Filtrage de séquences d'ADN pour la recherche de longues répétitions multiples

Pierre Peterlongo

THÈSE DE DOCTORAT

en Informatique

Filtrage de séquences d'ADN pour la recherche de longues répétitions multiples

Pierre Peterlongo

Numéro officiel
Université de Marne-la-Vallée

Thèse de doctorat

Spécialité : Informatique

présentée par

Pierre Peterlongo

pour l'obtention du titre de

Docteur en Informatique

sur le sujet

Filtrage de séquences d'ADN pour la recherche de longues répétitions multiples

directeurs de thèse

Maxime Crochemore

Marie-France Sagot

soutenue le 21 Septembre 2006 devant le jury composé de :

Mathieu Blanchette (Rapporteur)

Christian Gautier (Rapporteur et Examinateur)

Gregory Kucherov (Rapporteur et Examinateur)

Marie-France Sagot (Examinateur)

Nadia Pisanti (Examinateur)

Maxime Crochemore (Examinateur)

Jean Berstel (Examinateur)

Thierry Lecroq (Examinateur)

Remerciements

Trois années se sont écoulées depuis le début de cette thèse. L'une des caractéristiques qui m'a le plus frappé durant ces années a été la richesse des rapports humains qu'engendrent les travaux de recherche. J'aimerais donc, dans les lignes qui suivent, tenter d'ébaucher des remerciements pour toutes ces personnes qui ont compté et qui compteront certainement longtemps, tout en m'excusant par avance pour celles et ceux que j'oublie peut-être.

Le plus grand des mercis ne suffirait pas pour aller vers Marie-France Sagot avec qui je travaille depuis bientôt cinq ans. Malgré la distance géographique qui nous sépare, un emploi du temps surchargé et mon caractère pas toujours évident à maîtriser, elle a toujours su répondre à mes appels, me guider dans mon travail, répondre à mes doutes et mettre sur mon chemin des personnes remarquables. Elle est la première personne à m'avoir fait aimer la recherche. En quelques années, en plus d'avoir été ma responsable, elle est devenue une amie. Merci.

Merci à Maxime Crochemore, pour sa gentillesse, ses (bons) conseils, sa disponibilité mais aussi pour ces bons moments partagés à Londres et à Prague.

Je remercie chaleureusement mes trois rapporteurs de thèse, à savoir Mathieu Blanchette, Christian Gautier et Gregory Kucherov. Je vous remercie pour le travail méticuleux que vous avez effectué pour la relecture de mon manuscrit et aussi pour les remarques pertinentes et fortement intéressantes que vous avez émises.

Grazie mille à Nadia Pisanti, pour sa spontanéité, sa franchise, sa collaboration et pour avoir fait tant de kilomètres pour participer au jury.

Merci également à Jean Berstel et à Thierry Lecroq, qui, après avoir été mes professeurs, se sont intéressés à mon travail et ont de même accepté de participer au Jury.

I'd like to thank Costas Iliopoulos for its warm welcome and for the nice moments we spent together, drinking a beer or not, and talking about work or not. I hope we'll have the opportunity to work again together.

Um imenso obrigado ao Alair por toda sua gentileza e pela abertura de espírito que o definem. Ele marcou minha maneira de ver a pesquisa e um certa visão das relacões humanas em geral.

Merci à Fred Boyer pour tous ces bons moments passés à écouter de grands classiques de la chanson française et parallèlement à réfléchir efficacement à certains problèmes scientifiques.

Ces remerciements ne seraient pas complets sans remercier Laurent Marsan, qui m'a mis dans le bain, d'une part en m'enseignant l'algorithmique, en m'intéressant à la bio-informatique et enfin en me présentant Marie-France. Il est certain que cette thèse n'aurait pas vu le jour sans lui.

Un immense merci aussi à Julien Allali. J'ai usé de ses impressionnantes compétences techniques comme de sa gentillesse durant plus de trois années. J'espère que cela continuera.

Merci à Ben qui a été un ami contre vents bien avant le début de cette thèse. D'autres en profitent maintenant :-)

Ces quelques années au sein de l'IGM se sont déroulées dans une ambiance amicale et détendue. Merci donc à tous les habitants du quatrième étage pour l'environnement que vous savez créer grâce à votre bonne humeur. Merci en particulier à Frédérique Bassino pour nos travaux communs, mais aussi pour avoir écouté mes états d'âmes, à Julien Clément qui a supporté mes fausses notes et à Nico Bedon qui a toujours écouté mes histoires l'air intéressé. Merci d'autre part à Cathy et sa balise argos, Adrien et nos CROUS, Guillaume et ses bonnes soirées, Pierre et sa mauvaise humeur, Michal qui ne nous a jamais vu manger une entrée, Remi et ses cocas, Étienne qui m'a aidé à meu-

bler mon appartement et Zip qui remplit si souvent la cafet' de bonnes choses.

Merci aussi à Line et à Andrée qui ont puisé dans leur patience pour supporter mon (léger) manque d'organisation (passager).

Merci à Patrice pour son soutient technique mais aussi merci de m'avoir épargné.

Merci à Sophie Créno pour avoir testé ED'NIMBUS et pour m'avoir suggéré d'importantes modifications.

Merci à l'équipe BAOBAB (Biologie Algorithmique Ou Bien Algorithmique Biologie) de Lyon pour leur accueil et leur gentillesse. Merci en particulier à Claire pour sa collaboration et pour avoir, elle aussi, supporté mon humour.

Diécuyé à Jan Holub pour sa sympathie, sa *childishness* et nos collaborations. J'en profite également pour remercier Marie Faturova pour son accueil à Prague.

If you want a nice hot strong cup of tea, just ask Zig. You should trust me Zig, there are some wild boars here in France. Thanks again for all the time we spent together and for your friendship.

Un grand merci à tous ceux qui ont partagé le bureau et de (très) bons moments avec moi, merci donc à Édouardo et son accueil, Charlotte et son accent, Zig (again), Chloé et ses informations (je vais bientôt gagner une bouteille de champagne ?), Daminou et son humour, Serge et ses vaches, Kinoton et ses feuilles.

Merci également à mes amis (précieux) qui quotidiennement font que la vie est belle, Céline qui a beaucoup contribué a ce qui a été écrit durant cette thèse :-), Scott (come on Tiger), Vincent, Mathilde, Philou, Sophie, Nico, Benjamin, Matthieu, COrAline, Max, François, Seb, Laureline, Steeve, ... Merci aussi à tous ceux qui ont peuplé les salles d'infos durant la licence et la maîtrise, si je regrette aujourd'hui cet escalator, c'est pour l'ambiance que vous y avez créé, peau de pamplemousse.

Merci à Marie-Claire, Francis, Sophie, Pupuce et le ptitbo pour leur ac-

cueil toujours chaleureux et pour tous ces bons moments passés ensembles.

Un immense merci bien sur à ma famille la plus proche, Zé, Grand-Papa, Mam, Maman, Papa, Laurent. Il est certain que sans leur soutien et leur amour cette thèse n'aurait jamais ni débuté ni abouti.

Merci enfin à Émilie. Sans jamais douter une seconde, en plus de m'apporter tous les jours une dose de bonheur immense, elle m'a permis de croire en moi dans les moments de doutes et de profiter pleinement de la vie tout le reste du temps. Merci Mil :-).

Résumé

La génomique moléculaire fait face en ce début de siècle à de nouvelles situations qu'elle doit prendre en compte. D'une part, depuis une dizaine d'années, la quantité de données disponibles croît de manière exponentielle. D'autre part, la recherche dans le domaine implique de nouvelles questions dont les formulations *in silico* génèrent des problèmes algorithmiquement difficiles à résoudre.

Parmi ces problèmes, certains concernent notamment l'étude de réarrangements génomiques dont les duplications et les éléments transposables. Ils imposent que l'on soit en mesure de détecter précisément et efficacement de longues répétitions approchées et multiples dans les génomes. Par répétition multiple, nous désignons des répétitions ayant au moins deux copies dans une séquence d'ADN, ou ayant des copies dans au moins deux séquences d'ADN distinctes. De plus, ces répétitions sont approchées dans le sens où des erreurs existent entre les copies d'une même répétition.

La recherche de répétitions approchées multiples peut être résolue par des algorithmes d'alignements multiples locaux mais ceux-ci présentent une complexité exponentielle en la taille de l'entrée, et ne sont donc pas applicables à des données aussi grandes que des génomes. C'est pourquoi, de nouvelles techniques doivent être créées pour répondre à ces nouveaux besoins.

Dans cette thèse, une approche de filtrage des séquences d'ADN est proposée. Le but d'une telle approche est de supprimer rapidement et efficacement, parmi des textes représentant des séquences d'ADN, de larges portions ne pouvant pas faire partie de répétitions. Les données filtrées, limitées en majorité aux portions pertinentes, peuvent alors être fournies en entrée d'un algorithme d'alignement multiple local.

Les filtres proposés appliquent une condition nécessaire aux séquences pour n'en conserver que les portions qui la respectent. Les travaux que nous présentons ont porté sur la création de conditions de filtrage, à la fois effi-

caces et simples à appliquer d'un point de vue algorithmique. À partir de ces conditions de filtrage, deux filtres, NIMBUS et ED'NIMBUS, ont été créés. Ces filtres sont appelés exacts car il ne suppriment jamais de données contenant effectivement des occurrences de répétitions respectant les caractéristiques fixées par un utilisateur. L'efficacité du point de vue de la simplicité d'application et de celui de la précision du filtrage obtenu, conduit à de très bons résultats en pratique. Par exemple, le temps utilisé par des algorithmiques de recherche de répétitions ou d'alignements multiples peut être réduit de plusieurs ordres de grandeur en utilisant les filtres proposés.

Il est important de noter que les travaux présentés dans cette thèse sont inspirés par une problématique biologique mais ils sont également généraux et peuvent donc être appliqués au filtrage de tout type de textes afin d'y détecter de grandes portions répétées.

Abstract

Since a few years, molecular genomics has had to deal with new situations. First, the amount of data available is increasing exponentially. Second, research in this domain involves some new questions which lead to problems that are algorithmically difficult to solve.

Among such problems, some are related to the study of genomic rearrangements, including duplicated and transposable elements. Such a task requires the capacity to detect accurately and efficiently long multiple approximate repetitions in the genomes. A multiple repetition refers to a repetition having at least two copies in a DNA sequence, or having copies in a least two distinct DNA sequences. Furthermore, the repetitions involved are called approximate because their occurrences are distant from another by some errors like insertions, deletions and substitutions.

The problem of searching for long multiple approximate repetitions may be solved by multiple local alignment algorithms. Such algorithms have a complexity that is exponential with the size of the input. Therefore they cannot be applied to data as big as genomes. This is the reason why new techniques have to be created to address these new problems.

In this PhD thesis, a filtration approach for comparing DNA sequences is proposed. The goal of this approach is to remove accurately and efficiently, from texts representing DNA, large portions that cannot contain an occurrence of a repetition. Filtered data, which in general will then correspond to the relevant portions, may be used as input of a multiple local alignment algorithm.

The filters proposed apply a necessary condition on the sequences. Only portions of sequences respecting this condition are conserved. The work presented deals with the creation of filtration conditions. Such conditions have to be both efficient and, from an algorithmic point of view, easy to apply. Using the provided filtration conditions, two filters, Nimbus and Ed'Nim-

BUS were created. These filters are called exact because the condition applied guarantees that no relevant part of the data may be filtered out. Its efficiency, both in terms of the accuracy of the filtration and of the time consumption, leads to very good practical results. For instance, the time spent by repetition extraction algorithms or multiple alignment algorithms may be reduced by several orders of magnitude using one of the proposed filters.

It is worth to notice that the work presented in this PhD thesis was motivated by biology, however, it is generic and can thus be used to filter of any other kinds of text with the aim to detect long multiple repeated portions.

Notes de lecture

Les particularités de ce manuscrit sont les suivantes :

Glossaire. Un glossaire est proposé page 237. Lors de leur première apparition dans le texte (en dehors de l'introduction,) les mots possédant une définition dans le glossaire sont suivis d'une étoile en exposant. Par exemple : «*Les premières traces de vie connues sont des Stromatolites* datant de 3,5 milliards d'années [...]*».

Index. Page 233, un index permet de retrouver la (les) localisation(s) dans le manuscrit des mots présentant une difficulté ou un intérêt particulier.

Illustrations. Lorsque la provenance n'est pas spécifiée dans la légende des figures, celles-ci ont été créées par l'auteur.

Table des matières

Table des figures

Liste des Algorithmes

Notations

Chaînes de caractères

- Σ : Alphabet (ensemble de caractères) ;
- Σ^* : Ensemble des chaînes de caractères constructibles sur Σ ;
- Pour une chaîne de caractères s :
  - $|s|$: Longueur de s ;
  - $s[i]$: $(i+1)^{\text{ième}}$ caractère de la chaîne s ;
  - $s[0]$: Premier caractère de s ;
  - $s[i, j]$: Facteur de s débutant à la position i et terminant à la position j incluse ;
  - $s[i \dots]$: Suffixe de s débutant à la position i incluse ;
  - $s[\dots j]$: Préfixe de s terminant à la position j incluse.

Répétitions

- L : Longueur des occurrences des répétitions ;
- r : Nombre d'occurrences d'une répétition ;
- d : Nombre maximal d'erreurs autorisées entre toute paire d'occurrences.

Filtrage

- k : Longueur des k-facteurs ;
- p : Nombre minimum de k-facteurs partagés ;
- m : Nombre de séquences ;
- n : Taille moyenne des séquences.

Introduction

La découverte de l'ADN, suivie quelques dizaines d'années plus tard par le séquençage de génomes complets, ont conduit à la production d'une énorme quantité de données. Celles-ci recèlent des informations primordiales qui devraient nous permettre d'apporter des éléments de réponse à certaines questions fondamentales. D'un point de vue biologique, ces données sont nécessaires à la compréhension de notre passé, vis-à-vis de notre évolution depuis l'apparition des premières cellules et de notre place dans le monde, en regard des autres espèces. D'un point de vue plus pratique, l'analyse de ces données peut nous permettre de comprendre le fonctionnement de certaines maladies, et ainsi nous aider à lutter contre elles.

L'importante quantité d'informations disponible a naturellement rapproché l'informatique et la biologie, deux domaines a priori distincts, pour créer la bio-informatique. En effet, l'œil du généticien ne suffisait plus pour extraire les informations contenues dans ces nouvelles données. Rappelons, par exemple, que le génome humain contient environ trois milliards de nucléotides rendant impossible son analyse globale sans avoir recourt à la puissance de calcul offerte par l'outil informatique.

L'algorithmique du texte, domaine de l'informatique antérieur aux problèmes posés par la génétique, a vu son champ d'application élargi par l'arrivée de ces données. Les séquences d'ADN, codées sous forme de textes sur un alphabet à 4 lettres (A, C, T et G), se prêtent très bien au traitement par des algorithmes déjà existants. Cependant, de nouvelles questions sont posées, menant à la modification, voire à la création de nouveaux algorithmes destinés à être appliqués et optimisés pour ce nouveau type de données, dont la «grammaire» est inconnue. La résolution de ces problèmes résulte alors d'une collaboration entre le biologiste et l'informaticien. Ceci nécessite une compréhension mutuelle, de la part de l'informaticien pour appréhender les spécificités du problème posé et de la part du biologiste pour prendre en

compte les particularités et les limitations de l'informatique. Les échanges entre ces deux domaines sont extrêmement motivants, et les deux disciplines s'enrichissent mutuellement.

* *

*

L'augmentation de la quantité de données disponible est extraordinaire, par exemple la taille de la plus grosse banque de séquences européenne, EMBL («European Molecular Biology Laborator») croit de manière exponentielle depuis plus de 25 ans (voir figure 1.10, page 42) pour contenir aujourd'hui environ 132 milliards de nucléotides. En parallèle, de nouveaux problèmes sont soulevés par le monde de la biologie. Ainsi, la nécessité de développement de nouveaux programmes est grande. Ces nouveaux programmes doivent donc, à la fois résoudre des problèmes souvent complexes et nécessitant une grande précision d'analyse, et être de plus capable de traiter une masse de données très importante en un temps «raisonnable».

Le but des travaux présentés dans cette thèse est précisément de chercher à accélérer des algorithmes pour lesquels il n'existe pas encore de solution en temps raisonnable. Les algorithmes concernés dans cette thèse visent à détecter dans les séquences d'ADN des portions répétées. La détection de répétitions représente l'une des activités clefs de la génomique. Une telle activité permet, entre autres, d'inférer des connaissances, de retracer l'évolution des espèces, de comprendre la structure des génomes ou encore le développement et la transmission de certaines maladies.

En particulier, l'étude de réarrangements génétiques impliquant les duplications et les transpositions de gènes représente un domaine mobilisant beaucoup d'énergie de recherche. De tels réarrangements jouent un rôle primordial dans l'évolution et permettent de conserver la diversité. De plus, de possibles liens existent entre la dynamique des génomes et certaines maladies comme le cancer. Ainsi, ce type de phénomènes présente un vif intérêt pour les biologistes. Les répétitions auxquelles nous nous intéressons se manifestent sous forme de longues portions apparaissant répétées au moins deux fois dans les séquences d'ADN (possiblement sur différentes séquences). Malgré l'intérêt majeur porté à ces phénomènes, il n'existe pas de moyen informatique permettant la détection de telles répétitions en un temps raisonnable. Ceci s'explique par la longueur des répétitions impliquées et par leur taux de similarité qui peut être assez faible, rendant difficile leur détection. De plus,

les phénomènes de réarrangements concernent des génomes entiers. Les outils de recherche de répétitions doivent donc être capable de traiter plusieurs millions de nucléotides.

Il existe aujourd'hui des méthodes approchées permettant de détecter rapidement de telles répétitions. Les méthodes approchées, en opposition aux méthodes exactes, désignent les méthodes dont l'aboutissement n'est pas garanti d'être la solution optimale. Dans le cadre de cette thèse, nous nous sommes focalisé sur les méthodes exactes. Il existe un moyen de détecter de manière exacte de longues répétitions multiples (apparaissant deux fois ou plus) dans les séquences. Ce moyen est appelé l'alignement multiple local. La limitation de cette méthode réside dans le fait qu'elle nécessite des temps de calculs exponentiels par rapport à la taille des données. Ceci se traduit, par exemple, par des centaines d'années de calcul pour comparer quatre génomes d'un million de nucléotides chacun.

Nous avons cherché à apporter une méthode combinant des temps de calculs raisonnables avec une solution exacte. Pour arriver à relever un tel défi, nous proposons d'utiliser un **filtre**. L'idée est d'appliquer aux données initiales un traitement supprimant rapidement de larges portions ne pouvant pas faire partie d'une répétition. Une condition nécessaire est appliquée aux séquences. Les portions qui ne respectent pas cette condition ne peuvent pas être une occurrence d'une répétition, et sont supprimées. Ainsi, les données filtrées contiennent toutes les répétitions initiales, mais aussi des portions de séquences respectant la condition mais ne faisant pas partie de répétitions. Un programme d'alignement multiple local est alors appliqué aux séquences conservées.

Nous proposons dans cette thèse deux filtres, appelés NIMBUS et ED'NIMBUS. Ces filtres appliquent une condition nécessaire basée sur le nombre de mots partagés par les occurrences des répétitions recherchées. Nous constatons que des séquences «similaires» partagent un certain nombre de mots. Ceci implique que des séquences ne partageant «pas assez» de mots ne peuvent être considérées comme similaires. Les filtres que nous présentons utilisent une définition formelle de la notion de similarité. Ils détectent et suppriment les portions de séquences ne partageant pas au moins un certain nombre de mots avec d'autres portions de séquences.

Comme nous le constaterons dans cette thèse, une telle approche permet de réduire de plusieurs ordres de grandeurs certains calculs de détection de répétitions. Par exemple, nous sommes parvenus, par l'utilisation de ces filtres, à réduire certains calculs de plusieurs heures à quelques minutes, tout

en obtenant les mêmes résultats. Ceci offre alors aux biologistes l'opportunité de disposer de résultats exacts de détection de répétitions en un temps acceptable.

* *

*

Il existe de nombreuses techniques de filtrage, exacte ou non, utilisant différentes méthodes et créées pour résoudre différents problèmes biologiques. Dans certaines situations, l'utilisation de filtres est indispensable pour obtenir, en temps raisonnable, les résultats attendus par biologistes.

Cependant, il n'existe pas à notre connaissance de filtre permettant d'accélérer la détection de répétitions possédant plus de deux occurrences. Une telle absence représente un manque important d'outil pour étudier la dynamique des génomes et les répétitions associées. C'est ce constat qui a motivé les travaux que nous présentons dans ce manuscrit.

* *

*

La suite de ce manuscrit est divisé en cinq chapitres. Dans le premier chapitre nous présentons le contexte biologique des travaux proposés dans cette thèse. Nous nous attarderons en particulier sur la présentation des répétitions dans les génomes et de leur impact.

Le deuxième chapitre propose une transition vers l'algorithmique du texte. Nous y introduirons la notion d'algorithmique, et nous nous focaliserons sur le problème de détection de similarités en présentant les solutions existantes.

Le troisième chapitre est voué à dresser un état de l'art des connaissances et des algorithmes dédiés aux filtrage dans les textes. Nous y présenterons les filtres existants en nous attardant particulièrement sur deux filtres, appelés QUASAR et SWIFT, présentant un intérêt particulier vis-à-vis des thèmes développés dans cette thèse.

Les travaux que nous avons réalisé ont conduit à la création de deux algorithmes de filtrage, NIMBUS et ED'NIMBUS. Ceux-ci sont présentés, respectivement, dans les quatrième et cinquième chapitres. Pour chacun d'entre eux, nous présenterons le cadre de filtrage, la condition nécessaire appliquée et l'algorithme associé. Nous donnerons de plus pour chacun de ces chapitres

le résultat de tests visant à estimer les qualités mais aussi les défauts des solutions proposées.

Enfin, en conclusion de ce mémoire, nous dresserons un résumé des méthodes créées et des résultats obtenus. Nous y présenterons l'éventail des directions que nous chercherons à suivre par la suite. Nous y verrons que certains travaux futurs viserons à améliorer les qualités des filtres proposés alors que d'autres chercherons à appliquer les idées utilisées à d'autres problèmes biologiques.

Chapitre 1

Un aperçu du contexte biologique

Les travaux présentés dans cette thèse ont pour ambition d'apporter une solution à certains défis soulevés par la biologie moléculaire. Afin de placer ces travaux dans leur contexte, nous présenterons dans ce chapitre les principes fondamentaux de la génomique avant de nous focaliser plus spécifiquement sur les points précis qui ont motivé les travaux que nous proposons dans ce manuscrit.

Ainsi, et malgré la tentation de décrire en détail nombre de phénomènes passionnants, ce chapitre propose une introduction très brève à la génétique plutôt qu'une présentation générale des connaissances de la biologie moléculaire. Pour obtebnir de plus amples informations, de nombreux ouvrages ont pour vocation de présenter les mécanismes de la vie. Nous invitons ainsi les lecteurs curieux à s'y référer. Certains ouvrages très complets [ABL+90, Lew05] permettent une compréhension fine et profonde de la biologie moléculaire alors que de nombreux livres de vulgarisation comme [Kel03] sont disponibles.

Nous entamerons ce chapitre par une présentation des principaux éléments des êtres vivants, en parlant notamment des cellules et de leur contenu. Dans une deuxième section, nous décrirons les mécanismes qui se basent sur l'information génétique contenue dans les cellules pour synthétiser les éléments caractéristiques des êtres vivants. La section 1.3 est quant à elle dédiée à la présentation des phénomènes de mutation et de leurs possibles conséquences sur l'organisme et les espèces. Nous y parlerons plus en détail des répétitions existantes dans les génomes qui sont d'un grand intérêt pour les généticiens. Une partie des travaux effectués dans cette thèse sont axés sur la découverte de ces répétitions. Enfin, dans la section 1.4, nous verrons com-

ment, une partie de l'information génétique peut être représentée sous forme textuelle pour être exploitable par un ordinateur.

* *

*

Avant d'entrer dans le vif du sujet, gardons à l'esprit que tous les phénomènes biologiques sont le résultat de réactions chimiques partielles, multiples et complexes. Ainsi, dans ce domaine, une grande partie des règles connues sont vraies sous certaines conditions uniquement et la plupart sont sujettes à des exceptions. Les notions présentées dans ce chapitre ne doivent donc pas être considérées comme des faits établis, d'autant plus que la recherche en biologie apporte elle-même des découvertes venant occasionnellement modifier les connaissances acquises.

Ce chapitre a été rédigé en regroupant des informations provenant des sources suivantes :

- Le très complet ouvrage *Biologie moléculaire de la cellule* [ABL+90]
- Le site web de INFOBIOGEN : `http://www.infobiogen.fr`
- Le site de GÉNET, réseau d'enseignement en génétique : `http://www.univ-tours.fr/genet/`
- Le campus virtuel de l'Université Paris-Sud : `http://formation.etu.u-psud.fr/biologie/genetique`

1.1 Les protagonistes de la vie

Où se trouve la limite entre la matière inerte et le vivant ? La réponse à cette question en apparence simple n'est pas si évidente. D'un point de vue courant, il s'agit d'une étendue temporelle, entre la naissance et la mort. Cette définition est elle même liée à la notion de mort qui n'est pas non plus simple à définir. D'un point de vue scientifique, la définition du vivant est basée sur un ensemble de critères vérifiés simultanément ou non. Ces critères portent sur la capacité de réaction, de motricité, de croissance, de consommation et de reproduction.

Cependant, ces critères ne sont pas toujours satisfaisants. Prenons par exemple le feu, celui-ci consomme de l'énergie, pourtant nous ne pouvons pas considérer le feu comme de la matière vivante. C'est pourquoi les trois points suivants ont été ajoutés à ces définitions. Les organismes vivants se

FIG. 1.1 – Plus ancienne trace connue de vie sur terre, ces stromatolites précambrien semblent dater de 3,5 milliards d'années (Source : *National Park Service (NPS), U.S. Department of the Interior 2005*)

maintiennent grâce à l'homéostasie* et ils contiennent des cellules*. Un troisième point, indique que les organismes vivants contiennent des molécules à base de carbone. Ce dernier point est source de contestations car nous pouvons émettre l'hypothèse de l'existence possiblement extra-terrestre de vie non basée sur le carbone. Notons que les premières traces de vie connues sont des Stromatolites* datant de 3,5 milliards d'années dont une image est présentée dans la figure 1.1.

C'est sur la notion de cellule que nous allons nous attarder, nous allons chercher à comprendre leur constitution et leur fonctionnement. Ces entités sont primordiales car elles contiennent l'information génétique*, qui détermine les caractéristiques des individus.

1.1.1 Les cellules

Le mot cellule vient du latin *cellula*, ce qui signifie *petite chambre* dont l'origine est la cellule monastique. En biologie, il s'agit d'une unité structurale

et fonctionnelle constituant tout ou une partie d'un être vivant. La cellule, liée à la notion de vie, est capable de se reproduire et de réagir aux conditions extérieures.

Différents types de cellules

Il existe deux types élémentaires de cellules : les cellules des organismes eucaryotes et les cellules des organismes procaryotes. La différence majeure entre ces deux structures réside dans le fait que les cellules procaryotes ne possèdent pas de noyau (du grec *pro* : avant et *caryon* : noyau) alors que les cellules eucaryotes en possèdent un (du grec *eu* : avec et *caryon* : noyau). Les cellules eucaryotes sont plus complexes tant dans leur organisation structurale que dans leur comportement. Les organismes tels que les champignons, plantes et animaux sont composés de cellules eucaryotes alors que les cellules procaryotes se trouvent principalement chez les bactéries. L'être humain est un exemple d'espèce contenant des cellules eucaryote. Selon les estimations, il contient environ 50 000 milliards de cellules classées selon 220 types différents en fonction des tissus dont elles font partie.

Composition des cellules

Même si leur composition diffère en fonction des espèces* et de leur rôle au sein des individus de ces espèces, certains constituants de base, les *briques* de la vie sont communs à tout type de cellule.

- L'ADN* contient l'information génétique. Une description détaillée de la composition et du rôle de l'ADN est présentée dans la section 1.1.2. L'ADN représente environ 0.25 % de la cellule.
- L'ARN* constitue environ 1.1 % de la cellule. Les ARNs se divisent en sous types et possèdent de nombreux rôles distincts, en particulier ils sont utilisés lors de la synthèse (voir section 1.2) des protéines.
- Les protéines* sont le constituant principal de la cellule dont elles représentent environ 18 % (le constituant majoritaire étant l'eau présent à 70 %). Elles sont synthétisées à partir de l'ADN par un processus que nous décrirons dans la section 1.2. Une description de la structure et du rôle des protéines sera présentée dans la section 1.1.3.

À quoi sert une cellule ?

Les cellules d'une entité vivante se différencient et s'organisent pour créer les organes et assurer la fonction de ceux-ci. Les cellules *communiquent* les unes avec les autres via des substances chimiques et peuvent ainsi influencer le comportement global d'un ensemble de cellules cibles spécifiques. De plus, elles échangent avec l'extérieur pour obtenir les matières nécessaires à leur fonctionnement et pour éliminer les matériaux devenus inutiles pour elles. Les cellules sont des *usines* à protéines. Elles sont capables de fabriquer, comme nous le verrons dans la section 1.2, différentes protéines comme par exemple des hormones ou des anticorps. Les caractéristiques des protéines créées par les cellules dépendent des gènes (voir section 1.1.2) contenus dans l'ADN des cellules.

Les cellules ont une durée de vie variant de quelques jours à plusieurs dizaines d'années. Par exemple, les cellules de la peau ont une durée de vie d'environ trois semaines alors que les cellules du cerveau (les neurones) peuvent exister pendant toute la durée de vie d'un individu. Pendant leur cycle de vie, les cellules sont capables de se reproduire pour former deux cellules identiques, comme nous le verrons dans la section 1.3.1.

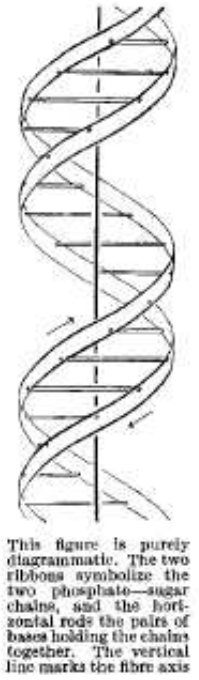
Avant d'entrer dans les détails des comportements de création de protéines ou de reproduction des cellules, nous présentons deux principaux acteurs de la cellule que nous avons déjà évoqués, à savoir l'ADN et les protéines.

1.1.2 L'ADN

Le mot ADN est un acronyme pour désigner l'Acide DésoxyriboNucléique. L'acide désoxyribonucléique est une molécule que l'on retrouve, à l'exception des virus à ARN, dans la totalité des organisme vivants. L'ADN est le support de l'hérédité car il peut se recopier (voir section 1.3.1) et se transmettre aux descendants.

Structure de l'ADN

L'ADN se compose d'une structure en double hélice découverte conjointement par Rosalind Franklin, James Watson et Francis Crick qui les ont publiés dans *Nature* en 1953 [WC53]. L'illustration originale de cette structure telle quelle était présentée dans *Nature* est visible dans la figure 1.2.



This figure is purely diagrammatic. The two ribbons symbolize the two phosphate—sugar chains, and the horizontal rods the pairs of bases holding the chains together. The vertical line marks the fibre axis

FIG. 1.2 – La structure en double hélice de l'ADN telle quelle était présentée dans l'article original de *Nature* [WC53].

La double hélice d'ADN est donc composée de deux brins d'ADN. Chaque brin est constitué d'une succession de nucléotides*. Un nucléotide est composé d'un groupe phosphate lié à un sucre appelé désoxyribose et à une base azotée 'A' (adénine), 'C' (cytosine), 'G' (guanine) ou 'T' (thymine). La figure 1.3 présente la composition de l'ADN.

L'information génétique est générée par la succession de nucléotides différenciés par leur bases azotées. Nous verrons dans la section 1.2 comment cette information est utilisée pour synthétiser les protéines et ainsi pour donner ses caractéristiques à l'organisme.

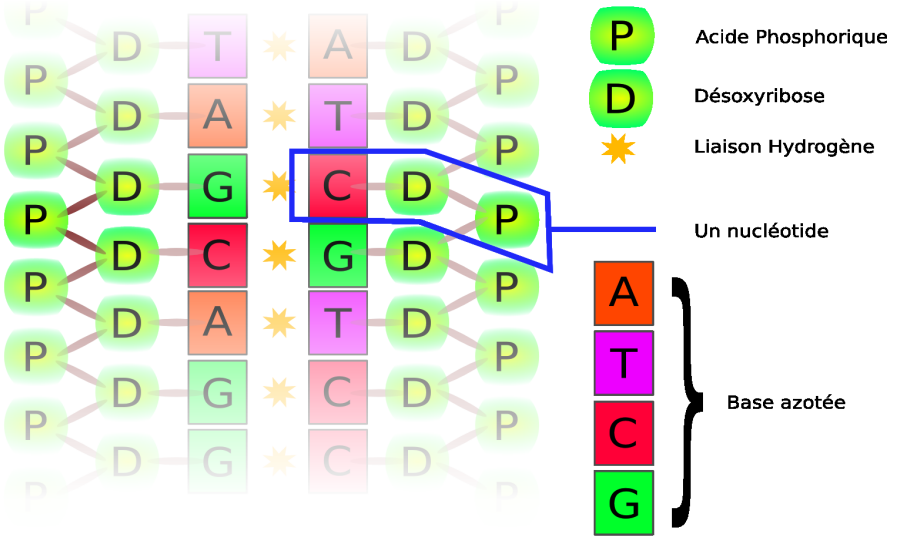


FIG. 1.3 – Composition de l'ADN

1.1.3 Les protéines

Une protéine est une succession d'acides aminés*. Les protéines sont synthétisées à partir des molécules d'ADN (voir section 1.2). En fonction des propriétés (hydrophobicité, charge électrique, ...) des acides aminés qui les composent, les protéines prennent différentes conformations dans l'espace et interagissent différemment les unes avec les autres. La forme tridimensionnelle est l'une des caractéristiques principales des protéines, c'est elle qui, majoritairement, en détermine la fonction. Les protéines ont des rôles très divers, comme par exemple un rôle messager (protéines impliquées dans des processus de signalisation cellulaire), structural (impliquées dans la cohésion structurale entre cellules) ou encore enzymatique (catalysent certaines réactions biochimiques).

Les protéines, étant directement dépendantes des gènes dont elles sont issues, représentent un chaînon du lien entre le génotype* (constitution gé-

nétique de l'individu) et le phénotype* (ensemble des caractéristiques biochimiques, physiologiques et morphologiques d'un individu).

1.2 De l'ADN aux protéines

Dans cette section, nous présentons ce qu'il est courant d'appeler le *dogme central* de la biologie moléculaire. Le dogme central explique les processus permettant de synthétiser des protéines en fonction de l'information contenue dans l'ADN. Crick a émis l'idée de ce dogme central en 1958 [Cri58] et l'a confirmé en 1970 [Cri70]. La figure 1.4 illustre le dogme central.

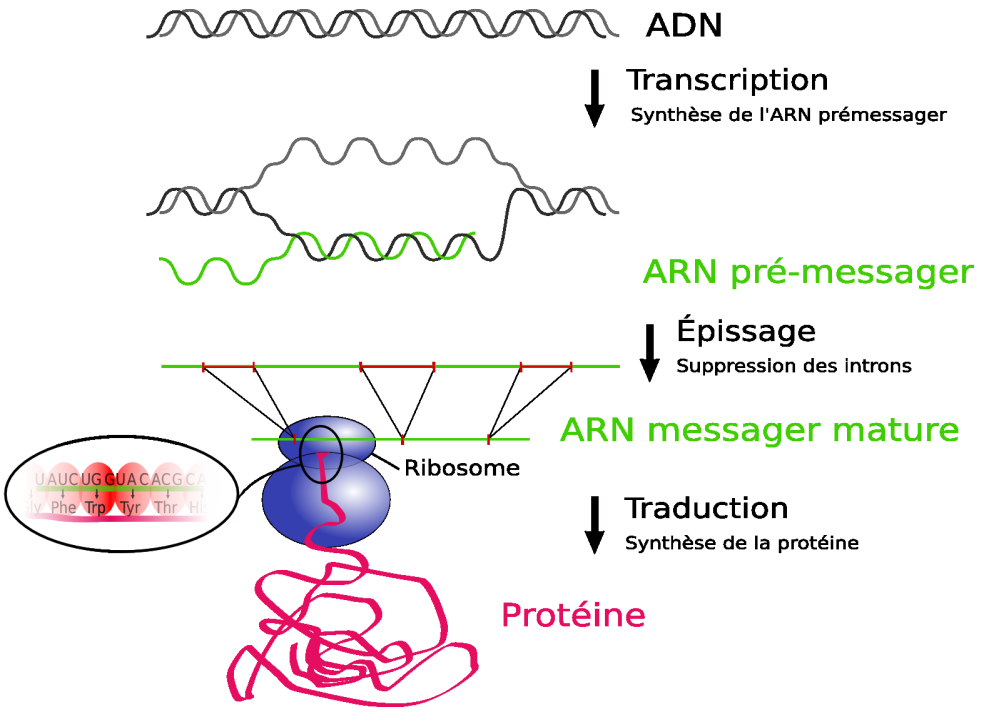


FIG. 1.4 – Dogme central de la biologie moléculaire.

Une portion d'ADN *codant* pour une protéine est appelé un gène*. La synthèse d'une protéine par un gène se fait en plusieurs étapes distinctes :

1. La première étape s'appelle la transcription* . Cette étape consiste au passage de l'ADN du gène en ARN appelé ARN pré-messager. Lors de cette étape, l'information génétique est strictement conservée. Chaque nucléotide de l'ADN est transcrit en un nucléotique de l'ARN messager. Les transcriptions sont les suivantes : 'C' est transcrit en 'G' (et vice versa), 'T' est transcrit en 'A' et 'A' est transcrit en un nouveau nucléotide, l'uracile noté 'U'.
2. À la suite d'une phase dite d'épissage* (phase propre aux eucaryotes) qui consiste en l'excision d'une partie de l'ARN pré-mature (les introns) et à la réunion des parties restantes (les exons), l'ARN pré-messager est ainsi transformé en ce qui est appelé l'ARN messager mature.
3. Enfin lors de la traduction*, les protéines sont synthétisées par la lecture de l'information génétique contenue dans l'ARN messager mature. Lors de la traduction, la correspondance entre les nucléotides de l'ARN et les acides aminés des protéines est déterminée par ce qui est appelé le code génétique. Chaque triplet de nucléotides (appelé un codon*) détermine le choix d'un acide aminé correspondant dans la protéine. Le code génétique* mettant en relation les codons avec les acides aminés est présenté sur la table 1.1. Le code génétique est le même chez la quasi totalité des organismes vivants (chez les mitochondries, quatre codons sur soixante quatre ont une traduction différente de ce qui est présenté sur cette table).

Remise en cause du dogme central Il est nécessaire de noter que ce dogme central est aujourd'hui partiellement remis en cause. Les principales transgressions sont les suivantes :

- **La transcription inverse**. Elle a été découverte comme faisant partie du cycle réplicatif des rétrovirus, dont le plus connu est le virus du SIDA, le HIV. Le génome des rétrovirus est constitué d'ARN, qui est transcrit en ADN après infection et va s'intégrer dans le génome de l'hôte, d'où il est à nouveau transcrit pour former de l'ARN et des protéines virales.
- **La réplication de l'ARN**. Ceci est une caractéristique de la plupart des virus à ARN, qui produisent leurs propres enzymes pour répliquer leurs génomes
- **La pseudo-réplication de protéines**. Les prions sont des agents infectieux avec des caractéristiques similaires aux virus, mais qui ne

contiennent pas d'acides nucléiques. Le dogme voudrait qu'ils ne puissent pas se répliquer, et ne puissent donc pas être infectieux. Il semble maintenant que les prions sont en effet des protéines porteuses d'information, mais que cette information n'est que structurelle : les prions, dérivés de protéines du cerveau, peuvent induire chez des protéines normales un changement de conformation qui se propage graduellement, et peut être transmis d'individu à individu.

1ère base	2ème base				3ème base
	U	**C**	**A**	**G**	
U	Phe(F)	Ser(S)	Tyr(Y)	Cys(C)	**U**
	Phe(F)	Ser(S)	Tyr(Y)	Cys(C)	**C**
	Leu(L)	Ser(S)	**STOP**	**STOP**	**A**
	Leu(L)	Ser(S)	**STOP**	Trp(W)	**G**
C	Leu(L)	Pro(P)	His(H)	Arg(R)	**U**
	Leu(L)	Pro(P)	His(H)	Arg(R)	**C**
	Leu(L)	Pro(P)	Gln(Q)	Arg(R)	**A**
	Leu(L)	Pro(P)	Gln(Q)	Arg(R)	**G**
A	Ile(I)	Thr(T)	Asn(N)	Ser(S)	**U**
	Ile(I)	Thr(T)	Asn(N)	Ser(S)	**C**
	Ile(I)	Thr(T)	Lys(K)	Arg(R)	**A**
	Met(M)	Thr(T)	Lys(K)	Arg(R)	**G**
G	Val(V)	Ala(A)	Asp(D)	Gly(G)	**U**
	Val(V)	Ala(A)	Asp(D)	Gly(G)	**C**
	Val(V)	Ala(A)	Asp(D)	Gly(G)	**A**
	Val(V)	Ala(A)	Asp(D)	Gly(G)	**G**

TAB. 1.1 – Le code génétique.
À chaque codon correspond un acide aminé. Chaque acide aminé est codé par 1 à 6 codons.

Initiation de la transcription.

L'initiation de la transcription d'un gène est sujet à la présence d'un facteur de transcription* en amont du gène. Le facteur de transcription initie et est indispensable à l'expression d'un gène. La fixation de ce facteur sur la molécule d'ADN résulte de l'action simultanée d'un ensemble d'éléments fonctionnels de l'ADN en amont du gène. La figure 1.5 illustre ce propos.

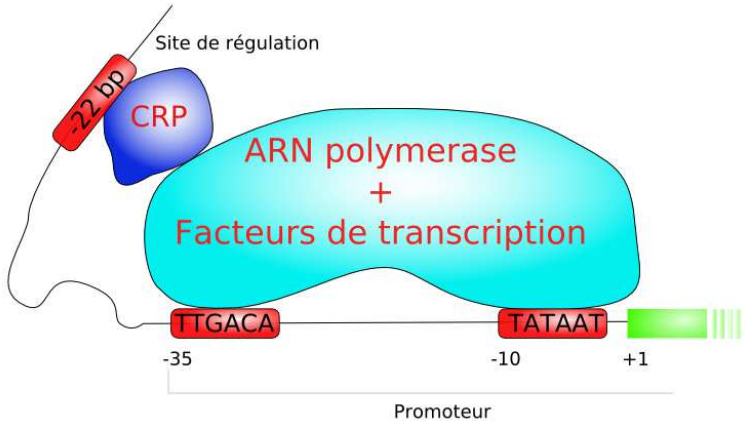


FIG. 1.5 – Facteur de transcription interagissant avec trois éléments fonctionnels pour se fixer à la molécule d'ADN

1.3 Les mutations

Au cours de l'évolution des espèces, des mutations s'opèrent sur le matériel génétique. Celles-ci apparaissent chez certains individus d'une espèce et sont héréditaires. Un individu affecté d'une mutation la transmettra à son éventuelle descendance et ainsi de suite. Les mutations peuvent expliquer en partie l'origine des différenciations entre espèces ainsi que des différenciations entre individus. Les mutations sont classées en deux principaux types :

- Les mutations dites alléliques (aussi appelées mutations ponctuelles). Si nous faisions une analogie avec l'érosion, ce type de mutation pourrait

se comparer à l'érosion lente due à l'effet de l'eau et du vent sur les reliefs. Concrètement, un allèle* est une forme particulière d'un gène par rapport à un autre. Deux allèles d'un même gène dérivent l'un de l'autre par des mutations. Les protéines codées par des allèles d'un même gène seront similaires à quelques caractéristiques près. Les mutations alléliques, fruit de la modification d'un ou de quelques nucléotides, ne sont pas observables par des techniques optiques. La section suivante offre plus de détails sur ce type de mutations.

– Les mutations chromosomiques qui peuvent elles, en terme d'évolution, être comparées à de grands tremblements de terre. Ce type de mutation se traduit par de grands échanges ou réarrangements génétiques au sein des génomes. Ces mutations sont observables au microscope optique. Nous irons plus en détails sur ce point dans la section 1.3.2.

1.3.1 Les mutations alléliques

Le concept de mutation allélique est étroitement lié à la réplication de l'ADN. Nous allons donc nous attarder sur les mécanismes de réplication de l'ADN qui sont à l'origine d'une partie des mutations alléliques.

La réplication de l'ADN

Comme nous l'avons évoqué dans la section 1.1, les cellules ont pour caractéristique d'avoir la capacité de se reproduire. Une cellule est en effet capable de se dupliquer en deux cellules identiques. Les phénomènes mis en jeux sont nombreux et complexes. Entre autres, les molécules d'ADN contenues dans les cellules sont dupliquées par un phénomène appelé la réplication. Comme schématisé dans la figure 1.6, lors de la réplication les deux brins de la molécule d'ADN se séparent, et deux nouvelles molécules identiques sont ainsi créées. Cette réaction nécessite un ensemble d'enzymes* appelé le complexe enzymatique de réplication.

Comme nous l'avons évoqué dans l'introduction de ce chapitre les mécanismes biologiques ne sont jamais fiables à 100 %. Ainsi, durant la réplication de la molécule d'ADN, il arrive que des erreurs dues à des instabilités chimiques surviennent au sein de la suite de nucléotides. Cependant les cellules mettent en oeuvre des mécanismes de réparation qui peuvent restaurer l'information génétique originale. Ces mecanismes peuvent s'exercer après la réplication si des erreurs subsistent. Toutefois, comme la fiabilité des proces-

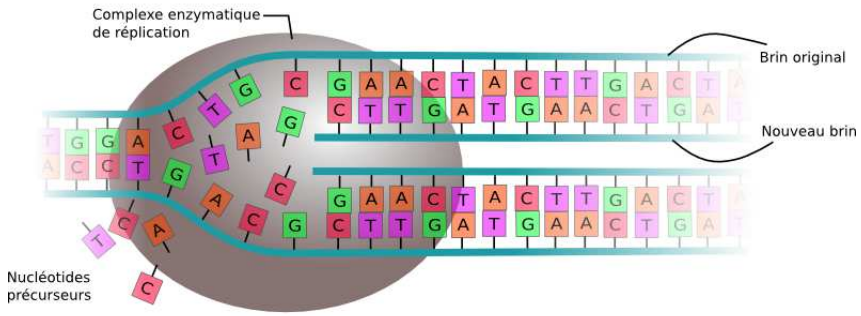


FIG. 1.6 – Schéma de la réplication de l'ADN. Sur la gauche de la figure, une molécule d'ADN s'ouvre pour former deux brins d'ADN. Le complexe enzymatique permet cette séparation. Sur chaque brin séparé viennent s'apparier de nouveaux nucléotides et génèrent ainsi un nouveau brin. Les deux nouvelles molécules d'ADN présentes à droite sur la figure sont alors identiques à la molécule d'origine.

sus n'est jamais garantie, il arrive, soit que ces processus soient insuffisants, soit qu'ils induisent eux mêmes de nouvelles mutations.

Différents types de mutations alléliques

Les mutations alléliques peuvent avoir différentes conséquences sur les séquences d'ADN. Des nucléotides peuvent être modifiés, nous parlons alors de substitution*. D'autre part, un ou plusieurs nucléotides peuvent être supprimés, nous parlons alors de suppression*. Parallèlement, il arrive que des nucléotides soient ajoutés, c'est le cas de l'insertion*. Certaines mutations inversent sur quelques positions l'ordre des nucléotides sur une partie de la séquence d'ADN, il s'agit des inversions. Ces différents types de possibilité de mutations sont représentés dans la table 1.2.

Gène original	Tant va la cruche à l'eau qu'à la fin elle se casse
Substitution	Tant va la cruche à l'eau qu'à la fin elle se lasse
Délétion	Tant va la ruche à l'eau qu'à la fin elle se casse
Insertion	Tant va la cruche à l'eau qu'à la fin elle se laisse
Inversion	Tant va la ehcurc à l'eau qu'à la fin elle se casse
Échange	Tant va la fin à l'eau qu'à la cruche elle se casse

TAB. 1.2 – Exemples de mutations par analogie avec des erreurs dans un texte.

1.3.2 Les mutations chromosomiques

Les mutations chromosomiques concernent un ensemble de chromosomes, des chromosomes entiers ou des parties de chromosomes. Elles résultent de variations dans la structure des chromosomes ou de la variation dans leur nombre. Les mutations chromosomiques, à l'inverse des mutations alléliques, sont observables au microscope optique. Les variations sur la structure des chromosomes sont appelées des remaniements chromosomiques. Il peut s'agir de délétion d'un fragment de chromosome ou bien de duplication d'une partie de l'ADN. De manière plus surprenante, il existe des *transferts horizontaux* qui échangent du matériel génétique entre individus d'espèces distinctes. Il est aujourd'hui reconnu que ces transferts horizontaux influent sur la synthèse protéique [POH$^+$01, GBV02] ainsi qu'avec les facteurs de transcription [LLB$^+$00,ELN$^+$01]. D'autres remaniements, appelés des réarrangements, ne modifient pas la quantité d'ADN. Il peut s'agir d'inversions (une partie de l'ADN est inversée) ou de transposition : une partie de chromosome est déplacée à une autre position du même chromosome. Deux fragments d'un chromosome peuvent échanger mutuellement leur place, nous parlons alors de transposition réciproque.

* *

*

Ce type de mutation peut avoir de nombreuses incidences. Par exemple, une inversion ou une transposition peut couper un gène en deux rendant la transcription de celui-ci infaisable ou incomplète. Il est courant qu'une duplication contienne entièrement un gène. Un gène peut ainsi être intégralement recopié à plusieurs endroits du génome. Pour autant, les copies de ces gènes ne seront pas forcément viables et n'auront que peu de chance d'être exprimées. Toutefois, une des conséquences de ces duplications est que les séquences d'ADN sont en grande partie formées de ce qui est appelé des **éléments répétés**.

D'autres types de mutations chromosomiques sont à l'origine de ce qui est appelé les éléments transposables*. Ces éléments sont des portions d'ADN capables de s'exciser pour se replacer à une position différente (locus*). Les éléments transposables représentent une portion importante des grands génomes :

- 17% du génome de *Arabidopsis thaliana* (plante de la famille des Brassicacées),
- 18% du génome de *Drosophila melanogaster* (petite mouche aux yeux rouges),
- 42% du génome de *Homo sapiens* (humain),
- 50% du génome de *Zea maïs* (maïs).

Certains éléments transposables sont le résultat d'ARNs normalement transcrits via l'ADN (comme présenté en section 1.2), puis rétrotranscrits* en ADN ailleurs dans le génome. Ce type d'éléments transposables est fréquent chez les végétaux, la levure et les mammifères. D'autres types d'éléments transposables résultent de l'excision de fragments d'ADN. La transposition de ceux-ci se fait grâce à des séquences répétées situées aux extrémités de l'élément transposable. Ce type d'élément transposable est courant chez les insectes ou les bactéries.

Ces éléments peuvent avoir de profondes répercussions sur les individus affectés. Ils représentent une grande part des répétitions intra-génomiques décrites dans la section 1.3.4.

Sur deux espèces distinctes, des gènes ayant un ancêtre commun sont appelés des gènes **homologues**. Ces gènes homologues peuvent avoir été différenciés suite à une duplication, il s'appellent alors des gènes **paralogues**. Si leur différenciation s'explique par une spéciation, ils sont alors appelés des gènes orthologues*.

En perturbant l'organisation des génomes, les mutations chromosomiques jouent un rôle important sur l'évolution. Ces mutations peuvent être la source de graves maladies mais peuvent aussi avoir des répercussions positives sur

les individus touchés et ainsi contribuer à la diversification des espèces.

1.3.3 Mutation et évolution

La théorie de l'évolution décrit le processus qui a conduit des premières formes de vie à la quantité foisonnante d'espèces connues aujourd'hui. Charles Darwin base ses principes exprimés dans l'ouvrage l'«Origine des espèces» [Dar59] sur ce qui est appelé **la sélection naturelle**. Depuis, plusieurs théories se sont succédées pour parvenir à une théorie généralement admise. Cependant, celle-ci n'est toujours pas acceptée par tous. Les hypothèses à propos de l'évolution sont nombreuses, parfois contradictoires et certainement incomplètes.

* *

*

Les variations au sein d'une espèce sont dues à des mutations qui apparaissent dans les portions fonctionnelles des génomes. Les mutations apparaissent au hasard, le plus souvent l'organisme atteint est gravement affecté et sa descendance n'est pas assurée si ces mutations se transmettent. Dans ce cas, la mutation n'est pas répercutée sur d'autres individus de l'espèce et n'a pas d'influence sur l'avenir de cette dernière.

Certaines mutations, que nous appellerons «neutres» sont sans conséquence pour l'organisme. Par exemple, les mutations peuvent avoir lieu dans une région non-fonctionnelle de l'ADN et n'ont aucune incidence sur le phénotype de l'individu. D'autre part, une mutation apparaissant en troisième position des codons est souvent sans conséquence. Par exemple, changer CGU pour CGA ne modifie pas l'acide aminé produit par le codon. Une telle mutation est appelée une mutation silencieuse. Les mutations neutres sont sans conséquence pour l'avenir de l'espèce concernée.

Enfin, parfois les mutations modifient la séquence d'acides aminés produite. En conséquence, il est possible qu'un caractère de l'individu affecté soit modifié et se transmette de manière héréditaire. Par la suite, le principe de sélection naturelle implique la conservation ou non de ce caractère. Les mutants défavorisés (dans leur rôle social, ou par rapport à leur environnement), auront tendance soit à mourir plus jeune que leurs congénères, soit à difficilement parvenir à se reproduire. Ces mutants laisseront moins, voire pas de descendance du tout. Les mutants neutres garderont les mêmes capacités

de vie et de reproduction que les autres individus de leur espèce. Les mutations neutres sont aléatoirement distribuées au reste de l'espèce. Enfin, les mutants avantagés résisteront mieux au milieu extérieur et seront plus aptes à se reproduire. Les mutants avantagés auront ainsi tendance à se développer au détriment du reste de la population de l'espèce.

1.3.4 L'ADN et les répétitions

Les travaux présentés dans cette thèse portent sur la recherche de répétitions dans les génomes. Ces répétitions sont l'une des caractéristiques les plus étonnantes de l'ADN, particulièrement chez les eucaryotes. Par exemple, le chromosome Y de l'homme est quasiment intégralement constitué de répétitions.

Nous appellerons *répétitions* les parties similaires (selon un certain degré de similarité) entre génomes ou au sein d'un même génome. L'étude des répétitions représente un des enjeux majeurs des recherches en génomique. Deux types de répétitions sont étudiées, pour des raisons bien différentes. Il s'agit des répétitions internes aux génomes et les répétitions entre génomes.

Les répétitions entre génomes

L'une des conséquences du processus de l'évolution, est que si une mutation néfaste atteint une partie fonctionnelle* du génome, celle-ci n'aura que peu de chances de se transmettre aux générations futures pour les raisons expliquées précédemment. À l'inverse, une mutation affectant une partie non-fonctionnelle du génome n'aura pas d'effet sur la descendance de l'individu touché et cette mutation sera ainsi transmise de générations en générations.

En conséquence, les parties fonctionnelles du génome sont sensibles à ce qui s'appelle la pression de sélection*, à l'inverse des parties non fonctionnelles. Ainsi entre les génomes de deux individus d'une même espèce ou non, les parties fonctionnelles auront tendance à être similaires quand les parties non fonctionnelles pourront, même si cela n'est pas toujours le cas, être très différentes.

Ce dernier point est primordial en génétique. Pour schématiser, nous pouvons résumer cette idée par «Entre deux génomes, l'information importante est similaire alors que les parties *inutiles* sont différentes». Ceci n'est bien entendu qu'une version simpliste de la réalité mais c'est un des fondements des études en génomique. Ces portions similaires entre différents génomes

sont appelées les répétitions entre génomes. Elles sont d'un grand intérêt car elles permettent d'identifier les parties fonctionnelles ainsi que d'inférer de la connaissance. Par exemple, un même gène chez deux individus est en un certain sens une répétition. Si une information est connue sur un gène et que, par l'étude des répétitions, l'orthologue de ce gène est détecté sur le génome d'un autre individu, l'information peut être appliquée à ce nouveau gène.

En outre, l'étude de ces répétitions apporte de nombreuses informations en phylogénie*. En effet, l'étude de la présence ou non de répétitions entre deux individus d'espèces distinctes apporte des informations pour détecter leurs ancêtres communs et d'estimer la date de leur différenciation. Ceci permet ainsi de reconstruire l'arbre phylogénique du vivant. Ernst Haeckel (1834-1919) fut l'un des fondateurs des travaux de construction de l'arbre phylogénique du vivant. En 1879, il proposa dans «*The Evolution of Man*» une version aujourd'hui largement modifiée de l'arbre du vivant axé sur l'homme dont une représentation est visible dans la figure 1.7.

Les répétitions internes aux génomes

À l'inverse des répétitions entre génomes, les répétitions internes aux génomes sont des répétitions apparaissant au sein du génome d'un même individu.

Les éléments transposables (aussi appelé les transposons) se manifestent par de telles répétitions. Le rôle de ces éléments est encore mal connu. Ils sont essentiels dans la préservation de la diversité et ont une fonction importante dans la dynamique des génomes, en particulier pour ce qui est de la régulation. En outre, ils ont un lien possible avec certaines maladies comme le cancer.

L'influence des éléments transposables a, semble-t'il, été sous-estimée. Ainsi, ils font aujourd'hui l'objet de recherches importantes, visant à comprendre leur fonctionnement et leurs conséquences.

* *

*

L'analyse des séquences d'ADN, entre autres de leurs répétitions, est source de connaissances et de compréhension de nombreux phénomènes de la génétique moléculaire. Bien entendu, afin d'être en mesure de les analyser,

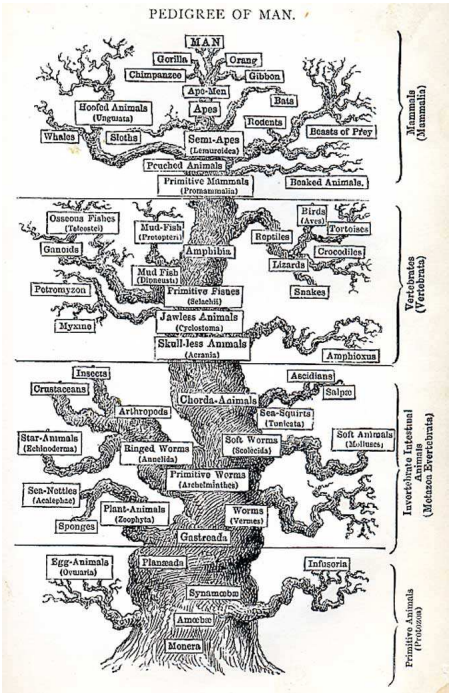


FIG. 1.7 – Un des premiers arbres phylogéniques de l'évolution des espèces, publié en 1879 dans «*The Evolution of Man*» par Ernst Haeckel

il faut disposer des séquences d'ADN sous forme compréhensible par l'humain. Dans la section suivante, nous présentons les techniques utilisées pour séquencer le génome ainsi que divers représentations de celui-ci.

1.4 L'ADN perçu comme un texte

Nous sommes aujourd'hui capable de séquencer les génomes, c'est-à-dire d'en connaître la suite de nucléotides. Les outils utilisés pour le séquençage des génomes se basent sur la technique proposée en 1977 par Sanger [SNC77]. La méthode a été considérablement améliorée depuis cette date, grâce notamment aux séquenceurs automatiques et à la localisation des nucléotides par

des marqueurs fluorescents. Il est aujourd'hui envisageable de séquencer des portions d'ADN de manière rapide et efficace, ceci dans un but, par exemple, judiciaire ou de diagnostique médical.

En raison de leur taille, les molécules d'ADN ne peuvent être séquencées d'un seul bloc. Elles sont donc découpées en morceaux dont la taille varie de 200 à 700 nucléotides afin de pouvoir être techniquement séquençables. Ces fragments d'ADN ainsi séquencés doivent ensuite être assemblés pour reconstruire la molécule originale. L'assemblage se fait grâce au chevauchement entre les différents fragments. En 2001, deux organismes ont publié parallèlement les séquences composant le génome humain. Il s'agit de *Celera* dont l'article est paru dans *Science* [Ven01] et de l'HGP (Human Genome Project) dont le texte paru dans *Nature* [Con01]. À l'image du maïs, d'*Escherichia coli*, d'*Arabidopsis thaliana*, de la Drosophile, ou du chien, bien d'autres génomes ont été séquencés. Une masse gigantesque de données est rendue publique et offre aux généticiens une partie importante du matériel nécessaire à leurs travaux.

Malgré le travail minutieux que représente le séquençage, des erreurs peuvent se produire dans les génomes séquencés. Des erreurs locales de lecture d'un nucléotide peuvent se produire. De plus, lors de l'assemblage des fragments, il est possible que certaines erreurs viennent modifier la suite de nucléotides ainsi identifiée. Ces erreurs peuvent être dues à de mauvais placements de fragments les uns par rapport aux autres, ou à des fragments inversés par rapport à leur sens réel. En effet, l'assemblage est fortement biaisé par la présence d'un grand nombre de répétitions qui peuvent conduire à créer des chevauchements considérés comme valides dans des régions qui ne devraient pas en contenir. Aussi, une portion répétée peut être détectée comme unique lors de l'assemblage des fragments. Enfin, il arrive simplement qu'aucun fragment ne couvre certaines régions du génome qui ne seront donc pas séquencées. Les études basées sur les génomes ainsi séquencés doivent donc prendre en compte ces imperfections dans leurs conclusions.

1.4.1 Représentation de l'ADN

Les travaux informatiques présentés dans cette thèse portent sur la recherche de similarités dans les séquences d'ADN. Ces séquences peuvent se caractériser par la succession de nucléotides qui les composent, mais aussi par diverses caractéristiques physico-chimiques qui leur confèrent leur conformation dans l'espace ainsi que des interactions avec d'autres éléments de la

cellule. Les similarités que nous recherchons pourraient en conséquent être des similarités de forme, physico-chimique ou de suite de nucléotides. Bien que ces trois types de similarités soient intéressantes à étudier, nous nous limiterons à l'étude de similarités de la composition nucléotidique. Ceci est dû, d'une part au fait que l'information manque en ce qui concerne la structure tridimensionnelle et les caractéristiques physico-chimiques de la molécule, et d'autre part au fait que ces deux derniers aspects sont très fortement influencés par la composition nucléotidique. Nous pouvons supposer que des portions similaires d'ADN auront des caractéristiques proches.

Nous travaillerons ainsi uniquement sur la suite de nucléotides des séquences. Par abus de langage, cette suite de nucléotides sera simplement appelée la séquence. La représentation de cette séquence se fait alors naturellement dans un fichier texte. Plusieurs formats sont utilisés pour représenter les séquences et l'un des plus employé est appelé le forma FASTA dont un exemple est donné dans la figure 1.8. Cette représentation offre la possibilité d'être à la fois lisible par l'être humain et utilisable par l'ordinateur.

À certaines positions dans les séquences, des incertitudes peuvent exister. Pour décrire ces incertitudes, l'UIPAC (International Union of Pure and Applied Chemistry) a créé un alphabet présenté dans la figure 1.9.

1.4.2 Utilisation des séquences d'ADN

La plupart des séquences d'ADN sont publiquement disponibles sur internet. Pour se donner une idée de l'importance des travaux de séquençage des génomes, la plus grosse base de données européenne de nucléotides EMBL [CAA+06], contient à ce jour (2006) environ 146 milliards de nucléotides pour 80 millions d'entrées. D'autre part, l'évolution de cette base de données traduit aussi l'intérêt que les scientifiques portent à la question de l'étude des génomes. Comme ceci peut se constater dans la figure 1.10, la taille de la base de données suit une croissance exponentielle depuis 1982.

Notes de lecture.

- Étant donné que les travaux présentés dans cette thèse portent sur la bio-informatique, ils sont utilisés pour traiter des textes représentant des séquences d'ADN. Ainsi, dans la suite de ce manuscrit, nous utiliserons le terme **séquence** pour désigner un texte représentant une séquence d'ADN. De plus, nous utiliserons le terme de **mot** pour désigner une portion continue d'une séquence.

```
>gi|86449942|gb|DQ385482.1| Escherichia coli
GCGAATGCGCGGCCACGAGGTCAACTTCATCTGCG
CCGACGATGCCCACGGTACACCGATCATGCTGAAA
GCTCAGCAGC
>gi|86449914|gb|DQ385468.1| Escherichia coli
GGTCTGCACGGCGTTGGTGTTTCGGTAGTAAACGC
CCTGTCGCAAAAACTGGAGCTGGTTATCCAGCGCG
AGGGTAAAATTCACCGTCAG
>gi|86449946|gb|DQ385484.1| Escherichia coli
GCGAATGCGCGGCCACGAGGTCAACTTCATCTGCG
CCGACGATGCCCACGGCACGCCGATCATGCTTAAA
GCTCAGCA
```

FIG. 1.8 – Un exemple de séquence au format FASTA. Les séquences sont séparées par des commentaires dont les lignes débutent par le caractère '>'. Ces lignes sont utilisées pour y stocker des informations telles la position et la longueur des séquences. Cet exemple contient des séquences d'*E. coli* volontairement tronquées.

– Les séquences étant composées de bases nucléotidiques, l'unité utilisée pour désigner la longueur des séquences est la **base**, simplement notée *b*. De plus, sachant que les séquences sur lesquelles nous travaillons sont de grande taille, nous utiliserons le terme kilobase (noté Kb) pour désigner mille bases. Pour désigner un million de bases nous utiliserons le terme mégabase noté Mb.

Rôle de la bio-informatique

Les généticiens et biologistes ont donc entre leurs mains une formidable quantité de données dont ils cherchent à extraire l'information. Parmi les études réalisées sur ces données, certaines utilisent de petits échantillons de l'ordre d'une centaine de nucléotides et peuvent éventuellement être effectuées directement «à la main» sans avoir recours à l'informatique. À l'inverse, certains travaux comme ceux présentés dans cette thèse portent sur des quan-

Symbole	Association	Commentaires
A		Adénine
C		Cytosine
G		Guanine
T		Thymine
U		Uracyle
R	A, G	Purine
Y	C, T	Pyrimidine
W	A, T	Forte interaction des paires de bases
S	C, G	Faible interaction des paires de bases
M	A, C	
K	G, T	
B	C, G, T	tout sauf A
D	A, G, T	tout sauf C
H	A, C, T	tout sauf G
D	A, C, G	tout sauf T
N	A, C, G, T	n'importe quel symbole

FIG. 1.9 – Alphabet IUPAC des nucléotides permettant de décrire toute incertitude sur les nucléotides.

tités de données beaucoup plus grandes. Supposons, par exemple, que l'on cherche à retrouver dans une séquence de quelques millions de nucléotides un «mot» de taille 10. Ce travail en apparence simple serait de l'ordre de l'impossible à effectuer par une lecture humaine de la séquence. Le même travail, effectué par un ordinateur, d'une part trouvera le résultat sans possibilité d'erreur (à la différence d'un travail effectué par une personne), et d'autre part, sera accompli en quelques secondes seulement. Ainsi, l'informatique et la biologie se sont naturellement rapprochées pour créer la bio-informatique faisant ainsi collaborer les protagonistes de ces deux domaines.

Le domaine de la bio-informatique, qui est très vaste, touche toute forme de collaboration entre les deux domaines. Cette collaboration peut naître de besoins comme la représentation structurale d'éléments biologiques ou l'ana-

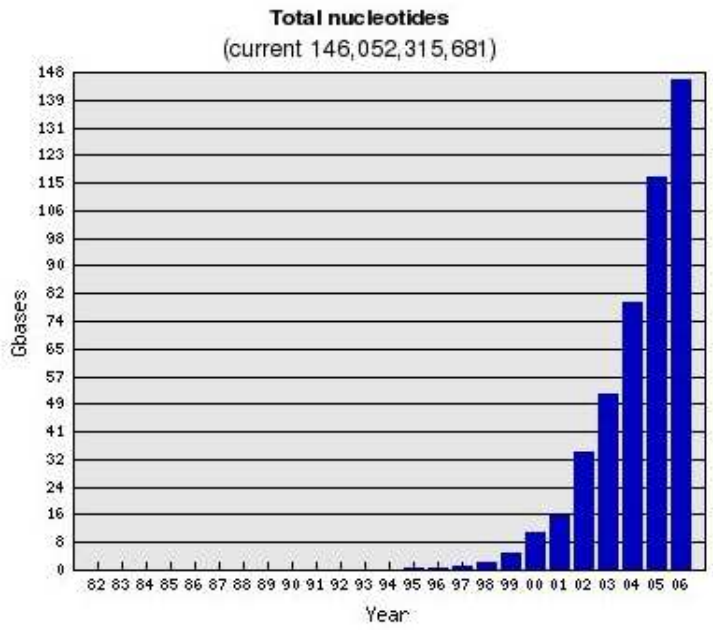


FIG. 1.10 – Évolution de la taille de la base de donnée EMBL. La quantité d'information suit une progression exponentielle depuis 1982. (Graphique généré le 5 Septembre 2006 ©EMBL-EBI `http://www.ebi.ac.uk/embl/index.html`)

lyse d'images par exemple. D'une manière plus précise, lorsque l'attention de la bio-informatique se porte sur l'analyse des séquences génomiques, on parle alors d'**informatique génomique**.

1.4.3 L'alignement de séquences d'ADN

La notion d'alignement de séquences se rapporte à la comparaison de deux ou de plus de deux séquences d'ADN. Cette notion d'alignement de séquences est particulièrement importante dans de nombreux domaines biologiques. En effet, elle permet par exemple de détecter les différences ou les similarités entre les génomes. Cette détection peut être couplée à une notion de score

qui permet de mesurer une distance entre les génomes et ainsi d'estimer une date de divergence entre deux (ou plus de deux) espèces. Une grande partie des travaux effectuées par les généticiens et biologistes se base sur des comparaisons de séquences génétiques et utilise ainsi l'alignement de séquences.

Conclusion

Dans ce chapitre, nous avons introduit les bases de la génomique. Nous avons pu constater que le matériel génétique subit au cours du temps des modifications de différent types. Certaines modifications, appelées mutations alléliques, altèrent localement une séquence d'ADN par des insertions, des suppressions ou des substitutions de nucléotides. D'autres modifications, appelées mutations chromosomiques, concernent de grandes portions du matériel génétique. Ce dernier type de mutations génère de longues portions répétées présentant un intérêt majeur dans l'évolution des espèces ou jouant un rôle dans certaines pathologies.

L'étude des répétitions (ou similarités) dans les séquences d'ADN est l'un des travaux fondamentaux de la génétique. Or, détecter à l'œil nu de telles similarités est proche de l'impossible. L'informatique apporte une solution à ce problème. Cependant, la détection de similarités, lorsque celles-ci sont grandes et fortement dégénérées, est un problème difficile à résoudre, spécialement pour de grandes séquences. De surcroît, un degré de complexité supplémentaire est introduit lors de la recherche de similarités apparaissant plus de deux fois. Or ce type de similarités a une place majeure en biologie moléculaire.

Les travaux présentés dans cette thèse sont axés autour de la recherche de similarités dans les séquences d'ADN. Les similarités auxquelles nous nous intéressons sont des similarités longues et apparaissant un nombre de fois possiblement supérieur à deux. Naturellement, ces travaux sont dédiés à la conception d'algorithmes permettant des alignements multiples de séquences.

Dans le chapitre suivant, nous ferons une introduction de certaines notions d'algorithmique et de complexité. Ensuite nous décrirons les problèmes fondamentaux de l'algorithmique du texte ayant un lien avec les travaux proposés dans cette thèse.

Chapitre 2

Présentation de l'algorithmique du texte

Ce chapitre présente le domaine de l'algorithmique du texte. Nous avons constaté dans la section précédente que les séquences d'ADN peuvent être modélisées sous forme de textes. Les algorithmes* que nous créons et que nous appliquons sont donc naturellement des algorithmes qui utilisent des textes. Ce chapitre présente donc, après une introduction à l'algorithmique, deux principaux défis inhérents aux textes. Il s'agit des problèmes dits de *localisation* et des problèmes dits d'*extraction* dans les textes.

- La localisation consiste, étant donné un «mot» relativement court par rapport à un texte, à retrouver les occurrences approchées ou non de ce mot dans ce texte.
- L'extraction consiste à rechercher des portions de texte sans les connaître *a priori*. Il peut s'agir de la recherche de portions répétées, approximativement ou non, dans un même texte ou dans plusieurs.

Après avoir présenté dans la section 2.2.2 des notions de distances entre textes ainsi que les idées fondamentales pour calculer ces distances, nous présenterons dans les sections 2.2.3 et 2.2.4 les méthodes utilisées pour localiser et pour extraire des mots dans des textes.

* *

*

Afin d'aider les lecteurs peu habitués à la notion d'algorithmique, nous en-

tamons ce chapitre par une section d'introduction à l'algorithmique. Cette section définit les notions d'algorithme, de complexité et d'heuristique. Nous invitons les lecteurs à l'aise avec ces notions à se rendre directement à la section 2.2 page 52.

2.1 Introduction à l'algorithmique

Cette section a pour but de permettre au lecteur peu familier avec le domaine de l'algorithmique d'en comprendre les concepts de base afin de pouvoir assimiler les notions utilisées dans la suite de ce manuscrit.

Nous commencerons par décrire et définir ce qu'est un algorithme avant d'en présenter un bref historique avec les exemples les plus célèbres. Nous verrons ensuite qu'il existe des algorithmes qui peuvent potentiellement «se tromper». Enfin, nous présenterons des notions relatives à ce qui est appelé la complexité d'un algorithme, notions qui servent à estimer le temps et la mémoire nécessaires au déroulement de celui-ci.

2.1.1 Qu'est-ce qu'un algorithme ?

Pour les personnes qui ne sont pas coutumière de l'informatique, la notion d'algorithme est, semble-t'il, perçue comme quelque chose de très complexe. Hors, il n'en est rien. Un algorithme désigne simplement une suite d'instructions à effectuer dans un certain ordre pour obtenir un certain résultat. Par exemple, la suite d'instructions de l'algorithme 1 est un algorithme pour

Algorithme 1 Algorithme pour prendre le beurre dans un réfrigérateur

Nécessite : Réfrigérateur contenant du beurre
Effectue : «prendre le beurre»
1: «ouvrir la porte du réfrigérateur»
2: «prendre le beurre»
3: «fermer la porte du réfrigérateur»
4: **renvoie** «Le beurre»

prendre le beurre dans un réfrigérateur.

Une suite d'instructions simples sans possibilité de *contrôle* limiterait grandement les capacités des algorithmes. Ainsi, il est possible d'utiliser au

sein des algorithmes ce qui est appelé des «structures de contrôle», permettant d'effectuer des opérations sous certaines conditions et de répéter des opérations un certain nombre de fois. Pour être plus exact, notre algorithme d'extraction du beurre du réfrigérateur deviendrait alors l'algorithme 2.

Algorithme 2 Algorithme pour prendre le beurre dans un réfrigérateur, seconde version

Nécessite : Réfrigérateur contenant du beurre
Effectue : «prendre le beurre»
1: **si** «la porte du réfrigérateur est fermée» **alors**
2: «ouvrir la porte du réfrigérateur»
3: **fin si**
4: «prendre le beurre»
5: «fermer la porte du réfrigérateur»
6: **renvoie** «Le beurre»

De même, à l'aide d'opérateurs permettant la répétition d'opérations, nous sommes en mesure de vider l'intégralité du réfrigérateur comme présenté dans l'algorithme 3.

Algorithme 3 Algorithme pour vider un réfrigérateur

Nécessite : Réfrigérateur
1: **si** «la porte du réfrigérateur est fermée» **alors**
2: «ouvrir la porte du réfrigérateur»
3: **fin si**
4: **tant que** «le réfrigérateur n'est pas vide» **faire**
5: «Retirer un élément du réfrigérateur»
6: **fin tant que**
7: «fermer la porte du réfrigérateur»

Ces algorithmes sont en apparence naïfs, mais ils utilisent les structures de contrôles qui permettent, en les couplant à la notion de variable que nous verrons juste après, de créer tous les algorithmes qui composent l'ensemble des programmes que nous utilisons quotidiennement.

Les algorithmes puisent leur utilité et leur puissance dans le fait qu'ils sont capables de manipuler des données sous forme de variables. Ces variables peuvent représenter des nombres, des caractères ou des chaînes de caractères. Parmi les instructions classiques d'un algorithme, il est possible de comparer,

d'assigner, ou de modifier par toute opération d'arithmétique les variables utilisées.

Usuellement un algorithme «prend en entrée» des données et peut proposer une sortie, comme par exemple un nombre.

Algorithme 4 Algorithme de calcul de la somme d'un ensemble d'entiers

Nécessite : Ensemble d'entiers $E = \{E_1, E_2, \ldots, E_n\}$
Effectue : Somme des éléments de E

```
1: somme ⇐ 0
2: pour i de 1 à n faire
3:     somme ⇐ somme + E_i
4: fin pour
5: renvoie somme
```

L'algorithme 4 présente un exemple complet d'une suite d'instructions utilisant une structure pour répéter des opérations (ligne 2) et utilise des variables. Il est à noter (ligne 3) que les assignations de variables se font par convention de la droite vers la gauche. Ainsi à la ligne 3, la valeur de la variable *somme* est changée pour son ancienne valeur à laquelle nous ajoutons l'élément courant de l'ensemble E, à savoir E_i.

* *

*

Nous avons maintenant présenté tous les éléments permettant de créer des algorithmes. Il est intéressant de noter que la plupart des langages permettant d'écrire des algorithmes offrent certaines possibilités supplémentaires, mais celles-ci ne représentent que des conforts d'écriture dans le sens où elles ne sont pas indispensables et peuvent toujours être ramenées aux opérateurs vus précédemment.

Avant d'aller plus loin dans la présentation de l'algorithmique, nous ouvrons une parenthèse pour évoquer son histoire.

2.1.2 Bref historique de l'algorithmique

L'algorithmique a été remise au goût du jour avec l'arrivée de l'informatique, cependant il ne s'agit pas d'une science récente, loin de là. Les premiers algorithmes dont nous ayons retrouvé une description formelle datent

de l'époque des Babyloniens (environs 1800 avant J.-C.) qui les utilisaient pour des calculs appliqués au commerce et aux impôts. Le terme *algorithme* n'apparaît que beaucoup plus tard. Le mathématicien persan Al Kwarizmi (780-850) systématise l'algorithmique dans un ouvrage décrivant des méthodes de calculs algébriques. Au Moyen Âge son nom devient *algorisme* avant que la mathématicienne Ada Lovelace (1815-1852) ne transforme ce nom en *algorithme*, terme utilisé aujourd'hui.

2.1.3 Complexité d'un algorithme

En fonction des problèmes qu'ils résolvent et des données qu'ils ont à traiter, les algorithmes consomment du temps et de la mémoire pour arriver à une solution. Un algorithme est associé à une notion dite de complexité* qui permet d'estimer les ressources nécessaires en fonction des données qu'il a à traiter.

La notation employée pour désigner la complexité d'un algorithme est une fonction «$O(f(n))$», où $f(n)$ est une fonction mathématique de n. La variable n désigne alors la quantité d'informations manipulées par l'algorithme.

Formellement $g = O(f)$ (qui se prononce *g est un grand O de f*)

$$\text{si } \exists\ M \in \mathbb{N},\ K \in \mathbb{R} \mid \forall\ n \geq M,\ g(n) \leq K \times f(n)$$

La fonction $f(n)$ peut être n'importe quelle fonction mathématique, les plus courantes sont les suivantes,

- $O(1)$: complexité constante,
- $O(log(n))$: complexité logarithmique,
- $O(n)$: complexité linéaire,
- $O(n^2)$: complexité quadratique,
- $O(n^3)$: complexité cubique,
- $O(n^c)$: complexité polynomiale (avec $c > 1$ constant),
- $O(c^n)$: complexité exponentielle (avec $c > 1$ constant),
- $O(n!)$: complexité factorielle.

Par exemple un algorithme de complexité $O(1)$ désigne un algorithme dont le temps d'exécution ne dépend pas de la quantité de données à traiter. Notre premier exemple d'algorithme («ouvrir la porte du réfrigérateur» - «prendre le beurre» - «fermer la porte du réfrigérateur») est un exemple d'algorithme de complexité $O(1)$. L'algorithme 4 de calcul de somme est, quant à lui, un algorithme de complexité $O(n)$ (dit linéaire) car, pour effectuer son calcul, il doit prendre en compte un par un tous les éléments de la somme.

La connaissance de la complexité d'un algorithme permet alors d'estimer le temps nécessaire pour effectuer un calcul en fonction de la taille des données à traiter. Par exemple, sur un ordinateur usuel capable d'effectuer 10^9 opérations par seconde, traiter plus de 90 données par un algorithme de complexité $O(2^n)$ prendrait plus de temps que celui écoulé depuis la création de notre système solaire ! Inutile alors d'espérer pouvoir utiliser ce type d'algorithme sur de grands ensembles de données, et ce quel que soit le type de machine utilisé.

La notion de complexité apporte une indication sur l'utilisation de ressources des algorithmes en fonction de la quantité de données à traiter. Cependant, il faut se méfier de cette notion car elle peut induire en erreur dans certains cas. Par exemple, il peut arriver qu'un algorithme A, ayant une complexité en temps en $O(n)$, s'exécute moins rapidement sur une certaine plage de valeurs qu'un algorithme B en $O(n^2)$. Admettons, par exemple, que A effectue 10000 opérations n fois, alors que l'algorithme B effectue une opération n^2 fois. Dans ce cas, pour des problèmes comprenant entre 0 et 10000 éléments, l'algorithme B, pourtant en $O(n^2)$, est plus rapide que l'algorithme A. L'analyse de la complexité ne permet donc pas de décider à elle seule de l'utilisation d'un algorithme plutôt que d'un autre pour résoudre un même problème. Elle ne donne qu'une idée des comportements asymptotiques des solutions utilisées.

Le tableau 2.1 permet d'estimer en fonction de la quantité de données à traiter et de la complexité de l'algorithme le temps nécessaire à l'exécution d'un calcul. Nous pouvons noter grâce à ce tableau que des algorithmes de complexité $O(\log n)$ ou $O(n)$ peuvent traiter en un temps *raisonnable*[1] jusqu'à un milliard de données. Nous observons aussi qu'un algorithme de complexité quadratique peut traiter raisonnablement des données jusqu'à l'ordre du million, mais qu'il lui faut plusieurs années pour traiter jusqu'à un milliard d'informations. Enfin, il est clair qu'un algorithme de complexité $O(n^3)$ n'est applicable que sur de petits jeux de données et ne peut être utilisé pour la comparaison de génomes par exemple.

[1] Nous employons le terme raisonnable pour désigner un algorithme dont le temps d'exécution est acceptable par l'utilisateur. En fonction des enjeux et des autres méthodes existantes, un temps d'exécution raisonnable pour un algorithme varie de quelques minutes à quelques jours.

	$O(\log n)$	$O(n)$	$O(n^2)$	$O(n^3)$
1000	10^{-8} sec.	10^{-6} sec.	10^{-3} sec.	1 sec.
10^6	2.10^{-8} sec.	10^{-3} sec.	10^3 sec.	10^9 sec. $\approx$ 32 ans
10^9	3.10^{-8} sec.	1 sec.	10^9 sec. $\approx$ 32 ans	10^{18} sec. $\approx 32.10^9$ ans

TAB. 2.1 – Une idée du temps d'exécution utilisé par un ordinateur pouvant traiter 10^9 informations par seconde en fonction de la complexité des algorithmes et de la quantité de données.
La colonne de gauche représente la quantité de données à traiter par des algorithmes dont la complexité est indiquée sur la première ligne.

2.1.4 Les algorithmes peuvent-ils se tromper ?

Il est courant que la seule solution existante pour résoudre certains problèmes possède une complexité limitant fortement la quantité de données traitable. Un exemple célèbre est le problème dit du «Parcours du voyageur de commerce» (PVC). Un voyageur doit se rendre dans un ensemble de n villes. Chaque ville est reliée à chacune des autres par une route. Le problème est de trouver le chemin passant par toutes les villes et faisant parcourir le moins de distance possible au voyageur. Si l'on cherche à trouver la solution exacte il faut tester tous les chemins possibles, c'est-à-dire $(n-1)!$ possibilités. En effet, en partant de sa première ville, le voyageur a le choix entre $n-1$ chemins correspondant à $n-1$ villes. Une fois dans la seconde ville, le voyageur peut choisir parmi $n-2$ villes, et ainsi de suite. Ainsi le nombre total de chemins à envisager pour trouver la (les) meilleure(s) solution(s) est $(n-1) \times (n-2) \times \cdots \times 2 \times 1 = (n-1)!$. La complexité de la résolution exacte du PVC est donc en $O(n!)$, c'est pourquoi, il est impossible de chercher à résoudre ce problème de manière exacte pour beaucoup de villes. À titre d'exemple, sur un ordinateur capable de traiter 10^9 opérations par secondes, résoudre ce problème pour $n = 20$ villes, nécessiterait environ 1800 années ce qui n'est évidemment pas envisageable.

Pour résoudre les problèmes dont la solution exacte nécessite trop de temps, il existe des solutions qui calculent des résultats approchés en un temps largement inférieur au temps nécessaire au calcul de la solution exacte.

Ces algorithmes sont appelés des heuristiques*. L'utilisation d'une heuristique bien choisie peut permettre de diminuer la classe de complexité d'un algorithme (par exemple d'exponentielle à polynomiale). Les heuristiques ne garantissent pas en général la qualité des solutions qu'elles proposent, même si parfois il est possible de garantir l'erreur induite par l'heuristique utilisée.

Le PVC possède beaucoup d'heuristiques, l'une des plus simples est appelée «le plus proche voisin». À chaque étape, il s'agit de choisir la ville non visitée la plus proche. Ceci ne garantit pas de trouver une meilleure solution mais son calcul en $O(n)$ est extrêmement rapide comparé au calcul de la solution exacte en $O(n!)$. De plus, cette solution ne permet pas d'estimer la qualité du résultat trouvé.

* *

*

Les notions d'algorithme, de complexité et d'heuristique étant maintenant définies, nous proposons dans les sections suivantes une présentation des principaux algorithmes appliqués aux textes.

2.2 Algorithmes pour la comparaison de textes

Dans cette section, nous abordons les principaux problèmes soulevés par l'algorithmique du texte. Les applications de ce domaine de recherche sont utilisées tous les jours par des millions de personnes sans même qu'ils ne s'en rendent compte. Les correcteurs orthographiques automatiques ou les moteurs de recherche sur internet en sont de bons exemples. De manière moins évidente, les algorithmes de reconnaissance d'empreintes digitales ou les algorithmes fusionnant plusieurs photos pour créer un panorama utilisent des algorithmes s'apparentant à des algorithmes sur des textes.

L'une des problématiques courante de l'algorithmique du texte consiste à retrouver de l'information dans un texte ou entre plusieurs textes. Ceci est peut-être l'une des difficultés mobilisant le plus d'énergie de recherche. Ceci est particulièrement vrai dans le domaine de la bio-informatique.

Retrouver de l'information se traduit usuellement par comparer des textes pour y trouver des portions similaires. Deux principaux types d'algorithmes du texte naissent alors :

- Les algorithmes qui, étant donné un mot comparativement court par rapport à un texte ou un ensemble de textes, donnent les positions où ce mot apparaît. Ces algorithmes, que nous appellerons «algorithmes de recherche sans erreurs», sont peu utilisés pour la bio-informatique.
  Les algorithmes de recherche sans erreur sont utilisés par exemple dans les éditeurs de texte pour retrouver un mot donné. Ces algorithmes n'ont cessé d'évoluer depuis près de 30 ans pour limiter au maximum, et souvent de manière très élégante, la complexité de recherche d'un mot dans un texte non indexé. Les recherches effectuées dans cette thèse ne vont pas dans la direction de la recherche exacte, nous ne citerons donc ici qu'un des articles précurseurs de ce type de problèmes, il s'agit de l'algorithme de *Knuth, Morris, Pratt* [KMP77] qui fixe les bases des méthodes de recherches de mots. Un livre de référence de Lecroq et Charras [CL04] propose un examen exhaustif des algorithmes de recherche sans erreur.
- Les algorithmes qui recherchent des similarités entre des textes, ou qui recherchent les occurrences approchées d'un mot dans un texte. Ce type de problème se décline en de très nombreuses variantes que nous expliciterons dans les sections 2.2.3 et 2.2.4.
  Les thèmes abordés dans cette thèse se tournent prioritairement vers les algorithmes adaptés à la recherche avec erreurs. Le but est alors de trouver, comme présenté dans la section 2.2.3, des occurrences de mots qui *ressemblent* à un mot recherché, ou bien de trouver dans un texte des mots qui se *ressemblent*, sans connaître ces mots à l'avance. Ceci sera présenté dans la section 2.2.4.
  Notons que le problème inverse existe : plutôt que de rechercher des portions de textes similaires, un article de Haubold [HPMW05] propose de rechercher les plus petites portions de texte non répétées.

À présent, nous introduisons quelques notations et définitions relatives à l'algorithmique du texte. Ces notations seront utilisées tout au long de ce manuscrit.

2.2.1 Définitions préliminaires

Définition 1. *Texte, alphabet, facteur*

Un **texte** *est une séquence de 0 caractère ou plus. Ces caractères appartiennent à un* **alphabet** *noté* Σ. *Un texte* s *de longueur* n *sur* Σ *est représenté par* $s[0]s[1]\dots s[n-1]$ *avec* $s[i] \in \Sigma$ *pour* $0 \leq i < n$. *La longueur d'un texte*

s est représentée par $|s|$. La notation $s \in \Sigma^*$ désigne un texte s sur Σ avec $|s| \geq 0$.

La notation $s[i,j]$ avec $i,\ j\ \in [0, n-1], i \leq j$, représente la suite de caractères $s[i]s[i+1]\dots s[j]$ de s. Nous nommons alors $s[i,j]$ un **facteur** *de s et nous disons que $s[i,j]$ apparaît à la position i dans s.*

Pour faciliter la lecture de ce manuscrit, un facteur sera aussi appelé un «*mot*». Cependant, dans nos travaux, la notion de «mot» ne respecte pas la notion usuelle car ceux-ci ne sont pas nécessairement des entités délimitées par des caractères particuliers comme des espaces ou des signes de ponctuation, comme ceci est le cas dans le langage courant.

D'autres notations et définitions relatives aux textes apparaîtront au fur et à mesure dans le manuscrit où elles seront nécessaires, ceci dans le but de laisser au lecteur le temps d'assimiler ces notations.

* *

*

La section suivante propose une formalisation de la notion de ressemblance entre des mots, ou bien de textes entiers.

2.2.2 Calculer la distance entre deux textes

La notion de ressemblance n'est pas une notion formelle. Sur quoi se baser pour affirmer que deux choses sont similaires ? Un objet peut ressembler à un autre selon différents critères, sa masse, son aspect, son odeur, sa forme... La notion de ressemblance est de surcroît une notion qui n'a rien d'intrinsèque, des objets pourront paraître similaires à certaines personnes et parfaitement distincts pour d'autres.

Même lorsqu'il s'agit de textes, portant composés de données discrètes, il est nécessaire de définir formellement la notion de distance permettant ainsi de décider si deux mots se ressemblent ou non.

Définition 2. *Distance*

Une distance sur un ensemble E est une application

$$\mathcal{D} : E \times E \to \mathbb{R}^+$$

telle que :

– $\forall x, y \in E : \mathcal{D}(x, y) = \mathcal{D}(y, x)$ *(symétrie) ;*
– $\forall x, y \in E : \mathcal{D}(x, y) = 0 \Leftrightarrow x = y$ *(séparation) ;*
– $\forall x, y, z \in E : \mathcal{D}(x, z) \leq \mathcal{D}(x, y) + \mathcal{D}(y, z)$ *(inégalité triangulaire).*

Dans le cadre de l'algorithmique du texte, deux distances sont principalement utilisées. Il s'agit de la distance dite de Hamming et de la distance d'édition que nous allons présenter dans les deux prochaines sections.

La distance de Hamming

La distance de Hamming* est une distance très simple. Entre deux mots de même longueur, la distance de Hamming est le nombre de substitutions de caractères nécessaire pour transformer le premier mot en le second. La figure 2.1 présente un exemple de distance de Hamming entre deux mots.

FIG. 2.1 – Exemple de distance de Hamming : pour transformer le mot $u = ATATAGTTAGTGC$ en $v = ATCTATTCAGTAC$ il faut effectuer 4 substitutions. Dans ce cas, on note $\mathcal{D}_h(u, v) = 4$.

Définition 3. *Distance de Hamming*
Soient u et $v \in \Sigma^*$ tels que $|u| = |v| = n$.
Soit $f : \Sigma \times \Sigma \rightarrow \{0, 1\}$ telle que

$$f(\alpha, \beta) = \begin{cases} 0 & si\ \alpha = \beta \\ 1 & sinon \end{cases}$$

On définit la distance de Hamming entre u et v par :

$$\mathcal{D}_h(u, v) = \sum_{i=0}^{n-1} f(u[i], v[i])$$

L'algorithme de calcul de la distance de Hamming entre deux textes u et v de longueur n se fait en temps $O(n)$. Il suffit de compter et d'additionner le nombre de positions où les caractères composant ces deux textes diffèrent.

La distance d'édition

La distance d'édition* a été introduite par Levenshtein [Lev66]. Elle est ainsi aussi appelée distance de Levenshtein. Cette distance ajoute à la distance de Hamming la possibilité de prendre en compte, lors de la comparaison de deux textes, des insertions et des délétions en plus des substitutions. La figure 2.2 présente un exemple de distance d'édition entre deux mots.

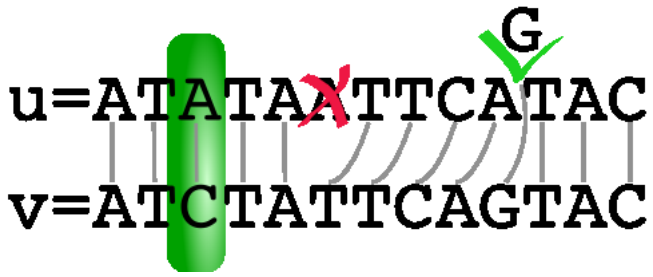


FIG. 2.2 – Exemple de distance d'édition : pour transformer le mot $u = ATATATTCATAC$ en $v = ATCTATTCAGTAC$ il faut effectuer une substitution (matérialisée par le rectangle coloré), une suppression (caractère 'A' barré), et une insertion (caractère 'G') dans u. Dans ce cas, nous notons $\mathcal{D}_e(u, v) = 3$.

Notation 1. *Distance d'édition*
Soient $u, v \in \Sigma^*$, nous notons $\mathcal{D}_e(u, v)$ la distance d'édition entre u et v.

Le calcul de la distance d'édition entre deux mots est moins aisé que le calcul de la distance de Hamming. Pour calculer la distance d'édition entre deux mots, nous avons recours à des algorithmes de programmation dynamique*. Needleman et Wunsch proposent en 1970 un tel algorithme [NW70], alors

qu'en 1981 Smith et Waterman modifient cet algorithme dans un second article de référence [SW81].

Supposons que l'on cherche à calculer la distance d'édition entre les mots u avec v. Ce calcul se fait via l'utilisation d'une matrice F, de taille $|u| \times |v|$, appelée matrice de programmation dynamique. Chaque case de la matrice correspond à un couple de positions (i, j) dans u et v et contient la distance entre $u[0, i]$ et $v[0, j]$. La case la plus *sud-est* de la matrice contient la distance d'édition entre u et v. Le calcul d'une case de la matrice dépend des valeurs des trois cases *nord*, *ouest* et *nord-ouest* qui l'entourent. Le calcul de chaque case $F_{i,j},\ i < |u|,\ j < |v|$ se fait de la manière suivante :

$$F_{i,j} = minimum \begin{cases} F_{i-1,j-1} + \begin{cases} 0 & \text{Si } u[i-1] = v[j-1] \quad \text{correspondance} \\ 1 & \text{Sinon} \qquad\qquad\qquad\quad \text{substitution} \end{cases} \\ F_{i-1,j} + 1 & \text{délétion} \\ F_{i,j-1} + 1 & \text{insertion} \end{cases} \tag{2.1}$$

Nous pouvons noter qu'en modifiant les constantes (ici 0 ou 1) du calcul précédent, il est possible de pénaliser plus ou moins certaines opérations comme la substitution par exemple. De même, plutôt que de calculer une distance en pénalisant les erreurs, il est possible d'attribuer des points aux similarités et calculer ainsi un score de similarité*. Dans ce cas, nous ne chercherons plus un «minimum» mais un «maximum». Cette méthode, permettant le calcul d'un score de similarité plutôt qu'une distance, est couramment utilisée et sera reprise en détail dans la section 2.2.4.

Le score de chaque case est déterminé selon le type de mutation qui a conduit à son calcul. En se basant sur le mot placé horizontalement dans la matrice, il est obtenu à partir de la case supérieure s'il s'agit d'une insertion, de la case gauche s'il s'agit d'une suppression et enfin de la case diagonale supérieure gauche s'il s'agit d'une correspondance ou d'une substitution. Ainsi, une fois le contenu de la matrice calculé, en partant de la case la plus sud-est il est possible de retrouver le(s) chemin(s) originel(s) de la case la plus nord ouest et qui a conduit au calcul de la distance. Ce parcours s'appelle le «back tracking».

La table 2.2 présente un exemple de calcul de distance par programmation dynamique.

L'algorithme de calcul de distance d'édition construit la matrice de programmation dynamique. Sa complexité pour deux mots de longueur n est

		A	T	T	G	A
	0	1	2	3	4	5
A	1	↖ **0**	↖1	←2	←3	←4
T	2	↑1	↖ **0**	←1	←2	←3
C	3	↑2	↑1	↖**1**	←2	←3
C	4	↑3	↑2	↑**2**	↖2	←3
G	5	↑4	↑3	↑3	↖**2**	↖3
A	6	↑5	↑4	↑4	↑3	↖**2**

TAB. 2.2 – Un exemple de matrice de programmation dynamique. Les mots comparés sont *ATTGA* et *ATCCGA*.
Dans chaque case, une flèche présente une possibilité (car il en existe plusieurs) de provenance du calcul. Une flèche oblique ↖ correspond à deux caractères égaux ou à une substitution. Par rapport au mot horizontal (*ATTGA*), une flèche horizontale ← correspond à une insertion alors qu'une flèche verticale ↑ correspond à une suppression. Comme ceci peut se constater dans la case la plus en bas à droite de la matrice, la distance entre ces deux mots est 2. Telles que les flèches sont proposées ici, la transformation de *ATTGA* à *ATCCGA* par une substitution du troisième *T* et *C*, *ATTGA* devient *ATCGA*. Ensuite, une insertion d'un *C* en quatrième position transforme *ATCGA* en *ATCCGA*.

donc en $O(n^2)$. Dans [CLZU02], Crochemore *et al.* ont proposé un algorithme sub-quadratique, en $O(\frac{hn^2}{\log n})$, où $h < 1$ est une variable représentant l'entropie du texte, basé sur la compression des textes pour résoudre ce même calcul.

* *

*

D'autres distances peuvent être définies et sont utilisées en particulier pour la comparaison de textes représentant des séquences génomiques. D'autres types d'erreurs peuvent également être pris en compte comme l'inversion ou l'amplification de caractères.

Certains calculs de distances se basent sur le taux d'apparition de chaque caractère. Dans ce cas, l'utilisation du terme *distance entre mots* est un abus de langage car deux mots peuvent avoir une distance nulle tout en étant différents. Par exemple, les mots CHIEN et NICHE sont constitués des mêmes caractères. Leur distance basée sur les taux d'apparition des lettres est donc 0, bien que ces deux mots ne soient pas identiques.

* *

*

À l'aide des distances précédemment introduites, nous nous sommes libérés de l'exactitude et nous sommes capables de considérer un mot comme étant une occurrence approchée d'un autre mot. Ceci est d'une importance capitale dans tous les domaines de l'algorithmique du texte où les données sont bruitées. Nombre de défis posés par la bio-informatique s'inscrivent précisément dans l'un de ces domaines et la quasi totalité des problèmes abordés doivent permettre cette flexibilité.

Deux activités inhérentes notamment à la bio-informatique sont la recherche d'éléments approchés et l'extraction de répétitions dans un ou plusieurs texte(s). Nous allons dans les deux sections suivantes décrire ces problèmes et les moyens de les résoudre.

2.2.3 Rechercher des éléments dans un texte - Le Pattern Matching

Ce problème consiste à retrouver dans un jeux de données toutes les occurrences d'un modèle pris en paramètre. Ce problème est communément connu sous son nom anglais de «Pattern Matching». Le modèle que l'on cherche est appelée le *pattern*. Le pattern peut être simplement un mot dont nous recherchons des occurrences dans le texte. Les occurrences recherchées peuvent être approchées ou exactes. Dans ce dernier cas, il s'agit du problème de recherche sans erreur que nous avons évoqué dans l'introduction de la section 2.2. Le pattern peut aussi être une expression rationnelle et peut contenir des variables. Par exemple le pattern $ATTATA[5-12]TGAG[AT]$ peut être utilisé pour retrouver les occurrences débutant par le mot $ATTATA$ suivi d'un espace comprit entre 5 et 12 caractères pour être finalement terminé

par le mot $TGAGA$ ou $TGAGT$. Dans ce cas, le pattern sera appelé un motif structurés* et le problème s'appelle naturellement la recherche de motifs structurés.

La recherche de motifs structuré est un problème très différent du cas où l'on cherche «simplement» une occurrence approchée d'un mot. Nous allons donc traiter séparément le cas de la recherche approchée d'un mot du cas de la recherche de motifs.

Recherche approchée d'un mot

Nous allons nous focaliser sur le problème suivant. Soient $P, T \in \Sigma^*$, avec $|P| < |T|$. Le but est de trouver l'ensemble
$\{i \in [0, |T| - |P|] \mid \mathcal{D}_e(P, T[i, i + |P| - 1]) < d\}$, avec $d \in \mathbb{N}$, distance maximale entre le pattern et ses occurrences dans le texte.

Une catégorie d'algorithmes de recherche approchée utilise le «*Neighbor Joining*» [Mye94] qui consiste à générer tous les mots m tels que $\mathcal{D}_e(P, m) < d$ et à rechercher dans le texte les occurrences exactes des mots m.

Ce type de problème peut lui aussi être résolu grâce à l'utilisation de la programmation dynamique. L'idée est de calculer la distance entre le pattern et le texte comme présenté dans la table 2.2. La différence réside dans le fait que chaque position du texte est considérée comme un début possible du pattern. Ceci se traduit par le fait que la première ligne de la matrice de programmation dynamique est alors initialisée avec des zéros (dans le cas où texte est placé horizontalement, si celui-ci est placé verticalement, la première colonne est initialisée avec des zéros). La dernière ligne contient dans chacune de ses cases une distance entre le pattern et une partie du texte. À chaque fois qu'une de ces cases contient une distance inférieure à d, ceci signifie qu'une occurrence approchée du pattern a été trouvée dans le texte. Il faut alors appliquer le *back tracking* pour connaître la ou les position(s) correspondante(s).

Un exemple d'utilisation de matrice de programmation dynamique pour résoudre ce problème est donné dans la figure 2.3.

Lors du calcul de la matrice de programmation dynamique, il n'est pas nécessaire de prendre en compte les cases dont le score dépasse la distance maximale voulue. Le temps de calcul de l'algorithme peut ainsi être amélioré en supprimant le calcul de certaines portions de la matrice. Cependant, la complexité théorique de cet algorithme est en $O(|T| \times |P|)$.

Position		0	1	2	3	4	5	6	7
Texte		A	T	T	G	A	G	T	G
Pat.	0	0	0	0	**0**	0	0	0	0
A	↑1	↖0	↖1	↖1	↖1	↖**0**	↖1	↖1	↖1
T	↑2	↑1	↖0	↖1	↖2	↑1	↖**1**	↖1	↖2
T	↑3	↑2	↑1	↖0	←1	↑2	↖2	↖**1**	↖2
G	↑4	↑3	↑2	↑1	↖0	←1	↖2	↑2	↖**1**

TAB. 2.3 – Utilisation d'une matrice de programmation dynamique pour effectuer une recherche approchée des occurrences d'un mot.
Le but est de trouver l'ensemble I des occurrences de $P = ATTG$ dans $T = ATTGAGTG$ avec $\mathcal{D}_e(P, T[i, i + |P| - 1]) < d \ \forall i \in I$. Nous pouvons constater que le pattern apparaît à quatre reprises dans le texte à des positions se terminant aux emplacements 2, 3, 4, et 7. Le back tracking pour la position 7 est représenté par des caractères gras, montrant que le pattern $ATTG$ a subit une substitution pour être transformé en $AGTG$ correspondant ainsi à $T[4, 7]$.

Pattern avec expression rationnelle

La recherche dans un texte d'un pattern contenant une expression rationnelle est beaucoup plus difficile à effectuer que la recherche d'un pattern sous une certaine distance uniquement. Pour résoudre ce problème, il faut utiliser des structures de données dédiées, et avoir recours à des algorithmes adaptés au type de pattern recherché.

Les travaux présentés dans cette thèse ont conduit à la création d'une structure de données appelée le tableau des bi-facteurs. Cette structure permet d'indexer un texte et d'y retrouver rapidement les occurrences d'un pattern comprenant un mot de longueur fixée, suivi d'un nombre fixé de caractères sans importance suivis d'un autre mot de longueur fixée. Cette structure de données, dont le but n'est pas spécifiquement de répondre au problème du pattern matching est présentée dans la section 4.4 page 120.

D'autres algorithmes sont adaptés à la recherche pour de nombreux types de patterns différents. Nous invitons les lecteurs curieux à consulter les

travaux de [PCGS04] ainsi que le livre de référence de Navarro et Raffinot [NR02].

* *

*

La section suivante présente différentes méthodes d'extraction de répétitions dans les textes. Ce sont précisément à ces problèmes que les méthodes de filtrage présentées dans ce manuscrit sont destinées.

2.2.4 Extraction de répétitions

Le problème de l'extraction de répétitions consiste à retrouver dans un texte ou un ensemble de textes des occurrences d'un modèle qui se trouvent répétés selon certains critères. Ces critères peuvent être très différents en fonction des besoins :

- Les répétitions recherchées peuvent prendre différents aspects :
  - Il peut s'agir de répétitions simples, comme par exemple rechercher entres plusieurs textes des portions similaires selon certains critères de similarités.
  - Les répétitions recherchées peuvent être composées d'un ensemble de répétitions simples, séparées par des distances contraintes ou non.
  - Un type de répétition étudié spécialement en biologie moléculaire fait l'objet de nombreux travaux. Il s'agit des répétitions dites «en tandem». Ces répétitions sont constituées de mots adjacents similaires concernant leur longueur et leur composition. Les travaux de cette thèse ne s'inscrivent pas dans la recherche de ce type de répétition. L'étude des répétitions en tandem est importante en biologie moléculaire. Ainsi, les outils de détection de ce type de répétitions sont nombreux [Ben97, Ben99, KBK03].
- Les conditions de répétitions elles-mêmes peuvent être très différentes selon les problèmes :
  - Les répétitions peuvent être recherchées dans un texte, ou bien dans un ensemble de textes.
  - Le nombre de répétitions recherchées est lui-même variable. Les problèmes visant à rechercher des entités répétées deux fois (dans un texte, ou réparties dans deux textes) ont des solutions souvent très

différentes de ceux cherchant des entités répétées plus de deux fois (ou dans plus de deux textes).

Ainsi une multitude de problèmes d'extraction se côtoient et, malgré les apparentes ressemblances dans les énoncés des problèmes, les solutions algorithmiques adaptées sont potentiellement très différentes.

Nous proposons à présent une présentation de problèmes d'extraction de répétitions ainsi que les solutions adoptées. L'une des méthodes courantes pour l'extraction de répétitions simples consiste à utiliser la programmation dynamique. La section suivante présente cette méthode.

Alignement de textes

L'alignement est une méthode couramment utilisée pour déterminer un score de similarité de ressemblance entre objets textuels. La programmation dynamique, telle qu'elle est décrite dans la section 2.2.2 peut être utilisée à ces fins.

Alignement global. Nous présentons maintenant l'utilisation de la programmation dynamique dans le but de trouver un score d'alignement entre textes. L'article de référence de Needleman et Wunsch [NW70] propose cette technique.

Les relations 2.1 entre les cases de la matrice de programmation dynamique présentée à la section 2.2.2 peuvent être modifiées dans ce but :

$$F_{i,j} = maximum \begin{cases} F_{i-1,j-1} + S(u[i-1], v[i-1]) & \text{alignement} \\ F_{i-1,j} + d & \text{délétion} \\ F_{i,j-1} + i & \text{insertion} \end{cases} \quad (2.2)$$

Pour calculer une mesure de similarité entre textes, il faut alors pénaliser les insertions et délétions en assignant des valeurs négatives à i et d. La valeur de $S(u[i-1], v[i-1])$ doit permettre de valoriser des correspondances entre caractères tout en pénalisant les substitutions. Il est possible de définir une relation entre les caractères alignés pour favoriser plus ou moins certaines relations. Un exemple de ce type de score est donné dans la matrice 2.4 pour un alphabet $\Sigma = \{A, C, G, T\}$ à quatre lettres.

Les alignements obtenus par l'application des relations 2.2 permettent d'obtenir une mesure de similarité entre deux textes. Ces relations peuvent

	-	A	C	G	T
A	1	-3	-5	-3	
C	-	1	-5	-5	
G	-	-	1	-3	
T	-	-	-	1	

TAB. 2.4 – Cette matrice propose une score pour l'alignement de chaque paire de caractères sur l'alphabet $\Sigma = \{A, C, G, T\}$.
Remarquons que les scores d'alignement sont négatifs pour les substitutions (par exemple $S(T, C) = -5$) alors que lorsqu'il s'agit d'une correspondance, les scores sont positifs (par exemple $S(C, C) = 1$).

être modifiées pour l'obtention de ce qui est appelé un alignement local permettant de retrouver des similarités locales entre deux textes.

Alignement local. L'alignement local de textes permet, étant donnés deux textes (assimilés à des séquences dans le cas de l'ADN), de retrouver parmi eux des portions similaires tout en utilisant la programmation dynamique. Les idées de base sont données dans l'article de Smith et Waterman [SW81]. D'autres outils comme FASTA [LP88] ou BLAST [AGM+90] sont utilisés massivement pour le calcul d'alignement locaux.

L'idée principale de cette méthode est de ne conserver dans la matrice que les valeurs supérieures à zéro. Lorsque le score d'alignement passe sous ce seuil, il est réinitialisé à zéro. Ainsi, toute portion similaire entre deux textes est détectée, et ce sans être influencée par les portions précédentes. Les relations 2.2 sont modifiées comme suit pour prendre en compte cette idée.

$$F_{i,j} = maximum \begin{cases} F_{i-1,j-1} + S(u[i-1], v[i-1]) & \text{alignement} \\ F_{i-1,j} + d & \text{délétion} \\ F_{i,j-1} + i & \text{insertion} \\ 0 & \text{réinitialisation} \end{cases} \quad (2.3)$$

Afin de trouver toutes les occurrences de portions de texte dont le score de similarité est supérieur à un seuil t fixé, il faut, lors de la création de la matrice de programmation dynamique grâce aux relations précédentes, identifier les cases dont le score est supérieur à t.

Un exemple de matrice de programmation dynamique pour l'alignement local est donné dans la table 2.5.

		position	0	1	2	3	4
		texte u	A	T	T	G	A
pos.	texte v	0	-2 → 0	-2 → 0	-2 → 0	-2 → 0	-2 → 0
0	A	-2 → 0	1	-1 → 0	-1 → 0	-1 → 0	1
1	T	-2 → 0	-1 → 0	2	1	-1 → 0	-1 → 0
2	C	-2 → 0	-1 → 0	0	1	0	-1 → 0
3	T	-2 → 0	-1 → 0	1	1	0	-1 → 0
4	G	-2 → 0	-1 → 0	-1 → 0	0	2	0
5	A	-2 → 0	1	-1 → 0	-1 → 0	0	3

TAB. 2.5 – Matrice de programmation dynamique pour l'alignement local. Cette matrice est créée à partir des relations 2.3. Dans cet exemple, les paramètres sont $d = -2$, $i = -2, \forall\ \alpha \in \Sigma, S_{\alpha,\alpha} = 1$ et $\forall\ \alpha, \beta \in \Sigma, \alpha \neq \beta, S_{\alpha,\beta} = -1$. Autrement dit, une insertion ou une délétion vaut -2, une substitution vaut -1 et l'alignement correct de deux caractères vaut 1.
En supposant que l'on cherche toutes les portions dont le score de similarité est supérieur ou égal à $t = 2$, cette matrice permettrait de trouver les couples de positions $(1,1), (3,4)$ et $(4,5)$. Ces couples correspondent respectivement à l'alignement de $u[0,1]$ avec $v[0,1]$ (AT avec AT), de $u[1,3]$ avec $v[1,4]$ (TTG avec $TCTG$) et enfin de $u[1,4]$ avec $v[1,5]$ ($TTGA$ avec $TCTGA$)

La construction de cette matrice de programmation dans le but d'effectuer un alignement local pour deux textes de longueur n se fait en temps $O(n^2)$.

Alignement multiple local. Les principes précédents d'alignement local peuvent être étendus à plus de deux textes. Il est en effet intéressant, étant donné un ensemble de textes, d'en extraire les portions similaires apparaissant dans chacune des entrées. Les relations 2.3 peuvent être généralisées pour aligner $m > 2$ textes en calculant pour chaque case d'une matrice à m dimensions, une valeur dépendant de $2^m - 1$ cases voisines.

Malgré la possibilité de factoriser les informations communes [LGS02], et donc de limiter en pratique le temps de calcul, la complexité de l'algorithme d'alignement multiple local pour m textes de longueur n est en $O(2^m \times n^m)$. Concrètement, cette complexité limite la possibilité d'utilisation de cet algorithme à quelques textes seulement. Par exemple, s'il s'agit de $n = 3$ textes de longueur $m = 1\ Mb$ chacun, sur un ordinateur capable d'effectuer 10^9 opérations par seconde, le calcul prendrait plus de 3000 années (en supposant la constante de la notation $O()$ au moins égale à 1. L'alignement multiple est l'un des problèmes clefs de la bio-informatique. En plus d'apporter une information essentielle à l'étude de la phylogénie elle permet la détection d'éléments fonctionnels comme les exons [BMO+03, OBL04] ou les séquences régulatrices [LHP+03, NOAR03] avec une aisance et une précision que les méthodes d'alignement deux à deux ne permettent pas.

Les solutions exactes, bien que de complexité exponentielle, ont été très étudiées depuis les premiers algorithmes apparus en 1970 [NW70, SW81, Tay86,BS87,HS88,Alt89,LAK89,SH89,Got93,Got96,THG94,NHH00,GDG01, SEL+03, HYT03, BCG+03, Edg04].

Dans de nombreux problèmes soulevés par la bio-informatique, la complexité exponentielle de ces algorithmes n'est pas acceptable. Ainsi diverses méthodes sont développées pour limiter la complexité et donc les temps de calcul et la mémoire requise par de tels calculs. Il existe deux principales méthodes pour limiter cette complexité :

- La première méthode, appelée l'alignement multiple progressif consiste à aligner l'ensemble des textes par paires. Notons que cette méthode est une heuristique. Nous présenterons brièvement les techniques utilisées dans le paragraphe suivant.
- La seconde méthode utilise le concept de graines, et, indirectement fait naître la notion de filtrage de textes. Les travaux relatifs à cette idée

à l'origine de ceux ayant motivé cette thèse, sont présentés dans le chapitre 3.

Des heuristiques pour le calcul d'alignements multiples. De multiple heuristiques existent dans le but de calculer des alignements multiples locaux. Ces heuristiques se déclinent en deux principales méthodes.

Une première classe d'heuristiques pour l'alignement multiple utilise l'idée de l'**alignement progressif**. Plutôt que de comparer tous les textes à tous les autres, l'idée est d'aligner à chaque étape une paire de textes proches pour créer un texte consensus. Le nombre de textes est alors diminué d'une unité. Cette étape est répétée pour n'obtenir plus qu'un unique texte.

Le choix des paires de textes à aligner est souvent subordonné soit à un calcul rapide des similarités deux à deux de toutes les paires de textes possibles à aligner (basée par exemple sur la fréquence d'apparition des caractères), soit à l'utilisation de données supplémentaires tel un arbre phylogénétique.

La complexité de cette heuristique est en $O(mn^2)$ pour m textes de longueur moyenne n.

Plusieurs méthodes peuvent être employées pour déterminer, à chaque étape, quelle paire de textes est alignée. Ainsi de nombreux algorithmes ont été développés, utilisant différents types de données et différentes méthode pour le choix des paires à aligner. CLUSTAL et CLUSTAL-W [HS88, THG94] et [Got93] sont les précurseurs de ce type de travaux. DCA [SMD97] propose un algorithme de type *diviser pour régner*[2]. D'autres méthodes plus récentes comme DIALIGN [MFDW98],T-COFFEE [NHH00], MAFFT [KMKM02] (dont le choix des alignements deux à deux est déterminé par une transformation de Fourier), MAVID [BP04] ou MUSCLE [Edg04] proposent des phases de raffinement et/ou des algorithmes rapides pour déterminer l'ordre d'alignement.

* *

*

D'autres heuristiques utilisent un principe basé sur des alignements locaux. Des similarités locales entre textes sont utilisées comme des **ancres**

[2]Diviser pour régner est une technique algorithmique consistant à diviser un problème de grande taille en plusieurs sous-problèmes. L'étape de subdivision est appliqué recursivement. Son nom est inspiré du proverbe latin : « *Divide ut imperes* ».

à partir desquelles l'alignement multiple global est effectué. Ces ancres sont alors étendues de part et d'autre grâce, par exemple, à l'utilisation de la programmation dynamique. Cette phase permet d'introduire de la flexibilité dans les résultats trouvés.

Cette technique impose de trouver des alignements multiples locaux, déplaçant ainsi le problème vers un autre point difficile à résoudre. Cependant les ancres utilisées peuvent être des mots exacts détectés, par exemple, grâce à l'utilisation d'un arbre des suffixes, ou d'un tableau des suffixes[3], ou peuvent résulter de techniques plus flexibles permettant de détecter des similarités locales selon certains critères. Cette dernière possibilité, à la source des méthodes dites de filtrage, seront présentées en détail dans le chapitre 3 page 79.

Dans le cas où les graines sont des mots exacts, la première phase consiste à rechercher des mots exactement répétés dans l'ensemble des textes ou dans un quorum de textes[4].

Algorithmes d'extraction de motifs structurés

Le problème de l'extraction de motifs structurés présente, lui aussi, un intérêt majeur dans l'algorithmique du texte. Dans ce cas le programme doit être capable d'extraire d'un texte, ou d'un ensemble de textes, un ensemble de motifs approchés séparés par des espaces. La longueur des motifs et des espaces sont contraints par des intervalles.

Ce type de contraintes est fréquent en génomique car certains éléments fonctionnels interagissent à partir de lieux distants sur l'ADN. Ces éléments sont par exemple nécessaires à l'initialisation de la transcription de l'ADN à l'ARN (*c.f.* section 1.2 page 26) et fortement soumis à la pression de sélection (*c.f.* section 1.3.4 page 35).

Diverses méthodes d'extraction de telles répétitions existent et se basent sur l'arbre des suffixes [MS00a, IMP+05, PCMS06, ILM+06] ou sur des automates [AHI+06]. Ces techniques souffrent toutes d'une complexité en temps importante. Ceci est partiellement du au fait que le nombre même de motifs possiblement détectés est lui-même exponentiel.

[3] La définition d'un suffixe ainsi qu'une présentation des structures de tableau ou d'arbre des suffixes sont données dans la section 2.3.

[4] Le quorum représente le nombre minimum de textes où les répétitions doivent apparaître.

* *

*

Nous présentons maintenant dans les sections suivantes, deux structures de données couramment utilisées en algorithmique du texte et que nous avons utilisé dans les travaux de cette thèse.

2.3 Structures de données d'indexation de textes

Les algorithmes du texte conduisent fréquemment à trouver toutes les positions d'un mot dans un texte. Dans un texte non indexé, une telle tâche nécessite à chaque fois de relire l'ensemble du texte. Aussi pour limiter la complexité en temps de ces algorithmes, des structures d'indexation sont employées. Ces structures, assimilées à des dictionnaires, permettent d'obtenir *rapidement* la liste des occurrences d'un mot dans un texte indexé.

Il existe un certain nombre de structures de données permettant d'indexer les textes. Celles-ci sont basées sur des arbres, sur des tableaux, sur des automates, sur des graphes, ou sur des tables de hashage. Dans ce qui suit, nous présentons les structures basées sur des arbres et sur des tableaux car elles sont utilisées dans les travaux présentés dans ce manuscrit. Les structures basées sur les arbres se déclinent sous de très nombreuses variantes. La plus connue est l'arbre des suffixes (que nous présentons dans la section 2.3.1). Nous pouvons signaler que de nombreux algorithmes utilisent le *radix tree* aussi appelé *patricia tree* [Mor68]. Notons qu'une dizaine de structures d'indexation sont des variantes basées sur des arbres [Bla99].

Récemment, des algorithmes ont permis la construction du tableau des suffixes en temps linéaire. Cette structure, présentée dans la section 2.3.2, est plus aisée à construire et elle consomme moins de mémoire que l'arbre des suffixes. Elle est de plus en plus employée dans l'algorithmique du texte. De plus, Abouelhoda *et al.*, présentent dans [AKO04] comment le tableau des suffixes peut permettre de résoudre une liste exhaustive de problèmes liés par essence à l'arbre des suffixes.

2.3.1 L'arbre des suffixes

L'arbre des suffixes est une structure de données dont les premiers algorithmes de construction efficaces ont été proposés par Weiner [Wei73] et Mc Creight [McC76]. Cette structure stocke tous les suffixes (*c.f.* définition formelle 4 ci-dessous) d'un texte sous forme d'un arbre (dont la définition 5 est donnée ci-dessous). Un exemple d'arbre des suffixes est donné dans la figure 2.3. Cette structure peut se généraliser à plusieurs textes et prend alors le nom d'arbre des suffixes généralisé. Un exemple de tel arbre est présenté dans la figure 2.4. Au sein de cette structure, nous stockons des informations relatives aux textes ainsi indexés. C'est pourquoi, il est possible de trouver quels chemins, et par conséquent quels facteurs, sont communs à l'ensemble des textes stockés dans la structure (ou respectant un certain quorum). Dans la figure 2.4, nous observons ainsi des facteurs répétés dans les textes indexés.

Définition 4. *Préfixe et suffixe*

Soient $w, u, v \in \Sigma^*$ tels que $w = uv$. Alors u est un **préfixe** *de w et v est un* **suffixe** *de w.*

Notation 2. *Préfixe et suffixe*

- *Le préfixe d'un mot $w \in \Sigma^*$ terminant à la position i est noté $w[\dots i]$*
- *Le suffixe d'un mot $w \in \Sigma^*$ débutant à la position i est noté $w[i \dots]$*

Définition 5. *Arbre*

Un **arbre** *est une structure de données composée de* **nœuds**, *connectés par des* **arêtes**. *À l'exception d'un nœud particulier appelé la* **racine**, *chaque nœud a exactement un* **père**. *Les nœuds peuvent avoir 0* **fils** *ou plus. Les nœuds n'ayant pas de fils sont appelés des* **feuilles**, *alors que les autres nœuds sont appelé des* **nœuds internes**.

Nous appelons le **niveau** *d'un nœud la somme des longueurs des arêtes qu'il faut traverser à partir de la racine pour atteindre ce nœud. Par définition, la hauteur de la racine est zéro.*

Définition 6. *Arbre des suffixes*

Un **arbre des suffixes** *est un arbre particulier dans lequel les arêtes sont étiquetées par des caractères d'un alphabet Σ. Pour un nœud u d'un arbre des suffixes, on note $path(u)$ la concaténation des caractères lus en suivant les arêtes menant de la racine de l'arbre de suffixes au nœud u. Deux arêtes quittant un nœud sont étiquetées par différents caractères.*

L'arbre des suffixes d'un mot w\$ (\$ étant un caractère spécial n'appartenant pas à Σ) est un arbre contenant exactement $|w| + 1$ feuilles telles que pour tout suffixe v de w\$, il existe exactement une feuille f telle que $path(f) = v$.

Par extension toute concaténation de caractères lus dans l'arbre des suffixes d'un mot w\$ est un facteur de w\$.

Un exemple d'arbre des suffixes est donné sur la figure 2.3.

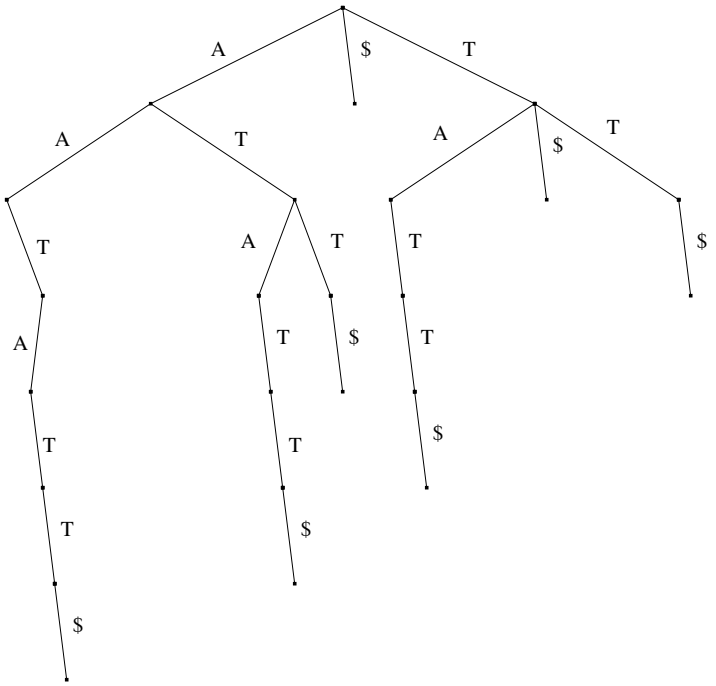


FIG. 2.3 – Exemple d'arbre des suffixes pour le texte $AATATT\$$. Bien que ceci ne soit pas explicitement représenté sur cette figure, des informations complémentaires peuvent être ajoutées dans la structure. Par exemple, la position des suffixes correspondants aux branches peuvent être stockés dans les nœuds ou dans les feuilles.

Définition 7. *Arbre des suffixes généralisé*
Un arbre des suffixes généralisé est un arbre des suffixes généralisé à plus d'un mot. Pour un ensemble de mots $w = \{w_1\$, w_2\$, \ldots, w_m\$\}$*, un tel arbre contient tous les suffixes de chacun des mots* $w_i\$$*.*

Un exemple d'arbre des suffixes généralisé est donné sur la figure 2.4.

* *

*

La structure d'arbre des suffixes généralisée ou non se construit en temps linéaire en la longueur du ou des textes à indexer. Un algorithme très élégant, dû à Mc Creight [McC76], permet d'atteindre cette complexité. Suite à la création de cet arbre, en observant tous les nœuds du niveau k, il est possible de détecter en temps linéaire tous les facteurs de longueur k répétés dans les textes.

Arbre des suffixes avec erreurs. Notons que Cole *et al.* dans [CGL04] proposent d'utiliser un arbre des suffixes permettant la recherche de répétitions approchées. Les répétitions distantes d'au plus d substitutions sur un texte de longueur n sont détectées en temps $O(n + \frac{(C_1 \log n)^d}{d!})$. Cette structure de donnée prend un espace $O(n\frac{(C_2 \log n)^d}{d!})$. Elle est construite en temps $O(n\frac{(C_2 \log d)^d}{d!})$ où C_1 et C_2 sont des constantes > 1.

* *

*

Nous proposons dans la section suivante une présentation d'une seconde structure de données très utilisée non seulement dans les travaux de cette thèse mais aussi en algorithmique du texte.

2.3.2 Le tableau des suffixes

Le tableau des suffixes, introduit en 1990 par Manber et Myers [MM90, MM93], permet d'indexer l'ensemble des suffixes d'un texte.

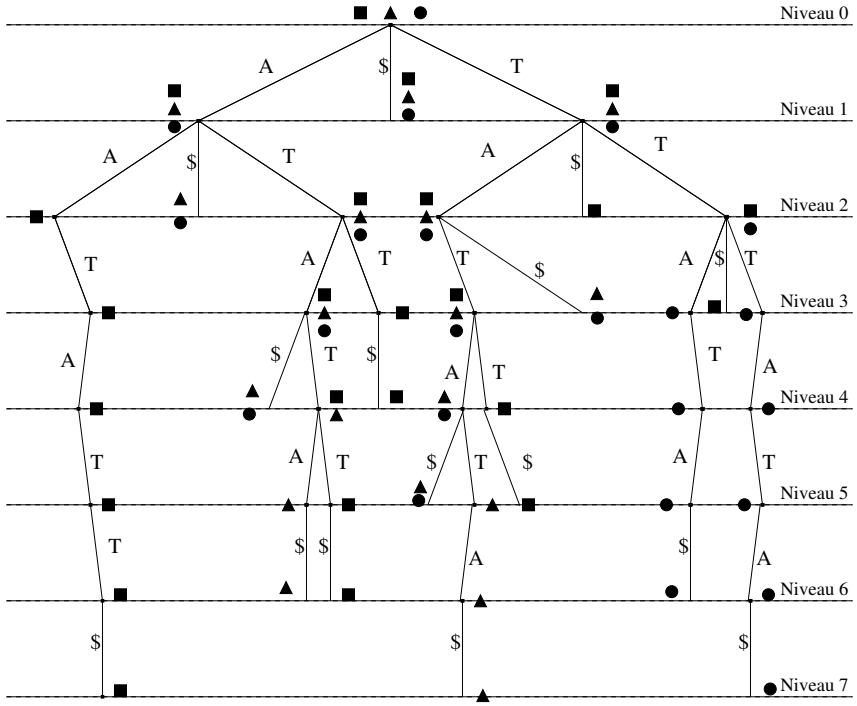


FIG. 2.4 – Exemple d'arbre généralisé des suffixes pour les textes $u = AATATT\$$, $v = TATATA\$$ et $t = TTTATA\$$. Les nœuds créés par le texte u sont représentés par un ■, ceux créés par v sont représentés par un ▲ et enfin ceux créés par t sont représentés par un ●. L'observation de cette figure permet de détecter au niveau 3 que les facteurs ATA et TAT sont présents dans les trois textes. En observant le niveau 4, nous pouvons aussi déduire de cette figure qu'aucun facteur de longueur quatre n'est présent simultanément dans ces trois textes. Cependant, à ce niveau, nous observons que le facteur $ATAT$ est, lui, présent dans le texte u et dans le texte v et que le facteur $TATA$ est présent dans les textes v et t. Enfin, en observant le niveau 5, nous concluons qu'aucun facteur de longueur au moins 5 n'est partagé entre deux textes.

Définition 8. *Tableau des suffixes*

Un **tableau des suffixes** *d'un texte s de longueur n est une permutation π de $\{0,\ 1,\ \ldots,\ n-1\}$ correspondant à l'ordre lexicographique des suffixes $s[i\ldots]$ de s.*

Nous notons $\underset{l}{\leq}$ le comparateur lexicographique entre deux mots. Alors, $s[\pi(0)\ldots] \underset{l}{\leq} s[\pi(1)\ldots] \underset{l}{\leq} \ldots \underset{l}{\leq} s[\pi(n-1)\ldots]$.

Généralement, une information supplémentaire est ajoutée au tableau des suffixes, il s'agit de la longueur du plus grand préfixe commun entre deux suffixes consécutifs du tableau des suffixes. Cette information, appelée «*Longest Common Prefix*» en anglais, sera notée **LCP**.

Un exemple de tableau des suffixes est donné sur la figure 2.5.

i	LCP	π	suffixe associé
0	0	2	$AACCAC$
1	1	6	AC
2	2	0	$ACAACCAC$
3	2	3	$ACCAC$
4	0	7	C
5	1	1	$CAACCAC$
6	2	5	CAC
7	1	4	$CCAC$

FIG. 2.5 – Exemple de tableau des suffixes pour le texte $ACAACCAC$. Dans la structure elle-même, seules les informations π et éventuellement LCP sont présentes. Ici les informations i et «suffixe associé» sont présentées pour aider à la compréhension.

* *

*

Construction du tableau des suffixes. Jusqu'en 2003, il n'existait pas de méthode de construction directe d'une telle structure en temps linéaire. Il était envisageable de construire un tableau des suffixes en temps linéaire pour un texte s, mais ceci nécessitait de construire préalablement l'arbre des suffixes de s dans le but d'utiliser cet arbre pour ensuite créer le tableau des suffixes proprement dit. En 2003, trois algorithmes ont vu le jour simultanément pour construire, sans intermédiaire, la permutation π en temps linéaire [KS03, KSPP03, KA03]. La construction en temps linéaire de la LCP est donnée dans un algorithme de Kasai *et al.* [KLA+01].

Utilisation du tableau des suffixes. Le tableau des suffixes permet différentes utilisations en fonction des besoins :

- Recherche de la plus longue répétition : Pour connaître la longueur et la position de la plus longue répétition dans un texte, il suffit, une fois le tableau des suffixes de ce texte créé, de parcourir la colonne LCP de ce tableau. Si une valeur plus grande que les précédentes est atteinte, une répétition de longueur maximale est trouvée, sa position et sa longueur sont stockées.
- Recherche de la liste des positions d'un mot donné : Connaissant un mot, il est possible d'utiliser un tableau des suffixes pour déterminer ses occurrences dans un texte. Principalement deux méthodes existent pour cela :

  1. La première méthode proposée par Sim *et al.* [SKPP03] consiste à ajouter une colonne de la longueur de l'alphabet $|\Sigma|$ contenant pour chaque caractère de Σ la position dans le tableau du premier suffixe commençant par ce caractère. Les caractères du mot recherché sont ensuite lus de la droite vers la gauche limitant à chaque itération l'espace dans le tableau où peut se trouver des suffixes débutant par ce mot. Une fois tous les caractères du mot lus, les positions où un suffixe débute par ce mot (donc les positions des occurrences de ce mot) sont connues. Cet algorithme nécessite l'ajout d'une structure de longueur $|\Sigma|$ et permet de trouver les positions en un temps linéaire en la longueur du mot. Notons de plus que cette méthode permet de détecter dans le texte des mots de n'importe quelle longueur.
  2. La seconde méthode, utilisée dans les travaux présentés dans ce manuscrit, est utilisée pour détecter des occurrences de mots dont

la longueur est fixée à l'avance, disons k. Pour obtenir rapidement la liste des occurrences de n'importe quel mot de longueur k, une colonne supplémentaire est ajoutée au tableau. Cette colonne indique pour chaque mot de longueur k la position de sa première apparition dans le tableau des suffixes. Cette colonne est de longueur $|\Sigma|^k$ (nombre possible de mots de longueur k distincts) ou la longueur du texte si celle-ci est inférieure à $|\Sigma|^k$. Ainsi lorsqu'un mot est recherché dans le tableau des suffixes, cette colonne permet en temps constant d'obtenir sa liste d'occurrences.

Conclusion

Dans ce chapitre, nous avons proposé une introduction à l'algorithmique du texte suivie d'une présentation des principaux problèmes d'extraction de répétitions. Nous avons pu constater que dans le cas de l'alignement multiple local (ou global), les solutions exactes ne sont pas envisageables sur de gros textes, et les programmes nécessitant de tels alignements ont recours à des heuristiques. Ceci se confirme dans la figure 2.6 qui présente les principaux outils d'alignement de textes biologiques.

L'un des buts principaux des travaux de cette thèse est de proposer une alternative à ces heuristiques qui fournissent un résultat dont on ne peut garantir la qualité.

La complexité des algorithmes exacts d'alignements multiples étant incompressible, l'idée est de limiter la quantité de données prise en entrée par ces algorithmes. Nous avons ainsi créé des filtres permettant de supprimer des données à traiter de grands sous-ensembles qui ne peuvent faire partie d'un alignement multiple local.

Le chapitre suivant présente le concept de filtres et propose un état de l'art sur les filtres existants.

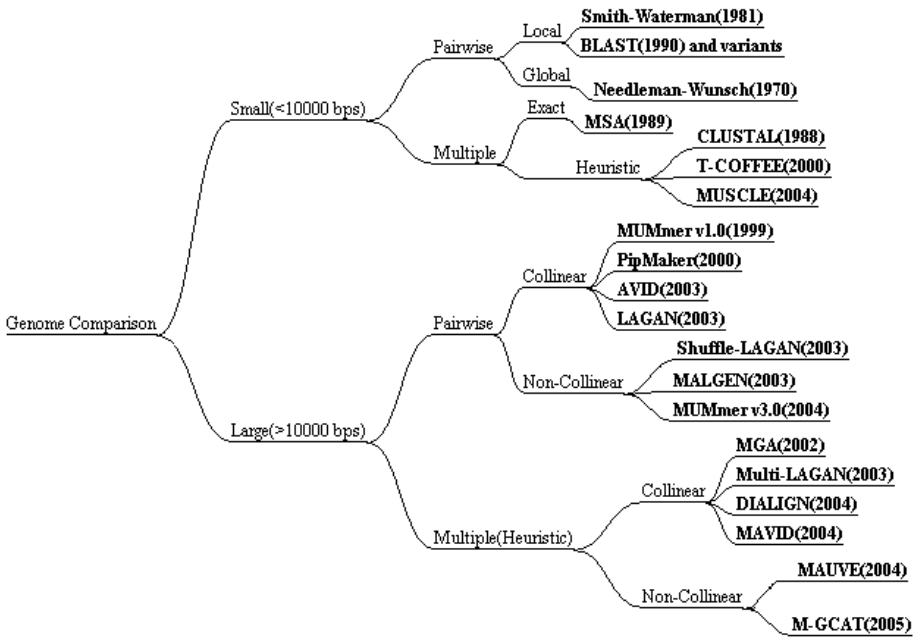


FIG. 2.6 – Principaux algorithmes de comparaisons de séquences génomiques jusqu'en 2005. Le point important à noter sur ce diagramme est l'absence de programme exact dédié à la comparaison multiples de grandes séquences.

Chapitre 3

L'état de l'art à propos du filtrage de textes

Ce chapitre présente un état de l'art à propos du filtrage des séquences textuelles. Le but des filtres que nous présentons est d'accélérer les algorithmes de détection de similarités locales ou de recherche de motifs dans un texte.

Dans la section 3.1, nous introduisons la notion de filtre ainsi qu'un ensemble de définitions employées dans la suite de ce chapitre. Nous présentons ensuite trois catégories de filtres utilisés pour la recherche de similarités. La première catégorie, présentée dans la section 3.2, est basée sur des mots de longueur maximale, la seconde catégorie (section 3.3) se base quant à elle sur des ensembles de mots de longueur fixée. La troisième catégorie (section 3.4) utilise des sous-suites de caractères, dans la section 3.5, nous présenterons des filtres qui utilisent une technique assimilable à un changement d'espace vectoriel pour filtrer les textes en amont d'un algorithme de recherche de motifs.

3.1 La notion de filtre

Un filtre* désigne une méthode utilisée pour supprimer rapidement d'un jeux de données des sous-ensembles qui ne satisfont pas certaines conditions. Un filtre peut être utilisé dans le cas où une méthode particulièrement consommatrice en mémoire et/ou en temps doit être appliquée à des données

dont seule une sous partie est pertinente. Cette méthode perd alors beaucoup de temps et/ou de mémoire sur des données n'offrant aucun intérêt. Le filtre permet de supprimer des données non pertinentes avant de les traiter par la méthode initiale.

Le filtre doit être comparativement très rapide et peu consommateur de mémoire par rapport à la méthode en amont de laquelle il est appliqué. La figure 3.1 donne une illustration de la notion de filtre.

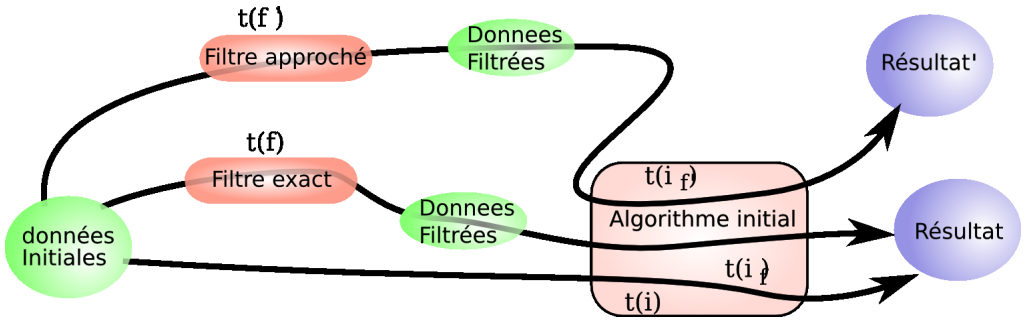


FIG. 3.1 – Schématisation du concept de filtre exact et approché. Un algorithme dit initial traite un ensemble de données en temps $t(i)$.
Un filtre exact filtre ces données en temps $t(f)$ et l'algorithme initial traite alors les données ainsi filtrées en temps $t(i_f)$ pour obtenir les mêmes résultats. Le but d'un filtre est d'avoir $t(f) + t(i_f) < t(i)$.
Un filtre approché est aussi représenté. De même dans ce cas, le temps de filtrage $t(f')$ additionné au temps d'exécution $t(i_{f'})$ de l'algorithme sur les données filtrées est aussi inférieur au temps $t(i)$ d'exécution de l'algorithme sur les données initiales. Cependant ici, le filtre pouvant éventuellement supprimer des données pertinentes, le résultat final est possiblement différent du résultat initial.

Un filtre permet donc de supprimer d'un ensemble de données celles qui ne satisfont pas un critère déterminé. Comme présenté dans la figure 3.1, deux types de filtres existent :

- les filtres «exacts», qui ne suppriment aucune portion satisfaisant le critère requis,

– les filtres dits «approchés» qui peuvent potentiellement supprimer des données respectant ce critère. Ce type de filtre s'apparente à une heuristique.

De manière formelle, nous avons la définition suivante de filtre exact*.

Définition 9. *Filtre exact*

Soient $\mathcal{P}$ une propriété sur un ensemble de données $\mathbb{E} = \{E_0, E_1, \ldots, E_{n-1}\}$, $\mathbb{E}' = \{E_i \in \mathbb{E} \mid E_i$ respecte $\mathcal{P}\}$, $\mathcal{C}$ une condition telle que $\mathcal{P} \vDash \mathcal{C}$ et $\mathbb{F} = \{E_i \in \mathbb{E} \mid E_i$ respecte $\mathcal{C}\}$.

Un filtre exact pour $\mathcal{C}$ est une fonction qui conserve de $\mathbb{E}$ l'ensemble $\mathbb{F}$. Nous le dénotons par $\mathfrak{F}_{\mathcal{C}}$.

Nous pouvons observer que $\mathcal{C}$ est une condition nécessaire mais non suffisante pour qu'un élément E_i satisfasse la propriété $\mathcal{P}$. Ainsi, dans ce cas, $\mathbb{E}' \subset \mathbb{F}$: certaines portions sont conservées par le filtre mais ne satisfont pas la propriété $\mathcal{P}$.

Ces observations sont schématisées dans la figure 3.2

L'idée générale appliquée pour filtrer des données, est de choisir une condition $\mathcal{C}$ qui soit plus «aisée» à vérifier que la propriété $\mathcal{P}$. La vérification de cette condition doit s'effectuer rapidement comparativement au temps nécessaire à la vérification de $\mathcal{P}$. De plus, $\mathbb{F} \setminus \mathbb{E}'$ doit être petit pour limiter les données conservées à tort.

* *

*

Dans certaines situations, nous pouvons être amené à créer des filtres dits approchés*. Ce type de filtres peut éventuellement supprimer des portions de données satisfaisant $\mathcal{P}$. Ceci peut être le cas dans deux situations distinctes :

– la méthode utilisée pour détecter les portions respectant la condition nécessaire $\mathcal{C}$ est une heuristique. Dans un tel cas, il est possible que certaines portions satisfaisant $\mathcal{P}$, ne soient pas conservées.
– $\mathcal{C}$ est choisit de telle manière que les données respectant $\mathcal{P}$ ne satisfassent pas nécessairement $\mathcal{C}$. Ceci peut être le cas lorsqu'il n'existe pas de condition $\mathcal{C}$ facilement identifiable et rapide à détecter telle que $\mathcal{P} \vDash \mathcal{C}$.

Comme ceci est schématisé dans la figure 3.3, dans l'une ou l'autre de ces situations il est envisageable que $\mathbb{E}' \setminus (\mathbb{F} \cap \mathbb{E}') \neq \emptyset$.

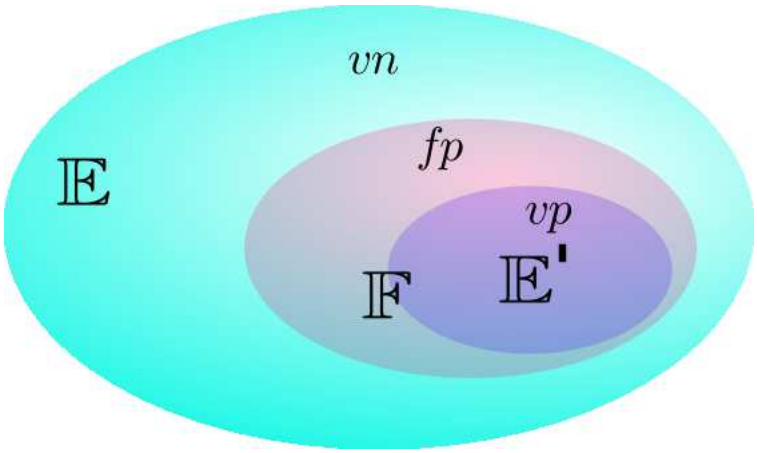


FIG. 3.2 – Schématisation des ensembles vis-à-vis d'un filtrage exact. Sur cette figure, l'ensemble $\mathbb{E}'$ satisfait une propriété $\mathcal{P}$ sur un ensemble $\mathbb{E}$. Le filtre $\mathfrak{F}_{\mathcal{C}}$, appliquant une condition nécessaire $\mathcal{C}$ de $\mathcal{P}$ conserve l'ensemble $\mathbb{F}$. L'ensemble $\mathbb{E}'$ est alors un sous-ensemble de $\mathbb{F}$, signifiant que toute portion satisfaisant la propriété $\mathcal{P}$ est conservée, alors que toute portion conservée ne respecte pas nécessairement cette propriété.
Les symboles vn, fp et vp désignent respectivement les vrai-négatifs, les faux-positifs et les vrai-positifs. Notons qu'un filtre exact ne génère pas de faux-négatifs.

3.1.1 Spécificité et sensibilité d'un filtre

Pour formaliser les notions de spécificité et de sensibilité, nous utilisons les définitions suivantes

Définition 10. *Faux-positifs, vrai-positifs, faux-négatifs, vrai-négafifs*
- *Les éléments conservés (positif) à tort (faux) par un algorithme sont appelées des faux-positifs*, il sont notés fp.*
- *Les éléments conservés (positif) à raison (vrai) par un algorithme sont appelées des vrai-positifs*, il sont notés vp.*
- *Les éléments rejetés (négatif) à tort (faux) par un algorithme sont appelées des faux-négatifs*, il sont notés fn.*

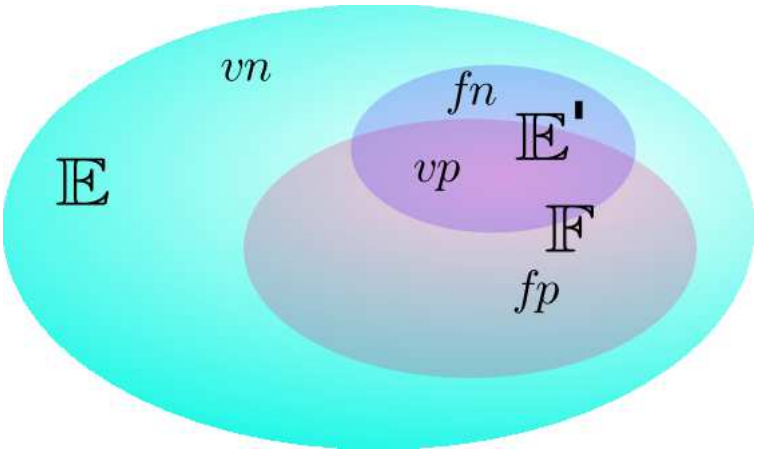


FIG. 3.3 – Schématisation des ensembles vis-à-vis d'un filtrage approché. Sur cette figure, l'ensemble $\mathbb{E}'$ satisfait une propriété $\mathcal{P}$ sur un ensemble $\mathbb{E}$. Soit le filtre applique une condition non nécessaire de $\mathcal{P}$, soit il applique de manière heuristique une condition nécessaire de $\mathcal{P}$. L'ensemble conservé est $\mathbb{F}$. Dans le cas des filtres approchés, $\mathbb{E}'$ n'est pas nécessairement inclus dans $\mathbb{F}$. Ainsi l'ensemble $\mathbb{E}' \setminus (\mathbb{F} \cap \mathbb{E}')$ représente les données *ratées* par le filtre. Les symboles vn, fn, vp et fp désignent respectivement les vrai-négatifs, les faux-négatifs, les vrai-positifs et les faux-négatifs.

– *Les éléments rejetés (négatif) à raison (vrai) par un algorithme sont appelées des vrai-négatifs*, il sont notés vn.*

Ces notions sont présentées dans les figures 3.2 et 3.3.

Spécificité

Comme nous l'avons mentionné, un filtre conserve *trop* de données comparativement à la propriété considérée. La spécificité* du filtre permet de mesurer la proportion de données correctement supprimées par un filtre par rapport à la quantité totale de données à supprimer.

Définition 11. *Spécificité*

La spécificité d'un filtre $\mathfrak{F}_{\mathcal{C}}$ est donnée par la formule :

$$spe(\mathfrak{F}_{\mathcal{C}}) = \frac{|vn|}{|vn| + |fp|}$$

D'un point de vue ensembliste, la spécificité est définie comme suit :

$$spe(\mathfrak{F}_{\mathcal{C}}) = \frac{|\mathbb{E} \setminus (\mathbb{E}' \cup \mathbb{F})|}{|\mathbb{E} \setminus \mathbb{E}'|}$$

Notons que $spe(\mathfrak{F}_{\mathcal{C}}) \in [0, 1]$.

Avec cette notation, plus la spécificité est proche de 1, meilleur est le filtre, et *vice-versa*.

Sensibilité

La sensibilité* d'un filtre $\mathfrak{F}_{\mathcal{C}}$ mesure la proportion de la quantité de données correctement conservées par le filtre par rapport à la quantité de données satisfaisant la propriété $\mathcal{P}$ de $\mathfrak{F}_{\mathcal{C}}$.

Définition 12. *Sensibilité*
La sensibilité d'un filtre $\mathfrak{F}_{\mathcal{C}}$ est donnée par la formule :

$$sens(\mathfrak{F}_{\mathcal{C}}) = \frac{|vp|}{|vp| + |fn|}$$

D'un point de vue ensembliste, la sensibilité est définie comme suit :

$$sens(\mathfrak{F}_{\mathcal{C}}) = \frac{|(\mathbb{E}' \cap \mathbb{F})|}{|\mathbb{E}'|}$$

Notons que $sens(\mathfrak{F}_{\mathcal{C}}) \in [0, 1]$.

Nous attirons l'attention sur le fait que, par définition, pour un filtre exact $\mathfrak{F}_{\mathcal{C}}$, le nombre de faux-négatifs est nul, donc, $sens(\mathfrak{F}_{\mathcal{C}}) = 1$ (ceci se vérifie de même par le fait que, dans ce cas, $\mathbb{E}' \cap \mathbb{F} = \mathbb{E}'$).

3.1.2 Filtres pour la recherche de répétitions

Dans le cadre de la bio-informatique, et plus généralement dans le cadre de l'algorithmique du texte, les filtres sont employés dans le but de retrouver des similarités.

Nous définissons ci-dessous formellement la notion de similarité.

Définition 13. (L, d)*-similarité* *Deux textes (ou séquences)* (s, s') *avec* $|s| = L$*, distants d'au plus* d *erreurs (substitutions s'il s'agit de la distance de Hamming ou insertions, délétions et substitutions s'il s'agit de la distance de Levenshtein) sont une* (L, d)*-similarité*.*

Notons que $L - d \leq |s'| \leq L + d$*.*

Dans le cadre de la recherche de (L, d)-similarités, la condition $\mathcal{C}$ des filtres créés est principalement basée sur l'intuition suivante :

Idée de base du filtrage de séquences.

- Entre deux mots «similaires», certaines portions sont exactement conservées.
- Deux mots ne partageant pas «assez» de portions exactement conservées, ou ne partageant pas de portions exactement conservées «assez longues», ne peuvent être, selon les conditions souhaitées, considérés comme similaires.

Afin d'illustrer l'idée de base précédente, la figure 3.4 permet de visualiser deux mots similaires et leurs sous parties communes.

Les filtres de séquences utilisent intensément la notion de mots de longueur k, que nous définissons formellement ci-dessous comme étant des k-facteurs ou des k-facteurs partagés.

Définition 14. k*-facteur,* k*-facteur partagé*

Un mot de longueur k *est appelé un* k*-facteur*. Un* k*-facteur sera dit* **partagé** *par deux séquences ou plus s'il apparaît exactement présent dans ces séquences. Dans ce cas, nous disons que les séquences* **partagent** *le* k*-facteur.*

De nombreux filtres pour la recherche de similarités entre séquences utilisent des conditions basées sur l'idée présentée ci-dessus. Les portions conservées sur lesquelles sont basées les filtres sont indifféremment appelées des graines* (terme que nous emploierons) ou des ancres*. L'idée de ce filtrage

FIG. 3.4 – Idée de base du filtrage de séquences. Entre les séquences $s = ATATAATTCATAC$ et $s' = ATCTATTCAGTAC$, de longueur 12, distants d'une substitution, d'une insertion et d'une délétion nous retrouvons 4 mots exactement conservés de longueur variable (de 2 à 4).

de séquences, qui date du début des années 70 [Har71], est déclinée sous de très nombreuses formes possibles. Les graines peuvent prendre différents aspects. Elles peuvent être des mots contigus (utilisés seuls ou conjointement avec d'autres mots), ou représenter des sous-séquences* (notons qu'une sous-séquence telle quelle est usuellement utilisée en informatique désigne une séquence formée à partir d'une séquence originale, en en supprimant certains caractères, sans modifier l'ordre relatif des éléments conservés). Les mots constituant ces graines peuvent être directement extraits des séquences filtrées, ou bien peuvent provenir des séquences ayant subit des transformations.

Dans la section 3.2, nous proposons une description des algorithmes, assimilables à des filtres, utilisant des k-facteurs de longueur maximale. Nous présentons dans la section 3.3 un état de l'art des filtres utilisant une condition nécessaire basée sur un unique k-facteur ou sur plusieurs k-facteurs. Ensuite, la section 3.4 présente les algorithmes de filtrage utilisant, non pas des k-facteurs formés d'une suite continue de caractères, mais formés de sous-séquences. Enfin, nous présentons dans la section 3.5 des filtres dont les graines sont constituées de portions de séquences auparavant transformées par un changement de référentiel.

3.2 Filtres utilisant un k-facteur de longueur maximale

Dans le but de détecter des (L, d)-similarités, il est possible de créer une condition nécessaire basée sur la longueur du plus grand k-facteur plutôt que sur le nombre de k-facteurs après avoir fixé k.

Dans cette section, après avoir formellement introduit la condition de filtrage utilisant un unique long k-facteur, nous présenterons les filtres ou les algorithmes assimilés à des filtres approchés qui sont basés sur une telle condition.

3.2.1 Condition de filtrage utilisant un plus long k-facteur

Une condition nécessaire pour le filtrage utilisant le plus long k-facteur partagé se base sur la proposition due à Pevzner [PW95] suivante.

Proposition 1. *Longueur du plus grand k-facteur partagé*

Les deux séquences d'une (L, d)-similarité (pour la distance de Hamming ou d'édition) partagent au moins un k-facteur de longueur

$$k = \left\lfloor \frac{L}{d+1} \right\rfloor .$$

Exemple 1. *Longueur du plus grand k-facteur partagé*

Soient $s = aggGgagAaa$ et $s' = aggAgagGaa$ deux séquences de longueur $L = 10$, avec $\mathcal{D}_h(s, s') = 2$ (les positions présentant une substitution sont indiquées par des caractères en majuscule). Ainsi, selon la proposition 1, s et s' partagent au moins un k-facteur de longueur $k = \lfloor \frac{L}{d+1} \rfloor = \lfloor \frac{10}{2+1} \rfloor = 3$. Ceci se vérifie avec les k-facteurs agg et gag.

Notons qu'aucune répartition de deux erreurs entre s et s' limiterait la longueur du plus grand k-facteur partagé à moins de 3 caractères.

Ainsi, à partir de la proposition 1, nous pouvons créer un filtre basé sur la condition suivante :

Condition 1. *Plus long k-facteur partagé*

$\mathcal{C}_1$ = À tout mot de longueur L, conservé par un filtre de recherche de (L, d)-similarités, doit correspondre un autre mot de longueur L tel que ces deux mots partagent au moins un k-facteur de longueur $k_{max} = \lfloor \frac{L}{d+1} \rfloor$.

L'utilisation de la condition $\mathcal{C}_1$ pour la création d'un filtre dans le but de rechercher des (L, d)-similarités possède l'avantage d'être extrêmement simple à mettre en œuvre. Cette idée a été utilisée dès 1992 par Ukkonen [Ukk92].

Il est en effet suffisant d'indexer dans un arbre des suffixes (structure présentée dans la section 2.3.1 page 70) ou dans un tableau des suffixes (structure présentée dans la section 2.3.2 page 72) tous les k_{max}-facteurs des séquences à filtrer. L'une ou l'autre de ces structures permet ensuite d'obtenir en temps linéaire tous les k_{max}-facteurs partagés, fournissant ainsi les positions des éventuelles (L, d)-similarités.

L'inconvénient majeur de cette condition est qu'elle est trop *faible* dans le sens où de nombreuses portions sont conservées bien que possédant plus de d erreurs. Ainsi, l'application d'une telle condition conduirait à des filtres de très faible spécificité.

L'exemple suivant permet de comprendre ce phénomène.

Exemple 2. *Faiblesse de la condition $\mathcal{C}_1$*

Les séquences $s = ATGGgagATA$ et $s' = CTACgagCTC$ sont distantes de 6 substitutions. Pourtant un filtre appliquant la condition de filtrage $\mathcal{C}_1$ dans le but de rechercher des $(10, 2)$-similarités conserve s et s' car ils partagent un 3-facteur (gag).

Ainsi, avec ces paramètres, toute zone contenant un 3-facteur partagé respecte la condition $\mathcal{C}_1$ et est donc conservée par un filtre appliquant cette condition. Or sur deux séquences aléatoires pour un modèle de Bernoulli sur un alphabet à 4 lettres, un 3-facteur donné apparaît partagé avec une probabilité $\frac{1}{64}$. Si l'on suppose que les positions sont indépendantes, sur deux séquences de longueur 10 par exemple, la probabilité de trouver aléatoirement au moins un 3-facteur partagé est d'environ 79 %.

* *

*

Nous proposons à présent un état de l'art des filtres qui basent leur techniques de filtrage sur le plus long k-facteur partagé.

3.2.2 Méthodes utilisant des longs k-facteurs

Malgré les considérations précédentes à propos de la faiblesse d'un filtrage basé sur un unique long k-facteur partagé, cette idée est largement employée

indirectement dans des algorithmes de détection de répétitions. Ainsi, des algorithmes très utilisés tels que BLAST [AGM+90] ou FASTA [LP85] sont basés sur l'idée de long k-facteur partagé (pas nécessairement le plus long). Ces algorithmes, prenant en considération l'idée des k-facteurs dans une phase de pré-traitement des données, peuvent être considérés comme des filtres approchés. En effet, ils n'appliquent pas une condition nécessaire basée sur un calcul combinatoire formel assurant la pertinence du résultat.

Nous pouvons à l'inverse citer les travaux de Wu et Manber [WM92b] et Baeza-Yates et Perlberg [BYP96] qui, dans le but d'accélérer un algorithme exact de *pattern matching*, appliquent le concept de plus long k-facteur partagé comme une phase de filtrage pour pré-traiter leurs données.

L'idée générale de ce type d'algorithme est de détecter un k-facteur partagé par plusieurs séquences ou présent plusieurs fois dans la même séquence (en fonction de l'utilisation), puis de l'étendre à sa gauche et à sa droite par programmation dynamique, autorisant ainsi un certain nombre d'erreurs. Nous pouvons citer par exemple [LLDA03] ou BLAST qui, bien qu'étant une heuristique fondé sur cette idée, est très amplement utilisé. Son serveur principal [NCBI] traite en moyenne 10000 requêtes quotidiennes.

* *

*

Un ensemble d'algorithmes [DKF+99,HKO02,BDC+03,BCG+03,DMBP04a, DMBP04b] utilisent aussi les plus longs k-facteurs dans le but de détecter des similarités locales ou de créer des alignements multiples. L'idée communément admise pour l'application des plus longs k-facteurs pour le filtrage approché est que si une portion de séquence assez longue apparaît exactement conservée entre deux séquences (ou plus), alors il est presque certain que cette portion de séquence fasse parti de l'alignement. Ces algorithmes créés pour effectuer rapidement des alignements (deux à deux [DKF+99] ou multiples [HKO02,BDC+03,DMBP04a,DMBP04b]) fonctionnent en trois étapes :

1. La première étape consiste à rechercher des k-facteurs partagés. Ceci est généralement effectué à l'aide d'une structure d'indexation des facteurs comme un arbre ou un tableau des suffixes (structures présentées dans la section 2.3 page 69). Ces k-facteurs, assimilés à des graines, sont appelés des *MUM, Maximal Unique Match* ou des *MEM Maximal Exact Match*.

2. Ensuite, un sous-ensemble des graines détectées sont chaînées. Cette étape permet de sélectionner quelles graines font effectivement partie de l'alignement.

   Plusieurs possibilités sont envisagées. Il est possible d'utiliser un calcul de la plus longue sous-séquence croissante [DKF+99]. En fonction de leurs positions, les graines sont assignées à des nombres. Le but est de trouver une suite croissante de nombres la plus longue possible.

   Plus simplement, des conditions sur les positions des graines à prendre en compte dans le chaînage sont appliquées, comme c'est le cas dans [BDC+03, DMBP04a, DMBP04b].

   Enfin, dans [HKO02], les positions des graines sont transformées en un graphe. Ensuite, le graphe est parcouru pour y détecter un chemin maximisant le nombre de graines employées.

3. La dernière phase consiste à «*fermer*» les trous entre les graines. Cela signifie que les espaces situées entre les graines sélectionnées sont alignés.

   Dans le cas de [DKF+99, BDC+03], ceci est fait à l'aide de la programmation dynamique.

   Pour [DMBP04a, DMBP04b], le choix s'est porté vers un appel récursif de la méthode sur les données à aligner.

   Pour finir, dans le cas de [HKO02], une autre heuristique, CLUSTAL-W (présentée dans [HS88, THG94]), est utilisée pour effectuer cet alignement.

* *

*

Nous présentons maintenant, dans la section suivante, des filtres basés sur la détection d'ensembles de k-facteurs dont la longueur k est fixée *a priori*.

3.3 Filtres utilisant des k-facteurs continus

Dans cette section, nous présentons des filtres de séquences dont la condition nécessaire de filtrage est basée sur un nombre minimal de k-facteurs dont la longueur k est fixée *a priori*.

3.3.1 Condition de filtrage utilisant un ensemble de k-facteurs de longueur a priori fixée

Une condition de recherche de (L, d)-similarité peut s'écrire en utilisant la proposition due à Pevzner [PW95] suivante.

Proposition 2. *Nombre de k-facteurs partagés*
Les deux séquences d'une (L, d)-similarité partagent au moins

$$p = L - k(d + 1) + 1 \text{ } k\text{-facteurs.}$$

Exemple 3. *Nombre de k-facteurs partagés*
Soient $s = aggGgagAaa$ et $s' = aggAgagGaa$ deux séquences de longueur $L = 10$, avec $\mathcal{D}_h(s, s') = 2$ (les positions présentant une substitution sont indiquées par des caractères en majuscule). En fixant $k = 3$, selon la proposition 2, ces deux séquences partagent au moins $p = L - k(d+1) + 1 = 10 - 3 \times 3 + 1 = 2$ k-facteurs.
Dans le cas présent, s et s' partagent les k-facteurs agg et gag. Notons que s et s' pourraient partager plus de 2 k-facteurs.

Ainsi, un filtre de recherche de (L, d)-similarité peut utiliser la proposition 2 appliquant la condition suivante.

Condition 2. *Nombre de k-facteurs partagés*
$\mathcal{C}_2$ = À tout mot de longueur L, conservé par le filtre doit correspondre un second mot de longueur L, tel que ces deux mots partagent au moins $p = L - k(d + 1) + 1$ k-facteurs (possiblement chevauchants).

3.3.2 Aperçu des filtres utilisant des k-facteurs partagés

L'idée basée sur le nombre de k-facteurs partagés pour détecter rapidement des similarités locales est assez ancienne. En 1987, elle a été appliquée par Karp et Rabin [KR87] pour détecter des occurrences exactes d'un mot dans un texte. Par la suite, elle a été utilisée dès 1988 pour la recherche de similarités par Lipman et Pearson [LP88], par Owolabi et McGregor [OM88], et encore par Grossi et Luccio [GL89] en 1989. Dans [JU91], cette notion est formalisée et la condition $\mathcal{C}_2$ est donnée pour des k-facteurs non chevauchants.

Dans ce cas, la condition devient :

$$\mathcal{C'}_2 = \left\lfloor \frac{L}{k} \right\rfloor - d.$$

Cette condition est employée dans l'algorithme D2 proposé par Hide [HBD94]. Notons que le filtrage basé sur un ensemble de k-facteurs n'est pas utilisé exclusivement par des algorithmes appliqués à la biologie moléculaire, loin de là. Ce type de filtre est aussi appliqué dans d'autres domaines faisant intervenir des chaînes de caractères comme dans les commandes de systèmes d'exploitation [WM92a] ou en architecture des ordinateurs [RW94] par exemple.

* *

*

Nous détaillons à présent deux algorithmes de filtrage, QUASAR [BCF+99] et SWIFT [RSM05], dont les concepts sont proches de ceux présentés dans les travaux de cette thèse.

3.3.3 QUASAR, algorithme de recherche de similarités locales

L'algorithme QUASAR [BCF+99] recherche entre une séquence de référence s_r et un ensemble de séquences $\mathbf{S}$ présentes dans une base de données, les séquences $s \in \mathbf{S}$ localement similaires à s_r. Une séquence $s \in \mathbf{S}$ est dite localement similaire à s_r si il existe une paire $(s_r[i, i+L-1], s')$ de mots telle que

- $s_r[i, i+L-1]$ est un mot de s_r de longueur L et s' est un mot de s.
- $\mathcal{D}_e(s_r[i, i+L-1], s') \leq d$, avec d fixé par l'utilisateur. Autrement dit, le couple $s_r[i, i+L-1]$ et s' est une (L, d)-similarité.

Afin de simplifier le problème, les séquences de $\mathbf{S}$ sont considérées comme une unique séquence S constituée des séquences de $\mathbf{S}$ concaténées. L'algorithme de filtrage des séquences est dans ce cas une application directe de la condition $\mathcal{C}_2$. Ainsi, toute (L, d)-similarité comprend un mot de longueur L de s_r et un mot de S partageant $p = L - k(d+1) + 1$ k-facteurs avec le premier.

Cette condition est utilisée lors d'une phase de filtrage des séquences pour supprimer des portions de séquences dans s_r ou S qui ne peuvent contenir de (L, d)-similarité. Nous présentons dans la section suivante les moyens mis en œuvre pour tester rapidement cette condition.

Application de la condition $\mathcal{C}_2$

Dans ce qui suit, nous présentons les méthodes employées pour déterminer si le premier mot de longueur L de s_r ($s_r[O, L-1]$) respecte la condition $\mathcal{C}_2$ avec un mot de S. Nous présenterons ensuite comment tester la condition sur $s_r[1, L]$ puis $s_r[2, L+1]$ et ainsi de suite.

Blocs chevauchants. Afin de détecter les (L, d)-similarités de $s_r[O, L-1]$ dans S, il faut identifier tous les mots de longueur $L+d$ de S qui partagent au moins p k-facteurs avec $s_r[O, L-1]$.

Un moyen simple pour résoudre ce problème serait d'assigner un compteur à chaque mot de longueur $L+d$ de S ($|S| - L - d + 1$ compteurs) et d'incrémenter les compteurs de tous les mots contenant un k-facteur présent dans $s_r[O, L-1]$.

Ainsi, tous les mots dont la valeur du compteur serait supérieure à p seraient considérés comme de potentielles (L, d)-similarités de $s_r[O, L-1]$.

Une telle manière de procéder est problématique du point de vue de la mémoire nécessaire pour stocker les $|D| - L - d + 1$ compteurs, mais aussi au niveau du temps requis pour l'incrémentation de tous les compteurs ainsi que du temps nécessaire à la vérification des blocs dont la valeur du compteur est supérieure à p.

Ainsi, de manière similaire aux travaux présentés dans [JU91, WM92b], plutôt que de comparer les k-facteurs contenus dans $s_r[O, L-1]$ avec le contenu de tous les mots de longueur $L+d$ de S, S est partitionné en «*blocs*» de longueur b ($b \geq 2L$). Un compteur est alors assigné à chaque bloc. Le compteur d'un bloc est incrémenté lorsqu'un k-facteur présent dans $s_r[O, L-1]$ est aussi présent dans ce bloc.

Afin que tout mot de S étant une (L, d)-similarité de $s_r[0, L-1]$ soit entièrement contenu dans au moins un bloc, et ce quelle que soit sa position, les blocs sont répartis sur S toutes les $\lfloor \frac{b}{2} \rfloor$ positions. Une telle répartition de blocs de longueur b conduit les blocs à se chevaucher sur $\lfloor \frac{b}{2} \rfloor$ positions. La figure 3.5 propose une illustration de cette idée.

Ainsi, les tests d'application de la condition $\mathcal{C}_2$ se font sur $\left\lfloor \frac{|S|}{L} \right\rfloor$ blocs plutôt que sur $|S| - L - d + 1$ blocs.

Cette stratégie est à double tranchant. Elle permet de diminuer le temps et la mémoire utilisée, cependant elle conduit à un filtre ayant une plus mauvaise spécificité. En effet, la condition $\mathcal{C}_2$ compte le nombre minimum de k-facteurs

contenus dans deux séquences de même longueur, or, le partitionnement de S en blocs conduit à comparer une séquence de longueur L avec des séquences de longueur $\geq 2L$. La condition $\mathcal{C}_2$ reste vraie, mais la probabilité d'obtenir par hasard des k-facteurs partagés est plus grande.

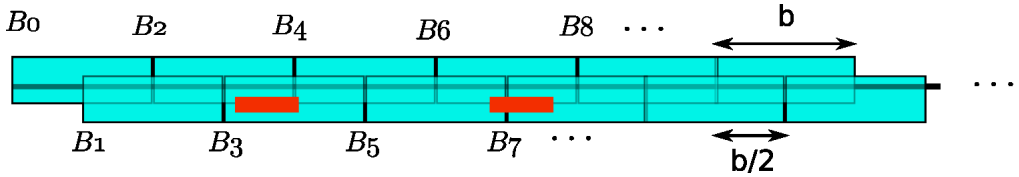


FIG. 3.5 – Blocs chevauchants de la moitié de leur longueur. Avec une telle répartition des blocs, tout mot de longueur L est entièrement contenu dans au moins un des blocs de longueur $b \geq 2L$, chevauchants sur $\lfloor \frac{b}{2} \rfloor$ positions. Deux mots de longueur L sont représentés par des rectangles rouges. Nous constatons que quelle que soit la position d'un tel mot, il est nécessairement entièrement contenu dans un bloc. Par exemple, ici le premier mot est contenu dans le bloc B_3 alors que le second est contenu dans le bloc B_6.

Indexation des k-facteurs dans un tableau des suffixes. Afin d'incrémenter rapidement les compteurs des blocs, il est nécessaire, lorsqu'un k-facteur est rencontré dans $s_r[0, L-1]$, de connaître rapidement la liste des positions des occurrences de ce k-facteur dans S. Ceci est fait à l'aide de l'utilisation d'un tableau des suffixes (structure présentée dans la section 2.3.2 page 72). Pour un k-facteur donné, grâce à cette structure, la liste des positions de ce k-facteur est obtenue en temps constant, permettant ainsi une mise à jour rapide des compteurs des blocs.

Test de la condition sur $s_r[1, L]$, $s_r[2, L+1]$, etc... Une fois les calculs effectués pour $s_r[0, L-1]$, trouver les similarités avec $s_r[1, L]$ ne nécessite pas de recalculer toutes les valeurs des compteurs. En effet, lors du passage de $s_r[0, L-1]$ à $s_r[1, L]$, un k-facteur ($s_r[0, k-1]$) disparaît et un k-facteur ($s_r[L-k+1, L]$) apparaît. Ainsi, tous les compteurs des blocs contenant le k-facteur $s_r[0, k-1]$ sont décrémentés et ceux contenant le k-facteur $s_r[L-k+$

$1, L]$ sont incrémentés. Lors de ces modifications, il est possible de détecter quels blocs ne partagent plus p k-facteurs avec $s_r[1, L]$, et, à l'inverse, quels blocs parviennent à partager p k-facteurs avec $s_r[1, L]$. Cette mise à jour est ainsi faite très rapidement, c'est-à-dire en $O(|S|)$. Nous dirons alors que nous utilisons une fenêtre glissante.

Affinage du résultat. Une fois la phase de filtrage effectuée pour une portion $s_r[i, i + L - 1]$ de s_r, l'algorithme BLAST [AGM+90] est utilisé pour affiner le résultat et détecter quels étaient effectivement les blocs représentant une (L, d)-similarité de $s_r[i, i + L - 1]$.

* *

*

Dans la section suivante, nous présentons SWIFT, un autre algorithme de détection de (L, d)-similarité utilisant la condition $\mathcal{C}_2$.

3.3.4 SWIFT Algorithme de recherche de similarités locales par Rasmussen, Stoye et Myers

Nous présentons dans cette section un algorithme récent (2005) de filtrage pour détecter des similarités locales par Rasmussen, Stoye et Myers [RSM05]. Cet article présente une méthode innovante de filtrage de similarités dont la longueur n'est pas bornée et dont le nombre maximal d'erreurs est fonction de la longueur des répétitions recherchées. Plus précisément, les techniques proposées permettent, entre deux séquences s et s', le filtrage pour la détection de $(L, \lfloor \epsilon L \rfloor)$-similarités pour tout L supérieur à L_0 fixé avec $0 \leq \epsilon < 1$ fixé.

Nous commençons la description des conditions de filtrage pour la détection de $(L_0, \lfloor \epsilon L_0 \rfloor)$-similarités avec L_0 fixé, avant de présenter l'extension au filtrage de similarités dont la longueur n'est pas fixée.

Détection de $(L_0, \lfloor \epsilon L_0 \rfloor)$-similarités avec L_0 fixé

Le filtrage pour la détection de $(L_0, \lfloor \epsilon L_0 \rfloor)$-similarités avec L_0 fixé applique une fois encore la condition $\mathcal{C}_2$ concernant le nombre de k-facteurs partagés, contenu dans une $(L_0, \lfloor \epsilon L_0 \rfloor)$-similarité. L'idée pour appliquer cette

condition permettant de trouver des couples $(u, v) \in \Sigma^*$ avec $|v| = L_0$ et $\mathcal{D}_e(u, v) \leq \lfloor \epsilon L_0 \rfloor$ est la suivante. Des portions de la matrice de programmation dynamique en forme de parallélogramme sont observées. Les parallélogrammes contiennent L lignes et d diagonales (voir figure 3.6), nous parlons alors de parallélogrammes de longueur $L \times d$. De tels parallélogrammes s'étendent ainsi sur $L + d$ colonnes.

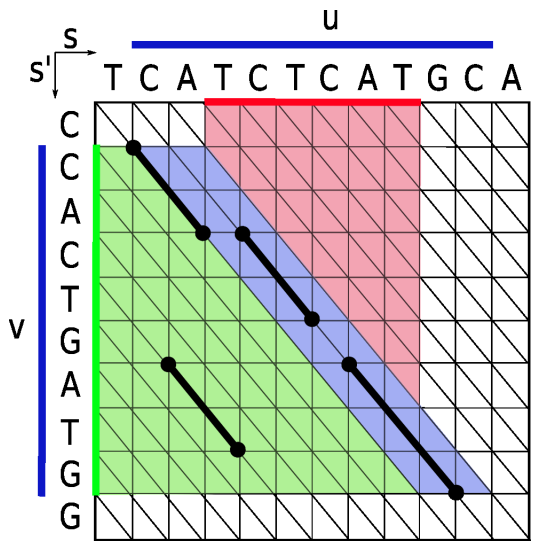


FIG. 3.6 – Parallélogramme de détection de k-facteurs. Un parallélogramme de longueur 8×2 (symbolisé en bleu) contient 4 2-facteurs partagés (dont deux sont chevauchants) symbolisés par des traits diagonaux épais. Dans cet exemple, les mots correspondants au parallélogramme dans s et s' contiennent deux sous mots, $u = CATCTCAT$ et $v = CACTGATG$, qui partagent le bon nombre de k-facteurs. Ces deux mots sont conservés dans le cadre d'un filtrage détectant des $8, 2$-similarités.

Pour que les séquences correspondant à la projection du parallélogramme sur s et sur s' puissent contenir une $(L_0, \lfloor \epsilon L_0 \rfloor)$-similarité, ce parallélogramme doit contenir au moins $p = L - k(d+1) + 1$ k-facteurs partagés avec $d = \lfloor \epsilon L_0 \rfloor$.

L'utilisation du parallélogramme permet d'imposer que les k-facteurs par-

tagés apparaissent à des positions similaires ($\pm d$ par rapport aux deux mots définis par le parallélogramme) dans les répétitions.

Ceci est visible sur l'exemple présenté dans la figure 3.6, où l'on peut constater qu'un des k-facteurs partagé (AT) n'est pas pris en considération car il n'est pas contenu dans le parallélogramme.

Si un parallélogramme de longueur $L \times \lfloor \epsilon L_0 \rfloor$ contient moins de p k-facteurs, les mots correspondants dans s et s' ne peuvent contenir de $(L_0, \lfloor \epsilon L_0 \rfloor)$-similarité et les positions correspondantes sont supprimées par le filtre.

Les séquences s et s' sont donc filtrées en supprimant toutes les positions débutant un parallélogramme de longueur $L \times \lfloor \epsilon L_0 \rfloor$ contenant moins de p k-facteurs partagés.

Détection de $(L, \lfloor \epsilon L \rfloor)$-similaritée avec $L \geq L_0$

Afin de permettre le filtrage de similarités dont la longueur n'est pas fixée a priori, la condition de filtrage précédente est étendue pour la recherche de $(L, \lfloor \epsilon L \rfloor)$-similarités avec $L \geq L_0$ variable.

Le filtrage est basé sur la propriété suivante : si une $(L, \lfloor \epsilon L \rfloor)$-similarité avec $L \geq L_0$ existe alors il existe nécessairement un parallélogramme de longueur $\mathcal{L} \times \delta$ contenant un nombre minimum p de k-facteurs partagés.

Les valeurs des paramètres p, $\mathcal{L}$ et δ sont données dans la proposition 3.

Proposition 3. *Soit (u, v) avec $|v| = L \geq L_0$, $\mathcal{D}_e(u, v) < \lfloor \epsilon L \rfloor$.*

Dans une telle situation il existe un parallélogramme de longueur $\mathcal{L} \times \delta$ contenant au moins p k-facteurs partagés avec :

$$p = min\left\{U(L_0, k, \epsilon), U(L_1, k, \epsilon)\right\}, \qquad U(L, k, \epsilon) = L + 1 - k(\lfloor \epsilon L \rfloor + 1),$$
$$\mathcal{L} = p - 1 + k(\delta + 1), \qquad L_1 = \lceil (\lfloor \epsilon L_0 \rfloor + 1)/\epsilon \rceil,$$
$$\delta = \left\lfloor \frac{2p + k - 3}{1/\epsilon - k} \right\rfloor.$$

De cette proposition naît la condition de filtrage suivante.

Condition 3. *Nombre de k-facteurs partagés dans une répétition de longueur au moins L_0*

$\mathcal{C}_3$ = À toute $(L, \lfloor \epsilon L \rfloor)$-similarité ($L \geq L_0$) correspond au moins un parallélogramme de longueur $\mathcal{L} \times \delta$ contenant au moins p k-facteurs partagés, avec p, $\mathcal{L}$ et δ donnés dans la proposition 3.

L'application de la condition $\mathcal{C}_3$ se fait en considérant les régions qui ne contiennent pas de parallélogramme de longueur $\mathcal{L} \times \delta$ contenant au moins p k-facteurs partagés. Sur ces portions de séquences, il est impossible qu'apparaisse une occurrence de $(L, \lfloor \epsilon L \rfloor)$-similarité, avec $L \geq L_0$. Ces régions sont donc supprimées par le filtre.

Recherche de parallélogrammes de longueur $\mathcal{L} \times \delta$ contenant au moins p k-facteurs partagés

Pour appliquer la condition de filtrage $\mathcal{C}_3$, il est nécessaire d'être en mesure de détecter l'ensemble des parallélogrammes de longueur $\mathcal{L} \times \delta$ contenant au moins p k-facteurs partagés.

Afin d'effectuer cette opération rapidement, l'algorithme SWIFT, présenté dans [RSM05], recherche des parallélogrammes de longueur $\mathcal{L} \times (\delta + \Delta)$ chevauchants de δ positions. Plus les parallélogrammes à rechercher sont «*épais*», plus leur recherche est rapide. La recherche de tels parallélogrammes se base, comme dans le cas de QUASAR, sur l'utilisation de compteurs stockant le nombre de k-facteurs partagés entre portions de séquences. Or, rechercher des parallélogrammes de longueur $\mathcal{L} \times \delta$ nécessite $|s| - \delta$ compteurs, alors que rechercher des parallélogrammes de longueur $\mathcal{L} \times (\delta + \Delta)$ nécessite $\left\lceil \frac{|s| - \delta - \Delta}{\Delta + 1} \right\rceil$ compteurs.

La complexité théorique de SWIFT est en $O(|s'| + |s| \times |s'|)$, et peut être réduite à $O(|s'| + \frac{|s| \times |s'|}{|\Sigma|^k})$ si l'on considère que les séquences contiennent l'ensemble des $|\Sigma|^k$ k-facteurs possibles sur Σ.

* *

*

Les algorithmes de filtrage que nous avons présentés, basés sur le comptage de k-facteurs partagés, présentent de très bonnes caractéristiques. Notons par exemple que la commande *agrep* d'UNIX emploie un tel filtre [WM92a]. Cependant comme cela est présenté dans [SS98] par Sutinen et Szpankowski, ces méthodes sont tributaires d'une faiblesse due au fait que le nombre minimum de k-facteurs partagés diminue rapidement avec le taux d'erreur demandé par l'utilisateur. Une solution peut être de diminuer la longueur des k-facteurs recherchés mais cela augmente la probabilité de trouver des k-facteurs aléatoires partagés. Par exemple, sur un alphabet à 4 lettres, la probabilité qu'un 2-facteur donné apparaisse à une position dans une séquence

aléatoire pour un modèle de Bernoulli est de $\frac{1}{16}$. Ainsi, la probabilité pour qu'un 3-facteur apparaisse par chance dans un texte de longueur 100, est de $1-\left(1-\frac{1}{64}\right)^{97} \approx 78\%$ (en admettant pour simplifier le calcul que les positions sont indépendantes). Une telle probabilité conduit à un taux de faux-positifs important et ainsi à des filtres ayant une mauvaise spécificité.

* *

*

Malgré ce handicap, d'autres types de filtres basés sur des k-facteurs ont vu le jour. Plutôt que de considérer des k-facteurs composés d'une suite continue de caractères, ces filtres utilisent comme graine des k-facteurs composés d'un ensemble de caractères provenant de positions non contiguës (sous-séquences). Nous nommons ces k-facteurs des k-facteurs *éclatés*. Nous présentons dans la section suivante les principaux algorithmes de filtrage impliquant des k-facteurs éclatés.

3.4 Filtres utilisant des k-facteurs *éclatés*

Dans cette section, nous présentons les principaux algorithmes de filtrage qui emploient des facteurs provenant de sous-séquences.

Nous débutons cette partie par une introduction aux k-facteurs éclatés, avant de présenter dans la section 3.4.2 les principaux algorithmes utilisant ce type de données.

3.4.1 Introduction aux k-facteurs éclatés

Un k-facteur éclaté* est définit comme suit.

Définition 15. *k-facteur éclaté*

Soit s un texte de longueur n. Une **forme** *est un ensemble d'entiers strictement croissants $f = \{f_1 = 0, ..., f_k\}$.*

Pour tout $i \leq n - f_k + 1$,

$$s_i^f = s[i + f_1]s[i + f_2] \dots s[i + f_k]$$

*est un k-***facteur éclaté** *débutant à la position i pour la forme f.*

Pour une forme f, $f_k + 1$ est appelée **l'envergure** *de f et k est appelé le* **poids** *de f.*

Notation 3. *Représentation d'une forme*

Afin de faciliter la lecture de ce manuscrit, une forme $f = \{f_1 = 0, ..., f_k\}$ *est notée par une succession de caractères '#' et '−'. Une forme débute et se termine toujours par un '#'. Chaque caractère '#' représente un* f_i *et est séparé du caractère '#' précédent par* $f_i - f_{i-1} - 1$ *caractères '−'.*

Avec une telle notation, une forme contient k *caractères '#' et son envergure est égale au nombre total de caractères qui la composent.*

Exemple 4. *Représentation d'une forme*

La forme $f = \{0, 2, 5, 10\}$ *est notée* $\#-\#--\#-----\#$.

Exemple 5. *k-facteur éclaté*

Soit s *le texte* $AGTACGATCGGGTA$ *et* f *la forme* $\#-\#--\#-----\#$.

Pour la forme f*, le k-facteur éclaté débutant position 2 dans le texte* s *est* $TCTT$.

Définition 16. *k-facteur éclaté partagé*

Vis-à-vis de deux séquences ou plus, un k-facteur éclaté est dit **partagé** *s'il apparaît dans l'ensemble de ces séquences.*

3.4.2 Algorithmes de filtrage utilisant des k-facteurs éclatés

L'utilisation de k-facteurs éclatés pour le filtrage de séquences a été introduite en 1993 par Califano et Rigoutsos [CR93]. En 1995, Pevzner et Waterman [PW95], appliquent des formes régulières du type $\#---\#---\#$. Ils proposent de plus le calcul du nombre minimum de k-facteurs éclatés partagés, associés à ce type de formes.

Certains algorithmes [AMS+97,Ken02] assimilables à des filtres approchés utilisent des k-facteurs éclatés pour initialiser un alignement ou pour détecter spécifiquement des régions homologues codantes [BBV04].

Des filtres proprement dits, emploient des k-facteurs éclatés [BK01,MTL02, BKS03,FCLST05] ou des familles de k-facteurs éclatés pour la recherche de similarités ou pour la recherche d'éléments [LM03,SB04,YWC+04,XBLM04, KNR05b].

L'utilisation de k-facteurs éclatés augmente grandement la difficulté de calcul du nombre minimum de k-facteurs éclatés partagés par des occurrences

de similarités. Or la détermination d'un tel seuil est indispensable pour obtenir une condition nécessaire de filtrage permettant de créer des filtres exacts. Dans le cas des familles de k-facteurs éclatés, la difficulté, pour un type de similarité donné, est de déterminer une famille de formes. Dans ce cas, il faut que pour tout couple de mots d'une similarité, il existe une des formes de la famille définissant un k-facteur éclaté partagé.

Ces difficultés conduisent au fait qu'à l'exception de [FCLST05], tous les filtres basés sur ce type de graines sont des filtres approchés et ont donc une sensibilité inférieure à 1. C'est pourquoi à une forme de k-facteur éclaté donnée, une sensibilité est associée pour un problème particulier. La sensibilité d'une forme permet d'estimer la quantité de données que le filtre appliquant cette forme élimine à tort par rapport à la quantité de données qu'il conserve. Dans [KNR05a] Kucherov *et al.* proposent un cadre global pour l'analyse de la sensibilité de telles graines. Dans [BKS03] Buhler *et al.* proposent un algorithme permettant de calculer la sensibilité de formes de k-facteurs éclatés pour des problèmes de recherche de similarités sous la distance de Hamming. D'autre part Buhler *et al.* proposent aussi un algorithme appelé MANDALA, permettant de créer des graines maximisant la sensibilité du filtre associé.

Le calcul des formes à appliquer pour maximiser la sensibilité des filtres est un problème difficile. De plus, le type de formes à utiliser dépend du type de répétitions recherchées. Dans [SB06], Sun et Buhler proposent une étude de formes en fonction du type de séquences d'ADN à aligner.

Notons que le concept de k-facteur éclaté a aussi été appliqué à l'alignement de protéines. Dans ce cas, les distances ne prennent plus en compte uniquement des substitutions, des insertions ou des délétions. Les lettres mises en relation ne sont plus considérées comme égales ou différentes mais sont alors liées les unes aux autres par des scores. Des techniques, proposées par Brown *et al.* [BBV05, Bro04], proposent des filtres basés sur des k-facteurs éclatés pour ce type d'alignement.

* *

*

Filtres utilisant des k-facteurs éclatés et distance d'édition. Les filtres basés sur des k-facteurs éclatés conduisent, au détriment d'une sensi-

bilité inférieure à 1, à des filtres ayant une meilleure spécificité que les filtres utilisant des k-facteurs continus. Cependant, à l'exception des travaux présentés dans [BK02] par Burkhardt et Karkkainen (où un k-facteur éclaté ne contient qu'un trou dont la longueur est incrémentée puis décrémentée d'une unité pour accepter une éventuelle insertion ou délétion), l'intégralité des filtres prenant en compte ce type de k-facteurs fondent leurs calculs et leurs expérimentations sur des répétitions dont la distance est basée sur une distance de Hamming. Ceci représente une sévère limitation dans l'application possible de ce type de filtre pour la bio-informatique.

* *

*

Nous présentons dans la section suivante un type de filtres en marge de ce que nous avons présenté précédemment, transformant les séquences avant de les comparer.

3.5 Filtres utilisant un changement d'espace vectoriel

Nous présentons dans cette section une idée de filtrage heuristique pour la recherche de répétitions deux à deux. Cette idée est utilisée dans les articles de Aghili *et al.* [AAA03, ASAA04].

Le but est de supprimer rapidement d'un ensemble de séquences $\mathbf{S}$, un maximum de positions où des occurrences approchées d'une séquence s_r de référence ne peuvent apparaître.

L'idée appliquée dans ces algorithmes est de transformer l'ensemble les séquences dans un espace vectoriel propre aux séquences originales $\mathbf{S}$. La transformation appliquée est choisie de manière à limiter la longueur des séquences à comparer.

Plus précisément, les séquences $\mathbf{S}$ sont découpées en b blocs, chaque bloc est alors remplacé par un vecteur de fréquence d'apparition. Le vecteur de fréquence d'apparition d'un bloc est de longueur $|\Sigma|^k$. Chaque position i de ce vecteur représente la fréquence d'apparition du k-facteur dont l'étiquette est i dans le bloc. Chaque k-facteur possède une étiquette distincte, par exemple avec $k = 2$, sur l'alphabet $\{A, C, G, T\}$, AA possède l'étiquette 0,

AC l'étiquette 1, *AG* l'étiquette 2, etc. Ainsi, si pour un bloc donné le 2-facteur *AA* apparaît 2 fois, son vecteur de fréquence d'apparition débute par 2.

Les vecteurs de fréquence d'apparition des blocs sont alors regroupés dans une matrice S' à laquelle une transformation matricielle, appelée la *Single Value Decomposition*, est appliquée. Cette décomposition conduit à la création de trois nouvelles matrices U, D et V telles que $S' = U \times D \times {}^{tr}V$ (${}^{tr}V$ désignant la matrice transposée de V).

Les données comparées sont alors chacune des lignes de la matrice $S \times V$ avec le vecteur $s'_r \times V$, où s'_r représente le vecteur de fréquence d'apparition de s_r. Ceci permet ainsi de réduire l'espace original de comparaison.

Cette technique permet, par l'application de la *Single Value Decomposition*, de diminuer l'espace de recherche d'occurrences approchées. En effet, la dimension de la matrice V est inférieure à celle de la matrice originale S'. Cette heuristique ne permet pas d'assurer la qualité du filtrage proposé ni de quantifier la complexité en temps.

Conclusion de chapitre

Dans ce chapitre, nous avons présenté un état de l'art sur le filtrage de textes. Plusieurs types de méthodes employant des k-facteurs ont étés présentées. Nous avons notamment examiné les filtres détectant un k-facteur partagé de longueur maximale, un ensemble de k-facteurs partagés de longueurs fixées ou des k-facteurs éclatés. Enfin, nous avons mentionné l'existence de filtres approchés utilisant un changement d'espace vectoriel.

Il est intéressant de noter que l'intégralité des techniques (heuristiques ou non) présentées dans cet état de l'art est destinée au filtrage de séquences pour la rechercher de répétitions deux à deux (dans une même séquence ou dans des séquences distinctes). Ainsi, toutes ces méthodes sont prévues pour comparer au plus deux entités. Le domaine du filtrage de textes souffre donc de l'absence totale de filtre multiple de séquences. Pourtant ce type de techniques serait d'une grande aide, entre autres pour les généticiens.

Les travaux que nous présentons dans cette thèse ont pour but de combler ce manque en proposant des techniques de filtrages destinées à détecter des similarités apparaissant deux fois ou plus dans un texte, ou bien apparaissant dans un ensemble d'au moins deux textes.

Nous présentons ainsi dans les chapitres suivants deux méthodes de filtrage destinées à la détection de similarités multiples. L'une s'applique à la distance de Hamming, alors que l'autre est utilisée pour la distance d'édition.

Chapitre 4

NIMBUS : filtre de séquences pour la distance de Hamming

Dans ce chapitre, nous présentons un filtre prévu pour permettre la recherche de longues répétitions dans des séquences d'ADN. L'une des particularités de ce filtre est d'être applicable à la recherche de répétitions ayant deux occurrences ou plus. Il n'est donc pas limité à la recherche de répétitions apparaissant exactement deux fois. Dans la suite de ce manuscrit, de telles répétitions sont appelées des répétitions multiples.

Ce filtre, appelé NIMBUS, permet de filtrer des séquences dans le but d'extraire des répétitions approchées dans le cadre de la distance de Hamming (distance basée sur les substitutions uniquement).

Nous débutons ce chapitre par une présentation de la finalité de NIMBUS. Comme nous l'avons mentionné dans le chapitre précédent, les filtres sont basés sur une condition de filtrage. La condition de filtrage utilisée par NIMBUS est présentée dans la section 4.2. Nous proposerons par la suite dans la section 4.3 une étude de l'algorithme permettant l'application de cette condition. Comme nous le constatons dans cette section, les performances de l'algorithme sont largement améliorées par l'utilisation d'un nouveau type d'objets que nous avons appelé les bi-facteurs. Les bi-facteurs que nous utilisons sont indexés dans une structure de données appelée le tableau des bi-facteurs. Une présentation de cette structure ainsi que de l'algorithme de création associé est proposée dans la section 4.4. La complexité de l'algorithme NIMBUS est présentée dans la section 4.5 et un ensemble de tests est proposé dans les sections 4.6, 4.7 et 4.8.

4.1 But du filtrage

Le filtre NIMBUS a pour but d'accélérer la recherche de répétitions multiples approchées dont les occurrences sont distantes entre elles par un nombre maximum de substitutions.

Ce filtre se décline selon deux applications légèrement différentes. La première application, que nous appelons MULTI-NIMBUS, permet de filtrer des ensembles de séquences d'ADN. Dans ce cas, les répétitions recherchées sont réparties dans un ensemble minimum de séquences distinctes. Un aperçu de ce filtre est donné dans la figure 4.1.

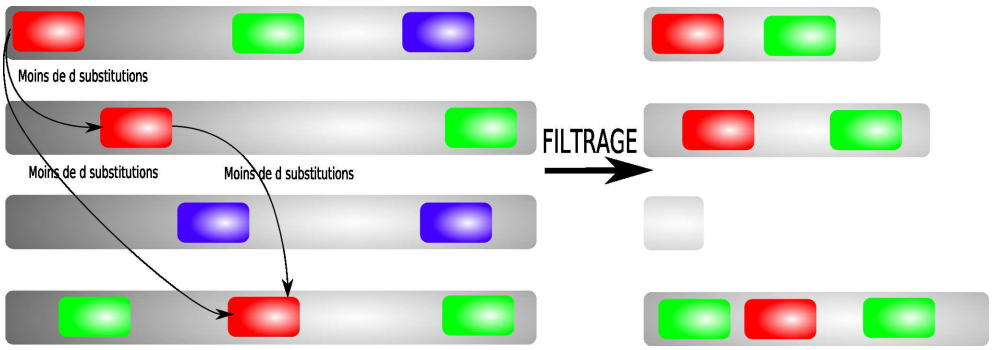


FIG. 4.1 – Aperçu de l'algorithme MULTI-NIMBUS. Une ensemble de séquences contient des répétitions. Supposons ici que l'on veuille identifier les répétitions réparties dans au moins trois séquences. Nous constatons que les répétitions vertes et rouges respectent ce critère. Elles sont alors conservées par le filtre. À l'inverse, la répétition bleue n'est répétée que dans deux séquences et le reste des séquences ne contient pas de répétitions. Ces zones sont donc en partie supprimées par le filtre.

La seconde application, appelée MONO-NIMBUS est utilisée pour filtrer une unique séquence pour en extraire les portions se trouvant répétées dans cette séquence. La figure 4.2 donne un aperçu de cette version du filtre.

Comme nous le verrons par la suite, la condition de filtrage ainsi que l'application algorithmique de MULTI-NIMBUS et MONO-NIMBUS sont très

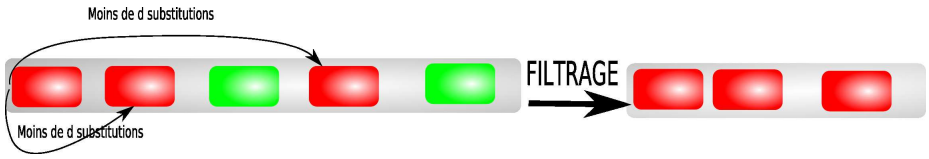


FIG. 4.2 – Aperçu de l'algorithme MONO-NIMBUS. Une séquence contient des répétitions. Supposons ici que l'on veuille conserver les répétitions ayant au moins trois occurrences dans la séquence. Nous constatons que seule la répétition dont les occurrences sont représentées en rouges respectent ce critère. Cette répétition est donc conservées par le filtre. À l'inverse, la répétition verte ne possède que deux copies, elle est donc supprimée par MONO-NIMBUS.

similaires. Aussi, pour éviter les redondances, nous ferons dans ce qui suit une présentation de l'application MULTI-NIMBUS. Les différences entre MULTI-NIMBUS et MONO-NIMBUS seront présentées dans la section 4.2.1.

Définition 17. *(L, r, d)-répétition*

*Une (L, r, d)-**répétition*** est un ensemble $\{\delta_1, \ldots, \delta_r\}$ tel que*

$$\forall\ i \in [1, r] : \delta_i \in \Sigma^* \text{ et } |\delta_i| = L$$
$$\forall\ i, j \in [1, r] : \mathcal{D}(\delta_i, \delta_j) \leq d.$$

$\mathcal{D}(\alpha, \beta)$ désigne une distance entre deux textes α et β.

S'il s'agit d'une distance de Hamming, nous parlerons de «(L, r, d)-répétition pour la distance de Hamming».

À l'inverse si $\mathcal{D}(\alpha, \beta)$ désigne une distance d'édition, nous parlerons de «(L, r, d)-répétition pour la distance d'édition».

Définition 18. *Occurrences d'une (L, r, d)-répétition*

Chacune des portions de séquence δ_i d'une (L, r, d)-répétition est appelée une **occurrence** *de cette répétition.*

Ainsi, toute paire d'occurrences d'une (L, r, d)-répétition pour la distance de Hamming sont distantes d'au plus d substitutions.

Dans ce chapitre, toutes les (L, r, d)-répétitions que nous allons manipuler sont des (L, r, d)-répétitions pour la distance de Hamming. Ainsi, afin de simplifier la lecture de ce présent texte, nous emploierons simplement le terme de «(L, r, d)-répétition» pour les désigner, et ce jusqu'à la fin de ce chapitre, page 140.

Étant données m séquences d'ADN, le but est d'extraire les mots de longueur L, répétés dans au moins $r \leq m$ séquences, avec au plus d substitutions entre chaque paire de répétitions. En d'autres termes, le but est d'extraire les (L, r, d)-répétitions d'un ensemble de r séquences parmi un ensemble de $m \geq r$ séquences. Le but du filtre (dont un aperçu des entrées et sorties est présenté dans l'algorithme 5) est alors d'éliminer des séquences d'entrées autant de positions que possible qui ne peuvent contenir de (L, r, d)-répétitions.

Algorithme 5 MULTI-NIMBUS

Nécessite :

- m séquences d'ADN,
- L : longueur des répétitions recherchées,
- $r \leq m$: nombre minimal d'occurrences des répétitions recherchées trouvées dans des séquences distinctes,
- d : distance de Hamming maximale entre chaque paire de répétitions recherchées.

Effectue : Supprime des m séquences des portions ne pouvant faire partie de (L, r, d)-répétitions.

4.2 Condition de filtrage

Étant donné m séquences, le but est de détecter les (L, r, d)-répétitions avec L, r et d déterminés par l'utilisateur.

L'idée principale de NIMBUS est d'appliquer aux séquences une condition nécessaire basée sur le nombre minimum de k-facteurs non chevauchant que doivent partager toutes les occurrences d'une (L, r, d)-répétition. Une telle condition est appelée une condition de filtrage multiple.

La condition de filtrage multiple que nous utilisons est basée sur le théorème suivant.

Théorème 1. *Nombre de k-facteurs non chevauchant partagés*

Soit p_r le nombre minimum de k-facteurs non chevauchant partagés par toutes les occurrences d'une (L, r, d)-répétition. Nous avons :

$$p_r = \left\lfloor \frac{L}{k} \right\rfloor - d - (r-2) \times \left\lfloor \frac{d}{2} \right\rfloor$$

Avant de prouver le théorème 1, nous donnons une intuition du point clef de la preuve. Pour donner cette intuition, nous considérons dans tout ce qui suit et ce jusqu'à la preuve, l'alignement virtuel de r séquences de longueur L.

Dans le cas trivial où ces séquences sont toutes identiques, elle partagent $\left\lfloor \frac{L}{k} \right\rfloor$ k-facteurs non chevauchant.

Afin de prouver le théorème 1, envisageons maintenant le pire scénario concernant le nombre de k-facteurs partagés (où ce nombre est minimal). Supposons que la distance de Hamming entre n'importe quelle paire de séquences soit exactement d. Si à la position i (avec $0 \leq i \leq L-1$), il y a une substitution entre n'importe quelle paire de séquences parmi les r séquences, alors aucun k-facteur comprenant cette position ne peut être partagé par l'ensemble des séquences. Une telle position est appelée **tâchée**. La figure 4.3 illustre la notion de position tâchée.

Le pire scénario apparaît quand un maximum de positions sont tâchées et que ces positions sont réparties toutes les k positions. Une telle répartition interdit un maximum de k-facteurs non chevauchant partagés parmi les $\left\lfloor \frac{L}{k} \right\rfloor$ k-facteurs initiaux possiblement partagés. Supposons que t positions soient tâchées ; alors, dans le pire des cas, au minimum $p_r = \left\lfloor \frac{L}{k} \right\rfloor - t$ k-facteurs sont partagés par toutes les séquences (c.f. figure 4.3).

Ainsi pour déterminer le nombre minimal p_r de k-facteurs partagés dans les r séquences, il faut déterminer le nombre maximum de positions tâchées t.

Dans le cas de deux séquences, le nombre de positions tachées dans une $(L, 2, d)$-répétition est par définition simplement au plus d et ainsi le nombre minimum p_2 de k-facteurs partagés est $p_2 = \left\lfloor \frac{L}{k} \right\rfloor - d$. Nous constatons dans ce cas que nous obtenons le même nombre minimum de k-facteurs non chevauchant partagés que dans la condition $\mathcal{C}'_2$ définie page 91.

L'observation principale que nous faisons à propos du nombre de positions tâchées est qu'ajouter une troisième séquence (distante deux à deux des deux premières de au plus d substitutions) ajoute au plus $\left\lfloor \frac{d}{2} \right\rfloor$ nouvelles positions tâchées. Ceci reste vrai lorsque l'on ajoute une quatrième séquence, puis

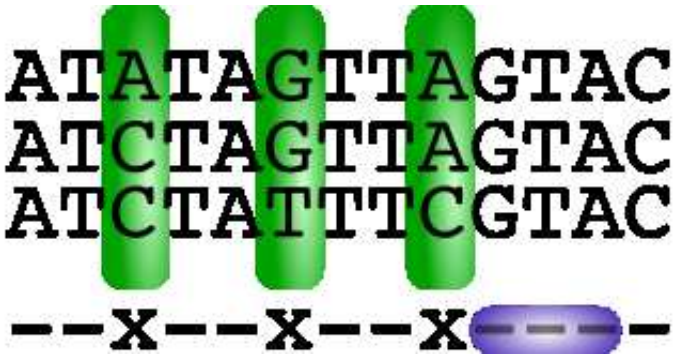


FIG. 4.3 – Sur cette figure sont représentées $r = 3$ séquences de longueur $L = 13$. Les positions tâchées, représentées par un rectangle vertical coloré ou par un 'x', correspondent aux positions où au moins l'une des r séquences a subit une substitution par rapport à une des autres. À l'inverse, sur une position non tâchée, toutes les séquences possèdent le même caractère.
Nous observons que le nombre minimum de k-facteurs non chevauchant partagés par toutes les séquences est égal à $\lfloor \frac{L}{k} \rfloor - t$ avec t le nombre de positions tâchées. Dans cet exemple, ce nombre de k-facteurs partagés est égal à $\lfloor \frac{13}{3} \rfloor - 3 = 1$. Nous constatons qu'il existe effectivement un 3-facteur partagé (dont une position possible est représentée par un rectangle horizontal).

une cinquième séquence, etc. Ce phénomène est assez étonnant, car nous pourrions penser que le fait d'ajouter une séquence distante de d substitutions avec les $(r-1)$ séquences déjà considérées pourrait ajouter jusqu'à $(r-1)d$ positions tâchées. Or ce n'est pas le cas, uniquement $\lfloor \frac{d}{2} \rfloor$ nouvelles positions peuvent devenir tâchées. Ceci est dû, comme nous le verrons dans la preuve, au fait que, nécessairement, de nombreuses substitutions se placent à des positions déjà tâchées.

Preuve 1. *Nombre de k-facteurs non chevauchant partagés*

Considérons le pire des cas concernant le nombre de k-facteurs partagés. Il s'agit du cas où ce nombre est minimal. Cette situation se présente lorsque la distance de Hamming entre chaque paire d'occurrences des (L, r, d)-

répétitions parmi r est égale à d, et lorsque les substitutions sont réparties uniformément sur l'ensemble des r séquences.

Nous rappelons que pour un alignement de séquences basé sur la distance de Hamming, une position tâchée correspond à une position où au moins une des séquences présente une substitution par rapport aux autres.

Nous observons que $p_r = \lfloor \frac{L}{k} \rfloor - \phi(r)$*, où* $\phi(r)$ *représente le nombre maximum de positions tâchées pour r séquences.*

Par définition, $\phi(2) = d$ *pour une* $(L, 2, d)$*-répétition. Nous montrons maintenant par récurrence sur r qu'en ajoutant une séquence à une* (L, r, d)*-répétition contenant au moins 2 séquences, nous pouvons ajouter au plus* $\lfloor d/2 \rfloor$ *positions tâchées comme nous l'avons mentionné ci-dessus. Formellement, nous montrons que* $\phi(r) \leq \phi(r-1) + \lfloor \frac{d}{2} \rfloor$ *pour tout* $r > 2$*.*

Nous commençons par montrer le résultat pour $r = 3$ *qui représente la base de la récurrence. Supposons que nous avons deux séquences, A et B distantes de d substitutions. L'alignement de A et B produit ainsi d positions tâchées. Nous ajoutons une nouvelle séquence C, distante de d substitutions avec A et de d substitutions avec B. Ceci signifie que potentiellement,* $2d$ *nouvelles positions peuvent être tâchées dans l'alignement de A, B et C. Cependant aux positions i déjà tâchées,* $A[i]$ *et* $B[i]$ *diffèrent impliquant que* $C[i]$ *diffère au moins avec l'une des deux. Ainsi, à chacune des d positions précédemment tâchées, une nouvelle position potentiellement tâchée est utilisée. Il ne reste donc que* $2d - d = d$ *potentielles nouvelles positions tâchées.*

Supposons qu'une nouvelle position tâchée soit localisée à une position j où $C[j]$ *est différent de, disons,* $A[j]$*. Sachant qu'à la position j, A et B sont égaux,* $C[j]$ *est aussi différent de* $B[j]$*. Ainsi, de manière similaire au comportement observé par rapport aux positions déjà tâchées, les potentielles d nouvelles positions tâchées restantes sont utilisées deux par deux. Remarquons que dans le cas où d est impair, la dernière substitution doit avoir lieu à une position préalablement tâchée, sinon, C aurait* $d + 1$ *substitutions avec A ou B. Ainsi, l'ajout de la séquence C a ajouté au plus* $\lfloor \frac{d}{2} \rfloor$ *nouvelles positions tâchées.*

Nous prouvons à présent la récurrence dans le cas général $r > 3$*. Pour la suite de cette preuve, nous supposons que d est pair. Dans le cas d impair, la remarque effectuée pour* $r = 3$ *reste vraie.*

Nous notons $A_1, \ldots, A_{r-1}$ *les séquences déjà alignées. Soit* A_r *la nouvelle séquence que nous considérons. Soit* $\phi(r-1)$ *le nombre de positions tâchées dans l'alignement de* $A_1, \ldots, A_{r-1}$*. En appliquant le même raisonnement que précédemment, au moins* $\phi(r-1)$ *des positions tâchées pouvant être ajoutées*

par A_r sont situées aux mêmes positions que les positions précédemment tâchées. Par hypothèse de récurrence, $\phi(r-1) \leq d + (r-3)d/2$, et ainsi il y a uniquement

$$(r-1)d - \phi(r-1) \leq (r-1)d - d - (r-3)d/2 = (r-7)/2 \times d$$

potentielles nouvelles positions pouvant être tâchées. Une nouvelle position j tâchée suppose alors que $A_r[j]$ diffère d'une des séquences précédentes. Cependant à la position j, toutes les $r-1$ précédentes séquences sont égales. Ainsi, les $(r-7)/2 \times d$ potentielles positions tâchées sont utilisées par groupe de $r-1$, donc il y a en fait au plus $\frac{(r-7)}{2(r-1)} \times d \leq d/2$ nouvelles positions tâchées.

Nous avons montré que $\phi(r) - \phi(r-1) \leq \lfloor \frac{d}{2} \rfloor$ pour tout $r > 2$, et donc que $p_r = \lfloor \frac{L}{k} \rfloor - \phi(2) - (r-2) \times \lfloor \frac{d}{2} \rfloor = \lfloor \frac{L}{k} \rfloor - d - (r-2) \times \lfloor \frac{d}{2} \rfloor$. Ce qui complète la preuve ■

4.2.1 Application à MONO-NIMBUS ou MULTI-NIMBUS.

Nous ouvrons ici une parenthèse pour remarquer que les propriétés que nous avons énoncées sur les (L, r, d)-répétitions sont valables aussi bien pour des (L, r, d)-répétitions apparaissant dans une unique séquence que pour des (L, r, d)-répétitions dont les occurrences sont réparties sur r séquences. Or, dans le cas de MONO-NIMBUS les occurrences des (L, r, d)-répétitions détectées se trouvent dans une unique séquence.

Du théorème 1, nous déduisons la condition nécessaire suivante que nous utilisons comme condition de filtrage multiple.

Condition 4. *Nombre de k-facteurs non chevauchant partagés dans une (L, r, d)-répétition*

$\mathcal{C}_4$ = Un filtre multiple de (L, r, d)-répétitions conserve tout ensemble de r chaînes de caractères de longueur L partageant au moins p_r k-facteurs avec

$$p_r = \left\lfloor \frac{L}{k} \right\rfloor - d - (r-2) \times \left\lfloor \frac{d}{2} \right\rfloor .$$

Nous présentons maintenant les solutions retenues pour rechercher le plus efficacement possible les ensembles de r chaînes de caractères de longueur au plus L partageant au moins p_r k-facteurs.

4.3 Algorithme appliquant la condition $\mathcal{C}_4$

Comme nous l'avons mentionné, deux versions proches de l'algorithme NIMBUS ont été créées. Nous commençons par présenter MULTI-NIMBUS recherchant des répétitions dans un ensemble de séquences distinctes. Dans la section 4.3.2 nous présentons les modifications à apporter à MULTI-NIMBUS pour obtenir MONO-NIMBUS où le filtrage s'applique aux (L, r, d)-répétitions apparaissant dans une unique séquence.

But de MULTI-NIMBUS. MULTI-NIMBUS prend en entrée les paramètres L, r et d ainsi que m séquences, avec $m \geq r$. Si celle-ci n'a pas été imposée par l'utilisateur, l'algorithme décide une valeur k à utiliser pour appliquer la condition $\mathcal{C}_4$ qui indique alors le nombre minimum p_r de k-facteurs partagés par les (L, r, d)-répétitions. La valeur de k est choisie suite à des expérimentations empiriques.

Le but de MULTI-NIMBUS est de filtrer rapidement et efficacement les séquences afin d'en supprimer les régions qui ne peuvent contenir de (L, r, d)-répétition. Ceci se fait par l'application de la condition $\mathcal{C}_4$ où l'on impose aux différentes occurrences d'appartenir à différentes séquences. Les portions de séquence respectant cette condition sont conservées alors que celles ne respectant pas cette condition sont éliminées par le filtre.

Définition 19. *p^L-ensemble*

Un ensemble de p k-facteurs apparaissant dans une portion de séquence de longueur L est appelé un p^L-ensemble.

Dans le cas de NIMBUS, L étant un paramètre fixe, nous utilisons la notation de p_r-ensemble pour désigner un p_r^L-ensemble.

MULTI-NIMBUS recherche tous les p_r-ensembles partagés par au moins r séquences parmi les m séquences prises en entrées. Toutes les positions faisant partie d'une portion de séquence contenant un p_r-ensemble répété dans au moins r séquences sont conservées par le filtre. À l'inverse, les autres positions sont supprimées par le filtre.

Utilisation de bi-facteurs. Afin de permettre une recherche efficace des p_r-ensembles répétés dans au moins r séquences, nous utilisons un nouveau type d'objet appelé les bi-facteurs*.

Définition 20. *Bi-facteur*

Un bi-facteur est formé de deux facteurs séparés par une portion de séquence non considérée que nous appelons un trou.

Formellement un ***(k, g, k′)-bi-facteur*** *est constitué de la concaténation d'un facteur de longueur k, d'un trou de longueur g et enfin d'un second bi-facteur de longueur k′. Le facteur* $s[i, i+k-1]s[i+k+g, i+k+g+k'-1]$ *est un (k, d, k′)-bi-facteur apparaissant à la position i dans une séquence s.*

Lorsque que $k = k'$*, nous utilisons la notation* ***(k, g)-bi-facteur****.*

Afin de simplifier la lecture de ce manuscrit, nous utiliserons parfois le terme de **bi-facteur***, sans spécifier k et d.*

Un exemple de bi-facteur est donné dans la figure 4.4.

Notons que dans le cas de NIMBUS, nous n'utiliserons que des (k, g)-bi-facteurs.

Notation 4. *p_r-ensemble répété*

Afin d'alléger le texte, nous utiliserons le terme de p_r-r-ensemble pour désigner un p_r-ensemble répété dans au moins r séquences parmi m.

FIG. 4.4 – Exemple de (3, 3, 2)-bi-facteur apparaissant à la position 1 de la séquence $ATATAGTTAG$. Ce bi-facteur est constitué du 3-facteur TAT et du 2-facteur TA.

Indexation des bi-facteurs. Les bi-facteurs apparaissant dans au moins r séquences parmi les m prises en entrée sont indexés dans une structure de données dédiée appelée le **tableau des bi-facteurs***. Cette structure de données est présentée dans la section 4.4. Elle permet d'obtenir en temps constant les listes des occurrences des bi-facteurs débutant par un k-facteur spécifié.

Déroulement de l'algorithme. L'idée principale pour détecter les p_r-r-ensembles est de détecter un bi-facteur possédant un «grand» trou et répété dans au moins r-séquences puis d'ajouter $p_r - 2$ k-facteurs dans ce trou.

Plus précisément, nous sélectionnons un bi-facteur dont le trou est de longueur au moins $(p_r - 2) \times k$ (permettant de contenir au moins $p_r - 2$ k-facteurs) et au plus $L - 2k$ (afin que les répétitions détectées respectent les limites de longueur L). Nous appelons ce bi-facteur le bi-facteur frontière. Un bi-facteur frontière représente par définition un 2-r-ensemble. Nous tentons alors d'étendre ce 2-r-ensemble à un p_r-r-ensemble. Pour étendre un i-r-ensemble à un $(i+1)$-r-ensemble, nous trouvons un bi-facteur (répété au moins dans r séquences) débutant par le même k-facteur que le bi-facteur frontière. Ce nouveau bi-facteur est appelé un bi-facteur d'extension interne.

Lors de l'extension interne d'un i-r-ensemble à un $(i+1)$-r-ensemble, la longueur du trou du bi-facteur d'extension interne doit être supérieure à $(p_r - i - 1) \times k$ afin de permettre d'étendre le $(i+1)$-r-ensemble obtenu avec $p_r - i - 1$ bi-facteurs d'extension interne. D'autre part, la longueur de ce trou doit être évidemment inférieure à la longueur du trou du dernier bi-facteur d'extension interne utilisé pour créer le précédent i-r-ensemble moins k. Cette manière de procéder permet l'insertion de nouveaux k-facteurs (non-chevauchant) à gauche du dernier k-facteur inséré. La figure 4.5 donne une représentation de ces contraintes.

Les occurrences du $(i+1)$-r-ensemble résultant de l'insertion d'un nouveau k-facteur sont le résultat de l'intersection des positions du bi-facteur d'extension interne avec les positions du i-r-ensemble.

Durant l'extension d'un i-r-ensemble à un $(i+1)$-r-ensemble, si l'on ne trouve pas de bi-facteur d'extension interne envisageable, alors l'algorithme reprend les paramètres du précédent $(i-1)$-r-ensemble et tente de l'étendre à l'aide d'un autre bi-facteur d'extension interne que celui qui avait conduit au i-r-ensemble.

Lors de la procédure, si des p_r-r-ensembles sont détectés, alors les portions de séquences couvertes par ces ensembles sont conservées par le filtre.

Un exemple d'une telle construction est donnée dans la figure 4.6.

Supposons que l'on cherche des portions de séquences de longueur au plus L répétées dans au moins $r = 3$ séquences contenant au moins $p_r = 3$ k-facteurs partagés.

Lors de la première étape, un bi-facteur composé des k-facteurs appelés

A et B, apparaissant dans les séquences 1, 3, 4, 5 et 7 est sélectionné. Notons que ce bi-facteur apparaît effectivement dans au moins $p_r = 3$ séquences sans quoi il n'aurait pas été sélectionné. Lors de la seconde étape, un bi-facteur, commençant par le même k-facteur A et ayant un second k-facteur appelé C est utilisé comme bi-facteur d'extension interne. Notons que ce bi-facteur possède une longueur de trou d'une part assez courte pour que les k-facteurs B et C ne se chevauchent pas, et d'autre part assez longue pour accepter par la suite l'insertion de nouveaux k-facteurs (cf. figure 4.5).

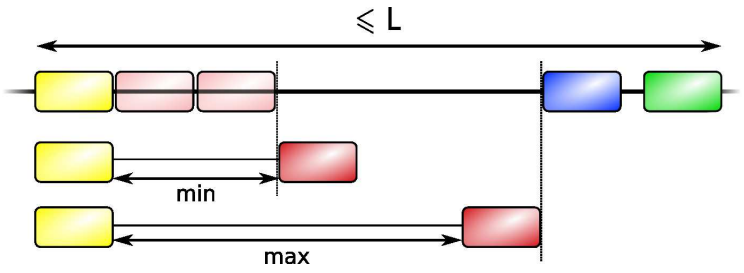


FIG. 4.5 – Contraintes sur la longueur d'un trou de bi-facteur d'extension interne. Dans cet exemple nous cherchons à créer un 6-ensemble. Nous supposons que nous avons déjà un 3-ensemble (représenté par les k-facteurs jaune, bleu et vert). Nous sommes en train d'insérer un bi-facteur d'extension interne (dont les k-facteurs sont respectivement jaune et rouge).
La longueur maximale du trou de ce bi-facteur est égale à la longueur du trou du précédent bi-facteur utilisé (représenté en jaune et bleu) moins k. Ceci permet aux k-facteurs rouge et bleu de ne pas se chevaucher.
La longueur minimale du trou de ce bi-facteur est égale à l'espace nécessaire à l'insertion des k-facteurs restants (représentés en transparence). Dans ce cas il reste deux k-facteurs. La longueur minimale du trou est donc $2 \times k$.

Une intersection des indexes des séquences des bi-facteurs $A - B$ (1, 3, 4, 5 et 7) et $A - C$ (1, 3, 5, 6 et 7) est alors effectuée pour connaître les séquences dans lesquelles apparaît le 3-3-ensemble ainsi créé (1, 3, 5, 7). Lors de la troisième et dernière étape, un nouveau bi-facteur d'insertion débutant par le même k-facteur A est ajouté au précédent 3-3-ensemble. De même, une

intersection des positions est effectuée. Il en résulte un 4-3-ensemble apparaissant dans les séquences 1, 3 et 5, respectant ainsi la condition d'apparaître dans au moins $p_r = 3$ séquences.

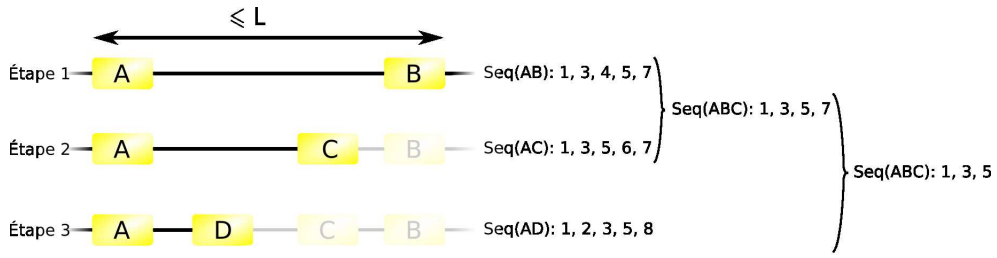


FIG. 4.6 – Exemple de construction d'un 4-3-ensemble. *Afin de simplifier la compréhension de cette figure, nous nous contentons de faire référence aux séquences dans lesquelles apparaissent les bi-facteurs et les i-r-ensembles alors que l'algorithme considère des positions dans les séquences.*

Afin d'extraire l'intégralité des p_r-r-ensembles possibles, nous réitérons l'idée décrite ci-dessus sur l'ensemble des bi-facteurs frontière possibles. Ainsi, tous les (k, g)-bi-facteurs avec $g \in [(p_r - 2)k, L - 2k]$ sont considérés comme des bi-facteurs frontière possibles d'un p_r-r-ensemble. De plus, lors de l'extension d'un i-r-ensemble à un $(i+1)$-r-ensemble, tous les bi-facteurs d'extension interne possibles doivent être testés.

* *

*

Le pseudo-code de l'algorithme associé à la détection des p_r-r-ensembles est présenté dans l'algorithme 6.

4.3.1 Commentaires à propos de l'algorithme 6.

Fonction intersection. L'algorithme 6 utilise (ligne 19) la fonction intersection qui prend en paramètre deux listes d'entiers et qui en conserve

Algorithme 6 MULTI-NIMBUS, algorithme de détection des p_r-r-ensembles

Nécessite : Valeurs p_r, L, k, r

Effectue : Positions des p_r-r-ensembles présents dans au moins r séquences.

1: **pour** g de $(p_r - 2)k$ à $L - 2k$ **faire**
2: **pour** tous les (k, g)-bi-facteurs bf **faire**
3: **si** bf débute par un nouveau premier k-facteur **alors**
4: Création des tableaux des bi-facteurs débutant par ce k-facteur
5: **fin si**
6: DÉTECTION RÉCURSIVE ($g-k$, POSITIONS(bf), 2, premier k-facteur de bf)
7: **fin pour**
8: **fin pour**
9: **renvoie** Positions obtenue par la détection récursive

DÉTECTION RÉCURSIVE

Nécessite : valeur $gmax$, $positions$, nb k-facteurs, k-facteur A

Effectue : Détecte des positions de p_r-r-ensembles en ajoutant récursivement des bi-facteurs d'extension interne.

10: **si** Nombre séquence ($positions$) $< r$ **alors**
11: stop DÉTECTION RÉCURSIVE // Pas assez de séquences
12: **fin si**
13: **si** nb k-facteurs $= p_r$ **alors**
14: Sauve positions
15: stop DÉTECTION RÉCURSIVE // p_r-r-ensemble détecté
16: **fin si**
17: **pour** g de $p_r - (nbKFactors + 1) \times k$ à $gmax$ **faire**
18: **pour** tous les (k, g)-bi-facteurs bf débutant par le k-facteur A **faire**
19: $positions$ = INTERSECTION($positions$, POSITIONS(bf))
20: DÉTECTION RÉCURSIVE ($g-k$, $positions$, nb k-facteurs $+1$, premier k-facteur de bf)
21: **fin pour**
22: **fin pour**

les éléments communs. Dans le contexte de NIMBUS, ces listes sont triées et cette intersection se fait donc en temps linéaire de la somme des longueurs des listes.

Utilisation du tableau des bi-facteurs. Ceci n'est pas explicitement spécifié dans l'algorithme 6, toutefois aux lignes 2 et 18, afin de trouver le plus rapidement possible l'ensemble des bi-facteurs frontière possibles (ligne 2) ou l'ensemble des bi-facteurs débutant par un k-facteur spécifique (ligne 18), le tableau des bi-facteurs est utilisé. Nous rappelons qu'une description de cette nouvelle structure de données est proposée dans la section 4.4.

Remarque importante à propos des bi-facteurs. Nous pouvons remarquer que lorsqu'un bi-facteur frontière est choisit (ligne 2), tous les bi-facteurs utilisés par la suite débutent par le même k-facteur (appelé A dans le pseudo-code de l'algorithme). Ainsi, pour limiter la mémoire nécessaire, il n'est pas indispensable de stocker l'ensemble des bi-facteurs d'une séquence ou d'un jeux de séquences. À un instant donné, seuls les bi-facteurs débutant par un k-facteur commun spécifique sont indexés.

* *

*

Nous ouvrons à présent une parenthèse afin de présenter les quelques nuances qui existent entre MULTI-NIMBUS et MONO-NIMBUS.

4.3.2 Filtrage de (L, r, d)-répétitions dans une séquence unique : MONO-NIMBUS

Dans le cas de MONO-NIMBUS, une (L, r, d)-répétition dans une unique séquence s sera définie comme un ensemble $\{\delta_1, \ldots, \delta_r\}$ avec $\delta_i = s[\gamma_i, \gamma_i+L-1]$, tel que $0 \leq \gamma_i \leq |s|-L$ et $\forall i, j \in [1, r]\ \mathcal{D}_h(s[\gamma_i, \gamma_i+L-1], s[\gamma_j, \gamma_j+L-1]) \leq d$.

MULTI-NIMBUS ne nécessite qu'une légère modification afin qu'il détecte non pas des (L, r, d)-répétitions réparties dans au moins r séquences mais plutôt des (L, r, d)-répétitions réparties dans une unique séquence. En effet, le théorème 1 demeure vrai que les occurrences des (L, r, d)-répétitions soient réparties dans des séquences distinctes ou non. Ainsi, dans le cas de MONO-NIMBUS, la condition $\mathcal{C}_4$ est appliquée à des portions de séquences appartenant à une unique séquence.

L'algorithme 6 ne nécessite alors qu'une modification de sa ligne 10 pour obtenir MONO-NIMBUS. Cette ligne devient alors :

si Nombre occurrences (*positions*) $< r$ **alors**

De surcroît, les bi-facteurs indexés dans le tableau des bi-facteurs sont ceux apparaissant au moins r fois au lieu de ceux apparaissant dans au moins r séquences.

* *

*

Nous présentons maintenant la structure de données appelée le tableau des bi-facteurs qui est largement utilisé dans l'algorithme 6 et qui permet un accès rapide aux positions des bi-facteurs.

4.4 Structure de données utilisée : le tableau des bi-facteurs

L'algorithme présenté dans la section précédente utilise des bi-facteurs. À plusieurs reprises, celui-ci a besoin de connaître toutes les positions d'un bi-facteur donné. Afin d'effectuer cette tâche le plus rapidement possible, les bi-facteurs sont indexés dans une structure de données appelée le tableau des bi-facteurs. Le tableau des bi-facteurs indexe ainsi l'ensemble des bi-facteurs d'une séquence unique (MONO-NIMBUS) ou d'un jeu de séquences (MULTI-NIMBUS).

Le tableau des bi-facteurs est un tableau des suffixes* (structure de données présentée dans la section 2.3.2 page 72) permettant de trier dans l'ordre lexicographique non pas des suffixes mais des (k, g)-bi-facteurs dont la structure (k et g) est fixée. Ainsi, l'ordre lexicographique prend en compte uniquement le contenu des deux k-facteurs de chaque bi-facteur et non pas le contenu du trou.

Cette structure de données permet d'accéder aux bi-facteurs commençant par un k-facteur donné en temps constant.

Cette présentation du tableau des bi-facteurs est générale dans le sens où nous présentons cette structure de données pour l'indexation de (k_1, d, k_2)-bi-facteurs. Cependant, dans le cas de NIMBUS, k_1 et k_2 désignent la même valeur et l'algorithme utilisé pour la création du tableau des (k_1, g)-bi-facteurs est légèrement simplifié par rapport à la procédure présentée dans ce qui suit. Nous indiquons dans la section 4.4.3 ces légères nuances.

* *

*

Notons qu'avant la parution de travaux simultanés [KS03, KSPP03, KA03] pour la construction du tableau des suffixes en temps linéaire, l'indexation des bi-facteurs avait été envisagée dans un arbre naturellement appelé l'**arbre des bi-facteurs**. Cette structure de données possède une complexité de création moins bonne en temps et en mémoire que celle de la création du tableau des bi-facteurs. Cependant un article présentant cette construction, intéressante d'un point de vue algorithmique, est proposée en annexe, page 201.

* *

*

Comme nous l'avons évoqué dans les commentaires à propos de l'algorithme 6, NIMBUS n'utilise à un instant donné que des bi-facteurs débutant par le même k-facteur. La méthode de construction du tableau des bi-facteurs que nous présentons dans la section suivante prend en compte cette remarque et n'indexe que les bi-facteurs débutant par un k-facteur spécifié. Cette manière de procéder présente des avantages et des inconvénients comme nous l'indiquons dans la section 4.4.4.

4.4.1 Construction du tableau des bi-facteurs pour une séquence s

La structure du tableau des bi-facteurs se base sur un tableau des suffixes pour sa construction. Nous supposons dans ce qui suit qu'un tableau des suffixes et sa colonne LCP sont déjà construits pour la séquence s.

Nous commençons cette présentation en listant les idées utilisées pour construire le tableau des (k_1, d, k_2)-bi-facteurs débutants par le k-facteur appelé A.

1. Nous donnons à chaque k_1-facteur une étiquette. Par exemple pour une séquence d'ADN avec $k_1 = 2$, le k_1-facteur AA possède l'étiquette 0, AC possède l'étiquette 1, AG l'étiquette 2 et ainsi de suite (dans la mesure où la séquence d'ADN contient l'ensemble des 2-facteurs possibles).

Ainsi, une colonne appelée $etiquette_1$ est créée contenant pour chaque suffixe l'étiquette du k_1-facteur débutant ce suffixe.

De manière similaire, si $k_1 \neq k_2$ (ce qui n'est pas le cas pour les bi-facteurs indexés pour NIMBUS), une seconde colonne, appelée $etiquette_2$ stocke les étiquettes des k_2-facteurs débutant chaque suffixe.

Dans la suite de ce manuscrit, nous appelons un bi-facteur ($etiquette_1$-$etiquette_2$) un bi-facteur dont les k_1-facteurs et k_2-facteurs possèdent, respectivement, les étiquettes $etiquette_1$ et $etiquette_2$.

2. Pour chaque suffixe, l'étiquette du k_1-facteur apparaissant $k_1 + g$ positions avant la position courante est stockée. La colonne associée est appelée *prédécesseur*. Ainsi, nous considérons que chaque suffixe indexé débute le **second** k-facteur d'un (k_1, g, k_2)-bi-facteur, l'étiquette du **premier** k-facteur étant alors stockée dans la colonne *prédécesseur*.

3. Nous construisons le tableau des (k_1, g, k_2)-bi-facteurs débutant par le k_1-facteur dont l'étiquette est A comme décrit ci-dessous.

   La colonne des *prédécesseurs* est traversée de haut en bas. Lorsque celle-ci contient une $etiquette_1$ égal à A, une nouvelle position est ajoutée dans le tableau des (k_1, g, k_2)-bi-facteurs. Sachant que le tableau des suffixes est ordonné dans l'ordre lexicographique des suffixes et donc des k_2 facteurs, cette manière de procéder assure que deux bi-facteurs consécutifs débutant par le k-facteur A ainsi créés sont ordonnés par rapport à l'étiquette de leur second k-facteur.

   La construction du tableau des (k_1, g, k_2)-bi-facteurs est ainsi faite que pour chaque (k_1, g, k_2)-bi-facteur (A-B), une liste de positions correspondante est stockée.

Nous présentons maintenant plus en détails les trois étapes précédemment introduites.

Donner une étiquette aux k-facteurs. Afin de donner à chaque k_1-facteur distinct une étiquette différente, la colonne LCP est lue de haut en bas. L'étiquette du k_1-facteur correspondant aux i^{ime} suffixe dans le tableau des suffixes, appelée $etiquette_1[i]$, est créée comme suit pour tout $i \in [0, n-1]$:

$$etiquette_1[i] = \begin{cases} 0 & \text{si } i = 0 \\ etiquette_1[i-1] + 1 & \text{si } lcp[i] \leq k_1 \\ etiquette_1[i-1] & \text{sinon} \end{cases}$$

La colonne *etiquette*$_2$ est créée de manière similaire pour k_2. Dans le tableau 4.1, nous voyons un exemple de tableau des suffixes, auquel les colonnes *etiquette*$_1$ et *etiquette*$_2$ ont été ajoutées.

Exemple 6. *Calcul de l'étiquette d'un k-facteur*

Dans le tableau 4.1, nous pouvons constater que l'ensemble des etiquette$_1$ *des* k_1*-facteurs indexés des positions 1 à 6 dans le tableau 4.1 est égal à 1. Ceci signifie que toutes ces positions débutent par le même* k_1*-facteur. Il s'agit ici du 2-facteur AA.*

Lors du passage de la ligne 6 à la ligne 7, la LCP devient inférieure à $k_1(=2)$*, donc etiquette*$_1$ *est incrémentée de 1. L'étiquette 2 est alors assignée au 2-facteur AC.*

Donner à chaque position l'étiquette du k-facteur apparaissant en première position. Pour chaque suffixe, l'étiquette du k_1-facteur apparaissant k_1+g positions avant doit être connu. La colonne *prédécesseur* stocke cette information pour chaque position. Cette colonne est construite comme suit :

$$\forall\ i\ \in [0, |s|-1],\ pred[i] = etiquette\Big[\pi^{-1}\big[\pi[i]-k_1-g\big]\Big],$$

où $\pi^{-1}[p]$ représente l'index dans le tableau des suffixes où le suffixe $s[p\ldots]$ apparaît.

Exemple 7. *Calcul de prédécesseur*

Dans le tableau 4.1, la colonne prédécesseur est indiquée. Par exemple, $prédécesseur[0] = 4$*. Cette valeur correspond à l'étiquette du* k_1*-facteur débutant à la position* $12 = 15-1-2 = \pi[0]-k_1-g$*. Cette étiquette est égale à* $etiquette_1[13]$*, la valeur* 13 *correspondant à l'index du suffixe débutant à la position* 12 *détecté grâce à la colonne (non représentée)* π^{-1}*.*

Le tableau ainsi agrémenté des colonnes *etiquette*$_1$, *etiquette*$_2$ et *prédécesseur* est appelé le tableau des suffixes complété.

Création du tableau des bi-facteurs débutant par un k_1-facteur spécifique. Le tableau des bi-facteurs contient dans chacune de ses lignes le couple (*etiquette*$_1$, *etiquette*$_2$) correspondant au bi-facteur indexé.

Dans notre cas, nous faisons le choix de n'indexer en même temps que les bi-facteurs débutant par le même k_1 facteur que nous appelons A. Ainsi,

i	π	Suffixe associé	LCP	$etiquette_1$	$etiquette_2$	*prédécesseur*
0	15	A	0	0	0	4
1	14	AA	1	1	1	4
2	0	$AAACA\ldots$	2	1	2	$\emptyset$
3	4	$AAACC\ldots$	3	1	2	1
4	1	$AACAA\ldots$	2	1	3	$\emptyset$
5	5	$AACCA\ldots$	3	1	3	2
6	9	$AACCC\ldots$	4	1	3	2
7	2	$ACAAA\ldots$	1	2	4	$\emptyset$
8	6	$ACCAA\ldots$	2	2	5	3
9	10	$ACCCA\ldots$	3	2	5	4
10	13	CAA	0	3	6	3
11	3	$CAAAC\ldots$	3	3	6	1
12	8	$CAACC\ldots$	3	3	6	3
13	12	$CCAA$	1	4	**7**	**1**
14	7	$CCAAC\ldots$	4	4	**7**	**1**
15	11	$CCCAA\ldots$	2	4	8	3

TAB. 4.1 – Tableau des suffixes complété par les colonnes $etiquette_1$, $etiquette_2$ et *prédécesseur* pour le texte $AAACAAACCAACCCAA$ pour des (2, 1, 3)-bi-facteurs.
Nous pouvons noter lignes 13 et 14 que les valeurs des $etiquette_2$ et *prédécesseur* sont égales. Ceci signifie qu'aux positions correspondantes (7 et 12), débute un 3-facteur représentant le second k-facteur d'un (2,1,3)-bi-facteur (1-7) répété. Nous en déduisons qu'aux positions 4 et 9 débute un bi-facteur répété ce qui est effectivement le cas comme nous le vérifions avec le bi-facteur AA-CCA débutant à ces deux positions.

chaque ligne contient un couple (A, $etiquette_2$) et la liste des occurrences de ce bi-facteur. Ce tableau est obtenu à l'aide des informations contenues dans le tableau des suffixes complété. La construction se base sur l'observation que pour tout $i \in [0, |s|-1]$, le tableau des suffixes complété contient l'information que le bi-facteur ($pr\acute{e}d\acute{e}cesseur[i] - etiquette_2[i]$) apparaît à la position $\pi[i] - k_1 - g$. Ainsi, pour construire le tableau des bi-facteurs, nous traversons la colonne *prédécesseur* de haut en bas. Lorsque *prédécesseur*[i] vaut A, soit un

nouveau bi-facteur $(A - etiquette_2[i])$ est créé avec comme première position $\pi[i] - k - g$, soit ce bi-facteur $(A - etiquette_2[i])$ avait déjà été rencontré et la nouvelle position $\pi[i] - k - g$ est ajoutée dans la liste des occurrences de ce bi-facteur.

Exemple 8. *Remplissage du tableau des bi-facteurs.*

Un exemple de tableau des bi-facteurs débutant par le k_1-facteur AA est donné dans la table 4.2. Ce tableau indexe les (2, 1, 3)-bi-facteurs pour le texte $AAACAAACCAACCCAA$ dont le tableau des suffixes complété a été donné dans la table 4.1.

La première ligne de ce tableau contient le bi-facteur $(1 - 2)$ apparaissant à la position 1. En effet, la première valeur de prédécesseur égale à 1 (étiquette de AA) est détectée ligne 3 dans le tableau 4.1. Donc le bi-facteur associé débute par le k-facteur AA, son second k_2-facteur possède l'étiquette $etiquette_2[3] = 2$ (correspondant à AAA) et il apparaît dans le texte à la position $1 = 4 - 2 - 1 = \pi[3] - k_1 - g$.

i	$etiquette_1 - etiquette_2$	(2, 1, 3)-bi-facteur associé	position(s)
0	1-2	$AA - AAA$	1
1	1-6	$AA - CAA$	0
2	1-7	$AA - CCA$	9, 4

TAB. 4.2 – Exemple de tableau des (2,1,3)-bi-facteurs débutant par le 2-facteur AA pour le texte $AAACAAACCAACCCAA$.

4.4.2 Complexité de création d'un tableau des bi-facteurs

Nous étudions à présent la complexité de création des tableaux des bi-facteurs. Étant donné que la complexité est la même qu'il s'agisse de MONO-NIMBUS ou de MULTI-NIMBUS, nous employons le terme générique de NIMBUS et considérons que la longueur totale de la séquence ou des séquences d'entrée est n.

Complexité en espace. La complexité en espace de création d'un tableau des bi-facteurs est en $O(n)$. En effet, les structures utilisées sont toutes linéaires en espace. De plus, pour être plus précis, jamais plus de quatre colonnes ne sont nécessaires simultanément. Ainsi, la mémoire effective utilisée par un tableau des bi-facteurs est de $16 \times n$ octets.

Complexité en temps. Les trois étapes de création du tableau des bi-facteurs sont de simples traversées de colonnes du tableau des suffixes et sont donc linéaires en temps.

Toutefois, durant la dernière étape il peut être nécessaire d'obtenir les listes d'occurrences des bi-facteurs ordonnées ce qui n'est pas le cas tel que l'algorithme est décrit ci-dessus. Ainsi, lorsqu'une nouvelle occurrence d'un bi-facteur est trouvée, plutôt que d'ajouter sa position à la fin de la liste de ses positions, l'insertion se fait en place, à la suite d'une recherche dichotomique. Le tableau des bi-facteurs obtenu est alors appelé **tableau des bi-facteurs ordonné**. Pour l'analyse de complexité de création de tableaux des bi-facteurs ordonnés, nous considérons le cas moyen où les bi-facteurs apparaissent tous manière homogène dans la séquence (chaque (k_1,g,k_2)-bi-facteur apparaissant ainsi en moyenne $\left\lfloor \frac{n}{|\Sigma|^{k_1+k_2}} \right\rfloor$ fois). Ainsi, la dernière étape se fait en $O\left(n \times \log \frac{n}{|\Sigma|^{k_1+k_2}}\right)$.

Enfin, une fois le tableau des bi-facteurs construit, les informations relatives au tableau des suffixes complété deviennent inutiles et ne sont plus stockées en mémoire.

4.4.3 Utilisation de tableaux de bi-facteurs pour NIMBUS

Dans le cas de NIMBUS, afin d'accélérer les phases d'intersection de listes de positions, les tableaux des bi-facteurs utilisés sont ordonnés. D'autre part, dans le cas de NIMBUS les bi-facteurs indexés sont des (k, g)-bi-facteurs.

Cette simplification permet de n'utiliser qu'une unique colonne *etiquette* au lieu des deux colonnes $etiquette_1$ et $etiquette_2$.

Aussi le temps de construction d'un tableau des bi-facteurs pour NIMBUS est en $O\left(n \times \log \frac{n}{|\Sigma|^k}\right)$.

NIMBUS a besoin simultanément de (k, g)-bi-facteurs où g varie entre 0 et $L - 2k$. C'est pourquoi, $L - 2k$ tableaux des bi-facteurs sont créé. Ainsi, la complexité en temps liée à la construction des tableaux des bi-facteurs

est en $O\left((L-2k) \times n \log \frac{n}{|\Sigma|^k}\right)$. Sachant que pour stocker un tableau des bi-facteurs pour un premier k-facteur spécifique, la mémoire nécessaire est en $O\left(\frac{n}{|\Sigma|^k}\right)$, la quantité totale de mémoire utilisée pour stocker l'ensemble des $L-2k$ tableaux des bi-facteurs est en $O\left((L-2k) \times \frac{n}{|\Sigma|^k}\right)$.

4.4.4 Avantages et inconvénients de l'indexation limitée

Comme nous l'avons mentionné, NIMBUS utilise toujours le même premier k-facteur pour l'ensemble des bi-facteurs employés lors de la détection récursive de p_r-r-ensembles. Ainsi, plutôt que d'indexer et donc de stocker en mémoire l'ensemble des positions de tous les bi-facteurs des séquences, seules celles commençant par un premier k-facteur déterminé sont indexés. Ceci possède des avantages et des inconvénients.

Avantage. L'avantage de cette manière de procéder est qu'elle permet de limiter la mémoire utilisée pour l'indexation des bi-facteurs à $O\left((L-2k) \times \frac{n}{|\Sigma|^k}\right)$. L'indexation de tous les bi-facteurs possibles aurait une complexité en mémoire en $O\left((L-2k) \times n\right)$. Ceci permet donc de diviser par $|\Sigma|^k$ la quantité de mémoire utilisée.

Inconvénient. Toutefois ceci impose de recréer les tableaux des bi-facteurs pour chaque possible premier k-facteur (ligne 4 de l'algorithme 6). Ainsi, en ce qui concerne la création des tableaux des bi-facteurs ordonnés, la complexité en temps est en $O\left(|\Sigma|^k \times (L-2k)n \times \log \frac{n}{|\Sigma|^k}\right)$ au lieu d'être en $O\left((L-2k)n \times \log \frac{n}{|\Sigma|^k}\right)$. Ceci augmente donc d'un facteur $|\Sigma|^k$ le temps d'exécution.

4.5 Étude de la complexité de NIMBUS

Supposons que NIMBUS ait à filtrer m séquences (dans le cas de MONO-NIMBUS, $m = 1$), chacune de longueur ℓ. La longueur totale des données à filtrer est donc $n = \ell \times m$.

4.5.1 Quantité de mémoire utilisée par NIMBUS

La mémoire utilisée par NIMBUS est déterminée par l'utilisation des tableaux des bi-facteurs. Comme nous l'avons mentionné plus haut, pour filtrer des répétitions de longueur L, cette complexité est en $O\left((L-2k) \times \frac{n}{|\Sigma|^k}\right)$.

4.5.2 Complexité en temps de NIMBUS

Temps utilisé pour la création du tableau des bi-facteurs. Comme nous l'avons vu précédemment, le temps de création des tableaux des bi-facteurs est en $O\left((L-2k) \times n \times \log \frac{n}{|\Sigma|^k}\right)$. Dans NIMBUS, afin de réduire la quantité de mémoire utilisée par ces tableaux, nous avons fait le choix de créer ces tableaux pour chaque premier k-facteur possible. Ainsi, la complexité complète de création des tableaux des k-facteurs (effectuée à la ligne 4 de l'algorithme 6) est en $O\left(|\Sigma|^k \times (L-2k) \times n \times \log \frac{n}{|\Sigma|^k}\right)$.

Temps utilisé par l'algorithme NIMBUS. Nous débutons ce paragraphe par une importante mise en garde. L'analyse de la complexité en temps présentée ici est une approximation grossière de la borne supérieure. En effet nous supposons que le temps nécessaire pour la procédure récursive de l'extraction de l'algorithme 6 dépend uniquement du nombre de k-facteurs à insérer. Cette supposition est en pratique fausse car d'autres paramètres viennent diminuer le temps nécessaire. En effet, en fonction de la longueur du trou autorisé, de la répartition des k-facteurs et de la densité de répétitions effectives dans la ou les séquences, le temps d'exécution diminue largement.

Afin d'analyser la complexité en temps, nous notons $T(nb\_kfacteurs)$ le temps nécessaire à l'exécution de la procédure récursive de l'algorithme 6 en fonction du nombre de k-facteurs déjà insérés. Avec une telle notation, le temps total d'exécution de NIMBUS est en $O\left(L \times \ell \times T(2)\right)$.

Pour tout $nb\_kfacteurs < p_r$:

$$T(nb\_kfacteurs) = \underbrace{\underbrace{L}_{\text{longueur trou}} \times \underbrace{\min(|\Sigma|^k, \ell)}_{\text{bi-facteurs d'extension interne}}}_{\text{remplacé par } Z \text{ dans la suite}} \times \left(\underbrace{n}_{\text{intersections}} + \underbrace{T(nb\_kfacteurs+1)}_{\text{Appel récursif}} \right)$$

$$
\begin{aligned}
T(2) &= Z \times (n + T(3)) = Z \times n + Z \times T(3) \\
&= n \times Z + Z \times \big(Z \times n + Z \times T(4)\big) = n \times (Z + Z^2) + Z^2 \times T(4) \\
&\vdots \\
&= n \times \sum_{i=1}^{p_r - 2} Z^i + Z^{p_r - 2} T(p_r) \\
&= n \times \frac{Z^{p_r - 1} - Z}{Z - 1} + Z^{p_r - 2} \qquad \text{(Notons que } T(p_r) = O(1)\text{)} \\
T(2) &= O(n \times Z^{p-1})
\end{aligned}
$$

Ainsi, le temps total d'exécution est en $O\left(L \times \ell \times n \times Z^{p_r - 1}\right)$ avec $Z = L \times \min(|\Sigma|^k, \ell)$.

Comme nous l'avons mentionné, et comme cela se vérifie dans la figure 4.7, ce résultat est une borne supérieure grossière du pire des cas.

Par exemple, et pour donner une intuition de la complexité moyenne, nous ne prenons pas en compte dans ce calcul le fait que $T(i)$ décroît avec l'augmentation de i. Ceci est dû au fait que la longueur possible des trous décroît, diminuant ainsi le nombre envisageable de bi-facteurs d'extension interne.

D'autre part, si nous considérons des séquences très peu diversifiées d'entropie très faible, très peu d'appels à la fonction intersection seront effectués mais cette fonction aura effectivement une complexité en temps d'exécution proche de $O(n)$.

À l'inverse, si l'on considère que les bi-facteurs distincts sont uniformément répartis le long des séquences, l'intersection des listes (ligne 19 de l'algorithme 6) est faite en temps $O\left(\frac{n}{|\Sigma|^k}\right)$. Dans ce cas, la complexité globale d'extraction chute à $O\left(L \times \ell \times \frac{n}{|\Sigma|^k} \times Z^{p_r - 1}\right)$.

Ces considérations sont confirmées par les tests effectués, présentés dans la section 4.6.

Complexité en temps de NIMBUS. Si l'on prend en plus en compte la construction des tableaux des bi-facteurs, la complexité en temps complète de NIMBUS sur des séquences non dégénérées (composées de caractères uniformément répartis) est en :

$$
O\left(|\Sigma|^k \times (L - 2k) \times n \times \log \frac{n}{|\Sigma|^k} + L \times \ell \times \frac{n}{|\Sigma|^k} \times Z^{p_r - 1}\right)
$$

avec $Z = L \times \min(|\Sigma|^k, \ell)$.

Limitation du temps d'exécution. Sans pour autant diminuer la complexité théorique en temps de NIMBUS, nous utilisons une *astuce* pour limiter dans certains cas le temps nécessaire à l'exécution de NIMBUS.

Supposons que l'utilisateur emploie NIMBUS dans le but de filtrer des répétitions ayant une grande fréquence d'apparition (r grand) et un fort taux de substitutions (d grand). Dans ce cas, afin d'imposer que le nombre p_r de k-facteurs partagés soit suffisamment grand (au moins supérieur à zéro) pour permettre le filtrage, la valeur de k doit être assez faible (inférieure à 4 par exemple). Or une faible valeur de k augmente la complexité théorique comme nous venons de le voir précédemment mais aussi le temps d'exécution en pratique comme nous vérifions dans la section 4.6.

Afin de limiter ce problème, dans ce type de situation, l'algorithme est appliqué en deux passes :

- Lors de la première passe, NIMBUS est utilisé en forçant r à être égal à 2 et ce quelque soit la valeur demandée par l'utilisateur. La condition de filtrage est plus faible que celle demandée initialement par l'utilisateur et permet d'appliquer NIMBUS avec des valeurs de k plus grandes. Cette première passe permet de filtrer grossièrement mais rapidement les données proposées par l'utilisateur.
- Durant la seconde passe, sur les séquences pré-filtrées, NIMBUS est utilisé avec les paramètres initiaux spécifiés par l'utilisateur. Les données étant déjà pré-filtrées l'importance du temps d'exécution dû à une faible valeur de k se font moins sentir.

Nous appelons cette procédure la **double passe**. Les résultats trouvés avec ou sans la double passe, sont strictement identiques.

4.6 Tests de «consommation» et de spécificité

Dans cette section, nous présentons des résultats de tests effectués dans le but d'estimer les ressources utilisées par NIMBUS en ce qui concerne le temps et la mémoire consommés. Ces tests ont été effectués sur un Pentium 3, à 1,2 GHz possédant 512 Mb de mémoire sur un prototype du programme codé en C. D'autre part, des tests permettant d'estimer la spécificité de NIMBUS sur des séquences aléatoires, aussi bien que sur des séquences d'ADN réelles, sont présentés.

Nous commençons par des résultats cherchant uniquement à étudier le temps et la mémoire utilisés par NIMBUS sur des séquences générées.

4.6.1 Tests d'utilisation des ressources

Les tests proposés ici permettent de mettre en évidence qu'en pratique, les ressources utilisées sont inférieures aux complexités théoriques donnée dans la section 4.5.

Tests en fonction du type de séquences filtrées. Les résultats présentés dans la figure 4.7 montrent le temps et la mémoire utilisés en fonction du type de données filtrées.

Nous pouvons observer que la mémoire utilisée est linéaire avec la longueur des données à filtrer, sans différence en fonction du type de données. Cependant, nous pouvons noter que les courbes représentant la mémoire utilisée en fonction de la longueur des données présentent un saut aux alentours de 16 Mb. Nous expliquons ce phénomène par la gestion de la mémoire du langage C. Notons qu'en moyenne la mémoire utilisée représente environ 28 fois la longueur des séquences prises en entrée.

Le temps d'exécution dépend quant à lui du type de données à filtrer. Sur un ensemble de séquences identiques, l'intégralité des positions sont conservées par le filtre. Ceci représente la borne supérieure pour la complexité car dans ce cas, l'ensemble des bi-facteurs d'extension interne (ligne 18 de l'algorithme 6) parvient à détecter des (L, r, d)-répétitions.

À l'inverse, lorsque toutes les séquences sont distinctes, une quantité minime de bi-facteurs d'extension interne parvient à étendre les i-r-ensembles et dans ce cas, le nombre d'appels récursif est extrêmement limité. Ceci conduit, comme nous pouvons le constater dans la partie droite de la figure 4.7, à une complexité linéaire avec la longueur des données.

4.6.2 Tests de spécificité

Nous présentons dans cette section les résultats de tests axés sur la spécificité de NIMBUS. Pour différents paramètres, nous donnons le pourcentage de données initiales conservées par le filtre ainsi que, parmi elles, celles qui ne contiennent pas de répétitions respectant les contraintes spécifiées par l'utilisateur. Ces tests présentent en parallèle le temps d'extension nécessaire et la mémoire utilisée.

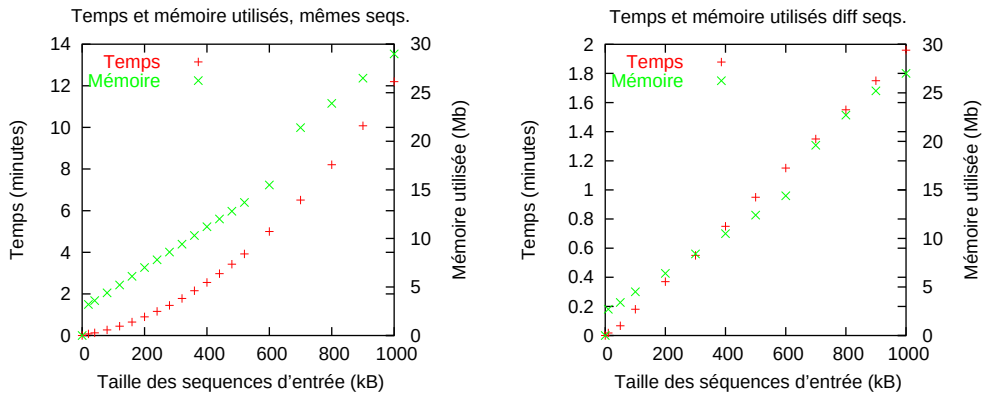


FIG. 4.7 – Temps et mémoire utilisés par NIMBUS en fonction de la longueur des données à filtrer. Les données reportées ici sont le résultat de la moyenne de 50 tests. Les séquences sont aléatoirement générées grâce à l'utilisation du langage AWK. À chaque test, 10 séquences d'ADN de longueur égales sont générées. Sur la partie gauche de la figure, les 10 séquences sont identiques, alors que sur la droite de la figure les 10 séquences sont aléatoires sans lien les unes avec les autres. Sur ces données, MULTI-NIMBUS a été exécuté avec les paramètres $L = 100$, $k = 6$, $d = 7$ et $r = 3$ ce qui impose $p_3 = 6$.

Dans le tableau 4.8, nous présentons le résultat de l'exécution du filtre sur quatre types de séquences de longueur 1 *Mb*. Les trois premières séquences sont des séquences aléatoires et contiennent, respectivement, des répétitions ayant 2, 5, et 100 occurrences de longueur 100 distantes deux à deux de 10 substitutions. Sur chacune de ces 3 séquences, nous avons utilisé MONO-NIMBUS pour filtrer des répétitions de longueur $L = 100$, apparaissant au moins $r = 2$, 3, et 4 fois et présentant au plus $d = 10$ substitutions deux à deux.

La quatrième séquence est un fragment du génome de la souche MC58 de la bactérie *Neisseria meningitidis*. Les génomes de *Neisseria* sont connus pour l'abondance et la diversité de la longueur et de la fréquence des répétitions qu'ils contiennent [T*al.*00]. La longueur des éléments répétés varie de 10 bases à plus de 30000 bases. En fonction de leur longueur, les répétitions contenues

dans cette séquence peuvent posséder plus de 200 occurrences. Ceci explique pourquoi cette souche de *Neisseria meningitidis* a déjà été utilisée [KBK03] pour des tests de programmes d'identification d'éléments répétés.

Nous avons appliqué MONO-NIMBUS à un fragment de longueur 1 Mb de cette séquence afin de filtrer des répétitions de longueur $L = 100$, apparaissant au moins $r = 2$, 3, et 4 fois et présentant au plus $d = 10$ substitutions deux à deux.

Les tests cherchant des répétitions possédant $r = 2$ occurrences ont conduit à l'utilisation de $k = 6$ apportant de bons résultats (moins de 5 minutes pour toutes les séquences aléatoires). Nous observons que sur la séquence MC58, le temps d'exécution est largement plus important (de 53 à 63 minutes) en raison du haut taux de répétitions dans cette séquence.

Pour $r = 3$ et 4, afin d'éviter que le filtre ne nécessite trop de temps, nous avons appliqué la double passe mentionnée précédemment. Les paramètres utilisés lors de la première passe étaient alors $r = 2$ et $k = 6$. La présentation des résultats concernant les temps d'exécution est alors naturellement découpée en deux parties correspondant, respectivement, au temps nécessaire pour la première passe et au temps nécessaire pour la seconde. Comme nous pouvons le constater dans le tableau 4.8, le temps utilisé pour la seconde passe est négligeable par rapport au temps utilisé pour la première passe. Ceci est dû au fait que la première passe filtre 89 à 99 % des séquences, ainsi la seconde passe s'applique à des séquences au moins dix fois plus courtes que les séquences originales. Ceci explique de même pourquoi la quantité de mémoire utilisée pour la seconde passe est négligeable par rapport à celle utilisée pour la première passe.

Avec le paramètre $r = 3$, la seconde passe utilise $k = 5$ alors que pour le paramètre $r = 4$ la valeur de k utilisée est $k = 4$. Avec une telle valeur de k, la spécificité du filtre est moindre que pour des valeurs de k supérieures. C'est pourquoi, dans le cas de MC58, un ensemble plus large de positions est conservé en filtrant des répétitions apparaissant au moins quatre fois qu'en filtrant des répétitions apparaissant au moins trois fois.

Notons que sans utiliser la double passe, sur MC58 par exemple, avec $r = 3$ le temps d'exécution est d'environ 12 heures (au lieu de 54 minutes) tout en obtenant bien évidemment le même résultat.

La spécificité observée en pratique sur les séquences dont les répétitions sont connues est très bonne (plus de 98 %). Ceci est dû au fait que les séquences sur lesquelles sont effectués ces tests contiennent des portions totalement aléatoires ponctuées par des zones contenant des répétitions distantes

d'au plus 10 substitutions. Comme nous le verrons dans la figure 4.9 sur des séquences contenant des répétitions distantes de $d + \epsilon$ (avec $\epsilon > 0$) substitutions, NIMBUS conserve certaines de ces répétitions et la quantité de taux de faux-positifs est donc plus importante.

Seq. Filtrée		2 Répétitions	5 Répétitions	100 Répétitions	MC58
Mémoire utilisée (Mb)		28	28	29	30
$r = 2$	Temps (Mn)	4.8	4.8	5	53
	Conservé	406 (0.04 %)	1 078 (0.10 %)	22 293 (2.2 %)	127 782 (12.7 %)
	Spécificité	99.97 %	99.94 %	98.76 %	Inconnu
$r = 3$	Temps (Mn)	4.8 + 0	4.8 + 0.1	5 + 0.5	53 + 0.9
	Conservé	0 (0 %)	1 078 (0.10 %)	21 751 (2.2 %)	92 069 (9.21 %)
	Spécificité	100 %	99.94 %	98.81 %	Inconnu
$r = 4$	Temps (Mn)	4.8 + 0	4.8 + 0.1	5 + 0.5	53 + 10
	Conservé	0 (0 %)	1 066 (0.11 %)	21 915 (2.2 %)	106 304 (10.63 %)
	Spécificité	100 %	99.94 %	98.8 %	Inconnu

FIG. 4.8 – Comportement de NIMBUS sur quatre types de séquences de longueur 1 Mb pour filtrer des répétitions ayant $r = 2,\ 3$ et 4 occurrences. Les trois premières séquences sont des séquences aléatoires contenant respectivement des répétitions possédant 2, 5, et 100 occurrences distantes deux à deux de 10 substitutions. Les tests effectués sur ces séquences ont été répétés 50 fois, nous présentons ici la moyenne des résultats. La quatrième séquence est une portion de la souche MC58 de *Neisseria meningitidis* de longueur 1 Mb.
Les lignes «Conservé» désignent le nombre et le pourcentage de positions conservées.

Test de performances. Afin de mettre à jour un des points faibles de NIMBUS, nous avons procédé de la manière suivante : nous avons généré aléatoirement 10 séquences de longueur 2000 contenant chacune une occurrence d'une répétition de longueur 100. Nous avons exécuté MULTI-NIMBUS sur ces séquences afin de filtrer des répétitions de longueur $L = 100$, apparaissant au moins $r = 3$ fois et présentant au plus $d = 10$ substitutions deux à deux. Notons qu'utiliser $r = 10$ aurait été la solution la plus appropriée. Cependant, avec de tels paramètres la valeur de p_r aurait été négative conduisant à un filtre inutile.

Les répétitions générées contenaient 10, 12, 14, ... 50 erreurs deux à deux,

simulant ainsi des données contenant des répétitions sont «bruitées». La figure 4.9 présente la quantité de nucléotides conservées par rapport à la quantité de nucléotides contenues dans les répétitions présentes dans les séquences, en fonction du nombre de substitutions quelles contiennent. Ces tests nous informent quant à la qualité du filtre. Un filtre «parfait», ou bien une méthode exacte conserverait exactement 100 % des répétitions (le graphique indiquerait alors la valeur 1) lorsque celles-ci contiennent d substitutions (ou moins) et aucune répétition ne serait conservée lorsque ce nombre est supérieur à d (le graphique indiquerait alors 0).

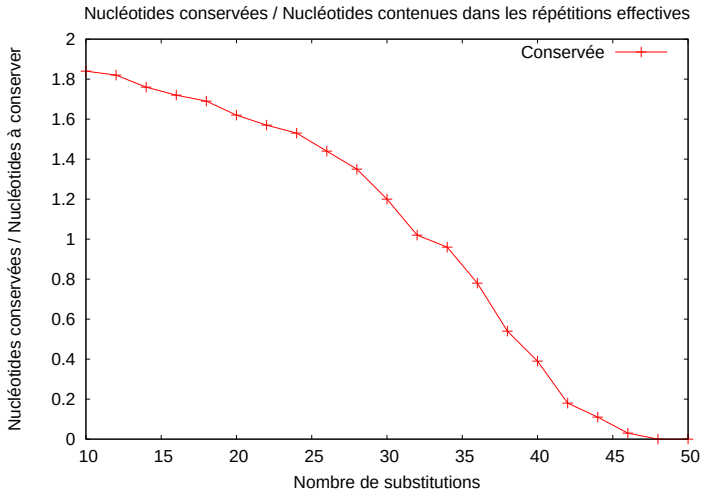


FIG. 4.9 – Nombre de nucléotides conservées par NIMBUS par rapport au nombre de nucléotides contenues dans les répétitions présentes dans les séquences, en fonction du nombre de substitutions séparant les occurrences de ces répétitions. Dix séquences ont été filtrées, chacune contenant exactement une occurrence d'une répétition. Chaque valeur est le résultat de la moyenne de 50 tests.

Discussion à propos de la performance de NIMBUS

Les résultats indiqués dans la figure 4.9 ne présentent pas NIMBUS sous son meilleur angle. En effet, la probabilité de détecter des faux-positifs est grande car nous recherchons des répétitions possédant trois occurrences dans des ensembles de séquences où les répétitions possèdent effectivement dix occurrences. La probabilité que trois mots respectent la condition nécessaire est grande car il existe de nombreux ensembles possibles de trois mots.

Nous avons donc effectué les mêmes tests sur des ensembles de trois séquences au lieu de dix. Les résultats sont présentés dans la figure 4.10. Sur cette figure, nous constatons que le filtre est plus efficace lorsque les données sont constituées de 3 séquences que lorsqu'elles sont constituées de 10 séquences. Par exemple, pour les occurrences des répétitions distantes en effet de 30 substitutions, lorsque les occurrences sont recherchées dans trois séquences, NIMBUS en conserve 28 %. Lorsque celles-ci sont recherchées dans dix séquences, NIMBUS conserve l'intégralité des répétitions effectivement présentes dans les séquences.

Toutefois, quel que soit le nombre d'occurrences de répétitions présentes dans les séquences, ces résultats mettent tout de même en avant le fait que la condition de filtrage $\mathcal{C}_4$ conduit à un important taux de faux-positifs. Par exemple, nous constatons sur cette dernière figure que lorsque les occurrences des répétitions présentes dans les séquences sont deux à deux distantes de 25 substitutions, NIMBUS, filtrant pourtant des répétitions distantes d'au plus 10 substitutions, les conserve toutes.

La condition de filtrage multiple $\mathcal{C}_4$ introduit un résultat théorique intéressant. Cependant, comme nous l'avons constaté, cette condition s'avère décevante dans certains cas. Ceci s'explique par le fait que le nombre minimum de k-facteurs partagés par les répétitions décroît rapidement avec le nombre de répétitions recherchées.

* *

*

Malgré ces considérations, NIMBUS s'applique très efficacement pour l'accélérer l'exécution d'algorithmes destinés à la détection de longues répétitions ou bien à l'alignement multiple local. Nous présentons dans les deux sections suivantes le résultat de tests visant à utiliser NIMBUS pour l'accélération de tels programmes.

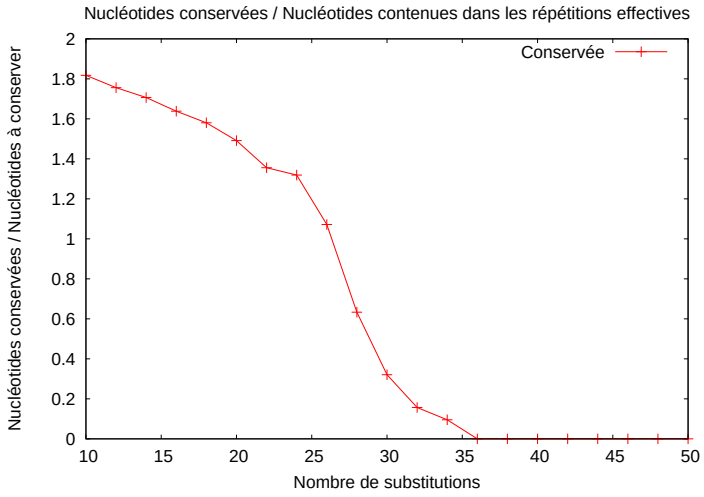


FIG. 4.10 – Nombre de nucléotides conservées par NIMBUS par rapport au nombre de nucléotides contenues dans les répétitions présentes dans les séquences, en fonction du nombre de substitutions séparant les occurrences de ces répétitions. Trois séquences ont été filtrées, chacune contenant exactement une occurrence d'une répétition. Chaque valeur est le résultat de la moyenne de 50 tests.

4.7 Utilisation du filtre pour accélérer la détection de longues répétitions

Afin de tester la capacité de NIMBUS à accélérer les algorithmes d'inférence de longues répétitions, nous avons utilisé un algorithme d'extraction de «motifs structurés» (mentionné dans la section 2.2.3 page 59) appelé RISO [CFOS05]. Nous avons employé cet algorithme sur un ensemble de 6 séquences d'ADN pour rechercher des motifs de longueur 40 apparaissant dans chaque séquence avec au plus 3 substitutions par rapport à un modèle. Un résultat a été obtenu par RISO seul en 230 secondes. En filtrant précédemment les données par NIMBUS, ce même calcul a été effectué en 0,14 secondes alors que le temps de filtrage a été de 1.1 secondes. Les résultats trouvés par

RISO sur les données filtrées ou sur les données non filtrées sont bien entendu les mêmes.

Ainsi, la procédure complète pour l'obtention de tels résultats a été réduite grâce à l'utilisation de NIMBUS de 230 secondes à 1.24 secondes, conduisant ainsi à un algorithme 185 fois plus rapide.

* *

*

Nous présentons à présent une application de NIMBUS pour l'accélération d'un programme d'alignement multiple local.

4.8 Utilisation de NIMBUS pour l'accélération d'un algorithme d'alignement multiple local

Comme nous l'avons déjà mentionné dans le chapitre 2, l'utilisation de la programmation dynamique pour trouver une solution exacte lors de l'alignement multiple local de m séquences de longueur n est effectué en temps $O(2^m n^m)$ et en espace $O(n^m)$.

En pratique, les possibilités d'application d'algorithmes d'alignements multiples locaux sont fortement limitées par une telle complexité. Ainsi, les méthodes recherchant des solutions exactes ne sont utilisées que pour des petites séquences. Cette constatation a conduit à la création de multiples heuristiques comme cela est présenté dans la section 2.2.4. Une alternative pour diminuer le temps d'exécution tout en conservant une méthode exacte est d'employer NIMBUS pour filtrer les séquences afin de supprimer toutes les portions non pertinentes, c'est-à-dire les portions trop distantes pour faire partie de l'alignement local final. Ensuite, l'algorithme d'alignement utilise les séquences filtrées au lieu des séquences initiales.

Nous avons appliqué cette technique pour accélérer un alignement multiple local effectué grâce à l'algorithme GLAM [FHSW04]. Nous avons créé un ensemble de 5 séquences aléatoires de longueur totale 1 *Mb*. Chaque séquence contenait exactement une occurrence d'une répétition de longueur 100 avec au plus 10 substitutions deux à deux.

Sur ces données, GLAM a trouvé un résultat après plus de 10 heures de calcul. Sur ces mêmes données, NIMBUS a filtré les séquences en approximativement 5 minutes. GLAM a alors été utilisé sur ces séquences filtrées pour trouver le même résultat, cette fois-ci en 15 secondes. Ainsi, le temps complet de la procédure d'alignement multiple local a été réduit grâce à l'utilisation de NIMBUS de plus de 10 heures à moins de 6 minutes conduisant à une procédure plus de 100 fois plus rapide.

Toutefois, il est nécessaire de rappeler que les données générées pour ce test contenaient des répétitions dont les occurrences étaient distantes par des substitutions uniquement. Si ces séquences avaient été des séquences d'ADN réelles, ou bien si des insertions ou des suppressions avaient été ajoutées dans les répétitions présentes dans les séquences, les résultats trouvés par GLAM sur les séquences initiales ou filtrées auraient été différents.

4.9 Conclusion

Nous avons présenté un filtre permettant de supprimer rapidement de larges portions de séquences d'ADN ne contenant pas de répétitions. Les paramètres des répétitions considérées par NIMBUS sont définis par l'utilisateur. Celui-ci peut spécifier la longueur ainsi que la fréquence des répétitions recherchées. De plus, il spécifie le nombre maximum de substitutions entre chaque paire d'occurrence des répétitions.

Les premiers résultats présentés dans ce chapitre sont encourageants et permettent de traiter en temps raisonnable des séquences d'ADN de l'ordre d'un mégabase. Ce filtre peut être utilisé entre autres comme une phase de pré-traitement des données pour un alignement multiple local. Sur des séquences artificielles, nous avons, grâce à l'utilisation de NIMBUS, obtenu un alignement multiple local en 6 minutes à la place de plus de 10 heures.

Malgré ces premiers résultats positifs, NIMBUS se base sur une condition qui ne prend pas en compte les possibles insertions et délétions qui peuvent avoir eu lieu entre les répétitions. D'un point de vue biologique, malgré l'avancée proposée par NIMBUS en matière de filtrage multiple, cette restriction est trop importante pour permettre aux biologistes d'utiliser cet outil de manière efficace lors d'études sur les répétitions dans les séquences.

C'est pourquoi les idées développées pour NIMBUS ont étés mises à profit pour créer un second filtre, appelé ED'NIMBUS, ayant exactement la même

finalité que NIMBUS mais en acceptant les insertions et délétions entre les paires de répétitions filtrées. Autrement dit, ED'NIMBUS filtre des séquences pour en extraire des répétitions basées sur la distance d'édition.

D'autre part, comme nous l'avons évoqué dans la section 4.6.2, l'idée d'appliquer une condition de filtrage directement sur un ensemble de séquences est théoriquement élégante mais conduit parfois à certains résultats décevants concernant la spécificité.

Comme nous le constaterons dans le chapitre suivant, non seulement la condition de filtrage utilisée par ED'NIMBUS est distincte de celle employée par NIMBUS, mais en plus, l'application algorithmique conséquente est elle même très dissemblable dans le cas d'ED'NIMBUS.

Chapitre 5

Ed'Nimbus : filtre pour la distance d'édition

Nous présentons dans ce chapitre le second filtre de séquences créé durant cette thèse. Le but de ce filtre de séquences, appelé Ed'Nimbus, est de filtrer des séquences pour en extraire certaines répétitions multiples, approchées dans le cadre de la distance d'édition et dont les spécifications sont données par l'utilisateur.

Le filtre présenté ici se différencie du filtre Nimbus, présenté dans le précédent chapitre, par le type de distance autorisée entre les occurrences des répétitions qu'il conserve. Nimbus a été créé dans le but de filtrer des séquences pour en extraire des répétitions multiples dont la distance maximale est basée sur un nombre de substitutions uniquement. Dans le cas d'Ed'Nimbus, la distance maximale entre les occurrences des répétitions est basée sur une distance d'édition, ainsi bornée par un nombre de substitutions mais aussi d'insertions et de délétions.

L'application présentée dans ce chapitre se différencie évidemment par la condition de filtrage mais, comme nous le verrons par la suite, elle se différencie également largement par l'algorithme associée ainsi que par la qualité de ses résultats.

* *

*

Nous débuterons ce chapitre par une section présentant le but de l'algorithme Ed'Nimbus ainsi que la condition nécessaire utilisée pour le filtrage.

Nous verrons dans cette première section que le filtre ED'NIMBUS existe sous deux formes appelées MULTI-ED'NIMBUS et MONO-ED'NIMBUS. Par la suite, nous présenterons dans la section 5.2 l'algorithme utilisée pour MULTI-ED'NIMBUS. La section suivante (section 5.3), présente les quelques différences qui existent entre MULTI-ED'NIMBUS et MONO-ED'NIMBUS dans l'algorithme.

Nous proposons dans la section 5.4 une analyse de complexité.

La section 5.5 est ensuite destinée à exposer un ensemble de tests permettant d'estimer les qualités d'ED'NIMBUS ainsi qu'à observer le comportement du programme en fonction des paramètres utilisés.

Nous présentons des résultats préliminaires quant à une application biologique possible d'ED'NIMBUS dans la section 5.6

Les filtres NIMBUS et ED'NIMBUS ayant des caractéristiques proches, nous nous proposons d'étudier les différences de performances qui existent entre ces deux filtres. Ceci est fait dans la section 5.7.

Enfin, avant de conclure, nous exposons dans la section 5.8, une présentation de l'interface web dédiée à ED'NIMBUS.

5.1 But d'ED'NIMBUS

Le but d'ED'NIMBUS est très proche de celui de NIMBUS, à savoir supprimer un maximum de portions de séquences qui ne peuvent contenir d'occurrences de (L, r, d)-répétitions. La différence réside dans le fait que, dans le cas d'ED'NIMBUS, les (L, r, d)-répétitions considérées sont des (L, r, d)-répétitions pour la distance d'édition (voir la définition 17 page 107) alors que NIMBUS est prévu pour filtrer les séquences afin de trouver des (L, r, d)-répétitions pour la distance de Hamming. Précisons, que dans ce cas précis, nous considérons pour la distance d'édition que les insertions, délétions et substitutions comptent indifféremment comme une erreur.

Aussi, dans ce chapitre, toutes les (L, r, d)-répétitions que nous allons désigner sont des (L, r, d)-répétitions pour la distance d'édition. Afin de simplifier la lecture de ce présent texte, nous emploierons simplement le terme de «(L, r, d)-répétition» pour les désigner, et ce jusqu'à la fin de ce chapitre, page 193.

5.1.1 Deux versions, MULTI-ED'NIMBUS et MONO-ED'NIMBUS

Comme ceci était déjà le cas pour NIMBUS, ED'NIMBUS se décline selon deux versions. La première, appelée MULTI-ED'NIMBUS s'applique à au moins deux séquences d'ADN dans le but de les filtrer pour détecter des répétitions dont les occurrences sont réparties sur différentes séquences. La figure 5.1 propose un aperçu du but de MULTI-ED'NIMBUS. La seconde version, que nous avons baptisée MONO-ED'NIMBUS, s'applique à une unique séquence d'ADN qui est filtré afin de détecter des répétitions ayant un nombre minimum d'occurrences dans cette unique séquence. Une représentation simplifiée de MONO-ED'NIMBUS est donnée dans la figure 5.2.

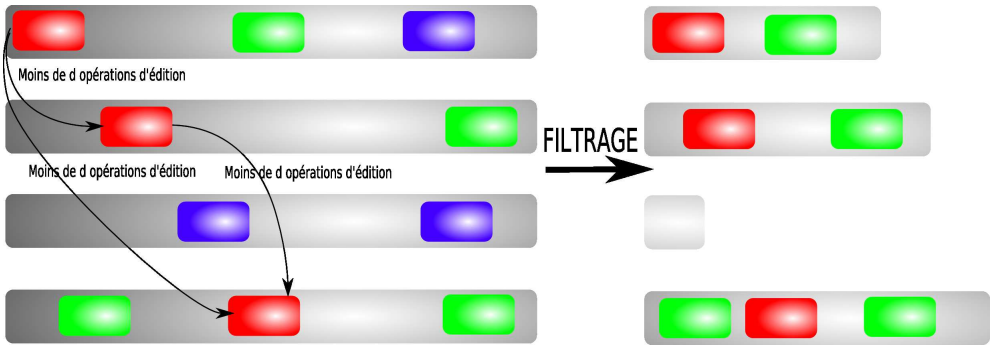


FIG. 5.1 – Aperçu de l'algorithme MULTI-ED'NIMBUS. Une ensemble de séquences contient des répétitions. Supposons ici que l'on veuille conserver des répétitions réparties dans au moins 3 séquences. Nous constatons que les répétitions vertes et rouges respectent ce critère. Elles sont alors conservées par le filtre. À l'inverse, la répétition bleue n'est répétée que dans 2 séquences et le reste des séquences ne contient pas de répétitions. Ces zones sont donc en partie supprimées par le filtre.

Les deux applications, MULTI-ED'NIMBUS et MONO-ED'NIMBUS, sont très similaires. Ainsi, plutôt que de présenter séparément ces deux versions d'ED'NIMBUS, sauf lorsque nous le mentionnons explicitement, les descriptions proposées dans ce chapitre porterons sur MULTI-ED'NIMBUS. Dans la

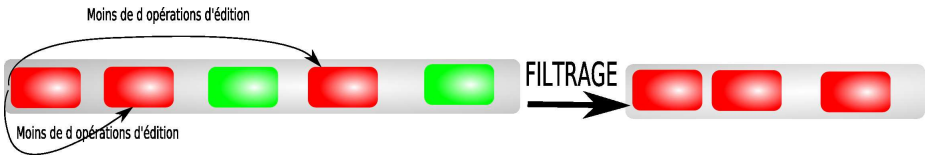


FIG. 5.2 – Aperçu de l'algorithme MONO-ED'NIMBUS. Une séquence contient des répétitions. Supposons ici que l'on veuille conserver les répétitions ayant au moins 3 occurrences dans la séquence. Nous constatons que seule la répétition dont les occurrences sont rouges respecte ce critère. Cette répétition est donc conservée par le filtre. À l'inverse, la répétition verte ne possède que deux occurrences, elle peut donc être supprimée par MONO-ED'NIMBUS.

section 5.3, nous présenterons les quelques modifications à appliquer à MULTI-ED'NIMBUS pour obtenir MONO-ED'NIMBUS.

MULTI-ED'NIMBUS**, présentation détaillée.** Supposons que nous cherchions à détecter parmi m séquences, les répétitions de longueur L ayant une occurrence dans au moins r séquences parmi m telle que la distance entre chaque paire d'occurrences soit d'au plus d opérations d'édition (insertions, délétions et substitutions). En d'autres termes, nous cherchons à détecter les (L, r, d)-répétitions dont les occurrences sont reparties dans au moins $r \leq m$ séquences. Les paramètres L, d et r, ainsi que les m séquences sont fournis par l'utilisateur. Le but de MULTI-ED'NIMBUS est de supprimer rapidement des m séquences, une majorité de positions ne pouvant faire partie d'une (L, r, d)-répétition.

De plus, comme les portions de séquences ne pouvant prendre part à une (L, r, d)-répétition sont éliminées par ED'NIMBUS, celui-ci peut éventuellement être utilisé comme une méthode approchée de détection de répétitions avec un très faible taux de faux-positifs. Dans la section 5.5.3, nous proposons une analyse des qualités de filtrage d'ED'NIMBUS confortant cette considération.

5.1.2 Idée principale de filtrage

L'idée principale de MULTI-ED'NIMBUS est basée sur la vérification d'une condition nécessaire concernant le nombre de k-facteurs possiblement chevauchants partagés par tout couple d'occurrences d'une (L, r, d)-répétition.

Pevzner et Waterman ont montré dans [PW95] que deux mots de longueur L distants d'au plus d opérations d'édition partagent au moins p k-facteurs avec

$$p = L - (d + 1) \times k + 1.$$

Ainsi, tout couple d'occurrences d'une (L, r, d)-répétition partage nécessairement $p = L - (d + 1) \times k + 1$ k-facteurs possiblement chevauchants.

De plus, nous nous basons sur l'observation évidente que l'ordre de ces p k-facteurs est conservé pour créer la condition de filtrage suivante.

Condition 5. *Nombre de k-facteurs possiblement chevauchants partagés par tout couple d'occurrences d'une (L, r, d)-répétition*

$\mathcal{C}_5$ = Un filtre multiple pour des (L, r, d)-répétitions pour la distance d'édition conserve tout ensemble de r chaînes de caractères $(\delta_1, \delta_2, \ldots, \delta_r)$ de longueur L tel que :

$$\forall\ i, j \in [1, r],\ \delta_i \text{ et } \delta_j \text{ partagent au moins } p \text{ } k\text{-facteurs,}$$
$$\textbf{dans le même ordre}, \text{ avec } p = L - (d + 1) \times k + 1.$$

Notons que, puisque les opérations d'insertions et de délétions sont autorisées entre les répétitions recherchées, les distances entre les k-facteurs partagés ne sont pas nécessairement conservées entre les occurrences des répétitions. Nous rappelons que ceci n'était pas le cas pour le filtre NIMBUS décrit dans le chapitre précédent.

En d'autres termes, toute occurrence d'une (L, r, d)-répétition partage deux à deux et avec chacune des $r - 1$ autres occurrences au moins p k-facteurs apparaissant dans le même ordre.

Notons que la condition $\mathcal{C}_5$ s'applique aux répétitions dans l'absolu, qu'elles soient répétées dans une unique séquence ou bien dispersées dans au moins r séquences. Aussi cette condition s'applique aussi bien à MULTI-ED'NIMBUS qu'à MONO-ED'NIMBUS.

* *

*

Dans la section suivante, nous présentons la solution retenue pour l'algorithme permettant de vérifier la condition $\mathcal{C}_5$.

5.2 MULTI-ED'NIMBUS, algorithme

Nous présentons dans cette section l'algorithme permettant l'application de la condition $\mathcal{C}_5$ sur un ensemble de séquences.

Avant d'entrer dans les détails de l'implantation de MULTI-ED'NIMBUS, nous ouvrons une parenthèse pour présenter une structure de données largement utilisée par celui-ci, nommée le *tableau des k-facteurs**.

Cette structure de données permet de retrouver en temps constant la liste des occurrences d'un k-facteur donné dans une séquence ou un ensemble de séquences.

* *

*

Après avoir présenté cette structure de données, nous proposerons, dans la section 5.2.2, un aperçu de la méthode que nous utilisons dans MONO-ED'NIMBUS. Ensuite une description détaillée de l'algorithme sera proposée dans la section 5.2.3.

5.2.1 Le tableau des k-facteurs

Le tableau des k-facteurs est un tableau permettant, pour un k-facteur donné, d'obtenir en temps constant la liste de ses occurrences dans une séquence ou un ensemble de séquences.

Notons que cette structure de données a été implicitement utilisée lors de la création du tableau des bi-facteurs exposé dans la section 4.4 du chapitre précédent. Nous présentons ici son adaptation spécifique aux besoins d'ED'NIMBUS.

Sa construction, effectuée pour une valeur fixée de k, se base sur un tableau des suffixes auquel des informations sont ajoutées.

Le tableau des k-facteurs pour une séquence s se compose entre autres d'un tableau des suffixes pour la séquence s et de la colonne *lcp* qui, rappelons-le, représente la longueur du plus long préfixe commun entre deux suffixes consécutifs du tableau des suffixes. À ces informations, viennent

s'ajouter deux colonnes, appelées *etiquette* et *index*. La colonne *etiquette* attribue à chaque ensemble de suffixes débutant par les mêmes k caractères la même étiquette. Ainsi, tout couple de suffixes débutant par une suite distincte de k caractères possède des étiquettes différentes. Cette colonne est construite en utilisant la même procédure que celle décrite pour la construction du tableau des bi-facteurs. Nous rappelons tout de même comment ceci est effectué. L'étiquette du k-facteur débutant le $i^{\text{ème}}$ suffixe du tableau des suffixes, appelé $etiquette[i]$ est construit de la manière suivante :

$$\forall\ i \in [0, |s|-1] :\ etiquette[i] = \begin{cases} 0 & \text{si } i = 0 \\ etiquette[i-1] + 1 & \text{si } lcp[i] \leq k \\ etiquette[i-1] & \text{sinon.} \end{cases}$$

La seconde colonne ajoutée au tableau des suffixes pour obtenir le tableau des k-facteurs est appelée *index*. Elle permet de retrouver en temps constant la liste des positions des occurrences d'un k-facteur donné. Le tableau *index* est construit de telle sorte que $index[i]$ représente l'index dans le tableau des suffixes de la première occurrence du suffixe débutant par le k-facteur dont l'étiquette est i. La construction de cette colonne est triviale : lors de la construction de la colonne *etiquette*, lorsque la valeur $etiquette[i]$ est incrémentée de 1, nous stockons l'information $index[etiquette[i]] = i$.

Un exemple de tableau des k-facteurs est donné dans la figure 5.3

Cette version du tableau des k-facteurs remplit le rôle qui lui est assigné. Ce dernier permet d'obtenir pour un k-facteur dont on connaît l'étiquette, la liste de ses occurrences.

Cependant, afin de limiter la quantité de mémoire utilisée, nous pouvons remarquer qu'une fois le tableau des k-facteurs créé, il n'est plus nécessaire de conserver les informations contenues dans la colonne *lcp*. De plus, les informations contenues dans π, *etiquette* et *index* peuvent être regroupées dans une unique colonne contenant, pour chaque étiquette, la liste des positions correspondantes. Cette colonne, représentant alors le tableau des k-facteurs dans son strict minimum, est appelée tkf.

Ainsi, l'utilisation d'une telle colonne permet à elle seule de fournir la liste des occurrences d'un k-facteur dont nous possèdons l'étiquette en $O(1)$.

Un exemple d'une telle colonne est donnée dans la figure 5.4.

Commentaires à propos de la construction du tableau des k-facteurs.

i	lcp	π	Suffixe associé		*etiquette*
0	0	2	**AA**CCAC$		0
1	1	6	**AC**$		1
2	2	0	**AC**AACCAC$		1
3	2	3	**AC**CAC$		1
4	0	7	**C$**		2
5	1	1	**CA**ACCAC$		3
6	2	5	**CA**C$		3
7	1	4	**CC**AC$		4

index	
0	0
1	1
2	4
3	5
4	7

FIG. 5.3 – Exemple de tableau des 2-facteurs pour le texte. $ACAACCAC$

tkf	
0	2
1	$6 \rightarrow 0 \rightarrow 3$
2	7
3	$1 \rightarrow 5$
4	4

FIG. 5.4 – Exemple de tableau des 2-facteurs réduit à la colonne tkf pour le texte $ACAACCAC$

- Pour indexer plus d'une séquence dans le tableau des k-facteurs, nous concaténons l'ensemble des séquences en une. En fonction de l'utilisation à laquelle le tableau des k-facteurs est destiné, les positions correspondantes peuvent être associées de leur numéro de séquence correspondante.
- Afin d'éviter les problèmes dus aux k-facteurs apparaissant au niveau de l'une des $k-1$ dernières positions du texte, nous considérons que le

texte est suivi d'autant de caractères spéciaux ($ par exemple), alphabétiquement plus petit que tous les autres.

- Dans le cas d'ED'NIMBUS, pour une position donnée dans la séquence, il est nécessaire de pouvoir connaître l'étiquette du k-facteur débutant à cette position. Ainsi, en plus de la colonne tkf, nous conservons les informations π et *etiquette* précédemment calculées. De plus, une colonne *inv* est construite de manière à ce que $inv[p]$ soit égal à l'index i tel que $\pi[i] = p$. Cette colonne permet de retrouver dans le tableau les informations correspondant à une position dans la séquence. La colonne *inv* est elle-même construite de manière triviale en temps linéaire en parcourant la colonne π, en assignant la valeur i à $inv\,[\pi[i]]$.
- Finalement, l'algorithme ED'NIMBUS utilise les colonnes π, *etiquette*, *inv* et tkf. Un exemple pour cet ensemble est donné dans la figure 5.5.

i	inv	π	$etiquette$
0	2	2	0
1	5	6	1
2	0	0	1
3	3	3	1
4	7	7	2
5	6	1	3
6	1	5	3
7	4	4	4

tkf	
0	2
1	$6 \rightarrow 0 \rightarrow 3$
2	7
3	$1 \rightarrow 5$
4	4

FIG. 5.5 – Exemple de tableau des 2-facteurs pour le texte $ACAACCAC$, tel qu'il est utilisé par ED'NIMBUS.

Complexité de création du tableau des k-facteurs. Nous rappelons que grâce aux travaux présentés dans [KS03, KSPP03, KA03] et [KLA+01], la mémoire utilisée ainsi que le temps de création du tableau des suffixes et de la *lcp* sont linéaires par rapport à la longueur des données à indexer. De plus, les opérations de calcul de la colonne *etiquette* et de la colonne *index* sont linéaires. Dans le cadre de l'utilisation du tableau des k-facteurs pour

ED'NIMBUS, nous ajoutons les colonnes inv et tkf. Le temps de création et la mémoire utilisée sont de même linéaires par rapport à la quantité de données à indexer.

Ainsi, le temps complet de création du tableau des k-facteurs pour une séquence s est en $O(|s|)$ en temps et en mémoire.

* *

*

Nous avons maintenant présenté les structures d'indexation utilisées par ED'NIMBUS. Dans la section suivante, nous donnons un aperçu de la méthode que nous employons pour appliquer la condition $\mathcal{C}_5$. Ensuite, dans la section 5.2.3, nous exposerons plus en détails l'algorithme appliqué.

5.2.2 Aperçu de la méthode pour MULTI-ED'NIMBUS

MULTI-ED'NIMBUS prend en entrée les paramètres L (longueur des répétitions recherchées), r (nombre minimum d'occurrences des répétitions recherchées), d (nombre maximum d'opérations d'édition autorisées entre chaque occurrence des répétitions recherchées) et k (longueur des k-facteurs). Notons que ce dernier paramètre est optionnel, si l'utilisateur ne le spécifie pas, le programme attribue la valeur 6 à k. Cette valeur est basée sur des résultats de tests (voir section 5.5.9) visant à analyser l'influence des paramètres sur les résultats. D'autre part, MULTI-ED'NIMBUS prend en entrée un ensemble de m séquences $\{s_1, s_2, \ldots, s_m\}$. Nous notons n la longueur moyenne de ces séquences et N leur longueurs cumulées. Ainsi $N = m \times n$.

Le nombre p de k-facteurs partagés par toute paire d'occurrences d'une (L, r, d)-répétition est calculé selon la formule $p = L - (d + 1) \times k + 1$ de la condition de filtrage $\mathcal{C}_5$. Nous rappelons qu'une clause supplémentaire de cette condition est que ces p k-facteurs partagés doivent nécessairement apparaître dans un ordre conservé entre les occurrences des répétitions détectées.

Une condition de filtrage allégée. La condition de filtrage $\mathcal{C}_5$ concerne l'ensemble des r^2 paires possibles d'occurrences d'une (L, r, d)-répétition. Un algorithme de filtrage appliquant strictement cette condition aurait une complexité en temps importante contenant le facteur r^2. Ainsi, pour l'application ED'NIMBUS, nous employons une condition de filtrage allégée, inspirée de la condition $\mathcal{C}_5$. Cette condition est la suivante :

Condition 6. *Nombre de k-facteurs possiblement chevauchants partagés par tout couple d'occurrences d'une (L, r, d)-répétition - Condition allégée*

$\mathcal{C}_6$ = Un filtre multiple de (L, r, d)-répétitions pour la distance d'édition conserve toute chaîne de caractères δ de longueur L, tel qu'il existe un ensemble de $r-1$ chaînes de caractères $(\delta_1, \delta_2, \ldots, \delta_{r-1})$ de longueur L telles que :

$$\forall\ i \in [1, r-1],\ \delta \text{ et } \delta_i \text{ partagent au moins } p \text{ } k\text{-facteurs}$$
$$\textbf{dans le même ordre}, \text{ avec } p = L - (d+1) \times k + 1.$$

En fonction de la situation, cette condition peut être précisée :
- Dans le cas de MULTI-ED'NIMBUS, les chaînes de caractères $(\delta_1, \delta_2, \ldots, \delta_{r-1})$ doivent nécessairement appartenir à $r-1$ séquences distinctes et distinctes de celle contenant δ.
- Dans le cas de MONO-ED'NIMBUS, les chaînes de caractères $(\delta_1, \delta_2, \ldots, \delta_{r-1})$, appartenant à une unique séquence, ne doivent pas se chevaucher.

Algorithme - Approche stricte. Une première approche pour appliquer la condition $\mathcal{C}_6$ pourrait être la suivante. Sur chaque séquence s_j prise en entrée par l'algorithme, une *fenêtre* de longueur L parcourt toutes les positions. Cette fenêtre est appelée une **fenêtre de référence**. La séquence sur laquelle se trouve cette fenêtre est alors appelée la **séquence de référence**. Pour chaque position de la fenêtre de référence, nous vérifions si les k-facteurs qu'elle contient sont partagés et dans le même ordre avec des mots de longueur L répartis sur au moins $r-1$ autres séquences.

Une telle approche devrait calculer, pour chacune des séquences s et pour chacune de ses positions $i \in [0, |s| - L]$, et pour chaque autre séquence s' et pour chacune de leurs positions $i' \in [0, |s'| - L]$, le nombre de k-facteurs partagés de $s[i, i+L-1]$ et $s'[i', i'+L-1]$ qui apparaissent dans le même ordre. Nous appelons l'algorithme associé à cette méthode **l'algorithme strict**.

Algorithme - Approche adoptée. Afin de limiter la complexité en temps due au fait de devoir considérer toutes les positions sur l'ensemble des séquences, nous appliquons le même type d'idée que celle utilisée dans [BCF+99]. Ainsi, les séquences sont partitionnées en **blocs** et le contenu de la fenêtre de référence est comparé non pas avec l'ensemble possible des mots de longueur L mais uniquement avec les blocs. Ces blocs sont de longueur $2L$ et sont

placés toutes les L positions. Ainsi, deux blocs consécutifs se chevauchent sur L positions. La figure 5.6 donne un aperçu graphique de la répartition de ces blocs. Une telle répartition de blocs assure que tout mot de longueur L se trouvant sur une séquence autre que la séquence de référence est entièrement inclus dans un bloc.

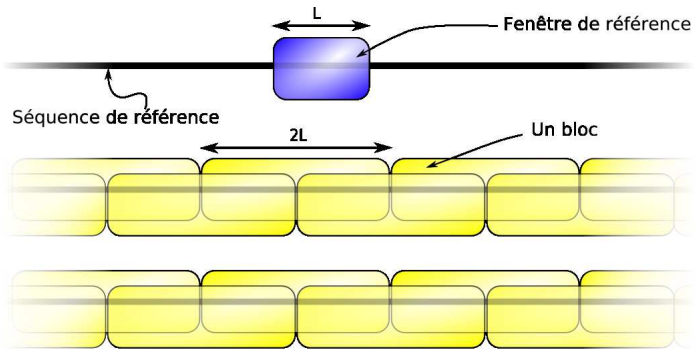


FIG. 5.6 – Représentation de la fenêtre de référence de longueur L et des blocs chevauchants de longueur $2L$.

Pour une position de la fenêtre de référence, afin de vérifier si celle-ci est conservée par l'application de la condition de filtrage $\mathcal{C}_6$, nous employons les deux étapes suivantes.

1. Lors d'une première étape, nous détectons quels blocs partagent au moins p k-facteurs avec la fenêtre de référence. Notons que durant cette étape nous ne vérifions pas l'ordre relatif des k-facteurs partagés.

   Les blocs qui partagent au moins p k-facteurs avec la fenêtre de référence sont appelés des **bons blocs**.

2. Lors d'une seconde étape, nous détectons, parmi les bons blocs, ceux qui partagent avec la fenêtre de référence au moins p k-facteurs apparaissant dans le même ordre. De tels blocs sont alors appelés des **blocs parfaits**. Dès lors que des blocs parfaits ont été détectés dans $r-1$ séquences distinctes, la fenêtre de référence est considérée comme une occurrence potentielle d'une (L, r, d)-répétition et est donc conservée

par le filtre. À l'inverse, s'il n'existe pas de blocs parfaits répartis sur $r-1$ séquences pour une fenêtre de référence, celle-ci ne peut être une occurrence d'une (L, r, d)-répétition. Elle est donc rejetée par le filtre.

Une fois que ces deux étapes sont effectuées, la fenêtre de référence est décalée d'une position vers la droite (cette fenêtre est ainsi une fenêtre glissante). Ce décalage a pour effet d'introduire un nouveau k-facteur dans la fenêtre et d'en supprimer un. Les deux étapes précédentes sont alors à nouveau effectuées, en prenant en compte la modification des k-facteurs contenus par la fenêtre de référence.

* *

*

Nous remarquons que le fait de partitionner les séquences en blocs conduit à appliquer la condition $\mathcal{C}_6$ sur une chaîne de caractères de longueur L avec un ensemble de chaînes de caractères de longueur $2L$. Or, la condition $\mathcal{C}_6$ s'applique à une chaîne de caractères de longueur L avec un ensemble de chaînes de caractères de même longueur.

Ceci conduit à une condition de filtrage légèrement plus faible que l'application stricte de la condition $\mathcal{C}_6$. Cependant, comme nous le verrons dans la section 5.5.3, les résultats de tests montrent qu'une telle approche perd très peu en qualité de filtrage tout en gagnant de manière drastique en temps de calcul par rapport à un filtre appliquant strictement la condition $\mathcal{C}_6$.

* *

*

Nous décrivons à présent l'algorithme MULTI-ED'NIMBUS plus en détail.

5.2.3 L'algorithme

Dans cette section, nous décrivons une à une les différentes étapes de l'algorithme utilisé par le filtre MULTI-ED'NIMBUS.

Construction du tableau des k-facteurs

L'algorithme débute par la construction du tableau des k-facteurs comme ceci est présenté dans la section 5.2.1. Un unique tableau des k-facteurs est créé pour l'ensemble des séquences s_j concaténées. Ceci assure l'intégrité concernant les étiquettes des k-facteurs. En effet, si les séquences avaient été indexées dans des tableaux des k-facteurs distincts, deux k-facteurs identiques de séquences différentes auraient pu avoir des étiquettes également différentes.

Utilisation de compteurs

Nous rappelons que dans l'approche adoptée dans l'algorithme ED'NIMBUS, les séquences sont découpées en blocs de longueur $2L$ apparaissant toutes les L positions.

Les blocs des séquences sont indexés de 0 à $\lfloor \frac{N}{L} - 1 \rfloor$. À chaque bloc b, un compteur est associé de telle sorte que $compteur[b]$ désigne le nombre de k-facteurs partagés entre le bloc b et la fenêtre de référence. Ainsi, le tableau *compteur* est de longueur $\lfloor \frac{N}{L} - 1 \rfloor$ et $\forall\, b \in [0, \lfloor \frac{N}{L} - 1 \rfloor]$, la valeur de $compteur[b]$ varie entre O et $L - k + 1$.

En addition au tableau *compteur*, nous utilisons un tableau permettant de connaître le nombre de bons blocs présents dans une séquence. Ce tableau, appelé *nbBonsBlocs*, est de taille m et contient dans $nbBonsBlocs[j]$ le nombre de bons blocs contenus par la séquence s_j.

Les deux tableaux, *compteur* et *nbBonsBlocs* sont initialisés à 0.

Initialisation de la fenêtre de référence

Sur la séquence s_1, la fenêtre de référence est initialement placée à la position 0. Nous rappelons que les autres séquences sont partitionnées en blocs. Le tableau *compteur* est alors rempli de la manière suivante : pour chaque k-facteur (dont nous désignons l'étiquette par $\mathcal{L}$), apparaissant à l'intérieur de la fenêtre de référence, pour chaque bloc b où $\mathcal{L}$ apparaît, $compteur[b]$ est incrémenté d'une unité.

Notons que grâce à l'utilisation du tableau des k-facteurs, nous obtenons en temps constant la liste des positions du k-facteur $\mathcal{L}$.

Le pseudo-code de l'initialisation de la fenêtre de référence est donné dans l'algorithme 7.

Durant l'incrémentation de $compteur[b]$, si sa valeur atteint p, alors le bloc b devient un bon bloc. Dans ce cas, en notant j l'index de la séquence où b apparaît, $nbBonsBlocs[j]$ est de même incrémenté. Ceci permet d'indiquer qu'un nouveau bon bloc existe dans la séquence j.

Algorithme 7 Initialisation de la fenêtre de référence

Nécessite : Le tableau des k-facteurs, les valeurs L, k, p et m
Effectue : Initialise les tableaux *compteur* et *nbBonsBlocs*

1: $\forall\, i \in [0, \lfloor \frac{N}{L} \rfloor]$ $compteur[i] \leftarrow 0$
2: $\forall\, i \in [0, m]$ $nbBonsBlocs[i] \leftarrow 0$
3: **pour** i de 0 à $L - k + 1$ **faire**
4: Utilisation du tableau des k-facteurs pour connaître le label $\mathcal{L}$ du k facteur apparaissant à la position i
5: **pour** tous les blocs b où $\mathcal{L}$ apparaît **faire**
6: incrémente $compteur[b]$
7: **si** $compteur[b] = p$ **alors**
8: incrémente $nbBonsBlocs[sequence[b]]$
9: **fin si**
10: **fin pour**
11: **fin pour**

Une fois que l'ensemble des k-facteurs appartenant à la fenêtre de référence ont été traités, il reste à détecter parmi les bons blocs détectés ceux qui possèdent au moins p k-facteurs dans un ordre conservé par rapport à cette fenêtre.

Détection des blocs parfaits parmi les bons blocs.

Lorsque le tableau *compteur* est à jour, nous vérifions qu'au moins $r - 1$ séquences contiennent au moins un bon bloc. Si ce n'est pas le cas, la fenêtre de référence n'est pas conservée par le filtre.

À l'inverse, si $r - 1$ séquences contiennent au moins un bon bloc, alors, nous détectons quels sont les blocs parfaits parmi ces bons blocs. Pour ce faire, pour chaque bon bloc appartenant à une séquence où aucun bloc parfait n'a encore été détecté, nous testons si le nombre de k-facteurs apparaissant dans le même ordre entre la fenêtre de référence et ce bon bloc est au moins égal à p. Pour effectuer ce test, nous calculons la valeur de la longueur de la **plus grande sous-séquence** entre la fenêtre de référence et le bloc considéré.

La plus grande sous-séquence entre deux mots, appelée LCS^* de par son appellation anglophone : «Longuest Common Subsequence», correspond à la plus longue sous-séquence de caractères apparaissant dans le même ordre dans les deux mots. Par exemple, entre les mots «informatique» et «biologie», la LCS est «ioie», correspondant aux lettres soulignées dans ces deux mots.

Dans le cas d'Ed'Nimbus, nous appliquons un calcul de la longueur de la LCS à la fenêtre de référence et un bloc après avoir transcrit ces deux séquences en remplaçant chacune de leurs positions par l'étiquette du k-facteur y débutant. Ce calcul permet de connaître, pour chaque bon bloc, le nombre de k-facteurs apparaissant partagés et dans le même ordre avec la fenêtre de référence.

Dans le cadre d'Ed'Nimbus, nous ne nous intéressons pas au contenu des LCS mais uniquement à leur longueur. Afin d'alléger le texte, nous emploierons la notation $|LCS|$ pour désigner la longueur d'une LCS.

Si le résultat du calcul de la $|LCS|$ atteint p entre la fenêtre de référence et un bon bloc, le bon bloc considéré devient un bloc parfait. Dans ce cas si le nombre de séquences contenant un bloc parfait reste inférieur à $r-1$, la recherche de bloc parfait se poursuit sur d'autres séquences. À l'inverse, si à la suite de ce calcul de $|LCS|$, $r-1$ séquences contiennent un bloc parfait, la fenêtre de référence est considérée comme une possible occurrence d'une (L,r,d)-répétition et elle est entièrement conservée par le filtre.

Si tous les bons blocs ont été testés et que toujours moins de $r-1$ séquences contiennent un bloc parfait, la fenêtre de référence ne peut pas être une possible occurrence de (L,r,d)-répétition. Elle n'est donc pas conservée par le filtre.

Le pseudo-code permettant de déterminer le nombre de séquences possédant un bloc parfait est donné dans l'algorithme 8.

Notons que le calcul de la $|LCS|$ (effectué à la ligne 3 de l'algorithme 8) est effectué par une fonction annexe, utilisant les méthodes de [HS77], décrites dans [Gus97].

Décalage de la fenêtre de référence.

Après avoir testé si une fenêtre apparaissant à la position i était possiblement une occurrence d'une (L,r,d)-répétition, il faut tester la fenêtre de référence apparaissant à la position $i+1$. Ceci ne demande pas de recalculer l'ensemble des compteurs des blocs en prenant en compte l'ensemble des k-facteurs présents dans la nouvelle fenêtre de référence. En effet, il suffit de

Algorithme 8 Détection des blocs parfaits parmi les bons blocs

Nécessite : Valeurs r et p
Effectue : Détecte si au moins $r-1$ séquences contiennent un bloc parfait

```
1: pour chaque bon bloc b faire
2:   si b est sur une séquence où aucun bloc parfait n'a été détecté alors
3:     si |LCS|(bon bloc, fenêtre de référence) ≥ p alors
4:       b est un bloc parfait
5:       Signale que sa séquence contient un bloc parfait
6:       Incrémente le nombre de séquences contenant un bloc parfaits
7:       si le nombre de séquences contenant un blocs parfait devient égal
         à r − 1 alors
8:         renvoie VRAI
9:       fin si
10:    fin si
11:  fin si
12: fin pour
13: renvoie FAUX
```

décrémenter le compteur de chaque bloc contenant le k-facteur de la position i qui «sort» de la fenêtre et d'incrémenter le compteur de chaque bloc contenant le k-facteur apparaissant à la position $i+1+L-k$ qui «entre» dans la fenêtre de référence.

Si après avoir décrémenté un compteur, celui-ci devient inférieur à p, ceci signifie qu'un bloc qui était un bon bloc n'est plus un bon bloc. Dans un tel cas, dans $nbBonsBlocs[j]$ (avec j correspondant à l'index de la séquence de cet ancien bon bloc) est décrémenté.

À l'inverse, si un compteur pour un bloc devient égal à p, alors, comme ceci était le cas lors de l'initialisation de la fenêtre de référence, une valeur de $nbBonsBlocs$ est incrémentée.

Le pseudo-code de la mise à jour de ces tableaux est donné dans l'algorithme 9.

À la suite de la mise à jour de ces tableaux, les calculs de $|LCS|$ sont repris comme décrit précédemment sur les bons blocs.

Algorithme 9 Mise à jour de la fenêtre de référence après décalage

Nécessite : Le tableau des k-facteurs, les valeurs L, k, p, m et i
Effectue : Met à jour les tableaux *compteur* et *nbBonsBlocs*

```
1: Utilisation du tableau des k-facteurs pour connaître l'étiquette ℒ du k facteur apparaissant à la position i
2: pour tous les blocs b où ℒ apparaît faire
3:    décrémente compteur[b]
4:    si compteur[b] = p − 1 alors
5:       décrémente nbBonsBlocs[sequence[b]]
6:    fin si
7: fin pour
8: Utilisation du tableau des k-facteurs pour connaître le label ℒ du k facteur apparaissant à la position i + 1 + L − k
9: pour tous les blocs b où ℒ apparaît faire
10:   incrémente compteur[b]
11:   si compteur[b] = p alors
12:      incrémente nbBonsBlocs[sequence[b]]
13:   fin si
14: fin pour
```

Vue complète de l'algorithme

Nous proposons dans l'algorithme 10 le pseudo-code complet de MULTI-ED'NIMBUS qui utilise les fonctions d'initialisation, de calculs de $|LCS|$ et de décalage vues précédemment.

* *

*

Avant de poursuive la description de l'algorithme ED'NIMBUS, nous ouvrons une parenthèse afin de décrire une méthode employée pour réduire le temps nécessaire à l'exécution du programme.

Accélération du temps de filtrage en pratique

Afin d'accélérer le filtrage en pratique, sans pour autant améliorer la complexité en temps théorique présentée dans la section suivante, nous appliquons l'idée suivante. Si un calcul de $|LCS|$ entre la fenêtre de référence

Algorithme 10 MULTI-ED'NIMBUS, vue globale

Nécessite : m séquences, paramètres L, d, r et k

Effectue : Supprime des zones qui ne peuvent être pas appartenir à des (L, r, d)-répétitions

1: Création du tableau des k-facteurs pour les m séquences.
2: $p \Leftarrow L - (d+1) \times k + 1$
3: Initialisation des compteurs pour la première fenêtre de référence et détection des bons blocs (Algorithme 7)
4: **si** nombre de séquences contenant un bloc parfait $\geq p$ (Algorithme 8) **alors**
5: Conserve la fenêtre
6: **sinon**
7: Supprime la position de la fenêtre
8: **fin si**
9: **pour** Chaque position de la fenêtre de référence **faire**
10: Mise à jour des compteurs - Détection des bons blocs (Algorithme 9)
11: **si** nombre de séquences contenant un bloc parfait $\geq p$ (Algorithme 8) **alors**
12: Conserve la fenêtre
13: **sinon**
14: Supprime la position de la fenêtre
15: **fin si**
16: **fin pour**
17: Écriture des résultats.

et un bon bloc est inférieur à p, disons $p - \Delta$, alors ce bon bloc ne peut pas devenir un bloc parfait avant au moins Δ décalages de la fenêtre de référence.

Ainsi, à chaque bloc, un indice est assigné permettant de savoir si, pour une position de fenêtre de référence donnée, un calcul de $|LCS|$ peu potentiellement transformer un bon bloc en bloc parfait. Si ce n'est pas le cas, le calcul de $|LCS|$ n'est pas effectué.

L'application de cette remarque permet de diminuer le temps d'exécution d'ED'NIMBUS de manière très significative sur des séquences contenant beaucoup de répétitions dont le taux de similarité est supérieur à la limite d. En effet, dans ce cas, de nombreux bons blocs sont conservés alors que peu d'entre eux sont des blocs parfaits.

* *

*

Nous présentons à présent les modifications à apporter aux procédures précédemment décrites pour obtenir le programme MONO-ED'NIMBUS.

5.3 De MULTI-ED'NIMBUS à MONO-ED'NIMBUS

La présentation que nous avons faite dans la section précédente s'appliquait à l'algorithme MULTI-ED'NIMBUS, filtrant pour rechercher des répétitions dont les occurrences sont réparties dans différentes séquences.

Afin d'obtenir MONO-ED'NIMBUS, filtre de répétitions dont les occurrences se trouvent dans une unique séquence, seules de très légères modifications doivent être appliquées aux procédures précédentes.

- Dans le cas de MONO-ED'NIMBUS, l'unique séquence est la séquence de référence. Elle est donc elle même partitionnée en blocs.
  Toutefois un point mérite d'être explicité. Considérons l'exemple représenté dans la figure 5.7. La fenêtre de référence est positionnée sur trois des blocs de la séquence. Pour éviter de comparer la fenêtre de référence avec elle-même, le bloc contenant entièrement la fenêtre de référence n'est pas pris en considération lors du calcul de la LCS (bloc représenté en rouge dans la figure 5.7). De plus, les deux blocs environnants (représentés en verts dans cette figure) sont sujet à un traitement particulier lors du calcul de la $|LCS|$. Durant ce calcul, seule la portion de longueur supérieur à $L - d$ de ces blocs ne contenant pas la fenêtre de référence est prise en considération. Ceci évite alors de détecter une répétition entre la fenêtre de référence et elle-même.
- Étant donné que dans le cas de MONO-ED'NIMBUS, il n'y a pas de contrainte quant à la position des occurrences des répétitions recherchées, il n'est plus nécessaire de maintenir un tableau *nbBonsBlocs* stockant le nombre de bons blocs contenus dans chaque séquence. Ainsi, une simple variable *nbBonsBlocs* est utilisée pour l'unique séquence.
  Ceci a pour effet de simplifier les lignes 2 et 8 de l'algorithme 7, ainsi que les lignes 5 et 12 de l'algorithme 9.
- De même que précédemment, des simplifications sont appliquées à l'algorithme 8. En effet, il n'est pas nécessaire de considérer une quantité

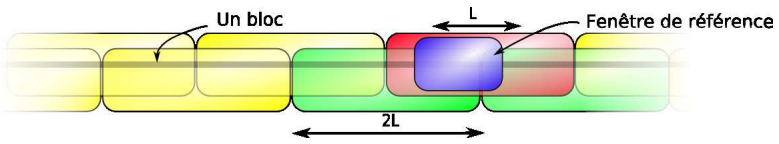


FIG. 5.7 – Représentation de la fenêtre de référence de longueur L et des blocs chevauchants de longueur $2L$ sur une unique séquence. Dans ce cas, le bloc rouge, contenant la fenêtre de référence (représentée en bleu), n'est pas pris en considération lors du calcul de la $|LCS|$. De plus les deux blocs représentés en vert sont soumis à un traitement particulier lors du calcul de la $|LCS|$.

de séquences contenant des blocs parfaits, mais uniquement une quantité de blocs parfaits dans une unique séquence. Le pseudo-code est donné dans l'algorithme 11.

Algorithme 11 Détection des blocs parfaits parmi les bons blocs - MONO-ED'NIMBUS

Nécessite : Valeurs r et p
Effectue : Détecte si au moins $r-1$ blocs parfaits existent

```
pour chaque bon bloc b faire
  si |LCS|(bon bloc, fenêtre de référence) ≥ p alors
    b est un bloc parfait
    Incrémente le nombre de blocs parfaits détectés
    si le nombre de séquences contenant un blocs parfait devient égal à
    r − 1 alors
      renvoie VRAI
    fin si
  fin si
fin pour
renvoie FAUX
```

– Notons enfin que le pseudo-code complet présenté dans l'algorithme 10 est identique dans le cas de MULTI-ED'NIMBUS ou de MONO-ED'NIMBUS à deux nuances près :

1. MONO-ED'NIMBUS utilise une unique séquence au lieu de m séquences.
2. MONO-ED'NIMBUS utilise l'algorithme 11 au lieu de l'algorithme 8 (lignes 4 et 11).

* *

*

Dans la section suivante, nous proposons une analyse de complexité conjointe des algorithmes MONO-ED'NIMBUS et MULTI-ED'NIMBUS.

5.4 Analyse de complexité

Dans cette section, nous proposons une étude générale de la complexité, qui est la même pour MONO-ED'NIMBUS ou pour MULTI-ED'NIMBUS. Nous rappelons que l'entrée de ces algorithmes est m séquences, chacune de longueur n ($m = 1$ dans le cas de MONO-ED'NIMBUS). Nous rappelons que N désigne la longueur cumulée de ces séquences.

Complexité en espace. Comme nous l'avons précédemment vu dans la section 5.2.1, la construction et le stockage du tableau des k-facteurs se fait en temps et espace linéaire (en $O(N)$). D'autre part, le tableau *compteurs* est de longueur $O\left(\left\lfloor \frac{N}{L} \right\rfloor\right)$. Enfin le tableau *nbBonsBlocs* est de longueur $O(m)$ dans le cas de MULTI-ED'NIMBUS et de longueur $O(1)$ dans le cas de MONO-ED'NIMBUS.

Ainsi, sachant que $m \ll N$, la complexité en espace d'ED'NIMBUS est en $O(N)$.

Complexité en temps. La complexité en temps que nous proposons dans cette partie distingue deux cas opposés. Le premier cas, auquel nous faisons référence par le terme «*dense*», correspond au cas où les séquences contiennent un haut taux de répétitions très similaires. Le second cas, à l'inverse, correspond au cas où les données ne contiennent que très peu de répétitions relativement distantes. Nous appelons cette deuxième situation le cas «*épars*».

D'autre part, dans le but de simplifier la lecture de cette section, nous considérons que l'ensemble des $|\Sigma|^k$ k-facteurs envisageables existent effectivement dans les séquences à filtrer. Cette supposition est réaliste dans notre cadre de filtrage de longues séquences d'ADN, où l'alphabet est limité aux quatre caractères A, C, T et G et sachant que les k-facteurs sont de longueur réduite ($k \lessapprox 8$).

Nous analysons séparément les trois algorithmes 7, 8 et 9, précédemment explicités. Notons que l'algorithme 11 possède la même complexité que l'algorithme 8.

- *Analyse de l'algorithme 7 d'initialisation de la fenêtre de référence.*
  La complexité en temps de l'algorithme 7 d'initialisation de la fenêtre de référence dépend de la ligne 3 où une boucle est effectuée $L - k$ fois. Il dépend également de la ligne 5 où une seconde boucle, interne à la première, est effectuée au plus $\left\lfloor \frac{N}{L \times |\Sigma|^k} \right\rfloor$ fois. Ainsi, la complexité de cette étape est en $O\left((L-k) \times \frac{N}{L \times |\Sigma|^k}\right)$. Sachant que $k \ll L$, cette complexité est en $O\left(L \times \frac{N}{L \times |\Sigma|^k}\right) = O\left(\frac{N}{|\Sigma|^k}\right)$. Ceci est valable à la fois dans le cas dense et épars. Notons toutefois que dans le cas artificiel où les séquences ne seraient composées que du même caractère, cette complexité serait en $O\left(L \times \frac{N}{L}\right) = O(N)$.
- *Analyse de l'algorithme 9 de décalage de la fenêtre de référence.*
  La complexité en temps de l'algorithme 9 est subordonnée aux lignes 2 et 9 où deux boucles non imbriquées sont répétées au plus $\left\lfloor \frac{N}{L \times |\Sigma|^k} \right\rfloor$ fois. Ainsi, la complexité en temps de cette étape est en $O\left(\frac{N}{L \times |\Sigma|^k}\right)$.
- *Détection du nombre de blocs parfaits par l'algorithme 8.*
  Nous rappelons que cet algorithme calcule la valeur de la $|LCS|$ de la fenêtre de référence avec une partie de ses bons blocs associés.
  Nous notons B le nombre de calculs de $|LCS|$ à effectuer. Les calculs de $|LCS|$ que nous effectuons ici sont faits entre un mot de longueur L (fenêtre de référence) et un mot de longueur $2L$ (bon bloc). Pour calculer la $|LCS|$ de deux mots u et v, la méthode que nous employons est effectuée en temps $O(\rho \log \rho')$. Ici, ρ désigne le nombre de lettres de u multiplié par leur nombre d'occurrences dans v. Ainsi, si u n'est composé que de A et que v n'est composé que de T, alors, $\rho = 0$. À l'inverse si u et v sont tous deux composés uniquement de la même lettre, $\rho = |u||v|$. Dans le cas dense $\rho = 2 \times L^2$, et dans le cas épars,

nous considérons que $\rho = L$.
La valeur ρ' est égale au résultat du calcul de la $|LCS|$, c'est-à-dire au plus L. Cependant, dans notre cas, nous cherchons uniquement à savoir si le résultat de la $|LCS|$ est supérieur à p ou non. Ainsi, ici, $\rho' = p$.
Ainsi, l'étape de détection des blocs parfaits se fait en

$$O(B \times L^2 \log p) \qquad \text{dans le cas dense}$$

et en

$$O(B \times L \log p) \qquad \text{dans le cas épars.}$$

Les étapes de décalage de fenêtre et de calcul de $|LCS|$ sont effectués N fois. Ainsi, la complexité en temps complète de l'algorithme ED'NIMBUS est la suivante :

$$\begin{aligned}
&O\Big(\underbrace{\frac{N}{|\Sigma|^k}}_{\text{Initialisation}} + N \times \big(\underbrace{\frac{N}{L \times |\Sigma|^k}}_{\text{décalage}} + \underbrace{L^2 \times \log p \times B}_{|LCS|} \big) \Big) \\
&= O\Big(\frac{N}{|\Sigma|^k} + \frac{N^2}{L \times |\Sigma|^k} + N \times L^2 \times \log p \times B \Big) \\
&= O\Big(\frac{N^2}{L \times |\Sigma|^k} + N \times L^2 \times \log p \times B \Big) \qquad \text{(cas dense)}
\end{aligned}$$

et

$$\begin{aligned}
&O\Big(\underbrace{\frac{N}{|\Sigma|^k}}_{\text{Initialisation}} + N \times \big(\underbrace{\frac{N}{L \times |\Sigma|^k}}_{\text{décalage}} + \underbrace{L \times \log p \times B}_{|LCS|} \big) \Big) \\
&= O\Big(\frac{N}{|\Sigma|^k} + \frac{N^2}{L \times |\Sigma|^k} + N \times L \times \log p \times B \Big) \\
&= O\Big(\frac{N^2}{L \times |\Sigma|^k} + N \times L \times \log p \times B \Big) \qquad \text{(cas épars)}
\end{aligned}$$

Notes à propos du nombre B de calculs de $|LCS|$. Pour une fenêtre de référence donnée, le nombre B de calculs de $|LCS|$ à effectuer est directement dépendant du nombre de bons blocs détectés. Nous notons ce nombre *nbb*. Ainsi, $B \leq nbb$.

Deux paramètres influent sur B :

1. Le nombre de bons blocs détectés :
   - Le nombre bons blocs détectés nbb peut être extrêmement faible. Dans le cas épars, nous supposons que ce nombre est inférieur à $r-1$ et que donc, aucun calcul de $|LCS|$ n'est effectué. Dans un tel cas, B est proche de 0.
   - À l'inverse, dans le cas dense, nous supposons que le pire des cas concernant le nombre de bons blocs détecté est atteint. Alors, dans une telle situation, $nbb = \lfloor \frac{N}{L} \rfloor$.
2. Le degré de similarité entre la fenêtre de référence et les bons blocs :
   - Dans le cas épars, si des calculs de $|LCS|$ sont effectués, ceux-ci conduiront à un résultat inférieur à p. Dans ce cas, l'algorithme 8 (ou 11) se déroulera jusqu'au bout.
   - Dans le cas dense, les calculs de $|LCS|$ conduiront immédiatement à un résultat supérieur à p. Ainsi, d'une part, les calculs de $|LCS|$ se termineront avant la fin du déroulement complet de l'algorithme 8 (ou 11 dans le cas de Mono-Ed'Nimbus), et d'autre part, $r-1$ blocs parfaits seront détectés avant d'avoir testé tous les bons blocs (ligne 8 de l'algorithme 8, ou ligne 6 de l'algorithme 8). Dans un tel cas, $B = r-1$.

Les considérations précédentes sont très complexes à analyser dans le cas moyen. Nous nous contentons ainsi de donner une borne supérieure grossière du pire des cas concernant le nombre de $|LCS|$ calculées. Ainsi, une borne supérieure pour B est $B = \lfloor \frac{N}{L} \rfloor$. Une telle borne supérieure conduit à une complexité en temps complète en

$$O\left(\frac{N^2}{L|\Sigma|^k} + N^2 \times L \times \log p\right) = O\left(N^2 \times L \times \log p\right) \quad \text{dans le cas dense,}$$

et en

$$O\left(\frac{N^2}{L|\Sigma|^k} + N \times L \times \log p\right) \quad \text{dans le cas épars.}$$

* *

*

Dans la section suivante, nous proposons un ensemble de tests effectués dans le but d'étudier la spécificité d'Ed'Nimbus ainsi que le temps de calcul et la mémoire nécessaires en pratique au filtrage de différences séquences d'ADN.

5.5 Tests expérimentaux

Nous proposons dans cette partie un ensemble de tests permettant d'évaluer les performance d'Ed'Nimbus, à la fois en ce qui concerne la spécificité, que les temps de calcul et la mémoire utilisée.

Une partie des tests que nous proposons (de la section 5.5.1 à la section 5.5.3 comprise) présentent en parallèle les résultats de calculs effectués par Ed'Nimbus avec les résultats de calculs effectués par l'algorithme strict mentionné dans la section 5.2.2. Comme nous l'avions évoqué, le fait de partitionner les séquences conduit à un filtre plus rapide mais aussi moins spécifique qu'un filtre strict.

Les tests que nous proposons, en plus d'apporter la possibilité d'évaluer les performances d'Ed'Nimbus, mettent en avant le fait que le gain concernant le temps d'exécution est extrêmement important par rapport à l'algorithme strict, alors que la perte de spécificité est minime.

Un ensemble de tests (de la section 5.5.5 à la section 5.5.9 incluse) permettent de visualiser le comportement d'Ed'Nimbus en fonction des différents paramètres appliqués.

L'intégralité des tests que nous présentons ont été effectués avec un Pentium 3, à 1,2 GHz possédant 512 Mb de mémoire.

5.5.1 Comparaisons des temps d'exécution, Ed'Nimbus contre l'algorithme strict

Pour comparer les temps d'exécution d'Ed'Nimbus et de l'algorithme strict, nous avons créé, sur l'alphabet $\{A, C, T, G\}$, quatre séquences aléatoires de mêmes longueurs. Dans chacune de ces séquences, nous avons placé une occurrence d'une répétition de longueur 100 distante de chacune des autres de d opérations d'édition aléatoires. Nous avons alors exécuté Ed'Nimbus et l'algorithme strict sur ces ensembles de séquences avec les paramètres $L = 100$, $d = 10$, $r = 4$ et $k = 6$, c'est-à-dire pour chercher les répétitions de taille 100, ayant 4 occurrences distantes deux à deux d'au plus 10 opérations d'édition, en utilisant des k-facteurs de taille 6. Nous avons fait varier la longueur des séquences de $n = 100$ à $n = 6000$, et donc la longueur totale des données de $N = 400$ à $N = 24000$. Nous n'avons pas effectué de tests comparés avec des séquences de longueurs plus importantes car l'algorithme strict requiert trop de temps de calcul. Chaque test a été effectué 50 fois. Les résultats moyens sont donnés dans la figure 5.8.

Dans cette figure, apparaît très clairement le fait que l'algorithme strict devient rapidement inutilisable avec l'augmentation de la longueur des séquences. À l'inverse, Ed'Nimbus conserve des temps de calcul très réduits (moins de 0.1 secondes) sur toutes les longueurs de séquences proposées dans ces tests.

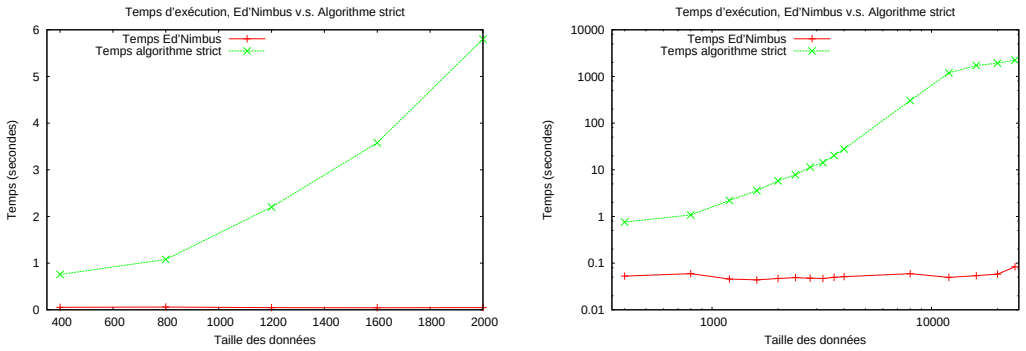


Fig. 5.8 – Comparaison en temps de l'algorithme strict et de Ed'Nimbus (Multi-Ed'Nimbus dans ce cas précis). Les données contiennent quatre séquences de longueurs égales. Les filtres ont été utilisés pour rechercher les $(100, 4, 10)$-répétitions, avec $k = 6$.
La figure de gauche présente les résultats pour des données allant de 350 à 2000 caractères. Sur la figure de droite, les mêmes tests ont été effectués pour des données dont la taille variait de 350 à 25000 caractères. Sur cette seconde figure, une échelle logarithmique a été utilisée.

5.5.2 Tests sur MC58

Nous avons réutilisé la séquence d'ADN MC58 que nous avions utilisé pour les tests de Nimbus décrits dans le chapitre précédent. Nous rappelons que cette séquence d'ADN, de *Neisseria meningitidis*, a la particularité de contenir un taux important de répétitions dont le nombre d'occurrences et la longueur sont très variables.

Les tests présentés ici ont été effectués sur une portion de longueur 1 Mb de cette séquence. En effet, le temps demandé par l'algorithme strict ne permettait pas d'utiliser des séquences de longueur trop importante[1]. Dans cette portion de séquence de MC58, nous avons employé ED'NIMBUS et le filtre strict avec les paramètres $L = 100$, $d = 10$, $r = 5$ et $k = 5$.

Ce calcul a été effectué en environ 2 minutes par ED'NIMBUS. Ces mêmes tests, effectués par l'algorithme strict, ont nécessité environ 7500 minutes, soit plus de 5 jours. Malgré cette immense différence de temps d'exécution, les résultats obtenus sont très similaires, l'algorithme strict conservant 7.2% des séquences initiales contre 7.3% pour ED'NIMBUS. Notons que la différence de mémoire utilisée par les deux méthodes conforte l'idée que la faible perte de qualité de filtrage par ED'NIMBUS est largement récompensée par le gain de consommation de ressources. Les tests précédents ont nécessité 30 Mb de mémoire dans le cas d'ED'NIMBUS contre 360 Mb dans le cas de l'algorithme strict.

* *

*

Les tests présentés ci-dessus permettent de constater les qualités d'ED'NIMBUS concernant sa faible consommation en ressources. Ils permettent en outre d'observer que la différence avec l'algorithme strict pour ce qui est des résultats obtenus est minime.

Cependant, ces tests ne permettent pas d'évaluer précisément la spécificité de ces algorithmes. Nous proposons donc dans ce qui suit des tests permettant d'analyser le comportement d'ED'NIMBUS et de l'algorithme strict sur des séquences dont les répétitions contenues dans les séquences sont connues.

5.5.3 Tests sur des séquences contenant des répétitions effectives bruitées

Nous présentons maintenant le résultat de tests permettant l'observation du comportement d'ED'NIMBUS sur des séquences contenant des répétitions connues dont le taux de substitutions est supérieur à la limite d demandée par l'utilisateur.

[1] Dans la section 5.5.4, ED'NIMBUS est appliqué, seul, à la totalité de la séquence MC58

De plus, ces mêmes tests ont été appliqués à l'algorithme strict afin de comparer les résultats de qualité de filtrage.

Ces tests ont été effectués de la manière suivante. Nous avons généré, sur l'alphabet $\{A, C, T, G\}$, un ensemble de 5 séquences de longueur 10000. Dans chacune de ces séquences, nous avons placé aléatoirement une occurrence de longueur 100 d'une répétition effective. Nous avons ainsi créé des ensembles avec des occurrences de répétitions effectives distantes de 10 à 24 opérations d'édition. Sur ces séquences, nous avons utilisé ED'NIMBUS et le filtre strict, appliqués avec $k = 5$ (valeur de k apportant une bonne spécificité) pour conserver les répétitions ayant $r = 5$ occurrences de longueur $L = 100$ et ayant une distance maximale de $d = 10$ opérations d'édition. Les résultats de ces tests sont présentés dans la figure 5.9.

Note à propos de la présentation des résultats Pour le test effectué ici ainsi que pour plusieurs autres tests à venir, nous ne présentons pas la spécificité des filtres. En effet, dès lors que la distance entre chaque paire d'occurrences des répétitions présentes dans les séquences est supérieur au paramètre d des répétitions recherchées par l'utilisateur, le filtre ne devrait conserver aucune portion de séquence. Dans une telle situation, le nombre de vrai-positifs est nul car aucune répétition présente dans les séquences ne respecte les conditions demandées par l'utilisateur. Ainsi, la spécificité n'apporte que peu d'information sur le comportement du filtrage obtenu. Une solution est alors de mesurer les qualités de filtrage en présentant ce que nous avons appelé le «taux de conservation*». Le taux de conservation indique le nombre de nucléotides conservées par ED'NIMBUS divisé par le nombre total de nucléotides appartenant aux occurrences des répétitions effectivement présentes dans les séquences (quelque soit leurs distances).

Dans les tests qui suivent, nous connaissons exactement quelles sont les répétitions effectivement présentes dans les séquences. Celles-ci sont artificiellement intégrées dans des séquences aléatoire ne contenant aucun répétition. Nous nous permettons d'affirmer que les séquences aléatoires ne contiennent pas de répétitions respectant les critères spécifié lors de l'application d'ED'NIMBUS, au regards de tests où nous avons appliqué ED'NIMBUS sur des séquences totalement aléatoire, en détectant aucune répétition dans 100 % des cas.

Si le taux de conservation est égal par exemple à 1.5, cela signifie que pour chaque occurrence de répétition effective de longueur L, $1.5L$ positions

sont conservées. À l'inverse, si le taux est égal à 0.5, ceci signifie que seule la moitié des répétitions présentes dans les séquences est conservée. Ceci peut être, par exemple, le cas lorsque les répétitions existantes contiennent plus d'erreurs que la limite demandée par l'utilisateur.

Analyse des résultats Ces résultats nous permettent deux observations :

1. Les qualités de filtrage d'Ed'Nimbus sont assez bonnes. Avec ces paramètres, l'intégralité des répétitions présentes dans les séquences, dont les occurrences sont distantes d'au moins 14 opérations d'édition de plus que la limite demandée sont supprimées. De plus, nous constatons que la moitié des répétitions présentes dans les séquences sont filtrées lorsque celles-ci contiennent 6 opérations d'édition supplémentaires par rapport à la limite fixée.
2. La différence de qualité de filtrage entre l'algorithme strict et Ed'Nimbus est minime. Ceci confirme les résultats obtenus sur la séquence d'ADN MC58. Nous constatons que, même sur des séquences bruitées, l'algorithme strict ne conduit qu'à un résultat très légèrement meilleur concernant la qualité de filtrage.

5.5.4 Tests sur de longues séquences

Afin de confirmer les résultats de tests obtenus précédemment, nous avons effectué une série de tests sur des séquences plus longues.

Ainsi, nous avons appliqué Mono-Ed'Nimbus sur la séquence MC58 entière (de longueur 2.2 Mb), ainsi que sur des séquences artificielles de même longueur. Ces séquences artificielles contenaient 3, 5 puis 100 occurrences de répétitions connues et réparties aléatoirement. La distance entre chacune des paires d'occurrences de ces répétitions était de 10 opérations d'édition. Nous avons utilisé Mono-Ed'Nimbus sur ces séquences en recherchant des $(100, r, 10)$-répétitions avec r valant successivement 3, 5, puis 10 (et dont les occurrences de longueur 100 étaient distantes d'au plus 10 opérations d'édition). Pour effectuer ces tests, nous avons utilisé le paramètre $k = 6$. Chaque test a été effectué 50 fois. Le résultat moyen est donné dans le tableau 5.1.

Ces résultats permettent de mettre en évidence le fait que les temps d'exécution sont très limités. Dans les résultats présentés ici, les temps de calcul sont de l'ordre d'une ou deux minutes, pour la séquence MC58, de longueur 2.2 Mb, contenant un taux de répétitions extrêmement important. Sur des

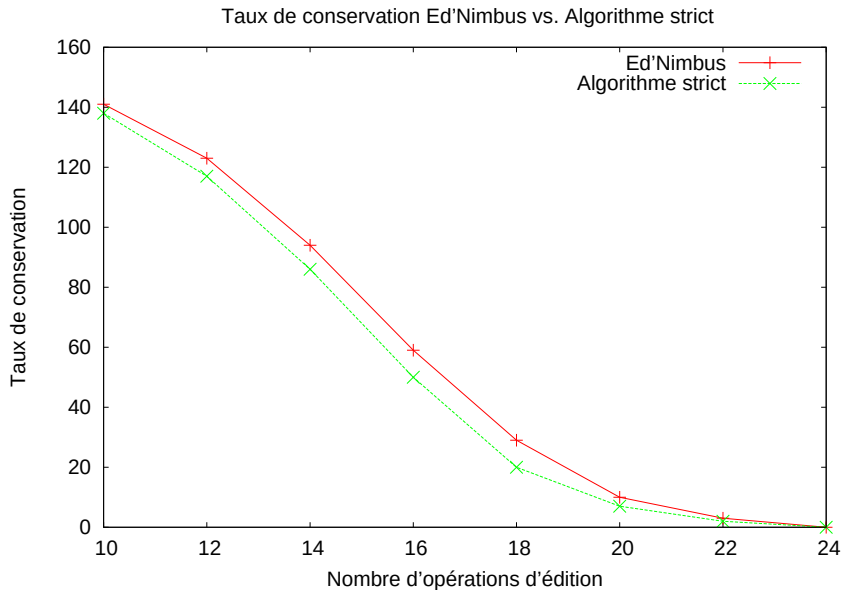


FIG. 5.9 – Taux de conservation d'ED'NIMBUS et de l'algorithme strict sur des séquences bruitées. Les deux algorithmes ont été employés pour filtrer 5 séquences de longueur 10000, contenant chacune une occurrence d'une répétition de longueur 100. Entre chaque paire d'occurrences de cette répétition, nous avons fait varier le nombre d'opérations d'édition de 10 à 24. Chaque test a été effectué 50 fois. Nous proposons sur cette figure la moyenne des résultats.

séquences de même longueur contenant moins de répétitions, les temps de calculs sont réduits à moins d'une minute.

D'autre part ces résultats permettent de constater les très bonnes performances d'ED'NIMBUS. En effet, nous constatons que pour ces séquences, le taux de faux-positifs est inférieur à 0.5 %.

* *

*

Séquence filtrée		3 Répétitions	5 Répétitions	100 Répétitions	MC58
Mémoire utilisée (Mb)		71	70	72	82
$r = 3$	Temps (Secs.)	56	56	57	94
	Conservé	545 (0.02 %)	965 (0.04 %)	21232 (0.96 %)	362156 (15.5 %)
	Spécificité	99.99 %	99.97 %	99.48 %	inconnu
$r = 5$	Temps (Secs.)	57	56	57	96
	Conservé	0 (0 %)	898 (0.04 %)	20892 (0.95 %)	207121 (9.11 %)
	Spécificité	100 %	99.98 %	99.5 %	inconnu
$r = 100$	Temps (Secs.)	57	56	56	131
	Conservé	0 (0 %)	0 (0 %)	17623 (0.80 %)	79822 (3.5 %)
	Spécificité	100 %	100 %	99.65 %	inconnu

TAB. 5.1 – Comportement d'ED'NIMBUS sur quatre types de séquences de longueur 2.2 Mb, pour rechercher des répétitions ayant $r = 3$, 5 et 100 occurrences. Les trois premières séquences sont des séquences aléatoires (sur l'alphabet $\{A, C, T, G\}$) contenant, respectivement des répétitions possédant 3, 5, et 100 occurrences, distantes deux à deux de 10 opérations d'édition. Les tests effectués sur ces séquences ont été répétés 50 fois, nous présentons ici la moyenne des résultats. La quatrième séquence est la souche MC58 de *Neisseria meningitidis* de longueur 2.2 Mb.
Les lignes étiquetées «Conservé» désignent le pourcentage de positions conservées et les lignes étiquetées «FP» désignent le taux de faux-positif.

Nous proposons dans les sections suivantes d'étudier le comportement du filtre ED'NIMBUS en fonction des paramètres.

5.5.5 Tests en fonction de la longueur des séquences, N

Afin d'évaluer le comportement d'ED'NIMBUS en fonction de la longueur des séquences, nous avons effectué des tests sur deux types de données distinctes.

Dans un premier temps, nous avons créé, sur l'alphabet $\{A, C, T, G\}$, une séquence aléatoire et nous l'avons répétée pour obtenir 10 occurrences. Ceci nous a permis de nous placer dans le cas dit «*dense*» (voir la section 5.4). Dans un second temps, sur ce même alphabet, nous avons généré aléatoirement un ensemble de 10 séquences, toutes de même longueur. Ainsi, nous nous sommes placés dans le cas dit «*épars*».

Nous avons fait varier la longueur totale des séquences de 100 Kb à 5000

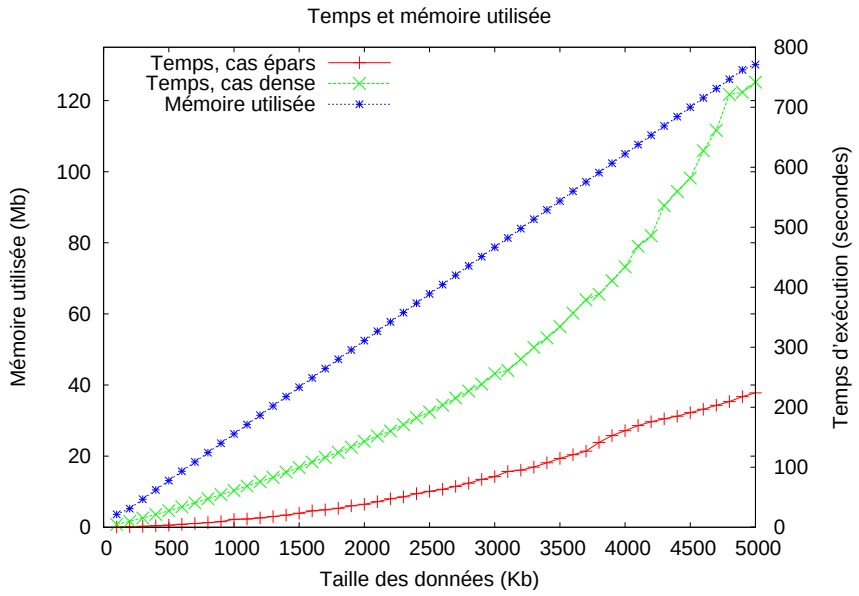


FIG. 5.10 – ED'NIMBUS, utilisation de la mémoire et temps de calcul en fonction de la longueur des séquences. MULTI-ED'NIMBUS a été utilisé pour filtrer des ensembles de 5 séquences aléatoires égales dans un cas et différentes dans l'autre. Les paramètres utilisés étaient $L = 100$, $r = 5$, $d = 10$, et $k = 5$. Chaque valeur est la moyenne de 50 tests.

Kb et nous avons reporté dans la figure 5.10, les résultat de mémoire et de temps consommés, dans les cas à la fois dense et épars.

Comme nous pouvons le constater dans cette figure, le temps d'exécution, est assez restreint dans les deux situations. Notons que dans le cas «dense», ED'NIMBUS effectue le filtrage en un peu plus de 12 minutes pour des séquences de longueur 5 Mb. Nous constatons de surcroît que dans le cas «épars», les temps de calculs sont beaucoup plus réduits par le fait que moins de calculs de $|LCS|$ sont effectués. Nous constatons par exemple que pour des séquences de longueur 5 Mb, le temps de calcul se limite à moins de 4 minutes.

Nous constatons de plus que la quantité de mémoire utilisée, est, comme

ceci est exprimé dans l'analyse de complexité (section 5.4), linéaire en la taille des données.

5.5.6 Tests en fonction de la longueur des répétitions, L

Comme nous l'avons exposé dans la section 5.4, la longueur des répétitions recherchées influe sur la complexité en temps théorique.

- Une augmentation de L diminue le temps d'initialisation et de mise à jour des compteurs (algorithmes 7 et 9).
- Une augmentation de L augmente le temps de calcul de $|LCS|$ (algorithme 8).

Afin d'observer le comportement pratique d'Ed'Nimbus en fonction de L, nous avons effectué des tests sur trois séquences aléatoires sur l'alphabet $\{A, C, T, G\}$, chacune de longueur 50000. Chacune de ces séquences contenait une occurrence d'une répétition de même longueur que la longueur des répétitions recherchées (L). Chaque paire d'occurrences était distante de $\lfloor \frac{L}{10} \rfloor$ opérations d'édition. Nous avons fait varier L de 50 à 1600, d'abord par pas de 10 (de $L = 50$ à $L = 500$), puis par pas de 20 (de $L = 500$ à $L = 1600$). Les résultats de ces expériences sont présentés dans la figure 5.11. Nous constatons dans cette figure que le paramètre L influe sur le temps d'exécution de manière importante. Nous avons distingué dans la figure 5.11 le temps utilisé par la phase d'indexation (création du tableau des k-facteurs), et la phase d'extraction des portions conservées. Nous constatons que le temps de création du tableau des k-facteurs est négligeable par rapport au temps d'extraction. De plus, nous avons distingué, parmi le temps utilisé pour l'extraction des portions conservées, le temps nécessaire à la mise à jour des compteurs du temps utilisé par les calculs de $|LCS|$.

Ceci permet de mettre en évidence le fait que l'augmentation du paramètre L influe essentiellement sur le temps de calcul de $|LCS|$. Nous observons de surcroît un comportement inattendu pour les valeurs de L entre 200 et 600 bases. Dans cet intervalle de valeurs, nous constatons une brusque augmentation du temps d'exécution avant qu'il ne baisse. Nous n'expliquons pas pour l'instant ce phénomène. Nous chercherons à l'analyser dans des travaux futurs.

Dans cette même figure 5.11, nous avons de surcroît indiqué la spécificité d'Ed'Nimbus en fonction de L. Nous constatons une baisse de la spécificité

avec l'augmentation de L. Ceci s'explique par le fait que lorsqu'une position est détectée comme démarrant possiblement une occurrence de répétition, L positions sont conservées. Ainsi, avec l'augmentation de L, même si le nombre de positions détectées à tort comme débutant une occurrence de répétition est stable, la spécificité diminue. En revanche, le taux de conservation (non indiqué sur la figure pour éviter de la surcharger) reste globalement stable avec l'augmentation de L (variant entre 1.5 et 1.7).

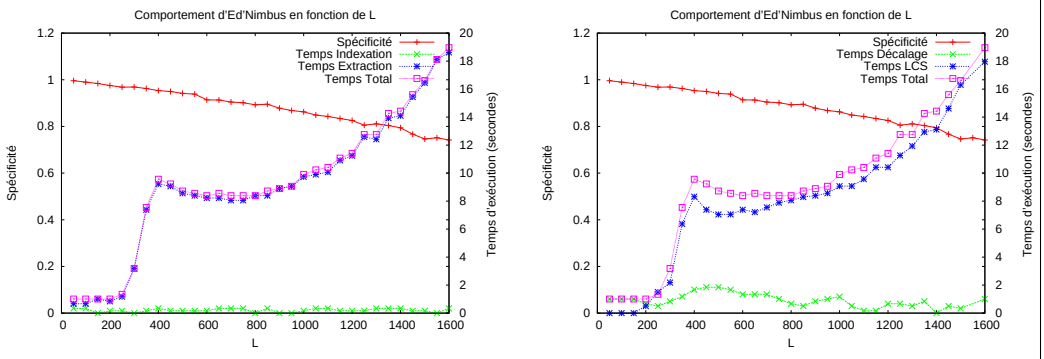


FIG. 5.11 – Comportement d'ED'NIMBUS en fonction de L. Sur l'alphabet $\{A, C, T, G\}$, trois séquences aléatoires de longueur 50000 ont été générées. Chacune de ces séquences contient une occurrence d'une répétition de même longueur que la longueur des répétitions recherchées (L). Chaque paire d'occurrences est distante de $\lfloor \frac{L}{10} \rfloor$ opérations d'édition. ED'NIMBUS a été utilisé pour filtrer ces séquences en utilisant les paramètres $d = \lfloor \frac{L}{10} \rfloor$, $r = 3$ et $k = 5$. Nous avons fait varier la valeur de L de $L = 50$ à $L = 1600$. Chaque valeur est la moyenne de 50 tests.
Dans la partie gauche de la figure, nous avons distingué le temps d'indexation (création du tableau des k-facteurs) du temps d'extraction des portions conservées.
Dans la partie droite de la figure, nous distinguons au sein du temps d'extraction le temps utilisé pour la mise à jour des compteurs du temps utilisé pour le calcul de $|LCS|$.

5.5.7 Tests en fonction du taux de substitutions, d

Le paramètre d influe sur la spécificité du filtre. En effet, moins l'utilisateur autorise d'erreurs entre chaque paire d'occurrences des répétitions recherchées, plus la condition nécessaire à appliquer est rigide. À l'inverse, si l'utilisateur autorise un nombre d'erreurs important entre les répétitions recherchées, la condition nécessaire à appliquer est plus flexible, et le taux de faux-positifs est plus important.

Du point de vue du temps d'exécution, lorsque le paramètre d est élevé (par exemple supérieur à 12 %), sachant que le nombre minimum p de k-facteurs partagés est faible (cf. tableau 5.2), de nombreux bons blocs sont détectés et de nombreux calculs de $|LCS|$ sont effectués. Ceci a pour effet d'augmenter considérablement le temps d'exécution.

Afin d'observer en pratique ces phénomènes, nous avons généré, sur l'alphabet $\{A, C, T, G\}$, trois séquences aléatoires de longueur 50000, contenant chacune une occurrence d'une répétition de longueur 100. Les occurrences étant distantes deux à deux de x opérations d'édition. Nous avons effectué des tests pour x variant de 0 à 15 et nous avons appliqué MULTI-ED'NIMBUS sur ces séquences avec les paramètres $L = 100$, $r = 3$, $k = 5$ et d variant simultanément avec x ($d = x$). Les résultats de ces expériences sont donnés dans la figure 5.12.

Les résultats présentés dans cette figure mettent en évidence l'une des limites d'ED'NIMBUS en particulier, mais aussi de l'ensemble des filtres basés sur des k-facteurs. Lorsque les répétitions recherchées sont trop dissemblables les unes des autres, ce type d'approche perd brutalement en efficacité du point de vue de la spécificité et, dans le cas d'ED'NIMBUS, du point de vue du temps d'exécution.

Nous pouvons toutefois noter que le temps moyen d'exécution augmente avec le paramètre d jusqu'à une certaine limite, avant de diminuer rapidement. Ceci s'explique par le fait que lorsque d dépasse une certaine valeur, les bons blocs sont tous des blocs parfaits. Ainsi, les calculs de $|LCS|$ sont limités à de très courte longueurs (inférieures à 10 par exemple pour $d = 19$ ou $d = 20$). C'est pourquoi, après avoir subi une forte augmentation, le temps de calcul moyen diminue brutalement. Ceci est observable dans la figure 5.13, où nous avons étendu les tests présentés dans la figure 5.12 à $d = 20$. Dans cette seconde figure, nous pouvons constater que, pour l'ensemble des paramètres choisis, un pic concernant le temps de calcul apparaît pour $d = 16$. Ceci s'explique par le fait que dans cette situation, de nombreux bons blocs

d	p
0	96
2	86
4	76
6	66
8	56
10	46
12	36
14	26
16	16
18	6
20	0

TAB. 5.2 – Valeurs de p en fonction de d, avec $L = 100$ et $k = 5$.

sont détectés, et que parmi ces bons blocs les calculs de $|LCS|$ conduisent à des résultats inférieurs mais proches du seuil p. Ainsi de nombreux de calcul infructueux de $|LCS|$ sont effectués. Ce pic correspond à la perte totale de spécificité. Toutefois, pour les valeurs de d supérieures à 16, le temps d'exécution redevient faible car les calculs de $|LCS|$ conduisent à un résultat supérieur à p. Ainsi peu de calculs de $|LCS|$ sont effectués.

5.5.8 Tests en fonction du nombre d'occurrences des répétitions présentes dans les séquences

Nous présentons dans cette section le comportement d'ED'NIMBUS en fonction du nombre d'occurrences de répétitions présentes dans les séquences.

Influence sur le temps d'exécution

Le paramètre concernant le nombre de répétitions recherché n'entre pas en compte dans le calcul de complexité. D'une point de vue algorithmique, ce paramètre n'a aucune influence sur le temps d'exécution ou sur la spécificité du filtre. Toutefois, en fonction du contenu des séquences, nous pourrions

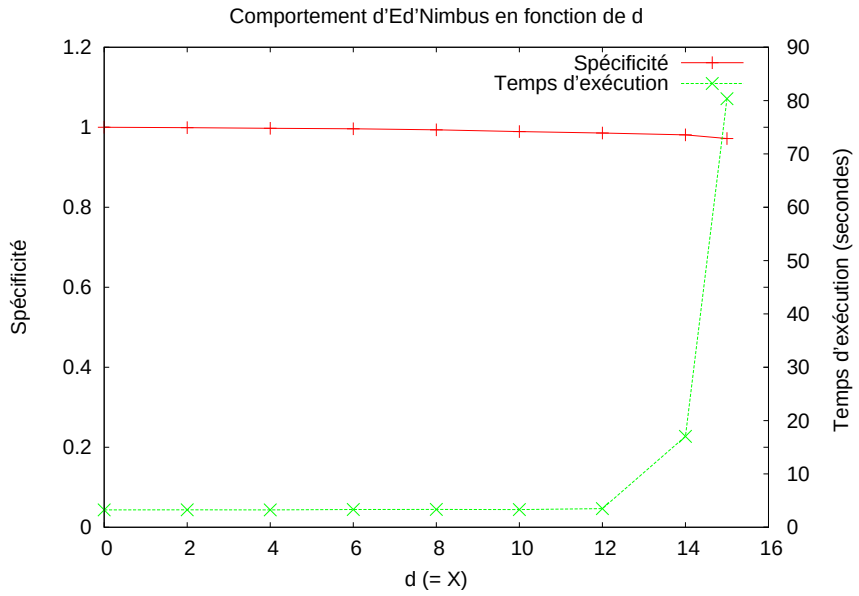


FIG. 5.12 – Comportement d'ED'NIMBUS en fonction de d. Sur l'alphabet $\{A, C, T, G\}$, trois séquences aléatoire de longueur 50000 ont été générées. Chacune de ces séquences contient une occurrence d'une répétition de longueur 100 distantes deux à deux de x opérations d'édition, X variant de 0 à 15. ED'NIMBUS a été utilisé pour filtrer ces séquences en utilisant les paramètres $L = 100$, $k = 5$ et $r = 3$. La valeur de d varie de $d = 0$ à $d = 15$, simultanément avec X ($d = X$).
Chaque valeur est le résultat moyen de 50 tests.

supposer que ce paramètre influence le temps d'exécution du programme. Or, l'influence de ce paramètre est très limitée.

En effet, supposons que le nombre minimum d'occurrences demandé par l'utilisateur soit supérieur au nombre d'occurrences des répétitions effectivement présentes dans les séquences. Dans ce cas, le nombre de bons blocs est très faible, voire nul. Ainsi, peu ou pas de calcul de $|LCS|$ sont effectués,

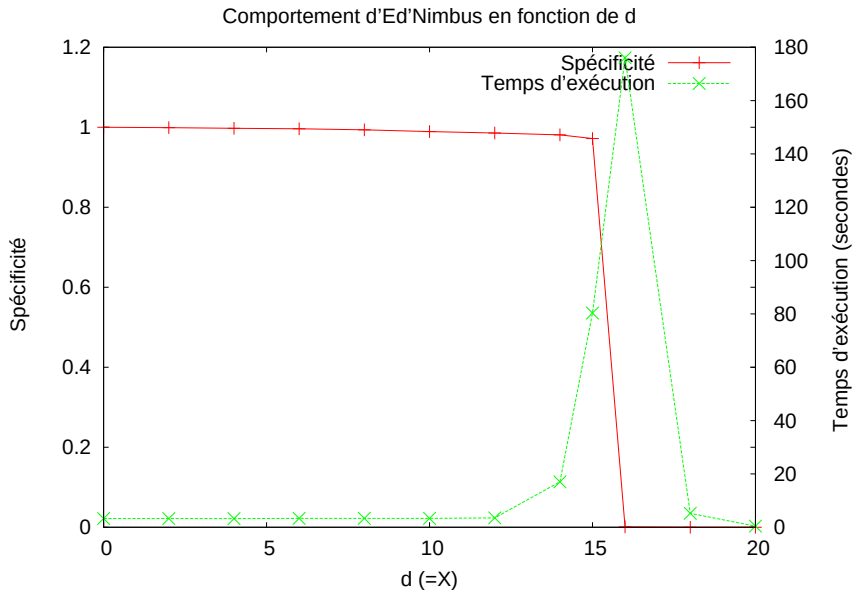


FIG. 5.13 – Comportement d'ED'NIMBUS en fonction de d. Sur l'alphabet $\{A, C, T, G\}$, trois séquences aléatoires de longueur 50000 ont été générées. Chacune de ces séquences contenait une occurrence d'une répétition de longueur 100. Les occurrences étaient distantes deux à deux de X opérations d'édition, X variant de 0 à 20. ED'NIMBUS a été utilisé pour filtrer ces séquences en utilisant les paramètres $L = 100$, $k = 5$ et $r = 3$. La valeur de d varie de $d = 0$ à $d = 20$, simultanément avec X ($d = X$).
Chaque valeur est le résultat moyen de 50 tests.

conduisant à un faible temps de calcul.

À l'inverse, si le nombre de répétitions existant dans les séquences est supérieur au nombre r de répétitions demandées par l'utilisateur, le nombre de bons blocs est important. Cependant, dans ce cas, dès que le bon nombre de blocs parfait est atteint, les calculs de $|LCS|$ sont stoppés, conduisant ainsi à un faible temps de calcul. Ainsi, le temps d'exécution est très peu dépendant du nombre du nombre de répétitions dans les séquences.

Ces considérations sont vérifiées dans la figure 5.14 où l'on peut consta-

ter que l'augmentation du nombre effectif de répétitions dans les séquences n'influe pas sur le temps de calcul.

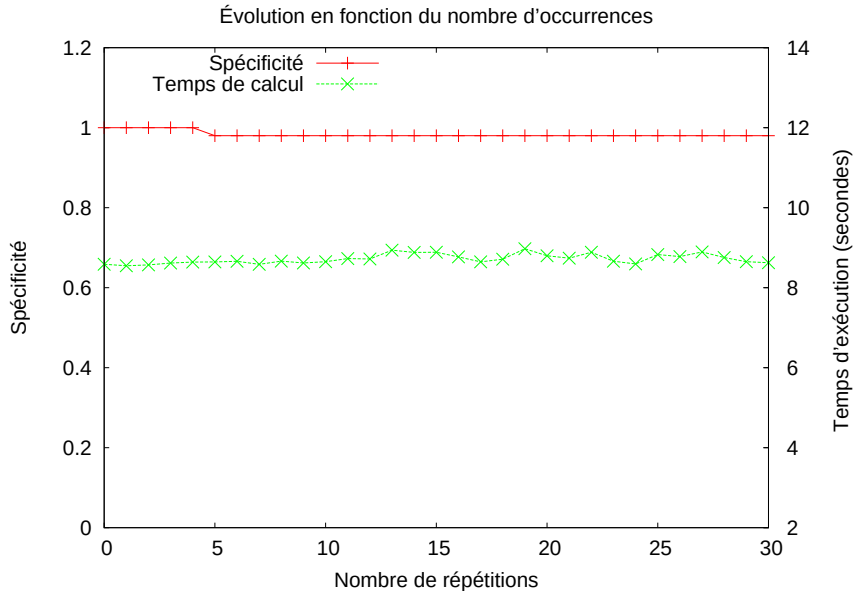


FIG. 5.14 – Comportement d'ED'NIMBUS en fonction du nombre d'occurrences des répétitions présentes dans les répétitions. Sur l'alphabet $\{A, C, T, G\}$, un ensemble de cinq séquences aléatoires de longueur totale 50 Kb a été généré. Une répétition ayant entre $x = 0$ à $x = 30$ occurrences de longueur 100, distantes par paires de 10 opérations d'édition a été introduite dans les séquences (à partir de $x = 5$, chaque séquence possède au moins une occurrence de la répétition).
MULTI-ED'NIMBUS a été utilisé pour filtrer ces séquences en utilisant le paramètre $k = 5$ pour rechercher des répétitions ayant au moins $r = 5$ occurrences de taille $L = 100$, distantes deux à deux d'au plus $d = 10$ opérations d'éditions.
Chaque valeur est le résultat moyen de 50 tests.

Influence sur la spécificité

Nous constatons dans la figure 5.14 que le nombre d'occurrences des répétitions présentes dans les séquences a une influence limitée sur la spécificité. Lorsque le nombre d'occurrences présentes dans les séquences est inférieur à la limite r demandée par l'utilisateur, Ed'Nimbus ne détecte aucune répétition. La spécificité est alors égale à 1.

Une fois que le nombre d'occurrences dans les séquences dépasse le nombre r demandé par l'utilisateur, Ed'Nimbus détecte toute les occurrences, quel que soit leur nombre, et ainsi, ne modifie pas la spécificité.

5.5.9 Tests en fonction de la longueur des k-facteurs

Nous présentons dans cette section des tests permettant de constater l'influence du paramètre k sur le comportement d'Ed'Nimbus.

Ce paramètre a une grande influence sur le comportement d'Ed'Nimbus, à la fois du point de vue de la spécificité que du point de vue du temps de calcul.

Influence de k sur la spécificité. Lorsqu'une très petite valeur est choisie pour k, le nombre minimal de k-facteurs p, partagés par les couples d'occurrences des répétitions recherchées, est grand (voir par exemple le tableau 5.3). Cependant, la probabilité pour que des petits k-facteurs apparaissent «par chance» est grande.

À l'inverse, une grande valeur de k diminue ce nombre p. Dans cette configuration, le nombre de k-facteurs partagés à détecter est limité mais la probabilité qu'un long k-facteur apparaisse par chance est petite.

Ainsi, il faut déterminer une valeur intermédiaire de k telle que la spécificité soit la meilleure. Parmi les tests que nous avons effectués, la meilleure valeur de k à appliquer pour maximiser la spécificité est $k = 4$ (voir la figure 5.15).

Influence de k sur le temps d'exécution. Dans les calculs de complexité effectués dans la section 5.4, nous avons constaté que le paramètre k, lorsqu'il augmente, diminue la complexité théorique. Cette diminution est due, d'une part, au fait qu'un petit k-facteur apparaît, en moyenne, plus souvent qu'un long k-facteur. D'autre part, la complexité théorique comprend un élément

en $\log p$. Or, p est un paramètre proportionnel à k. Nous pouvons constater la diminution de p en fonction de k dans le tableau 5.3.

De plus, en parallèle à la complexité théorique, du fait que les petits k-facteurs apparaissent plus souvent que les longs k-facteurs, l'utilisation de petits k-facteurs augmente le nombre de bons blocs détectés. Dans une telle situation, le nombre de calculs de $|LCS|$ à effectuer est lui aussi plus important lors de l'utilisation de petites valeurs de k.

D'un point de vue pratique, ceci se vérifie de manière très significative. Par exemple dans la figure 5.15, où l'on observe l'évolution du temps d'exécution en fonction de k, nous constatons que pour de très petites valeurs de k comme 2 ou 3, le temps d'exécution est beaucoup plus important que pour de plus grandes valeurs de k.

Notons que, pour ces tests, nous avons choisi de présenter le taux de conservation plutôt que la spécificité. Ce choix a été effectué car la spécificité dépend de la longueur des séquences, ce qui n'entre pas en compte pour ce résultat.

k	p
2	79
3	68
4	57
5	46
6	35
7	24
8	13
9	2
10	0

TAB. 5.3 – Valeurs de p en fonction de k, avec $L = 100$ et $d = 10$.

Lors du filtrage de très longues séquences, il peut être intéressant d'effectuer une première étape pour filtrer les séquences en utilisant une valeur de k assez grande, comme par exemple $k = 7$. Ceci permet d'obtenir une application rapide, mais aussi peu spécifique. Lors d'une seconde étape, un

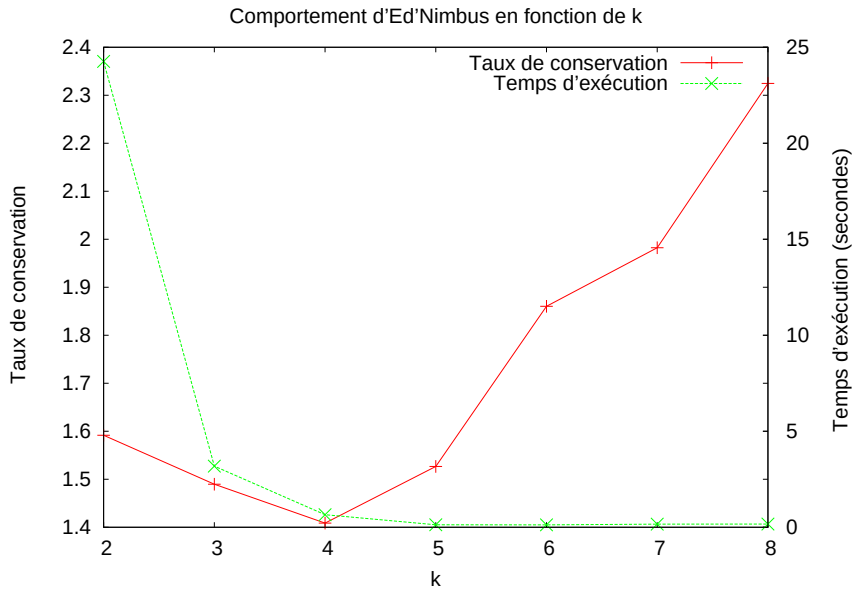


FIG. 5.15 – Comportement d'ED'NIMBUS en fonction de k. Sur l'alphabet $\{A, C, T, G\}$, trois séquences aléatoires de longueur 10000 ont été générées. Chacune de ces séquences contient une occurrence d'une répétition de longueur 100 dont les occurrences sont distantes deux à deux de 10 opérations d'édition. ED'NIMBUS a été utilisé pour filtrer ces séquences pour rechercher des répétitions ayant $r = 3$ occurrences de taille $L = 100$, distantes deux à deux d'au plus $d = 10$ opération d'édition. La valeur du paramètre k variant de $k = 2$ à $k = 10$. Chaque valeur est la moyenne de 50 tests.

filtrage utilisant une valeur de k plus petite, comme $k = 5$ par exemple, peut être appliqué sur les séquences résultantes. Ceci conduit à un filtrage plus spécifique et aussi plus lent, mais appliqué à des séquences préfiltrées de longueur limitée.

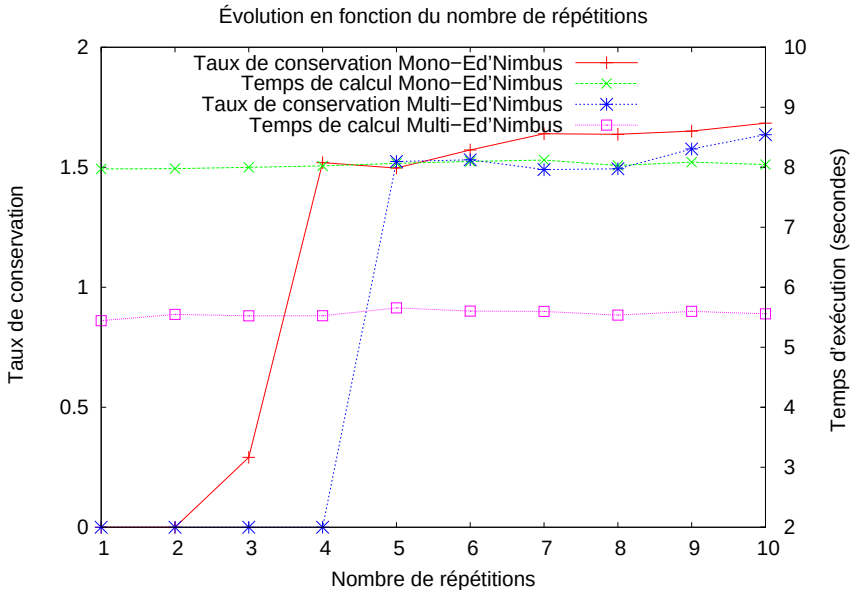


FIG. 5.16 – Comportement comparé de MONO-ED'NIMBUS et de MULTI-ED'NIMBUS en fonction du nombre d'occurrences des répétitions présentes dans les séquences.

Pour MONO-ED'NIMBUS **:** Sur l'alphabet $\{A, C, T, G\}$, nous avons généré une unique séquence de longueur 200000 Mb. Dans cette séquence, nous avons introduit une répétition possédant x occurrences de longueur 100. Chaque paire d'occurrences des répétitions présentes dans la séquence était distante de 10 opérations d'édition. Sur cette séquence, nous avons utilisé MONO-ED'NIMBUS avec les paramètres $L = 100$, $r = 5$, $d = 10$ et $k = 5$.

Pour MULTI-ED'NIMBUS **:** Sur l'alphabet $\{A, C, T, G\}$, nous avons généré 5 séquences de longueur totale 200000 Mb. Dans ces séquences, nous avons introduit une répétition possédant x occurrences de longueur 100. Chaque paire d'occurrences présentes dans les séquences était distante de 10 opérations d'édition. Nous avons réparti les occurrences de cette répétition de manière pseudo-aléatoire (en imposant au moins un occurrence par séquence quand ceci est possible) dans ces séquences. Sur cet ensemble de séquences, nous avons utilisé MULTI-ED'NIMBUS avec les paramètres $L = 100$, $r = 5$, $d = 10$ et $k = 5$.

Nous avons fait varier x de 0 à 10. Chaque valeur est le résultat moyen de 50 tests.

5.5.10 Différence entre Multi-Ed'Nimbus et Mono-Ed'Nimbus

Nous présentons dans cette section les différences concernant le temps de calcul et de qualité de filtrage entre les deux versions d'Ed'Nimbus : Multi-Ed'Nimbus et Mono-Ed'Nimbus

Différence concernant la qualité de filtrage

La version Mono-Ed'Nimbus d'Ed'Nimbus est handicapée par un défaut pour l'heure non résolu.

En effet, comme nous pouvons le constater dans la figure 5.17, une occurrence d'une répétition présente dans les séquences a de grandes chances de se retrouver en grande partie sur deux blocs. Dans de telles conditions, pour une occurrence d'une répétition présente dans les séquences, deux bons blocs seront considérés, pouvant mener à deux blocs parfaits. L'algorithme considère alors à tort que deux occurrences existent.

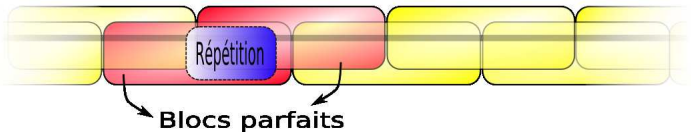


Fig. 5.17 – Représentation d'une occurrence d'une répétition conduisant à la détection de deux blocs parfaits. L'occurrence de la répétition, représentée en bleu est entièrement comprise dans le premier bloc représenté en rouge. Cependant, elle est aussi quasiment entièrement incluse dans le second bloc représenté en rouge. À la suite du calcul de $|LCS|$, ces deux blocs peuvent être considérés comme des blocs parfaits, biaisant ainsi le résultat.

Ce handicap conduit Mono-Ed'Nimbus à avoir une spécificité moins bonne que Multi-Ed'Nimbus lorsque des répétitions existent dans les séquences mais que le nombre de leurs occurrences est inférieur à la valeur demandée par l'utilisateur.

Pour évaluer ce handicap, nous avons effectué des tests dont les résultats sont présentés dans la figure 5.16.

Nous avons indiqué sur cette figure la différence en ce qui concerne le taux de conservation plutôt que la spécificité. Le taux de conservation, dans ce cas, met plus en relief le contraste en les deux versions d'ED'NIMBUS.

Nous constatons dans cette figure que lorsque les répétitions présentes dans les séquences possèdent un nombre d'occurrences légèrement inférieur au paramètre r demandé par l'utilisateur, MONO-ED'NIMBUS détecte ces répétitions créant ainsi des faux-positifs, alors que MULTI-ED'NIMBUS, ne les détecte pas. Dans les tests proposés dans la figure 5.16, ceci se vérifie lorsque le nombre de répétitions présentes dans les séquences est égal à 3 et 4.

Différence concernant le temps de calculs

Comme nous pouvons le constater dans la figure 5.16, MULTI-ED'NIMBUS est légèrement plus rapide que MONO-ED'NIMBUS sur des données similaires et avec les mêmes paramètres. Ceci s'explique par le fait que, pour MULTI-ED'NIMBUS, durant la recherche de bons blocs, la séquence sur laquelle se trouve la fenêtre de référence n'est pas prise en considération. Ainsi, moins de compteurs sont mis à jours dans le cas de MULTI-ED'NIMBUS, et surtout, moins de bons blocs sont détectés et testés par un calcul de $|LCS|$.

* *

*

Nous présentons maintenant quelques résultats préliminaires concernant l'application possible d'ED'NIMBUS à l'accélération de calculs effectués pour résoudre des problèmes biologiques.

5.6 Applications

Dans cette section, nous proposons des tests afin de percevoir les possibilités d'ED'NIMBUS à accélérer des méthodes d'alignements multiples locaux.

Nous commençons par une application sur des données artificielles avant d'utiliser ED'NIMBUS sur des données provenant de séquences d'ADN réelles.

5.6.1 Séquences générées

Nous rappelons que l'idée est de filtrer les séquences avant de les appliquer à un algorithme d'alignement multiple local.

Sur l'alphabet $\{A, C, T, G\}$, nous avons créé un ensemble de cinq séquences aléatoires de longueur totale 1 Mb. Chaque séquence contenait exactement une occurrence d'une répétition de longueur 100. Chaque paire d'occurrences était distante d'au plus 10 opérations d'édition.

Nous avons appliqué GLAM (un outil d'alignement multiple local, présenté dans [FHSW04]) sur ces séquences. Cet algorithme a détecté les répétitions artificiellement insérées en plus de dix heures de calcul.

Nous avons alors filtré ces séquences avec MONO-ED'NIMBUS. Cette opération a été effectuée en environ 10 secondes. GLAM a alors de nouveau été utilisé mais sur les séquences filtrées. Cette nouvelle étape a été effectuée en environ 15 secondes, détectant exactement le même résultat. Dans ce cas, l'utilisation d'ED'NIMBUS a ainsi permis de détecter un résultat en 25 secondes au lieu de plus de dix heures.

5.6.2 Séquences réelles

Nous avons appliqué le même processus que précédemment à de vraies séquences biologiques. Pour ce faire, nous avons utilisé des séquences du gène $GATA3$ qui est un gène impliqué dans de nombreux processus du développement [OLG+05]. Certains de ces processus sont identifiés et différents selon les espèces. Il est donc intéressant d'accomplir une analyse comparée de ce gène de différentes espèces par un alignement multiple local. Nous avons ainsi effectué un tel alignement multiple sur le locus du gène $GATA3$ de huit espèces : l'homme, la souris, le rat, le poulet, la grenouille, le poisson *fugu*, le poisson *zebrafish* et le poisson *tetraodon*. La longueur complète des séquences ainsi alignées était de 65000 nucléotides.

Nous avons appliqué GLAM sur ces séquences. Le temps de calcul a été de 10 minutes et 45 secondes pour détecter des répétitions de longueur 100. Nous avons ensuite appliqué MULTI-ED'NIMBUS avec les paramètres $L = 100$, $d = 8$, $r = 3$ et $k = 6$ sur ces mêmes séquences. Cette étape a été effectuée en $4.e^{-8}$ seconde (autrement dit en temps négligeable). Les séquences filtrées ont alors été données en entrée à GLAM qui a retrouvé le même résultat que précédemment en 24 secondes au lieu de plus de 10 minutes. Ainsi, l'alignement multiple local a été effectué environ 27 fois plus rapidement

grâce à l'utilisation d'ED'NIMBUS.

5.7 Comparaison de Nimbus d'ED'NIMBUS

Bien que NIMBUS et ED'NIMBUS ne soient pas directement destinés à la même utilisation, il peut être intéressant de comparer ces deux outils. En effet, ces deux algorithmes ont des applications très proches. Il est alors possible d'en comparer la qualité de filtrage ainsi que les temps de calcul.

Tests effectués. Nous avons repris les tests effectués dans la section 5.5.3 (présentés dans la figure 5.9) où nous avions appliqué à la fois ED'NIMBUS et l'algorithme que nous avions appelé strict sur des séquences contenant des répétitions de plus en plus bruitées. Ici nous avons effectué ces mêmes tests en utilisant NIMBUS et ED'NIMBUS (en limitant les différences entre les répétitions présentes dans les séquences à des substitutions afin de pouvoir utiliser NIMBUS). Les résultats de ces tests sont présentés dans la figure 5.18.

Cette figure nous permet de constater que l'algorithme ED'NIMBUS présente une qualité de filtrage très nettement meilleure que celle de NIMBUS. En effet, ED'NIMBUS filtre avec une meilleure spécificité que NIMBUS les séquences contenant des répétitions bruitées.

D'autre part, le temps moyen nécessaire pour l'exécution de ces tests est d'environ 10 secondes pour NIMBUS contre 0.4 seconde pour ED'NIMBUS.

En plus d'avoir une qualité de filtrage nettement supérieure à NIMBUS, ED'NIMBUS présente de bien meilleurs résultats en ce qui concerne le temps d'exécution. Du point de vue de la quantité de mémoire utilisée, celle-ci est sensiblement identique pour NIMBUS et ED'NIMBUS.

Ainsi, en plus d'être un outil applicable à la réalité biologique grâce au fait qu'il accepte mes insertions et les délétions entre les répétitions recherchées, ED'NIMBUS présente des caractéristiques nettement meilleures que NIMBUS.

Commentaires sur la différence concernant de qualité de filtrage. La différence de qualité de filtrage (mesurée par la différence en termes de taux de conservation) s'explique par l'observation suivante. La condition de filtrage $\mathcal{C}_4$ utilisée par NIMBUS s'applique à l'ensemble des r occurrences des (L, r, d)-répétitions recherchées. À l'inverse, la condition de filtrage $\mathcal{C}_5$ utilisée par ED'NIMBUS s'applique non pas à l'ensemble des r occurrences des répé-

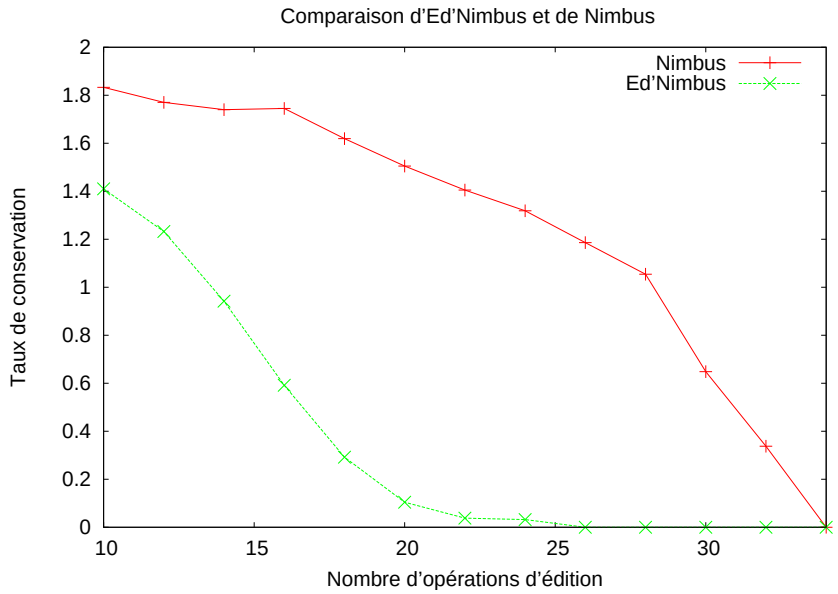


FIG. 5.18 – Taux de conservation comparé de NIMBUS et d'ED'NIMBUS sur des séquences bruitées. Nous reprenons les tests effectués dans la section 5.5.3 (présentés dans la figure 5.9) à la différence près que les répétitions présentes dans les séquences sont distantes par des substitutions uniquement.

titions mais à un sous ensemble de 2 occurrences de répétitions uniquement, tout en imposant que cette condition soit simultanément respectée $r-1$ fois.

Ainsi, dans le cas d'ED'NIMBUS, l'augmentation du nombre r d'occurrences de répétitions recherchées ne modifie par le nombre minimum de k-facteurs partagés par chaque couple d'occurrences de (L, r, d)-répétition. À l'inverse, dans le cas de NIMBUS, la condition nécessaire concernant le nombre minimum de k-facteurs partagés par toutes les occurrences d'une (L, r, d)-répétition décroît (rapidement) avec le nombre d'occurrences de chaque répétition recherchée. Ceci conduit à une qualité de filtrage bien moins bonne dans le cas de NIMBUS que dans le cas d'ED'NIMBUS.

Commentaires sur la différence de temps d'exécution. L'importante différence de temps d'exécution reflète les différentes techniques employées pour la vérification des conditions nécessaires par NIMBUS et ED'NIMBUS. Dans le cas de NIMBUS, la méthode utilisée pour détecter des ensembles k-facteurs séparés par une distance égale entre les différents ensembles conduit à une complexité plus importante. Nous rappelons que celle-ci contient un facteur exponentiel.

Dans le cas d'ED'NIMBUS, la condition nécessaire est appliquée de manière moins stricte. Ceci conduit à un algorithme plus rapide qu'un algorithme appliquant strictement la condition nécessaire, tout en perdant très peu de spécificité. Ceci explique la très importante différence de temps d'exécution de NIMBUS et ED'NIMBUS.

5.8 Interface ED'NIMBUS

Le filtre ED'NIMBUS possède une interface web dont une copie d'écran est donnée dans la figure 5.19. Cette interface se trouve à l'adresse suivante : `http://igm.univ-mlv.fr/~peterlon/ednimbus`.

Cette interface permet de soumettre des travaux à MONO-ED'NIMBUS ou MULTI-ED'NIMBUS. L'utilisateur peut choisir d'obtenir l'ensemble des résultats dans un unique fichier. Dans ce cas, ce fichier contient des informations quant aux paramètres utilisés, les positions correspondant aux portions de séquences conservées ainsi que les séquences conservées au format FASTA. Si l'utilisateur choisit d'obtenir les résultats sous forme de plusieurs fichiers, les informations concernant les paramètres et les positions conservées sont placées dans un fichier (d'extension «.out») et les séquences filtrées sont placées au format FASTA dans un autre fichier (d'extension «.fa»).

De plus, l'utilisateur peut demander à ce que les portions de séquences obtenues à la suite du filtrage soient séparées par autant de caractères «N» que de nucléotides supprimés par le filtre. Ceci permet de conserver l'information concernant les intervalles conservés, tout en remplaçant les portions non pertinentes par des «N». En conséquence, les outils d'alignements multiples locaux ne s'appliquent qu'aux portions pouvant contenir des répétitions tout en conservant leur positions relatives.

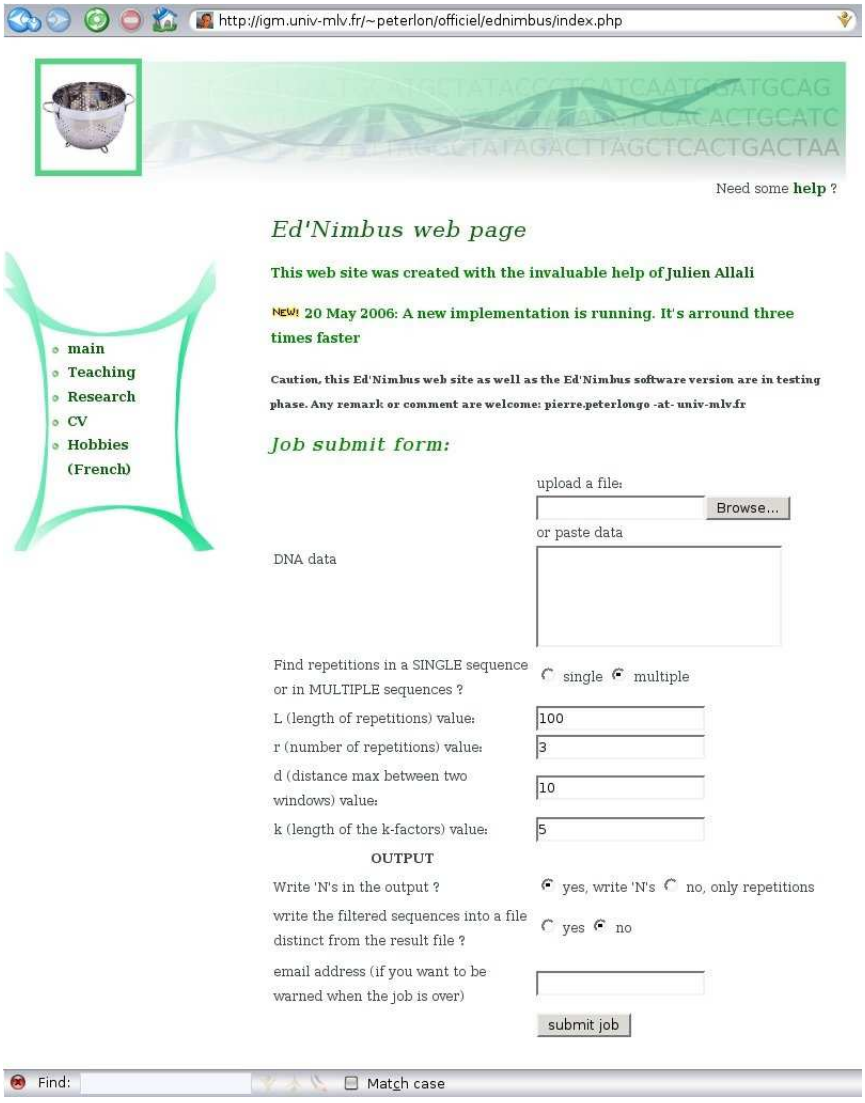


FIG. 5.19 – Interface web d'ED'NIMBUS. Cette page internet peut être trouvée à partir de la page `http://igm.univ-mlv.fr/~peterlon/`.

5.9 Conclusion

Nous avons présenté dans ce chapitre le programme ED'NIMBUS. ED'NIMBUS est un programme utilisant une nouvelle technique de filtrage prévue pour supprimer parmi des données représentant des séquences d'ADN, de larges portions ne pouvant faire parti de répétitions. Les répétitions considérées ici sont des répétitions ayant au moins deux occurrences. Chaque paire d'occurrences possède une distance contrainte par un nombre maximal d'opérations d'édition (insertions, délétions ou substitutions). Les répétitions filtrées peuvent faire parti d'une unique séquence (MONO-ED'NIMBUS) ou d'un ensemble de séquences (MULTI-ED'NIMBUS).

Le filtre se base sur une condition de filtrage utilisant le nombre minimum de mots de longueur fixée partagés par tout couple d'occurrences d'une répétition. L'ordre de ces mots est aussi pris en compte. Les portions de séquence ne respectant pas cette condition sont supprimées par le filtre.

Nous avons constaté dans la section 5.5 la bonne spécificité de ce filtre. Le temps d'exécution du programme est en outre assez faible. Par exemple, il est de l'ordre de la minute pour des séquences dont la longueur dépasse 2 millions de nucléotides (tableau. 5.1).

De plus, nous avons vu dans la section 5.6 que l'utilisation d'ED'NIMBUS permet d'accélérer des programmes d'alignements multiples locaux de plusieurs ordres de grandeur, aussi bien sur des séquences artificielles que sur des séquences réelles.

Nous pouvons enfin rappeler qu'ED'NIMBUS, disposant d'une très bonne spécificité, peut être utilisé seul comme une méthode heuristique de détection de répétitions. Dans ce cas, les résultat détectés contiennent effectivement les répétitions, mais aussi des portions de séquences ne contenant pas de répétition.

L'application ED'NIMBUS est implantée en C. Elle possède une interface libre sur internet permettant aux biologistes d'utiliser cette méthode à distance. Nous rappelons que l'interface est accessible à partir de l'adresse suivante : `http://igm.univ-mlv.fr/~peterlon/`.

Travaux futurs.

- ED'NIMBUS permet de filtrer des séquences pour y détecter des répétitions dont la longueur des occurrences est fixée à l'avance par l'utilisateur. D'un point de vue biologique, le fait de devoir fixer la longueur des séquences recherchées est problématique. L'un des axes de recherche

sur lequel nous allons travailler consiste à étudier des solutions permettant de supprimer cette contrainte. Pour ce faire, deux possibilités sont aujourd'hui envisagées. La première consiste à s'inspirer des idées présentées dans [RSM05] où une condition de filtrage pour trouver des similarités de longueur non bornée est proposée.
La seconde possibilité consiste à formaliser le fait que, avec la version courante d'Ed'Nimbus, la concaténation des portions non filtrées conduit à une méthode conservant, en pratique, des répétitions de longueur supérieure au paramètre L. Supposons que nous filtrons des répétitions dont la longueur des occurrences est L, distantes par paires de d opérations d'édition. Dans ce cas, le filtre conserve aussi des répétitions ayant des occurrences de longueur $x \times L$, distantes par paires de $x \times d$ opérations d'édition. En effet, une répétition dont les occurrences de longueur par exemple $2L$ sont distantes par paires de $2d$ opérations d'édition peut aussi être (en fonction de la répartitions des opérations d'édition) deux répétitions accolées dont les occurrences sont de longueur L et sont distantes par paires de d opérations d'édition. Ainsi, dans cet exemple, un filtre pour des répétitions de taille L distantes de d opérations d'éditions détectera aussi des répétitions de taille $2L$ distantes par $2d$ opérations d'éditions. Cependant, un tel filtre ne détectera par toutes ces répétitions.
Un filtre ainsi créé serait un filtre non exact. En fonction de la distribution des opérations d'édition, il est possible de «rater» certaines répétitions. Cet effet devra être formalisé et analysé afin d'estimer précisément la spécificité et la sensibilité d'un tel filtre.

– Afin d'améliorer encore la spécificité tout en conservant les qualités d'Ed'Nimbus concernant le temps d'exécution, nous allons étendre les recherches en supprimant l'utilisation de k-facteurs au sens premier comme base du filtrage. Nous allons chercher à utiliser dans la condition de filtrage des k-facteurs éclatés (comme décrit dans la section 3.4, page 99) ou des k-facteurs dits «*jumeaux*». Par k-facteur jumeaux, nous entendons deux k-facteurs identiques à l'exception d'un petit nombre de positions (non fixées à l'avance) où des substitutions sont autorisées.

Conclusion et perspectives

Dans cette thèse, nous nous sommes intéressés à un problème lié aux répétitions dans les génomes. Nous avons exposé les enjeux qui conduisent les généticiens à étudier ces répétitions. Nous nous sommes focalisés sur un type de répétitions représentant un intérêt particulier d'un point de vue biologique et pour lesquelles il n'existait pas de moyen de détection assez rapide pour être appliqué sur des données aussi grandes que des génomes complets. Les répétitions qui ont retenu notre attention sont les **longues répétitions multiples** (d'une centaine de nucléotides par exemple), possédant au moins deux occurrences.

Afin de permettre la détection de longues répétitions multiples en un temps raisonnable sur de grandes données, nous avons créé des **filtres**. Ces filtres permettent de supprimer rapidement des données de larges portions pour lesquelles nous avons détecté qu'aucune occurrence d'une répétition ayant des caractéristiques indiquées en entrée par l'utilisateur ne peut y être présente. Les données filtrées, de longueur largement inférieure aux données initiales, sont alors utilisées comme données d'entrée pour des algorithmes de détection de répétition.

Les travaux effectués de cette thèse ont conduit à la création de deux algorithmes de filtrage, Nimbus et Ed'Nimbus. Ces filtres sont utilisés pour détecter des répétitions dont les occurrences sont limitées deux à deux par une distance maximale. Dans le cas de Nimbus, cette distance est la distance de Hamming (nombre de substitutions). Dans le cas d'Ed'Nimbus, il s'agit d'une distance d'édition (nombre de substitutions, d'insertion et de suppressions). Ce second filtre, plus adapté à une application biologique, est disponible sur une interface internet : `http://igm.univ-mlv.fr/~peterlon/ednimbus/`.

Les résultats obtenus permettent, sur des séquences artificielles ou sur des séquences naturelles, d'accélérer de plusieurs ordres de grandeur le temps

de détection de répétitions multiples. Par exemple, sur des séquences du gène GATA3, nous avons recherché les répétitions entre 7 espèces à l'aide de GLAM [FHSW04], un algorithme d'alignement multiple local. L'utilisation d'ED'NIMBUS a permis de réduire de 27 fois le temps de calcul pour parvenir au même résultat.

Forts de ces premiers résultats prometteurs, nous allons poursuivre les travaux, d'une part en appliquant les idées déjà utilisées à d'autres problématiques de la génétique et, d'autre part, en essayant d'améliorer les filtres existants.

Perspectives

Application à la recherche de points de cassure

L'une des applications possibles des travaux réalisés consiste en la détection de points de cassures*.

Les points de cassures. Pour le génome d'une espèce, un point de cassure désigne une position où un réarrangement a eu lieu. La figure 5.20 montre un exemple de point de cassure. Des études menées en 1984 par Nadeau et Taylor [NT84] concluent que les points de cassure sont aléatoirement répartis dans les génomes. Cette théorie a depuis été mise en doute par des travaux récents [PT02, BPT04, PPT06]. Ces travaux émettent l'hypothèse que les cassures n'interviennent pas aléatoirement dans le génome mais plutôt à certaines positions préférentielles, appelées des «points chauds» (*hotspots*). Cette notion de zones privilégiées pour les points de cassure pourrait impliquer que ces points chauds possèdent certaines caractéristiques propres qu'il serait intéressant de déterminer. L'étude de ces points chauds, qui est un travail en cours, en collaboration avec Claire Lemaitre dans le cadre de son doctorat, implique d'être capable de détecter les points de cassures sur les génomes.

Comment détecter les points de cassure ? Nous proposons d'appliquer les concepts utilisés par NIMBUS et ED'NIMBUS pour détecter le point de cassure, par exemple sur une séquence R (*cf.* figure 5.20). Nous nous basons sur l'idée suivante. La séquence située à gauche du point de cassure sur la séquence R est similaire à la séquence A. Inversement, la séquence située à

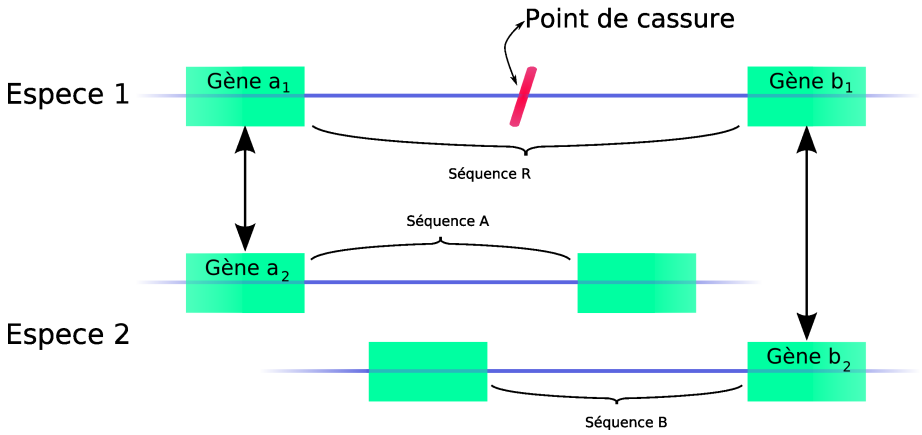


FIG. 5.20 – Exemple de point de cassure. Dans l'espèce 1, deux gènes a_1 et b_1 sont consécutifs sur le même chromosome. Dans l'espèce 2, les gènes orthologues a_2 et b_2 de ces deux gènes ne sont plus consécutifs, plus dans le même ordre ou les deux. Les gènes on donc été réarrangés, on dit qu'il y a eu une cassure entre les gènes a_1 et b_1 de l'espèce 1. La localisation où s'est produite cette rupture est appelée point de cassure.

droite du point de cassure sur la séquence R est similaire à la séquence B. Ainsi pour détecter le point de cassure sur la séquence R il faut déterminer la position p maximisant une fonction de score.

Nous proposons, pour calculer une fonction de score, d'appliquer une méthode basée sur le nombre de k facteurs partagés par la séquence à gauche (respectivement à droite) de i sur R et la séquence X. Une telle approche permet un calcul rapide par l'application des idées basées sur la fenêtre glissante (Section 5.2.2, page 150) et utilisant le tableau des k-facteurs comme structure d'indexation (section 5.2.1, page 146).

Développement d'Ed'Nimbus

Comme nous l'avons exprimé dans la conclusion du chapitre 5, le filtre Ed'Nimbus pourrait être amélioré selon plusieurs axes.

Éviter la contrainte sur la longueur fixée des répétitions. L'une des priorités est de permettre à Ed'Nimbus le filtrage de répétitions dont la longueur des occurrences n'est pas bornée. Nous avons évoqué deux possibilités pour effectuer cette modification. L'une ne modifierait pas le filtre actuel, elle formaliserait le fait que Ed'Nimbus, dans sa version courante permet de détecter des répétitions dont les occurrences sont de longueur supérieure à la limite proposée par l'utilisateur. La seconde modifierait la condition de filtrage selon les idées proposées dans [RSM05].

Améliorer la spécificité d'Ed'Nimbus**.** Les travaux futurs se dirigeront vers l'amélioration de la spécificité d'Ed'Nimbus. Nous avons envisagé deux possibilités pour effectuer cela. Nous pourrions utiliser des k-facteurs éclatés pouvant conduire à un filtrage exact avec une bonne spécificité. Nous avons d'autre part émis l'idée d'utiliser des k-facteurs jumeaux, identiques à quelques substitutions près.

Intégrer des informations structurales aux répétitions. Parmi les répétitions recherchées, certaines contiennent des structures internes. Les transposons en particulier sont présents dans les génomes sous forme de longues répétitions bordées de répétitions (directes ou non) plus courtes. Le travail associé à ces modifications nécessite une analyse en profondeur de la réalité biologique afin de formaliser le problème sous forme de modèles applicables d'un point de vue algorithmique.

Transformer Ed'Nimbus **en un algorithme d'alignement multiple local.** Dans sa version actuelle, Ed'Nimbus propose en sortie une concaténation de portion de séquences conservées. Ainsi, Ed'Nimbus ne permet pas d'identifier les répétitions elles-mêmes et contient de plus des faux-positifs.

Ainsi, même si Ed'Nimbus peut être employé dans le cadre de l'accélération de calculs d'alignements multiples locaux, son utilisation seule est limitée. Nous aimerions alors étendre Ed'Nimbus vers un algorithme complet permettant l'obtention d'alignements multiples locaux, ou bien, permettant d'obtenir les répétitions, et uniquement les répétitions, avec les liens entre les occurrences des répétitions trouvées. Ceci peut être envisagé selon deux axes, soit au travers d'une application exacte, soit par une heuristique. Dans ce dernier cas, les séquences initialement fournies par Ed'Nimbus peuvent être

utilisées comme graine pour des algorithmes tels que MUMMER [DKF+99], LAGAN [BDC+03], MGA [HKO02], CHAOS [BCG+03]....

Annexe A

L'arbre des bi-facteurs

Nous présentons dans cette section l'arbre des bi-facteurs (appelé «*gapped-factor tree*» en anglais) une structure de données qui a été étudiée en vue d'être utilisée dans l'application NIMBUS présentée dans le chapitre 4. Cette structure de donnée permet d'indexer les (k, g, k')-bi-facteurs d'un texte ou d'un ensemble de textes. Nous rappelons que la structure de bi-facteur est donnée dans la définition 20 page 114.

L'apparition d'algorithmes permettant la création de tableaux des suffixes en temps linéaire sans utilisation intermédiaire d'arbres des suffixes [KS03, KSPP03, KA03] a contribué à l'abandon de l'utilisation de cette structure de données pour l'algorithme NIMBUS. Il nous a cependant paru important de présenter dans ce manuscrit de thèse les travaux effectués sur cette structure de données. En effet, les solutions algorithmique envisagées pour indexer ce type particulier de données que sont les bi-facteurs sont assez intéressantes et peuvent éventuellement être utilisées pour résoudre d'autres problèmes.

Ce travail étend des résultat proposés par Allali dans [AS04] concernant l'arbre des k-facteurs qui est un arbre des suffixes tronqué à la hauteur k.

Nous incluons donc dans les pages qui suivent un article écrit en collaboration avec Julien Allali et Marie-France Sagot, actuellement en cours de soumission.

The gapped-factor tree

résumé

Nous présentons unes structure de donnée qui indexe des facteurs particuliers appelés les bi-facteurs. *Un bi-facteur contient un trou qui n'est pas pris en considération lors de l'indexation. La structure de données présentée est basée sur l'arbre des suffixes et indexe tous les bi-facteurs d'un texte pour une structure de bi-facteur donnée et uniquement ceux-là. La construction de cette structure de données est faite à la volée en temps et espace en $O(n \times |\Sigma|)$ avec n la taille des données à indexer et $|\Sigma|$ la taille de l'alphabet. Cette structure de données peut jouer un rôle important pour certains problèmes de pattern matching et d'inférence, comme c'est le cas par exemple en filtrage de textes.*
mots clefs : *arbre des suffixes, arbre des k-facteurs, indexation de textes, bi-facteur, arbre des bi-facteurs*

abstract

We present a data structure to index a specific kind of factors, that is of substrings, called gapped-factors. *A gapped-factor is a factor containing a gap that is ignored during the indexation. The data structure presented is based on the suffix tree and indexes all the gapped-factors of a text with a fixed size of gap, and only those. The construction of this data structure is done online in $O(n \times |\Sigma|)$ time and space, with n the length of the text and $|\Sigma|$ the size of the alphabet. Such a data structure may play an important role in some pattern matching and motif inference problems, for instance in text filtration.*
keywords : *suffix tree, k-factor factor tree, string index, gapped-factor, gapped-factor tree*

Introduction

The indexation and extraction of repeated short words (called *k-factors*[1] for words of length k) has become a widely used technique in many text algo-

[1] Another currently used term for designing k-factors is q-grams

rithmic problems. One can mention their use in, for instance, FASTA [LP85] and BLAST [AGM+90,AMS+97]. Indeed, many algorithms for efficiently computing string matches [ST95,NSTT00,GIJ+01] or alignments [MTL02,HKO02, LM03, BDC+03, BCG+03, Edg04, MDV05] use k-factors. In particular, filtration algorithms that have been created for quickly discarding large portions of the input before applying a more expensive algorithm on the remaining data are often based on the identification of such short repeated words [PW95, BCF+99, BK01, BK02, RSM05, PPBS05, KNR05b].

Among the exact filtration algorithms (exact in the sense that they discard only portions of the text that can not be part of the final solution sought), some consider k-factors composed of non consecutive letters [PW95, BK01,BK02,KNR05b], or sets of k-factors [BCF+99,RSM05,PPBS05]. Both present advantages for filtering purposes in comparison with single k-factors with no letters skipped as shown in [BK01,Kär02,KNR05b].

In order to efficiently use such k-factors, one needs data structures to index them. Depending on the kind of k-factor adopted, different types of data structures may be considered. For instance, sets of k-factors may be indexed in a hash table or using a labelling technique as proposed in [IMP+05]. In this paper, we introduce a data structure designed for the indexation of sub-words composed of a k-factor, a gap of length d not taken into account during the indexation and a k'-factor. Such a sub-word is called a *gapped-factor* as it contains a unique gap.

The new data structure is an adaptation of the suffix tree [McC76]. More precisely, the construction we describe in this paper is an adaptation of the construction of a k-factor tree [AS04], which itself is an extension of the Ukkonen construction of a suffix tree [Ukk95]. A k-factor tree is a tree indexing all k-factors of a text.

As indicated in Section A.4, the new data structure, called a *gapped-factor tree*, allows to extract in linear time all the repeated gapped-factors of a text or of a set of texts. Furthermore, it offers the possibility to obtain in time $O(k + k')$ the list of all the positions of a gapped-factor.

The paper is organised as follows. In Section A.1, we provide the context and some definitions about text and trees. In Section A.2, we formally introduce gapped-factors and the gapped-factor tree. In Section A.3, we present the algorithm to construct a gapped-factor tree for indexing the gapped-factors of a text after recalling the Ukkonen construction of a suffix tree and the Allali construction of a k-factor tree. We end by indicating two basic uses of gapped-factor trees.

A.1 Preliminaries

A *text*, also called a *string*, is a sequence of zero or more symbols from an alphabet Σ. A text t of length n is denoted by $t[0, n-1] = t_0 t_1 \dots t_{n-1}$, where $t_i \in \Sigma$ for $0 \leq i < n$. The length of t is denoted by $|t|$. A string w is a *factor* of t if $t = uwv$ for $u, v \in \Sigma^*$; in this case, the string w occurs at position $|u|$ in the string t. A k-factor denotes a factor of length k. If $t = uv$ for $u,\ v\ \in \Sigma^*$ then v is called a *suffix* of t. A suffix starting at position i in t is denoted by $t_{i\dots}$.

A tree is a data structure composed of **nodes** connected together by **edges**. Except for a special node called the **root**, each node has exactly one **father**. Nodes with no children are called the **leaves** while all other nodes are called the **internal nodes** of the tree. An internal node having at least two children is called a **branching node**.

We call the **depth** of a node $\mathcal{N}$ the sum of the lengths of the edges that need to be traversed from the root of the tree to reach $\mathcal{N}$. By definition, the depth of the root is thus 0.

Nodes and edges may be labelled. For instance, in Figure A.1, edges are labelled with letters from a given alphabet.

Let $\mathcal{N}$ be a node of a tree, we denote by $path(\mathcal{N})$ the text corresponding to the concatenation of the letters from a given alphabet labelling the edges from the root to $\mathcal{N}$.

For instance, if $\mathcal{N}_0$ denotes the leftmost leaf of the tree presented in Figure A.1, $path(\mathcal{N}_0) = AA$.

The suffix trie of a text t is a tree with edges labelled with elements of Σ. For each factor of t, there exists a node N such that $path(\mathcal{N})$ is equal to that factor. If t has an ending symbol, all nodes $\mathcal{N}$ for which the path from the root spells a suffix of t are leaves.

The *implicit* suffix tree of t is a tree with edges labelled by non-empty elements of Σ^*. The suffix tree is a compressed version of the suffix trie. Each internal node $\mathcal{N}$ of the suffix trie that has only one child is deleted and its two adjacent edges are replaced by an edge that goes from the father of $\mathcal{N}$ to its child. The label of the new edge is equal to the concatenation of the label of the edge going from the father of $\mathcal{N}$ to $\mathcal{N}$ and of the label of the edge from $\mathcal{N}$ to its child. This tree is called implicit because not all suffixes of t lead to a leaf. The true suffix tree is obtained when a special ending symbol \$ not in Σ is added at the end of t. A suffix tree indexes all the $|t|$ suffixes of a text t.

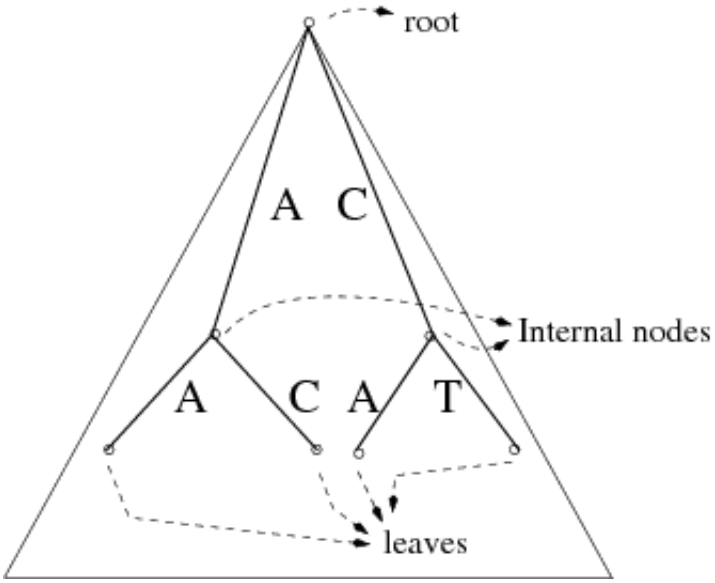


FIG. A.1 – Example of a tree labelled with letters from a given alphabet. Reading all paths from the root to the leaves, leads to the strings AA, AC, CA and CT

A.2 Gapped-factor tree

A gapped-factor tree indexes gapped-factors that are defined as follows :

Définition 21 (Gapped-factor)**.** *A* **gapped-factor** *is a concatenation of a factor of length k, a gap of length d and another factor of length k'. A gapped-factor occurring at position i in a text t is $t[i, i+k-1].t[i+k+d, i+k+d+k'-1]$. Such a gapped-factor is called a (k-d-k')-gapped-factor.*

An example of a (2-1-3)-gapped-factor is given in Figure A.2.

We propose to use a new data structure, called a *gapped-factor tree*, to index all the (k-d-k')-gapped-factors of a text or of a set of texts. This is a modification of the suffix tree [McC76] data structure. The gapped-factor tree takes into account the gap of length d of the gapped-factors it indexes. This mean that the tree contains a region up to which the k-factors are indexed as in a classical suffix tree, while below this region the second factors (of length k') of the (k-d-k')-gapped-factors starting with the same k-factor start from the same node. This regions is called the *invisible region*.

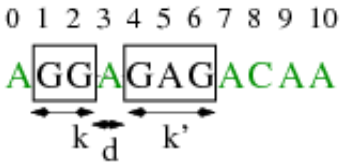


FIG. A.2 – Example of a (2-1-3)-gapped-factor. The first factor length is $k = 2$, the gap is of length $d = 1$ and the second factor has a length $k' = 3$. It occurs at position 1 in the text. With these parameters, the content of the gapped-factor occurring at position 1 is *GGGAG* composed by *GG* and *GAG*.

An intuitive idea of such a data structure is given in Figure A.3.

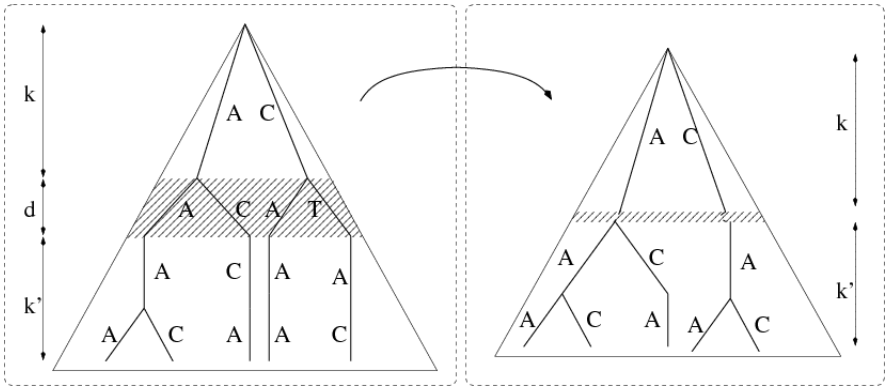


FIG. A.3 – An intuitive view of a gapped-factor tree. Even if this is not the way the gapped-factor tree is constructed, a gapped-factor tree can be seen as a truncated suffix tree where a part has been removed, provoking merges in the lower part of the tree.

Définition 22 (Path in a Gapped-Factor Tree)**.** *Let w be a (k-d-k')-gapped-factor starting at position $i < |t| - k - d - k'$ that is indexed in such a tree. Let $\mathcal{N}$ be the node at depth $z \leq k + k'$ corresponding to this (k-d-k')-gapped-factor. Then :*

$$path(\mathcal{N}) = \begin{cases} t[i, i+z-1] & \text{if } z \leq k \\ t[i, i+k-1].t[i+k+d, i+d+z-1] & \text{otherwise} \end{cases}$$

An example of gapped-factor tree and of a path in such a tree is presented in Figure A.4.

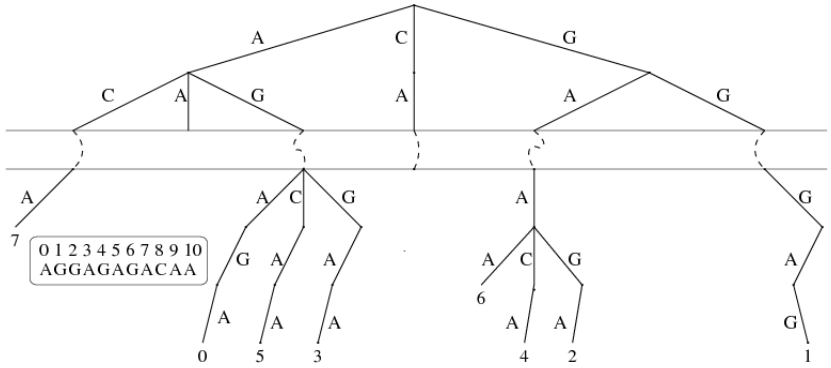


FIG. A.4 – Example of gapped-factor tree. The input sequence is *AGGAGAGACAA*. The dashed lines correspond to the invisible region of the tree. In this case, the gapped factors indexed are (2-1-3)-gapped-factors. The information attached to one of the leaves corresponds to the starting positions of a gapped-factor in the text.

In the next section, we present the algorithm which performs the online construction of a gapped-suffix tree.

A.3 Construction

The algorithm for constructing a gapped-factor tree is an extension of the algorithm for constructing a k-factor tree [AS04], which is itself an extension of the suffix tree construction algorithm due to Ukkonen [Ukk95]. Therefore, in the following, we start by presenting the construction of a suffix tree, then the one of a k-factor tree, and finally we describe the construction of a gapped-factor tree.

A.3.1 Ukkonen construction of the suffix tree

To present the Ukkonen algorithm, we follow the description given in [Gus97]. This algorithm constructs a full suffix tree of a text t in $O(|t|)$ time and space. An example of a suffix tree is given in the Figure A.5.

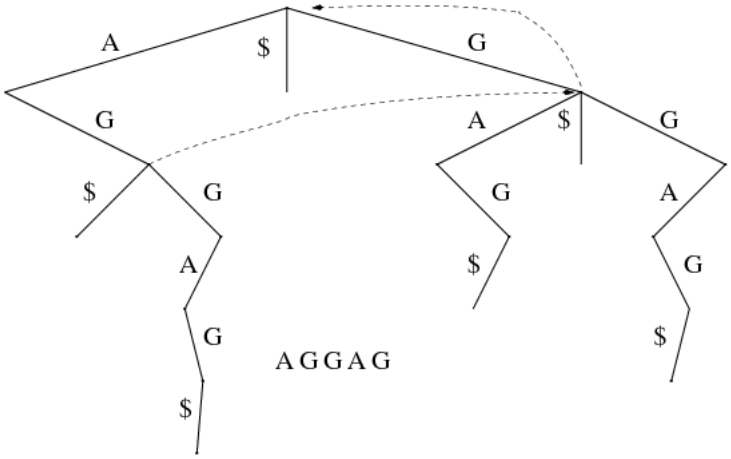


FIG. A.5 – Example of a suffix tree for the text *AGGAG*\$. The dashed lines represent the suffix links.

The algorithm is divided into $|t|$ phases. The i^{th} phase (for $0 \leq i < |t|$) consists in the insertion of all the $i+1$ suffixes of $t[0,i]$ into the tree. The naive approach divides each phase i into $i+1$ steps, one step j ($0 \leq j < i$) consisting in the insertion of the suffix $t[j,i]$ into the tree. This naive version of the construction algorithm is presented in Figure 12. Clearly this algorithm is in $O(|t|^3)$.

Algorithme 12 Naive suffix tree construction algorithm

Nécessite : A text t
Effectue : The suffix tree $ST(t)$ of t

```
1: pour i from 0 to |t| − 1 faire
2:    pour j from 0 to i faire
3:       Add(ST(t),t[j, i])
4:    fin pour
5: fin pour
```

The Ukkonen algorithm uses three *tricks* in order to reduce the time complexity to $O(|t|)$.

Before we present those three tricks, we describe the encoding of a suffix tree. The suffix tree created by this algorithm does not store the text : each node $\mathcal{N}$ contains a couple of integers (s,e) corresponding to the starting and

ending positions of the factor in the text that led to the creation of the node itself. In the following, we denote by $\mathcal{N}_{s,e}$ such a node. Thus, by definition, in the suffix tree of a text t, $path(\mathcal{N}_{s,e})$ is equal to $t[s,e]$.

The Ukkonen algorithm uses suffix links. A *suffix link* is an oriented link between two branching nodes of a suffix tree. Given a node $\mathcal{N}_{s,e}$, its suffix link is denoted by $S_l(\mathcal{N}_{s,e})$ and the node pointed by $S_l(\mathcal{N}_{s,e})$ is denoted by $S_n(\mathcal{N}_{s,e})$. In this case, $path\left(S_n(\mathcal{N}_{s,e})\right) = path(\mathcal{N}_{s,e})\big[1, |path(\mathcal{N}_{s,e})|\,\big]$. For instance, if $path(\mathcal{N}_{s,e}) = AGGT$, then, $path(S_n(\mathcal{N}_{s,e})) = GGT$.

In Figure A.5, the suffix links are represented by dashed lines.

We present the three ideas leading to a linear time complexity for constructing a suffix tree for the text t.

1. Let us assume that the suffix tree is constructed for $t[0, i-1]$. During the i^{th} phase, all the leaves have to be lengthened by one in order to take the character t_i into account. In other terms, the ending integer e of each leaf has to be incremented by one. Since by definition, all leaves have the same ending integer, the latter can be coded by a global variable that is incremented by one at each phase of the Ukkonen algorithm. This global variable is equal to i during phase i. Thus, the extension of the leaves is implicit and done in constant time.

2. (a) *Fast Insertion* : during the i^{th} phase, let $\mathcal{N}_{s,e}$ be the last branching node reached during the insertion of $t[j,i]$. By construction this node contains a suffix link. In this case, $t[j,i] = path(\mathcal{N}_{s,e}).w.\sigma$ where $w \in \Sigma^*$ and $\sigma \in \Sigma$. In order to insert $t[j+1,i]$, w (which is necessarily already in the tree) is read from $S_n(\mathcal{N}_{s,e})$ and σ is added if needed.

   To avoid having to read all the letters of w from $S_n(\mathcal{N}_{s,e})$, the following trick is used. At each branching node met during the reading of w, an edge is chosen depending on the current letter in w. Once the edge is identified, the node pointed by this edge is reached and we advance in the reading of w by the number of letters in the edge. The process is repeated while w is not totally read. Thus the complexity of the reading of w is related to the number of nodes traversed and not to $|w|$.

   If σ is added, a branching node is created. The suffix link of such a node points to the last branching node met during the next insertion (it can be a created one).

   The pseudo-code of this algorithm is given in annex in Figure 13.

(b) During phase i, all the suffixes of $t[0,i]$ have to be inserted. Yet if during the insertion of $t[j,i]$, this word is already in the tree, then, by definition, all the words $\{t[k,i], k \in [j,i]\}$ are already in the tree as well. In this case, the i^{th} phase stops here. Similarly, the $(i+1)^{th}$ phase can start inserting $t[j,i+1]$: with the implicit extension of the leaves, the factors $\{t[k,i+1], k \in [1,j-1]\}$ are already in the tree.

A pseudo-code of this construction algorithm is given in annex in Figure 14.

Each phase of the algorithm is not done in constant time. However the amortised construction time is linear with respect to the input text length. The demonstration of this complexity is given in [Gus97]. It consists in bounding the overall number of nodes traversed during all insertions.

A.3.2 Construction algorithm of a k-factor tree

The k-factor tree, also called truncated suffix tree, has been presented in [NAIP03] and [AS04]. A k-factor tree is a suffix tree cut such that each word spelt from the root to a leaf has a length bounded by k. An example of k-factor tree is given in Figure A.6. This structure finds applications in various areas such as data compression [NAIP03] [NP00] where the indexation is made over a sliding window, or string matching and computational biology [TDSG05] [MS00b] [PCMS06] where the length of the motifs searched for in the text is bounded.

The linear time construction algorithm we describe here is based on the Ukkonen suffix tree construction algorithm. For further details on implementation and proof of validity, the reader is referred to [AS04].

This algorithm is divided in two parts :

1. First part, we build the suffix tree for $t[0,k-2]$.
2. Second part, we add in $|t|-k+1$ phases the suffixes of $t[i-k,i]$ for i from $k-1$ to $|t|-1$.

The first part is achieved using the Ukkonen algorithm. During this part, the leaves created are added to a queue called $queue_{leaf}$.

In the second part, we have to modify the Ukkonen algorithm so that :

– for each phase i, we start by inserting $t[j,i]$ with j not smaller than $i-k+1$;

– implicit leaf extensions are stopped when the length k is reached for the path of a leaf.

To do this last point, we use the queue $queue_{leaf}$.

During the whole construction, each leaf created is added at the end of $queue_{leaf}$. In the second part, for each phase i, there are two possibilities : either $queue_{leaf}$ is empty or not.

Suppose $queue_{leaf}$ contains at least one leaf. Let $\mathcal{L}_{s,e}$ denote a leaf starting position s and ending position e. We then have $queue_{leaf} = \mathcal{L}^1_{s^1,e} \dots \mathcal{L}^p_{s^p,e}$. We start by fixing the end position of $\mathcal{L}^1_{s^1,e}$ to i, that is $\mathcal{L}^1_{s1,e}$ becomes $\mathcal{L}^1_{s^1,i}$. Indeed, we know that $path(\mathcal{L}^1_{s^1,i})$ has a length of k.

Suppose we are in phase $i = k - 1$. Then $queue_{leaf}$ contains at least one leaf which corresponds to the one created during the insertion of $t[0]$ in the tree (first insertion of the first phase). This leaf is $\mathcal{L}^1_{0,e}$. In phase $i = k$, the leaf is now $\mathcal{L}^1_{0,k-1}$, so its length is equal to k. If there is another leaf in the queue, it corresponds to $\mathcal{L}^1_{1,e}$ and it is clear that its length will be equal to k at the next phase. And so on, as the leaves $\mathcal{L}_{s,e}$ are created with s incremented by one between two leaves.

Once the leaf at the beginning of $queue_{leaf}$ is fixed, we apply again the Ukkonen algorithm from the last leaf in $queue_{leaf}$ (the last created which can be the one we have just fixed). At the end of phase (*i.e.* no leaf created during the last insertion), we remove the leaf at the head of $queue_{leaf}$.

We describe now the case when there is no leaf in the queue. Suppose there were a leaf in the queue at the previous phase $i - 1$. By fixing the end value of this leaf, we have fixed the leaf corresponding to $t[i - k, i - 1]$. Then we started by inserting $t[i - k + 1, i - 1]$ in the tree. This insertion did not create a leaf ($queue_{leaf}$ is empty in phase i) and lead to a position p in the tree that corresponds to the spelling of $t[i-k+1, i-1]$. In the current phase i, since $queue_{leaf}$ is empty, we have to start by inserting $t[i - k + 1, i]$ in the tree. This can be done in constant time by trying to insert $t[i]$ from the position p. If this insertion creates a leaf, its end value is directly set to i (not added in $queue_{leaf}$) and it is used to try to insert $t[i - k + 1, i]$. If no leaf is created, then we continue by trying to insert $t[i - k + 1, i]$ from the leaf reached (we know that the insertion of $t[i - k, i]$ leads to a leaf since the path length of the leaves is bounded by k). If the insertion of $t[i - k + 1, i]$ does not create a leaf, we use the position reached in the tree to start the next phase.

A pseudo-code of this algorithm is given in annex in Figure 14.

The time and space complexities of the algorithm are linear in the size of the input text (see [AS04] for details).

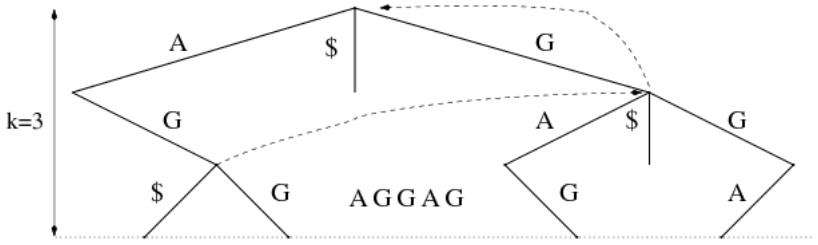


FIG. A.6 – Example of a k-factor tree. for the text $AGGAG\$$ with $k = 3$.

A.3.3 Gapped-factor tree construction

We now present the construction algorithm of a gapped-factor tree (**gft** for short). Once again, the construction algorithm is done online. As shown in Figure A.4, a gft is composed of three different regions : the upper part of depth k, the invisible region corresponding to the gap of length d, and the lower part of depth k' :

1. During the construction of the gft, the first region is treated exactly as for a k-factor tree. The queue containing the leaves in extension is denoted by $queue_{leaf\_up}$.
2. When a leaf reaches the depth k, it enters in the invisible region for d phases. To simulate this behaviour, a queue is created that contains the leaves in extension in the invisible region. This queue is denoted by $queue_{invisible}$. Leaves entering $queue_{invisible}$ stay inside for d phases. During those phases, leaves inside the queue are ignored. After d phases, a leaf in the queue is virtually reaching the depth $k + d$. It is then removed from the queue.
3. The construction algorithm of the lower part of the tree is again very similar to the one of a k-factor tree. All the tricks applied for the suffix tree construction are still available. Once more a queue is used to store the leaves in extension in the lower part of the tree. This queue is denoted by $queue_{leaf\_low}$. The ending integer of the leaves in extension in the queue is the global variable i. The leaves stay in the queue during

k' phases before they become leaves that stay fixed, and contain the positions of the gapped-factors corresponding to the path leading to them from the root.

However, for the construction of the lower part, the use made of suffix links is slightly different than in the upper part of the tree. This is due to the following particularity of the gapped-factor tree : a node in the lower part of the tree may have up to $|\Sigma|$ suffix links. Indeed, one node in this tree may correspond to several paths. According to the first letter in the invisible region leading to a node, the suffix link to follow will not be the same. Figure A.7 illustrates this observation.

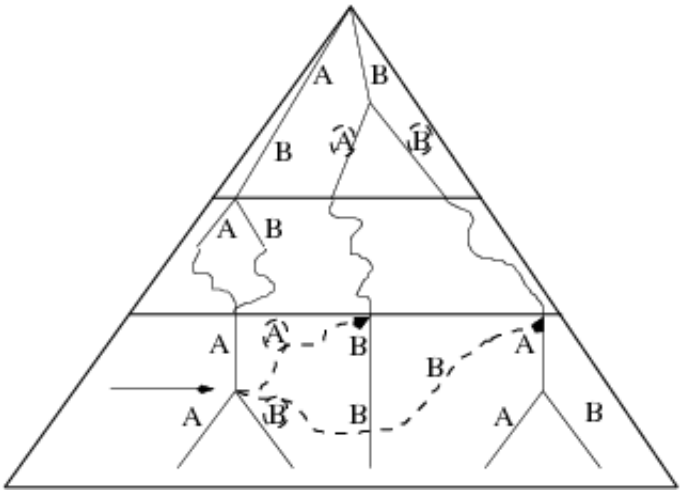


FIG. A.7 – Example of multiple suffix links. The node pointed by an arrow has two suffix links (in dotted line). One is labelled with an A and the other is labelled with a B. The correct suffix link to follow depends on the path that leads to the node. If the node is reached reading $ABA.w$ ($w \in \Sigma^*$), the correct suffix link to follow is the one labelled with an A; it goes to a node reachable reading the text $BA.w$. Any other suffix link leaving the node would be labelled differently and would reach a node corresponding to the text $B.\sigma.w$, with $\sigma \in \Sigma$ and $\sigma \neq A$.

The algorithm 19 given in annex gives an overview of the whole gapped-factor tree construction algorithm.

Complexity of the Gft construction

The algorithm for constructing a gft uses all the tricks employed by Ukkonen and Allali to lead to a linear time and memory complexity. However, the multiple suffix links add a multiplicative term in $|\Sigma|$ to both complexities. Thus the total time and memory complexity for the construction of a gapped-factor tree for a text t is in $O(|t| \times |\Sigma|)$. One can notice that once the gapped-factor is constructed, the (multiple) suffix links are not useful anymore and can be removed. In this case, the memory complexity falls back to $O(|t|)$.

Generalisation to more than one text

As for the suffix tree or the k-factor tree, the gft can be extended to a *generalised gapped-factor* tree and accept a set of $m > 1$ texts $t_0,\ t_1,\ \ldots,\ t_{m-1}$.

In this case, each text $i \in [0, m-1]$ ends with a special character $\$_i$ and the leaves are labelled not only with the positions of a gapped-factor but also with the sequence number in $[0, m-1]$ where the factors occur. The complexity for constructing a generalised gapped-factor tree is in $O\left(\left(\sum_{i=0}^{m-1} |t_i|\right) \times |\Sigma|\right)$.

A.4 Basic uses of a gapped-factor tree

To find all the positions where a (k-d-k')-gapped-factor occurs in a text given a (k-d-k')-gapped-factor tree for the text one needs to find the leaf corresponding to the given gapped-factor. This is done straightforwardly by traversing the gapped-factor tree from the root to the node as in a suffix tree. The list attached to the leaf corresponds to the positions of the occurrences of the gapped-factors.

This algorithm takes a time proportional to the number of nodes traversed, which is in the worst case $k+k'$. Thus retrieving the positions of a given (k-d-k')-gapped-factor is done in $O(k + k')$.

The gft data structure allows also to easily find all the repeated gapped-factors of a text or of a set of texts. If we are interested in finding all gapped-factors occurring at least r times in a text, for r a positive integer, we just have to visit the leaves. For each leaf, if the number of elements of the list

attached to it is greater or equal to r, the corresponding gapped-factor is considered as repeated.

As the number of elements of each list may be stored in the leaves, this extraction is done in time proportional to the number of leaves. If n denotes the length of the indexed text, the number of leaves is no greater than n. The extraction is therefore done in time $O(n)$.

In the generalised case, one may want to extract all gapped-factors occurring in at least r different texts. In this case, to each leaf is attached the number of different texts in which the corresponding gapped-factor occurs. Thus extracting all gapped-factors occurring in at least r different texts is done by checking each leaf in constant time leading to a complexity in $O(\sum_{i=1}^{m} |t_i|)$.

Conclusion

We presented a new data structure used for indexing factors containing a gap (called the gapped-factors). This data structure is based on the suffix tree structure. Furthermore, we indicated an online construction algorithm of this data structure for a text t on an alphabet Σ in $O(|t| \times |\Sigma|)$ time and space. This algorithm is based on the Ukkonen algorithm for constructing a suffix tree.

A.5 Pseudo-codes

Algorithme 13 Fast Insertion

Nécessite : $\mathcal{N}$,t,$start$,end

Effectue : Insert a string $t_{start...end}$ from a node $\mathcal{N}$ assuming that the tree is already constructed for $t_{start...end-1}$ from $\mathcal{N}$

1: $endJump \leftarrow false$
2: **tant que** (not $endJump$) and $((end - start) \neq 0)$ **faire**
3: set $child$ to the child of $\mathcal{N}$ that starts with the letter t_{start}
4: **si** $(end - start) \geq length(\mathcal{N}, child)$ **alors**
5: $start \leftarrow start + length(\mathcal{N}, child)$
6: $\mathcal{N} \leftarrow child$
7: **sinon**
8: $endJump \leftarrow true$
9: **fin si**
10: **fin tant que**
11: **si** $(end - start) = 0$ and $\mathcal{N}$ has not a child for letter t_{end} **alors**
12: add a child to $\mathcal{N}$ with edge label start equal to end
13: **fin si**
14: $e \leftarrow$the label of the edge between $\mathcal{N}$ and $child$
15: **si** $e_{end-start+1} \neq s_{end}$ **alors**
16: split e at position $end - start$
17: add a leaf with start position equal to end to the new node
18: **fin si**

Algorithme 15 Suffix tree and k-factor tree construction

Nécessite : R, t, k, i, $\underline{queue_{leaf}}$, $lastLeaf$

Effectue : One phase of the construction of the suffix tree and of the k-factor tree. The underline parts stand only for the k-factor tree construction.

1: $endPhase \leftarrow false$
2: **répète**
3: $forward \leftarrow length(Father(lastLeaf), lastLeaf) - 1$
4: **si** $S_l(Father(lastLeaf))$ is undefined and $Father(lastLeaf)! = R$ **alors**
5: $forward \leftarrow forward + length(Father(Father(lastLeaf)), Father(lastLeaf))$
6: **si** $Father(Father(lastLeaf))$ is R **alors**
7: AddString(R,t,$i - forward + 1$,i)
8: **sinon**
9: AddString($S_l(Father(Father(lastLeaf)))$,$t$,$i - forward$,$i$)
10: **fin si**
11: **sinon**
12: **si** $Father(lastLeaf)$ is R **alors**
13: AddString(R,t,$i - forward + 1$,i)
14: **sinon**
15: AddString($S_l(Father(lastLeaf))$,t,$i - forward$,i)
16: **fin si**
17: **fin si**
18: **si** a node was created during the previous step **alors**
19: set the suffix link of this node to the last node reached during the insertion
20: **fin si**
21: **si** a leaf was created in the call to AddString **alors**
22: set $lastLeaf$ to this leaf
23: add this leaf at the end of $\underline{queue_{leaf}}$
24: **fin si**
25: **si** no node was created during the call to AddString **alors**
26: $endPhase \leftarrow true$
27: **fin si**
28: **tant** not $endPhase$

Algorithme 16 Factor Tree

Nécessite : R, t , k , $queue_{leaf}$
Effectue : The k-factor tree of t

do the first $k-1$ phases using Suffix_Tree algorithm, filling $queue_{leaf}$ with each new leaf created
pour i **from** k **to** $|t|$ **faire**
  si $queue_{leaf}$ is not empty **alors**
    set $lastLeaf$ to the leaf at the end of $queue_{leaf}$
  sinon
    add t_i from last position reached during the last insertion
    si a leaf is created **alors**
      add this leaf at the end of $queue_{leaf}$
      set $lastLeaf$ to this leaf
    sinon
      set $lastLeaf$ to the leaf reached
    fin si
  fin si
  Phase (R, t , k, i, $queue_{leaf}$, $lastLeaf$)
  remove the leaf at the head of $queue_{leaf}$ and set its end value to i
fin pour
renvoie R

Algorithme 17 Suffix Tree

Nécessite : t
Effectue : The suffix tree of t

Add to R a leaf L with edge label t_0
$lastLeaf \leftarrow L$
pour i **from** 1 **to** $|t|-1$ **faire**
  Phase (R, t , k, i, $lastLeaf$)
fin pour
renvoie R

Algorithme 18 Lower_Part_Tree

Nécessite : R, t , k , d, i, $queue_{leaf\_low}$, $lastLeafLow$

Effectue : A construction phase of the lower part of the gapped-factor tree

1: $endPhaseLow \leftarrow false$
2: **répète**
3: $forward \leftarrow length(Father(lastLeafLow), lastLeafLow) - 1$
4: **si** $S_l(t, Father(lastLeafLow))$ is defined but not labeled with the good character **alors**
5: Point the $Father(lastLeafLow)$ node as the node created during the previous step
6: **fin si**
7: **si** $S_l(t_\alpha, Father(lastLeafLow))$ is undefined and $Father(lastLeafLow)! = R$ **alors**
8: $forward \leftarrow forward+length(Father(Father(lastLeafLow)), Father(lastLeafLow))$
9: **si** $Father(Father(lastLeafLow))$ is R **alors**
10: $AddString(R,t,i - forward + 1,i)$
11: **sinon**
12: AddString($S_l(t_\alpha, F(F(lastLeafLow)))t,i - forward,i)$
13: **fin si**
14: **sinon**
15: **si** $Father(lastLeafLow)$ is R **alors**
16: $AddString(R,t,i - forward + 1,i)$
17: **sinon**
18: $AddString(S_l(t_\alpha, F(lastLeafLow),t,i - forward,i)$
19: **fin si**
20: **fin si**
21: **si** a node was created during the previous step **alors**
22: set the suffix link labeled t_α of this node to the last node reached during the insertion
23: **fin si**
24: **si** a leaf was created during the call to AddString **alors**
25: set $lastLeafLow$ to this leaf
26: add this leaf at the end of $queue_{leaf\_low}$
27: **fin si**
28: **si** no node was created in the call to AddString **alors**
29: $endPhaseLow \leftarrow true$
30: **fin si**
31: **si** the width of the last position reached during the fast insertion was $\leq k + d$ **alors**
32: $endPhaseLow \leftarrow true$
33: **fin si**
34: **tant** not $endPhaseLow$

NOTE : α is the first character in the invisible region on the $lastLeafLow$ path

Algorithme 19 GappedFactor_Tree

Nécessite : R, t , k , d, $queue_{leaf\_up}$, $queue_{leaf\_low}$, $queue_{invisible}$
Effectue : Complete construction algorithm of a gapped factor tee

1: do the first k phases using Suffix_Tree algorithm, filling $queue_{leaf\_up}$ with each new leaf created
2: **pour** i **from** k **to** $|t|$ **faire**
3: **si** index of node at the head of $queue_{invisible} = k + 1$ **alors**
4: create a new edge from this node labeled t_i
5: add the new leaf at the end of $queue_{leaf\_low}$
6: remove the node at head of $queue_{invisible}$
7: **fin si**
8: **si** $queue_{leaf\_up}$ is not empty **alors**
9: set $lastLeafUp$ to the leaf at the end of $queue_{leaf\_up}$
10: **sinon**
11: add t_i from last position reached during the last insertion on the upper part of the tree
12: **si** a leaf is created **alors**
13: add this leaf at the end of $queue_{leaf\_up}$
14: set $lastLeafUp$ to this leaf
15: **sinon**
16: set $lastLeafUp$ to the leaf reached
17: **fin si**
18: **fin si**
19: $Phase(R, t, k, i, queue_{leaf\_up}, lastLeafUp)$
20: remove the pseudo leaf at the head of $queue_{leaf\_up}$
21: **si** the pseudo leaf is new **alors**
22: set the pseudo leaf index value to 0 (invisible zone)
23: add the pseudo leaf at the end of $queue_{invisible}$
24: **fin si**
25: **si** $i > k + d$ *//the lower part of the tree is on construction* **alors**
26: **si** $queue_{leaf\_low}$ is not empty **alors**
27: set $lastLeafLow$ to the leaf at the end of $queue_{leaf\_low}$
28: **sinon**
29: add t_i from last position reached during the last insertion on the low part of the tree
30: **si** a leaf is created **alors**
31: add this leaf at the end of $queue_{leaf\_low}$
32: set $lastLeafLow$ to this leaf
33: **sinon**
34: set $lastLeafLow$ to the leaf reached
35: **fin si**
36: **fin si**
37: $Lower\_Part\_Tree(R, t, k, d, i, queue_{leaf\_low}, lastLeafLow)$
38: **si** $i \geq k + d + k$ *// End the extention of the tree* **alors**
39: remove the leaf at the head of $queue_{leaf\_low}$
40: set the leaf end value to i

Bibliographie

[AAA03] S. A. Aghili, D. Agrawal, and A. El Abbadi. BFT : A Relational-based Bit Filtration Technique for Efficient Approximate String Joins in Biological Databases. *String Processing and Information Retrieval (SPIRE 2003)*, 2857 of LNCS :326–340, 2003.

[ABL+90] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J.D. Watson. *Biologie moléculaire de la cellule.* Médecine-Science Flammarion, 1990.

[AGM+90] S.F. Altschul, W. Gish, W. Miller, E.M. Myers, and D.J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3) :403–410, 1990.

[AHI+06] P. Antoniou, J. Holub, C. S. Iliopoulos, B. Melichar, and P. Peterlongo. Finding Common Motifs with Gaps using Finite Automata. In *Proceedings of the 11th International Conference on Implementation and Application of Automata (CIAA'06)*, 2006.

[AKO04] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing Suffix Trees With Enhanced Suffix Arrays. *Journal Discrete Algorithms*, 2(1) :53–86, 2004.

[Alt89] S. F. Altschul. Gap Costs for Multiple Sequence Alignment. *Jounal of Theoretical Biology*, 138 :297–309, 1989.

[AMS+97] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped BLAST and PSI–BLAST : a New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 25 :3389–3402, 1997.

[AS04] J. Allali and M.F. Sagot. The at Most k-deep Factor Tree. Technical Report #2004-03, Institut Gaspard Monge, Université de Marne-la-Vallée, 2004.

[ASAA04] S. A. Aghili, O. D. Sahin, D. Agrawal, and A. El Abbadi. Efficient Filtration of Sequence Similarity Search Through Singular Value Decomposition. In *Fourth IEEE Symposium on Bioinformatics and Bioengineering (BIBE'04)*, 2004.

[BBV04] B. Brejova, D.G. Brown, and T. Vinar. Optimal Spaced Seeds for Homologous Coding Regions. *Journal of Bioinformatics and Computational Biology*, 1(4) :595–610, January 2004.

[BBV05] B. Brejová, D. Brown, and T. Vinar. Vector seeds : An extension to spaced seeds. *Journal of Computer and System Science*, 70(3) :364–380, 2005.

[BCF+99] S. Burkhardt, A. Crauser, P. Ferragina, H. P. Lenhof, and M. Vingron. q-Gram Based Database Searching Using a Suffix Array (QUASAR). *Proceedings of the third annual international conference on Computational molecular biology (Recomb 99)*, pages 77–83, 1999.

[BCG+03] M. Brudno, M. Chapman, B. Göttgens, S. Batzoglou, and B. Morgenstern. Fast and Sensitive Multiple Alignment of Large Genomic Sequences. *BMC Bioinformatics*, 4 :66, 2003.

[BDC+03] M. Brudno, C. B. Do, G. M. Cooper, M.F. Kim, E. Davydov, E. D. Green, A. Sidow, and S. Batzoglou. LAGAN and Multi-LAGAN : Efficient Tools for Large-Scale Multiple Alignment of Genomic DNA. *Genome Research*, 13 :721–731, 2003.

[Ben97] G. Benson. Sequence Alignment With Tandem Duplication. *Proceedings of the 1st Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 27–36, 1997.

[Ben99] G. Benson. Tandem Repeats Finder : A Program to Analyze DNA Sequences. *Nucleic Acid Research*, 27(2) :573–580, 1999.

[BK01] S. Burkhardt and J. Karkkainen. Better Filtering with Gapped q-Grams. *12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001)*, 56 of LNCS :51–70, 2001.

[BK02] S. Burkhardt and J. Karkkainen. One-Gapped q-Gram Filters for Levenshtein Distance. *13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002)*, 2373 of LNCS :225–234, 2002.

[BKS03] J. Buhler, U. Keich, and Y. Sun. Designing Seeds for Similarity Search in Genomic DNA. In Webb Miller, Martin Vingron, Sorin Istrail, Pavel Pevzner, and Michael Waterman, editors, *Proceedings of the seventh annual international conference on Computational molecular biology (RECOMB-03)*, pages 67–75, New York, April 10–14 2003. ACM Press.

[Bla99] P. E. Black. *Algorithms and Theory of Computation Handbook*. CRC Press, pub-CRC :adr, 1999.

[BMO+03] D. Boffelli, J. McAuliffe, D. Ovcharenko, K.D. Lewis I. Ovcharenko, L. Pachter, and E.M. Rubin. Phylogenetic Shadowing of Primate Sequences to Find Functional Regions of the Human Genome. *Science*, 299(5611) :1391–1394, 2003.

[BP04] N. Bray and L. Pachter. MAVID : Constrained Ancestral Alignment of Multiple Sequences. *Genome Research*, 14 :693–699, 2004.

[BPT04] G. Bourque, P. A. Pevzner, and G. Tesler. Reconstructing the Genomic Architecture of Ancestral Mammals : Lessons From Human, Mouse, and Rat Genomes. *Genome Research*, 14 :507–516, 2004.

[Bro04] D. Brown. Multiple vector seeds for protein alignment. In *WABI : International Workshop on Algorithms in Bioinformatics, WABI, LNCS*, 2004.

[BS87] G. J. Barton and M. J. E. Sternberg. A Sstrategy for the Rapid Mmultiple Alignment of Protein Sequences : Confidence Levels from Tertiary 32 Structure Comparisons. *Journal of Molecular Biology*, 198 :327–337, 1987.

[BYP96] R. A. Baeza-Yates and C. H. Perlberg. Fast and Practical Approximate String Matching. *Information Processing Letters*, 59(1) :21–27, 1996.

[CAA+06] G. Cochrane, P. Aldebert, N. Althorpe, M. Andersson, W. Baker, A. Baldwin, K. Bates, S. Bhattacharyya, P. Browne, A. van den Broek, M. Castro, K. Duggan, R. Eberhardt, N. Faruque, J. Gamble, C. Kanz, T. Kulikova, C. Lee, R. Leinonen, Q. Lin, V. Lombard, R. Lopez, M. McHale, H. McWilliam, G. Mukherjee, F. Nardone, M. Pilar Garcia Pastor, S. Sobhany,

P. Stoehr, K. Tzouvara, R. Vaughan, D. Wu, W. Zhu, and R. Apweiler. EMBL Nucleotide Sequence Database : developments in 2005. *Nucleic Acid Research*, 34 :10–15, 2006.

[CFOS05] A. M. Carvalho, A. T. Freitas, A. L. Oliveira, and M-F. Sagot. A Highly Scalable Algorithm for The Extraction of Cis-regulatory Regions. *Advances in Bioinformatics and Computational Biology*, 1 :273–282, 2005.

[CGL04] R. Cole, L-A. Gottlieb, and M. Lewenstein. Dictionary Matching and Indexing with Errors and Don't Cares. In *Proceedings of the 36th annual ACM symposium on Theory of computing*, 2004.

[CL04] C. Charras and T. Lecroq. *Handbook of Exact String Matching Algorithms*. King's College London Publications, 2004.

[CLZU02] M. Crochemore, G. M. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted cost matrices. In D. Eppstein, editor, *Proceedings of the 13 th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 679–688. ACM-SIAM, 2002. Rapport I.G.M. 2001-08.

[Con01] Consortium. Initial Sequencing and Analysis of the Human Genome. *Nature*, 409 :860–921, 2001.

[CR93] A. Califano and I. Rigoutsos. Flash : A Fast Look-up Algorithm for String Homology. In *Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology*, pages 56–64, 1993.

[Cri58] F. Crick. On Protein Synthesis. *Symposia of the Society for Experimental Biology*, 12 :39–163, 1958.

[Cri70] F. Crick. Central Dogma of Molecular Biology. *Nature*, 227 :561–563, 1970.

[Dar59] C. Darwin. *L'Origine des espèces*. Flamarion, 1859.

[DKF+99] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of Whole Genomes. *Nucl. Acids. Res.*, 27(11) :2369–2376, 1999.

[DMBP04a] A.C. Darling, B. Mau, F.R. Blattner, and N.T. Perna. Mauve : Multiple Alignment of Conserved Genomic Sequence with Rearrangements. *Genome Research*, 14 :1394–1403, 2004.

[DMBP04b] A.E. Darling, B. Mau, F.R. Blattner, and N.T. Perna. GRIL : Genome Rearrangement and Inversion Locator. *Bioinformatics*, 20(1) :122–124, 2004.

[Edg04] R. C. Edgar. MUSCLE : Multiple Sequence Alignment with High Accuracy and High Throughput. *Nucleic Acids Research*, Vol. 32, No. 5 :1792–1797, 2004.

[ELN+01] L. Elnitski, J. Li, C.T. Noguchi, W. Miller, and R. Hardison. A Negative Cis-element Regulates the Level of Enhancement by Hypersensitive Site 2 of the beta -Globin Locus Control Region. *J. Biol. Chem.*, 276(9) :6289–6298, 2001.

[FCLST05] M. Farach-Colton, G. M. Landau, S. Cenk Sahinalp, and D. Tsur. Optimal Spaced Seeds for Faster Approximate String Matching. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005*, volume 3580 of *Lecture Notes in Computer Science*, pages 1251–1262. Springer, 2005.

[FHSW04] M.C. Frith, U. Hansen, J. L. Spouge, and Z. Weng. Finding Functionnal Sequence Elements by Multiple Local Alignement. *Nucleic Acid Research*, 32(1) :189–200, 2004.

[GBV02] P. Gilligan, S. Brenner, and B. Venkatesh. Fugu and Human Sequence Comparison Identifies Novel Human Genes and Conserved Non-Coding Sequences. *Genes*, 294(1) :35–44, 2002.

[GDG01] D. Gordon, C Desmarais, and P. Green. Automated Finishing with Autofinish. *Genome Research*, 11(4) :614–625, 2001.

[GIJ+01] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate String Joins in a Database (Almost) for Free. In *In Proc. of 27th Int'l Conf. on Very Large DataBases (VLDB 2001)*, pages 491–500, 2001.

[GL89] R. Grossi and F. Luccio. Simple and Efficient String Matching with k Mismatches. *Information Processing Letters*, 33 :113–120, 1989.

[Got93] O. Gotoh. Optimal Alignment Between Groups of Sequences and its Application to Multiple Sequence Alignment. *Computer applications in the biosciences*, 9(3) :361–370, 1993.

[Got96] O. Gotoh. Significant Improvement in Accuracy of Multiple Protein Sequence Alignments by Iterative Refinement as Assessed by Reference to Structural Alignments. *Journal of Molecular Biology*, 264(4) :823–838, 1996.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology*. Cambridge University Press, 1997.

[Har71] M. C. Harrison. Implementation of the Substring Test by Hashing. *Commun. ACM*, 14(12) :777–779, 1971.

[HBD94] W. Hide, J. Burke, and D.B. Davison. Biological Evaluation of d2, an Algorithm for High-Performance Sequence Comparison. *J Comput Biol.*, 1(3) :199–215, 1994.

[HKO02] M. Höhl, S. Kurtz, and E. Ohlebusch. Efficient Multiple Genome Alignment. *ISMB (Supplement of Bioinformatics)*, Vol. 18 :S312–S320, 2002.

[HPMW05] B. Haubold, N. Pierstorff, F. Möller, and T. Wiehe. Genome Comparison Without Alignment Using Shortest Unique Substrings. *BMC Bioinformatics*, 6 :123, 2005.

[HS77] J. W. Hunt and T. G. Szymanski. A Fast Algorithm for Computing Longest Common Subsequences. *CACM*, 20(5) :350–353, May 1977.

[HS88] H.G. Higgins and P.M. Sharp. CLUSTAL : a Package for Performing Multiple Sequence Alignment on a Microcomputer. *Gene*, 73 :237–244, 1988.

[HYT03] K.-F. Huang, C.-B. Yang, and K.-T. Tseng. An Efficient Algorithm for Multiple Sequence Alignment. 2003.

[ILM+06] C. S. Iliopoulos, I. Lee, M. Mohamed, M. S. Rahman, and W. F. Smyth. Finding Patterns with Variable Length Gaps or Don't Cares. In *Proceedings of the 12th Annual International Computing and Combinatorics Conference (COCOON'06)*, 2006.

[IMP+05] C. S. Iliopoulos, J. Mchugh, P. Peterlongo, N. Pisanti, W. Rytter, and M-F. Sagot. A First Approach to Finding Common Motifs With Gaps. *International Journal of Foundations of Computer Science*, 16(6) :1145–1154, 2005.

[JU91] P. Jokinen and E. Ukkonen. Two Algorithms for Approximate String Matching in Static Texts. In *MFCS*, pages 240–248, 1991.

[KA03] P. Ko and S. Aluru. Space Efficient Linear Time Construction of Suffix Arrays. *14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, 2676 of LNCS :203–210, 2003.

[Kär02] J. Kärkkäinen. Computing the Threshold for q-Gram Filters. *Proceedings of the 8th Scandinavian Workshop on Algorithm Theory (SWAT 2002)*, 2368 of LNCS :348–357, 2002.

[KBK03] R. Kolpakov, G. Bana, and G. Kucherov. MREPS : Efficient and Flexible Detection of Tandem Repeats in DNA. *Nucleic Acid Research*, 31(13) :3672–3678, 2003.

[Kel03] E. Fox Keller. *Le Siècle du gène*. Gallimard, 2003.

[Ken02] W.J. Kent. BLAT—The BLAST-Like Alignment Tool. *Genome Research*, 12(4) :656–664, 2002.

[KLA+01] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time Longest-Common-Prefix Computation in Suffix Arrays and its Applications. *12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001)*, 2089 of LNCS :181–192, 2001.

[KMKM02] K. Katoh, K. Misawa, K. Kuma, and T. Miyata. Mafft : a novel method for rapid multiple sequence alignment based on fast fourier transform. *Nucleic Acid Research*, 30(14) :3059–3066, 2002.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1) :323–350, 1977.

[KNR05a] G. Kucherov, L. Noé, and M. Roytberg. A Unifying Framework for Seed Sensitivity and its Application to Subset Seeds. *WABI*, 3692 of LNCS :251–263, 2005.

[KNR05b] G. Kucherov, L. Noe, and M. Roytberg. Multiseed Lossless Filtration. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 02 :51–61, 2005.

[KR87] R. M. Karp and M. O. Rabin. Efficient Randomized Pattern-matching Algorithms. *IBM J. Res. Dev.*, 31(2) :249–260, 1987.

[KS03] J. Kärkkäinen and P. Sanders. Simple Linear Work Suffix Array Construction. *International Colloquium on Automata, Languages and Programming (ICALP 2003)*, 2719 of LNCS :943–955, 2003.

[KSPP03] D.K. Kim, J.S. Sim, H. Park, and K. Park. Linear-time Construction of Suffix Arrays. *14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, 2676 of LNCS :186–199, 2003.

[LAK89] D J Lipman, S F Altschul, and J D Kececioglu. A Tool for Multiple Sequence Alignment. *Proc Natl Acad Sci*, 86(12) :4412–4415, 1989.

[Lev66] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Cybernetics and Control Theory*, 10 :707–710, 1966.

[Lew05] B. Lewin. *Essential Genes*. Prentice Hall, 2005.

[LGS02] C. Lee, C. Grasso, and M.F. Sharlow. Multiple Sequence Alignment Using Partial Order Graphs. *Bioinformatics*, 18(3) :452–464, 2002.

[LHP+03] L.A. Lettice, S.J.H Heaney, L.A. Purdiea, L. Li, P. de Beer, B.A. Oostra, D. Goode, G. Elgar, R.E Hill, and de E. Graaff. A Long-Range Shh Enhancer Regulates Expression in the Developing Limb and Fin and is Associated with Preaxial Polydactyly. *Human Molecular Genetics*, 12(14) :1725–1735, 2003.

[LLB+00] G.G. Loots, R.M. Locksley, C.M. Blankespoor, Z.E. Wang, W. Miller E.M., Rubin, and K.A. Frazer. Identification of a Coordinate Regulator of Interleukins 4, 13, and 5 by Cross-Species Sequence Comparisons. *Science*, 288 :136–140, April 2000.

[LLDA03] A. Lefebvre, T. Lecroq, H. Dauchel, and J. Alexandre. FORRepeats : Detects Repeats on Entire Chromosomes and Between Genomes. *Bioinformatics*, 19(3) :319–326, 2003.

[LM03] M. Li and B. Ma. PatternHunter II : Highly Sensitive and Fast Homology Search. *Genome Informatics*, 14 :164–175, 2003.

[LP85] D. J. Lipman and W. R. Pearson. Rapid and Sensitive Protein Similarity Searches. *Science*, 227 :1435–1441, 1985.

[LP88] D. Lipman and W. Pearson. Improved Tools for Biological Sequence Comparison. In *PNAS*, pages 2444–2448, 1988.

[McC76] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the Association of Computing Machinery*, 23(2) :262–272, 1976.

[MDV05] M. Michael, C. Dieterich, and M. Vingron. Siteblast Rapid and Sensitive Local Alignment of Genomic Sequences Employing Motif Anchors. *Bioinformatics*, 21(9) :2093–2094, 2005.

[MFDW98] B Morgenstern, K Frech, A Dress, and T Werner. DIALIGN : Finding Local Similarities by Multiple Sequence Alignment. *Bioinformatics*, 14 :290–294, 1998.

[MM90] U. Manber and G. Myers. Suffix Arrays : A New Method for On-Line String Searches. In *SODA*, pages 319–327, 1990.

[MM93] U. Manber and G. Myers. Suffix Arrays : A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5) :935–948, 1993.

[Mor68] D. R. Morrison. PATRICIA : Practical Algorithm to Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4), October 1968.

[MS00a] L. Marsan and M.-F. Sagot. Algorithms for Extracting Structured Motifs Using a Suffix Tree With Application to Promoter and Regulatory Site Consensus Identification. *J. Comput. Bio.*, 7(3/4) :345–360, 2000.

[MS00b] L. Marsan and M-F. Sagot. Extracting Structured Motifs Using a Suffix Tree - Algorithms and Application to Promoter Consensus Identification. In *RECOMB*, pages 210–219, 2000.

[MTL02] B. Ma, J. Tromp, and M. Li. Patternhunter : Faster and More Sensitive Homology Search. *Bioinformatics*, 18(3) :440–445, 2002.

[Mye94] E. Myers. A Sublinear Algorithm for Approximate Keyword Matching. *Algorithmica*, 4-5 :345–374, 1994.

[NAIP03] J.C. Na, A. Apostolico, C.S. Iliopoulos, and K. Park. Truncated Suffix Trees and their Application to Data Compression. *Theoretical Computer Science*, 304(1-3) :87–101, 2003.

[NCBI] NCBI. http ://www.ncbi.nlm.nih.gov/blast/.

[NHH00] C. Notredame, D.G. Higgins, and J. Heringa. T-Coffee : A Novel Method for Fast and Accurate Multiple Sequence Alignment. *Journal of Molecular Biology*, 302(1) :205–217, 2000.

[NOAR03] M.A Nobrega, I. Ovcharenko, V. Afzal, and E.M. Rubin. Scanning Human Gene Deserts for Long-Range Enhancers. *Science*, 302(5644) :413, 2003.

[NP00] J.C. Na and K. Park. Data Compression with Truncated Suffix Trees. *Data Compression Conference (DCC 2000)*, IEEE Computer Society, online edition : http ://computer.org/proceedings/dcc/0592/0592toc.htm :565, 2000.

[NR02] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.

[NSTT00] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing Text with Approximate q-Grams. *11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, 1848 of LNCS :350–363, 2000.

[NT84] J. H. Nadeau and B. A. Taylor. Lengths of Chromosomal Segments Conserved Since Divergence of Man and Mouse. *PNAS*, 81(3) :814–818, 1984.

[NW70] S.B. Needleman and C.D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of two Proteins. *Journal of Molecular Biology*, 48 :443–453, 1970.

[OBL04] I. Ovcharenko, D. Boffelli, and G.G. Loots. eShadow : A Tool for Comparing Closely Related Sequences. *Genome Research*, 14(6) :1191–1198, 2004.

[OLG+05] I. Ovcharenko, G.G. Loots, B.M. Giardine, M. Hou, J. Ma, R.C. Hardison, L. Stubbs, and W. Millers. Mulan : Multiple-Sequence Local Alignment and Visualization for Studying Function and Evolution. *Genome Research*, 15 :184–194, 2005.

[OM88] O. Owolabi and D. R. McGregor. Fast Approximate String Matching. *Software Practice and Experience*, 18(4) :387–393, April 1988.

[PCGS04] N. Pisanti, M. Crochemore, R. Grossi, and M.-F. Sagot. A Comparative Study of Bases for Motif Inference. *String Algorithmics KCL publications*, 1 :195–226, 2004.

[PCMS06] N. Pisanti, A.M. Carvalho, L Marsan, and M-F. Sagot. RISOTTO : Fast Extraction of Motifs with Mismatches. *Proceedings of the 7th Latin American Theoretical Informatics Symposium*, 3887 of LNCS :757–768, 2006.

[POH+01] L.A. Pennacchio, M. Olivier, J.A. Hubacek, J.C. Cohen, D.R. Cox, J.-C. Fruchart, R.M Krauss, and E.M. Rubin. An Apolipoprotein Influencing Triglycerides in Humans and Mice Revealed by Comparative Sequencing. *Science*, 294(5540) :169–173, 2001.

[PPBS05] P. Peterlongo, N. Pisanti, F. Boyer, and M-F. Sagot. Lossless Filter for Finding Long Multiple Approximate Repetitions Using a New Data Structure, the Bi-factor Array. *String Processing and Information Retrieval (SPIRE 2005)*, 3772 of LNCS :179–190, 2005.

[PPT06] Q. Peng, P.A. Pevzner, and G. Tesler. The Fragile Breakage versus Random Breakage Models of Chromosome Evolution. *PLoS Comput Biol*, 2(2) :e14, 2006.

[PT02] P.A. Pevzner and G. Tesler. Genome Rearrangements in Mammalian Evolution : Lessons From Human and Mouse Genomes. *Genome Research*, 13 :37–45, 2002.

[PW95] P.A. Pevzner and M. Waterman. Multiple Filtration and Approximate Pattern Matching. *Algorithmica*, 13 :135–154, 1995.

[RSM05] K. R. Rasmussen, J. Stoye, and E. W. Myers. Efficient q-Gram Filters for Finding All ϵ-Matches Over a Given Length. *9th Annual International Conference, Research in Computational Molecular Biology (Recomb 2005)*, 3678 of LNCS :189–203, 2005.

[RW94] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *Computer*, 27(3) :17–28, March 1994.

[SB04] Y. Sun and J. Buhler. Designing Multiple Simultaneous Seeds for DNA Similarity Search. In *Annual International Conference on (Research in) Computational (Molecular) Biology*, volume 8, 2004.

[SB06] Y. Sun and J. Buhler. Choosing the Best Heuristic for Seeded Alignment of DNA Sequences. *BMC Bioinformatics*, 7 :133, 2006.

[SEL+03] S. Schwartz, L. Elnitski, M. Li, M. Weirauch, C. Riemer, A. Smit, NISC Comparative Sequencing Program, E. D. Green, R. C. Hardison, and W. Miller. Multipipmaker And Supporting Tools : Alignments and Analysis of Multiple Genomic DNAsequences. *Nucleic Acid Research*, 31(13) :3518–3524, 2003.

[SH89] S. Subbiah and S.C. Harrison. A Method for Multiple Sequence Alignment with Gaps. *Journal of Molecular Biology*, 209 :539–548, 1989.

[SKPP03] J.S. Sim, D.K. Kim, H. Park, and K. Park. Linear-Time Search In Suffix Arrays. In *14th Australasian Workshop on Combinatorial Algorithms (AWOCA)*, 2003.

[SMD97] J. Stoye, V. Moulton, and A.W.M. Dress. DCA : An Efficient Implementation of the Divide-And-Conquer Approach to Simultaneous Multiple Sequence Alignment. *Bioinformatics*, 13(6) :625–626, 1997.

[SNC77] F. Sanger, S. Nicklen, and A. R. Coulson. DNA Sequencing with Chain-terminating Inhibitors. In PNAS, editor, *Proceedings of the National Academie of Science*, volume 74, pages 5463–5467, 1977.

[SS98] E. Sutinen and W. Szpankowski. On the Collapse of q-gram Filtration. In *FUN with Algorithms*, pages 178–193, 1998.

[ST95] E. Sutinen and J. Tarhio. On Using q-Gram Locations in Approximate String Matching. *Third Annual European Symposium, (ESA 95)*, 979 of LNCS :327–340, 1995.

[SW81] T.F. Smith and M.S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1) :195–197, 1981.

[T*al.*00] H. Tettelin and *al.* Complete Genome Sequence of Neisseria Meningitidis Serogroup B Strain MC58. *Science*, 287(5459) :1809–1815, 2000.

[Tay86] W.R. Taylor. Identification of Protein Sequence Homology by Consensus Template Alignment. *Journal of Molecular Biology*, 188 :233–258, 1986.

[TDSG05] P. Thébault, S. DeGivry, T. Schiex, and C. Gaspin. Combining Constraint Processing and Pattern Matching to Describe and Locate Structured Motifs in Genomic Sequences. In *Fifth IJCAI-05 Workshop on Modelling and Solving Problems with Constraints, Edindurgh, Scotland*, pages 330–337, 2005.

[THG94] J.D. Thompson, D.G. Higgins, and T.J. Gibson. CLUSTAL W : Improving the Sensitivity of Progressive Multiple Sequence

Alignment Through Sequence Weighting, Position-specific Gap Penalties and Weight Matrix Choice. *Nucleic Acid Research*, 22(22) :4673–4680, 1994.

[Ukk92] E. Ukkonen. Approximate String-Matching with q–Grams and Maximal Matches. *Theoretical Computer Science*, 92 :191–211, 1992.

[Ukk95] E. Ukkonen. On-line Construction of Suffix-Trees. *Algorithmica*, 14 :249–260, 1995.

[Ven01] The Human Genome. *Science*, 291 :1304–1351, 2001.

[WC53] J. Watson and F. Crick. Molecular Structure of Nucleic Acids. *Nature*, 4356 :737–738, 1953.

[Wei73] P. Weiner. Linear Pattern Matching Algorithms. In *FOCS*, pages 1–11, 1973.

[WM92a] S. Wu and U. Manber. Agrep — A Fast Approximate Pattern-Matching Tool. In *Proc. USENIX*, pages 153–162, 1992.

[WM92b] S. Wu and U. Manber. Fast Text Searching Allowing Errors. *Commun. ACM*, 35(10) :83–91, 1992.

[XBLM04] J. Xu, D.G. Brown, M. Li, and B. Ma. Optimizing Multiple Spaced Seeds for Homology Search. *15th Annual Symposium on Combinatorial Pattern Matching (CPM 2004)*, 3109 :47–58, 2004.

[YWC+04] I.H. Yang, S.H. Wang, Y.H. Chen, P.H. Huang, L. Ye ; X. Huang, and K.M. Chao. Efficient Methods for Generating Optimal Single and Multiple Spaced Seeds. In *Fourth IEEE Symposium on Bioinformatics and Bioengineering (BIBE'04) in Bioinformatics and Bioengineering*, pages 411 – 416, 2004.

Index

Glossaire

(L, d)-similarité
: Deux chaînes de caractères dont l'une est de longueur L et telle que la distance (de Hamming ou d'édition) séparant ces deux chaînes est inférieure ou égale à d.

(L, r, d)-répétition
: Ensemble de r chaînes de caractères de longueur L telles que la distance (de Hamming ou d'édition) entre toute paire d'éléments de cet ensemble est inférieure ou égale à d

LCS, Longuest Common Subsequence
: Plus grande sous-séquence entre deux textes

k-facteur
: Mot de longueur k.

k-facteur éclatés
: Ensemble de k caractères d'une séquence s formés d'une sous-séquences de s.

Acide aminé
: Petite molécule dont l'enchaînement compose les protéines. Il existe 20 acides aminés différents utilisés pour fabriquer les protéines.

Acide désoxyribonucléique (ADN)
: Macromolécule biologique formée de l'assemblage linéaire de quatre nucléotides. L'ADN est le principal composant des chromosomes et le support biologique de l'information génétique.

Acide ribonucléique (ARN)
: Macromolécule biologique formée de l'assemblage linéaire de ribonucléotides.

Algorithme
: Suite finie d'étapes, réalisées dans un ordre déterminé, à un nombre fini de données afin d'arriver à un certain résultat.

Allèle
: Formes différentes d'un gène, dérivant les unes des autres par mutation.

Ancre
: Portion de séquence, contigu ou non, utilisée par les filtres de recherche de répétitions.

Arbre Phylogénique
: Représentation en forme d'arbre traduisant les relations de parenté entre des organismes.

Bi-facteur
: Ensemble de deux facteurs, de longueurs potentiellement différentes, espacés par zéro caractère ou plus

Cellule (biologie)
: Unité structurale et fonctionnelle constituant tout ou partie d'un être vivant.

Code génétique
: Système de correspondance permettant de traduire une séquence d'acides nucléiques en protéine. Dans ce système, un triplet de nucléotides, ou codon, désigne un acide aminé.

Codon
: Triplet de nucléotides dans l'ADN ou l'ARN qui détermine le choix d'un acide aminé donné dans une synthèse protéique. Le code étant dégénéré, la plupart des acides aminés possèdent plusieurs codons.

Complexité
: Quantité minimale intrinsèque de ressources (temps, mémoire, taille de la sortie, etc.) nécessaire pour résoudre un problème ou pour exécuter un algorithme.

Distance d'édition
: Nombre minimal d'insertions, délétions et substitutions pour transformer un mot en un autre.

Distance de Hamming
: Nombre minimal de substitutions pour transformer un mot en un autre mot de même longueur.

Elément transposable
: Fragments d'ADN insérés dans le génome d'un organisme hôte ayant la propriété remarquable de se déplacer et de se multiplier dans un génome.

Enzyme
: Protéine permettant d'accélérer jusqu'à des millions de fois les réactions chimiques du métabolisme se déroulant dans le milieu cellulaire ou extracellulaire.

Epissage
: Processus englobant l'excision des introns et la réunion des exons dans l'ARN pré-messager ; on obtient alors l'ARN messager mature.

Espèce
: Groupe d'individus semblables ayant la capacité de se reproduire entre eux.

Facteur de transcription
: Protéine qui régule la transcription d'un gène au niveau des sites de fixation : «*binding sites*»).

Faux-négatifs
: Élément rejeté (*négatif*) à tort (*faux*) par un algorithme.

Faux-positifs
: Élément conservé (*positif*) à tort (*faux*) par un algorithme.

Filtre
: Algorithme permettant de supprimer des données non pertinentes pour un usage ultérieur.

Filtre approché
: Filtre pouvant éventuellement supprimer des données potentiellement pertinentes.

Filtre exact
: Filtre ne supprimant aucune donnée potentiellement pertinente.

Génotype

Constitution génétique d'un individu incluant les gènes qui ne sont pas exprimés contrairement au phénotype.

Gène

Une définition possible de «gène» est la suivante : Unité de transmission héréditaire de l'information génétique. Un gène est un segment d'ADN (ou d'ARN chez virus), qui comprend la séquence codant pour une protéine, et les séquences qui en permettent et régulent l'expression.

Gène orthologue

Gènes d'espèces différentes dont les séquences sont homologues, dérivent d'un même gène ancestral et ont divergé à la suite d'un évènement de spéciation. Peuvent ou non avoir la même fonction.

Gène paralogue

Gènes d'espèces différentes dont les séquences sont homologues, dérivent d'un même gène ancestral et ont divergé à la suite d'une duplication. Peuvent ou non avoir la même fonction.

Graine

Portion de séquence, contigu ou non, utilisée par les filtres de recherche de répétitions.

Heuristique

Algorithme donnant une solution dont l'exactitude n'est pas garantie.

Homéostasie

Capacité à conserver l'équilibre de fonctionnement en dépit des contraintes extérieures.

Information génétique

Ensemble des caractères d'un individu vivant. L'information génétique est contenue dans les cellules sous forme d'ADN.

Insertion

Mutation génétique due à l'insertion d'un ou de plusieurs nucléotides.

Locus

Localisation (site) précise sur un chromosome (peut être un gène ou toute autre position choisie).

Motif structuré
: Ensemble de motifs associés par des contraintes de distances.

Nucléotide
: Base azotée. Unité de construction des acides nucléiques, résultant de l'addition d'un sucre (ribose pour l'ARN et désoxyribose pour l'ADN), d'un groupement phosphate et d'une base azotée. Il existe quatre nucléotides différents pour l'ADN : adénine (A), thymine (T), guanine (G), cytosine (C) et quatre nucléotides différents pour l'ARN : uracile (U), guanine (G), cytosine (C), adénine (A).

Partie fonctionnelle
: Portion du génotype ayant une influence sur le phénotype.

Phénotype
: Manifestation apparente de la constitution du génome. Caractéristique biochimique, physiologique et morphologique d'un individu déterminé par son génotype et l'environnement dans lequel il est exprimé.

Phylogénie
: Voir Arbre Phylogénique.

Point de cassure
: Pour un génome, position où un réarrangement a eu lieu.

Pression de sélection
: Ensemble des contraintes du milieu qui agissent sur une population en favorisant le développement des individus les plus adaptés à ce milieu.

Programmation dynamique
: Programmation permettant le calcul de la solution d'un problème par calculs successifs de solutions de sous-problèmes.

Protéine
: Constituant principal des cellules. Les protéines résultent de la traduction de l'ARN lui même résultant de la transcription de l'ADN.

Rétrotranscription
: Transcription inverse de l'ARN en ADN.

Score de similarité
: Mesure permettant de calculer une similarité entre textes.

Sensibilité d'un filtre

Mesure de la proportion de données correctement conservées par un filtre par rapport à la quantité totale de données à conserver.

Sous-séquence (d'une séquence)

Séquence formée à partir d'une séquence originale, en en supprimant certains caractères, sans modifier l'ordre relatif des éléments conservés.

Spécificité d'un filtre

Mesure de la proportion de données correctement supprimées par un filtre par rapport à la quantité totale de données à supprimer.

Stromatolites

Roche fossile dont la structure est présumée avoir été élaborée par des organismes microscopiques, bactéries et algues.

Substitution

Mutation génétique due au remplacement d'un nucléotide par un autre.

Suppression

Mutation génétique due à la suppression d'un ou de plusieurs nucléotides.

Tableau des k-facteurs

Tableau indexant dans l'ordre lexicographique l'ensemble des k-facteurs d'un texte

Tableau des bi-facteurs

Tableau indexant dans l'ordre lexicographique les bi-facteurs d'un texte

Tableau des suffixes

Structure de données indexant dans l'ordre lexicographique l'ensemble des suffixes d'un texte

Taux de conservation

Nombre de nucléotides conservées par un filtre divisé par le nombre total de nucléotides appartenant aux occurrences des répétitions effectivement présentes dans les séquences.

Traduction

Mécanisme au cours duquel les protéines sont synthétisées grâce à la lecture du message génétique inscrit dans l'ARN messager mature.

Transcription
: Mécanisme au cours duquel l'ARN messager est produit à partir de la copie d'un des brins de la molécule d'ADN.

Transposon
: Voir Élément transposable.

Vrai-négatifs
: Élément rejeté (*négatif*) à raison (*vrai*) par un algorithme.

Vrai-positifs
: Élément conservé (*positif*) à raison (*vrai*) par un algorithme.