ceci nous donne le pseudo code **Algorithme 4.3**.

L'algorithme *Shift* reçoit en entrée le motif x, la séquence de référence y_0, la variable *case*, les tableaux $mpNext, pref_x^0, B$, les positions i_p, j_p de la dernière variation rencontrée c_p dans respectivement x et y_0. Il renvoie en sortie les positions d'occurrences gauches de x dans toutes les séquences d'entrée. Ainsi, on peut représenter l'algorithme *Shift* par le pseudo code **Algorithme 4.4**.

4.3 Complexités

Proposition 6 *L'algorithme PreKmpBis*$(x, m, pref_x^0, pref_x^1)$ *calcule le tableau* kmp_x^1 *en temps et espace* $O(m)$ *pour un motif* x *de longueur* m *et les tableaux* $pref_x^0$ *et* $pref_x^1$*,*

L'exactitude provient du Lemme 5. La complexité en temps est du au fait que les deux

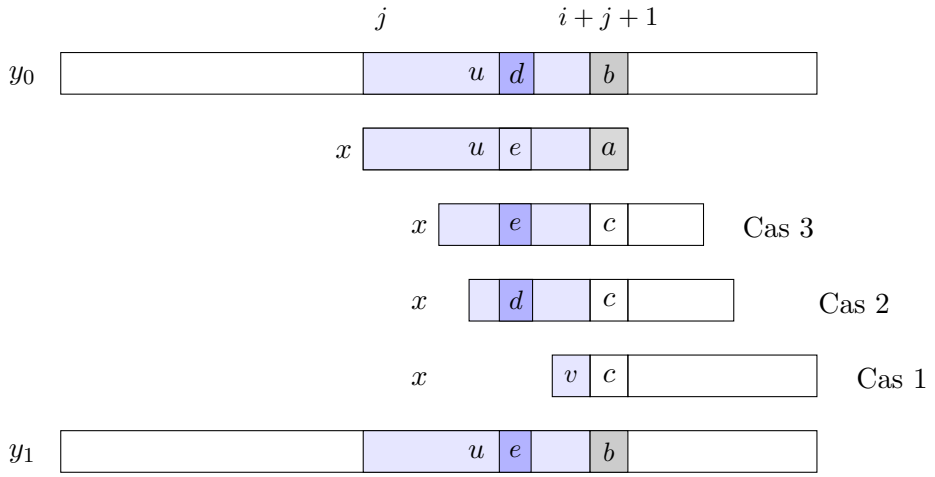


FIGURE 4.8 – Décalage dans le Cas 3 (v est un bord de u et $c \neq b$). Le préfixe $u = x[0\,..\,i]$ est reconnu dans y_1 et non pas dans y_0. On a $b = y_1[i+j+1] = y_0[i+j+1] \neq x[i] = a$. Alors un décalage doit être effectué en utilisant kmp_x^1. On passe au Cas 1 si après décalage la fenêtre dépasse la variation e dans y_1. On passe au Cas 2 si après décalage la fenêtre contient la variation e dans y_1 mais le préfixe aligné est reconnu dans y_0. On reste dans le Cas 3 si après décalage la fenêtre contient la variation e dans y_1 et celle-ci permet de reconnaitre le préfixe dans y_1.

boucles s'exécutent en temps $O(m)$ et que toutes les autres instructions s'exécutent en temps constant. La complexité en espace est du au fait que l'algorithme mis à part x, $pref_x^0$, $pref_x^1$ et kmp_x^1 n'a besoin que de la variable i.

Nous obtenons les résultats suivants.

Theorème 7 *L'algorithme KMPE_Search s'exécute en temps $O(n)$.*

La preuve est similaire à celle du Théorème 4.

4.4 Exemple Illustratif

$x =$ `AACATACA`, $m = 8$

i	0	1	2	3	4	5	6	7	8
$x[i]$	`A`	`A`	`C`	`A`	`T`	`A`	`C`	`A`	
$pref_x^0[i]$	8	1	0	1	0	1	0	1	
$pref_x^1[i]$	0	0	2	0	4	0	2	0	0
$kmp_x^0[i]$	-1	-1	1	-1	1	-1	1	-1	1
$kmp_x^1[i]$	$\emptyset$	(1,0)	$\emptyset$	(1,0)	(0,2)	(1,0)	$\emptyset$	(1,0)	(0,2),(0,4)

Algorithme 4.3 : *KMPE_Search*

1: **Données** : $x, m, y_0, n, Z = \{(g, j_p, c_p)\}, kmp_x^0, kmp_x^1$
2: **Résultats** : positions gauches d'occurrences de x dans toutes les séquences d'entrées.
3: **Auxiliaires** : $case = 1$, $i = 0$, $j = 0$.
4: **tant que** $j < n$ **faire**
5: **si** $case = 1$ **alors**
6: **si** $\exists\{(g, j_p, c_p)\} \in Z \mid j_p = j$ **alors**
7: **tant que** $i > -1$ et $y_0[j] \neq x[i]$ et $c_p \neq x[i])$ **faire**
8: $i = kmp_x^0[i]$
9: **fin tant que**
10: **si** $i > -1$ **alors**
11: $i_p = i$
12: **si** $x[i] = y_0[j]$ **alors**
13: $case = 2$
14: **sinon**
15: $case = 3$
16: **fin si**
17: **fin si**
18: **sinon tant que** $i > -1$ et $y_0[j] \neq x[i]$ **faire**
19: $i = kmp_x^1$
20: **fin tant que**
21: **fin si**
22: **sinon tant que** $i > -1$ et $y_0[j] \neq x[i]$ **faire**
23: $(i, case) = Shift(x, i, case, kmp_x^0, kmp_x^1)$
24: **fin tant que**
25: **fin si**
26: $i = i + 1$
27: **si** $i = m$
28: **Output (j - i)**
29: $(i, case) = Shift(x, i, case, kmp_x^0, kmp_x^1)$
30: **fin si**
31: $j = j + 1$
32: **fin si**
33: **fin tant que**

$Y = \{y_0, y_1\}$, $r = 2$, $n = 17$
$y_0 =$ `ATGCTAGCAAGATACAG`, $y_1 =$ `ATGCTAGCAACATACAG`
$Z = (\{1\}, 10, \texttt{C})$

Algorithme 4.4 : *Shift*

1: **Données** : $x, i, j, case, kmp_x^0, kmp_x^1, pref_x^0, i_p, c_p, j_p$
2: **Résultats** : $(i, case)$
3: **si** $case = 1$ **alors**
4: $i = kmp_x^0[i]$
5: **sinon**
6: **si** $\exists pos = i - kmp_x^1[i].length$ et $i_p = pos + kmp_x^1[i].pos \mid x[kmp_x^1[i]).pos] = cp$
7: **si** $i - pos > kmp_x^0[i]$ **alors**
8: $i = i - pos$
9: $case = 2$
10: **sinon**
11: $i = kmp_x^0[i]$
12: **si** $j - i > jp$
13: $case = 0$
14: **fin si**
15: **fin si**
16: **sinon**
17: $i = kmp_x^0[i]$
18: **si** $j - i > jp$ **alors**
19: $case = 0$
20: **fin si**
21: **fin si**
22: **fin si**

Première tentative

	↓															
y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
		≠														
x	A	A	C	A	T	A	C	A								
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$
Décalage de longueur 2 donné par $i - kmp_x^0[i] = 1 - -1$

Deuxième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
			≠													
x			A	A	C	A	T	A	C	A							
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$ Décalage de longueur 1 donné par $i - kmp_x^0[i] = 0 - -1$

Troisième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
				$\neq$												
x				A	A	C	A	T	A	C	A						
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$ Décalage de longueur 1 donné par $i - kmp_x^0[i] = 0 - -1$

Quatrième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
					$\neq$											
x					A	A	C	A	T	A	C	A					
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$ Décalage de longueur 1 donné par $i - kmp_x^0[i] = 0 - -1$

Cinquième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
							$\neq$									
x						A	A	C	A	T	A	C	A				
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$ Décalage de longueur 2 donné par $i - kmp_x^0[i] = 1 - -1$

Sixième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
								$\neq$								
x								A	A	C	A	T	A	C	A		
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 1$ Décalage de longueur 1 donné par $i - kmp_x^0[i] = 0 - -1$

Septième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
									=	=	$\neq$	=	=	=	=	=	
x									A	A	C	A	T	A	C	A	
											=						
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

$case = 3$ Une occurrence de x à la position 8, Décalage de longueur 8 donné par $i - kmp_x^0[i] = 7 - -1$, $case = 3$

4.5 Conclusion

Dans ce chapitre, nous avons proposé un nouvel algorithme, appelé KMPE [Nsira *et al.*, 2017]. permettant de traiter le problème de recherche exacte d'un seul motif dans un

ensemble de séquences fortement similaires. L'algorithme KMPE est une manière d'optimiser encore plus l'algorithme MPE présenté dans le chapitre qui précède. En effet, KMPE préserve la propriété fondamentale de l'algorithme KMP classique qui fait qu'il soit encore plus avantageux que l'algorithme MPE comme suit :

(*P1*) Des informations tirées des égalités partielles entre l'**ensemble de séquences fortement similaires** et le motif sont utilisées afin d'éviter des décalages inutiles (dont on est sûr à l'avance qu'ils n'aboutissement pas à une reconnaissance du motif). Ainsi, de meilleurs décalages sont réalisés en utilisant des bords à distance de Hamming 0 et d'autres à distance de Hamming 1. Cependant, KMPE est restreint à un modèle particulier de données où on suppose que les variations ne sont que de type substitutions et sont distances par un certain écart.

La complexité de KMPE dépend de la longueur de la séquence de référence et elle est de $O(n)$.

Dans le chapitre qui suit, nous proposons un autre algorithme qui vise à optimiser encore plus la recherche d'un motif dans un ensemble de séquences fortement similaires afin de réaliser de meilleur résultats en pratique.

Chapitre 5

FSE : Algorithme de Recherche Rapide Étendu

Introduction

Dans ce chapitre, nous proposons un nouvel algorithme incrémental, appelé FSE [Nsira *et al.*, 2014a], pour localiser toutes les occurrences exactes d'un seul motif dans un ensemble de séquences fortement similaires.

En effet, l' algorithme FSE permet d'étendre les variantes de l'algorithme de Boyer-Moore [Boyer and Moore, 1977] de façon similaire que l'algorithme de recherche rapide [Cantone and Faro, 2005].

Le reste du chapitre est organisé comme suit :

Dans la première section 5.1, nous présentons l'algorithme rapide. Dans la deuxième section 5.2, nous décrivons notre nouvel algorithme FSE. Dans la troisième section 5.3, nous calculons les complexités théoriques de l'algorithme FSE. Dans la quatrième section 5.4, nous donnons un exemple illustratif. Enfin, dans la dernière section 5.5, nous présentons une conclusion à ce chapitre.

5.1 FS : Algorithme de Recherche Rapide

D'abord, nous commençons par présenter les notions utilisées par l'algorithme FS.

5.1.1 Préliminaires

Définition 5.1 *Soit x un motif de longueur $m \geq 0$, le tableau $suff_x^0$ associé à x est défini comme suit :*

Pour tout i, $0 \leq i \leq m$, $suff_x^0[i]$ mémorise la longueur de plus long suffixe commun entre $x[0\,..\,i]$ et le motif x même.

Définition 5.2 *[Boyer and Moore, 1977] Soit x un motif de longueur $m \geq 0$, le tableau bc_x associé à x est défini comme suit :*

Pour tout $c \in \Sigma$, le tableau bc_x mémorise la position de sa dernière occurrence dans dans x. Formellement :

$$bc_x[c] = \min\{0 \leq i < m \mid x[m-1-i] = c\} \cup \{m\}. \tag{5.1}$$

Définition 5.3 *[Boyer and Moore, 1977] Soit x un motif de longueur $m \geq 0$. La **Condition de Suffixe** et la **Condition d'Occurrence** de caractère sont deux conditions booléennes définies pour chaque position i sur x et tout décalage s de x comme suit :*

$$SC_x^0(i,s) = \begin{cases} 0 < s \leq i+1 \text{ et } x[i+1-s\,..\,m-1-s] \text{ est un suffixe de } x \\ ou \\ i+1 < s \text{ et } x[0\,..\,m-1-s] \text{ est un suffixe de } x \end{cases}$$

et

$$OC_x^0(i,d) = \begin{cases} 0 < s \leq i \text{ et } x[i-s] \neq x[i] \\ ou \\ i < s. \end{cases}$$

(voir les figures 5.1 et 5.2).

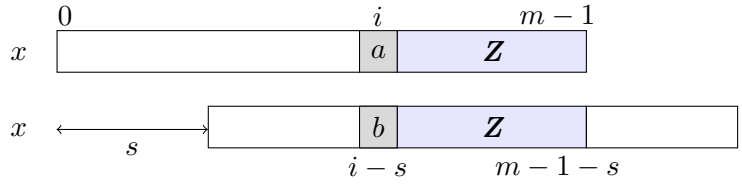


FIGURE 5.1 – Si $0 < s \leq i+1$ alors $x[i+1-s\,..\,m-1-s] = x[i+1\,..\,m-1]$ et si $0 < s \leq i$ alors $x[i-s]] \neq x[i]$.

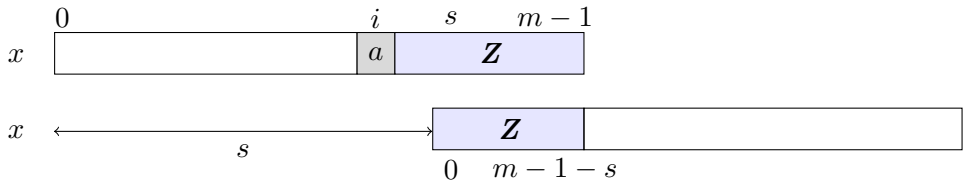


FIGURE 5.2 – Si $i+1 < s$ alors $x[0\,..\,m-1-s] = x[s\,..\,m-1]$.

Définition 5.4 *Soit x un motif de longueur $m \geq 0$, le tableau gs_x^0 associé à x est défini comme suit :*

Pour tout i, $0 \leq i \leq m$, on a :

$$gs_x^0[i] = \min\{s \mid SC_x^0(i,s) \text{ et } OC_x^0(i,s) \text{ sont satisfaites}\}. \tag{5.2}$$

ou également :

$$gs_x^0[i] = \min\{s \mid x[i-s+1\,..\,m-1-s] = x[i+1\,..\,m-1] \text{ et } x[i-s] \neq x[i]\}. \tag{5.3}$$

$gs_x^0[i]$ est bien défini puisque les conditions sont satisfaites pour $s = m$.

5.1.2 Description

L'algorithme FS utilise la notion de fenêtre glissante afin de parcourir une séquence d'entrée de droite à gauche en commençant par le caractère le plus à droite de la fenêtre. L'algorithme FS se décompose en deux phases principales :

(a) Une phase de prétraitement : au cours de laquelle on construit les tableaux bc_x et gs_x^0 mémorisant les décalages à effectuer vers la droite, à la fin de chaque itération, sur une fenêtre balayant une séquence y d'entrée.

(b) Une phase de recherche : au cours de laquelle on localise toutes les occurrences d'un motif x dans la séquence y.

Phase de prétraitement

La phase de prétraitement de l'algorithme FS est assurée par les procédures *Calcul_*bc_x et *Calcul_*gs_x^0 permettant de construire respectivement les tableaux bc_x et gs_x^0 associés à un motif x.

La procédure *Calcul_*bc_x permet de remplir le tableau bc_x par les valeurs des décalages permettant d'aligner chaque caractère de l'alphabet avec son occurrence la plus à droite dans $x[0\,..\,m-2]$ s'il en existe. Cette procédure se décompose en deux étapes comme suit :

(i) Durant la première étape, pour tout $c \in \Sigma$, on initialise $bc_x[c]$ par la valeur m de la longueur de x ;

(ii) Durant la seconde étape, pour tout caractère $x[i]$, $0 \leq i \leq m-2$, on mémorise dans $bc_x[x[i]]$ la valeur $m-1-i$ qui correspond à la position maximale d'occurrence de $x[i]$ dans x.

La procédure *Calcul_*gs_x^0 se base sur le tableau $suff_x^0$. En adoptant cette procédure, on opère comme suit :

(i) D'abord, pour tout $i \leq m$, on initialise $gs_x^0[i]$ par m ;

(ii) Ensuite, à l'aide d'une première boucle sur les valeurs de i allant de $m-2$ à -1, on vérifie si $i = -1$ ou bien si $suff_x^0[i] = i+1$. Si la condition est vérifiée alors on mémorise dans $gs_x^0[j]$ la valeur $m-1-i$, $0 \leq j < m-1-i$. On utilise une deuxième boucle pour mémoriser dans $gs_x^0[m-1-suff_x^0[i]]$ la valeur $m-1-i$, pour $0 \leq i \leq m-2$.

Phase de recherche

La phase de recherche de l'algorithme FS est comme suit :

On considère une tentative à une position j sur y, où $0 \leq j < n$:

(*i*) Si $y[j+m-1] \neq x[m-1]$ alors bc_x est utilisé pour décaler la fenêtre en cours. Lors du prochain décalage, $y[j+m-1]$ sera aligné avec son occurrence la plus à droite dans $x[0\,..\,m-2]$ si elle existe, sinon le décalage est effectué en passant juste la fenêtre en cours ;

(*ii*) Sinon, pour toute autre situation gs_x^0 est alors utilisé pour décaler la fenêtre en cours.

5.2 FSE : Algorithme de Recherche Rapide Étendu

D'abord, nous commençons par présenter des préliminaires utiles pour l'algorithme FSE et nécessaires pour la compréhension de ce qui suit.

5.2.1 Préliminaires

Définition 5.5 *[Nsira* et al.*, 2014a] Soit x un motif de longueur m, le tableau $suff_x^1$ associé à x est défini comme suit :*

Pour tout i, $0 \leq i < m-1$, $suff_x^1[i]$ mémorise la longueur maximale des suffixes de x à distance de Hamming 1 apparaissant à la position droite i sur x. Formellement :

$$suff_x^1[i] = \max\{\ell \mid Ham(x[i-\ell+1\,..\,i], x[m-\ell\,..\,m-1]) = 1\}. \tag{5.4}$$

De manière équivalente, soit $\ell = lcsuff(x[0\,..\,i], x)$, ainsi

$$suff_x^1[i] = \ell + 1 + lcsuff(x[0\,..\,i-\ell-1], x[0\,..\,m-\ell-2]) \tag{5.5}$$

(voir la figure 5.3).

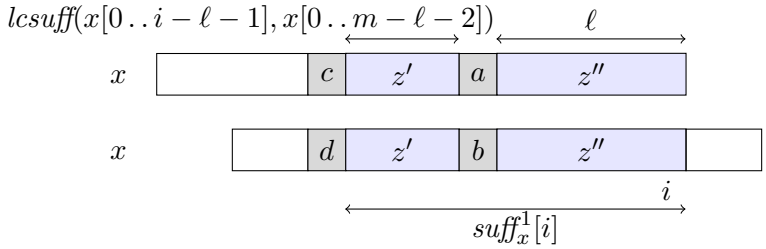


FIGURE 5.3 – $suff_x^1[i] = \ell+1+lcsuff(x[0\,..\,i-\ell-1], x[0\,..\,m-\ell-2])$ où $\ell = lcsuff(x[0\,..\,i], x])$.

Définition 5.6 *[Nsira* et al.*, 2014a] Soit x un motif de longueur $m \geq 0$, SC_x^1 est une condition booléenne définie comme suit :*

Pour toute position i sur x, $0 \leq i \leq m-1$, et pour chaque décalage s de x par on a :

$$SC_x^1(i,s) = \begin{cases} 0 < s \leq i+1 \text{ et } Ham(x[i+1-s\,..\,m-1-s], x[i+1\,..\,m-1]) = 1 \\ ou \\ i+1 < s \text{ et } Ham(x[0\,..\,m-1-s], x[d\,..\,m-1]) = 1. \end{cases}$$

(voir les figures 5.4 et 5.5).

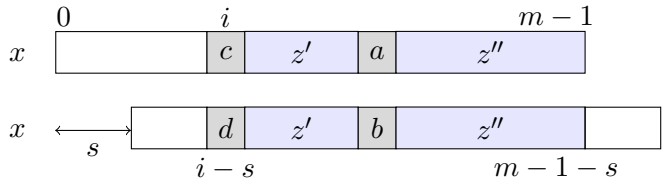


FIGURE 5.4 – Si $0 < s \leq i+1$ alors $Ham(x[i+1-s\,..\,m-1-s], x[i+1\,..\,m-1]) = 1$ et si $0 < s \leq i$ alors $x[i-s]] \neq x[i]$.

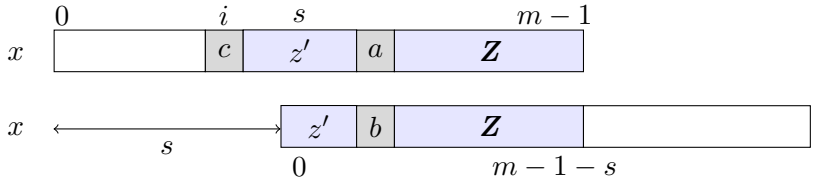


FIGURE 5.5 – Si $i+1 < s$ alors $x[0\,..\,m-1-s] = x[s\,..\,m-1]$.

Définition 5.7 *[Nsira* et al.*, 2014a] Soit x un motif de longueur $m \geq 0$, le tableau gs_x^1 associé à x est défini comme suit :*

Pour toute de position i sur x, $0 \leq i \leq m-1$ on a :

$$gs_x^1[i] = \min\{s \mid SC_x^1(i,s) \text{ et } OC_x^0(i,s) \text{ sont satisfaites}\} \tag{5.6}$$

ceci revient à dire que :

$$gs_x^1[i] = \min\{s \quad | \quad \begin{array}{ll} Ham(& x[i-s+1\,..\,m-1-s], x[i+1\,..\,m-1]) = 1 \\ et & x[i-s] \neq x[i]\}. \end{array} \tag{5.7}$$

Il est à noter que $gs_x^1[i]$ est bien défini puisque les conditions sont satisfaites pour $s = m$.

Définition 5.8 *[Nsira* et al.*, 2014a]*

Soit x un motif de longueur $m \geq 0$, le tableau gs_x associé à x est défini comme suit :

Pour toute de position i sur x, $0 \leq i \leq m-1$ on a :

$$gs_x[i] = \min\{gs_x^0[i], gs_x^1[i]\} \tag{5.8}$$

5.2.2 Description

Par extension de l'algorithme FS séquentiel, l'algorithme FSE utilise une feneêtre glissante afin d'aligner un motif donné avec toutes les séquences d'entrée simultanément. Les propriétés de l'algorithme FS sont préservées telles que :

(i) La condition de suffixe et la condition d'occurrence de caractères sont fondamentales pour calculer des décalages lorsque l'on considère des suffixes à distance de Hamming 1 ;

(ii) Mémoire faible sur les tentatives.

L'algorithme FSE se décompose en deux phases principales :

(a) Une phase de prétraitement : au cours de laquelle on construit les tableaux gs_x^1 et gs_x mémorisant de nouveaux décalages à effectuer, à la fin de chaque itération, sur la fenêtre positionnée couramment sur les séquences d'un ensemble $Y = \{y_0, y_1, \ldots, y_{r-1}\}$ d'entrée.

(b) Une phase de recherche : au cours de laquelle on localise toutes les occurrences d'un motif x dans l'ensemble Y d'entrée.

L'algorithme de recherche FSE considère la totalité de la séquence de référence y_0 et une liste Z représentant les séquences y_g, $1 \leq g \leq r-1$ par les variations qu'elles contiennent par rapport à y_0.

Une vue d'ensemble de l'algorithme FSE est comme suit. Lors de comparaison d'un seul motif donné avec l'ensemble de séquences fortement similaires :

(i) Si une inégalité est trouvée immédiatement sur le caractère le plus à droite de la fenêtre en cours sur l'ensemble de séquences fortement similaires (pour ceci il faut vérifier dans y_0 et dans Z), alors l'algorithme effectue un décalage par le minimum des décalages de dernières occurrences des caractères courants dans les séquences ;

(ii) Si une inégalité est trouvée à une comparaison ultérieure de la tentative courante, alors l'algorithme effectue un décalage par la valeur minimale entre le décalage du bon suffixe classique et le nouveau décalage du bon suffixe précalculé pour les suffixes du motif à distance de Hamming 1.

Phase de Prétraitement

Au cours de cette phase de prétraitement, on construit les tableaux utilisés par l'algorithme FSE pendant sa phase de recherche. En fait, pour la construction des tableaux bc_x et gs_x^0 on utilise des algorithmes appelés respectivement *PreBmBc* et *PreBmGs* [Charras and Lecroq, 2004b].

Dans les sous-paragraphes suivants, nous développons les algorithmes permettant de construire les tableaux gs_x^0 et gs_x^1.

Construction des tableaux gs_x^0 et gs_x^1

D'après ce qui a été décrit précédemment, le calcul des décalages du bon suffixe dépend essentiellement du calculs des suffixes. Ainsi, avant de calculer les tableaux gs_x^0 et gs_x^1, on doit répondre aux requêtes des longueurs maximales des suffixes communs à distance de Hamming 0 et 1.

Il est clair que $suff_x^0$ et $suff_x^1$ sont analogues aux tableaux $pref_x^0$ et $pref_x^1$ (voir le chapitre 3) lorsque l'on considère le miroir du motif. Ainsi le calcul de $suff_x^0$ et $suff_x^1$ peut être fait de façon similaire.

En fait, la construction de $suff_x^0$ et $suff_x^1$ se fait en utilisant un algorithme linéaire, appelé *Suffixes* [Charras and Lecroq, 2004b].

La construction de $suff_x^0$ et $suff_x^1$ peut se faire à l'aide de pseudo code suivant, où on reçoit en entrée un motif x de longueur m.

Algorithme 5.1 : *SuffixesBis*

Données : x, m
Résultats : $suff_x^0, suff_x^1$
$suff_x^0[m-1] = m$
$g = m - 1$
pour i de $m-2$ à 0 **faire**
    si $i > g$ et $suff_x^0[i + m - 1 - f] \neq i - g$ **alors**
        $suff_x^0[i] = \min(suff_x^0[i + m - 1 - f], i - g)$
    sinon si $i < g$ **alors**
        $g = i$
    fin si
    $f = i$
    tant que $g \geq 0$ et $x[g] = x[g + m - 1 - f]$ **faire**
        $g = g - 1$
    fin tant que $suff_x^0[i] = f - g$
    Instructions ajoutées
    $\ell = \ell - suff_x^0[i] - 1$
    tant que $j \geq 0$ et$x[\ell] = x[m - 1 - i + \ell]$ **faire**
    $\ell = \ell - 1$
    fin tant que
    si $\ell \neq i - suff_x^0[i] - 1$ **alors**
        $suff_x^1[i] = i - \ell$
    sinon
        $suff_x^1[i] = 0$
    fin si
    fin si
fin tant que

La construction de gs_x^1 peut se faire en utilisant un algorithme appelé, *PreBmGsBis*. Cet algorithme se base sur l'utilisation de $suff_x^1$, en utilisant la la relation suivante :

$$gs_x^1[m - 1 - suff_x^1[i]] \leq m - 1 - i \tag{5.9}$$

pour $0 \leq i \leq m - 2$.

En adoptant l'algorithme *PreBmGsBis*, on traite des bords à distance de Hamming 1 de x trouvés avec $suff_x^1[i] = i + 1$. On consière les valeurs de i selon l'ordre croissant.

La formulation de l'algorithme *PreBmGsBis* peut se réaliser par le pseudo code qui suit. En entrée, on reçoit un motif x de longueur m et le tableau $suff_x^1$ associé. En sortie,on fournit le tableau gs_x^1.

Algorithme 5.2 : *PreBmGsBis*

```
1: Données : x, m, suff_x^1
2: Résultats : gs_x^1
3: Initialisation
4: pour i de 0 à m − 1 faire
5:     gs_x^1[i] = m
6: fin pour
7: Bords à distance de Hamming 1
8: k = 0
9: pour i de m − 2 à −1 faire
10:    si i = −1 ou suff_x^1[i] = i + 1 alors
11:        tant que k < m − 1 − i faire
12:            gs_x^1[k] = m − 1 − i
13:            k = k + 1
14:        fin tant que
15:    fin si
16: fin pour
17: Re-occurrences de suffixes distance de Hamming 1
18: pour i de 0 à m − 2 faire
19:    gs_x^1[m − 1 − suff_x^1[i]] = m − 1 − i
20: fin pour
```

Lemme 8 *Pour* $0 \leq i \leq m-2$, *si* $suff_x^1[i] = i + 1$ *alors, pour* $0 \leq j < m - 1 - i$, $gs_x^1[j] \leq m - 1 - i$.

La supposition que $suff_x^1[i] = i + 1$ est équivalente à $x[0\,..\,i]$ est un suffixe de x à distance de Hamming 1. Soit j un indice qui satisfait $0 \leq j < m - 1 - i$. La condition $SC_x^1(j, m-1-i)$ est satisfaite depuis $m - 1 - i > j$ et $x[0\,..\,m-(m-1-i)-1] = x[0\,..\,i]$ est un suffixe de x. Il est en est de même pour la condition $OC_x^0(j, m-1-i)$ puisque $m - 1 - i > j$. Ceci montre, par définition de gs_x^1, que $gs_x^1[j] \leq m - 1 - i$ comme indiqué.

Lemme 9 *Pour* $0 \leq i \leq m-2$, *on a* $gs_x^1[m - 1 - suff_x^1[i]] \leq m - 1 - i$.

Si $suff_x^1[i] < i + 1$, la condition $SC_x^1(m - 1 - suff_x^1[i], m - 1 - i)$ est satisfaite puique nous avons d'une part $m - 1 - i \leq m - 1 - suff_x^1[i]$ et d'autre part $x[i - suff_x^1[i] + 1\,..\,i] = x[m - 1 - suff_x^1[i] + 1\,..\,m - 1]$. De plus, la condition $OC_x^0(m - 1 - suff_x^1[i], m - 1 - i)$ est

aussi satisfaite depuis $x[i - suff_x^1[i]] \neq x[m-1-suff_x^1[i]]$ par définition de $suff_x^1$. Ainsi $gs_x^1[m-1-suff_x^1[i]] \leq m-1-i$.

Maintenant si $suff_x^1[i] = i+1$, par le Lemme 8, nous avons en particulier pour $j = m-1-suff_x^1[i] = m-i-2$, l'inégalité $gs_x^1[j] \leq m-1-i$. Ceci met fin à la preuve.

Phase de recherche

La phase de recherche de l'algorithme FSE est assurée par l'algorithme *FSE_Search*. Lors des comparaisons, l'algorithme *FSE_Search* compare les caractères du motif avec ceux de la séquence de référence de l'ensemble de séquences d'entrée et vérifie au fur et à mesure l'existence de variations dans les autres séquences. Ceci fait que toutes les séquences sont parcourues simultanément en utilisant la même boucle.

Étant donnè $Y = \{y_0, \ldots, y_{r-1}\}$ un ensemble de sèquences fortement similaires de même longueur et un motif x de longueur m, l'algorithme *FSE_Search* fonctionne sous l'hypothése que toutes les variations dans les sèquences y_g, $1 \leq g \leq r-1$, sont de type substitution et que deux variations sont distantes par au moins $M \geq m$.

En adoptant l'algorithme *FSE_Search*, on opére comme suit.

On considère une tentative à une position j, $j < n$, sur l'ensemble de séquences fortement similaires.

D'abord, on commence par comparer les facteurs de la position j à la position $j+m-1$ des séquences fortement similaires avec le motif x de droite à gauche. Ensuite, si on trouve une inégalité à une position i de x ou bien si on reconnait le motif alors les valeurs de gs_x^0 et gs_x^1 satisfaisaient les deux conditions pour effectuer des décalages corrects et valides. La première condition assure que le facteur reconnu $v = y_0[j+i+1\,..\,j+m-1]$ (impliquant la cas où le facteur est reconnu dans les séquence(s) de l'ensemble Z) est aligné avec son occurrence la plus à droite comme facteur du motif si une telle occurrence existe. Si cette occurrence n'existe pas alors le décalage permettra d'aligner le plus long préfixe de v qui correspond à un préfixe de x (toujours en tenant en compte des variations présentes dans Z). La deuxième condition assure qu'après décalage le caractère $b = y[j+i]$ est aligné avec le caractère c qui est différent de $x[i]$ (celui qui était aligné avec b juste avant le décalage).

Ainsi, lors du parcours du motif x de droite à gauche à la position en cours j de la fenêtre sur les séquences on a :

(a) Si $x[m-1] \neq y_0[j+m-1]$ et $x[m-1] \neq c$, où $c \in \Sigma$ et $(\mathcal{G}, j+m-1, c) \in Z$ alors on décale la fenêtre par $\min\{bc_x[y_0[j+m-1]], bc_x[c]\}$.

(b) Si $x[i+1\,..\,m-1]$ est reconnu dans certaines séquences et $x[i] \neq y_0[j+i]$, où $0 \leq i \leq m-2$, $x[i] \neq c$ pour tout$c \in \Sigma$ tel que $(\mathcal{G}, j+m-1, c) \in Z$ alors la longueur de décalage de la fenêtre est donné par $\min\{gs_x^0[i], gs_x^1[i]\}$.

L'algorithme *FSE_Search* peut être décrit par le pseudo-code qui suit. Il reçoit en paramétre un motif x de longueur m, la séquence de référence y_0 de longueur n, la liste Z des variations triées selon un ordre croissant des positions sur les séquences et les tableaux bc_x et gs_x. Il renvoie en sortie les positions gauches d'occurrences de x dans les séquences.

Algorithme 5.3 : *FSE_Search*

1: **Données** : $x, m, y_0, n, Z, gs_x, bc_x$
2: **Résultats** : Positions j d'occurrences gauches de x dans toutes les séquences d'entrées
3: $j = 0$
4: **tant que** $j < n$ **faire**
5: **tant que** $j < n - m$ et $\min\{bc_x[y_0[j+m-1]], bc_x[c]\forall(\mathcal{G}, j+m-1, c) \in Z\} > 0$ **faire**
6: $j = j + \min(bc_x[y_0[j+m-1]], bc_x[c]\forall(\mathcal{G}, j+m-1, c) \in Z$
7: **fin tant que**
8: **si** $j < n$ **alors**
9: $i = m - 2$
10: **tant que** $i \geq 0$ et $x[i] = y_0[i+j]$ ou $\exists(\mathcal{G}, i+j, x[i]) \in Z$ **faire**
11: $i = i - 1$
12: **fin tant que**
13: **si** $i < 0$ **alors**
14: **Output (j)**
15: $i = 0$
16: **fin si**
17: $j = j + gs_x[i]$
18: **fin si**
19: **fin tant que**

5.3 Complexités

Proposition 10 *Soit σ la taille de l'alphabet. Le calcul de bc_x peut être effectué en temps et espace $O(m+\sigma)$. L'algorithme SuffixesBis calcule $suff_x^0$ et $suff_x^1$ en temps $O(m^2)$ et en espace $O(m)$. Le calcul de gs_x^0 et gs_x^1 peut être effectué en temps et espace $O(m)$, avec $suff_x^0$ et $suff_x^1$ donnés. Ainsi les algorithmes de prétraitement appliqués à un motif de longueur m s'exécute en temps $O(m^2+\sigma)$ et nécessite un espace supplémentaire $O(m+\sigma)$.*

Theorème 11 *L'algorithme FSE_Search trouve toutes les occurrences de x dans les r séquences de Y representées par y_0 et Z en temps $O(mnr)$.*

Durant une tentative à une position j sur un ensemble de séquences, l'algorithme détérmine s'il existe une occurrence de x dans les lignes 10-12. Il reste à montrer que tous les décalages éffectués par l'algorithme sont valides. Les décalages éffectués dans la ligne 10 sont valides de la minimalité du tableau bc_x. Les décalages effectués dans la ligne 10 sont valides de la minimalité des tableaux gs_x^0, gs_x^1 et gs_x.

Dans sa forme actuelle, l'algorithme *FSE_Search* calcule uniquement la position des occurrences du motif dans au moins une séquence de Y. Si quelqu'un veut aussi calculer les séquences actuelles où le motif apparaît, il est possible de maintenir un ensemble de numéros de séquence selon les trois cas détaillés dans le chapitre. 3.

Dans le meilleur des cas l'algorithme *FSE_Search* peut séxecuter en temps $O(n/m)$ time. Nous allons voir dans ce qui suit que bien que sa complexité du temps très pessimiste, il fonctionne très bien en pratique.

5.4 Exemple Illustratif

$x =$ AACATACA, $m = 8$

a	A	C	G	T
$bc_x[a]$	0	1	8	3

i	0	1	2	3	4	5	6	7
$x[i]$	A	A	C	A	T	A	C	A
$gs_x^0[i]$	7	7	7	7	4	7	2	1
$gs_x^1[i]$	4	4	4	4	2	2	1	1
$gs_x[i]$	4	4	4	4	2	2	1	1

$Y = \{y_0, y_1\}$, $r = 2$, $n = 17$
$y_0 =$ ATGCTAGCAAGATACAG, $y_1 =$ ATGCTAGCAACATACAG
$Z = (\{1\}, 10, \mathtt{C}))$
Première tentative

								↓								
y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
x	A	A	C	A	T	A	C	A									
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

Décalage de longueur 1 donné par $bc_x[y_0[7]] = bc_x[\mathtt{C}]$

Deuxième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
							≠	=	=								
x		A	A	C	A	T	A	C	A								
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

Décalage de longueur 2 donné par $gs_x[5]$

Troisième tentative

											↓					
y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
x				A	A	C	A	T	A	C	A						
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G
											↑					

Décalage de longueur 1 donné par $\min\{bc_x[y_0[10]], bc_x[y_1[10]] = \min\{bc_x[\mathtt{G}], bc_x[\mathtt{C}]\}$

Quatrième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
									≠	=	=	=				
x					A	A	C	A	T	A	C	A					
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

Décalage de longueur 2 donné par $gs_x[4]$

Cinquième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
													≠	=		
x							A	A	C	A	T	A	C	A			
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

Décalage de longueur 2 donné par $gs_x[6]$

Sixième tentative

														↓		
y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
x								A	A	C	A	T	A	C	A		
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

Décalage de longueur 1 donné par $bc_x[y_0[14]] = bc_x[\texttt{C}]$

Septième tentative

y_0	A	T	G	C	T	A	G	C	A	A	G	A	T	A	C	A	G
x									=	=	=	=	=	=	=	=	
x									A	A	C	A	T	A	C	A	
y_1	A	T	G	C	T	A	G	C	A	A	C	A	T	A	C	A	G

Décalage de longueur 4 donné par $gs_x[0]$

5.5 Conclusion

Dans ce chapitre, nous avons proposé un nouvel algorithme, appelé FSE [Nsira *et al.*, 2014a] permettant de traiter le problème de recherche exacte d'un seul motif dans un ensemble de séquences fortement similaires. Cet algorithme étend l'algorithme de recherche rapide FS [Cantone and Faro, 2005] séquentiel de la recherche d'un seul motif dans une seule séquence. En fait, l'algorithme FS permet d'utiliser des variantes de l'algorithme Boyer-Moore [Boyer and Moore, 1977]. De ce fait, notre nouvel algorithme permet d'étendre ces variantes afin de les adapter au problème qui nous intéresse.

Bien que la complexité de l'algorithme FSE soit quadratique en théorique, nous allons montrer dans ce qui suit qu'il permet de réaliser de meilleur résultats par rapport aux deux algorithmes présentés dans les chapitres précédants. En fait, l'algorithme FSE préserve la propriété fondamentale de l'algorithme de Boyer-Moore classique, i.e., il est de plus en plus rapide avec l'augmentation de la longueur du motif. Cependant l'algorithme FSE fonctionne avec un modèle restreint des données en ne considérant que des variations de type substitutions et en se limitant à un certain écart entre deux variations consécutives.

Chapitre 6

Étude Expérimentale

Introduction

Dans ce chapitre, nous effectuons une étude expérimentale sur nos algorithmes, MPE [Nsira *et al.*, 2014b] (voir chapitre 3), KMPE [Nsira *et al.*, 2017] (voir chapitre 4) et FSE [Nsira *et al.*, 2014a], (voir chapitre 5) traitant le problème de recherche exacte d'un seul motif dans un ensemble de séquences fortement similaires.

Dans un premier temps, nous faisons une étude comparative de nos algorithmes avec un des plus efficaces algorithmes séquentiels de recherche incrémentale. Ceci est dans l'objectif de déduire l'apport de nos algorithmes par rapport aux algorithmes classiques. Ainsi, notre objectif est de répondre à la question suivante : à partir de quel nombre de séquences est-il plus avantageux d'adopter nos algorithmes ?

Dans un deuxième temps, nous faisons une étude comparative entre nos algorithmes afin d'évaluer leurs performances. Ensuite, nous avons décidé de comparer notre algorithme le plus performant avec l'unique solution, selon nos connaissances, permettant de combiner l'indexation et la recherche incrémentale dans des séquences fortement similaires.

Les expériences sont munies sur des séquences d'ADN construites à partir de l'alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Rappelons que nos algorithmes sont restreints et ne considèrent que des variations de type substitution dans les séquences et supposent qu'il n'y a pas deux variations dans une même fenêtre. Nous avons expérimenté nos algorithmes sur des données simulées afin d'avoir une analyse proche de la réalité. Nous les avons générées selon un modèle adapté aux restrictions de nos algorithmes. Nous avons effectué des tests sur des séquences biologiques réelles dont nous ne considérons que les substitutions.

Dans notre étude comparative nous avons utilisé les algorithmes suivants :

— Un algorithme de recherche incrémental, nommé FJS [Franek *et al.*, 2007]. Cet algorithme est un des algorithmes séquentiels les plus efficaces et fonctionne bien avec les séquences d'ADN [Faro and Lecroq, 2010]. Ainsi pour l'algorithme FJS nous l'avons exécuté en considérant les séquences fortement similaires en entrée une par une. Puis nous avons calculé la somme totale des temps d'exécution.

— Un algorithme appelé *Extended Naive* qui effectue des comparaisons de gauche à droite à chaque tentative et il est sans mémoire d'une tentative à la suivante. Il

effectue des décalages de longueur 1 après chaque tantative.
— L'algorithme MPE [Nsira *et al.*, 2014b].
— L'algorithme KMPE [Nsira *et al.*, 2017].
— L'algorithme FSE [Nsira *et al.*, 2014a]..
— La structure JST [Rahn *et al.*, 2014].

Notons que tous les algorithmes FJS, *Extended Naive*, MPE, KMPE et FSE ajoutent une copie du motif à la fin de la séquence de référence y_0 comme sentinelle (afin de tester la fin de la séquence uniquement lorsqu'une occurrence du motif est trouvée ce qui permet d'économiser beaucoup de temps sur toutes les tentatives).

Nous avons implémenté les algorithmes de façon à signaler les positions et les numéros des séquences où le motif apparait.

Le reste du chapitre est organisé comme suit :

Dans la section 6.1, nous donnons des résultats obtenus à partir d'une étude comparative réalisée entre nos algorithmes et l'algorithme FJS. Dans la section 6.2, nous étudions le comportement de notre algorithmes FSE sur différentes longueurs de motifs. Dans la section 6.3, nous évaluons les performances de l'algorithme FSE vis-à-vis à celle de la structure JST. Dans la section 6.4, nous comparons les algorithmes que nous adoptons pour précalculer les tableau $pref_x^0$ et $pref_x^1$ (symétriquement $suff_x^0$ et $suff_x^1$) avec d'autres basés sur l'utilisation des RMQ. Dans la section 6.5, nous donnons une conclusion à ce chapitre.

6.1 Résultats de Comparaison avec un Algorithme de Recherche Incrémentale Classique

Afin de mettre en évidence l'utilité des algorithmes que nous proposons vis-à-vis des algorithmes classiques, nous devons répondre à la question suivante : à partir de quel nombre de séquences fortement similaires est-il plus avantageux d'utiliser nos algorithmes plutôt que des algorithmes de recherche incrémentale classiques les plus efficaces ?

6.1.1 Données pseudo-aléatoires

Nous utilisons des séquences simulées de longueur 150 MO dont 30% à 50% des variations sont dans les mêmes positions dans les différentes séquences. D'abord, nous avons généré unse séquence de référence, puis nous l'avons mutée à des positions aléatoires avec des nucléotides alátoires. Nous avons utilisé le générateur WELL [Panneton *et al.*, 2006] sur l'alphabet de l'ADN $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Ensuite, nous avons filtré les variations pour ne considérer que celles distantes de 500 positions.

Pour les motifs, nous les avons séléctionnés aléatoirment à partir de la séquence de référence en faisant varier les longueurs de 8 à 128.

6.1.2 Données réelles

Pour les données réelles, nous avons utilisé comme séquence de référence le chromosome humain 7 à partir de la construction hg19 (Génome Humain version 19). Il a environ

152Mbp.

Pour les variations, nous avons utilisé toutes les SNV (homozygotes et hétérozygotes) de tous les exomes de 5 patients souffrant de la maladie d'Alzheimer précoce à transmission autosomique dominante [Pottier *et al.*, 2012] en mettant au rebut les indels.

6.1.3 Méthodologie

Nous avons implémenté nos algorithmes en langage de programmation C sur une machine de 12 GO de RAM et un CPU 4-core de fréquence $2,27$ GHz.

En pratique, l'ensemble Z est implémenté sous forme de tableau avec 3 lignes (position de la variation j, symbole de la variation c et l'ensemble de séquences $\mathcal{G}$ contenant la variation). Ce tableau est trié selon un ordre croissant des positions des variations.

Pour chaque longueur du motif, nous répétons les tests 100 fois et nous mesurons la moyenne des temps d'exécutions de recherche.

6.1.4 Évaluation en fonction du nombre de séquences

Nous avons mesuré les temps d'exécution de l'algorithme FJS et de nos algorithmes pour différentes valeurs du nombre r, de séquences de l'ensemble Y. Pour chacune de ces valeurs, nous avons fait varier la valeur de longueur du motif m.

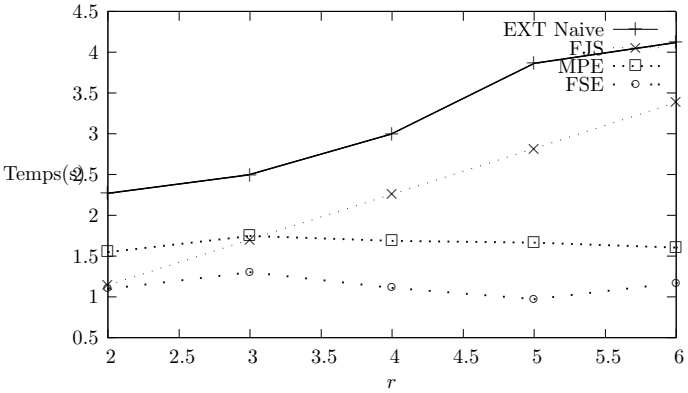
Ainsi, les tests consistent à chercher individuellement 100 motifs pour chaque longueur du motif, dans des ensembles de séquences simulées et réelles. Nous avons effectué nos tests sur des motifs de longueurs 8, 16, 64 et 128 sur des ensembles Y contenant 2 à 6 séquences.

Les figures 6.1, 6.2, 6.3 et 6.4 représentent les résultats sur des motifs de longueurs 8, 16, 64 et 128. L'axe des abscisses représente le nombre de séquences r dans Y tandis que l'axe des ordonnées représente les temps d'exécution. Pour la clarté de la représentation, nous avons choisi d'omettre les résultats obtenus avec l'algorithme KMPE.

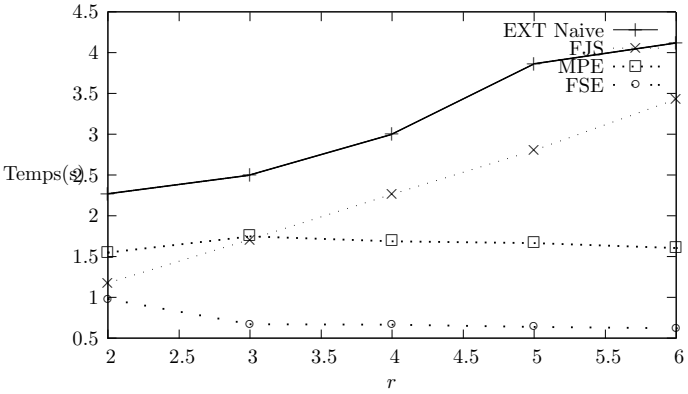
Comme on peut le constater, les algorithmes se comportent de la même façon pour les données pseudo-aléatoires et pour les données réelles. Nous constatons que toutes les courbes ont la forme d'une droite. En observant la performance globale de tous les algorithmes, nous pouvons déduire que l'augmentation du nombre de séquences r fait augmenter les temps d'exécution notamment pour les algorithmes FJS, *Extended Naive* et MPE.

Les courbes montrent que l'avantage de l'algorithme MPE par rapport à l'algorithme FJS classique n'est visible qu'à partir d'un ensemble Y de séquences fortement similaires dont $r > 3$. Nous pouvons expliquer ceci par les deux raisons suivantes :

1. Quand $r < 3$, l'algorithme MPE se comporte comme l'algorithme MP classique. Ceci est du d'une part à la grande similarité entre les séquences de Y, et d'autre part est du à l'hypothèse de l'écart entre deux variations consécutives dans les séquences (Rappelons que les variations voisines sont distantes d'au moins 500, ceci implique qu'elles sont réparties avec une probabilité inférieure ou égale à 50%). Par ce fait, la phase de recherche effectue plus des décalage avec des bords à distance de Hamming



(a) Résultats expérimentaux avec 100 motifs de longueur 8 sur des séquences pseudo-aléatoires.



(b) Résultats expérimentaux avec 100 motifs de longueur 8 sur des séquences réelles.

FIGURE 6.1 – Résultats expérimentaux avec 100 motifs de longueur 8.

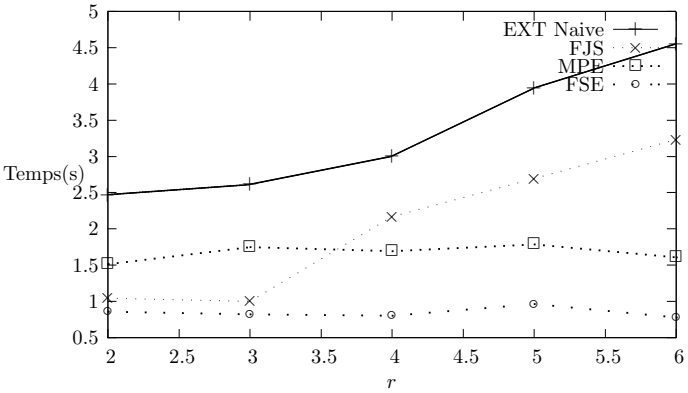
0, i.e., de type MP, plus que des décalages à distance de Hamming 1, i.e., de type MPE.

2. Le comportement hybride de l'algorithme FJS lui permettant de combiner les caractéristiques des algorithmes de Knuth-Morris-Pratt (KMP) et de Boyer-Moore (BM). En fait, en adoptant l'algorithme FJS, on opère comme suit :

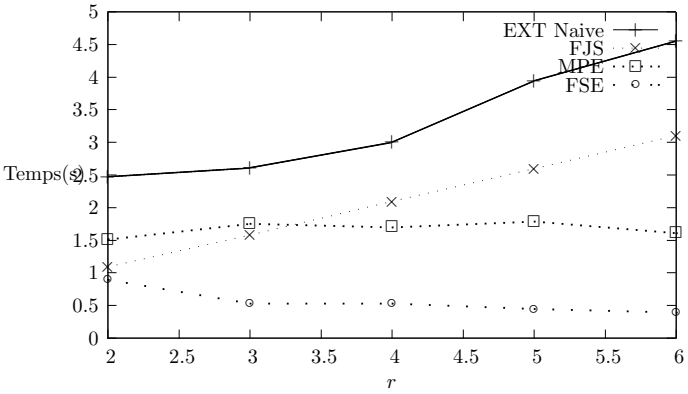
   (*i*) D'abord, conformément à l'algorithme BM, on commence par comparer les caractères de droite à gauche. Tant qu'on trouve une inégalité, on décale la fenêtre en utilisant les variantes de l'algorithme BM.

   (*ii*) Ensuite, si on trouve une égalité alors l'algorithme se comporte comme l'algorithme KMP. Ainsi, on compare les caractères de gauche à droite. Si on ne trouve pas d'inégalité jusqu'au dernier caractère du motif alors on décale la fenêtre par un décalage de type KMP. On revient ensuite à la première étape.

   Puisqu'on pratique les algorithmes KMP et BM sont avantageux par rapport à l'algo-



(a) Résultats expérimentaux avec 100 motifs de longueur 16 sur des séquences pseudo-aléatoires.



(b) Résultats expérimentaux avec 100 motifs de longueur 16 sur des séquences réelles.

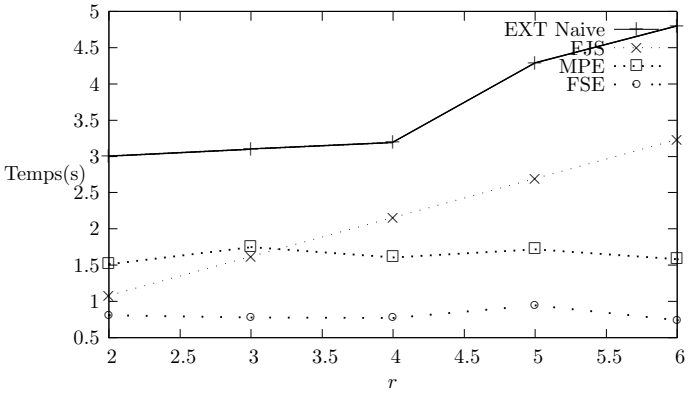
FIGURE 6.2 – Résultats expérimentaux avec 100 motifs de longueur 16.

rithme MP [Faro *et al.*, 2016] alors ceci assure que l'algorithme FJS est plus efficace que l'algorithme MPE lorsqu'il s'agit d'un ensemble Y de r séquences dont $r < 3$.

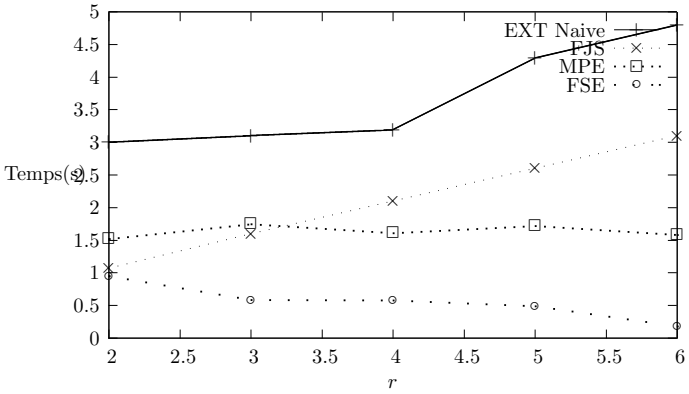
Plus que le nombre de séquences augmente ($r \geq 3$) plus que l'algorithme MPE calcule des décalages avec des bords à distance de Hamming 1. De fait que ces bords sont plus longs que les bords de type MP, le nombre de comparaisons est réduit et par conséquence le temps d'exécution est amélioré.

Nous observons que l'augmentation de r (plus que 3) induit une petite baisse dans la courbe de l'algorithme MPE. Puis nous pouvons remarquer une stagnation de la courbe (quasi constante). Nous pouvons expliquer cette allure comme suit : le nombre de variations est presque négligeable par rapport à la longueur des séquences (à cause de la restriction de notre hypothèse) ce qui fait que le temps d'exécution dépend essentiellement de la longueur de la séquence de référence.

En observant toutes les performances, nous pouvons déduire que l'algorithme FSE réalise de bons résultats. Les résultats montrent qu'il est le plus rapide avec les 4 longueurs



(a) Résultats expérimentaux avec 100 motifs de longueur 64 sur des séquences pseudo-aléatoires.



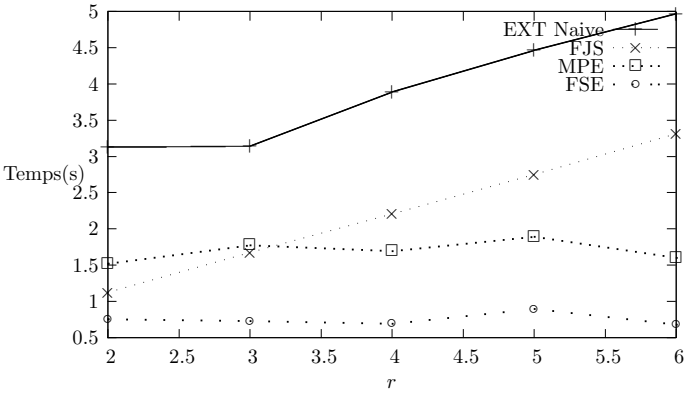
(b) Résultats expérimentaux avec 100 motifs de longueur 64 sur des séquences pseudo-aléatoires.

FIGURE 6.3 – Résultats expérimentaux avec 100 motifs de longueur 64.

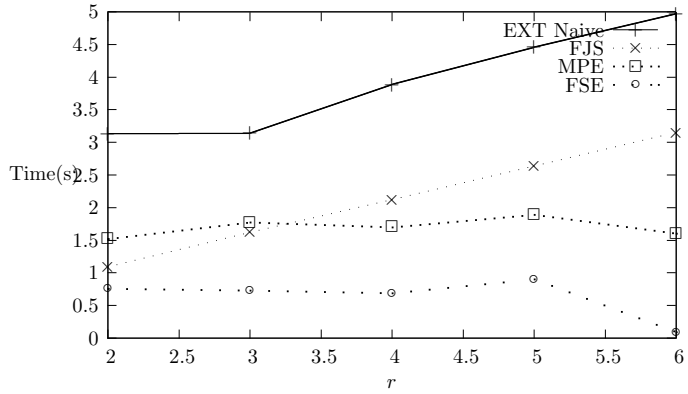
des motifs et même pour un nombre de séquence $r < 2$. Ce résultat est naturel. En effet, l'optimalité de l'algorithme rapide classique à été prouvée expérimentalement par [Faro *et al.*, 2016] pour différents jeux de données dont notamment les séquences d'ADN. Par extension, l'algorithme FSE préserve les caractéristiques de l'algorithme rapide séquentiel (la vérification du caractère le plus à droite dans toutes les séquences ne fait pas ralentir la boucle rapide).

6.2 Évaluation de l'algorithme FSE en fonction des longueurs des motifs

Nous avons décidé d'étudier le comportement de l'algorithme FSE lorsque l'on fait augmenter les longueurs des motifs. L'ensemble de séquences pseudo-aléatoires que nous avons utilisé est toujours le même. Nous avons fait varier les longueurs des motifs pour



(a) Résultats expérimentaux avec 100 motifs de longueur 128 sur des séquences pseudo-aléatoires.



(b) Résultats expérimentaux avec 100 motifs de longueur 128 sur des séquences pseudo-aléatoires.

FIGURE 6.4 – Résultats expérimentaux avec 100 motifs de longueur 128.

des valeurs de $m \in \{8, 16, 64, 128\}$ et nous avons relevé les temps d'exécution pour chaque longueur.

La figure 6.5 représente les temps d'exécution pour les différentes valeurs de m et avec un ensemble Y contenant 6 séquences fortement similaires . Nous pouvons constater que la courbe décroit en faisant augmenter la valeur de m. Nous déduisons que par extension, l'algorithme FSE préserve les avantages de l'algorithme FS classique. Des études expérimentales [Faro and Lecroq, 2010] ont montré que pour des motifs suffisamment longs, l'algorithme FS classique devient de plus en plus rapide (les longueurs des décalages augmentent). Nous avons alors les mêmes constations. De plus, le fait même d'utiliser des suffixes à distance de Hamming 1, qui peuvent ramener des décalages plus courts, l'algorithme reste rapide.

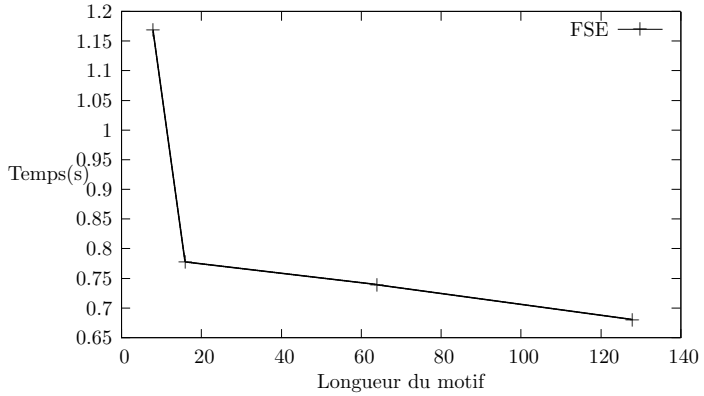


FIGURE 6.5 – Résultats éxpérimentaux de l'algorithme FSE avec 6 séquences pseudo-aléatoires.

6.3 Résultats de Comparaison avec un Algorithme de Recherche Incrémentale dédiées aux Séquences Fortement Similaires

Nous avons comparé l'algorithme FSE avec la structure JST [Rahn *et al.*, 2014] (voir la section).

6.3.1 Méthodologie et jeu de données

Pour évaluer leur structure, René *et al.* [Rahn *et al.*, 2014] ont utilisé des données en provenance du projet *1000 Genomes Project* (*1000 Genomes Project Consortium, 2012)*. Ils ont utilisé le chromosome humain 1 à partir de la construction hg19 au format fasta. Il a environ 250 Mbp. Pour les variations, ils ont récupéré le fichier en format vcf du chromosome 1 [1]. Ils ont mis en place un outil pour générer un nouveau format, appelé *gdf* à partir du format vcf. Le format gdf permet tout simplement de stocker les variations sous une forme plus compacte que le format vcf [Deorowicz and Grabowski, 2011]). Ainsi, il est devenu possible de stocker 1092 génotypes (sans la référence) en utilisant uniquement 446 MO au lieu de 87 GO requis pour le format vcf [Rahn *et al.*, 2014].

Pour la recherche exacte de motif avec la structure JST, René *et al.* [Rahn *et al.*, 2014] ont choisi de fournir des foncteurs pour des algorithmes de recherche incrémentale tel que : l'algorithme *naif*, l'algorithme *Horspool* [Horspool, 1980] etc. Les tests ont été effectués sur différents ensembles de séquences représentant des génotypes de 1, 32, 64, 128, 256, 512 et 1092. Nous avons choisi d'utiliser le même jeu de données. Ainsi, nous avons muni des tests sur 100 motifs en faisant varier la longueur des motifs m. Les valeurs considérées sont 32, 64, 128, 256. Nous avons comparé notre algorithme FSE avec le foncteur de l'algorithme Horspool de JST. Les résultats obtenus sont révélés dans les tables 6.1, 6.2, 6.3, 6.4.

1. Des détails sur le format vcf sont donnés via http ://samtools.github.io/hts-specs/VCFv4.2.pdf

Nombre de séquences	Temps total	
	FSE	JST
1	1.10	1.94
2	1.30	2.78
33	1.10	3.58
65	0.97	4.73
129	1.16	6.10
257	1.29	23.55
513	1.32	41.72
1093	1.38	87.45

TABLE 6.1 – Résultats obtenus avec 100 motifs de longueur $m = 32$

Nombre de séquences	Temps total	
	FSE	JST
1	0.85	1.79
2	0.82	2.65
33	0.80	3.44
65	0.96	4.41
129	0.77	5.99
257	0.84	24.63
513	0.97	50.24
1093	1.12	101.18

TABLE 6.2 – Résultats obtenus avec 100 motifs de longueur $m = 64$

Nombre de séquences	Temps total	
	FSE	JST
1	0.66	1.79
2	1.10	2.65
33	0.65	3.51
65	0.52	4.47
129	0.69	6.08
257	0.78	25.89
513	0.84	51.25
1093	0.92	103.46

TABLE 6.3 – Résultats obtenus avec 100 motifs de longueur $m = 128$

Comme le montrent les valeurs dans les tables ci-dessus pour le jeu de données utilisé, l'algorithme FSE est le plus rapide en pratique.

Nombre de séquences	Temps total	
	FSE	JST
1	0.53	1.92
2	0.97	2.34
33	0.48	3.83
65	0.44	4.79
129	0.58	6.62
257	0.66	26.75
513	0.71	52.57
1093	0.88	104.95

TABLE 6.4 – Résultats obtenus avec 100 motifs de longueur $m = 265$

6.4 Résultats de Comparaison entre des Méthodes De Calcul de Tableax De Préfixes

Rappelons que nos algorithmes se basent essentiellement sur l'utilisation des tableaux élémentaires précalculés pendant une phase essentielle de prétraitement qui fournit les informations nécessaires aux algorithmes de recherche pour calculer des décalages valides. Comme mentionné auparavant, la phase de prétraitemment s'appuient sur l'utilisation des tableaux des plus long préfixes communs entre un motif et ses facteurs : $pref_x^0$, $pref_x^1$ (de manière symétrique $suff_x^0$ et $suff_x^1$).

Il nous a alors semblé incontournable de se poser la question principale : existe-t-il une méthode alternative permettant de calculer les tableaux $pref_x^0$, $pref_x^1$ facilement et rapidement tout en garantissant un meilleur compromis espace/temps ?

Compte tenue, de la nature fondamentale du problème du RMQ, (voir la section 1.5) lui permettant de surgir dans des solutions de plusieurs problèmes du domaine de l'Algorithmique du Texte, par le biais de requêtes, nous avons décidé de l'adapter pour construire les tableaux $pref_x^0$ et $pref_x^1$ et de faire une étude comparative avec les algorithmes que nous utilisons (voir la section 4.2.2).

Construction des tableaux $pref_x^0$ et $pref_x^1$ en utilisant les RMQ

Nous allons détailler dans cette partie, la procédure de calcul de $pref_x^0$ et $pref_x^1$ par le biais des reqêtes de type RMQ (voir la section 1.5).

Supposons que les tableaux des plus longs préfixes communs, de suffixes et de suffixes renversé respectivement LCP_x, SA_x et ISA_x d'une séquence y aient été construits au préalable en temps linéaire. La procédure de construction est plus limpide grâce au lemme suivant [Ohlebusch, 2013] :

Lemme 12 *Soient y une séquence de longueur n et LCP_y le tableau des préfixes communs associé. Pour tout i, $0 \leq i \leq j \leq n-1$, on a :*

$$\ell = |lcp(y[i\,..\,n{-}1], y[j\,..\,n{-}1])| = \begin{cases} |y[i\,..\,n-1]| & si\ i = j, \\ LCP_x[rmq_{LCP_x}(ISA[i]+1, ISA_x[j])] & si\ i \neq j\ et\ ISA_x[i] < ISA_x[j], \\ LCP_x[rmq_{LCP_x}(ISA[j]+1, ISA_x[i])] & si\ \ i \neq j\ et\ ISA_x[j] < ISA_x[i]. \end{cases}$$

L'adaptation pour calculer $pref_x^0$ et $pref_x^1$ en utilisant les RMQ est assurée par un algorithme appelé *PrefixesRmq*. En adoptant cet algorithme, on opère comme suit :

D'abord, pour tout i, $1 \leq i \leq m-1$, on utilise deux variables *left* et *right* initialisées avec respectivement la valeur minimale et la valeur maximale entre $ISA_x[0], ISA_x[i]$. On mémorise dans $pref_x^0[i]$ la valeur $LCP_x[rmq_{LCPx}(left{+}1, right)]$. Ensuite, on refait la même procédure pour les indices $pref_x^0[i]{+}1$ et $i{+}pref[i]{+}1$ afin de calculer le préfixe avec erreur, s'il en existe, entre les positions 0 et i. Si un tel préfixe existe alors nous mémorisons sa valeur dans $pref_x^1[i]$. Ceci donne le pseudo code suivant, où en entrée on reçoit x, LCP_x, SA_x et ISA_x et en sortie on renvoie les tableaux $pref_x^0$ et $pref_x^1$:

Algorithme 6.1 : *PrefixesRmq*

Données : x, LCP_x, SA_x, ISA_x
Résultats : $pref_x^0, pref_x^1$
Auxiliaires : $left, right, \ell$
$left = 0$
$pref_x^1[0] = 0$
pour i de 1 à $m-1$ **faire**
    $left = \min(ISA_x[0], ISA_x[i])$
    $right = \max(ISA_x[0], ISA_x[i])$
    $pref_x^0[i] = LCP_x[rmq_{LCP_x}(left+1, right)]$
    $\ell = rmq_{LCP_x}(pref_x^0[i]+1, ISA_x[i+pref[i]+1])$
    $left = \min(pref_x^0[i]+1, ISA_x[i+pref[i]+1])$
    $right = \max(pref_x^0[i]+1, ISA_x[i+pref[i]+1])$
    $\ell = LCP_x[rmq_{LCP_x}(left+1, right)]$
    si $\ell \neq 0$ **alors**
        $pref_x^1[i] = pref_x^0[i] + 1 + \ell$

fin pour

Implantation

Plusieurs solutions ont été proposées pour résoudre le problème des RMQ. Nous avons choisi d'utilisé la solution proposée par Johannes *et al.* [Fischer and Heun, 2007].

Nous avons utilisé une implémentation donnée par une librairie C++, disponible en ligne sous le lien https ://github.com/simongog/sdsl-lite.

Comparaison expérimentale

Nous avons écrit un programme qui extrait aléatoirement 500 motifs à partir d'une séquence de réference en faisant varier la valeur de m de 2 à 2^{15}. Pour chaque motif nous avons calculé les tableaux $pref_x^0$ et $pref_x^1$ en utilisant l'algorithme *PrefixesBis* et l'algorithme *PrefixesRmq* Ensuite, nous avons mesuré les temps d'exécution.

Dans la figure 6.6 nous présentons les résultats dans une courbe tracée pour chaque longueur de motif m (en abscisse) contre les moyennes des côuts en temps (en ordonnée).

Pour des petites valeurs de m, nous pouvons remarquer que les temps d'exécution obtenus avec les deux algorithmes sont quasi équivalents. Néanmoins, il est nettement visible que les temps d'exécution pour l'algorithme *PrefixesRmq* ont tendance à augmenter avec l'augmentation de la valeur de m. En revanche, le comportement de l'algorithme *PrefixesBis* reste le même avec l'augmentation de la valeur de m. Ostensiblement, l'algorithme *PrefixesBis* nous offre un temps d'exécution relativement rapide et stable.

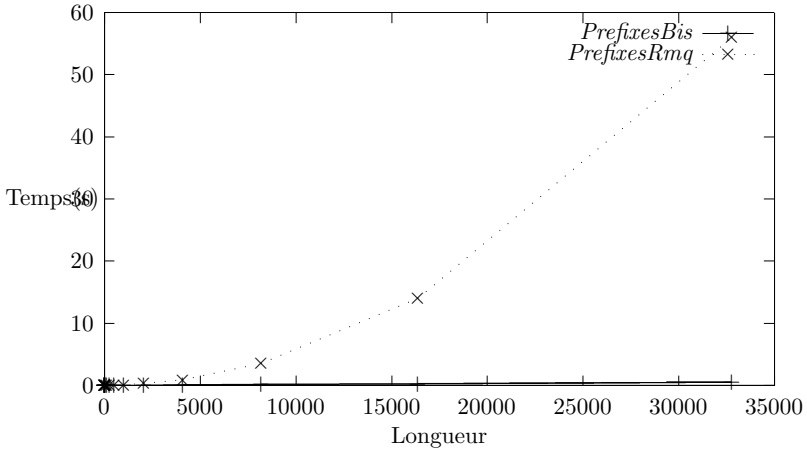


FIGURE 6.6

6.5 Conclusion

Dans ce chapitre, nous avons fait une étude expérimentale des algorithmes MPE, KMPE et FSE sur des données synthétiques et sur des données biologiques réelles. Nous avons effectué une étude comparative de ces trois algorithmes avec l'algorithme FJS [Franek *et al.*, 2007]. Ceci nous permet d'évaluer l'avantage de nos algorithmes par rapport aux algorithmes classiques de recherche incrémentale de motif les plus efficaces. Une étude comparative de nos algorithmes entre eux nous permet d'évaluer leurs performances au niveau de la consommation en temps de calcul et occupation de l'espace mémoire. Nous avons aussi comparé notre algorithme le plus rapide en pratique, FSE, avec la structure JST [Rahn *et al.*, 2014]. Rappelons que cette dernière, s'intéresse au même problème que nous. Les résultats obtenus montrent que l'algorithme FSE est plus efficace en pratique.

Cependant, les algorithmes MPE, KMPE, FSE ne fonctionnent que sous l'hypothèse que les variations sont de type substitution et que deux variations voisines sont distantes par un certain écart. Ainsi, nous avons simulé des données synthétiques et nous avons adapter les données réelles que nous avons utilisé afin de répondre aux restrictions de nos algorithmes.

En bref, les résultats obtenus, montrent que nos algorithmes de recherche incrémentale d'un motif au sein d'un ensemble de séquences fortement similaires, en utilisant un modèle de données particulier répondant à nos restrictions, sont les plus efficaces en pratique.

Pourvu que la phase de prétraitement est une phase primordiale, nous avons réalisé une étude comparative des algorithmes que nous adoptons pour la construction des tableaux $pref_x^0$ et $pref_x^1$ (de manière symétrique $suff_x^0$ et $suff_x^1$) avec d'autres basés sur l'utilisation des RMQ (voir la section 1.5). Les résultats trouvés montrent, que les algorithmes que nous utilisons sont plus efficaces.

Conclusion Générale

Dans cette thèse, nous nous sommes intéressés au problème de recherche incrémentale de motifs dans un ensemble de séquences d'ADN fortement similaires. Dans ce cadre, nous avons développé trois algorithmes [Nsira *et al.*, 2014b; Nsira *et al.*, 2017; Nsira *et al.*, 2014a]

(*i*) Le premier algorithme développé est l'algorithme MPE [Nsira *et al.*, 2014b]. Ce dernier est une extension de l'algorithme de Morris-Pratt classique [Morris and Pratt, 1970] à la recherche incrémentale d'un motif dans un ensemble de séquences fortement similaires. En fait, l'algorithme MPE fait appel à la notion de bords, précalculés pendant une phase de prétraitement, afin de réaliser des décalages valides, i.e., sans risque de perte d'occurrences, après chaque tentative de localisation du motif dans les séquences d'entrée. Ainsi, l'algorithme MPE utilise des bords à distance de Hamming 0 de type MP et de nouveaux bords à distance de Hamming 1. Cet algorithme est de complexités $O(n)$ en temps de calcul.

(*ii*) Le second algorithme développé est l'algorithme KMPE [Nsira *et al.*, 2017]. Cet algorithme est une extension de l'algorithme de Knuth-Morris-Pratt classique à la recherche incrémentale d'un motif dans un ensemble de séquences fortement similaires. Cet algorithme permet d'optimiser les décalages effectués par l'algorithme MPE afin d'éviter de réaliser des tentatives de recherche inutiles (propriété fondamentale de KMP par rapport à MP). Il est de complexité $O(n)$.

(*iii*) Enfin, le troisième algorithme que nous avons développé est l'algorithme FSE [Nsira *et al.*, 2014a]. Cet algorithme est une extension de l'algorithme rapide séquentiel à la recherche incrémentale d'un motif dans un ensemble de séquences fortement similaires. L'algorithme FSE est de complexité $O(mnr)$. Bien que sa complexité théorique soit quadratique, il est le plus efficace en pratique.

Nos trois algorithmes sont restreints et fonctionnent avec un modèle spécifique de données où on ne considère que des variation de type substitution dans les séquences d'entrée et où on suppose qu'une fenêtre de longueur égale à celle du motif ne contient qu'une seule variation. En effet, nos algorithmes jouent le rôle d'une étude préliminaire permettant d'ouvrir la porte sur le problème de recherche incrémentale de motifs dans un ensemble de séquences fortement similaires.

D'après l'étude expérimentale que nous avons faite sur des jeux de données synthétiques et sur des jeux de données réelles, on peut conclure qu'en pratique nos algorithmes sont efficaces. L'algorithme FSE détient la pire complexité en temps de calcul mais il est le plus rapide en pratique.

Comme perspectives à notre travail de thèse, on peut :
— Améliorer nos algorithmes afin de tenir en compte n'importe quel type de variation

entre une séquence de référence et les autres séquences (non seulement des substitutions) ;

— Être capable d'effectuer la recherche de motifs approximative ;

— Considérer la recherche de motifs dans des séquences compressées. D'un point de vue pratique, ce travail constitue une étape importante afin de permettre d'effectuer la recherche de motifs en utilisant le format CRAM [Fritz *et al.*, 2011].

Liste des publications

Revues internationales avec comité de lecture

1. N. Ben Nsira, T. Lecroq, and M. Elloumi, On-line String Matching in Highly Similar DNA Sequences. *Mathematics in Computer Science*, 11(2) : 113-126, 2017.
2. N. Ben Nsira, T. Lecroq, and M. Elloumi, A fast Boyer-Moore type pattern matching algorithm for highly similar sequences. *IJDMB* 13(3) : 266-288, 2015.

Articles dans des conférences internationales avec comité de lecture

1. N. Ben Nsira, T. Lecroq, and M. Elloumi, A fast pattern matching algorithm for highly similar sequences. *BIBM* 2014 : 32-38.
2. N. Ben Nsira, T. Lecroq, and M. Elloumi, Single Pattern matching in highly similar sequences. *The First Computer Science University of Tunis El Manar PhD Symposium*, to appear.
3. N. Ben Nsira, T. Lecroq, and M. Elloumi, On-line String Matching in Highly Similar DNA Sequences. *ICABD* 2014 : 16-22.

Chapitres dans un livre

1. N. Ben Nsira, T. Lecroq, and M. Elloumi. *Algorithms for Next-Generation Sequencing Data : Techniques, Approaches, and Applications*, chapter Algorithms for Indexing Highly Similar DNA Sequences, Springer International Publishing, Cham, pages 3-39, 2017.

Références

[Aho and Corasick, 1975] cité page 32
A. V. Aho and M. J. Corasick. Efficient string matching : An aid to bibliographic search. *Communications of the ACM*, 18(6) :333–340, 1975.

[Alatabbi *et al.*, 2012] cité page 28, 31, 31, 31, 32, 32, 32, 33
A. Alatabbi, C. Barton, C. S. Iliopoulos, and L. Mouchard. Querying highly similar structured sequences via binary encoding and word level operations. In Lazaros S. Iliadis, Ilias Maglogiannis, Harris Papadopoulos, Kostas Karatzas, and Spyros Sioutas, editors, *Proceedings of the International Workshop on Artificial Intelligence Applications and Innovations, AIAI 2012, Part II*, volume 382 of *IFIP Advances in Information and Communication Technology*, pages 584–592. Springer, 2012.

[Amihood *et al.*, 2004] cité page 12
A. Amihood, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2) :257–275, 2004.

[Aoe *et al.*, 1992] cité page 16
J. Aoe, K. Morimoto, and T. Sato. An efficient implementation of trie structures. *Software : Practice and Experience*, 22(9) :695–721, 1992.

[Arroyuelo *et al.*, 2006] cité page 27, 27, 27, 27
D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In Moshe Lewenstein and Gabriel Valiente, editors, *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching, CPM 2006*, volume 4009 of *Lecture Notes in Computer Science*, pages 318–329, Barcelona, Spain, 2006. Springer.

[Bender and Farach-Colton, 2000] cité page 9
M. A. Bender and M. Farach-Colton. The lca problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94. Springer-Verlag, 2000.

[Bender *et al.*, 2005] cité page 9
M. A. Bender, M. Farach-colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57 :2005, 2005.

[Blumer *et al.*, 1985] cité page 18, 18
A. Blumer, J.Blumer, D.Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automation recognizing the subwords of a text. *Theoretical Computer Science*, 40 :31–55, 1985.

[Blumer *et al.*, 1987] cité page 19, 20, 20
A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3) :578–595, 1987.

[Boyer and Moore, 1977] cité page 1, 77, 77, 78, 88
R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10) :762–772, 1977.

[Breslauer and Galil, 2011] cité page 7, 7
D. Breslauer and Z. Galil. Real-time streaming string-matching. In *Annual Symposium on Combinatorial Pattern Matching*, pages 162–172. Springer, 2011.

[Breslauer *et al.*, 2011] cité page 7, 7
D. Breslauer, R. Grossi, and F. Mignosi. Simple real-time constant-space string matching. In *Annual Symposium on Combinatorial Pattern Matching*, pages 173–183. Springer, 2011.

[Burrows and Wheeler, 1994] cité page 22, 23, 23, 24
M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital SRC Research, 1994.

[Cantone and Faro, 2005] cité page 77, 88
D. Cantone and S. Faro. Fast-search algorithms : New efficient variants of the boyer-moore pattern-matching algorithm. *Journal of Automata, Languages and Combinatorics*, 10(5-6) :589–608, 2005.

[Chan *et al.*, 2011] cité page 36
T. M . Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the ram, revisited. In *Proceedings of the twenty-seventh annual symposium on Computational geometry*, pages 1–10. ACM, 2011.

[Charras and Lecroq, 2004a] cité page 6, 6, 7, 7
C. Charras and T. Lecroq. *Handbook of exact string matching algorithms*. Citeseer, 2004.

[Charras and Lecroq, 2004b] cité page 82, 83
C. Charras and T. Lecroq. *Handbook of exact string matching algorithms*. King's College Publications, 2004.

[Consortium and others, 2010] cité page 1
1000 Genomes Project Consortium et al. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319) :1061–1073, 2010.

[Crawford *et al.*, 1998] cité page 8
T. Crawford, C. S. Iliopoulos, and R. Raman. String-Matching Techniques for Musical Similarity and Melodic Recognition. *Computing in Musicology*, 11 :71–100, 1998.

[Crochemore and Lecroq, 2009] cité page 14, 14, 14, 15, 15, 16
M. Crochemore and T. Lecroq. Trie. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 3179–3182. Springer Verlag, 2009.

[Crochemore and Rytter, 1994] cité page 5
M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[Crochemore and Rytter, 2003] cité page 6, 6, 7, 7, 7, 7
M. Crochemore and W. Rytter. *Jewels of stringology : text algorithms*. World Scientific, 2003.

[Crochemore and Vérin, 1997] cité page 19, 19, 19
M. Crochemore and R. Vérin. On compact directed acyclic word graphs. In Jan Mycielski, Grzegorz Rozenberg, and Arto Salomaa, editors, *Structures in Logic and Computer Science, A Selection of Essays in Honor of Andrzej Ehrenfeucht*, volume 1261 of *Lecture Notes in Computer Science*, pages 192–211. Springer, 1997.

[Crochemore *et al.*, 2007] cité page 5, 5, 6, 6, 6, 7, 16, 47, 47, 48, 49, 50, 67
M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

[Deorowicz and Grabowski, 2011] cité page 96
Sebastian Deorowicz and Szymon Grabowski. Robust relative compression of genomes with random access. *Bioinformatics*, 27(21) :2979–2986, 2011.

[Do *et al.*, 2012] cité page 33, 34, 34, 35, 35, 35, 36, 36, 36, 37, 37
H. H. Do, J. Jansson, K. Sadakane, and W. K. Sung. Fast relative Lempel-Ziv self-index for similar sequences. In Jack Snoeyink, Pinyan Lu, Kaile Su, and Lusheng Wang, editors, *Proceedings of the Joint International Conference on Frontiers in Algorithmics and Algorithmic Aspects in Information and Management, FAW-AAIM 2012*, volume 7285 of *Lecture Notes in Computer Science*, pages 291–302, Beijing, China, 2012. Springer.

[Dundas, 1991] cité page 16
J. A. Dundas. Implementing dynamic minimal-prefix tries. *Software : Practice and Experience*, 21(10) :1027–1040, 1991.

[Ervin, 1998] cité page 14
K. D. Ervin. *The art of computer programming : sorting and searching*, volume 3. Pearson Education, 1998.

[Farach, 1997] cité page 17
M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science, FOCS 1997*, pages 137–143, Miami Beach, FL, USA, 1997.

[Faro and Lecroq, 2010] cité page 7, 89, 95
S. Faro and T. Lecroq. The exact string matching problem : A comprehensive experimental evaluation (2010). *CoRR abs/1012.2547*, 2010.

[Faro and Lecroq, 2012] cité page 7
S. Faro and T. Lecroq. A fast suffix automata based algorithm for exact online string matching. In *International Conference on Implementation and Application of Automata*, pages 149–158. Springer, 2012.

[Faro and Lecroq, 2013] cité page 6
S. Faro and T. Lecroq. The exact online string matching problem : A review of the most recent results. *ACM Computing Surveys (CSUR)*, 45(2) :13, 2013.

[Faro *et al.*, 2016] cité page 7, 7, 92, 93
S. Faro, T. Lecroq, S. Borzı, S. Di Mauro, and A. Maggio. The string matching algorithms research tool. In *Prague Stringology Conference 2016*, page 99, 2016.

[Faro, 2016] cité page 7
S. Faro. Exact online string matching bibliography. *arXiv preprint arXiv :1605.05067*, 2016.

[Ferragina and Manzini, 2000] cité page 22, 22, 22, 23, 24
P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000*, pages 390–398, Redondo Beach, CA, USA, 2000.

[Ferragina and Manzini, 2001] cité page 25
P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of the 12th annual ACM-SIAM Symposium On Discrete Algorithms, SODA 2001*, pages 269–278, Washington, D.C., USA, 2001. Society for Industrial and Applied Mathematics.

[Ferragina and Manzini, 2005] cité page 22, 22, 22, 22, 23, 23, 27
P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4) :552–581, 2005.

[Ferragina *et al.*, 2004] cité page 25
P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly fm-index. In Alberto Apostolico and Massimo Melucci, editors, *Proceedings of the 11th International Conference on String Processing and Information Retrieval, SPIRE 2004*, volume 3246 of *Lecture Notes in Computer Science*, pages 150–160, Padova, Italy, 2004. Springer.

[Ferragina *et al.*, 2007] cité page 25
P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007.

[Ferragina *et al.*, 2009] cité page 25
P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes : From theory to practice. *Journal of Experimental Algorithmics (JEA)*, 13 :12, 2009.

[Firdose and Nilay, 2014] cité page 6
A. G. Firdose and K. Nilay. Hardware based string matching algorithms : A survey. *International Journal of Computer Applications*, 88(11), 2014.

[Fischer and Heun, 2006] cité page 9
J. Fischer and V. Heun. *Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE*, pages 36–48. Springer Berlin Heidelberg, 2006.

[Fischer and Heun, 2007] cité page 35, 99
Johannes Fischer and Volker Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 459–470. Springer Berlin Heidelberg, 2007.

[Franek *et al.*, 2007] cité page 89, 100
Frantisek Franek, Christopher G Jennings, and William F Smyth. A simple fast hybrid pattern-matching algorithm. *Journal of Discrete Algorithms*, 5(4) :682–695, 2007.

[Fritz *et al.*, 2011] cité page 104
M.H.-Y. Fritz, R. Leinonen, G. Cochrane, and E. Birney. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Research*, 21 :734–740, 2011.

[Grossi and Vitter, 2000] cité page 25, 25, 25, 25, 25, 26
R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 397–406, New York, NY, USA, 2000.

[Grossi and Vitter, 2005] cité page 25
R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2) :378–407, 2005.

[Grossi *et al.*, 2003] cité page 22, 25, 25
R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th annual ACM-SIAM Symposium On Discrete Algorithms, SODA 2003*, pages 841–850, Baltimore, MD, USA, 2003.

[Grossi *et al.*, 2004] cité page 25
R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression : Experiments with compressing suffix arrays and applications. In *Proceedings of the 15th annual ACM-SIAM Symposium On Discrete Algorithms, SODA 2004*, pages 636–645, New Orleans, LA, USA, 2004. Society for Industrial and Applied Mathematics.

[Holub and Crochemore, 2003] cité page 20
J. Holub and M. Crochemore. On the implementation of compact DAWG's. In Jean-Marc Champarnaud and Denis Maurel, editors, *Proceedings of the 7th International Conference on Implementation and Application of Automata, CIAA 2002, Revised Papers*, volume 2608 of *Lecture Notes in Computer Science*, pages 289–294, Tours, France, 2003. Springer.

[Holub *et al.*, 2001] cité page 8, 8
J. Holub, C. Iliopoulos, B. Melichar, and L. Mouchard. Distributed pattern matching using finite automata. *JALC*, 6 :191–204, 01 2001.

[Horspool, 1980] cité page 96
R.Nigel Horspool. Practical fast searching in strings. 10 :501–506, 06 1980.

[Huang *et al.*, 2010] cité page 28, 28, 28, 29, 29, 29, 29, 29, 30, 31, 31, 31, 31
S. Huang, T. W. Lam, W. K. Sung, S. L. Tam, and S. M. Yiu. Indexing similar DNA sequences. In Bo Chen, editor, *Proceedings of the 6th International Conference on Algorithmic Aspects in Information and Management, AAIM 2010*, volume 6124 of *Lecture Notes in Computer Science*, pages 180–190. Springer, Weihai, China, 2010.

[Inenaga *et al.*, 2006] cité page 19, 19, 19
S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. In Moshe Lewenstein and Gabriel Valiente, editors, *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching, CPM 2006*, volume 4009 of *Lecture Notes in Computer Science*, pages 169–180, Barcelona, Spain, 2006. Springer.

[Itoh and Tanaka, 1999] cité page 21
H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix arrays. In *Proceedings of String Processing and Information Retrieval Symposium, 1999 and International Workshop on Groupware*, pages 81–88, 1999.

[Kalsi *et al.*, 2008] cité page 6
P. Kalsi, H. Peltola, and J. Tarhio. Comparison of exact string matching algorithms for biological sequences. In *Bioinformatics Research and Development*, pages 417–426. Springer, 2008.

[Kärkkäinen and Sanders, 2003] cité page 20, 20, 20, 21
J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming, ICALP 2003*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955, Eindhoven, The Netherlands, 2003. Springer.

[Kärkkäinen and Ukkonen, 1996] cité page 27, 27
J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In N. Ziviani, R. Baeza-Yates, and K. Guimar aes, editors, *In Proceedings of the 3rd South American Workshop on String Processing, WSP 1996*, International Informatics, pages 141–155. Carleton University Press, 1996.

[Kim *et al.*, 2003] cité page 20, 20, 20, 21
D. K. Kim, J. S. Sim nd H. Park, and K. Park. Linear-time construction of suffix arrays. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, CPM 2003*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199, Morelia, Michocán, Mexico, 2003. Springer.

[Knuth *et al.*, 1977] cité page 1, 62, 63, 63
D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2) :323–350, 1977.

[Ko and Aluru, 2003] cité page 20, 20, 20, 21
P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, CPM 2003*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210, Morelia, Michocán, Mexico, 2003. Springer.

[Kurtz, 1999] cité page 17
S. Kurtz. Reducing the space requirement of suffix trees. *Software-Practice and Experience*, 29(13) :1149–71, 1999.

[Kuruppu *et al.*, 2010] cité page 33, 33
S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In Edgar Chávez and Stefano Lonardi, editors, *Proceedings of the 17th International Symposium on String Processing and Information Retrieval, SPIRE 2010*, volume 6393 of *Lecture Notes in Computer Science*, pages 201–206, Los Cabos, Mexico, 2010. Springer.

[Larsson and Sadakane, 2007] cité page 21
N. Jesper Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3) :258–272, 2007.

[Mäkinen and Navarro, 2005] cité page 25
V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1) :40–66, 2005.

[Manber and Myers, 1993] cité page 20, 20, 20, 21
U. Manber and G. Myers. Suffix arrays : a new method for on-line string searches. *SIAM Journal on Computing*, 22(5) :935–948, 1993.

[Maniscalco and Puglisi, 2006] cité page 21
M. A. Maniscalco and S. J. Puglisi. Faster lightweight suffix array construction. In *Proceedings of the 17th Australasian Workshop On Combinatorial Algorithms*, pages 16–29, Ayers Rock, Uluru, Australia, 2006.

[Manzini and Ferragina, 2004] cité page 21
G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1) :33–50, 2004.

[McCreight, 1976] cité page 17, 40
E. D. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2) :262–272, 1976.

[Melichar *et al.*, 2005] cité page 6
B. Melichar, J. Holub, and J. Polcar. Text searching algorithms. *Available on : http ://stringology. org/athens*, 2005.

[Morris and Pratt, 1970] cité page 1, 47, 61, 103
Jr. J. Morris and V. Pratt. *A linear pattern-matching algorithm.* 1970.

[Munro and Nekrich, 2015] cité page 9
J. Ian Munro and Y. Nekrich. *Compressed Data Structures for Dynamic Sequences*, pages 891–902. Springer Berlin Heidelberg, 2015.

[Na *et al.*, 2013a] cité page 39, 39, 40, 40, 40
J. C. Na, H. Park, M. Crochemore, J. Holub, C. S. Iliopoulos, L. Mouchard, and K. Park. Suffix tree of alignment : An efficient index for similar data. In Thierry Lecroq and Laurent Mouchard, editors, *Proceedings of the 24th International Workshop On Combinatorial Algorithms, IWOCA 2013*, volume 8288 of *Lecture Notes in Computer Science*, Rouen, France, 2013. Springer.

[Na *et al.*, 2013b] cité page 41, 41, 41, 42, 42
J. C. Na, H. Park, S. Lee, M. Hong, T. Lecroq, L. Mouchard, and K. Park. Suffix array of alignment : A practical index for similar data. In Moshe Lewenstein Oren Kurland and Ely Porat, editors, *Proceedings of the 20th International Symposium on String Processing and Information Retrieval, SPIRE 2013*, volume 8214 of *Lecture Notes in Computer Science*, pages 243–254, Jerusalem, Israel, 2013. Springer.

[Navarro and Mäkinen, 2007] cité page 24, 25, 38
G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys.*, 39(1), 2007.

[Navarro and Raffinot, 2002] cité page 5, 6
G. Navarro and M. Raffinot. *Flexible pattern matching in strings - practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002.

[Navarro, 2002] cité page 27
G. Navarro. Indexing text using the Ziv-Lempel trie. In Alberto H. F. Laender and Arlindo L. Oliveira, editors, *Proceedings of the 9th International Symposium on String Processing and Information Retrieval, SPIRE 2002*, volume 2476 of *Lecture Notes in Computer Science*, pages 325–336, Lisbon, Portugal, 2002. Springer.

[Nekrich, 2007] cité page 31
Y. Nekrich. Orthogonal range searching in linear and almost-linear space. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, editors, *Proceedings of the 10th International Workshop on Algorithms and Data Structures, WADS 2007*, volume 4619 of *Lecture Notes in Computer Science*, pages 15–26. Springer, Halifax, Canada, 2007.

[Nekrich, 2009] cité page 36
Y. Nekrich. Orthogonal range searching in linear and almost-linear space. *Computational Geometry*, 42(4) :342–351, 2009.

[Nsira *et al.*, 2014a] cité page 1, 1, 1, 77, 80, 80, 81, 81, 88, 89, 90, 103, 103
N. Ben Nsira, T. Lecroq, and M. Elloumi. A fast pattern matching algorithm for highly similar sequences. In *2014 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2014, Belfast, United Kingdom, November 2-5, 2014*, pages 32–38, 2014.

[Nsira *et al.*, 2014b] cité page 1, 1, 1, 12, 47, 49, 49, 61, 89, 90, 103, 103
N. Ben Nsira, T. Lecroq, and M. Elloumi. On-line string matching in highly similar DNA sequences. In *Proceedings of the 2nd International Conference on Algorithms for Big Data , Palermo, Italy, April 07-09, 2014.*, pages 16–22, 2014.

[Nsira *et al.*, 2017] cité page 1, 1, 1, 63, 65, 75, 89, 90, 103, 103
N. Ben Nsira, T. Lecroq, and M. Elloumi. On-line string matching in highly similar dna sequences. *Mathematics in Computer Science*, pages 1–14, 2017.

[Ohlebusch, 2013] cité page 9, 9, 10, 11, 98
E. Ohlebusch. *Bioinformatics Algorithms : Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013.

[Pandiselvam *et al.*, 2009] cité page 5
P. Pandiselvam, T. Marimuthu, and R. Lawrance. A comparative study on string matching algorithms of biological sequences, 2009.

[Panneton *et al.*, 2006] cité page 90
François Panneton, Pierre L'ecuyer, and Makoto Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software (TOMS)*, 32(1) :1–16, 2006.

[Patrascu, 2008] cité page 35, 35, 36
Mihai Patrascu. Succincter. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 305–313. IEEE, 2008.

[Pottier *et al.*, 2012] cité page 91
C Pottier, D Hannequin, S Coutant, A Rovelet-Lecrux, D Wallon, S Rousseau, S Legallic, C Paquet, S Bombois, J Pariente, et al. High frequency of potentially pathogenic sorl1 mutations in autosomal dominant early-onset alzheimer disease. *Molecular psychiatry*, 17(9) :875–879, 2012.

[Prieur, 2007] cité page 6
E. Prieur. *Méthodes et structures de données pour l'indexation et la détection de répétitions dans les séquences biologiques : les vecteurs de suffixes.* PhD thesis, Université de Rouen, 2007.

[Procházka and Holub, 2014] cité page 37, 37, 38, 39
P. Procházka and J. Holub. Compressing similar biological sequences using FM-index. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014*, pages 312–321. IEEE, 2014.

[Puglisi *et al.*, 2007] cité page 21
S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2) :4, 2007.

[Rahn *et al.*, 2014] cité page 42, 42, 43, 43, 43, 43, 43, 43, 90, 95, 96, 96, 96, 100
R. Rahn, D. Weese, and K. Reinert. Journaled string tree - a scalable data structure for analyzing thousands of similar genomes on your laptop. *Bioinformatics*, 30(24) :3499–3505, 2014.

[Reuter *et al.*, 2015] cité page 1, 17
J. A. Reuter, D. V. Spacek, and M. P. Snyder. High-throughput sequencing technologies. *Molecular Cell*, 58(4) :586 – 597, 2015.

[Roman, 2015] cité page 11
Steven Roman. *An Introduction to Catalan Numbers.* Birkh&#228 ;user Basel, 1st edition, 2015.

[Sadakane, 2003] cité page 22, 25
K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2) :294–313, 2003.

[Salson, 2009a] cité page 6
M. Salson. *Structures d'indexation compressées et dynamiques pour le texte*. PhD thesis, université de Rouen, 2009.

[Salson, 2009b] cité page 22
M. Salson. *Structures d'indexation compressées et dynamiques pour le texte*. PhD thesis, Université de Rouen, Dec 2009.

[Sanger *et al.*, 1977] cité page 1
F. Sanger, S. Nicklen, and AR. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of The National Academy of Sciences of The United States Of America*, 74(12) :5463–5467, 1977.

[Schürmann and Stoye, 2005] cité page 21
K. B . Schürmann and J. Stoye. An incomplex algorithm for fast suffix array construction. In Camil Demetrescu, Robert Sedgewick, and Roberto Tamassia, editors, *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics, ALENEX /ANALCO 2005*, pages 77–85, Vancouver, BC, Canada, 2005. SIAM.

[Sedgewick and Flajolet, 1996] cité page 14
R. Sedgewick and P. Flajolet. *An introduction to the analysis of algorithms.* Addison-Wesley-Longman, 1996.

[Shendure and Ji, 2008] cité page 1, 17
J. Shendure and H. Ji. Next-generation dna sequencing. *Nature biotechnology*, 26(10) :1135–1145, 2008.

[Stephen, 1994] cité page 5, 6
G. A. Stephen. *String searching algorithms*, volume 3. World Scientific, 1994.

[Ukkonen, 1995] cité page 17, 17, 17, 17, 19
E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3) :249–260, 1995.

[von Bubnoff, 2008] cité page 1, 17
A. von Bubnoff. Next-generation sequencing : the race is on. *Cell*, 132(5) :721–723, 2008.

[Vuillemin, 1980] cité page 11
J. Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4) :229–239, 1980.

[Weiner, 1973] cité page 17
P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory, SWAT (FOCS) 1873*, pages 1–11, Iowa City, IA, USA, 1973. IEEE Computer Society.

[Weinstein *et al.*, 2013] cité page 1
John N Weinstein, Eric A E. Collisson, Gordon B Mills, Kenna R Mills Shaw, Brad A Ozenberger, Kyle Ellrott, Ilya Shmulevich, Chris Sander, Joshua M Stuart, Cancer Genome Atlas Research Network, et al. The cancer genome atlas pan-cancer analysis project. *Nature genetics*, 45(10) :1113–1120, 2013.

[Willard, 1983] cité page 35
D. E. Willard. Log-logarithmic worst-case range queries are possible in space θ (n). *Information Processing Letters*, 17(2) :81–84, 1983.

[Zheng-Bradley and Flicek, 2016] cité page 1
Xiangqun Zheng-Bradley and Paul Flicek. Applications of the 1000 genomes project resources. *Briefings in functional genomics*, page elw027, 2016.

[Ziv and Lempel, 1978] cité page 27, 27
J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5) :530–536, 1978.