Recherche de similarités dans les séquences d'ADN : modèles et algorithmes pour la conception de graines efficaces

Laurent Noé

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Recherche de similarités dans les séquences d'ADN : modèles et algorithmes pour la conception de graines efficaces

THÈSE

présentée et soutenue publiquement le 30 septembre 2005

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Laurent Noé

Composition du jury

Président :	Dominique Lavenier,	Directeur de Recherche, CNRS, IRISA
Rapporteurs :	Maxime Crochemore,	Professeur, IGM, Université de Marne-la-Vallée
	Alain Denise,	Professeur, LRI, Université Paris-Sud
Examinateurs :	Alexander Bockmayr,	Professeur, LORIA, Université Henri Poincaré
	Gregory Kucherov,	Directeur de Recherche, CNRS, LIFL
	Hélène Touzet,	Chargé de Recherche, CNRS, LIFL

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Remerciements

Je voudrais tout d'abord remercier Alain Denise et Maxime Crochemore d'avoir accepté d'être les rapporteurs de ce manuscrit. Je tiens également à remercier Alexander Bockmayr, Dominique Lavenier et Hélène Touzet pour avoir accepté de faire partie du jury de thèse.

Un merci particulier à Gregory Kucherov pour m'avoir encadré durant ces trois années et mon stage de DEA, pour sa gentillesse, ses conseils, et son dynamisme. Pour sa compréhension, sa sympathie, et aussi pour nos collaborations, je n'oublierai pas Mikhail Roytberg. Je voudrais également remercier Yann Ponty, pour s'être intéressé à mon problème de génération aléatoire de séquences de score maximal.

Bien entendu, je remercie tous les relecteurs de ce manuscrit pour leurs remarques et leurs corrections. Enfin c'est avec plaisir que j'ai travaillé dans l'équipe Adage, j'en remercie tous les membres présents, les anciens, et tous ceux qui ont été de passage.

Je tiens également à remercier les nombreuses personnes que j'ai eu l'occasion de côtoyer au sein du LORIA et lors de séminaires et conférences.

Merci à ma famille, mes amis, tous ceux qui m'ont supporté (dans tous les sens du terme), et surtout pendant les moments difficiles de la rédaction.

Viva(n) los T quatro libres!
Viva la libertad!!

Table des figures

Liste des tableaux

Table des matières

Notations

Chaînes de caractères

Alphabets

- Σ : alphabet des séquences ($\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ dans le cas de séquences d'ADN).
- $\mathcal{A}$: alphabet des alignements, partition de $\Sigma \times \Sigma$.
- $\mathcal{B}$: alphabet des graines sous-ensemble.

Mots

- α : chaîne de caractères sur $\mathcal{A}^*$.
- $|\alpha|$: longueur de la chaîne de caractères α.
- α_i ou $\alpha[i]$: $i^{ème}$ lettre de la chaîne de caractères α ($i \in [1..|\alpha|]$).
- $\alpha[i..j]$: fragment de la chaîne de caractères α composé des lettres $\alpha_i\alpha_{i+1}\dots\alpha_j$.
- $\alpha \cdot \gamma$: opérateur de concaténation des chaînes α et γ. On le notera également $\alpha\gamma$.

Ensembles

- $\varnothing$ désigne l'ensemble vide.
- 2^E, représente pour un ensemble E, l'ensemble des sous-ensembles de E.
- $E \cup F$ désigne l'union de deux ensembles E et F.
- $E \uplus F$ désigne l'union disjointe de deux ensembles E et F.
- $E \cap F$ désigne l'intersection de deux ensembles E et F.
- $E \backslash F$ désigne la différence entre deux ensembles.
- $E \times F$ désigne le produit cartésien entre deux ensembles.

Expressions régulières

Soit l'alphabet Σ, on considère l'ensemble Σ^* des mots formés sur cet alphabet.

- ε représente le mot vide (longueur 0).
- $E \cdot F$ représente, pour deux ensembles de mots E et F l'opérateur de concaténation : si $u \in E$ et $v \in F$ alors le mot $uv \in E \cdot F$. On le notera également EF.
- $E|F$ représente, pour deux ensembles de mots E et F, l'union $E \cup F$.
- E^* représente la *fermeture de Kleene* pour un ensemble E : il s'agit de l'ensemble contenant le mot vide ε, ainsi que tous les mots formés par concaténation d'un ou plusieurs mots de E.
- E^+ désigne $E^* \backslash \varepsilon$.

Automates / Automates probabilistes

- $T =< Q_T, q_T^0, q_T^F, \mathcal{A}, \psi_T >$: automate fini déterministe
  - Q_T est un ensemble fini d'états.
  - $q_T^0 \in Q_T$ est l'état initial.
  - $q_T^F \subset Q_T$ sont les états finaux.
  - $\mathcal{A}$ est l'alphabet des mots reconnus par l'automate (ici nous prenons l'exemple de l'alphabet des alignements)
  - ψ_T est la fonction de transitions $\psi_T : Q \times \mathcal{A} \to Q$.
- $G =< Q_G, q_G^0, Q_G^F, \mathcal{A}, \rho_G >$: automate fini probabiliste
  - Q_G est un ensemble fini d'états.
  - $q_G^0 \in Q_G$ est l'état initial.
  - $q_G^F \subset Q_G$ sont les états finaux.
  - $\mathcal{A}$ est l'alphabet des mots reconnus par l'automate (ici nous prenons l'exemple de l'alphabet des alignements).
  - ρ_G est la fonction de transition probabiliste : $Q_G \times \mathcal{A} \times Q_G \to [0, 1]$ décrivant les probabilités associées aux transitions. Cette fonction vérifie $\forall q \in Q_G, \sum_{q' \in Q_G, a \in \mathcal{A}} \rho_G(q, a, q') = 1$.

Introduction

Les travaux présentés dans cette thèse concernent le problème de la recherche de similarités dans des séquences, à l'aide de techniques *d'indexation* et de méthodes regroupées généralement sous le dénominatif de *filtrage du texte*. Le domaine d'application visé ici est principalement celui des séquences génomiques : l'**ADN**, l'**ARN**. Leurs caractéristiques (petit alphabet, évolution rapide) comparées aux textes en langage naturel rendent leur indexation délicate, et défavorisent en général les méthodes de comparaison efficaces.

La comparaison de séquences peut être qualifiée de «couteau suisse» de la bio-informatique. Elle est utilisée à de nombreux niveaux depuis le séquençage afin d'assembler des fragments en **contigs**, jusqu'à l'analyse de portions isolées d'un génome à des fins d'annotation. La génomique comparative qui analyse les régions conservées entre plusieurs génomes se base en général sur des outils de recherche de similarités. En discernant les fragments de séquences qui évoluent moins vite entre différentes espèces, elle permet par exemple de découvrir de nouvelles proteines, de nouveaux ARN ou d'inférer des mécanismes de régulation potentiels. C'est une solution à la fois peu coûteuse, car l'analyse est réalisée **in silico**, et universelle en terme de choix du nombre d'espèces et des séquences à analyser.

Avec l'apparition de génomes complets de mammifères, les approches comparatives ont connu un essor récent. La comparaison des génomes de l'homme et de la souris puis la comparaison croisée de différentes espèces de mammifères ont permis par exemple d'isoler des similarités fortes entre espèces à l'aide de méthodes de regroupement [8]. Elles ont également soulevé des questions sur le rôle de certaines de ces régions fortement conservées [77] puisque la majorité d'entre elles sont présentes dans des régions non-codantes pour des protéines. Le manque de données fait qu'il est actuellement difficile d'évaluer les fonctions associées. Il est possible de calculer dans les meilleurs cas des structures secondaires conservées et d'inférer des rôles potentiels, ou d'isoler des sites de fixation, par exemple pour des facteurs de transcription. Ces travaux actuels partent d'espèces proches, et ne concernent que des régions à très forte conservation, relativement faciles à détecter. Cependant, lorsque les espèces comparées ont un ancêtre commun éloigné, les mécanismes de régulation ont évolué à des degrés divers : régions codants pour des les ARN, et les sites de régulations ont conservé la trace de nombreuses mutations, par comparaison aux CDS qui sont en général mieux conservés. Les régions recherchées sont ainsi plus bruitées et la recherche de similarités devient alors plus difficile.

Les algorithmes exacts de comparaison de séquences [83, 97] ont été conçus afin de rechercher toutes les similarités vérifiant certains critères relatifs à un **système de score**. Mais leur coût en temps est quadratique et limite leur utilisation à de courtes séquences d'au plus quelques dizaines de milliers de lettres. Les meilleures améliorations réalisées sur ces algorithmes exacts (comme par exemple [34]) donnent des coûts sous-quadratiques en temps, mais restent encore insuffisantes

pour une utilisation pratique sur des génomes complets. Des méthodes heuristiques de *filtrage* ont donc été mises en place, afin d'isoler les fragments de texte potentiellement similaires. Parmi les plus récentes, citons notamment celles basées sur l'heuristique de *filtrage* à l'aide du modèle de *graines espacées* [75, 59], que nous considérons et étendons dans cette thèse.

* *
*

L'ADN peut être vu comme un texte qui, transmis d'individu à individu, ou de cellule mère à cellules filles, subit principalement des erreurs de recopie (mutations). Ces mutations sont majoritairement constituées par des substitutions, et interviennent sans préférence particulière prononcée sur l'ensemble de la séquence. Elle font par exemple évoluer les gènes dupliqués sur une même espèce (ces derniers sont qualifiés de *paralogues*) ou les gènes présents sur des espèces qui ont divergé par **spéciation** (*orthologues*). Le principal problème se pose lors de la reconnaissance des copies : elles sont bruitées, et l'algorithmique du texte classique se heurte alors à un problème de combinatoire lors de la recherche.

Le principe du *filtrage* consiste à localiser des indications de positions potentielles de similarités avant d'évaluer ces dernières. Cette approche élimine, pour deux séquences, la plupart des paires de positions qui ont peu de chance de contenir une similarité. Elle accélère donc en pratique la recherche, mais en contrepartie, ne garantit pas toujours de détecter toutes les similarités : il s'agit d'une heuristique. Elle ne diminue pas en particulier la borne maximale en temps de l'algorithme qui reste quadratique (plus exactement sous quadratique si l'on utilise l'approche proposée dans [34]) : en effet la taille potentielle du résultat qui peut être obtenu lors d'une comparaison est quadratique dans le pire cas.

Les algorithmes de recherche heuristique de similarités proposés filtrent les séquences en procédant à une recherche de fragments communs du texte [73, 74, 5, 81, 46, 98, 6, 103]. Les fragments de texte recherchés sont contigus et de taille fixée. Si une similarité contient un fragment commun de taille suffisante, alors elle est détectée par l'heuristique, sinon perdue. Des méthodes de *hachage aléatoire* [28, 22] permettent d'améliorer statistiquement la sensibilité en procédant, non plus à une recherche de fragment contigus, mais de fragments dits *espacés* . Un fragment espacé prend en considération, à une position donnée sur une séquence, non la totalité des lettres sur un intervalle, mais seulement un sous-ensemble. Ce sous-ensemble est défini par une liste d'indices fixés de manière aléatoire (voir la figure 1).

FIG. 1 – *Fragment contigu et espacé*

De manière intuitive, cette approche autorise donc certaines mutations (les substitutions) à apparaître sous les positions non considérées par la liste d'indices. Le principe de *hachage aléatoire* a fait l'objet d'études afin de déterminer si certains sous-ensembles d'indices sont plus sensibles que d'autres : ces études [75, 59] ont mené au concept de *graines espacées optimisées* : une *graine espacée optimisée* est un ensemble d'indices choisis afin de détecter au mieux les similarités issues

d'un modèle. C'est dans ce contexte qui couvre à la fois un problème d'algorithmique du texte, de filtrage et d'optimisation que nous nous situons.

Le travail théorique de cette thèse porte principalement sur de nouveaux modèles de graines et de similarités, ainsi que des méthodes et algorithmes permettant de concevoir et d'analyser les graines sur des modèles de similarités. D'un point de vue pratique, nous avons mis en application nos algorithmes et méthodes dans des logiciels dont certains ont été rendus publics (YASS et HEDERA).

* *
*

Ce mémoire de thèse est divisé en 5 chapitres.

Le *premier chapitre* est consacré au cadre biologique, et présente des concepts nécessaires à la compréhension du problème. Le *second chapitre* présente le contexte informatique. Il précise les éléments requis en algorithmique du texte et théorie des automates. Le *troisième chapitre* donne un état de l'art de la recherche de similarités, au travers des méthodes *d'alignement local*. Nous présentons les méthodes exactes de recherche de similarités, suivies des méthodes heuristiques, avec notamment la notion de *graine espacée*. Les quatrième et cinquième chapitres décrivent les apports de cette thèse. Nous abordons dans le *quatrième chapitre* deux extensions développées pour le modèle de graine espacée. La première conditionne le calcul des graines espacées sur des alignements dits homogènes. La seconde, plus générale, propose à la fois un modèle global pour le calcul des graines espacées, ainsi qu'une extension du modèle de graine espacée : les *graines sous-ensemble*. Le modèle de graine sous-ensemble a été appliqué aux séquences d'ADN sous la forme du modèle de *graines à transitions*, proposées dans le logiciel YASS. Le logiciel HEDERA a également été conçu afin de générer des graines à transitions efficaces. Le *cinquième chapitre* développe plus spécifiquement le problème du *filtrage sans perte* pour un autre problème de recherche de similarités que nous avons considéré. Étant donnés deux entiers m, k avec $k < m$, le problème consiste à trouver un *filtre* qui permette de détecter toutes les similarités dont les copies sont de longueur m et qui ont au plus k erreurs de substitution. Nous décrivons dans ce chapitre les méthodes précédemment proposées pour résoudre ce problème, avant de nous intéresser au filtrage à l'aide de graines espacées multiples. Nous analysons en détail cette méthode et la mettons en œuvre pour le problème de la conception d'oligonucléotides.

Les prémices du travail décrit dans le quatrième chapitre ont été présentés aux conférences *European Conference on Computational Biology 2002* et *Research in Computational Molecular Biology 2003* sous la forme de posters. La méthode de calcul des graines espacées sur des alignements homogènes a été publiée dans les actes de la conférence *Bioinformatics and Bioengineering 2004* [63]. La méthode de filtrage sans perte à l'aide de graines multiples a été publiée dans les actes de la conférence *Combinatorial Pattern Matching 2004* [64] et une description complète est parue dans *Transactions on Computational Biology and Bioinformatics* [65]. Le modèle global pour le calcul des graines espacées et le modèle des graines sous-ensemble vont être publiés dans les actes du *Workshop on Algorithms in Bioinformatics 2005* [66]. Une description du logiciel YASS a été publiée dans les actes de la conférence *Journées Ouvertes Biologie Informatique Mathématiques 2004* [87] ainsi que dans le journal *BMC Bioinformatics* [88]. Une description de l'interface web du logiciel YASS a été publiée dans un numéro spécial du journal *Nucleic Acids Research* [89].

Ces travaux ont également fait l'objet de présentations lors de séminaires d'*Actions Spécifiques* ou de séminaires invités. J'ai pour ma part réalisé les exposés lors des séminaires suivants : l'*AS*

Indexation du texte et Découverte de motifs en décembre 2004 à Lille, l'*AS Index et Motifs* en mai 2004 à Nantes, le *Séminaire Symbiose* en mars 2004 à Rennes, le *Séminaire Mathématiques pour le génome* en décembre 2003 à Evry, l'*AS Index et Motifs* en octobre 2003 à Montpellier, l'école *Jeunes Chercheurs en Algorithmique et Calcul Formel* en mars 2003 à Marne-la-Vallée, l'*AS algorithmique et séquences* en janvier 2003 à Nancy.

Chapitre 1

Notions de biologie

Sommaire

1.1 Ce qui a contribué à la naissance de la Bio-informatique

Le terme *Bio-informatique* (*Bioinformatics*) n'apparaît dans la littérature scientifique qu'au tout début des années 90. La littérature anglaise distingue le terme *Bioinformatics* de *Computational Biology*, branche qui s'intéresse aux problèmes biologiques impliquant les algorithmes et les problèmes de combinatoire.

Pour mieux comprendre l'évolution de cette discipline, il faut retourner au XIX^e siècle, en Autriche : Gregor (de son vrai prénom Johann) Mendel découvre alors les lois de l'indépendance de l'hybridation des caractères. Ce capucin morave réalise en 1865 des expériences sur des pois, croisant des lignées pures avec des lignées présentant un ou plusieurs défaut(s) (défaut d'aspect, de couleur) : il découvre les notions de caractère récessif/dominant tout en y associant les indépendances relatives entre caractères. Ces études seront oubliées pendant 35 ans, pour être redécouvertes au début du XX^e siècle simultanément par Correns, Von Teschernak et De Vries. Le concept de gène est inventé par Johannsen en 1905. La théorie chromosomique de l'hérédité se développe avec les travaux de Morgan sur la drosophile : l'information est alors associée aux *chromosomes*.

Ce sont les découvertes issues de la biologie moléculaire qui permettront de comprendre le principe de fonctionnement de la machinerie moléculaire. Avery, Maleod et McCarty identifient en 1944 l'*acide désoxyribonucléique* (ADN) comme le support matériel de l'information génétique. En 1953, Watson et Crick font la (désormais très célèbre) découverte du modèle en double hélice de l'ADN, d'après les travaux de R. E. Franklin. En 1958, Crick découvre les ARN de transfert. Le code génétique est caractérisé de manière globale par Crick et Brenner en 1961 : c'est la découverte de l'association entre tri-nucléotides et acides aminés, association qui est

complètement déchiffrée en 1966 par Khorana. En 1965, Monod, Lwoff et Jacob identifient les premiers mécanismes de régulation des gènes. Les **exons** et **introns** seront découverts en 1977.

Les grands travaux de séquençage en masse débutent avec l'apparition, en 1995, de la séquence du chromosome de la bactérie *Hemophilus Influenzae*, en 1996, des séquences des chromosomes de la levure *Saccharomyces Cerevisiae*. L'intérêt est immédiat, puisqu'il conduit à des conclusions d'importance, en particulier sur les gènes alors non connus qui sont comparés avec des gènes connus d'autres organismes : plus de la moitié présentent des similitudes qui permettent d'inférer des rôles putatifs.

Le séquençage en masse s'est accéléré avec l'apparition de nombreuses séquences de **procaryotes** , en particulier de nombreux micro-organismes **pathogènes** , ainsi que de génomes d'**eucaryotes**, allant du génome de *Caenorhabditis Elegans* (ver) pour le plus ancien en 1995 au génome de *Canis Lupus Domestica* (chien) en 2005, en passant par celui de *Mus Musculus* (souris) en 2003 et celui de *Homo Sapiens* (humain) en 2001.

1.2 Le génome et l'ADN

Le **génome** contient l'information nécessaire pour construire et maintenir le fonctionnement d'une **cellule**, unité de base du vivant. Le génome est composé d'un ou de plusieurs **chromosomes**. Cet ensemble de chromosomes peut être vu comme un support d'information ; le *métabolisme*, c'est-à-dire le fonctionnement des cellules, est quant à lui assuré en majorité par des protéines issues de l'information du génome. Dans les génomes *eucaryotes*, l'information est portée par plusieurs chromosomes. Dans les bactéries, le génome est en général constitué d'un seul chromosome circulaire.

Les chromosomes sont eux-mêmes formés à partir d'une molécule d'ADN (*acide désoxyribonucléique*). Une molécule d'ADN est formée de deux brins complémentaires liés entre eux par des liaisons hydrogènes.

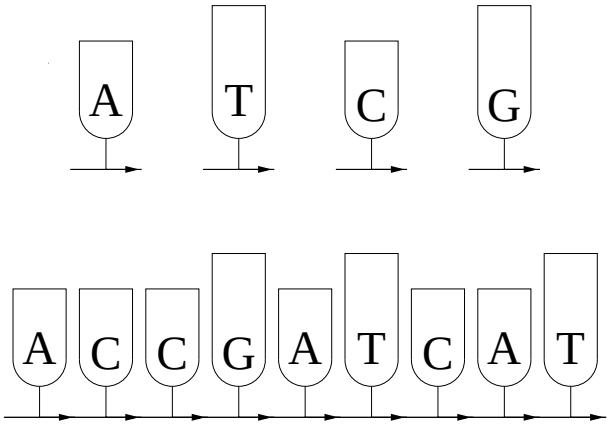


FIG. 1.1 – *Nucléotides et chaîne de nucléotides*

Chaque brin d'ADN est construit à partir d'une longue chaîne de **nucléotides** dont un composant est choisi parmi quatre bases A,T,G,C (*Adénine, Thymine, Guanine, Cytosine*). Ces nucléotides sont chaînés entre eux à l'aide un groupe sucre-phosphate. Ce groupe est polarisé, ce qui donne une orientation à la molécule (voir la figure 1.1).

La molécule d'ADN double-brin est construite sur un principe de complémentarité des bases : l'appariement des deux brins d'ADN dépend de l'affinité (complémentarité) entre les bases A et T d'une part, et les bases G et C d'autre part (voir la figure 1.2).

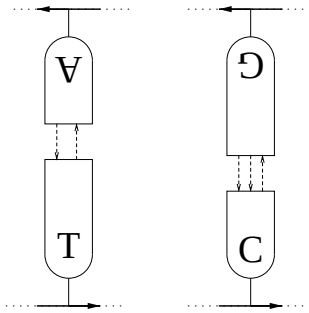


FIG. 1.2 – *Complémentarité des bases*

Cette complémentarité est due à des liaisons chimiques entre les nucléotides appelées *liaisons hydrogènes* (représentées par des flèches en pointillé dans la figure 1.2). Ces liaisons sont réputées être moins solides que les liaisons de covalence classiques des groupes sucre-phosphate. Elles sont cependant suffisamment robustes pour lier deux brins d'ADN à condition que ces derniers soient *complémentaires* (à une lettre sur la première correspond son complément sur la deuxième molécule) mais également *inversés* (l'orientation des deux brins est opposée). Le phénomène de liaison d'un brin à son complémentaire inversé s'appelle *hybridation* (Figure 1.3).

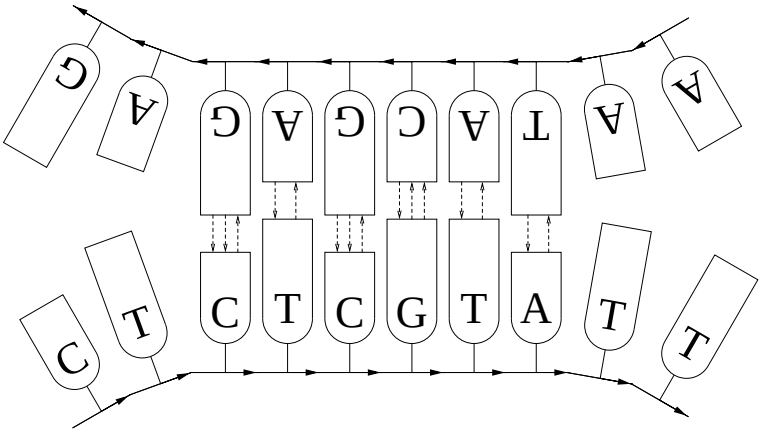


FIG. 1.3 – *Hybridation de séquences complémentaires inversées*

1.3 Information génétique

Le dogme de Crick (qui a découvert avec Watson la structure en double hélice de l'ADN) dit que l'information transmise par un être à sa progéniture ne repose que sur l'ADN. Certaines parties de l'ADN sont *transcrites* en molécules d'ARN (la molécule d'ARN est équivalente à celle de l'ADN, hormis les bases T qui sont substituées par des bases *uracile* U). Ces molécules d'ARN messager sont ensuite *traduites* (terme anglais *translated*) en chaînes polypeptidiques. Les protéines se replient sur elles-mêmes pour prendre une structure tri-dimensionnelle complexe (on parle de *conformation*). Elles interagissent ensuite pour assurer le métabolisme de la cellule.

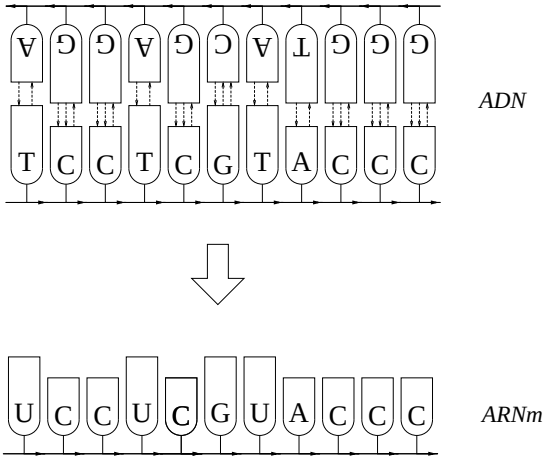


FIG. 1.4 – *Transcription*

1.3.1 Transcription

La **transcription** transforme des parties spécifiques de l'ADN en ARN simple brin appelé *ARN messager* (noté *ARNm* dans l'illustration donnée à la figure 1.4). Cette transformation est faite à l'aide de protéines appelées *facteurs de transcription*, et d'une autre protéine, l'ARN polymérase. L'ARN produit est transformé chez les *eucaryotes* . Il s'agit de l'*épissage* : des sous-séquences appelées *introns* sont retirées de l'ARN, laissant une chaîne formée par la concaténation des parties restantes appelées *exons*.

1.3.2 Traduction

TAB. 1.1 – *Code génétique universel*

UUU Phe	UCU Ser	UAU Tyr	UGU Cys
UUC Phe	UCC Ser	UAC Tyr	UGC Cys
UUA Leu	UCA Ser	UAA Stop	UGA Stop
UUG Leu	UCG Ser	UAG Stop	UGG Trp
CUU Leu	CCU Pro	CAU His	CGU Arg
CUC Leu	CCC Pro	CAC His	CGC Arg
CUA Leu	CCA Pro	CAA Gln	CGA Arg
CUG Leu	CCG Pro	CAG Gln	CGG Arg
AUU Ile	ACU Thr	AAU Asn	AGU Ser
AUC Ile	ACC Thr	AAC Asn	AGC Ser
AUA Ile	ACA Thr	AAA Lys	AGA Arg
AUG STR	ACG Thr	AAG Lys	AGG Arg
GUU Val	GCU Ala	GAU Asp	GGU Gly
GUC Val	GCC Ala	GAC Asp	GGC Gly
GUA Val	GCA Ala	GAA Glu	GGA Gly
GUG Val	GCG Ala	GAG Glu	GGG Gly

La **traduction** est l'étape qui transforme un *ARN messager* en *protéine*. Cette étape fait intervenir un complexe nommé *ribosome* composé de deux ARN et de protéines, ainsi que des

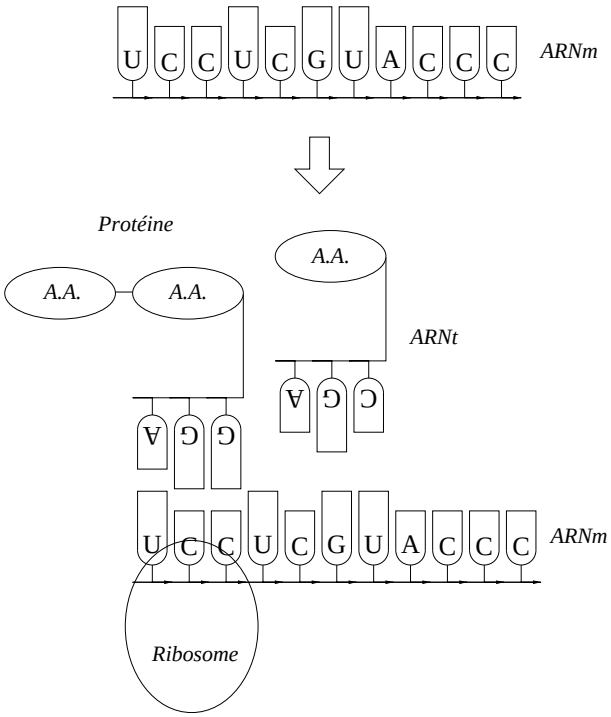


FIG. 1.5 – *Traduction*

ARN de transfert (notés *ARNt* dans l'illustration donnée à la figure 1.5).

Une protéine est composée par une chaîne d'acides aminés (il existe 20 types d'acides aminés différents). Chaque acide aminé est encodé par un triplet de nucléotides appelé *codon* : comme il existe 20 types acides aminés et que le codage utilisé autorise 64 combinaisons, le code utilisé est donc redondant (voir la table 1.1). Un codon particulier AUG sert d'amorce (Start) au *cadre de lecture*, l'ARN est ensuite lu par triplet jusqu'à ce qu'un *codon stop* (défini par UGA,UAA,UAG) soit trouvé.

Le code génétique établissant la correspondance entre codons et acides aminés est identique pour la plupart des organismes : il s'agit du *code génétique universel*. Seuls certains *protozoaires*, et beaucoup de mitochondries ont un code légèrement différent de celui donné dans la table 1.1.

1.4 Évolution des chromosomes

Lors des opérations de **méiose** (processus de division cellulaire qui sépare les chromosomes afin de former les gamètes), les chromosomes subissent des transformations comme les translocations. Il s'agit d'échanges de fragments entre deux chromosomes. Mais durant tout le processus de duplication (**mitose**) qui amène un gamète fécondé à un individu, des erreurs de recopie apparaissent : la séquence d'un chromosome évolue. On distingue parmi ces erreurs, des plus fréquentes au moins fréquentes :

- les **mutations ponctuelles** qui remplacent un nucléotide à une position donnée par un autre (il s'agit par exemple d'une substitution d'une guanine G par une cytosine C). Ces mutations ponctuelles sont séparées en deux familles appelées **transitions** et **transversions**. Elles sont définies de la manière suivante : parmi les 6 paires de substitutions possibles entre nucléotides différents,

  - les substitutions $A \leftrightarrow G$ et $C \leftrightarrow T$ sont appelées **transitions**
  - les 4 autres substitutions sont appelées **transversions**
- les insertions ou suppressions d'un ou plusieurs nucléotides. Ces deux évènements sont appelés **indels**
- les duplications dues à un appariement décalé. Ce sont ces dernières qui sont responsables des répétitions dites **en tandem**
- les duplications classiques dues à un double **crossing-over**, qui donnent en général des **répétitions distantes**

Les duplications (et de manière indirecte les indels) produisent donc deux types de répétitions
- les *répétitions en tandem*, dont les occurrences des copies sont juxtaposées. Ces répétitions sont assez fréquentes aux extrémités des chromosomes (**télomères**) ainsi qu'au centre des chromosomes (**centromère**).
- les *répétitions distantes* sur un ou plusieurs chromosome(s), dues généralement à des duplications, mauvais ré-appariements ou produites par des **transposons**, éléments mobiles du génome. Les répétitions distantes sont réparties sur l'ensemble du génome.

L'évolution due aux mutations ponctuelles les plus fréquentes (substitutions, indels) fait que les copies d'une répétition évoluent. En conséquence, les copies sont de moins en moins ressemblantes et donc de plus en plus difficiles à détecter. En particulier si aucune contrainte n'apparaît, les copies ont tendance à évoluer assez rapidement pour ne plus être discernables. Ce qui est intéressant, bien entendu, ce sont les contraintes qui influent sur cette évolution.

1.5 Mutations et contraintes

Nous nous intéressons dans cette partie aux conséquences des mutations sur les séquences, en prenant plus particulièrement l'exemple des séquences codantes. Nous ferons une analyse rapide des contraintes expliquant la conservation particulière de ces séquences.

Une séquence codante peut être vue comme un fragment d'ADN dont la longueur est un multiple de 3. En effet cette séquence est traduite, selon le code génétique (voir la table 1.1) à l'aide d'une correspondance entre un **codon** (tri-nucléotide) et un acide aminé.

Si l'on considère en détail la table 1.1, il est assez facile de voir qu'une mutation sur la première ou la deuxième base d'un codon change en général complètement l'acide aminé traduit. La protéine construite sera affectée, et peut être alors potentiellement non fonctionnelle, et très probablement moins efficace. Une mutation sur la troisième base d'un codon est :
- sans conséquence (à deux exceptions près) s'il s'agit d'une transition.
- dans la moitié des cas sans conséquence s'il s'agit d'une transversion,

Ce type de mutation est alors qualifié de **mutation silencieuse** Ceci explique potentiellement la fréquence des mutations sur les troisièmes bases des codons, et la sur-abondance des substitutions de type transition, lors de la comparaison de séquences codantes possédant une origine commune.

Bien entendu, les mutations peuvent être soumises à des contraintes autres que celles provenant de régions codantes. Mais les mécanismes ne sont pas forcément compris, ou pas encore connus. En particulier, les mécanismes de régulations (promoteurs) dont l'importance n'est plus à démontrer sont en général mieux conservés, car une mutation sur les motifs associés au promoteur est souvent peu avantageuse ou létale (l'individu ne naîtra pas). Il est par conséquent intéressant de découvrir de tels mécanismes et les motifs associés en comparant plusieurs génomes d'espèces relativement proches [100]. Et cela reste en général un problème difficile.

Chapitre 2

Traitement des séquences

Sommaire

Dans ce chapitre, nous rappelons quelques concepts d'algorithmique. Ce chapitre est divisé en deux sections : la partie 2.1 donne quelque notions d'algorithmique du texte, tandis que la partie 2.2 a pour but de rappeler des concepts de théorie des automates. En particulier, deux algorithmes de minimisation d'automates (partie 2.2.3) et un algorithme de construction d'automate de Aho-Corasick (partie 2.2.4) sont présentés.

2.1 Algorithmique du texte

La comparaison de séquences est un « couteau suisse » de la bio-informatique. De manière générale, l'analyse des parties ressemblantes entre plusieurs séquences, sur le même génome ou des classes déterminées de génomes (bactéries, familles de génomes eucaryotes, etc), permet de mieux comprendre leurs origines, les évolutions, et les liens de parenté entre espèces. Elle permet également de formuler des hypothèses sur les rôles des séquences conservées dans le cadre de la génomique comparative.

Nous allons considérer dans cette partie les méthodes de comparaison de séquences classiques issues de l'algorithmique du texte. Après avoir introduit les événements évolutifs qui affectent les séquences (partie 2.1.1), nous considérerons deux distances, la distance de *Hamming* (partie 2.1.2.1) et la distance de *Levenstein* (partie 2.1.2.2). Nous verrons ensuite la représentation sous

forme d'alignement (partie 2.1.3), afin de nous intéresser à l'algorithme du calcul exact de la distance de Levenstein (partie 2.1.4).

2.1.1 Comparaison de séquences

Les séquences évoluent selon un certain nombre d'opérations d'édition que nous considérerons élémentaires. Ces opérations élémentaires transforment un mot en un autre mot et sont réversibles, ce qui permettra d'y associer une propriété de *symétrie*.

Parmi ces opérations, on distingue :

- les opérations dites de *substitution*, qui transforment un caractère en un autre caractère.
- les insertions et suppressions d'un ou plusieurs caractères. Ces opérations sont appelées *indels* par contraction des termes anglais *insertion* et *deletion*.

Il est également possible de considérer d'autres opérations comme :

- les amplifications et les contractions [12]
- les permutations de deux lettres successives ou de plusieurs lettres [11]
- les inversions de lettres.

Nous prendrons en compte dans cette partie uniquement un modèle simple basé sur les *substitutions* et sur les *indels*.

EXEMPLE 1 : *substitutions / indels*

`AGACTTA` → `AGAGTTA`	substitution de la quatrième lettre C par un G
`AGACTTA` → `AGAGTCTTA`	insertion des lettres GT en quatrième position
`AGACTTA` → `AGATTA`	suppression de la lettre C de la quatrième position

2.1.2 Distances

Nous rappelons dans cette partie la définition de distance, et considérons un certain nombre de définitions de distance pour la comparaison de séquences.

Une fonction $d : \Sigma^* \times \Sigma^* \to \mathbb{R}$ est qualifiée de *distance* sur Σ si les propriétés suivantes sont vérifiées pour tout $u, v \in \Sigma^*$.

Positivité : $d(u, v) \geq 0$

Identité : $u = v \iff d(u, v) = 0$

Symétrie : $d(u, v) = d(v, u)$

Inégalité triangulaire : $\forall w \in \Sigma^* \quad d(u, v) \leq d(u, w) + d(w, v)$

Nous donnons ici quelques exemples de distances issues de [33].

Parmi les distances originales, citons notamment la distance **préfixe**, **suffixe**, et **facteur** définies comme $d(u, v) = |u| + |v| - 2 \times |L(u, v)|$ où $L(u, v)$ désigne respectivement le plus long préfixe commun, suffixe commun, ou facteur commun à u et v.

Parmi les distances classiques, utilisées de manière plus fréquente, on considère généralement deux distances, celle dite de *Hamming* et celle dite de *Levenstein* (ou d'édition) ainsi que leurs extensions décrites ci-après.

2.1.2.1 Distance de Hamming

La **distance de Hamming** $d_{\mathcal{H}}$ est définie sur deux mots de même longueur. Elle correspond au nombre de positions pour lesquelles les caractères de deux mots diffèrent. Nous n'autorisons

et ne considérons donc pour la distance de Hamming que les opérations de type *substitution*

EXEMPLE 2 : *distance de Hamming*

$d_{\mathcal{H}}(\texttt{AGACTTA}, \texttt{AGGCTCA})$	$= 2$	substitution du second `A` par `G` et substitution du dernier `T` par `C`.
$d_{\mathcal{H}}(\texttt{AGACTTA}, \texttt{AGAGTTA})$	$= 1$	substitution de `C` par `G`.

2.1.2.2 Distance de Levenstein

La **distance de Levenstein** $d_{\mathcal{L}}$ est définie sur deux mots comme étant le nombre minimal d'opérations choisies parmi les *substitutions* et les *indels* pour transformer le premier mot en le second mot.

EXEMPLE 3 : *distance de Levenstein*

$d_{\mathcal{L}}(\texttt{AGACTTA}, \texttt{AGATTT})$	$= 2$	suppression de la lettre `C` et substitution de la dernière lettre `A` par `T`.
$d_{\mathcal{L}}(\texttt{AGACTTA}, \texttt{AGAGTTA})$	$= 1$	substitution de la lettre `C` par `G`.

2.1.2.3 Extensions de la distance de Levenstein

Une première extension (appelée *distance généralisée de Levenstein*) consiste à associer un coût à chacune des opérations unitaires. Les opérations de substitution, d'insertion, et de suppression (délétion) possèdent chacune un coût (dépendant des lettres/ de la lettre associée)

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

A la fonction $Sub(a, b)$ est généralement associée une matrice de substitution : cette matrice est symétrique et possède des 0 sur sa diagonale, les autres éléments de la matrice sont positifs (et doivent vérifier la propriété de distance sur Σ, en particulier *l'inégalité triangulaire*). Les coûts $Del(a)$ et $Ins(a)$ doivent être strictement positifs pour tout $a \in \Sigma$.

Voici un exemple de matrice :

EXEMPLE 4 : *matrice de distance de Levenstein généralisée sur l'alphabet* $\Sigma = \{\texttt{A}, \texttt{T}, \texttt{G}, \texttt{C}\}$

$S(a,b)$	A	T	G	C
A	0	2	1	2
T	2	0	2	1
G	1	2	0	2
C	2	1	2	0

Par exemple, $S(\texttt{A}, \texttt{T}) = 2$ signifie que la substitution de la lettre `A` par la lettre `T` donne un coût de 2.

Le coût d'une suite d'opérations unitaires est égal à la somme des coûts unitaires des opérations associées. La distance de Levenstein généralisée $d_{\mathcal{L}_g}(u, v)$ $(u, v \in \Sigma^*)$ est alors définie

comme le *coût minimal* d'un ensemble d'opérations unitaires qui transforme u en v.

EXEMPLE 5 : *distance de Levenstein Généralisée*

Les coûts des substitutions $S(a,b)$ sont donnés par la matrice de l'exemple précédent. Nous y associons les coûts $Del(a) = 3$ et $Ins(a) = 3$ pour tout $a \in \Sigma$

$d_{\mathcal{L}_g}(\texttt{AGACTTA}, \texttt{AGATTT})$	$= 4$	suppression de la dernière lettre `A` ($Del(\texttt{A}) = 3$) et substitution de la lettre `C` par `T` ($Sub(\texttt{C}, \texttt{T}) = 1$).
$d_{\mathcal{L}_g}(\texttt{AGACTTA}, \texttt{AGATTTA})$	$= 1$	substitution de la lettre `C` par `T` ($Sub(\texttt{C}, \texttt{T}) = 1$).

2.1.3 Alignements

Afin de représenter l'ensemble des opérations permettant de transformer un mot u en un mot v, nous pouvons donner une liste (arbitrairement ordonnée) des opérations unitaires associées.

EXEMPLE 6 : *liste d'opérations associées à la transformation* `TCCAGACTTA` → `TCGCAGATTG`

`TCCAGACTTA` → `TCCAGACTTG`	substitution du dernier `A` par `G`
`TCCAGACTTG` → `TCCAGATTG`	suppression d'un `C`
`TCCAGACTTG` → `TCGCAGATTG`	insertion d'un `G`

Cependant la représentation la plus couramment adoptée lorsque la distance est de type Levenstein généralisée repose sur le principe d'**alignement**. A chacune des opérations unitaires est associée une représentation sur un alphabet $\mathcal{A}$ des alignements. L'alphabet des alignements est lui-même défini comme produit cartésien sur l'alphabet $\Sigma \cup \{\text{-}\}$: $\mathcal{A} = (\Sigma \cup \{\text{-}\}) \times (\Sigma \cup \{\text{-}\})$. Le caractère - désigne ici le mot vide. Sur l'alphabet des alignements, les opérations unitaires sont représentées de la manière suivante :

- une substitution de la lettre a par b ($a, b \in \Sigma$) est notée par le tuple $\binom{a}{b}$.
- une insertion de la lettre $a \in \Sigma$ est notée par le tuple $\binom{\text{-}}{a}$,
- une suppression de la lettre $a \in \Sigma$ est notée par le tuple $\binom{a}{\text{-}}$.

Si nous ordonnons l'ensemble des opérations unitaires permettant de transformer un mot u en un mot v (tri selon la position où l'opération est appliquée), alors cet ensemble forme un mot sur $\mathcal{A}^*$ associé à la transformation. Il s'agit de sa représentation sous forme d'*alignement*.

EXEMPLE 7 : *alignement associé à la transformation* `TCCAGACTTA` → `TCGCAGATTG`

$$\binom{\texttt{T}}{\texttt{T}}\binom{\texttt{C}}{\texttt{C}}\binom{-}{\texttt{G}}\binom{\texttt{C}}{\texttt{C}}\binom{\texttt{A}}{\texttt{A}}\binom{\texttt{G}}{\texttt{G}}\binom{\texttt{A}}{\texttt{A}}\binom{\texttt{C}}{-}\binom{\texttt{T}}{\texttt{T}}\binom{\texttt{T}}{\texttt{T}}\binom{\texttt{A}}{\texttt{G}}$$

plus simplement écrit

$$\binom{\texttt{TC-CAGACTTA}}{\texttt{TCGCAGA-TTG}}$$

2.1.4 Calcul de la distance de Levenstein généralisée

Nous avons défini un certain nombres de distances, en particulier la distance de Levenstein généralisée. Nous avons également considéré des méthodes de visualisation des opérations élémentaires d'édition.

Nous allons voir ici la méthode utilisée pour calculer la distance de Levenstein généralisée, minimisant le coût de la suite d'opérations d'édition. Nous utiliserons pour cela une structure de données afin de représenter les opérations d'éditions : le graphe d'édition.

2.1.4.1 Graphe d'édition

Le **graphe d'édition** [33] $G(u,v) =< S, A >$ de deux mots u et v est une structure de données permettant de recenser les opérations d'édition compatibles pour transformer u en v, et, plus généralement un préfixe de u et un préfixe de v.

L'ensemble des sommets S est construit à partir du produit cartésien de l'ensemble des préfixes de u et v : $S = u[1..i] \times v[1..j]$ pour tout $i \in [0..|u|], j \in [0..|v|]$ ($i = 0$ ou $j = 0$ correspondront respectivement au mot vide). L'ensemble des arêtes A est quant à lui construit entre deux sommets selon les opérations unitaires compatibles permettant d'atteindre un sommet $s' \in S$ à partir d'un sommet $s \in S$.

Cet ensemble d'arêtes compatibles pour un sommet $s' =< i, j >$ (sommet associé à la paire $u[1..i] \times v[1..j]$) est le suivant :

- si $i > 0$, alors une arête associée à une suppression de la lettre $u[i]$ relie le sommet $s =< i-1, j >$ vers le sommet $s' =< i, j >$.
- si $j > 0$, alors une arête associée à l'insertion de la lettre $v[j]$ à la position j relie le sommet $s =< i, j-1 >$ vers le sommet $s' =< i, j >$.
- si $i > 0$ et $j > 0$, alors une arête associée à la substitution de $u[i]$ par $v[j]$ relie $s =< i-1, j-1 >$ vers le sommet $s' =< i, j >$.

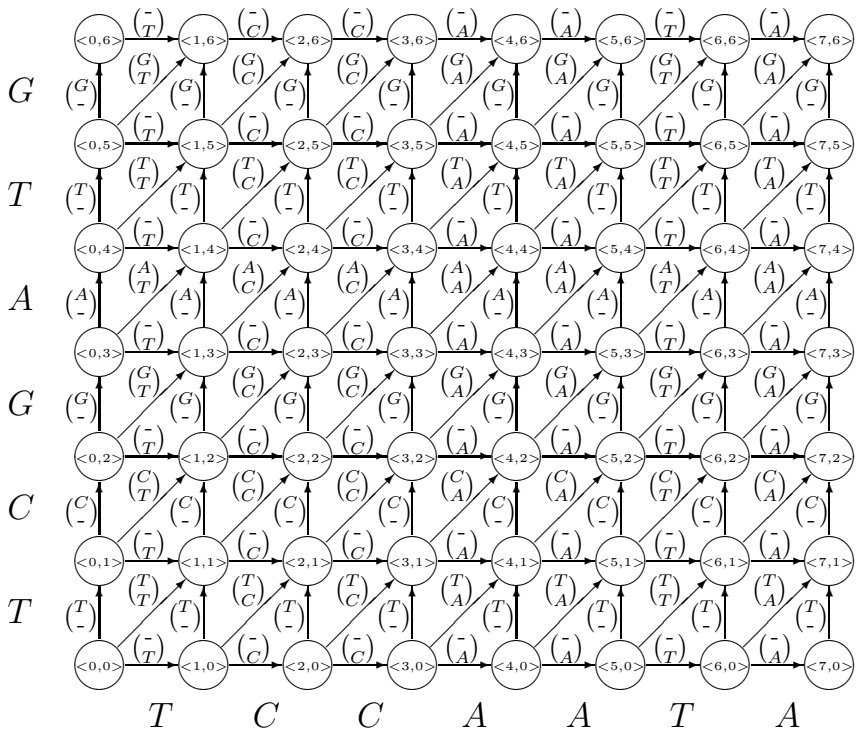


FIG. 2.1 – *Graphe d'édition des mots* $u =$ `TCCAATA` *et* $v =$ `TCGATG`

2.1.4.2 Récurrence sur le graphe d'édition

Sur le graphe d'édition $G(u,v) =< S, A >$, nous considérons l'ensemble des chemins $C(i,j)$ partant du sommet $< 0,0 >$ vers le sommet $< i,j >$. Pour un chemin donné $c \in C(i,j)$, son ensemble d'arêtes possède un coût qui correspond au coût de l'ensemble des opérations d'édition associées à ces arêtes.

Afin de calculer la distance de Levenstein, nous devons trouver le chemin $c' \in C(|u|,|v|)$ dont le coût est minimal. Or, par décomposition, ce chemin *minimal* ne dépend que des chemins préfixes $c \in C(i,j)$ $(i < |u|, j < |v|)$ dont le coût est également minimal.

Il est donc possible d'établir une relation de récurrence entre le coût $T(i,j)$ d'un chemin minimal $c \in C(i,j)$ et le coût des chemins minimaux qui lui sont préfixes : en particulier, d'après le graphe d'édition, la relation pour $T(i,j)$ peut être obtenue en fonction de $T(i-1,j-1)$, $T(i,j-1)$ et $T(i-1,j)$ comme :

$$T(i,j) \quad = \quad min \begin{cases} T(i-1,j-1) + Sub(u[i],v[j]) & \text{si } 0 < i \leq |u| \text{ et } 0 < j \leq |v| \\ T(i,j-1) + Del(u[i]) & \text{si } 0 \leq i \leq |u| \text{ et } 0 < j \leq |v| \\ T(i-1,j) + Ins(v[j]) & \text{si } 0 < i \leq |u| \text{ et } 0 \leq j \leq |v| \end{cases}$$

L'algorithme associé calcule la récurrence en deux étapes en utilisant une matrice T de dimensions $(|u|+1) \times (|v|+1)$:

- la première étape consiste à initialiser les bords de la matrice, c'est-à- dire affecter les valeurs associées aux éléments $T[i,0]$ pour $i \in [0..|u|]$ et aux éléments $T[0,j]$ pour $j \in [0..|v|]$.
- la deuxième étape (boucle interne de l'algorithme 1 calcule la récurrence pour finalement donner la distance d'édition entre les mots u et v (valeur $T(|u|,|v|)$).

Algorithme 1 : *Calcul de la distance de Levenstein*

Entrées : deux mots u et v sur un alphabet Σ,
une fonction de substitution $Sub : \Sigma^2 \to \mathbb{R}$,
deux fonctions d'indels $Ins : \Sigma \to \mathbb{R}$, $Del : \Sigma \to \mathbb{R}$
Sorties : la distance de Levenstein entre u et v
Données : une matrice T de dimension $(|u|+1) \times (|v|+1)$
$T[0,0] \leftarrow 0$;
pour i **de** 1 **à** $|u|$ **faire**
  $T[i,0] \leftarrow T[i-1,0] + Del(u[i])$;
pour j **de** 1 **à** $|v|$ **faire**
  $T[0,j] \leftarrow T[0,j-1] + Ins(v[j])$;
  pour i **de** 1 **à** $|u|$ **faire**

$$T[i,j] = min \begin{pmatrix} T[i-1,j-1] + Sub(u[i],v[j]), \\ T[i,j-1] + Del(u[i]), \\ T[i-1,j] + Ins(v[j]) \end{pmatrix}$$

retourner $T[|u|,|v|]$;

2.1.5 Calcul de l'alignement associé

Nous avons vu comment calculer la distance de Levenstein, nous allons maintenant considérer l'alignement associé. Il est obtenu en suivant les arêtes activées par le chemin de coût minimal

dans le graphe d'édition.

Une arête entre deux sommets $< i, j >$ et $< i', j' >$ est *activée* par un chemin minimal $c \in C(i, j)$ $(i \leq |u|, j \leq |v|)$ si et seulement si, dans la matrice T, la correspondance $T[i, j]$,$T[i', j']$ est minimale en score *i.e.* si :

- $i = i' + 1$ et $j = j' + 1$ implique $T[i, j] = Sub(u[i], v[j]) + T[i', j']$
- $i = i'$ et $j = j' + 1$ implique $T[i, j] = Ins(v[j]) + T[i', j']$
- $i = i' + 1$ et $j = j'$ implique $T[i, j] = Del(u[i]) + T[i', j']$

Pour trouver les arêtes qui sont activées par le chemin minimal $\in C(|u|, |v|)$, il suffit de parcourir les arêtes actives *en sens inverse* depuis le *puits* associé $< |u|, |v| >$ jusqu'à la *source* $< 0, 0 >$ (voir la figure 2.2). Nous obtenons ainsi les opérations d'éditions associées à la distance de Levenstein généralisée.

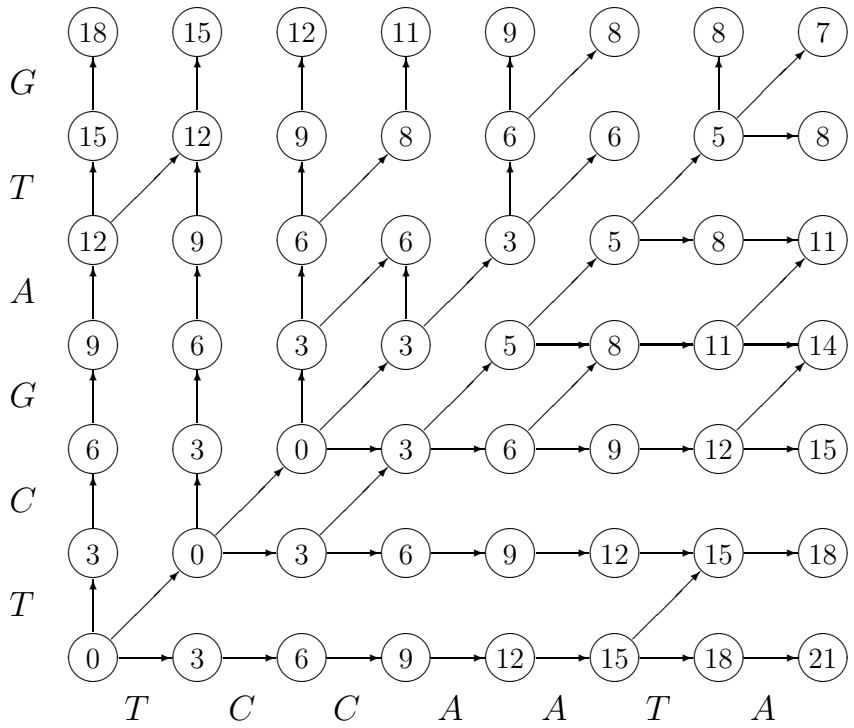


FIG. 2.2 – *Arêtes actives du graphe d'édition des mots* $u =$ `TCCAATA` *et* $v =$ `TCGATG`. *Les valeurs de la matrice de programmation dynamique* $T[i, j]$ *sont indiquées pour chaque sommet du graphe d'édition. Les chemins minimaux entre le sommet source* $< 0, 0 >$ *et le sommet puits* $< 6, 7 >$ *donnent un coût de* 7. *Les alignements associés à ces chemins sont représentés dans la figure 2.3*

FIG. 2.3 – *Alignements associés aux chemins minimaux du graphe d'édition de la figure 2.2. Ces deux alignements correspondent aux opérations d'éditions obtenus lors du parcours d'un des deux chemins minimaux entre le sommet source* $< 0, 0 >$ *et le sommet puits* $< 6, 7 >$

$$\begin{pmatrix} \texttt{TC-GATG} \\ \texttt{TCCAATA} \end{pmatrix} \qquad \begin{pmatrix} \texttt{T-CGATG} \\ \texttt{TCCAATA} \end{pmatrix}$$

2.1.6 Fonction d'indel affine

Il est en pratique fréquent qu'une insertion/une suppression ne soit pas uniquement d'une lettre mais de plusieurs lettres. Plutôt que de pénaliser chaque indel unitaire avec un coût, il est plus intéressant d'associer une fonction de coût $\lambda(l)$ selon la longueur l de l'indel créé. Cette fonction peut permettre, par exemple, de pénaliser fortement la création d'un indel, mais de moins pénaliser son extension. Parmi les fonctions possibles, nous considérons ici les fonctions affines car elles possèdent de bonnes propriétés. En particulier il existe une extension de l'algorithme de programmation dynamique pour la distance de Levenstein qui permet de réaliser le calcul des indels avec des fonctions affines.

Soit $\lambda(l)$ la fonction de coût associée aux indels :

$$\lambda(l) = d + e \cdot (l-1)$$

avec $d, e \in \mathbb{R}$, constantes associées respectivement à l'ouverture/l'extension d'un indel (en général $d > e$). Afin de mettre à jour un élément i, j de la matrice T, nous devons nous souvenir du coût minimal de toutes les insertions et suppressions amonts terminant en i, j. Il est cependant possible de calculer ces deux quantités à l'aide de deux autres matrices de programmation dynamiques I et D de mêmes dimensions que T : les trois matrices T, I, D seront mutuellement dépendantes durant le calcul. $I[i,j]$ désigne le coût minimal d'un alignement dont la dernière insertion se termine en i, j. Réciproquement, $D[i,j]$ désigne le coût minimal d'un alignement dont la dernière suppression se termine en i, j.

La relation de récurrence complétée est alors donnée par :

$$T(i,j) = min \begin{cases} T(i-1,j-1) + Sub(u[i], v[j]) & \text{si } 0 < i \leq |u| \text{ et } 0 < j \leq |v| \\ I(i-1,j-1) + Sub(u[i], v[j]) & \text{si } 0 < i \leq |u| \text{ et } 0 < j \leq |v| \\ D(i-1,j-1) + Sub(u[i], v[j]) & \text{si } 0 < i \leq |u| \text{ et } 0 < j \leq |v| \end{cases} \tag{2.1}$$

$$I(i,j) = min \begin{cases} T(i-1,j) + d & \text{si } 0 < i \leq |u| \text{ et } 0 \leq j \leq |v| \\ I(i-1,j) + e & \text{si } 0 < i \leq |u| \text{ et } 0 \leq j \leq |v| \end{cases} \tag{2.2}$$

$$D(i,j) = min \begin{cases} T(i,j-1) + d & \text{si } 0 \leq i \leq |u| \text{ et } 0 < j \leq |v| \\ D(i,j-1) + e & \text{si } 0 \leq i \leq |u| \text{ et } 0 < j \leq |v| \end{cases} \tag{2.3}$$

En pratique il est assez fréquent de «fusionner» les matrices D et I. Nous définissons alors une matrice E de mêmes dimensions que T : $E[i,j]$ correspond alors logiquement au coût de l'alignement dont le dernier indel se termine en i, j.

$$T(i,j) = min \begin{cases} T(i-1,j-1) + Sub(u[i], v[j]) & \text{si } 0 < i \leq |u| \text{ et } 0 < j \leq |v| \\ E(i-1,j-1) + Sub(u[i], v[j]) & \text{si } 0 < i \leq |u| \text{ et } 0 < j \leq |v| \end{cases} \tag{2.4}$$

$$E(i,j) = min \begin{cases} T(i-1,j) + d & \text{si } 0 < i \leq |u| \text{ et } 0 \leq j \leq |v| \\ E(i-1,j) + e & \text{si } 0 < i \leq |u| \text{ et } 0 \leq j \leq |v| \\ T(i,j-1) + d & \text{si } 0 \leq i \leq |u| \text{ et } 0 < j \leq |v| \\ E(i,j-1) + e & \text{si } 0 \leq i \leq |u| \text{ et } 0 < j \leq |v| \end{cases} \tag{2.5}$$

Cette définition n'est en théorie valable que si $max_{a,b \in \Sigma^2} Sub(a,b) \leq 2 \cdot e$: en effet, la définition donnée dans les équations (2.1–2.3) n'autorise pas deux suites d'opérations d'insertion

et de suppression sur deux positions successives. Or, la définition des équations (2.4–2.5) le permet, ce qui implique que, lorsque $max_{a,b\in\Sigma^2} Sub(a,b) > 2 \cdot e$, le coût d'un indel composé d'une succession d'insertions et de suppressions juxtaposées peut alors être moins élevé que le coût des substitutions.

En pratique, même si cette condition n'est pas respectée, le résultat obtenu est très rarement différent de celui donné par la programmation dynamique complète. Cette solution a été par exemple retenue dans le logiciel YASS [88] lors du calcul des alignements. Le logiciel YASS est présenté à la partie 4.6.1.

2.1.7 La représentation Dot-plot

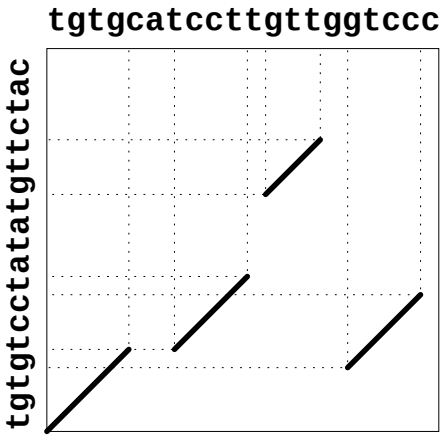


FIG. 2.4 – *Principe du dot-plot*

Le **dot-plot** est une représentation souvent sous-estimée pour visualiser les similarités entre deux séquences, ou sur une même séquence. La représentation sous forme d'alignement représentait les similarités en terme de *séquence*. Par opposition, un dot-plot est une représentation des similarités en terme de *positions*. Il s'agit d'une représentation planaire indiquant les positions relatives des similarités sur deux séquences et en précisant éventuellement le niveau de ressemblance. Les deux séquences u et v à comparer sont positionnées respectivement sur l'axe des x/ l'axe des y du plan. Lorsque deux fragments aux positions i et j sur les séquences u et v sont jugés similaires selon un critère donné, alors un point (dont l'intensité peut varier selon le niveau de similarité) est placé sur le plan.

Ainsi une répétition relativement longue, donc composée de fragments juxtaposés successifs, sera représentée par un trait diagonal dû à la succession de points. La figure 2.4 illustre le concept du dot-plot : les fragments communs à deux séquences dont la longueur est de 3 lettres sont représentés dans le plan, donnant ainsi un aperçu de la distribution des similarités communes. Il est également possible de représenter le brin complémentaire inversé en superposition, ce qui permet de visualiser pour une séquence d'ADN, les répétitions complémentaires inversées et directes sur le même dot-plot. La figure 2.5 est une illustration et ses exemples ont été réalisés sur l'interface Web de YASS (voir la partie 4.6.1).

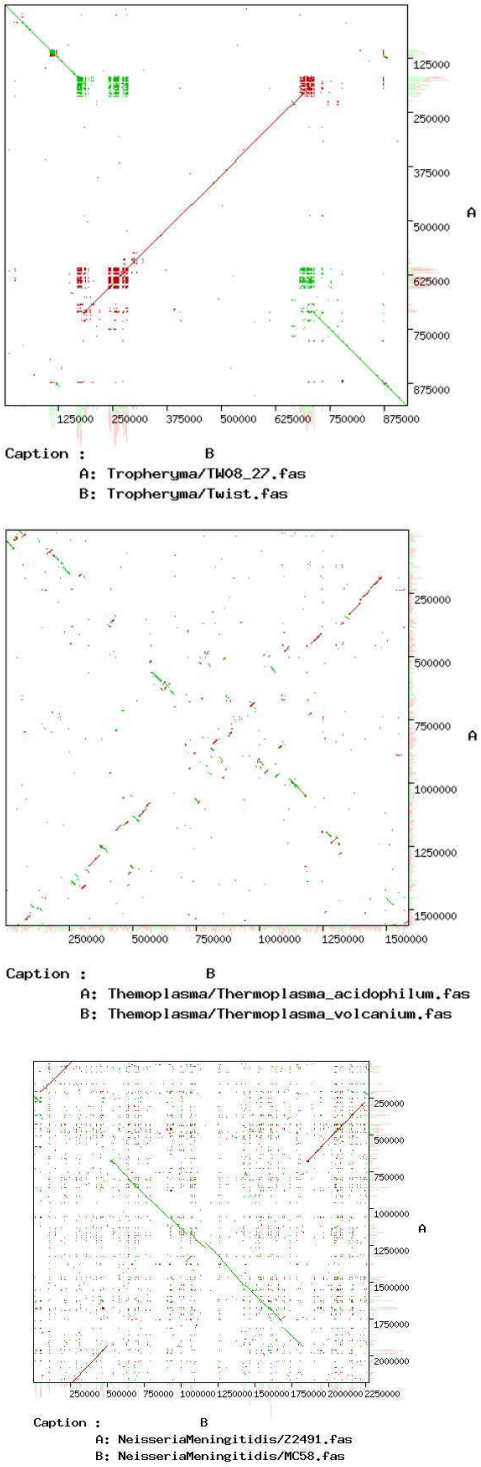


FIG. 2.5 – *Exemple de dot-plots : le premier dot-plot montre un répétition inversée au centre lors de la comparaison de deux souches de la bactérie* Tropheryma, *Les deux autres dot-plots laissent apparaître des* inversions *et* translocations *plus complexes lors de comparaisons respectives de souches de bactéries* Thermoplasma *et* Neisseria.

2.2 Automates

Nous avons vu dans la partie précédente des principes de l'algorithmique du texte. Dans le cadre du traitement des séquences, la section suivante est consacrée à la théorie des automates.

Les **automates** sont des machines à états finis qui, sur la lecture d'une entrée (d'un mot) sur un alphabet Σ, se déplacent d'état en état selon des transitions associées à chaque état. Nous nous intéresserons principalement dans cette partie aux automates finis déterministes, ceux pour lesquels il existe au plus une transition associée à chaque état pour chaque lettre de l'alphabet Σ.

Après avoir énoncé la définition d'un automate fini déterministe (partie 2.2.1), ainsi que sa représentation sous forme de graphe (partie 2.2.1.2), nous donnerons le théorème de minimalité afin d'expliquer la raison théorique d'une minimisation unique. Nous présenterons deux algorithmes pour la minimisation d'automates, celui proposé par Moore (partie 2.2.3.1) ainsi celui de Hopcroft (partie 2.2.3.2). Enfin nous considérerons un automate particulier, l'automate de Aho-Corasick (partie 2.2.4), qui permet de rechercher un ensemble de mots donnés sur un texte. Cet automate sera utilisé dans la partie 3.7.7.1.

Les algorithmes donnés dans cette partie sont basés sur ceux publiés dans [35, 13]. Certaines parties sont également basées sur [52].

2.2.1 Automates finis déterministes

2.2.1.1 Définition

Un automate fini déterministe A est défini par 5 éléments :

- un ensemble fini d'états $Q = \{q_0, q_1, \ldots\}$,
- un alphabet d'entrée Σ,
- un ensemble de transitions défini par la fonction de transition $\psi : Q \times \Sigma \rightarrow Q$,
- un *état initial* $q_0 \in Q$,
- un sous-ensemble d'états $Q_F \subset Q$ nommés *états finaux*.

Nous noterons, pour un automate A, ses éléments sous la forme $A =< Q, \Sigma, q_0, Q_F, \psi >$.

Notons qu'un automate fini peut également être non-déterministe : il suffit de modifier la définition de la fonction ψ. ψ est alors définie comme $\psi : Q \times \Sigma \rightarrow 2^Q$. Dans ce cas, pour un état $q \in Q$ donné et une lettre $a \in \Sigma$, la fonction de transition ne renvoie plus comme résultat un seul état, mais un sous-ensemble de Q.

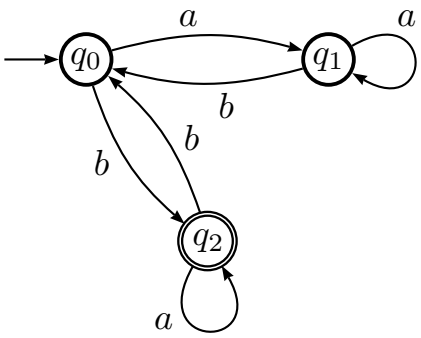
2.2.1.2 Diagramme de transitions

Le **diagramme de transitions** associé à l'automate A est un graphe orienté, dont les nœuds représentent les états Q de l'automate A, et les arêtes représentent les transitions et sont libellées par les lettres associées aux transitions.

Pour un état $q \in Q$, et un mot $u \in \Sigma^*$, nous noterons $\psi(q, u)$ l'état atteint à partir de l'état

q en suivant la séquence de transitions correspondant aux symboles de u.

EXEMPLE 8 : *automate A et diagramme de transitions associé*



Nous définissons l'automate $A =< Q, \Sigma, q_0, Q_F, \psi >$ avec les ensembles suivants et la fonction de transition suivante :
- $Q = \{q_0, q_1, q_2\}$,
- $\Sigma = \{a, b\}$,
- $\psi : (q_0, a) = q_1, (q_0, b) = q_2, (q_1, a) = q_1, (q_1, b) = q_0, (q_2, a) = q_2, (q_2, b) = q_0$,
- q_0,
- $Q_F = \{q_2\}$,

Le diagramme de transitions associé à l'automate A est donné ci-dessus.

2.2.1.3 Propriétés

Nous donnons dans cette partie quelques définitions et propriétés des automates.

Définition 1 (mots acceptés par l'automate A) *L'automate* accepte *un mot $u \in \Sigma^*$ si il existe une séquence de transitions correspondant aux symboles de u, de l'état initial q_0 jusqu'à un état final $q \in Q_F$ ($\psi(q_0, u) \in Q_F$).*

EXEMPLE 9 : *mots acceptés par l'automate A*

Dans l'exemple 8, les mots *aabb*,*b*,*ba*,*baa* sont acceptés par A, alors que les mots *a*,*aa*,*aab*,*ab* ne le sont pas.

Définition 2 (langage L reconnu par l'automate A) *Le langage L* reconnu *par un automate est constitué de l'ensemble des mots x qui sont acceptés par l'automate A. Ce langage peut éventuellement être infini. Ce langage est dit* **régulier**

EXEMPLE 10 : *langage L reconnu par l'automate A*

Dans l'exemple 8, le langage L équivalent à l'expression régulière $(a^+b|ba^*b)^*ba^*$ est reconnu par l'automate A.

Définition 3 (automate total/complet) *Un automate A est dit* **total** *ou* **complet** *si pour tout état $q \in Q$ et tout $a \in \Sigma$ il existe un état $q' \in Q$ tel que $\psi(q, a) = q'$.*

Définition 4 (états récurrents/transients) *Un état $q \in Q$ est dit* **récurrent** *si il existe un mot non-vide $u \in \Sigma^+$ tel que $\psi(q,u) = q$. Autrement cet état est dit* **transient**.

EXEMPLE 11 : *propriétés de l'automate A*
Dans l'exemple 8, l'automate A est *complet* sur Σ, et tous les états de A sont récurrents.

2.2.2 Automate minimal

Une propriété intéressante sur les automates, est qu'il est souvent possible de réduire leur taille (leur nombre d'états) tout en reconnaissant le même langage L que l'automate initial. Cette propriété appelée *minimisation* est très intéressante non seulement du point de vue de la compacité, mais également lorsque des calculs doivent être réalisés sur le langage régulier associé L. En effet, il est alors préférable de réaliser les calculs sur l'automate minimisé.

Théorème 1 (automate minimal) *Parmi tous les automates reconnaissant un langage régulier L donné, il existe un* **automate minimal** *en nombre d'états [78], et ce dernier est unique à un isomorphisme près (i.e., à un renommage des états).*

Propriété 1 (états équivalents et automate minimal) *Afin d'obtenir l'automate minimal, il faut pouvoir trouver les états dits équivalent. Deux états p et q ($p \in Q$, $q \in Q$) sont* équivalents *(*équivalents *au sens de Nérode) si*

$$\forall u \in \Sigma^* \quad \psi(p,u) \in Q_F \iff \psi(q,u) \in Q_F$$

On notera cette relation $p \equiv q$.

Cette **relation d'équivalence** $\equiv$ *est* régulière à droite *: $p \equiv q \implies \forall a \in \Sigma \quad \psi(p,a) \equiv \psi(q,a)$ La relation est calculée de manière récursive comme limite de la suite décroissante $E_{r(0)} \supset E_{r(1)} \supset \cdots \supset E_{r(i)} \supset E_{r(i+1)}$ avec*

$$(p,q) \in E_{r(0)} \text{ ssi } p \in Q_F \iff q \in Q_F$$

et pour $i > 0$

$$(p,q) \in E_{r(i+1)} \text{ ssi } (p,q) \in E_{r(i)} \text{ et } \forall a \in \Sigma \quad \big(\psi(p,a), \psi(q,a)\big) \in E_{r(i)}$$

Si pour une certaine valeur i, $E_{r(i)} = E_{r(i+1)}$, alors $\forall j > i \quad E_{r(j)} = E_{r(i)} \triangleq \equiv$.

L'automate minimal A_m *de A est composé d'un ensemble d'états correspondant au nombre de classes de la relation d'équivalence $\equiv$ des états de A. L'état initial correspond alors à la classe qui contient l'état initial de A et les états finaux sont ceux associés à des classes des états finaux de A.*

2.2.3 Algorithmes de minimisation d'automates

La plupart des algorithmes de minimisation d'automates procèdent par raffinements successifs des classes d'équivalences jusqu'à l'obtention d'une partition stable d'états. Les plus courants sont quadratiques en la taille de l'automate initial, comme l'algorithme de Moore [13] présenté dans l'algorithme 2. Il est cependant possible d'obtenir, à l'aide d'un algorithme proposé par Hopcroft en 1971 [13], une minimisation d'automate faite en temps $\mathcal{O}(|\Sigma| \cdot n \cdot log(n))$ dans le pire cas pour un automate de taille n présenté dans l'algorithme 3.

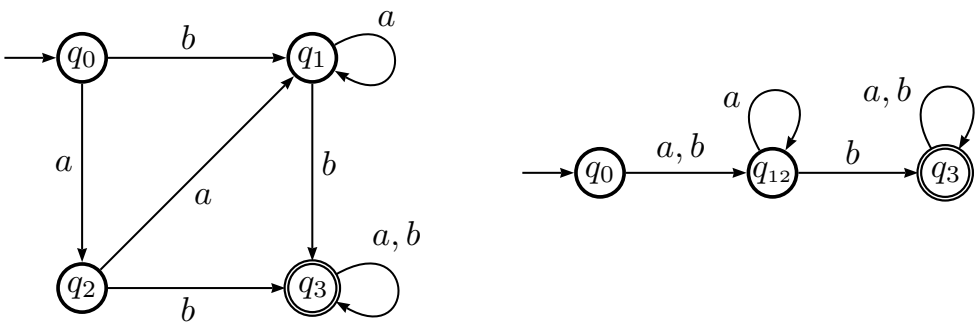
2.2.3.1 Algorithme de Moore

L'algorithme de minimisation de Moore [80] procède à un partitionnement successif $\mathcal{P}_f$ des classes d'équivalences de l'automate $A =< Q, \Sigma, q_0, Q_F, \psi >$ jusqu'à l'obtention d'une partition stable des états $\mathcal{P}_e$. On défini de manière naturelle la relation d'équivalence $p \equiv_{\mathcal{P}} q$ entre deux états p et q selon une partition d'états $\mathcal{P}$ si p et q sont dans la même partie de $\mathcal{P}$.

Algorithme 2 : *Minimisation de Moore*

Entrées : un automate $< Q, \Sigma, q_0, Q_F, \psi >$,
Sorties : une partition stable des états $\mathcal{P}_e$ correspondant aux états de l'automate minimal
Données : Partitions des états $\mathcal{P}_e, \mathcal{P}_f$
$\mathcal{P}_f \leftarrow \{Q_F, Q \backslash Q_F\}$;
répéter
  // $\mathcal{P}_e$ *est l'ancienne partition*;
  $\mathcal{P}_e \leftarrow \mathcal{P}_f$;
  // *raffiner la partition* $\mathcal{P}_f$ *avec* $\psi^{-1}(\mathcal{P}_f, a)$;
  pour $a \in \Sigma$ **faire**
    $\mathcal{P}_g \leftarrow \psi^{-1}(\mathcal{P}_f, a)$;
    $\mathcal{P}_f \leftarrow$ INTERSECTION$(\mathcal{P}_f, \mathcal{P}_g)$;
jusqu'à $\mathcal{P}_e = \mathcal{P}_f$;

EXEMPLE 12 : *minimisation d'automate à l'aide de l'algorithme de Moore*



Pour l'automate de gauche, la partition initiale des états de l'automate est $\mathcal{P}_{f_0} = \{Q \backslash Q_F, Q_F\} = \{\{q_0, q_1, q_2\}, \{q_3\}\}$. La partition $\psi^{-1}(\mathcal{P}_{f_0}, a)$ est égale à $\{\{q_0, q_1, q_2\}, \{q_3\}\}$ ce qui indique que la partition initiale $\mathcal{P}_{f_0}$ est stable pour a . En revanche, la partition $\psi^{-1}(\mathcal{P}_{f_0}, b)$ est égale à $\{\{q_0\}, \{q_1, q_2, q_3\}\}$, ce qui affine $\mathcal{P}_{f_0}$ en $\mathcal{P}_{f_1} = \{\{q_0\}, \{q_1, q_2\}, \{q_3\}\}$.
Nous constatons alors que cette partition est stable pour a et pour b (Les partitions $\psi^{-1}(\mathcal{P}_{f_1}, a)$ et $\psi^{-1}(\mathcal{P}_{f_1}, b)$ obtenues sont alors respectivement égales à $\{\{q_0, q_1, q_2\}, \{q_3\}\}$ et $\{\{q_0\}, \{q_1, q_2, q_3\}\}$). On peut donc conclure que l'automate minimisé correspond à la partition $\mathcal{P}_{f_0} = \{\{q_0\}, \{q_1, q_2\}, \{q_3\}\}$. Nous pouvons alors le représenter en fusionnant les états $\{q_1, q_2\}$ (automate de droite).

L'algorithme est basé sur le raffinement suivant : la fonction de raffinement $\psi^{-1}(\mathcal{P}_f, a)$ cor-

respond à la relation $p \equiv_{\psi^{-1}(\mathcal{P}_f,a)} q \iff \psi(p,a) \equiv_{\mathcal{P}_f} \psi(q,a)$: deux états p et q sont donc dans la même classe d'équivalence selon $\equiv_{\psi^{-1}(\mathcal{P}_f,a)}$, si et seulement si leurs successeurs selon a $\psi(p,a)$ et $\psi(q,a)$ étaient dans la même partie de $\mathcal{P}_f$.

La fonction INTERSECTION réalisant l'intersection de deux partitions peut être calculée en temps $\mathcal{O}(n)$. L'algorithme de Moore est exécuté en temps $\mathcal{O}(|\Sigma| \cdot n^2)$.

2.2.3.2 Algorithme de Hopcroft

L'idée principale de l'algorithme de **minimisation de Hopcroft** [51] consiste à raffiner la partition, non plus par une intersection avec une autre partition mais uniquement avec un seul sous-ensemble, que l'on choisira volontairement «petit». On suppose que l'automate est *complet* (voir la définition 3).

Algorithme 3 : *Minimisation de Hopcroft*

MINIMISATION(A)

Entrées : un automate $A =< Q, \Sigma, q_0, Q_F, \psi >$
Sorties : une partition stable des états $\mathcal{P}_e$ correspondant aux états de l'automate minimal
Données : Partitions des états $\mathcal{P}_e$, Ensembles d'états B, B', B'', C, P, Pile S d'éléments de $2^Q \times \Sigma$

$\mathcal{P}_e \leftarrow (Q_F, Q \backslash Q_F)$;
$C \leftarrow$ le plus petit ensemble entre Q_F et $Q \backslash Q_F$;
$S \leftarrow \varnothing$;
pour $a \in \Sigma$ **faire**
  AJOUTER$((C,a),S)$;
tant que $S \neq \varnothing$ **faire**
  $(P,a) \leftarrow$ RETIRETETE(S);
  pour $B \in \mathcal{P}_e$ *tel que* B *est raffiné par* (P,a) **faire**
    $B', B'' \leftarrow$ RAFFINER(B,P,a);
    // Séparer B *en* B' *et* B'' *dans la partition* $\mathcal{P}_e$;
    SEPARER$(B,B',B'',\mathcal{P}_e)$;
    MISEAJOUR(B,B',B'',S);
retourner $\mathcal{P}_e$;

MISEAJOUR(B,B',B'',S)

$C \leftarrow$ le plus petit ensemble entre B' et B'';
pour $b \in \Sigma$ **faire**
  si $(B,b) \in S$ **alors**
    REMPLACER$((B,b),S,(B',b),(B'',b))$;
  sinon
    AJOUTER$((C,a),S)$;

Nous définissons la fonction $\psi^{-1}(P,a)$ pour un ensemble d'états P et une lettre $a \in \Sigma$ comme

étant $\psi^{-1}(P,a) = \{q \quad | \quad \psi(q,a) \in P\}$. Un ensemble d'états B est *raffiné par la paire* (P,a) en B' et B'' si les ensembles $B' = B \cap \psi^{-1}(P,a)$ et $B'' = B \backslash B'$ sont non-vides tous les deux. Autrement, B est dit *stable pour la paire* (P,a).

L'algorithme 3 maintient un ensemble S de paires (P,a) pour raffiner la partition d'états $\mathcal{P}_e$. Pour chaque bloc $B \in P_e$ raffiné par une paire (P,a) en B' et B'', la partition $\mathcal{P}_e$ est remise à jour (B est remplacé par B' et B'') . Pour $a \in \Sigma$, si la paire (B,a) existe dans l'ensemble S, elle est remplacée par les paires (B',a) et (B'',a). Autrement, on ajoute à S *la plus petite* $(min\{|B'|,|B''|\})$ des paires (B',a) et (B'',a).

En effet, si B est stable par (P,a), et P est partitionné en P', P'', alors le partitionnement obtenu en raffinant B par (P',a) ou par (P'',a) est exactement le même.

Parmi les détails techniques importants, il faut noter que la fonction SEPARER$\big(B,B',B'',\mathcal{P}_e\big)$ utilisée dans l'algorithme 3 doit être linéaire en temps par rapport à $|B|$. Ceci implique d'utiliser pour les états des structures de données de type *listes indexées*. Le choix du plus petit ensemble entre B' et B'' donne une minimisation d'automate faite en temps $\mathcal{O}(|\Sigma| \cdot n \cdot log(n))$ dans le pire cas.

EXEMPLE 13 : *minimisation d'automate à l'aide de l'algorithme de Hopcroft*

Nous reprenons l'automate non minimisé de l'exemple 12. Après initialisation (algorithme 3), nous obtenons la partition initiale $\mathcal{P}_e = \big\{Q\backslash Q_F, Q_F\big\} = \big\{\{q_0,q_1,q_2\},\{q_3\}\big\}$, ainsi qu'une pile S contenant les tuples $(\{q_3\},a)$ et $(\{q_3\},b)$.
Une première itération dans la boucle principale de raffinement va d'abord, pour le tuple dépilé $\big(\{q_3\},a\big)$, ne trouver aucune partie de $\mathcal{P}_e$ raffinée. Une seconde itération pour le tuple $\big(\{q_3\},b\big)$ raffine la partie $\{q_0,q_1,q_2\}$ de $\mathcal{P}_e$ en $\{q_0\},\{q_1,q_2\}$. La mise à jour qui suit empile le tuple $\big(\{q_0\},b\big)$ dans S. Ce tuple ne raffinant aucune partie, il est dépilé et l'algorithme s'arrête, renvoyant alors la partition $\mathcal{P}_e = \big\{\{q_0\},\{q_1,q_2\},\{q_3\}\big\}$.

2.2.4 Automate de Aho-Corasick

Nous considérons dans cette partie l'**automate de Aho-Corasick** [2], un automate particulier qui permet de reconnaître un ensemble de mots W facteurs d'un texte, et son algorithme de construction, ainsi qu'un exemple illustratif.

L'algorithme de Aho-Corasick a été proposé dans le cadre de la recherche bibliographique, afin de localiser un ensemble de mots W dans un texte. Cette recherche est faite en temps linéaire en la taille du texte après un pré-traitement des mots de W. Nous pouvons voir cet algorithme comme une extension de celui de Knuth Morris et Pratt [36]. Il est utilisé dans la commande `fgrep` de Unix qui recherche un ensemble fini de chaînes de caractères dans des fichiers. Le principe du pré-traitement consiste à construire un automate qui, pour un ensemble de mots $W = \{w_1,w_2,\ldots,w_k\}$, accepte le langage $\Sigma^* \cdot (w_1|w_2|\ldots|w_k) \cdot \Sigma^*$. Nous noterons $n = |W| = \sum_{i=1}^{k} |w_i|$. L'algorithme 4 de construction de l'automate A se fait en temps $\mathcal{O}(n)$. Cet algorithme est divisé en trois étapes :

- la construction d'un automate en forme d'*arbre des préfixes* de l'ensemble des mots W. Cet automate noté Π est appelé *Trie*. Il sert de squelette à l'automate final : il possède déjà tous les états nécessaires, mais n'est pas *complet* au niveau des transitions. Cet arbre est construit par la fonction AHOCORASICKTRIE$\big(W\big)$ détaillée dans l'algorithme 5.
- la complétion de la fonction de transition afin de rendre l'automate *total*. Cette fonction est calculée à l'aide d'une fonction nommée BORD() qui indique, pour un état donné et donc un préfixe donné p, le plus long préfixe propre d'un mot de W *compatible* avec p. Un mot w de W est dit *compatible* avec p si et seulement si il existe $i \in [1..|p|-1]$ tel

que $w[1..i] = p[|p| - i + 1..|p|]$. La fonction Bord est calculée par la deuxième méthode CalculerFonctionBord(Π) détaillée dans l'algorithme 5.

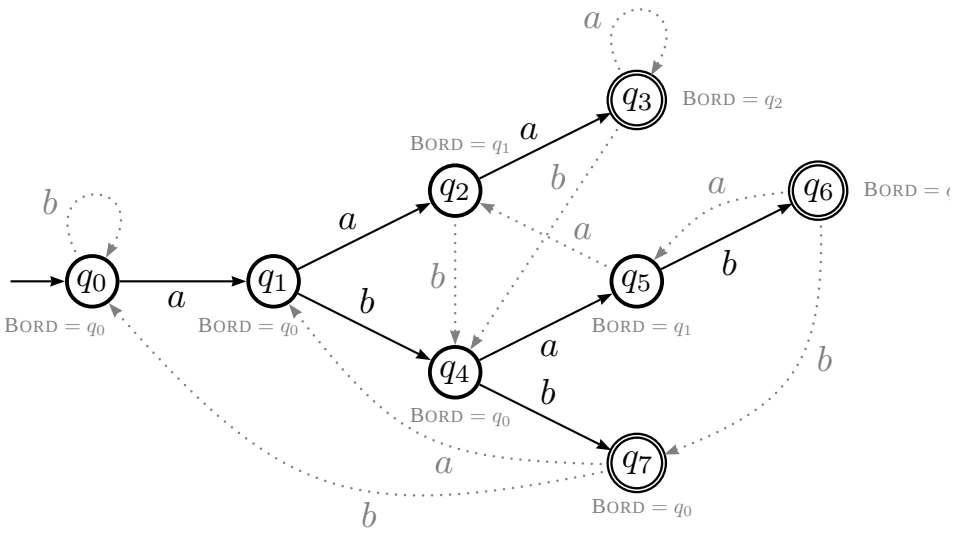
– la complétion de la fonction de transition est réalisée par la méthode CompleterTrie(Π). Notons dans ce cas que si la complétion est faite (construction d'un véritable automate), la complexité est alors en $\mathcal{O}(|\Sigma| \cdot n)$, puisque pour chaque état, il y a au plus $|\Sigma|$ transitions possibles.

Algorithme 4 : *Construction de l'automate de Aho-Corasick*

Entrées : un ensemble de mots $W = \{w_1, w_2, \ldots, w_k\}$ sur un alphabet Σ
Sorties : un automate $A =< Q, \Sigma, q_0, Q_F, \psi >$ reconnaissant les mots de W

AhoCorasick(W)
$A \leftarrow$ AhoCorasickTrie(W);
CalculerFonctionBord(A);
CompleterTrie(A);
retourner A;

Exemple 14 : *automate de Aho-Corasick des mots* $W = \{aaa, abab, abb\}$



L'exemple donné ci-dessus illustre les trois étapes de construction de l'automate. Dans un premier temps, lors de l'appel de la fonction AhoCorasickTrie (algorithme 4), le *Trie* Π des mots $W = \{aaa, abab, abb\}$ est construit (représenté en noir). Dans un second temps, la fonction Bord est calculée pour chaque état de Π par un parcours en largeur d'abord (représentée en gris) . Par exemple pour l'état q_6 et son préfixe associé $p =' abab'$, la fonction Bord retourne q_4 : en effet, le plus long préfixe propre d'un mot de W compatible avec p est ab. Enfin, les transitions manquantes sont ajoutées (représentées en pointillés gris).

Algorithme 5 : *Fonctions associées à la construction de l'automate de Aho-Corasick*

Entrées : un ensemble de mots $W = \{w_1, w_2, \ldots, w_k\}$ sur un alphabet Σ
Sorties : un automate $\Pi =< Q, \Sigma, q_0, Q_F, \psi >$ reconnaissant les mots de W

AhoCorasickTrie(W)
// Création d'un nœud racine ;
$Q \leftarrow q_0$;
pour $a \in \Sigma$ **faire** $\psi(q_0, a) \leftarrow q_0$;
// Ajouts des mots de W ;
pour $w \in W$ **faire**
  $q \leftarrow q_0$;
  $i \leftarrow 1$;
  // Parcours des états et des transitions déjà construits dans l'arbre;
  tant que $i \leq |w|$ **et** $\big(\psi(q, w[i]) \neq \varnothing$ **ou** $\psi(q, w[i]) \neq q_0\big)$ **faire**
    $q \leftarrow \psi(q, w[i])$;
    $i \leftarrow i + 1$;
  // Ajout de nouveaux états et de nouvelles transitions dans l'arbre;
  tant que $i \leq |w|$ **faire**
    $p \leftarrow$ CréeEtat(Q);
    $\psi(q, w[i]) \leftarrow p$;
    $q \leftarrow p$;
    $i \leftarrow i + 1$;
  $Q_F \leftarrow q$;
retourner Π;

CalculerFonctionBord(Π)
$file \leftarrow$ FileVide();
pour $a \in \Sigma$ **faire**
  si $q = \psi(q_0, a) \neq \varnothing$ **alors**
    Ajoute$(file, q)$;
    Bord$(q) \leftarrow q_0$;
tant que $file \neq \varnothing$ **faire**
  $q \leftarrow$ Enleve$(file)$;
  pour $a \in \Sigma$ **faire**
    si $p = \psi(q, a) \neq \varnothing$ **alors**
      Ajoute$(file, p)$;
      $r \leftarrow$ Bord(q);
      tant que $\psi(r, a) = \varnothing$ **faire** $r \leftarrow$ Bord(r);
      Bord$(p) \leftarrow \psi(r, a)$;

CompleterTrie(Π)
pour $q \in Q$ **faire**
  pour $a \in \Sigma$ **faire**
    si $\psi(q, a) = \varnothing$ **alors** $\psi(q, a) = \psi($Bord$(q), a)$;

Chapitre 3

Alignements heuristiques et filtrage avec perte

Sommaire

Dans ce chapitre, nous faisons un état de l'art sur le thème principal abordé dans cette thèse : la recherche de similarités dans les séquences d'ADN. Les *similarités* recherchées sont en général dues à une **homologie**, c'est-à-dire une duplication et une évolution distincte des copies suite à des mutations. Certains types de similarités qui apparaissent sur la *même séquence* (à des localisation distinctes) sont appelés *répétitions*. Parmi elles, on distingue, entre autres, les répétitions dites *en tandem* où les copies sont juxtaposées les unes aux autres.

Le but de la plupart des algorithmes/méthodes proposés dans ce chapitre est d'essayer de retrouver ces similarités, ou au moins un maximum d'entre elles à l'aide d'une évaluation par *système de score* (*scoring system*).

Nous considérerons dans un premier temps la notion de score (partie 3.1) avant d'aborder les algorithmes associés : nous verrons d'abord l'algorithme exact de Smith-Waterman (partie 3.2) pour ensuite nous intéresser à des méthodes heuristiques de recherche de similarités basées sur le principe de *filtrage* (partie 3.3), en particulier celles utilisant des index combinatoires ou par hachage (partie 3.4). Ces dernières servent à la recherche de similarités, mais nous donnerons également deux cas d'utilisation de ces structures pour la recherche de répétitions particulières, les répétitions en tandem (partie 3.5). Enfin, nous décrirons brièvement les méthodes à l'aide de hachages aléatoires (partie 3.6) qui sont à l'origine des méthodes à l'aide de graines espacées (partie 3.7).

3.1 Score et alignement local

Nous avons vu dans la partie 2.1.2.2 que la distance de Levenstein est définie comme le coût minimal d'un ensemble d'opérations d'édition : il s'agit de trouver un ensemble d'opérations unitaires (substitutions lettre à lettre, insertion, suppression) de coût minimal permettant de transformer une séquence en une autre séquence. La distance de Levenstein associe un coût positif à chacune des opérations d'édition, et les correspondances entre lettres ne sont affectées par aucun coût (coût nul).

Dans le cadre d'un système de score, le but n'est plus de *minimiser un coût* mais de *maximiser un score*. Nous allons définir dans un premier temps ce qu'est un système de score. Un système de score associe une pénalité (un *score* négatif) aux opérations d'édition. En contrepartie, il associe un *score* positif aux correspondances entre lettres. Un exemple de **matrice de scores** pour les substitutions sur les séquences d'ADN est donné dans l'exemple 15.

EXEMPLE 15 : *matrice de scores sur l'alphabet* $\Sigma = \{\mathtt{A}, \mathtt{T}, \mathtt{G}, \mathtt{C}\}$

$S(a,b)$	A	T	G	C
A	+1	-2	-1	-2
T	-2	+1	-2	-1
G	-1	-2	+1	-2
C	-2	-1	-2	+1

Le score de l'ensemble des opérations d'édition et des correspondances est «additif» : le score d'un alignement trouvé est égal à la somme des scores des opérations d'édition et des correspondances.

Il existe un algorithme de programmation dynamique permettant de calculer le score de deux séquences sur toute leur étendue (**alignement global**). Cet algorithme appelé **algorithme de Needleman-Wunsch**, est de structure semblable à celui proposé en parties 2.1.5-2.1.6.

Cependant, un avantage intéressant du système de score est qu'il n'oblige pas à évaluer deux séquences sur toute leur étendue. Il est en effet possible de ne considérer que des paires de fragments de séquences. Ce type de comparaison est appelé **alignement local**). Le score d'un alignement local doit être *maximal*, en ce sens que toute sous-partie stricte de l'alignement local trouvé doit toujours posséder un score inférieur à l'alignement.

Deux exemples d'alignements locaux maximaux sont donnés dans l'exemple 16.

EXEMPLE 16 : *alignements locaux de* $u =$ `GAAGAGCATGAC` *et* $v =$ `GAGCTGAGGAG`

Les scores des substitutions $S(a, b)$ *sont donnés par la matrice de l'exemple précédent. Nous y associons les scores d'indels* $Del(a) = -2$ *et* $Ins(a) = -2$ *pour tout* $a \in \Sigma$

Par rapport à ce système de score, les alignements locaux suivants sont trouvés

$\begin{pmatrix} \texttt{GAGCATGA} \\ \texttt{GAGC-TGA} \end{pmatrix}$ entre le fragment $u[4..11]$ et le préfixe $v[1..7]$, le score associé est de 5.

$\begin{pmatrix} \texttt{GAAGAG} \\ \texttt{GAGGAG} \end{pmatrix}$ entre le préfixe $u[1..6]$ et le suffixe $v[6..11]$, le score associé est de 4.

Observation sur les matrices de scores Afin de calculer les coefficients contenus dans une matrice de scores, le principe généralement adopté est le suivant : on veut savoir en quoi un alignement d'une ou plusieurs lettres est dit *significatif*, c'est-à-dire peu probable si l'on aligne des séquences aléatoires. Cette *significativité* peut être calculée en établissant un ratio entre deux probabilités, celle de l'évènement observé et celle de l'évènement attendu :

- le numérateur sera la probabilité que l'évènement observé, par exemple l'alignement d'une lettre A avec une lettre C, apparaisse sur des séquences possédant une *origine commune*. Cette origine commune peut être évaluée en prenant en compte un modèle d'évolution : c'est le cas des matrices PAM (*Percentage Accepted Mutation*) où le modèle simule l'évolution des séquences communes en appliquant un taux de mutation de 1% à la séquence d'ADN : en itérant cette opération, on obtient ainsi les matrices PAM30, PAM100, PAM250 [38]. D'autres modèles utilisent un jeu de données (un ensemble d'alignements obtenus sur un jeu de données) pour évaluer l'origine commune, il s'agit des matrices BLOSSUM [49].
- le dénominateur correspondra à l'évènement attendu dans le cas d'alignements générés aléatoirement : c'est donc la probabilité d'aligner deux lettres données par hasard, ce qui correspond au produit des fréquences.

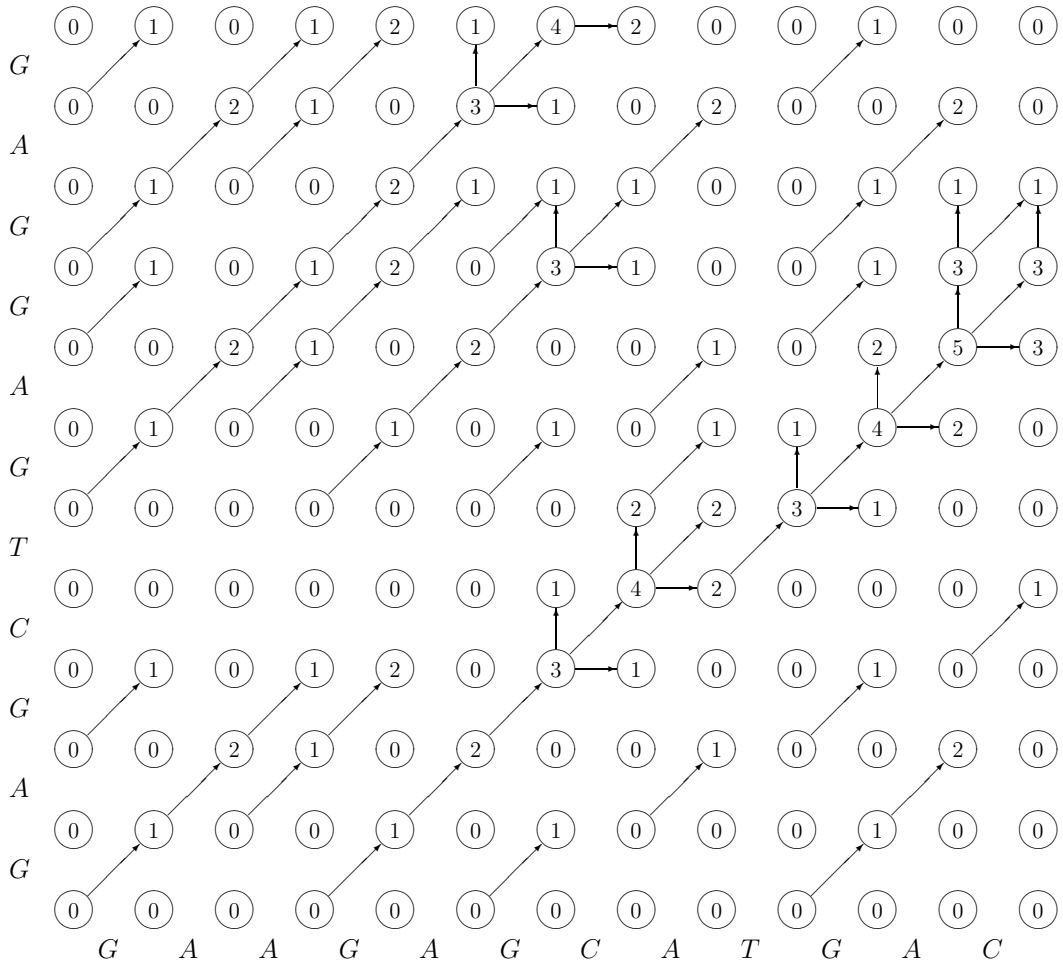
On obtient donc un ratio entre ce qui est observé et ce qui est attendu. De manière à réaliser un produit de probabilités plus rapidement, un *log* du ratio, plutôt que le ratio lui même, est calculé. Les quantités obtenues sont les coefficients de la matrice, et sont aussi appelées *log odds ratio*. Le passage au *log* permet de remplacer les opérations de multiplication de probabilités par des additions.

3.2 Algorithme de Smith-Waterman

Afin de calculer les alignements locaux entre deux séquences, il existe un algorithme exact, l'**algorithme de Smith-Waterman** [97]. Cet algorithme recense tous les alignements de score maximal positif entre deux séquences. Le principe de l'algorithme est similaire à celui utilisé pour calculer la distance de Levenstein (voir partie 2.1.4.2) : sur le graphe d'édition $G(u, v)$ de deux séquences u et v, on applique une relation de récurrence (équation (3.1)) permettant de localiser

les *chemins positifs maximaux.*

EXEMPLE 17 : *graphe de l'alignement local de* $u =$ `GAAGAGCATGAC` *et* $v =$ `GAGCTGAGGAG`



Afin de calculer les alignements locaux des séquences u et v (séquences et système de score de l'exemple précédent), l'algorithme de Smith-Waterman calcule sur le graphe $G(u, v)$, les scores $T(i, j)$ des *chemins positifs maximaux* terminant en $< i, j >$. Seules les arêtes actives ont été représentées.

La relation de récurrence pour le score $T(i, j)$ d'un chemin maximal positif est décomposée par rapport aux chemins maximaux positifs préfixes, donc dépend de $T(i-1, j-1)$, $T(i, j-1)$ et $T(i-1, j)$ (conditions 1,2 et 3 de l'équation (3.1)).

Comme l'alignement est local, un chemin peut donc s'initier (avoir sa source) à n'importe quel emplacement du graphe $G(u, v)$: la récurrence pourra donc commencer par 0 à tout point du graphe (dernière condition de l'équation (3.1)).

$$T(i,j) \quad = \quad max \begin{cases} T(i-1,j-1) + Sub(u[i], v[j]) & \text{si } 0 < i \leq |u| \text{ et } 0 < j \leq |v| \\ T(i,j-1) + Del(u[i]) & \text{si } 0 \leq i \leq |u| \text{ et } 0 < j \leq |v| \\ T(i-1,j) + Ins(v[j]) & \text{si } 0 < i \leq |u| \\ 0 \end{cases} \tag{3.1}$$

Pour la raison précédemment évoquée (alignements locaux), on ne sait pas a priori quand un alignement termine sur le graphe $G(u,v)$. Il est donc nécessaire de localiser, dans la matrice $T(i,j)$, les scores maximaux, au fur et à mesure du calcul. La recherche des alignements est alors faite à l'aide de la méthode de retour arrière identique à celle vue à la section 2.1.5.

3.3 Méthodes heuristiques : filtrage, sensibilité et sélectivité

L'algorithme de Smith-Waterman est un algorithme exact : il donne comme résultat *tous les alignements de score maximal positif*. L'inconvénient de cet algorithme est qu'il a une complexité quadratique en temps ($\mathcal{O}(|u| \times |v|)$), ce qui le destine à des séquences relativement courtes (au plus quelques milliers de bases). Dans le cadre de la comparaison de séquences de plusieurs millions de bases (chromosomes), il n'est plus applicable et doit donc être remplacé par des algorithmes plus rapides.

Les méthodes heuristiques ont été conçues afin d'accélérer le processus d'alignement local. Elles ne garantissent pas la complétude du résultat, mais permettent une recherche sur des séquences de taille plus importante. De manière à ne pas calculer les $|u| \times |v|$ possibilités d'alignements ($|u|$ positions possibles sur la première séquence et $|v|$ positions sur la deuxième séquence), une sélection des paires de positions $< i,j > \in [1..|u|] \times [1..|v|]$ potentiellement intéressantes est nécessaire. Cette sélection des paires de positions est réalisée à l'aide d'un *filtre*. Un filtre, dans le contexte de la recherche de similarités, permet de sélectionner rapidement les paires de positions $< i,j >$ sur deux séquences u et v susceptibles de contenir des similarités.

Définition 5 (*hit* et *critère de hit* d'un filtre) *On appelle* hit *l'événement qu'un filtre génère lorsqu'il détecte qu'une paire de positions $< i,j >$ sur deux séquences u et v est susceptible de contenir une similarité. Le critère employé pour déterminer les paires générant un* hit *est appelé* critère de hit *du filtre.*

Un *hit* ne signifie donc pas qu'une similarité est détectée de manière certaine à une position $< i,j >$, mais seulement qu'il est *possible* ou *probable* d'en obtenir un.

Parmi les filtres possibles pour la recherche de similarités, un exemple de filtre pour identifier les paires potentielles $< i,j >$ peut être la correspondance (égalité) entre les deux mots $u[i..i+k-1]$ et $v[i..i+k-1]$ pour un entier k donné. Ainsi, si des similarités possèdent au moins un mot de taille k en commun, elles peuvent être détectées. Sinon elles sont «oubliées» par l'heuristique. De manière assez naturelle, plus k sera grand et plus le filtre sera dit *sélectif* (et donc rapide), mais moins il sera sensible : moins de similarités posséderont un mot conservé de taille k. En revanche si k est petit, la *sensibilité* sera accrue, mais comme un mot court possède plus d'occurrences sur la séquence qu'un mot long, la sélectivité sera faible et l'algorithme lent.

De manière à définir formellement la **sélectivité** et la **sensibilité** d'un filtre, nous devons fixer un *critère de similarité*. Il peut s'agir par exemple d'un ensemble d'alignements de score supérieur à un seuil donné. Ces alignements peuvent être donnés, soit par un jeu de données pré-calculé, soit par un modèle.

Certains alignements vérifiant le critère de similarité sont détectés par le filtre, on parle alors de *vrais positifs*. D'autres vérifiant le critère sont perdus par le filtre : il s'agit alors de *faux négatifs*. Le filtre détecte par ailleurs des paires de positions (*hit*) qui ne vérifient pas le critère de similarité (mauvais alignements ou pas d'alignement du tout), on parle alors de *faux positifs*. Ces ensembles sont résumés dans la table 3.1.

TAB. 3.1 – *Corpus d'un filtre*

	vérifie/ne vérifie pas le critère de similarité	
détecté / non détecté par le filtre	*vrais positifs*	*faux positifs*
	faux négatifs	*vrais négatifs*

Il est possible de définir la *sensibilité* et la *sélectivité* à partir du cardinal des ensembles *vrais positifs*, *vrais négatifs*, *faux positifs* et *faux négatifs* . La définition est donnée dans les équations (3.2-3.3).

$$\text{Sensibilite} = \frac{\text{vrais positifs}}{\text{vrais positifs} + \text{faux negatifs}} \tag{3.2}$$

$$\text{Selectivite} = \frac{\text{faux positifs}}{\text{vrais positifs} + \text{faux positifs}} \tag{3.3}$$

Le modèle peut également générer des alignements en associant une probabilité d'apparition à chacun des alignements de l'ensemble généré. Dans ce cas, la *sensibilité* d'un filtre sur ce modèle est la probabilité que le filtre détecte un alignement du modèle.

EXEMPLE 18 : *sensibilité d'un filtre*

Supposons que notre modèle d'alignement soit tous les alignements de longueur 8 avec au plus une substitution. Il y a donc 9 alignements différents possibles selon la présence et la position de la substitution. Nous supposerons ces alignements équiprobables.
Supposons que le filtrage consiste à rechercher au moins une occurrence d'un mot de taille k.
- si $k \leq 4$, tous les alignements possèdent au moins un mot de taille 4 conservé (non affecté par la substitution), ce qui implique qu'ils sont tous détectés par le filtre : la sensibilité est donc égale à 1.
- si $k = 5$, seuls deux alignements parmi les 9 ne sont pas détectés (ceux dont la substitution est placée en $4^{ème}$ ou $5^{ème}$ position), la sensibilité est donc de $\frac{7}{9}$. Elle sera respectivement de $\frac{5}{9}$ pour $k = 6$, de $\frac{3}{9}$ pour $k = 7$, et de $\frac{1}{9}$ pour $k = 8$.

Un filtre est dit *sélectif* s'il élimine de manière efficace les paires de positions qui ne possèdent pas de similarités sous-jacentes. La sélectivité est définie dans l'équation (3.3) comme le ratio entre le nombre d'alignements détectés qui ne vérifient pas le critère de similarité et le nombre total d'alignements détectés par le filtre. Cette définition dépend de la taille du résultat, donc de la redondance des deux séquences. Un estimateur plus indépendant consiste alors à évaluer la probabilité d'obtenir un *faux positif* : la sélectivité peut être estimée par la probabilité que le filtre

génère un *hit* à une paire de positions sur deux séquences générées aléatoirement, c'est-à-dire la probabilité d'obtenir un *faux positif*.

EXEMPLE 19 : *sélectivité d'un filtre*

Supposons que le filtrage consiste à rechercher des mots similaires de taille k sur les séquences d'ADN (alphabet $\Sigma = \{A, T, G, C\}$). En supposant que les séquences d'ADN soient générées par un modèle de Bernoulli $\mathcal{B}$, la probabilité d'obtenir un *faux positif* est alors égale à la probabilité d'obtenir deux fois le même mot de taille k à deux positions i, j fixées sur deux séquences différentes u et v. La probabilité d'obtenir la même lettre est égale à $\sum_{a\in\Sigma} \mathcal{P}_{\mathcal{B}}(a)^2$, donc la sélectivité vaut $(\sum_{a\in\Sigma} \mathcal{P}_{\mathcal{B}}(a)^2)^k$. Si les lettres sont supposées équiprobables, la sélectivité est égale à $\frac{1}{4^k}$.

Maintenant que nous avons défini les principes et les mesures appliquées à un filtre, nous allons nous intéresser à l'implémentation pratique de ces derniers. Cette implémentation repose majoritairement sur une notion d'index.

3.4 Méthodes à base d'index

Le filtre sélectionne, dans l'ensemble des paires de positions $< i, j > \in [1..|u|] \times [1..|v|]$, celles qui vérifient le critère de *hit* : lorsque le critère de *hit* est adapté, il est alors possible d'obtenir pour une position i fixée, l'ensemble des positions j à l'aide d'un *index*. De manière à éclaircir la notion d'index, nous reprenons l'exemple précédent du filtre à l'aide de mots de taille k (k-mots). Ce filtre basé sur les k-mots possède le critère de *hit* suivant : la paire de positions $< i, j >$ est un *hit* si et seulement si $u[i..i + k - 1] = v[j..j + k - 1]$.

L'idée consiste alors à construire un *index* pour tous les k-mots du texte v. Cet index donne, pour un mot $w \in \Sigma^k$, la liste des positions de ses occurrences sur la séquence v. Cet index peut ainsi être utilisé pour localiser tous les *hits* du filtre en parcourant d'abord les k-mots du texte u, puis en identifiant les occurrences respectives sur v à l'aide de l'index associé.

Les méthodes de construction de l'index peuvent varier. On distinguera ici deux types d'index, ceux issus des structures discrètes de l'algorithmique du texte, et les autres plus traditionnels basés sur des *fonctions de hachage*.

3.4.1 Index combinatoire

L'algorithmique du texte et ses structures de données associées offrent de nombreuses possibilités d'utilisation et d'adaptation aux problèmes posés, qu'il s'agisse de la recherche de répétitions, ou de l'inférence de motifs. Dans cette partie, nous allons considérer deux cas d'application de structures de données à la construction d'index pour la recherche de similarités sur les séquences d'ADN. Il s'agit de la structure de l'arbre des suffixes, et de l'oracle des suffixes.

3.4.1.1 Arbre des suffixes

En algorithmique du texte, l'arbre des suffixes est l'une des structures les plus connues et fréquemment employée. Elle possède de nombreux avantages, sa taille et son temps de construction sont linéaires en la taille du texte pour la version compacte. La recherche du LCA (*Lowest Commun Ancestor* [48]) est faite en temps constant, ce qui permet des recherches de facteurs communs très rapides. Un exemple d'arbre des suffixes est donné à la figure 3.1.

De nombreux algorithmes (pour une référence plus générale, voir [7, 48]) allant de celui de Weiner [106], à la version incrémentale de Ukkonen [101] en passant par celui de McCreight [79],

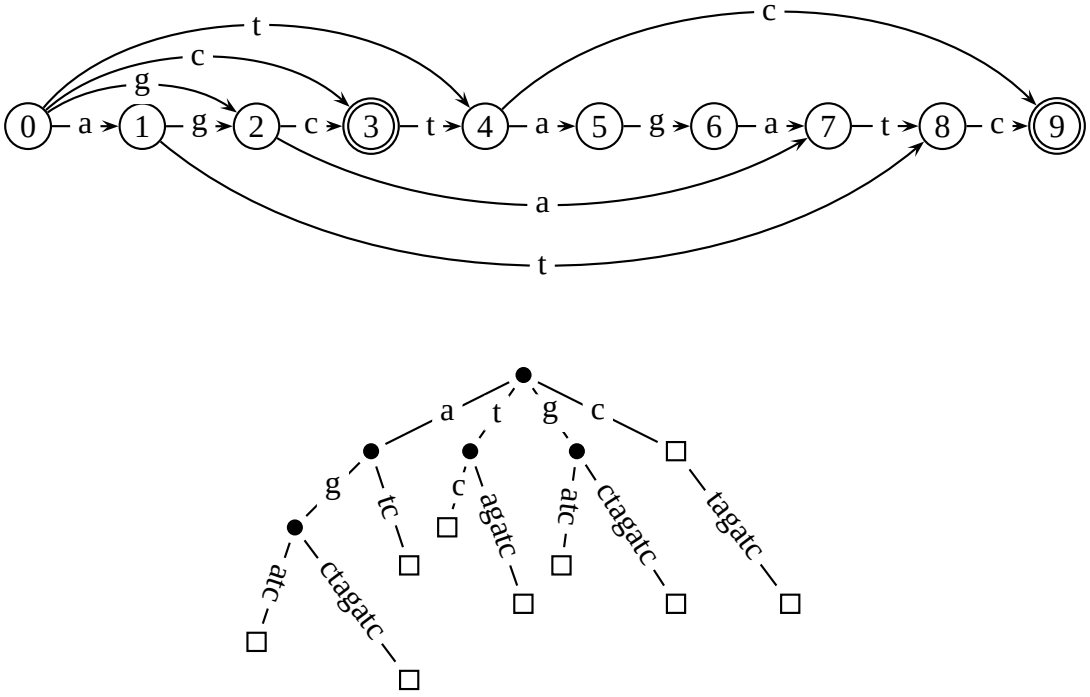


FIG. 3.1 – *Exemples d'oracle des suffixes/d'arbre compact des suffixes*

ont été donnés. Dans le cadre de la recherche de similarité sur les séquences d'ADN, deux logiciels ont été proposés qui utilisent cette structure de données.

REPUTER [68, 69] est un logiciel de recherche de répétitions, basé sur des travaux de Kurtz concernant la réduction de l'espace en mémoire des arbres des suffixes [67]. Le logiciel implante, entre autres, deux algorithmes nommés MMR (*Maximal Mismatch Repeats*) et MDR (*Maximal Difference Repeats*). Étant donnés deux entiers l, k, les deux algorithmes résolvent le problème de la recherche de similarités de taille $\geq l$ ayant au plus k erreurs, respectivement sur les distances de Hamming et de Levenstein. Le logiciel est extrêmement rapide, mais il a l'inconvénient de ne détecter que des similarités quasi-exactes avec ses paramètres par défaut.

MUMMER [39, 40] est un autre logiciel basé sur l'arbre des suffixes, qui réalise *l'alignement global* de deux séquences. Utilisant un arbre des suffixes généralisé sur les deux séquences à aligner, il recherche dans un premier temps des MUM (*Maximal Unique Match*) : le terme UM (*Unique Match*) désigne des fragments de texte dont une **unique copie** est présente dans chacune des deux séquences. Parmi ces UM, seuls ceux qui sont **maximaux** (MUM), c'est-à-dire non contenus dans une autre UM sont calculés par le logiciel.

Les MUM sont ensuite classés selon leur position sur la première séquence. Un algorithme de type LIS (*Longuest Increasing Subsequence*) permet de localiser les MUM ordonnés de manière

croissante et contiguë sur les deux séquences afin d'établir un squelette de l'alignement global. Ce squelette est enfin complété par des méthodes de programmation dynamique (comme par exemple l'algorithme de Smith-Waterman) ou des techniques plus spécifiques aux éléments localisés.

3.4.1.2 Oracle des suffixes

La localisation de motifs faite à l'aide d'un arbre des suffixes donne un résultat sans faux positifs, c'est-à-dire sans prédictions erronées de facteurs. L'inconvénient majeur d'une telle structure provient du fait que les algorithmes efficaces de construction des arbres des suffixes compacts sont loin d'être simples à implanter, en particulier lorsque ceux-ci sont en plus optimisés au niveau mémoire. De plus, même si la taille mémoire de l'arbre dépend linéairement de celle du texte, le facteur *mots machine par nucléotide* dans le cas de l'ADN reste important (5 mots par nucléotide), en conséquence, elle ne peut être appliquée à des séquences de taille trop importante.

Une structure de données annexe, l'oracle des suffixes, a été proposée en 1999 par Allauzen et coauteurs [4]. Elle possède, outre sa taille linéaire par rapport au texte, de bonnes propriétés quant à son facteur *mots machine par nucléotide*. Elle est d'une implémentation bien plus simple, possède une fonction de construction on-line «honnête». En contrepartie, l'oracle des facteurs reconnaît un langage qui n'est pas strictement celui des facteurs de la séquence, mais un sur-ensemble. En ce sens, il peut exister des faux positifs, c'est-à-dire des mots qui ne sont pas facteurs mais cependant acceptés par l'oracle : dans l'exemple de la figure 3.1, le mot `GCTC` est reconnu tout en n'étant pas facteur de `AGCTAGATC`. Cependant, il est possible d'utiliser les liens suffixes (exacts pour ces derniers) comme le fait le logiciel FORRepeats.

Le logiciel FORRepeats [70] proposé par Lefebvre et coauteurs utilise l'oracle des facteurs dans le cadre de la recherche de répétitions sur les séquences d'ADN. L'outil recherche les facteurs communs en utilisant les liens suffixes, calcule leur longueur (*longuest repeated suffixes*) tout en conservant un temps de calcul linéaire en la taille de la séquence.

3.4.2 Index par hachage

Les techniques combinatoires ont l'avantage de garantir des propriétés sur les séquences recherchées (taille minimale des répétitions, garantie de trouver par là même toutes les répétitions respectant certains critères). Cependant, un des problèmes souvent rencontrés concernant l'algorithmique du texte est qu'elle s'adapte mal à la notion de *recherche avec erreurs*. Le coût d'une recherche combinatoire lorsque des erreurs sont autorisées augmente de manière exponentielle, même sur des distances simples comme par exemple la distance de *Hamming*. Cette approche est d'un usage limité en pratique si les séquences sont divergentes. L'ADN étant soumis aux substitutions de nucléotides avec des taux relativement importants, il faut alors remédier à ce problème, en ne garantissant éventuellement pas la complétude du résultat.

L'idée du *filtrage avec perte* (où la complétude du résultat n'est plus garantie) a été une des première utilisées en pratique afin de localiser des répétitions. Le principe consiste à rechercher des éléments qui ont une forte probabilité d'apparaître dans les répétitions que l'on recherche. Cette démarche s'apparente en premier lieu à un procédé d'ingénierie pour essayer de résoudre partiellement un problème jugé difficile. Nous verrons qu'il n'en est rien et que le filtrage (qu'il s'agisse d'ailleurs de filtrage *avec perte* ou *sans perte*) est plutôt intéressant, ne serait-ce que pour les problèmes combinatoires qu'il peut poser sur la conception des filtres.

3.4.3 Recherche des k-mots

De nombreuses méthodes proposent un procédé de filtrage basé une recherche de courts fragments de texte qui sont supposés conservés entre deux séquences.

Une première approche consiste à prendre des fragments de texte contigus. Ces fragments de texte sont en général de taille fixée k sur l'alphabet des séquences Σ. Sous réserve que la plage d'entiers $[0..|\Sigma|^k - 1]$ puisse être codée dans un mot machine, il est alors possible d'associer un entier $key(u)$ à chaque mot $u \in \Sigma^k$ à l'aide d'une fonction bijective comme par exemple :

$$key(u) = \sum_{i=1}^{k} u[i] \times |\Sigma|^{i-1}$$

en supposant que l'alphabet Σ est codé par des entiers de 0 à $|\Sigma| - 1$. Ainsi, deux mots de taille k sur l'alphabet Σ peuvent être comparés en temps constant, à l'aide de leur code $key()$. Un autre avantage de cette fonction provient du fait que, si deux mots à encoder sont chevauchant ($u_1, u_2 \in \Sigma^k$ tels que $u_1[2..k] = u_2[1..k-1]$) alors le calcul du code du second mot peut être réalisé en temps constant à partir du code du premier. En effet, $key(u_2) = \frac{key(u_1) - u_1[1]}{|\Sigma|} + u_2[k] \times |\Sigma|^{k-1}$. Cette propriété permet d'encoder efficacement tous les k-mots d'une séquence donnée T de taille n en temps $\mathcal{O}(n)$.

Une fois l'encodage réalisé pour tous les k-mots d'un texte T, on peut associer, à tout k-mot $u \in \Sigma^k$, la liste (éventuellement vide) des positions de ses occurrences sur une séquence T. Cette liste peut être calculée, pour une séquence T de taille n, à l'aide

- d'un tableau d'entiers L de taille $n - k + 1$,
- de deux tableaux d'entiers F et B de taille Σ^k.

L'algorithme 6 provient de l'article [14]. Pour un k-mot dont le codage est donné par la valeur *code*, chaque case des tableaux $F[code]$ (*First*) et $B[code]$ (*Bucket*) donne respectivement la première/dernière position d'une occurrence trouvée de ce k-mot.

Le tableau L sert de *corps* à l'ensemble des listes. Associé à F, il possède la propriété suivant : si le k-mot u apparaît aux positions $I = \{i_1 < i_2 \ldots < i_r\}$ sur la séquence T, alors $F[key(u)] = i_1$, et $\forall j \in [1..r-1] \quad L[i_j] = i_{j+1}$. De ce fait $\underbrace{L[L[\ldots L[}_{j-1} F[key(u)]] \ldots]]$ donne la $j^{ème}$ occurrence du k-mot u sur T.

Bien que linéaire, cet algorithme doit être réadapté en pratique, car les listes construites posent un problème d'efficacité au niveau du cache du processeur.

3.4.4 Stratégies pour la recherche de similarités locales

Dans cette partie nous allons considérer un certain nombre d'approches choisies par des logiciels connus afin de localiser les similarités dans les séquences. Nous verrons en quoi ces approches sont quelquefois complémentaires, ou antagonistes, et surtout essayerons de comprendre l'évolution des techniques de filtrage proposées.

3.4.4.1 Le logiciel FASTA

Le logiciel FASTA [73, 74] (LFastA) est un des plus anciens logiciels connus de tous dans le domaine de la bio-informatique. De même que son équivalent pour les protéines FastP, le programme recherche des groupes de k-mots conservés situés sur des *diagonales* proches (voir la figure 3.2 à gauche). Ce terme *diagonale* provient de la vision que l'on a des alignements lorsque

Algorithme 6 : *Algorithme générant la liste des k-mots*

Entrées : un entier k, un texte T de taille n sur l'alphabet Σ
Sorties : deux tableaux d'entiers $F[0..|\Sigma|^k - 1]$ et $L[1..n-k+1]$
Données : $F[0..|\Sigma|^k - 1]$;
$B[0..|\Sigma|^k - 1]$;
$L[1..n-k+1]$;
pour $code \leftarrow 1$ **à** $|\Sigma|^k$ **faire**
  $F[code] \leftarrow -1$;
  $B[code] \leftarrow -1$;
pour $i \leftarrow 1$ **à** $n-k+1$ **faire**
  $code \leftarrow key(T[i..i+k-1])$;
  si $B[code] = -1$ **alors**
    $F[code] \leftarrow i$;
  sinon
    $L[B[code]] \leftarrow i$;
  $B[code] \leftarrow i$;

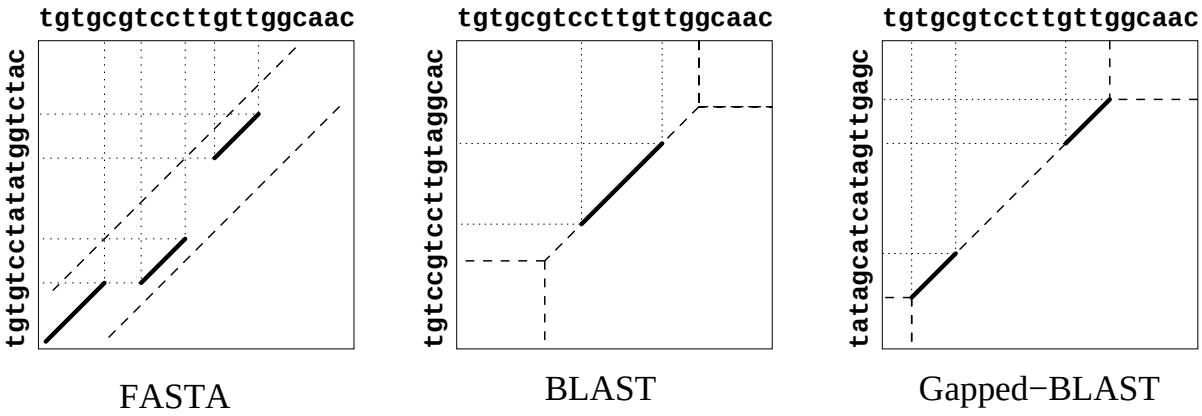


FIG. 3.2 – *Stratégies de filtrage adoptées par des logiciels classiques*

l'on trace un *dot-plot* : les traits provenant des fragments répétés apparaissent consécutivement sur des diagonales proches.

La taille typique pour un k-mot varie entre 4 et 6 lettres sur l'ADN, ce qui destine le programme à des séquences de 10000-100000 bases au plus. Le programme recherche les diagonales dont le score est supérieur à un seuil donné, et évalue les possibles regroupements entre les similarités trouvées.

3.4.4.2 le logiciel BLAST

Avec l'apparition de séquences de plusieurs centaines de milliers de nucléotides (bases de données), une autre stratégie a été adoptée, proposée par BLAST [5]. Plutôt que de considérer des k-mots de petite taille, BLAST n'utilise qu'un seul k-mot de taille bien supérieure ($k \in [11..13]$) (figure 3.2 au centre).

La présence d'un tel mot répété sur deux séquences est certes peu fréquent lorsque les similarités ont subi des mutations, mais la probabilité d'en obtenir au moins une reste élevée, ce

qui permet à BLAST d'obtenir un gain significatif en sélectivité par rapport à FASTA, tout en limitant la perte de sensibilité.

Contrairement à FASTA qui, à partir de la localisation des k-mots, détermine un squelette de l'alignement, BLAST ne connaît a priori, pour un alignement, qu'un seul k-mot répété sur les deux séquences à aligner. L'extension de ce k-mot en un alignement complet se fait alors à l'aide d'un algorithme de programmation dynamique heuristique similaire à celui de Needleman-Wunch [83]. Les différences principales avec ce dernier proviennent du fait que :

- les positions de départ de la programmation dynamique sont situées aux coordonnées données par les extrémités gauches (respectivement droites) des k-mots sur les deux séquences.
- Une heuristique est mise en place afin de limiter la programmation dynamique à un voisinage lorsque le score obtenu n'est pas jugé satisfaisant. Il s'agit du critère dit de *X-drop* : lors d'une d'étape de la programmation dynamique, le score obtenu est comparé au score maximal atteint lors des étapes précédentes. Si le score actuel diminue d'un certain seuil (seuil de *X-drop*) par rapport au score maximal, alors l'extension s'arrête. L'alignement de score maximal est retourné si toutes les possibilités d'extension donnent des scores qui vérifient le critère de *X-drop* .

Un dernier point particulier de BLAST concerne l'évaluation faite sur la *significativité* des alignements trouvés. La question posée est alors la suivante : en quoi un alignement résultat peut être exceptionnel, en d'autres termes quelle est son espérance d'apparaître par hasard si l'on avait fait une comparaison de séquences ne possédant aucune homologie a priori, mais générées selon un modèle aléatoire.

Cette question possède une réponse (asymptotique) selon la théorie développée par Karlin et Altschul [57, 58, 55, 56]. Sous l'hypothèse d'un bruit de fond Markovien $\mathcal{M}$, la distribution des alignements sans *indels* suit une loi de Gumbel [47]. Le logiciel donne alors, à l'aide de cette loi, pour un score s et selon un modèle de Markov $\mathcal{M}$, la probabilité de générer un alignement de score supérieur ou égal à s (P-value) sur des séquences aléatoires de même taille. Les paramètres K et λ proviennent du système de score choisi, lui-même issu du modèle de bruit de fond $\mathcal{M}$. La taille des deux séquences comparées est donnée par m et n.

$$E - value = K \cdot m \cdot n \cdot e^{-\lambda \cdot s}$$

$$P - value = 1 - e^{-E - value}$$

En 1997, une version modifiée de BLAST, nommée GAPPED-BLAST [6], introduit l'idée dite de *hit multiple* : BLAST ne basait sa recherche que sur la présence d'une seule répétition exacte de taille $k \in [11..13]$. GAPPED-BLAST propose d'adopter la stratégie suivante : plutôt que considérer la présence d'une seule répétition d'un k-mot, il vaut mieux le baser sur la présence de deux k'-mots ($k' < k$) relativement proches (figure 3.2 à droite). D'une part la sélectivité globale sera augmentée (moins de *hits* aléatoires), et la sensibilité sera meilleure sur de longues similarités.

Le critère de proximité défini par GAPPED-BLAST repose sur une diagonale commune, à la manière de FASTA, mais en fixant une borne supérieure sur l'espacement entre les deux k'-mots.

3.4.4.3 Variantes utilisant des mots contigus

De nombreuses variantes de BLAST ont été proposées, citons notamment MEGABLAST. Sa particularité, outre le fait que les k-mots recherchés sont plus grands que ceux BLAST, provient d'un algorithme d'alignement glouton [110]. Ce dernier calcule une partie de la matrice de programmation dynamique en utilisant le critère de *X-drop* sur le score (voir partie 3.4.4.2). Il gère

pour cela deux bornes sur le nombre maximum d'insertions / de suppressions de nucléotides respectant le critère de *X-drop* , ne calculant ainsi que les zones potentiellement intéressantes de la matrice de programmation dynamique.

BLAT [60] est également une variante de BLAST, il possède les caractéristiques suivantes du point de vue de la recherche :

- les k-mots utilisés peuvent autoriser une erreur (les k-mots peuvent être distants d'une unité au plus selon la distance de Hamming)
- l'index est réellement créé sur la base de données, et non sur la séquence requête comme c'est le cas pour BLAST.

La première caractéristique permet d'améliorer le ratio sensibilité/sélectivité, mais invoque alors un coût supplémentaire lors de la création de l'index, ce qui rend BLAT intéressant sur des bases statiques.

ASSIRC [103], autre logiciel basé sur la recherche de k-mots, possède une caractéristique particulière par rapport à BLAST, provenant de sa méthode d'extension des *hits*. L'originalité provient d'un filtre précédant l'extension par programmation dynamique : il s'agit d'un filtrage basé sur une marche aléatoire. Pour chaque nucléotide $a \in \Sigma$, on associe un vecteur $v(a)$ de déplacement dans le plan. On trace pour chaque sous séquence lue, sa courbe (succession des déplacements) générée en partant de l'origine. La transformation est injective : deux sous séquences ressemblantes vont voir leur courbes respectives évoluer dans un espace proche. En contrepartie, il n'est absolument pas sûr que deux courbes proches soient un témoin *garanti* de deux séquences semblables.

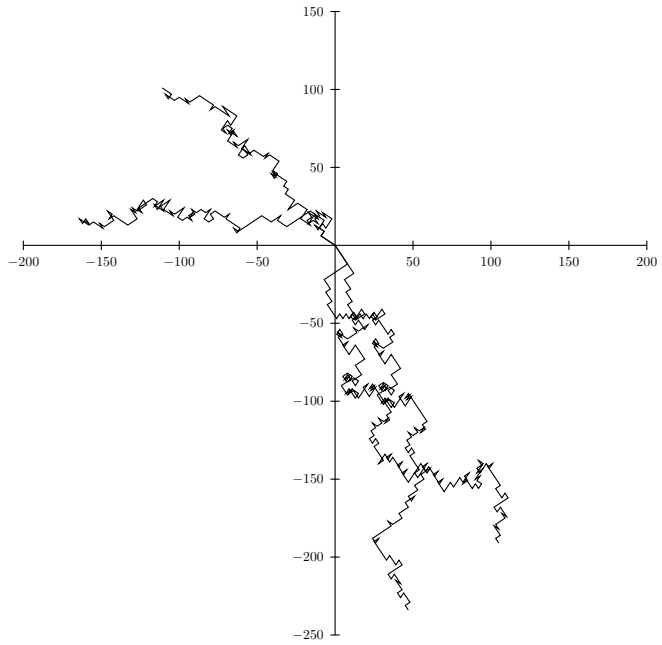


FIG. 3.3 – *Exemple de marche aléatoire*

Le critère retenu par ASSIRC pour considérer une similarité potentielle est basé sur la distance euclidienne entre les deux courbes évaluée à chaque pas. Un seuil maximal δ est fixé sur cette

distance. Si ce seuil est dépassé durant un nombre d'étapes τ, alors l'extension s'arrête. La figure 3.3 donne un exemple de marche aléatoire : les courbes situées sur le quadrant en bas à droite sont relativement similaires, ce qui laisse présager pour les séquences associées une similarité plutôt forte. En contrepartie, le quadrant haut gauche indique des différences importantes quant aux séquences associées.

3.5 Répétition en tandem

Les répétitions en tandem sont constituées de plusieurs copies contiguës d'un motif donné. Elles sont abondantes dans le génome, et sont évolutives. Dans cette famille, des répétitions appelées *micro-satellites* sont composées de fragments courts répétés et juxtaposés jusqu'à des centaines de fois. Ils sont quelquefois responsables de maladies génétiques graves (comme la maladie de Huntington).

Il est intéressant de noter que, dans le cadre de la recherche des répétitions en tandem, il existe également des approches combinatoires et des approches heuristiques.

3.5.1 Un exemple d'approche combinatoire : `mreps`

`mreps` [62, 61] est un logiciel de recherche de répétitions en tandem développé et maintenu par l'équipe Adage au Loria. Il est basé sur un algorithme entièrement combinatoire et permet de trouver toutes les répétitions en tandem dont les copies sont exactes. Il peut également travailler avec des copies approchées (k erreurs en considérant la distance de Hamming).

Le coût de la recherche dans le cas de répétitions en tandem exactes est linéaire, résultat intéressant du point de vue théorique. En effet, la sortie du programme, c'est-à-dire le nombre de répétitions en tandem dites *maximales* est linéaire par rapport à la taille de la séquence.

3.5.2 Un exemple d'approche heuristique : Tandem Repeat Finder

Tandem Repeat Finder [9, 10] est également un logiciel destiné à rechercher des répétitions en tandem. Il base sa recherche sur la création d'un index de k-mots, à la manière de Blast. Cependant, des critères statistiques lui permettent de distinguer les répétitions en tandem des autres répétitions. Cette recherche est faite à l'aide d'un pourcentage d'erreur maximal toléré entre les copies.

Les critères sont calculés selon un certain nombre de paramètres donnés par l'utilisateur (taux de mutation estimé entre les copies, taux d'indels attendu).

3.6 Hachages aléatoires

Jusqu'à présent, les méthodes heuristiques pour la recherche de similarités présentées étaient basées sur une recherche par k-mots (fragments contigus de taille k de texte communs à deux séquences). L'idée d'utiliser des fragments contigus paraît assez naturelle, et semble être la plus appropriée pour localiser des similarités.

En 1993, un logiciel nommé Flash [28] propose d'utiliser des *k-tuples en concaténant des fragments de texte non-contigus* (voir figure 3.4). Il s'agit d'une méthode nouvelle pour générer un index de recherche. La «forme» du fragment c'est-à-dire les positions relatives et les tailles des discontinuités/espacements de chaque k-mot non-contigu sont fixées une fois pour toutes avant la création de l'index, ce choix étant réalisé de manière aléatoire. Cette méthode possède l'avantage d'être très sensible. Mais, étonnamment, cette approche ne donnera pas de travaux

supplémentaires avant les années 2000 dans le cadre de la recherche de similarités sur les séquences d'ADN. En 2001, une méthode similaire est adoptée (de manière indépendante) dans [21]. Elle est analysée d'un point de vue probabiliste et améliorée. De manière plus formelle, l'index considère, pour chaque k-mot w du texte à indexer, non plus la totalité des lettres qui composent ce k-mot, mais uniquement un sous-ensemble donné par un ensemble d'indices $I \subseteq \{0, \ldots, k-1\}$. On admettra que $\{0\} \in I, \{k-1\} \in I$.

FIG. 3.4 – *Dans la partie gauche, un k-mot considère l'ensemble des lettres présentes : le mot obtenu sera* `GATCCTAC`. *Dans la partie droite, un k-mot «espacé» par l'ensemble d'indices* $I = \{0, 1, 2, 4, 6, 9, 10, 11\}$ *ne considère donc qu'un sous-ensemble des lettres : le mot obtenu sera alors* `GAT-C-A--CGA` *plus simplement ré-écrit* `GATCACGA`.

L'ensemble des indices I est fixé avant la construction de l'index. De manière à obtenir des gains plus importants en sensibilité, l'idée consiste à créer un index multiple en utilisant plusieurs ensembles d'indices $\{I_1, I_2, \ldots\}$.

Il est intéressant de noter que cette méthode se base sur des travaux antérieurs (*Locality-Preserving Hashing in Multidimensional Spaces* [53]) dont le domaine est assez éloigné. Ces travaux concernaient la recherche approchée dans des espaces de grande dimension d. Ils proposaient un procédé de hachage pour filtrer rapidement les candidats d'un problème de *plus proches voisins*.

3.7 Graines espacées

La technique de hachage aléatoire proposée dans [21, 24] a apporté une ouverture nouvelle vers des méthodes de *projection*. Plutôt que de baser sa recherche sur un mot contigu, le choix d'un élément moins régulier, c'est-à-dire d'un mot espacé défini par un ensemble d'indices I, a permis de localiser les répétitions avec une sensibilité accrue.

Cependant, tous les ensembles d'indices générés aléatoirement ne sont pas nécessairement plus sensibles que ne l'est l'approche traditionnelle à l'aide de mots contigus. Il existe au contraire des contre-exemples tels que l'ensemble d'indices généré est moins sensible que le cas contigu (voir l'article [23]). Le choix aléatoire de I étant remis en cause, il fallait donc songer à développer des méthodes pour évaluer un ensemble d'indices I.

Nous utiliserons désormais le terme de *graine espacée* pour désigner l'ensemble d'indices $I \subseteq \{0, \ldots, k-1\}$ à la fois fixés et évalués, et non pris au hasard. De manière plus générale, le terme de *graine* sera utilisé pour désigner une méthode associée à un ensemble d'indices, avec éventuellement des contraintes particulières sur ces derniers.

Nous donnons deux définitions plus formelles des *graines* et des *graines espacées*.

Définition 6 (graine) *On appellera* graine *toute expression régulière (mot, motif) servant de base au filtre pour la recherche de similarités. Une graine définit une classe particulière d'expressions régulières qui possède les propriétés suivantes :*

- *une graine π s'applique à une position donnée i sur une séquence u*
- *elle associe alors à tout mot $u[i..i+k-1]$ un entier à l'aide d'une fonction de hachage $key_\pi(u[i..i+k-1])$.*

Définition 7 (graine espacée) *Une* graine π *est dite* espacée *lorsque la fonction de hachage* $key_\pi(u[i..i+k-1])$ *associée n'implique qu'un sous-ensemble de lettres correspondant à un ensemble d'indices* I_π *fixés dans l'intervalle* $[0..k-1]$.

Définition 8 (poids et étendue d'une graine espacée) *Soit une graine espacée* π*, dont la fonction de hachage* $key_\pi(u[i..i+k-1])$ *associe à tout sous-mot de taille* k *un entier selon un ensemble d'indices* I_π*. On distingue pour cette graine deux quantités appelées le* poids *et* l'étendue *de la graine.*

- *le poids* $w(\pi)$ *d'une graine* π *correspond au nombre d'indices dans* I_π*. Il sera également noté* w*.*
- *l'étendue* $s(\pi)$ *d'une graine correspond à la taille* k *du mot recouvert par la graine. Il sera également noté* s*.*

Plutôt que représenter la graine par l'ensemble de ses indices $I_\pi = \{i_1, i_2 \dots i_w\}$, nous utiliserons une notation introduite par Burkhard et Karkkainnen qui représente les graines comme des mots sur l'alphabet $\{\#, -\}$.

Notation (Représentation d'une graine par un mot) *Une graine* π *donnée par un ensemble d'indice* $I_\pi = \{i_1, i_2 \dots i_w\}$ *peut être représentée par un mot de longueur* $i_w - i_1 + 1$ *sur l'alphabet* $\{\#, -\}$ *: le caractère* $\#$ *est utilisé pour représenter les indices de la graine et est appelé caractère* match*. Le caractère* $-$ *correspond aux autres positions et est appelé* joker*.*

EXEMPLE 20 : *représentation d'une graine par un mot*
Si la graine π est donnée par l'ensemble d'indices $I_\pi = \{0, 1, 4, 6\}$, alors sa représentation sous forme de mot sera donné par `##--#-#`. Le poids et l'étendue de la graine seront respectivement $w(\pi) = 4, s(\pi) = 7$.

De manière à créer un index à l'aide de l'algorithme 6, la fonction $key()$ associée doit être redéfinie, il est par exemple possible de prendre la définition suivante :

$$key(u) = \sum_{i=1}^{|I_\pi|} u\Big[1 + I_\pi(i)\Big] \times |\Sigma|^{i-1}$$

3.7.1 Conception des graines espacées

PATTERNHUNTER [75, 72] a été un des premiers logiciels à envisager cette conception de la *graine espacée*. Burkhard et Karkkainnen [25], et de manière indirecte Pevzner et Waterman [91], l'avaient proposée dans le cadre de la recherche de motifs approchée, cadre que nous considérerons d'ailleurs au chapitre 5.

Ma et coauteurs [59] proposent d'utiliser le principe suivant pour le choix de la graine : en définissant un modèle d'alignement, il est possible d'estimer la probabilité qu'a une *graine espacée* de détecter un alignement. Plus exactement, il est possible de calculer la probabilité d'obtenir, sur un alignement du modèle, au moins un fragment qui puisse être détecté par la *graine espacée*.

Cette probabilité est l'estimation, sur le modèle, de la sensibilité. On peut ainsi comparer différentes *graines espacées* et sélectionner les meilleures (les plus sensibles). Bien entendu, cette comparaison doit être réalisée selon un critère de sélectivité fixé, puisque les deux caractéristiques *sélectivité* et *sensibilité* sont liées.

Interviennent alors dans le problème posé :

- La classe des *graines espacées* à sélectionner : le choix se portera sur une classe de graines dont la sélectivité est la même.
- Le modèle d'alignement : il s'agit de donner la distribution probabiliste d'un ensemble de mots de $\mathcal{A}^*$. En général, le choix se porte sur l'ensemble des mots de $\mathcal{A}^n$. La distribution associée est quant à elle plus libre, mais son choix n'est pas neutre quant à la sélection des *graines espacées*.
- Le calcul de la sensibilité pour une graine espacée donnée et sur le modèle donné : l'algorithme proposé dépendra du choix du modèle d'alignement et de la distribution associée.

3.7.2 Classe des graines

Dans la classe que l'on va considérer, les graines espacées doivent avoir la même sélectivité : il s'agit de la probabilité pour le filtre de produire un *hit* à deux positions sur deux séquences aléatoires. En supposant que ces séquences soient générées de manière indépendante et identiquement distribué sur l'alphabet des séquences Σ, cette probabilité dépend, non de la position des indices I_π de la graine π, mais uniquement du nombre d'indices w, c'est-à-dire du poids $w(\pi)$ de la graine. La sélectivité sera alors égale à $(\sum_{a\in\Sigma}\mathcal{P}(a)^2)^w$ où $\mathcal{P}(a)$ est la probabilité de générer la lettre $a \in \Sigma$ dans le modèle de Bernoulli associé. Si l'on suppose que les lettres générées sont équiprobables, alors la probabilité d'un *hit* sur deux séquences générées aléatoirement est de $\frac{1}{|\Sigma|^w}$.

3.7.3 Modèles d'alignement

Le modèle d'alignement proposé par PatternHunter est un modèle d'alignement pour lequel on n'admet que les erreurs de type *substitution* entre lettres. Il est donc sans *indels*. Les alignements peuvent alors être représentés dans ce modèle comme des mots sur un alphabet binaire $\mathcal{A} = \{0, 1\}$ appelé *alphabet des alignements*. Le symbole 1 représente une correspondance (*match*) entre deux lettres de l'alphabet des séquences Σ tandis que le symbole 0 représente une substitution (*mis-match*). Ces alignements sont en général choisis en fixant leur taille n ($n = 64$ dans les articles [75, 72]) .

Exemple 21 : *représentation d'un alignement par une séquence binaire*

Pour l'alignement suivant,

```
ACATTAAGTACACTGAGAT
||||  ||  ||||   |||||
ACATCAAATACAGCGAGAT
```

la séquence binaire correspondante sera donnée par

$$a = 11110110111110011111$$

Pour chaque alignement dans le modèle, on associe sa probabilité d'apparition selon un modèle de Bernoulli $\mathcal{B}$ (Le modèle proposé à l'origine est $\mathcal{B} = \big(\mathcal{P}(1) = 0.7, \mathcal{P}(0) = 0.3\big)$).

Une graine π détecte un alignement α de taille n si et seulement s'il existe une position $i \in [1..n - s(\pi) + 1]$ telle que :

$$\forall j \in [0..s(\pi) - 1] \qquad \pi[j] = \# \implies \alpha[i + j] = 1$$

Si l'on suppose que la graine est représentée sous la forme d'un ensemble d'indices I_π, alors cela équivaut à :

$$\exists i \in [1..n - s(\pi) + 1] \qquad \forall j \in I_\pi \quad \alpha[i + j] = \mathtt{1}$$

Cette équation peut être également écrite sous la forme suivante :

$$\exists i \in [1..n - s(\pi) + 1] \qquad \alpha[i + I_\pi] = \mathtt{1}^{w(\pi)}$$

3.7.4 Algorithme KLMT

Nous abordons dans cette partie l'algorithme de programmation dynamique de [59] destiné à calculer la probabilité pour une graine de détecter un alignement du modèle précédemment exposé. Nous le nommerons KLMT d'après le nom de ses auteurs, mais il est cité dans la littérature comme « algorithme de Keich ».

Il calcule la sensibilité des graines sur un alignement binaire de longueur n généré par un modèle de Bernoulli de paramètre p. Soit $\alpha \in \{\mathtt{0},\mathtt{1}\}^n$ un alignement généré par un modèle de Bernoulli de paramètre p. ($\mathcal{P}(\mathtt{1}) = p, \mathcal{P}(\mathtt{0}) = 1-p$), soit π une graine de poids $w(\pi)$ et d'étendue $s(\pi)$.

Pour l'ensemble d'indices I_π d'une graine π, pour un alignement α de longueur n engendré aléatoirement, et une position i ($1 \leq i \leq n - s(\pi) + 1$), on considère l'événement A_i défini de la manière suivante : A_i est l'apparition d'un *hit* de la graine π à la position i sur l'alignement α :

$$A_i : \alpha[i + I_\pi] = \mathtt{1}^{w(\pi)}$$

Nous souhaitons calculer la probabilité que π génère au moins un *hit* dans un alignement α : $\mathcal{P}(\cup_{i=1}^{n-s(\pi)+1} A_i)$. Nous introduisons pour cela un mot binaire $b \in \{\mathtt{0},\mathtt{1}\}^*$ de taille au plus $s(\pi)$. Pour un entier $i \in [s(\pi)..n]$, nous considérons la probabilité $f(i, b)$ que la graine π génère un *hit* dans le préfixe $\alpha[1..i]$, avec la condition supplémentaire suivante : le suffixe de $\alpha[1..i]$ est donné par b.

$$f(i, b) = \mathcal{P}(\cup_{j=1}^{i-s(\pi)+1} A_j \quad | \quad \alpha[i - |b| + 1..i] = b)$$

La probabilité d'obtenir au moins un *hit* de π dans un alignement α est donnée par $f(n, \varepsilon)$. Un algorithme de programmation dynamique est mis en place pour calculer $f(i, b)$ pour i allant de $s(\pi)$ à n, et pour b choisi dans $B_1 \subset B = \{\mathtt{0},\mathtt{1}\}^l$ avec $l \leq s(\pi)$.

B_1 représente l'ensemble des chaînes b compatibles avec π : il s'agit des chaînes b telles que $A_{i-s(\pi)+1} \cap \{\alpha[i - |b| + 1..i] = b\} \neq \varnothing$, ce qui est équivalent à écrire $(\mathtt{1}^{s(\pi)-|b|}b)[I_\pi] = 1^{w(\pi)}$. B_1 est donné (par les auteurs) comme étant de taille au plus $s(\pi)2^{s(\pi)-w(\pi)}$ (*Note du doctorant : cette borne peut être améliorée en* $w(\pi)2^{s(\pi)-w(\pi)}$ *sur des graines* non dégénérées, *c'est-à-dire ne possédant pas de jokers sur leurs extrémités*).

A partir de ces définitions, il est possible de donner la décomposition récursive de $f(i, b)$.

- si $b \in B \backslash B_1$, alors la graine π ne détecte pas les mots binaires de la forme $\{\mathtt{0},\mathtt{1}\}^{s(\pi)-|b|}b$, ce qui implique que
$$f(i, b) = f(i - 1, b[1..|b| - 2])$$
- si $b \in B_1$ et $|b| = s(\pi)$ alors, par définition, π détecte le mot b ce qui implique
$$f(i, b) = 1$$

– sinon $b \in B_1$ et $|b| < s(\pi)$. Dans ce cas, l'on considère le caractère situé juste avant b. Celui ci étant généré par le modèle de Bernoulli, la décomposition suivante est faite, pour donner :

$$f(i,b) = p \cdot f(i, 1 \cdot b) + (1-p) \cdot f(i, 0 \cdot b)$$

Un algorithme de programmation dynamique permet alors de calculer les valeurs de $f[i,b]$, pour tout $i \in [s(\pi)..n]$ et pour tout $b \in B_1$.

Algorithme 7 : *Algorithme KLMT*

Entrées : Une graine π, une probabilité p, une longueur d'alignement n
Sorties : Probabilité que la graine π détecte l'alignement selon le modèle de Bernoulli $(\mathcal{P}(1) = p, \mathcal{P}(0) = 1-p)$

1. Calculer l'ensemble B_1.
2. Initialiser $f[i,b] = 0$ pour tout $i \in [0..s(\pi)-1]$ et pour tout $b \in B_1$.
3. Algorithme :

```
a pour i ← s(π) à n faire
      pour b ∈ B_1 en prenant d'abord les mots les plus courts faire
          si |b| = s(π) alors
              f[i,b] = 1;
          sinon
b             soit j = max(l) tel que 0·b[1..l] ∈ B_1 ;
              f[i,b] = (1−p)·f[i − |b| + j, 0·b[1..j]] + p·f[i, 1·b] ;
```

4. Retourner $f[n, \varepsilon]$;

La complexité de l'algorithme dépend de la taille de B_1. Les auteurs donnent $|B_1| < s(\pi)2^{s(\pi)-w(\pi)}$, et, par conséquent un algorithme en temps $\mathcal{O}\big(n \cdot s \cdot |B_1|\big) = \mathcal{O}\big(n \cdot s^2 \cdot 2^{s-w}\big)$ à cause des étapes (a) et (b) de l'algorithme 7.

3.7.5 Extension de l'algorithme KLMT

L'algorithme proposé précédemment possède l'inconvénient de n'être appliqué que sur un seul modèle d'alignement, celui généré par un modèle de Bernoulli. Cependant certaines régions de l'ADN possèdent des régularités au niveau des positions des substitutions. Par exemple les séquences codantes possèdent la propriété suivante : la troisième base d'un codon est moins informative (redondance du code génétique) et peut être mutée plus facilement sans changer l'acide aminé associé (l'explication est donnée en partie 1.5). Le modèle de Bernoulli n'étant pas adapté pour simuler ce type de régularité, des extensions ont été proposées.

Brejova et coauteurs [15] ont proposé une extension de l'algorithme KLMT pour des alignements générés par des *modèles de Markov cachées* (notés HMM). Le modèle de HMM correspond, en expressivité, aux automates probabilistes non déterministes (voir la partie 4.2.1.2). Il permet d'exprimer de nombreux modèles sur cette classe pour générer des alignements.

Un HMM $\mathcal{H}$ est composé d'un ensemble d'états $Q_{\mathcal{H}}$ et de transitions probabilistes, avec la particularité suivante qu'une lettre de l'alignement est générée sur un état et non sur une transition. Les auteurs considèrent alors les évènements suivants :

– $\textsc{Chaine}(q,b)$ correspond à la génération par $\mathcal{H}$ d'une chaîne b (représentant un fragment d'alignement) en partant d'un état $q \in Q_{\mathcal{H}}$. La probabilité d'un tel évènement est notée $\mathcal{P}(\textsc{Chaine}(q,b))$.

– $\textsc{Etat}(q,t,i)$ correspond au déplacement de l'état actif de $\mathcal{H}$ de q à t en i étapes. La probabilité d'un tel événement est notée $\mathcal{P}\big(\textsc{Etat}(q,t,i)\big)$.

L'algorithme proposé se base sur la même décomposition, en ajoutant quelques subtilités par rapport à l'algorithme 7 :

$f(i,b,q)$ est la probabilité que π détecte le préfixe $\alpha[1..i]$ terminant par b, en ajoutant désormais la condition supplémentaire que $\alpha[1..i]$ a été générée en partant de l'état q.

$$f(i,b,q) = \begin{cases} 0 & \text{si } i < s(\pi) \\ 1 & \text{si } b \in B_1 \text{ et } |b| = s(\pi) \qquad (1) \\ \sum_{t\in S} p_t \cdot f(i-|d|,c,t) & \text{si } b \notin B_1 \text{ , avec } b = c\cdot d \text{ tel que } c = \max_l b[1..l] \in B_1 \\ & \quad \text{avec } p_t = \mathcal{P}\big(\textsc{Etat}(q,t,|d|)|\textsc{Chaine}(q,b)\big) \\ p\cdot f(i,1\cdot b,q) & \text{si } b \in B_1 \\ +(1-p)\cdot f(i,0\cdot b,q) & \quad \text{avec } p = \mathcal{P}\big(\textsc{Chaine}(q,1\cdot b)|\textsc{Chaine}(q,b)\big) \end{cases} \tag{3.4}$$

Les deux premières lignes de l'équation (3.4) sont équivalentes à l'algorithme original KLMT. Par contre les troisième et quatrième définitions de l'équation (3.4) représentent une extension de l'algorithme. A l'étape trois, puisque la graine π ne peut détecter l'alignement à la dernière position, le suffixe b est réduit à son préfixe maximal c qui puisse être détecté par la graine. Dans ce cas, tous les états t pour lesquels le suffixe d peut être généré doivent être considérés. A l'étape quatre, le suffixe b qui est potentiellement détecté par π doit être étendu de manière à compléter l'algorithme.

Note : pour que l'algorithme reste semblable à KLMT, j'ai volontairement "retourné" le sens de calcul : cela implique que le calcul se fasse, non pas sur la graine π, mais sur la graine renversée $\pi_{s(\pi)}\dots\pi_1$.

En décomposant et pré-calculant les termes $\mathcal{P}\big(\textsc{Etat}(q,t,|d|)|\textsc{Chaine}(q,b)\big)$ et $\mathcal{P}\big(\textsc{Chaine}(q,1\cdot b)|\textsc{Chaine}(q,b)\big)$ des étapes 3 et 4, il est possible de réaliser le calcul en temps $\mathcal{O}\big(s^2\cdot 2^{s-w}\cdot(s^2+\sigma^3+\sigma^2\cdot n)\big)$ avec $\sigma = |Q_{\mathcal{H}}|$

3.7.6 Approche basée sur une décomposition récursive des corrélations

Dans [32], les auteurs proposent une formule et un algorithme associé calculant la probabilité pour une graine de détecter un alignement généré par un modèle de Bernoulli.

Bien que l'algorithme donné soit plus complexe ($\mathcal{O}(n\cdot s\cdot 2^{2(s-w)})$), l'intérêt et l'originalité de la formule provient du fait qu'elle est basée sur la méthode de décomposition classique, basée sur un calcul du temps d'attente et un principe de corrélation des mots. La définition du problème et donc les résultats obtenus sont exactement les mêmes.

3.7.7 Calcul de la sensibilité : approche basée sur des automates

Jusqu'à présent, la décomposition faite pour les graines impliquait de calculer un ensemble B_1 des suffixes potentiels. Cette approche donnait un algorithme en $\mathcal{O}(s^2 2^{s-w} n)$ dans le cas le plus simple (alignements générés par un modèle de Bernoulli). En 2003, Buhler et coauteurs [23] proposent d'utiliser une approche basée sur la construction d'un *automate fini déterministe*. Cette méthode a été précédement proposée dans un cadre plus général par Nicodème et coauteurs [84] afin de calculer la probabilité de la première occurrence ou du nombre d'occurrences de mots dégénérés.

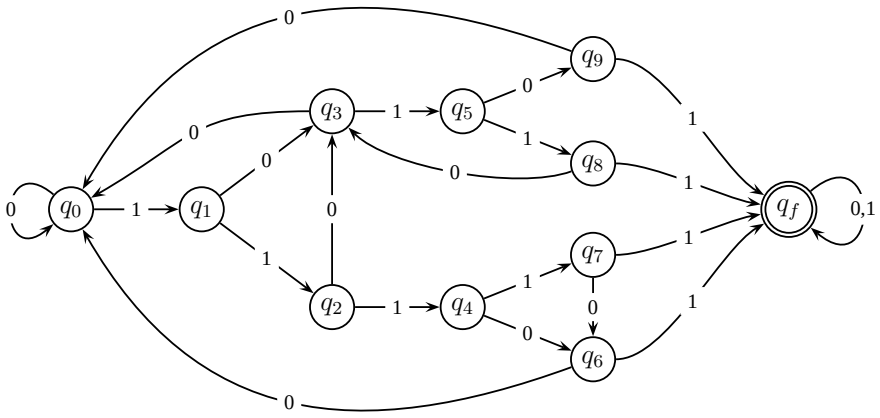
Appliqué au cas des graines, le procédé accélère le calcul, améliore la borne du temps final en $\mathcal{O}(w \cdot 2^{s-w} \cdot n)$ dans le cas d'alignements générés par un modèle de Bernoulli. L'approche proposée donne également, pour un modèle de Markov $\mathcal{M}_k$, un temps de $\mathcal{O}\big((w \cdot 2^{s-w} + 2^k) \cdot n\big)$.

Le principe repose sur un automate A_π qui accepte les alignements α (mots sur l'alphabet $\mathcal{A} = \{0,1\}$) qui sont détectés par la graine π, c'est-à-dire ceux pour lesquels il existe au moins un facteur $\alpha[i..i+s(\pi)-1]$ détecté par π. L'idée est alors de calculer la probabilité que cet automate A_π accepte un alignement généré par un modèle $\mathcal{M}$ donné.

3.7.7.1 Construction de l'automate

Nous décrivons ici la méthode proposée dans [23] pour la construction de l'automate A_π. Pour une graine donnée π (eg $\pi =$ `#-#-#`), l'ensemble des $2^{s(\pi)-w(\pi)}$ mots $\in \mathcal{A}^{s(\pi)}$ détectés par π sont considérés (eg $\{$`10101`, `10111`, `11101`, `11111`$\}$). L'algorithme de Aho-Corasick [2] donne alors une construction de l'automate A_π qui permet de reconnaître le langage des alignements contenant l'un de ces mots.

EXEMPLE 22 : *automate* A_π



L'automate A_π de la graine $\pi =$ `#-#-#` sur l'alphabet $\mathcal{A} = \{0,1\}$ est proposé ci-dessus. Cet automate reconnaît le langage $\{0,1\}^* \cdot \{$`10101`, `10111`, `11101`, `11111`$\} \cdot \{0,1\}^*$

Une description détaillée de l'algorithme de Aho-Corasick, en particulier du calcul de la *fonction* BORD, est donnée à la partie 2.2.4. Nous ne donnons que l'aperçu schématique de l'algorithme de construction de l'automate. Celui-ci se déroule en trois étapes :

- La première étape de l'algorithme, similaire à celle de Aho-Corasick, consiste à construire l'arbre des préfixes des mots détectés par π. Sa taille (nombre d'états) sur une graine non dégénérée est en $\mathcal{O}(w \cdot 2^{s-w})$.
- A l'aide de la *fonction* BORD il est possible de transformer cet arbre en automate total. Cette transformation consiste à ajouter des liens préfixes sur certains états lors de la lecture d'une lettre `0`.
- Une particularité, par rapport à l'automate de Aho-Corasick, concerne ici les états finaux : ces derniers sont rendus *absorbants* ($\forall a \in \mathcal{A} \quad \psi(q_f, a) = q_f$) de manière à reconnaître la première occurrence d'un mot détecté par π. De plus ces états finaux peuvent être

fusionnés car l'on souhaite uniquement savoir si un des mots détecté par π est présent, sans nécessairement le connaître.

3.7.7.2 Calcul des probabilités

L'automate A_π accepte une similarité $\alpha \in \mathcal{A}^*$ si et seulement si la graine π détecte α (détecte au moins un $s(\pi)$-mot facteur de α). En utilisant un algorithme de programmation dynamique, il est alors possible de calculer la probabilité que A_π accepte un alignement α de longueur n généré par un modèle de Markov $\mathcal{M}_k$ d'ordre k.

Nous décrivons le principe de l'algorithme proposé dans [23] : soit c un mot sur $\mathcal{A}^k$, et soit $\Phi_b(q)$ l'ensemble des états précédant un état q par la lecture d'une lettre $b \in \mathcal{A}$. $\mathcal{P}(q, i, c \cdot b)$ est défini comme la probabilité d'atteindre l'état q après avoir lu un préfixe d'alignement $\alpha[1..i]$ terminant par les $k+1$ lettres $c \cdot b$. Il est alors possible de calculer la probabilité d'atteindre tout état q à l'étape i (lecture d'un préfixe $\alpha[1..i]$) : cette probabilité dépend des probabilités d'être à un état précédant $\Phi_b(q)$ à l'étape $i-1$ et de la probabilité de générer la lettre b selon $\mathcal{M}_k$ dans le contexte c. En considérant la variable $k' = min\{k, i\}$ pour l'initialisation du modèle de Markov, la récurrence s'écrit alors sous la forme suivante :

$$\mathcal{P}(q, i, c \cdot b) = \mathcal{P}\big(\alpha[i] = b \mid \alpha[i-k'..i-1] = c\big) \times \sum_{q' \in \Phi_b(q)} \sum_{b' \in \mathcal{A}} \mathcal{P}(q', i-1, b' \cdot c) \tag{3.5}$$

$\mathcal{P}\big(\alpha[i] = b \mid \alpha[i-k'..i-1] = c\big)$ est donné par le modèle de Markov $\mathcal{M}_k$. La récurrence est initialisée par $P(q_0, 0, \varepsilon) = 1$ pour l'état initial q_0, les autres probabilités étant égales à 0. Le calcul est réalisé sur tous les états et les mots $c \cdot b$, en n étapes pour $i \in [1..n]$. La probabilité d'acceptation de l'alignement α par A_π est alors donnée par $\sum_{c \in \mathcal{A}^k} \mathcal{P}(q_f, n, c \cdot 1)$.

L'automate de Aho-Corasick est construit en temps linéaire, et sa taille est ici en $\mathcal{O}(w \cdot 2^{s-w})$ pour une graine π non dégénérée. Le nombre total de transitions est au plus deux fois le nombre d'états (l'automate est en effet construit sur un alphabet $\mathcal{A}$ binaire), ce qui implique que, pour chaque étape i, le coût de l'algorithme de calcul est en $\mathcal{O}(w \cdot 2^{s-w+k})$. Le coût total est donc de $\mathcal{O}(n \cdot w \cdot 2^{s-w+k})$ en temps.

Optimisation Il est également proposé dans [23] une extension de la construction de l'automate qui permet d'obtenir un automate B_π mieux adapté à l'algorithme de calcul par rapport à A_π : il s'agit d'un automate dont tous les états *récurrents* non-finaux ont la propriété d'être atteints par une chaîne c dont les k dernières lettres sont uniques. Les états *transients* sont atteints par un chaîne c unique éventuellement plus courte.

En notant Q l'ensemble d'états, q^f l'état final, et ψ la fonction de transition de B_π, cette propriété peut s'écrire :

$$q \in Q \backslash q^f \quad \exists c \in \mathcal{A}^k \quad | \quad \forall q' \in Q \quad \forall c' \in \mathcal{A}^k \quad \psi(q', c') = q \implies c' = c$$

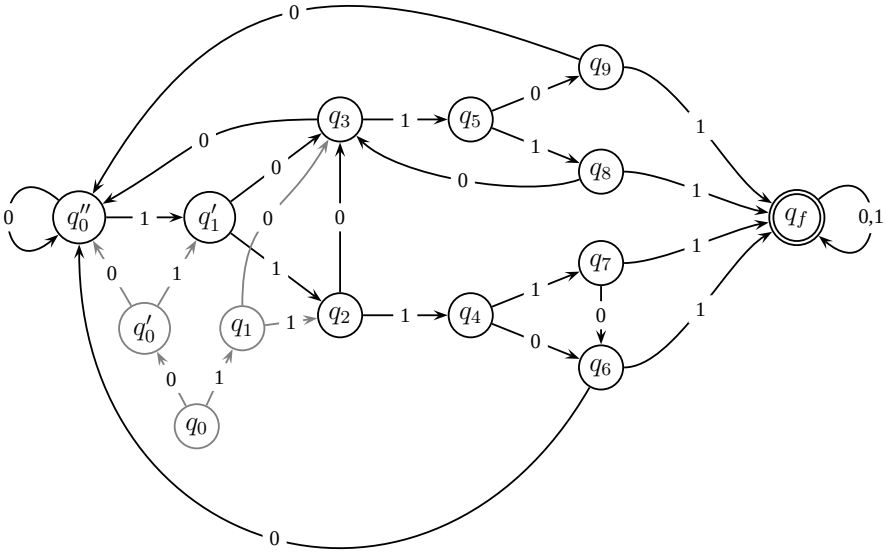
Elle permet ainsi d'associer une chaîne unique c_q de taille au moins k pour tout état *récurrents*, ce qui implique que la probabilité selon le modèle de Markov de rejoindre cet état est connue, simplifiant ainsi l'algorithme de programmation dynamique :

$$\mathcal{P}(q, i) = \sum_{q' \in \Phi_b(q)} \mathcal{P}\big(\alpha[i] = b \mid \alpha[i-k'..i-1] = c_{q'}\big) \times \mathcal{P}(q', i-1) \tag{3.6}$$

B_π est obtenu comme une extension de A_π : si la profondeur d'un état q dans le *trie* de A_π est $\geq k$, alors l'historique des k précédentes lettres pour joindre l'état q est connu, et celui-ci n'a

pas besoin d'être modifié. Autrement, une transformation est apportée à l'ensemble des états de profondeur inférieure à k et un certain nombre d'états initiaux sont ajoutés. Nous ne décrivons pas la procédure complète, mais celle-ci implique l'ajout d'au plus 2^k nouveaux états, et une remise à jour des transitions sortantes et entrantes des états de profondeur inférieure à k.

EXEMPLE 23 : *automate* B_π



L'automate B_π de la graine $\pi =$ `#-#-#` obtenu par extension de A_π pour un modèle d'ordre $k = 2$ est proposé ci-dessus. A l'exception des états *transients* q_0, q_0', q_1 (dont les mots sont respectivement $\varepsilon, \mathtt{0}, \mathtt{1}$), chaque état q de cet automate n'est atteint que par la lecture d'un mot de suffixe unique de taille $k = 2$.

Bien que la taille de B_π, en $\mathcal{O}\big(w \cdot 2^{s-w} + 2^k\big)$, soit supérieure à celle de A_π, le coût total de l'algorithme est cependant de $\mathcal{O}\big(n \cdot (w \cdot 2^{s-w} + 2^k)\big)$ à l'aide de la récurrence (3.6) uniquement paramétrée par les états de B_π et un entier $i \in [0..n]$.

3.8 Extensions du modèle de graines espacées

Les graines espacées ont été assez rapidement étendues vers d'autres modèles afin d'améliorer à nouveau la sensibilité de la recherche. Certains logiciels comme BLASTZ [96] ont par exemple autorisé une erreur de type *transition* sur toutes les positions associées à un symbole # de la graine, ce qui permettait d'améliorer le ratio sélectivité/sensibilité.

Des modèles comme les *graines vecteurs* [16, 18, 19] (*Vector Seeds*) ont été proposés, combinant à la fois des propriétés des graines espacées classiques, avec des techniques de score. Cette approche autorise des erreurs sur certaines positions, avec des contraintes de dépendance sur leur nombre. Une graine vecteur est un couple $\pi = (v, T)$, où v est un vecteur de nombres réels $\{v_1, v_2 \ldots, v_{s(\pi)}\}$, et où T est un réel donnant le seuil de la graine. On associe un score $S(a)$ à chacune des lettres a de l'alphabet des alignements $\mathcal{A}$. Pour un alignement $\alpha = \alpha_1\alpha_2 \ldots \alpha_n \in \mathcal{A}^n$, une graine vecteur π détecte l'alignement α si et seulement s'il existe une position $0 < i \leq n - s(\pi)$

telle que

$$\sum_{j=1}^{s(\pi)} v_j \times S(\alpha_{i+j-1}) \geq T$$

L'algorithme proposé pour le calcul de la sensibilité est une extension de l'algorithme KLMT, mais cette extension est cependant assez coûteuse en temps. De plus les graines vecteurs ne peuvent être détectées à l'aide d'un index simple comme c'est le cas pour les graines espacées.

D'autres modèles comme les graines à longueur variable (*variable length seeds*) sur des modèles de graines contiguës ou espacées ont été proposés [37]. Il s'agit de graines espacées dont les symboles *match* # peuvent être substitués dans un certain ordre par des symboles *jokers* −. Le poids des graines est ainsi variable : si le motif de départ de la graine et l'ordre des substitutions sont bien choisis, alors la sensibilité de l'ensemble des graines ainsi formé permet un gain en sensibilité. Enfin, l'extension proposée dans PatternHunter II [72] et considérée dans [99], consiste à utiliser, non plus une seule graine, mais plusieurs graines espacées qui sont conçues pour être efficaces en conjonction. Cette approche améliore également le ratio sélectivité/sensibilité. Des extensions de l'algorithme KLMT [59] ou à base d'automates [23] sont proposées pour des graines multiples.

Cependant le coût combinatoire causé par l'énumération de plusieurs graines ne permet pas de calculer de manière exacte l'ensemble de graines optimales, ce qui oblige les auteurs à utiliser des heuristiques (*Algorithme Glouton* en particulier). Il est intéressant de noter, qu'au niveau de la difficulté du problème de conception des graines, les auteurs de [72] ont montré la difficulté de calculer la sensibilité d'un ensemble de graines sur un alignement donné par un modèle de Bernoulli. Ce résultat a été affiné dans [85] où les auteurs démontrent que ce problème est également difficile pour une seule graine (deuxième partie de la preuve).

Chapitre 4

Conception de graines

Sommaire

Dans ce chapitre, nous présentons une partie du travail de cette thèse et donnons trois extensions relatives aux modèles de graines espacées et aux algorithmes associés pour le calcul de la sensibilité.

Nous proposons dans la partie 4.1 une extension de l'algorithme de KLMT (décrit en partie 3.7.4). Cette extension permet de mesurer la sensibilité d'une graine espacée sur un sous-ensemble d'alignements dits *homogènes*. Ce sous-ensemble d'alignements homogènes représente les alignements détectés par les algorithmes heuristiques. La mesure de la sensibilité sur ces alignements est donc d'autant plus justifiée qu'elle induit une différence assez significative par rapport aux méthodes traditionnelles. Nous étudions ensuite dans les parties 4.2–4.3 une modélisation plus générale permettant de calculer la sensibilité de graines en considérant un éventail plus large de paramètres, qui vont du modèle d'alignement au modèle de graine en passant par la distribution probabiliste associée à ces alignements. Nous proposons dans la partie 4.4 un nouveau modèle de

graine appelé *graine sous-ensemble* ainsi qu'un automate associé compact avec un algorithme de construction efficace. Nous donnons ensuite dans la partie 4.5 des résultats associés à la fois au propriétés de l'automate des *graine sous-ensemble* et à son utilisation dans le cadre de la partie 4.2. Nous présentons enfin dans la partie 4.6 deux logiciels issus de ce travail : le logiciel YASS recherche des similarités sur les séquences génomiques selon un modèle de *graines à transitions* issu du modèle de *graines sous-ensemble* ; le logiciel HEDERA conçoit des *graines à transitions* sensibles en utilisant toutes les méthodes proposées et développées dans ce chapitre.

Ces travaux ont été réalisés en collaboration avec G. Kucherov, Y. Ponty, et M. Roytberg.

4.1 Alignements homogènes

Jusqu'à présent, nous avons considéré les alignements comme des mots de taille n sans aucune restriction sur la composition de ces mots. En effet seul le modèle probabiliste donnait une probabilité d'apparition différente selon la composition du mot.

Il existe cependant des cas où il est justifié de se concentrer seulement sur certains alignements, jugés quelquefois plus intéressants. Par exemple, les alignements détectés par les algorithmes heuristiques comme BLAST, PATTERNHUNTER, YASS sont soumis au critère dit de *X-drop* (voir partie 3.4.4.2), deuxième filtre heuristique qui permet d'étendre les graines détectées en alignements. Tous les alignements détectés possèdent la propriété suivante : aucun fragment d'alignement ne possède un score supérieur au score de l'alignement total. Cette propriété distingue ces alignements nommés *MSP* d'autres alignements quelconques (*MSP* = *Maximal Scoring Segment Pair*). Nous allons dans cette partie étudier une extension de l'algorithme de KLMT associé aux alignements correspondants que nous appellerons *alignements homogènes*.

4.1.1 Calcul de la sensibilité des graines

Le but est ici de restreindre le calcul de la sensibilité des graines au sous-ensemble des alignements homogènes. Soit α un alignement de longueur n sur l'alphabet des alignements $\mathcal{A}$ (alphabet binaire). Nous considérons dans un premier temps un système de score $S = \{+s, -p\}$ associé à l'alphabet binaire $\mathcal{A} = \{0, 1\}$. La séquence des scores associée à $\alpha = \alpha_1\alpha_2 \dots \alpha_n$ est donnée par $S(\alpha) = S(\alpha_1)S(\alpha_2) \dots S(\alpha_n)$, et le score de l'alignement est $Score(\alpha) = \sum_{i=1}^{n} S(\alpha_i)$.

4.1.1.1 Définitions

Un alignement est homogène si et seulement si le score global de l'alignement $Score(\alpha)$ est strictement supérieur au score de chacune de ses sous-parties : $\forall j_1 \leq j_2 \in [1..n] \quad (j_1, j_2) \neq (1, n) \quad Score(\alpha) > Score(\alpha[j_1 \dots j_2])$. Afin de calculer la probabilité pour une graine π de détecter un alignement homogène, il est important :

- d'isoler la classe des alignements homogènes de l'ensemble des alignements. Afin de distinguer le concept *d'homogénéité*, nous avons choisi de représenter les alignements par des marches aléatoires.
- de savoir ensuite générer ces alignements de manière aléatoire uniforme.
- de savoir enfin estimer la probabilité qu'a une graine π de détecter un alignement homogène : nous avons étendu pour cela l'algorithme de KLMT afin de le restreindre aux alignements homogènes.

4.1.1.2 Représentation

Pour un mot binaire $\alpha = \alpha_1\alpha_2 \dots \alpha_n$, et sa séquence de scores associée $S(\alpha) = S(\alpha_1)S(\alpha_2) \dots S(\alpha_n)$, on considère l'évolution du score des préfixes de longueur k, $Score(\alpha_1 \dots \alpha_k)$. Cette évolution du score peut être représentée par une marche C sur $\mathbb{Z}^2$ partant de l'origine $(0, 0)$, se déplaçant à l'aide de deux vecteurs unitaires $(1, +s)$ et $(1, -p)$, pour aboutir au point (n, S) pour un alignement α de score S et longueur n (voir la figure 4.1).

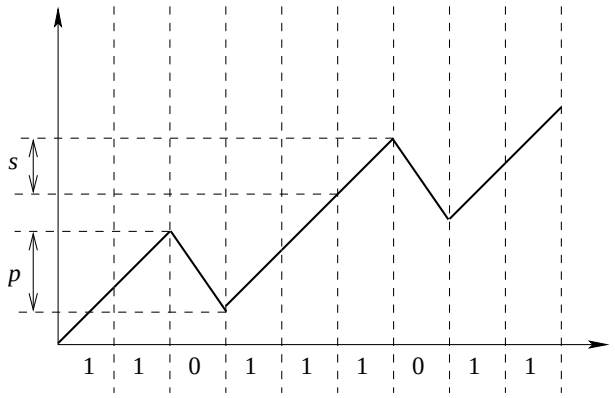


FIG. 4.1 – *Marche aléatoire C associée à l'alignement α*

Les alignements homogènes correspondent aux marches aléatoires C qui vérifient les deux conditions suivantes :

- la marche C est *positive* : $\forall (k, y) \in C, \quad k > 0 \implies y > 0$,
- la marche C est *culminante* : $\forall (k, y) \in C, k < n \implies y < S$.

Une marche aléatoire vérifiant cette condition sera appelée *marche culminante positive*.

4.1.1.3 Énumération et génération aléatoire

Nous allons énumérer les chemins culminants positifs associés aux alignements homogènes. Nous considérerons deux cas, selon que le score final S est atteint ou dépassé ou exactement atteint.

Score atteint ou dépassé

Nous considérons dans cette partie les *marches culminantes positives* qui atteignent ou dépassent un score S fixé. Soit C_n l'ensemble des *marches culminantes positives* de longueur n et de score atteint ou dépassé S. Une approche traditionnelle pour la génération aléatoire de séquences de taille fixée n issues d'un langage L consiste à compter les suffixes de ces séquences[107]. De cette manière, on peut générer de manière incrémentale les séquences issues de L. Cette approche est préférable à des techniques de génération avec rejet, où la génération est réalisée de manière uniforme sur toutes les séquences, pour ensuite éliminer celles qui ne respectent pas les contraintes. En particulier, la génération avec rejet dépend fortement des paramètres s et p choisis dans le système de score S. Elle peut être tolérée si les contraintes de rejet ne sont pas trop fortes (par exemple lorsque la probabilité d'un rejet est en $\mathcal{O}(\frac{1}{n})$). Dans notre cas, si s est plus petit que p (ce qui est souvent le cas en pratique), la probabilité de rejet tend vers 1 avec une vitesse de convergence exponentielle en n, donnant un algorithme en temps moyen exponentiel en n.

L'approche par dénombrement ou comptage se base sur le principe suivant. Soit L le langage à générer sur un alphabet $\mathcal{A} = \{a_1, \ldots, a_m\}$, et soit L_n l'ensemble des mots de L de longueur n. Soit w_p un préfixe d'une séquence de L_n. Nous appelons $P_a(w_p)$ la probabilité que w_p soit suivi de la lettre $a \in \mathcal{A}$ pour former un mot de L_n

$$P_a(w_p) = \frac{|\{w' | w_p a w' \in L_n\}|}{|\{w'' | w_p w'' \in L_n\}|}. \tag{4.1}$$

Lemme 1 (génération du langage L_n de manière uniforme) *Étant données les valeurs $P_a(w_p)$ pour tout $a \in \mathcal{A}$ et tous les préfixes w_p, il est possible de générer les mots de L_n de manière uniforme.*

Preuve (génération du langage L_n de manière uniforme) En partant du mot vide ε, on génère des lettres consécutives $\alpha_1, \ldots, \alpha_n$ avec les probabilités $P_{\alpha_1}(\varepsilon)$, $P_{\alpha_2}(\alpha_1)$, $P_{\alpha_3}(\alpha_1\alpha_2), \ldots,$ $P_{\alpha_n}(\alpha_1\alpha_2\cdots\alpha_{n-1})$. La probabilité de générer un mot $w = \alpha_1\alpha_2\alpha_3\cdots\alpha_n$ est alors

$$\begin{aligned} P(\alpha_1 \ldots \alpha_n) &= P_{\alpha_1}(\varepsilon) \cdot P_{\alpha_2}(\alpha_1) \cdot P_{\alpha_3}(\alpha_1\alpha_2) \cdot \ldots \cdot P_{\alpha_n}(\alpha_1\alpha_2\cdots\alpha_{n-1}) = \\ &\frac{|\{w'|\alpha_1 w' \in L_n\}|}{|L_n|} \cdot \frac{|\{w'|\alpha_1\alpha_2 w' \in L_n\}|}{|\{w'|\alpha_1 w' \in L_n\}|} \cdot \ldots \cdot \frac{|\{w'|\alpha_1 \ldots \alpha_n w' \in L_n\}|}{|\{w'|\alpha_1\cdots\alpha_{n-1} w' \in L_n\}|} = \frac{1}{|L_n|}, \end{aligned} \tag{4.2}$$

car $\{w'|\alpha_1 \ldots \alpha_n w' \in L_n\} = \{\varepsilon\}$. La procédure est donc une procédure de génération uniforme. □

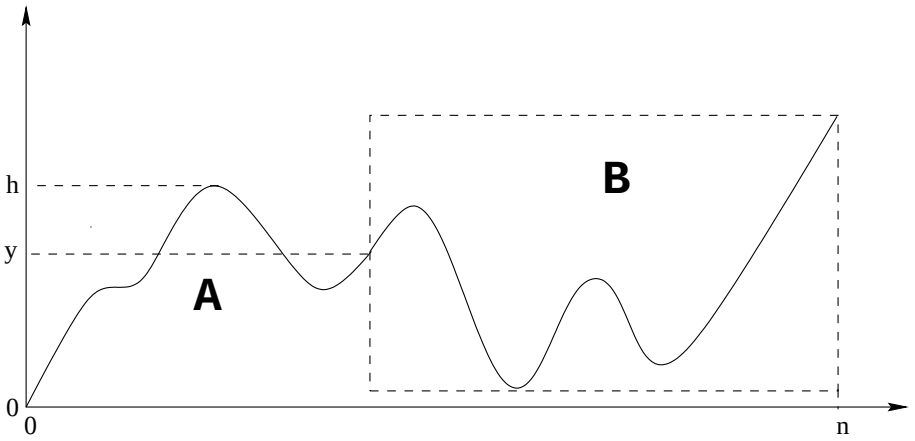


FIG. 4.2 – *Exemple de suffixes `B` associés au préfixe `A` de C_n dépendant uniquement de l'ordonnée maximale h et de l'ordonnée actuelle y*

Il est en général nécessaire de pré-calculer jusqu'à $|\mathcal{A}|^n$ valeurs de $P_\alpha(w_p)$. Cependant, dans le cas de séquences homogènes, il n'est pas obligatoire de faire ce calcul pour tous les préfixes w_p de manière individuelle, car la seule information requise pour un préfixe donné w_p concerne d'une part le score maximum qu'il a atteint h, ainsi que le score actuel y (voir la figure 4.2). Pour cela, nous introduisons le concept de chemin (h, y)-initialisé.

Définition 9 (chemin (h, y)-initialisé de longueur n) *Pour $h, y, n \geq 0$, $y \leq h$, un chemin (h, y)-initialisé est un chemin de longueur n partant de $(0, y)$ et allant à (n, S), tel que $S > h$.*

Nous noterons $C_{y,h,n}$ l'ensemble des chemins (h,y)-initialisés de longueur n.

Lemme 2 (relations entre chemins (h,y)-initialisés) *Supposons que w_p soit une marche positive de $(0,0)$ jusqu'à (k,y), et supposons que h soit son score maximum atteint. Alors $\{w' | w_p w' \in C_n\} = C_{y,h,n-|w_p|}$.*

La preuve est immédiate et illustrée à la figure 4.2.

Afin de compter le nombre de chemins (h,y)-initialisés de taille n (pour toutes valeurs h,y et n compatibles entre elles), nous utiliserons la décomposition récursive suivante des chemins (h,y)-initialisés. Un chemin (h,y)-initialisé sera représenté par une suite de vecteurs pris parmi $\{(1,s),(1,-p)\}$.

Lemme 3 (ensemble des chemins (h,y)-initialisés) *Pour $y,h \geq 0$,*

$$C_{y,h,1} = \begin{cases} (1,s) & \text{si } y+s > h \\ \varnothing & \text{sinon.} \end{cases} \tag{4.3}$$

et pour $k > 1$,

$$C_{y,h,k} = \begin{cases} (1,s) \cdot C_{y+s,\max(h,y+s),k-1} \uplus (1,-p) \cdot C_{y-p,h,k-1} & \text{si } y > p, \\ (1,s) \cdot C_{y+s,\max(h,y+s),k-1} & \text{sinon.} \end{cases} \tag{4.4}$$

Preuve (ensemble des chemins (h,y)-initialisés) Une marche positive culminante d'un seul pas ne peut pas être composée d'un pas $(1,-p)$. Elle ne peut donc être composée que d'un pas $(1,s)$ à condition que $h < y+s$. La condition générale est constituée de deux sous-cas, selon que le premier pas soit $(1,-p)$ ou $(1,s)$. Le premier pas n'est possible que si $y > p$, si le premier pas est $(1,s)$, alors le maximum atteint sera recalculé : $\max(h, y+s)$. □

Remarquons que la décomposition des $C_{y,h,n}$ ne pose pas d'ambiguïté, l'union étant toujours une union entre ensembles disjoints. De ce fait, le lemme 3 donne également une formule récursive pour calculer le nombre de chemins positifs $C_{y,h,k}$ pour les valeurs $1 \leq k \leq n$, $0 \leq y,h \leq s \cdot n$. Par programmation dynamique, cela peut être fait en espace et temps $\mathcal{O}(n^3)$.

Le lemme 2 permet de compter le nombre de suffixes pour tout préfixe w_p d'une marche de C_n. Soient $k = n - |w_p|$, où y est le score de l'extrémité finale du mot w_p, et h le score maximal atteint par w_p. Alors, la probabilité $P_1(w_p)$ que w_p soit suivi par un pas $(1,s)$ est donnée par $|C_{y+s,\max(h,y+s),k-1}|/|C_{y,h,k}|$. Dès que les valeurs $|C_{y,h,k}|$ sont calculées, il est possible de générer une séquence de C_n en temps $\mathcal{O}(n)$ en utilisant le lemme 1.

Score S atteint fixé

Nous apportons quelques modifications à la précédente approche afin de générer des séquences homogènes de score *fixé* S. Cela signifie que nous considérons des chemins positifs culminants avec un point culminant fixé S. Ce cas est plus simple, puisqu'il n'y a qu'un nombre fini de scores intermédiaires. De ce fait l'ensemble des séquences devient un langage régulier, pour lequel il existe un algorithme de génération en temps linéaire (en incluant le temps de pré-traitement) [50, 42]. Dans notre cas il est suffisant de pré-calculer une table de dimensions $n \times S$ pour mémoriser le nombre de suffixes de chemins positifs culminants partant de $(0,0)$ et aboutissant au point (i,s). Ces derniers peuvent être vus comme des marches à l'intérieur du rectangle dont les extrémités sont $(0,y)$ et (k,S). Soit $D^S_{y,k}$ l'ensemble de ces marches. Le lemme suivant établit la récurrence associée.

Lemme 4 *Pour $y \geq 0$,*

$$D^S_{y,1} = \begin{cases} (1,s) & si\ y+s=S \\ \varnothing & sinon. \end{cases} \tag{4.5}$$

et pour $k > 1$,

$$D^S_{y,k} = \begin{cases} (1,s)\cdot D^S_{y+s,k-1} \uplus (1,-p)\cdot D^S_{y-p,k-1} & si\ p < y < S-s, \\ (1,s)\cdot D^S_{y+s,k-1} & si\ y \leq p < S-s \\ (1,-p)\cdot D^S_{y-p,k-1} & si\ p < S-s \leq y \\ \varnothing & sinon. \end{cases} \tag{4.6}$$

L'union est ici également disjointe, donc la récurrence peut être utilisée pour compter le cardinal de chaque ensemble $D^S_{y,k}$. En utilisant les lemmes 1 et 4, on obtient un algorithme de génération uniforme en temps et espace $\mathcal{O}(S \cdot n)$.

4.1.1.4 Calcul de la sensibilité

Nous présentons désormais l'algorithme qui réalise le calcul de la sensibilité (probabilité d'un *hit*) sur une séquence aléatoire homogène. Cet algorithme est une extension de l'algorithme de programmation dynamique de [59] qui calcule la probabilité d'un *hit* sous le modèle uniforme des séquences. Nous restreignons ici l'ensemble des alignements aux alignements homogènes. Le principe de la construction repose sur la représentation des séquences homogènes par des marches culminantes positives sur le plan (voir la partie 4.1.1.2), ainsi que sur les formules de comptage (4.5) et (4.6).

Afin de simplifier la présentation, nous donnons l'algorithme pour une seule graine espacée π. Nous rappelons que nous représentons une graine espacée π par une chaîne sur l'alphabet $\{-,\#\}$. La longueur de cette chaîne est appelée *étendue* et sera notée $s(\pi)$. Le nombre de $\#$ de la graine est appelé *poids* et sera noté $w(\pi)$. Une graine π *détecte* un alignement $\alpha \in \{0,1\}^*$ s'il existe une position p telle que $\forall i \in [1..w(\pi)] \quad \pi_i = \# \implies \alpha_{i+p} = 1$.

Nous donnons désormais l'algorithme de programmation dynamique qui calcule la probabilité qu'une graine donnée π détecte un alignement homogène de longueur n et de score S selon le système de score $\{+s, -p\}$.

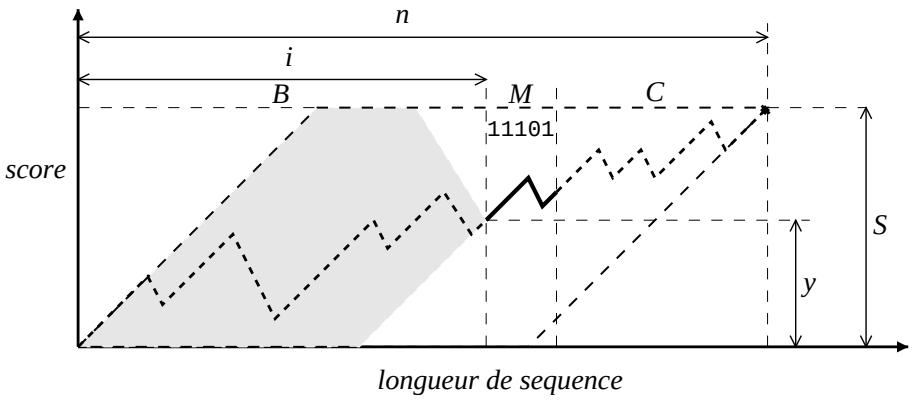


FIG. 4.3 – *Calcul de la sensibilité d'une graine sur des séquences homogènes*

Considérons un préfixe d'une séquence homogène aléatoire α, on suppose que ce préfixe termine par un suffixe M. Soit $\alpha = BMC$ et soit $|B| = i$ et $S(B) = y$. Soit $P(i,M,y)$ la

probabilité que π détecte le préfixe BM de α (voir la figure 4.3). Notre but est de calculer la probabilité $P(n, \varepsilon, S)$ en utilisant une décomposition récursive. Nous donnons d'abord les conditions initiales de la récursion :

(i) $P(i, M, y) = 0$, si $i + |M| < s(\pi)$,

(ii) $P(i, M, y) = 1$, si $|M| = s(\pi)$ et π détecte M.

Les conditions suivantes assurent de garder des séquences homogènes. La condition (iii) assure des scores strictement positifs tout en restant inférieurs à S. La condition (iv) est optionnelle mais permet de couper des branches de calcul infertiles : elle assure que la marche reste à l'intérieur de la partir grisée (voir la figure 4.3)

(iii) $P(i, M, y) = 0$, si $y \geq S$ et $i < n$, ou $y \leq 0$ et $i > 0$,

(iv) $P(i, M, y) = 0$, si $y > i \cdot s$ ou $y < S - (n - i) \cdot s$.

Les relations suivantes décrivent les étapes principales de la récursion :

(v) si π ne détecte pas le mot $1^{s(\pi)-|Mb|} \cdot Mb$ ($b \in \{0, 1\}$), alors $P(i, Mb, y) = P(i, M, y)$,

(vi) si $|M| < s(\pi)$, alors $P(i, M, y) = P_1 P(i - 1, 1.M, y - s) + P_0 P(i - 1, 0.M, y + p)$, avec P_1 et P_0 calculés selon l'équation (4.7) en utilisant les formules (4.5), (4.6).

$$P_1 = \frac{|D^S_{S-y+s,i-1}|}{|D^S_{S-y,i}|}, \quad P_0 = \frac{|D^S_{S-y-p,i-1}|}{|D^S_{S-y,i}|} \tag{4.7}$$

La condition (v) indique que si un suffixe Mb ne peut être détecté par la graine π, alors la dernière lettre b peut être enlevée. La condition (vi) est la plus délicate. Elle implique que si M est plus court que $s(\pi)$, mais peut éventuellement être détecté par π, alors M est étendu à gauche : deux sous-cas disjoints sont considérés, l'extension par une lettre 1 ou une lettre 0. La probabilité est alors décomposée en la somme de deux termes correspondant aux deux états associés de la marche aléatoire lors de l'extension à gauche de M (voir la figure 4.3).

Une manière de calculer ces deux probabilités P_0 et P_1 consiste en quelque sorte à retourner la figure (rotation de 180 degrés) et à considérer les marches comme provenant de (n, S) vers $(0, 0)$. Alors, les probabilités P_0 et P_1 sont calculées à l'aide de la formule (4.7) en utilisant la méthode de comptage décrite en partie 4.1.1.3.

Les marches qui contribuent aux probabilités P_0 et P_1 sont alors celles localisées dans la zone grisée de la figure 4.3.

La décomposition récursive de $P(n, \varepsilon, S)$ est réalisée de la manière suivante : en appliquant la condition (vi), la taille de M croît jusqu'à atteindre la longueur $s(\pi)$ pour les mots M qui sont détectés par la graine π. Ensuite, par une alternance des conditions (vi) et (v), la taille de M varie entre $s(\pi)$ et $s(\pi) - 1$ tandis que i décroît (à moins que l'une des conditions (i)-(iv) soit appliquée).

Une première borne de la complexité en temps de l'algorithme peut être donnée par $2^{s(\pi)} \cdot S \cdot n$. Il est cependant possible, en exploitant la structure des graines et en utilisant l'algorithme 7 présenté à la partie 3.7.4 de ramener la complexité en temps à $\mathcal{O}\big(s(\pi) \cdot 2^{s(\pi)-w} \cdot (s(\pi)^2 + S \cdot n)\big)$.

L'algorithme présenté peut être étendu à d'autres critères de *hit*, comme le *double hit* où un *hit* du filtre n'est défini que lorsque deux occurrences d'un *hit* simple d'une graine apparaissent dans un voisinage proche. Ce principe de *double hit* est proposé dans Gapped-Blast [6] (deux occurrences de graine non-recouvrantes) (partie 3.4.4.2) et utilisé dans Blat [60] (deux occurrences ou plus non-recouvrantes), PatternHunter [75] (deux occurrences de graine avec recouvrement possible), Yass [88] (recouvrement d'un nombre quelconque de graines).

De manière à étendre l'algorithme au cas de K occurrences d'une graine sans contraintes de recouvrement, il est suffisant de réaliser le calcul de la probabilité $P(i, M, y, k)$ où le paramètre k $(0 \leq k \leq K)$ signifie que k occurrences de la graine ont déjà été trouvées dans le mot BM.

La modification principale s'applique à la relation (ii), qui sera transformée en $P(i, M, y, k) = P(i, M^-, y, k-1)$, où le mot M^- est le préfixe du mot M de longueur $|M| - 1$.

Si le recouvrement entre deux graines successives est majoré par une constante Δ, la modification est réalisée de la même manière, à la différence que le mot M^- sera alors le préfixe du mot M de longueur $|M| - \Delta$. Si la stratégie de détection (*double hit*) impose une borne supérieure sur la distance entre deux graines avoisinantes, l'algorithme devient plus complexe, car un paramètre supplémentaire doit être mémorisé dans la récursion pour conserver la distance entre la dernière graine trouvée et le bord du mot.

4.1.2 Expériences

De manière à montrer le biais induit en ignorant la propriété d'homogénéité, nous comparons la probabilité d'un *hit*, pour différentes stratégies de graines, selon que les alignements soient homogènes ou arbitraires. Cette comparaison est réalisée pour différents scores et différentes longueurs d'alignements.

Les probabilités pour des alignements homogènes sont calculées à l'aide de l'algorithme de la section 4.1.1.4. Pour les alignements arbitraires, une version plus simple ne prenant pas en compte la propriété d'homogénéité est utilisée.

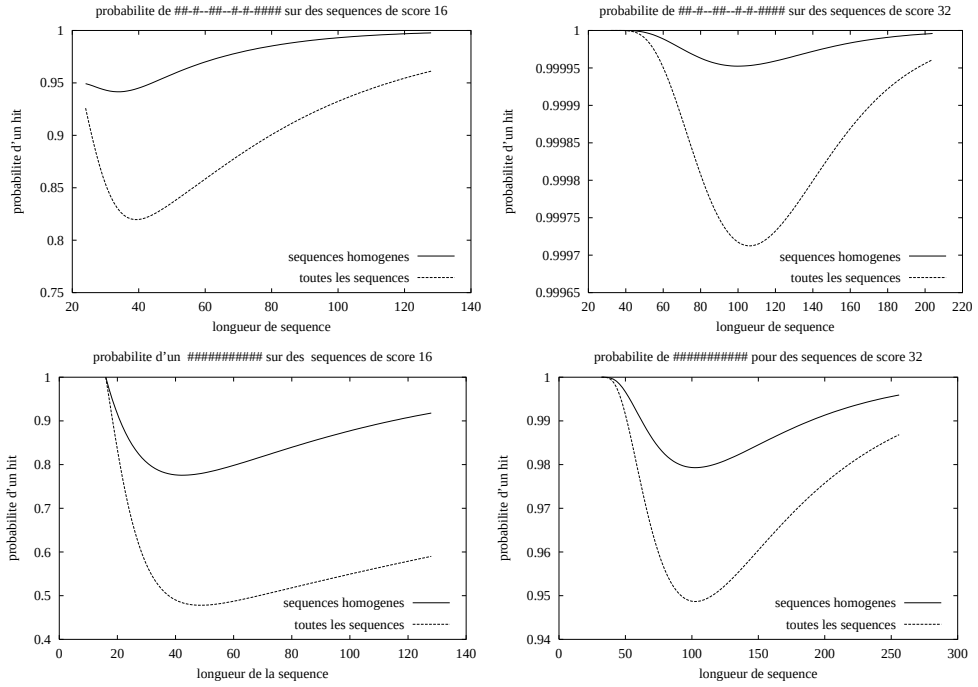


FIG. 4.4 – *Probabilité de* hit *d'une graine sur des alignements homogènes et des alignements arbitraires*

La figure 4.4 donne les résultats associés au calcul sur des alignements. Chaque courbe représente la probabilité pour une graine donnée, de détecter des alignements homogènes/arbitraires d'un score donné selon leur longueur.

Toutes les expériences de cette section utilisent le score par défaut de BLAST $\{+s = +1, -p = -1\}$. Les courbes des colonnes gauche et droite correspondent respectivement aux séquences de scores 16 et 32. Les deux courbes de la ligne supérieure correspondent à la graine `##-#--##--#-#-####` de poids 11 et étendue 18 utilisée par PATTERNHUNTER [75], les deux courbes de la ligne inférieure correspondent à la graine contiguë de poids 11 (BLAST 1 [5]).

Quelle que soit la graine choisie, les résultats montrent que le fait d'ignorer la condition d'homogénéité produit une sous-estimation importante de la sensibilité. La fraction des alignements perdus par les graines est deux fois moins importante dans le cas des alignements homogènes que dans le cas d'alignements arbitraires.

Un autre intérêt de la mesure de la sensibilité des graines concerne la conception des graines optimales pour la détection d'alignements. Une autre série d'expériences a été menée afin de comparer les graines optimales (les graines les plus sensibles) pour l'ensemble des alignements homogènes et l'ensemble des alignements arbitraires. Les résultats sont donnés dans la table 4.1.

Le calcul des *graines optimales* a été réalisé sur des alignements soit homogènes, soit arbitraires, de longueur 40 dont le score est entre 12 à 24. Les résultats de la table 4.1 ont été obtenus par recherche exhaustive sur toutes les graines dont l'étendue est inférieure ou égale à 20.

Les probabilités données ont été calculées par l'algorithme de la partie 4.1.1.4. Pour des paramètres de score et de longueur identiques, des graines optimales obtenues sont différentes selon que la probabilité est calculée sur des alignements arbitraires (probabilité $\mathcal{P}_a$) ou des alignements homogènes (probabilité $\mathcal{P}_h$). La probabilité la plus élevée est donnée en italique.

(n, S)	w	graine	$\mathcal{P}_h$	$\mathcal{P}_a$
$(40, 12)_h$	9	`###--#-##-###`	*0.986271*	0.902372
$(40, 12)_a$	9	`###--#--#-#-###`	0.983516	*0.917869*
$(40, 16)_h$	9	`###--#-##-###`	*0.998399*	0.988887
$(40, 16)_a$	9	`##--##-#-####`	0.998353	*0.989535*
$(40, 16)_h$	10	`###-##--#-####`	*0.98742*	0.938499
$(40, 16)_a$	10	`##--##-#-####`	0.98740	*0.942769*
$(40, 20)_h$	10	`###-#--###-###`	*0.999172*	0.996303
$(40, 20)_a$	10	`##-##--#-#-####`	0.999065	*0.996555*
$(40, 20)_{ha}$	11	`###-###-#--####`	*0.975462*	*0.993076*
$(40, 24)_{ha}$	11	`###-#--####-###`	*0.999891*	*0.999661*

TAB. 4.1 – *Graines optimales pour les alignements homogènes et les alignements arbitraires*

4.2 Modélisation à l'aide d'automates

Nous avons vu dans la précédente partie une extension de l'algorithme de KLMT adaptée aux séquences homogènes. La méthode permet un calcul de la sensibilité de graines espacées classiques sur des séquences homogènes. Mais elle reste assez spécialisée et ne peut par exemple s'adapter à des modèles comme les *graines à transitions* sans devoir émettre certaines hypothèses sur la probabilité des alignements : tous les alignements homogènes sont considérés comme équiprobables, ce qui n'est pas une hypothèse raisonnable quand certains alignements constitués par exemple uniquement de substitutions de type *transitions*, sont bien moins fréquents que des alignements mixant à la fois des substitutions de type *transitions* et *transversions*.

Le calcul de la sensibilité d'une graine peut être sujet à de nombreux choix de paramètres. Par exemple on peut se poser la question du *modèle de graine* à choisir. Mais il est aussi question du *modèle d'alignement*, et également de la *distribution des alignements*. Il est assez naturel de modéliser ces composantes de manière indépendante, de séparer par exemple le modèle de la graine et le modèle d'alignement. De même il est possible de distinguer le langage des alignements de la distribution probabiliste qui génère ces alignements.

4.2.1 Principe

De manière à estimer la sensibilité d'une graine ou d'un ensemble de graines donné, il faut calculer la probabilité qu'un mot aléatoire (un alignement cible) sur un alphabet d'alignement $\mathcal{A}$, généré à partir d'un certain langage (langage des alignements) selon une certaine distribution probabiliste, soit détecté par une graine ou un ensemble de graines donné.

De manière à calculer cette probabilité, nous allons modéliser les composantes suivantes :

- le langage des alignements cibles.
- la distribution probabiliste associée à ces alignements.
- le modèle de graine.

Ces trois composantes seront modélisées par des automates (éventuellement probabilistes) décrits respectivement dans les parties 4.2.1.1, 4.2.1.2, et 4.2.2. Le principe utilisé pour le calcul de la sensibilité est ensuite abordé à la partie 4.3. Il se base sur des produits d'automates et d'automates probabilistes successifs et permet d'obtenir un automate probabiliste mixte avec un seul état final sur lequel un algorithme de programmation dynamique est appliqué afin de calculer la probabilité d'atteindre l'état final.

4.2.1.1 Alignements cibles

Les alignements sont représentés comme des mots sur un alphabet $\mathcal{A}$. Cet alphabet peut être soit binaire (représentation des correspondances ou des substitutions), soit prendre en compte différentes substitutions dans le cas de l'ADN ou des protéines. On peux ainsi définir un alphabet représentant différentes familles de substitutions (transitions/transversion sur l'alphabet de l'ADN, propriétés chimiques communes sur l'alphabet des protéines).

On suppose que l'ensemble des alignements reste un ensemble fini, ce qui permet de le considérer comme un langage régulier $L_T \in \mathcal{A}^*$. L'automate associé à la reconnaissance de L_T sera un automate déterministe acyclique $T =< Q_T, q_T^0, q_T^F, \mathcal{A}, \psi_T >$.

Un exemple de ce type de langage peut être l'ensemble des alignements de taille fixée n. Il est possible d'introduire des contraintes supplémentaires comme l'homogénéité des alignements (voir la partie 4.1).

4.2.1.2 Distribution probabiliste des alignements

A chacun des mots du langage L_T, on doit associer une probabilité d'apparition. Cette probabilité sera donnée par un automate probabiliste (déterministe ou non). Cet automate probabiliste est un automate fini dont chaque transition est libellée à la fois par une lettre $a \in \mathcal{A}$ **et** une *probabilité de transition.*

Définitions

Définition 10 (automate probabiliste G) *Un automate probabiliste $G =< Q_G, q_G^0, \mathcal{A}, \rho_G >$ est composé :*

- *d'un ensemble fini d'états Q_G*
- *d'un état initial $q_G^0 \in Q_G$*
- *d'une fonction $\rho_G : Q_G \times \mathcal{A} \times Q_G \rightarrow [0, 1]$ décrivant les probabilités associées aux transitions. Cette fonction vérifie en particulier la clause suivante $\forall q \in Q_G, \sum_{q' \in Q_G, a \in \mathcal{A}} \rho_G(q, a, q') = 1$.*

Il est intéressant de remarquer que cet automate ne possède pas d'état final. En effet, son rôle n'est pas de reconnaître un langage, mais d'assigner des probabilités aux mots d'un autre langage. En particulier, il sera utilisé pour générer une distribution probabiliste sur l'ensemble des alignements du langage L_T. On supposera par ailleurs que l'automate G est *complet.*

Une transition $e =< q, a, q' >$ sur G ne sera définie que si sa probabilité associée $\rho_G(q, a, q')$ est strictement positive. On appelle a le *label* de la transition e noté $label(e)$. Un chemin est une liste de transitions correspondant au chemin dans le graphe de transition associé (voir partie 2.2.1.2).

Définition 11 (label et probabilités associés) *Pour tout chemin $P = (e_1, \ldots, e_n)$, on définit son* label *par le mot $label(P) = label(e_1) \ldots label(e_n)$ et sa probabilité par $\rho(P) = \prod_{i=1}^{n} \rho_G(e_i)$.*

Un chemin est considéré comme *initial* si la transition e_1 est de la forme $e_1 =< q_G^0, a, q' >$.

Définition 12 (probabilité d'un mot) *La probabilité $\mathcal{P}_G(w)$ d'un mot $w \in \mathcal{A}^*$ selon G est égale à la somme des probabilités de* **tous** *les chemins initiaux dont le label est donné par w.*

Il est intéressant de noter que lorsque G est déterministe, alors seul un chemin initial correspondra au mot w.

Définition 13 (probabilité d'un langage) *La probabilité d'un langage fini $L \subseteq \mathcal{A}^*$ selon G est égale à la somme des probabilités des mots $w \in L$.*

Exemple d'automates probabilistes

De manière à illustrer le concept d'automate probabiliste tels que nous le définissons, nous pouvons par exemple représenter un modèle de Bernoulli [75, 32, 31] par un automate probabiliste déterministe à un état. En effet, chaque lettre de l'alphabet $\mathcal{A}$ sera générée avec une probabilité associée à sa transition, et toutes les transitions boucleront sur l'état initial.

Les *modèles de Markov d'ordre k* [23, 99] peuvent être représentés par un automate probabiliste déterministe à $|\mathcal{A}|^{k+1}$ états. Les *modèles de Markov cachés* [17] peuvent également être représentés par des automates finis non-déterministes. Ces deux propriétés sont démontrées dans les deux sous-parties suivantes.

Modèle de Markov et automate probabiliste associé

Définition 14 (modèle de Markov d'ordre k) *Un modèle de Markov $\mathcal{M}$ d'ordre k sur l'alphabet $\mathcal{A}$ est défini par :*

$$P_{init} : \{u | u \in \mathcal{A}^*,\ |u| \leq k\} \to [0,1] \quad et \quad P_{tran} : \mathcal{A}^k \times \mathcal{A} \to [0,1]$$

vérifiant
- $P_{init}(\varepsilon) = 1$
- $\forall u \in \mathcal{A}^*,\ |u| < k,\ \sum_{a \in \mathcal{A}} P_{init}(ua) = P_{init}(u)$
- $\forall u \in \mathcal{A}^k,\ \sum_{a \in \mathcal{A}} P_{tran}(u,a) = 1$

La probabilité $P_{\mathcal{M}}(u)$ d'un mot $u \in \mathcal{A}^*$ selon le modèle de Markov $\mathcal{M}$ est définie de manière récursive par :

(i) si $|u| \leq k$ alors $P_{\mathcal{M}}(u) = P_{init}(u)$,

(ii) sinon $u = va$, $|v| \geq k$, $a \in \mathcal{A}$ alors $P_{\mathcal{M}}(u) = P_{\mathcal{M}}(v) \cdot P_{tran}(w,a)$, avec w représentant le suffixe de v de longueur k.

Lemme 5 (automate probabiliste associé au modèle de Markov d'ordre k) *Soit $\mathcal{M} =< P_{init}, P_{tran} >$ un modèle de Markov d'ordre k sur l'alphabet $\mathcal{A}$. Prenons un automate probabiliste $G(\mathcal{M}) =< Q, q_0, \mathcal{A}, \rho >$ défini par :*

$$\begin{aligned} Q &= \{u | u \in \mathcal{A}^* \ et\ |u| \leq k\}, \\ \rho(u,a,w) &= \begin{cases} P_{init}(w)/P_{init}(u) & si\ |u| < k,\ w = ua,\ P_{init}(u) > 0, \\ P_{tran}(u,a) & si\ |u| = k, |w| = k,\ avec\ w\ suffixe\ de\ ua, \\ 0 & sinon. \end{cases} \end{aligned}$$

Alors pour tout mot $w \in \mathcal{A}^*$, la probabilité de w selon $\mathcal{M}$ est égale à $\rho(w)$ selon $G(\mathcal{M})$.

Modèles de Markov cachés et automate probabiliste associé

Définition 15 (modèle de Markov caché) *Un modèle de Markov caché (HMM) sur un alphabet $\mathcal{A}$ est un quadruplet $\mathcal{H} =< Q_{\mathcal{H}}, p_0, \mathcal{A}, f, s >$ tel que*
- *$Q_{\mathcal{H}}$ est un ensemble fini d'états,*
- *$p_0 : Q_{\mathcal{H}} \to [0,1]$ est une distribution probabiliste initiale sur l'ensemble d'états cachés $Q_{\mathcal{H}}$,*
- *$f : Q_{\mathcal{H}} \times Q_{\mathcal{H}} \to [0,1]$ est la fonction de transition probabiliste entre états cachés,*
- *$s : Q_{\mathcal{H}} \times \mathcal{A} \to [0,1]$ est la fonction d'émission d'une lettre pour un état caché.*

Les propriétés suivantes doivent être vérifiées :

$$\forall q \in Q_{\mathcal{H}} \sum_{q' \in Q_{\mathcal{H}}} f(q,q') = 1 \tag{4.8}$$

$$\forall q \in Q_{\mathcal{H}} \sum_{a \in \mathcal{A}} s(q,a) = 1 \tag{4.9}$$

La probabilité d'une suite d'états cachés $(q_1, \ldots, q_n) \in Q_{\mathcal{H}}^n$ est donnée par $P_{state}(q_1, \ldots, q_n) = p_0(q_1) f(q_1, q_2) \ldots f(q_{n-1}, q_n)$. La probabilité conditionnelle de générer le mot $w = a_1 \ldots a_n$, étant donnée la suite d'états cachés $(q_1, \ldots, q_n)$ est donnée par $P_{cond}(a_1 \ldots a_n | q_1 \ldots, q_n) = s(q_1, a_1) \ldots s(q_n, a_n)$. Finalement, la probabilité de générer un mot $w = a_1 \ldots a_n$ selon un HMM $\mathcal{H}$ est donnée par

$$P_{\mathcal{H}}(w) = \sum_{(q_1, \ldots, q_n) \in Q_{\mathcal{H}}^n} P_{state}(q_1, \ldots, q_n) \cdot P_{cond}(a_1 \ldots a_n | q_1 \ldots q_n).$$

Lemme 6 (automate probabiliste associé à un modèle de Markov caché)
Pour un HMM $\mathcal{H} =< Q_{\mathcal{H}}, p_0, \mathcal{A}, f, s >$ *donné, nous associons l'automate* $G(\mathcal{H}) =< Q_G, q_0, \mathcal{A}, \rho >$ *de la manière suivante :*

- $Q_G = Q_{\mathcal{H}} \uplus \{q_0\}$, *l'état* $q_0 \notin Q_G$ *est l'état initial de l'automate probabiliste* $G(\mathcal{H})$,
- $\forall q \in Q_{\mathcal{H}}$, $\rho(q_0, a, q) = p_0(q) \cdot s(q, a)$,
- $\forall < q, q' > \in Q_{\mathcal{H}}^2$, $\rho(q, a, q') = f(q, q') \cdot s(q', a)$.

Alors pour tout mot $w \in \mathcal{A}^*$, *la probabilité de* w *selon* $\mathcal{H}$ *est égale à* $\rho(w)$ *selon* $G(\mathcal{M})$.

Il est intéressant de noter que la réciproque du lemme 6 est également vraie :

Lemme 7 (modèle de Markov caché associé à un automate probabiliste)
Soit $G =< Q_G, q_0, \mathcal{A}, \rho >$ *un automate probabiliste non-déterministe. Il existe alors un HMM* $\mathcal{H}(G)$ *avec au plus* $|Q_G| \cdot |\mathcal{A}|$ *états cachés telle que, pour tout mot* $w \in \mathcal{A}^*$, *la probabilité* $P_{\mathcal{H}(G)}(w)$ *est égale à* $\rho(w)$ *selon l'automate probabiliste* G.

Preuve (**modèle de Markov caché associé à un automate probabiliste**) Nous définissons le HMM $\mathcal{H}(G) =< Q_{\mathcal{H}}, p_0, \mathcal{A}, f, s >$ de la manière suivante :

$$
\begin{array}{rrcl}
 & Q_{\mathcal{H}} & = & Q_G \times \mathcal{A}, \\
\forall q \in Q_G \quad \forall a \in \mathcal{A} & p_0(< q, a >) & = & \rho(q_0, a, q), \\
\forall q \in Q_G \quad \forall a, a' \in \mathcal{A}^2 & s(< q, a >, a') & = & \begin{cases} 1 & \text{si } a = a' \\ 0 & \text{sinon.} \end{cases} \\
\forall q, q' \in Q_G^2 \quad \forall a \in \mathcal{A} & f(< q, a >, q') & = & \rho(q, a, q')
\end{array}
$$

Les fonctions f et s ainsi définies vérifient les équations (4.8) et (4.9). $\mathcal{H}(G)$ vérifie de par sa construction le lemme 7. □

D'après les lemmes 6 et 7, les automates probabilistes et les HMM génèrent la même classe de distributions probabilistes sur les mots. Les modèles de Markov sont une sous-classe de cet ensemble qui correspond à une certaine classe d'automates déterministes probabilistes. Il est intéressant de remarquer que la correspondance entre les HMMs et les automates probabilistes est similaire à la correspondance entre les automates de Moore et les machines de Mealy [102].

4.2.2 Automate des graines

Une graine peut être associée à une expression régulière particulière et peut donc être représentée par un automate déterministe. Cette technique a, par exemple, été utilisée par Buhler et coauteurs [23] (voir partie 3.7.7.1). Le calcul de la sensibilité d'une graine espacée classique est réalisé à l'aide d'un automate qui reconnaît l'ensemble des mots détectés par la graine. Nous noterons, pour une graine π donnée, L_π le langage régulier reconnu par π, et S_π un automate déterministe qui reconnaît L_π.

4.3 Calcul de la sensibilité

Nous abordons dans cette partie le calcul de la sensibilité d'une graine à partir des définitions précédemment introduites. La sensibilité est définie comme la probabilité qu'un mot aléatoire sur $\mathcal{A}$, généré selon un certain modèle d'alignement, soit détecté par une graine ou un ensemble de graines donné.

4.3.1 Principe

Une fois le modèle d'alignement (voir parties 4.2.1.1-4.2.1.2) et le modèle de graine (voir partie 4.2.2) établis, il est possible de calculer, à l'aide de l'approche que nous proposons, la sensibilité d'une graine. Prenons une graine ou un ensemble de graines π sur un modèle de graine donné. Nous supposons que le langage L_π des alignements détectés par π est régulier et donc reconnu par un automate déterministe S_π.

Prenons maintenant l'ensemble des alignements cibles, c'est-à-dire le langage L_T (voir partie 4.2.1.1), supposé régulier donc reconnu par un automate T, ainsi qu'un automate probabiliste G servant à affecter une probabilité d'apparition à tout mot du langage L_T. Sous ces hypothèses, la sensibilité de la graine ou de l'ensemble de graines π sera définie par la probabilité conditionnelle $\mathcal{P}_G(\alpha \in L_\pi | \alpha \in L_T)$ qu'un mot α du langage L_T appartienne à L_π :

$$P_G(\alpha \in L_\pi | \alpha \in L_T) = \frac{\mathcal{P}_G(\alpha \in L_T \cap L_\pi)}{\mathcal{P}_G(\alpha \in L_T)}. \tag{4.10}$$

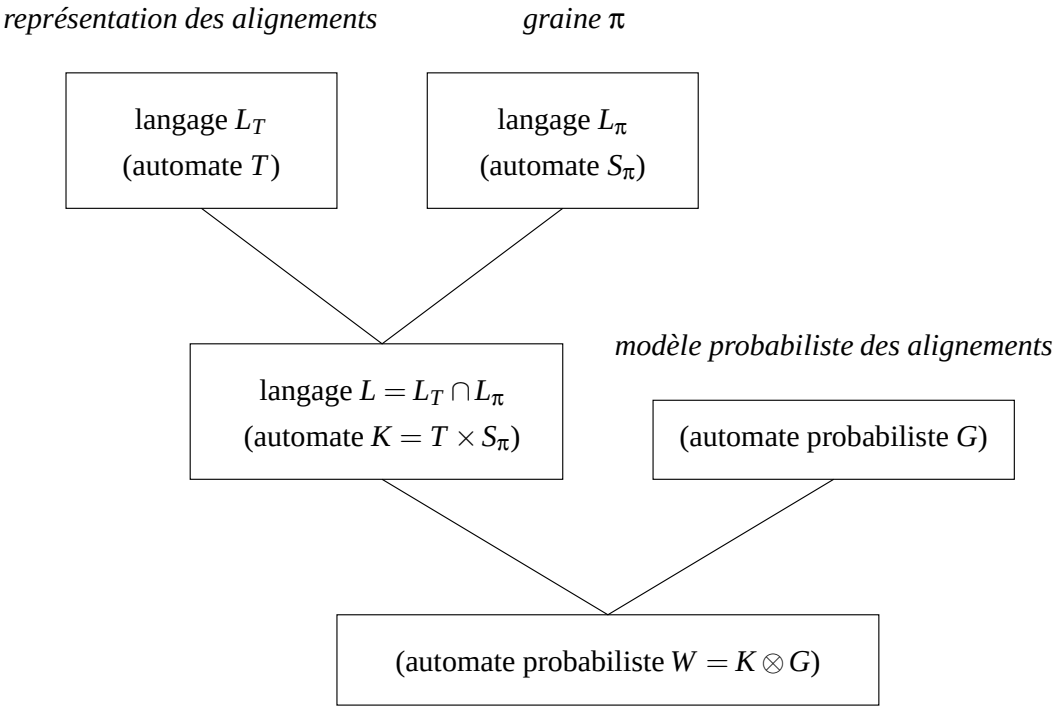


FIG. 4.5 – *Modélisation à l'aide d'automates*

Afin de calculer le numérateur, il est possible de générer un automate reconnaissant $L = L_T \cap L_\pi$, en réalisant le produit des automates T et S_π qui reconnaissent respectivement les

langages réguliers L_T et L_π. Nous appellerons $K =< Q_K, q_K^0, Q_K^F, \mathcal{A}, \psi_K >$ l'automate résultat (voir figure 4.5).

Pour calculer la probabilité $\mathcal{P}_G$ de générer un mot d'un langage régulier (issu par exemple de $L_T \cap L_\pi$), il est nécessaire de réaliser un produit entre un automate probabiliste sans état final G et un automate classique déterministe avec des états finaux K (voir figure 4.5). Ce produit donnera comme résultat un automate probabiliste W. Les transitions posséderont des probabilités associées à celles données par G et les états de cet automate seront finaux s'ils correspondent au produit d'un état final de K avec un état de G.

Le produit $W = K \otimes G$, sera donc défini de la manière suivante.

Définition 16 (produit probabiliste $W = K \otimes G$) *Soit un automate déterministe $K = < Q_K, q_K^0, Q_K^F, \mathcal{A}, \psi_K >$ et un automate probabiliste $G =< Q_G, q_G^0, \mathcal{A}, \rho_G >$ sans état final. Le produit $W = K \otimes G$ est un automate probabiliste $W =< Q_W, q_W^0, Q_W^F, \mathcal{A}, \rho_W >$ tel que :*

- $Q_W = Q_K \times Q_G$,
- $q_W^0 = (q_K^0, q_G^0)$,
- $Q_W^F = \{(q_K, q_G) | q_K \in Q_K^F, q_G \in Q_G\}$,
- $\rho_W((q_K, q_G), a, (q'_K, q'_G)) = \begin{cases} \rho_G(q_G, a, q'_G) & \text{si } \psi_K(q_K, a) = q'_K, \\ 0 & \text{sinon.} \end{cases}$

W sera un automate probabiliste non-déterministe si G est non-déterministe. Pour une transition $< (q_K, q_G), a, (q'_K, q'_G) >$ de l'automate W, on associe sa probabilité $\rho_W\big((q_K, q_G), a, (q'_K, q'_G)\big)$. L'automate W possède la propriété que tous les chemins allant de l'état initial à un état final sont étiquetés par des probabilités.

Définition 17 (chemin complet) *Un chemin de W est dit* complet *si sa source sur le diagramme de transitions est l'état q_W^0 et son puits sur le diagramme de transitions est un état final de W.*

Lemme 8 (probabilité d'un langage L selon un automate probabiliste G) *Étant donnés un automate probabiliste G et un langage L reconnu par un automate déterministe K, et soit $W = K \otimes G$. La probabilité $\mathcal{P}_G(L)$ de générer un mot du langage L est égale à la somme des probabilités de tous les chemins* complets *de W.*

Preuve (probabilité d'un langage L selon un automate probabiliste G) K est un automate déterministe donc les mots du langage L sont en bijection avec les chemins complets de K. Les chemins de G que l'on emprunte en suivant les labels des lettres d'un mot w sont également en bijection avec les ensembles de chemins de W qui acceptent le mot w.

$\mathcal{P}_G(w)$ est défini comme la somme des probabilités des chemins complets de G dont le label est w. Comme chaque chemin correspond à un chemin unique sur W, associé à la même probabilité, alors $\mathcal{P}_G(w)$ est égal à la somme des probabilités des chemins correspondants sur W. □

4.3.2 Algorithme

Nous avons vu au lemme 8 qu'il était possible de calculer la sensibilité d'une graine en considérant la probabilité du langage régulier associé. En effet, ce calcul se ramène à une somme des probabilités des chemins sur un automate produit W. Nous allons désormais le rendre plus explicite au niveau de la formulation et de l'algorithme associé.

Lemme 9 (complexité du calcul de $\mathcal{P}_G(L)$) *Soit L_T un ensemble fini d'alignements cibles sur un alphabet d'alignement $\mathcal{A}$, soit π une graine, et soit $L_\pi \subseteq \mathcal{A}^*$ l'ensemble de tous les alignements détectés par π. Soit $K =< Q_K, q_t^0, Q_K^F, \mathcal{A}, \psi_Q >$ un automate déterministe acyclique reconnaissant le langage $L = L_T \cap L_\pi$, et soit $G =< Q_G, q_G^0, \mathcal{A}, \rho >$ un automate probabiliste associant une distribution probabiliste sur l'ensemble L_T ; Alors $\mathcal{P}_G(L)$ peut être calculé en temps*

$$\mathcal{O}(|Q_G|^2 \cdot |Q_K| \cdot |\mathcal{A}|) \tag{4.11}$$

et en espace

$$\mathcal{O}(|Q_G| \cdot |Q_K|). \tag{4.12}$$

Preuve (complexité du calcul de $\mathcal{P}_G(L)$) Le lemme 8 dit que la probabilité d'un langage régulier L selon un modèle donné par un automate probabiliste G peut être calculée en réalisant la somme des probabilités de tous les chemins complets associés à L dans l'automate W. Comme K est supposé être un automate acyclique, l'automate probabiliste W est également acyclique, et la somme de tous les chemins complets peut être calculée à l'aide d'algorithmes de programmation dynamique [102] classiques sur les graphes acycliques orientés [44]. La complexité en temps est alors proportionnelle au nombre de transitions dans W. W a au plus $|Q_G| \cdot |Q_K|$ états, et pour chaque lettre de $\mathcal{A}$, chaque état possède au plus $|Q_G|$ transitions sortantes. □

Le lemme 8 propose une approche souple pour le calcul de la sensibilité d'une graine, en décomposant les éléments qui interviennent comme modèles, à l'aide de trois automates (voir la figure 4.5) :

- un automate déterministe acyclique T qui représente l'ensemble des alignements cibles sur l'alphabet d'alignement $\mathcal{A}$ (comme tous les mots d'une longueur donnée n, vérifiant par exemple des propriétés supplémentaires comme l'homogénéité).
- un automate probabiliste G (en général non déterministe) qui associe une distribution probabiliste sur l'ensemble des alignements cibles. Cet automate peut par exemple représenter un modèle de Bernoulli, un modèle de Markov d'ordre k, ou un modèle de Markov caché (dans ce cas il sera par exemple non-déterministe).
- un automate déterministe S_π représentant une graine π par l'ensemble des alignements détectés par π.

Une fois que ces automates sont définis, le lemme 9 permet un calcul des probabilités $\mathcal{P}_G(L_T \cap L_\pi)$ et $\mathcal{P}_G(L_T)$ de manière à estimer la sensibilité d'une graine π selon l'équation (4.10).

Il est intéressant de remarquer que si l'automate probabiliste G est déterministe, comme dans le cas d'un modèle de Bernoulli ou de Markov, alors la complexité en temps est en $\mathcal{O}(|Q_G| \cdot |Q_K| \cdot |\mathcal{A}|)$.

En général, la complexité pratique de l'algorithme peut être largement améliorée en minimisant les automates déterministes associés (algorithmes de la section 2.2.3), en particulier lorsque le nombre de graines composant π est multiple.

Cependant le calcul de la probabilité d'une graine n'améliore pas intrinsèquement la sensibilité d'un modèle de graines sur les alignements réels. Il faut pour cela définir de nouveaux modèles de graines, plus expressifs pour prendre en compte les propriétés des séquences biologiques et leur évolution. Mais il doit rester suffisamment simple pour permettre une indexation et une recherche efficace des *hits*, tout en conservant un algorithme rapide pour le calcul de la sensibilité.

Dans la partie suivante, nous considérerons plus spécifiquement l'automate S_π servant à représenter le langage L_π reconnu par une graine, et définirons un nouveau modèle de *graine sous-ensemble* à la fois sensible et dont la taille pour l'automate associé est équivalente à celui des *graines espacées*.

4.4 Graines sous-ensemble

Jusqu'à présent la notion d'erreur dans un alignement était vue comme une propriété binaire : soit les deux lettres d'une similarité étaient les mêmes, soit elles différaient.

Les graines espacées traditionnelles utilisent cette propriété qui permet une implémentation par hachage aisée. Nous avons vu dans l'état de l'art 3.8 d'autres généralisations des graines espacées comme les *graines vecteurs*. Ce modèle de graine proposé dans [16] permet d'utiliser un alphabet $\mathcal{A}$ de taille quelconque, et le modèle de graine permet une définition flexible du critère de *hit*. En contrepartie, les *graines vecteurs* ne sont pas directement indexables, ce qui implique un critère de *hit* plus complexe à évaluer.

Dans la partie 4.4.1, nous considérons un nouveau modèle de graine appelé *graine sous-ensemble*. Ce modèle possède l'avantage de modéliser un filtre sur un alphabet d'alignement $\mathcal{A}$ non nécessairement binaire. Il possède également la propriété de pouvoir modéliser les méthodes d'indexation directe.

Nous proposons ensuite, dans la partie 4.4.2, une méthode de construction d'un automate reconnaissant le langage défini par une *graine sous-ensemble* sur l'alphabet $\mathcal{A}$ des alignements. Cette approche est différente de l'automate de Aho-Corasick, et génère un automate dont le nombre d'états est en $\mathcal{O}(w \cdot 2^{s-w})$, donc indépendant de la taille de l'alphabet (s et w désignent respectivement l'étendue et le nombre de symboles # de la graine). Ceci implique que la sensibilité de *graines espacées* classiques ou de *graines sous-ensemble* peut être calculée avec la même complexité dans le cadre proposé dans la partie 4.2.

Enfin nous présentons dans la partie expérimentale (partie 4.5) que, même sur un alphabet $\mathcal{A}$ binaire, notre automate génère un plus petit nombre d'états en pratique.

4.4.1 Modèle

Nous présentons dans cette partie le modèle des graines sous-ensemble et définissons le critère de *hit* d'une graine sous-ensemble. Nous définissons également certaines quantités associées à une graine sous-ensemble.

L'alphabet des séquences Σ est supposé fixé. Soit $\mathcal{A}$ l'alphabet des alignements, défini par une partition de $\Sigma \times \Sigma$. Nous supposerons que $\mathcal{A}$ possède la lettre `1`, synonyme de *match* : il s'agit de la partie composée des éléments $a \times a$, pour tout $a \in \Sigma$.

Nous allons définir le principe de graine sous-ensemble sur un alphabet d'alignement $\mathcal{A}$.

Définition 18 (alphabet sous-ensemble et graine sous-ensemble) *Une graine sous-ensemble est définie par un mot sur un alphabet $\mathcal{B}$, qui possède les propriétés suivantes.*

- *les lettres de $\mathcal{B}$ représentent des sous-ensembles de l'alphabet d'alignement $\mathcal{A}$ qui contiennent la lettre* `1` *($\mathcal{B} \subseteq \{\mathcal{P} \cup \{\texttt{1}\}\}$ avec $\mathcal{P} \subseteq \mathcal{A}$).*
- *$\mathcal{B}$ possède une lettre # qui représente le sous-ensemble* $\{\texttt{1}\}$.

Définition 19 (critère de *hit* d'une graine sous-ensemble) *Une graine sous-ensemble représentée par un mot $b_1 b_2 \dots b_s \in \mathcal{B}^s$ détecte un alignement $\alpha_1 \alpha_2 \dots \alpha_n \in \mathcal{A}^n$ si et seulement s'il existe i ($0 \leq i \leq n - s$) tel que $\forall j \in [1..s]$, $\alpha_{i+j} \in b_j$.*

Nous donnons ici quelques propriétés :

Définition 20 (#-poids d'une graine) *Le* #-poids *d'une graine sous-ensemble π est le nombre de lettres # contenues dans π.*

De manière à éclaircir le principe, nous prenons le cas des graines à transitions du logiciel YASS (présenté à la partie 4.6.1) :

EXEMPLE 24 : *graines sous-ensemble, cas des graines à transitions*
Soit l'alphabet des alignements $\mathcal{A} = \{\mathtt{1}, \mathtt{h}, \mathtt{0}\}$, où les lettres $\mathtt{1}, \mathtt{h}, \mathtt{0}$ représentent respectivement un match, une substitution de type transition, ou une substitution de type transversion (voir partie 1.4). L'alphabet des graines est défini par $\mathcal{B} = \{\#, @, -\}$ représentant les sous-ensembles $\{\mathtt{1}\}$, $\{\mathtt{1}, \mathtt{h}\}$, et $\{\mathtt{1}, \mathtt{h}, \mathtt{0}\}$. Ainsi la graine $\pi = \#@-\#$ détecte l'alignement `10h1h1101` aux positions 4 et 6. Le #-poids de π est de 2.

Contrairement au poids des graines espacées classiques, le #-poids ne peut servir de mesure pour la sélectivité des graines. En effet, dans l'exemple 24, les lettres @ devraient avoir un poids d'un demi, donnant à la graine π un poids de 2.5. En effet, si pour tout $a \in \mathcal{A}$ on considère l'évènement $a \in \#$, sa probabilité sur deux séquences i.i.d est de $\frac{1}{4}$. L'information alors apportée est donc de deux bits pour un symbole #. En revanche, selon les mêmes hypothèses, la probabilité $a \in @$ est de $\frac{1}{2}$, soit un bit d'information. En conséquence, le poids d'un symbole @ sera deux fois moindre.

4.4.2 Automate associé

Nous proposons dans cette partie une définition d'un automate compact pour le modèle des *graines sous-ensemble* et un algorithme de construction rapide de ce dernier.

4.4.2.1 Définition

Soit un alphabet d'alignement $\mathcal{A}$, un alphabet des graines $\mathcal{B}$ défini d'après la définition 18, et une graine $\pi = \pi_1\pi_2 \dots \pi_s \in \mathcal{B}^*$ d'étendue s et de #-poids w.

Définition 21 (ensemble des préfixes restreints R_π) *Nous définissons R_π comme un ensemble d'entiers : cet ensemble représente les longueurs de tous les préfixes de π dont la dernière lettre n'est pas de type #. On appellera cet ensemble R_π* ensemble des préfixes restreints. *La taille de cet ensemble est $|R_\pi| = s - w$.*

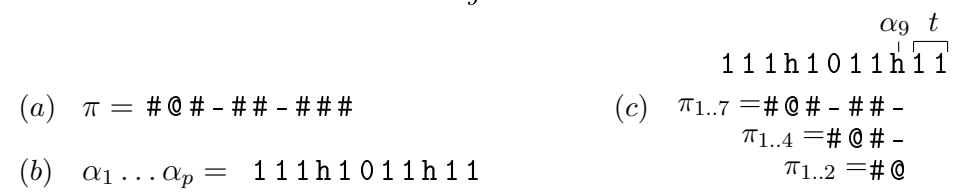
Nous allons désormais définir un automate $S_\pi =< Q, q_0, Q_f, \mathcal{A}, \psi : Q \times \mathcal{A} \to Q >$ qui reconnaisse l'ensemble des alignements détectés par la graine π. Nous décrivons tout d'abord la représentation des états, pour ensuite nous intéresser à la fonction de transition associée.

Définition 22 (états de l'automate S_π) *Les états Q de l'automate S_π sont définis comme des paires $< X, t >$ avec*
- $t \in [0..s(\pi)]$,
- $X \subseteq R_\pi$.

Les états vérifient l'invariant suivant : supposons que l'automate ait lu un préfixe $\alpha_1 \dots \alpha_p$ d'un alignement, pour arriver à l'état actuel $< X, t >$. Alors
- *t désigne la longueur du plus long suffixe de $\alpha_1 \dots \alpha_p$ de la forme $\mathtt{1}^i$, $i \leq s$.*
- *X est l'ensemble des longueurs de tous les préfixes restreints $r_i \in R_\pi$ telles que $\pi_1 \cdots \pi_{r_i}$ détecte le suffixe de $\alpha_1 \cdots \alpha_{p-t}$.*

De manière à illustrer cette définition, un exemple est nécessaire.

EXEMPLE 25 : *états de l'automate d'une graine sous-ensemble*



Nous reprenons le cadre des définitions introduites dans l'exemple 24. Soit une graine π et un préfixe d'alignement $\alpha_1 \dots \alpha_p$ de longueur $p = 11$ (éléments donnés dans les parties (a) et (b) de la figure ci-dessus).
La longueur t du plus long suffixe $\mathtt{1}^*$ de $\alpha_1 \dots \alpha_p$ est 2. La dernière lettre différente de `1` est donc $\alpha_{p-t} = \alpha_9 = \mathtt{h}$. L'ensemble R_π des longueurs des préfixes restreints dans π est $\{2, 4, 7\}$, ce qui implique que π possède 3 préfixes possibles terminant aux positions de R_π (partie (c)). Les préfixes $\pi_{1..2}$ et $\pi_{1..7}$ détectent les suffixes respectifs de $\alpha_1\alpha_2 \dots \alpha_9$, mais le préfixe $\pi_{1..4}$ ne détecte pas. Ainsi l'état courant après lecture de $\alpha_1\alpha_2 \dots \alpha_p$ sera $< X = \{2, 7\}, t = 2 >$.

Nous avons défini les états courants, il nous reste à spécifier l'état initial et les états finaux. L'état initial q_0 de S_π sera naturellement défini par $< \varnothing, 0 >$. Les états finaux sont ceux pour lesquels le suffixe de $\alpha_1 \dots \alpha_p$ est détecté par la graine : il s'agit de tous les états $q =< X, t >$ tels que $max\{X\} + t = s$. Bien entendu, il est possible de fusionner les états finaux comme pour l'algorithme proposé dans la partie 3.7.7.1.

Il nous reste à définir la fonction de transition afin de conserver l'invariant lors de la lecture d'une nouvelle lettre.

Définition 23 (fonction de transition de l'automate S_π) *La fonction de transition de l'automate S_π est définie de la manière suivante :*

si q est un état final, il est rendu absorbant ($\forall a \in \mathcal{A}$, $\psi(q, a) = q$).

si $q =< X, t >$ n'est pas un état final alors :

- *si $a = \mathtt{1}$ alors $\psi(q, a) =< X, t + 1 >$,*
- *sinon $\psi(q, a) =< X_U \cup X_V, 0 >$ avec*

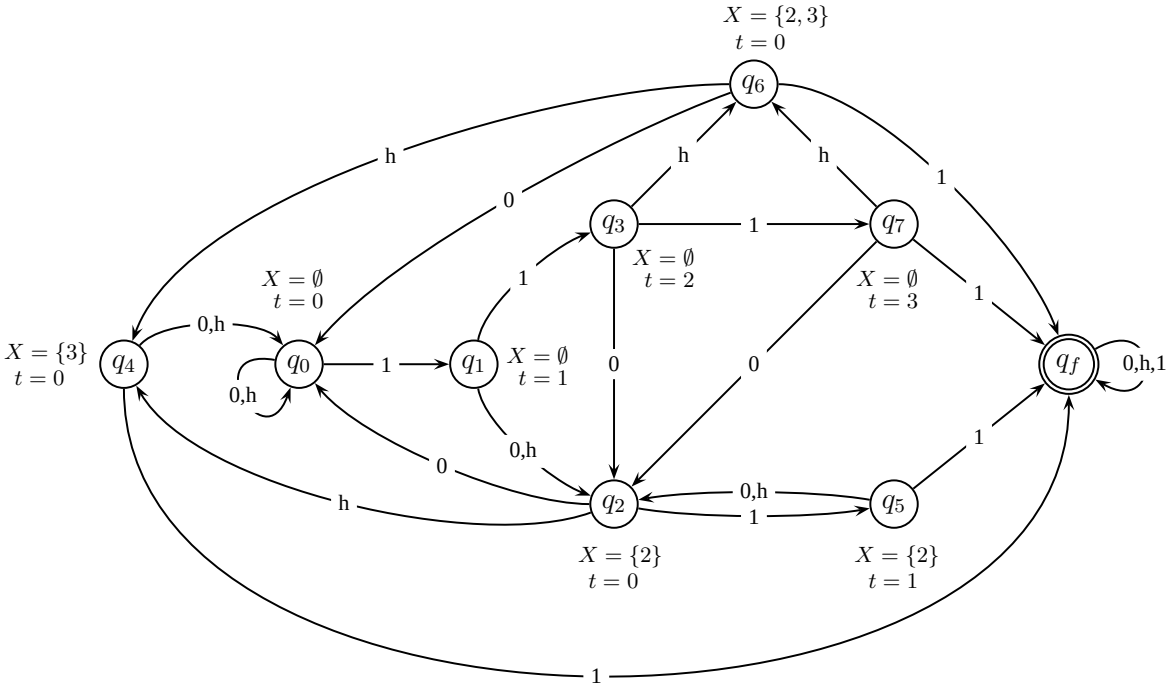
  (i) $X_U = \{x | x \leq t + 1 \text{ et } a \in \pi_x\}$

  (ii) $X_V = \{x + t + 1 | x \in X \text{ et } a \in \pi_{x+t+1}\}$

Lemme 10 (reconnaissance de L_π) *L'automate S_π ainsi défini accepte l'ensemble des alignements L_π que la graine π détecte.*

Preuve (**reconnaissance de L_π**) Nous pouvons vérifier par induction que la condition d'invariant est préservée par la fonction de transition ψ. Dans le cas de l'état initial $< \varnothing, 0 >$ qui est atteint entre autre par la lecture du mot vide, cette condition est vérifiée. Lors de la lecture d'une lettre `1`, la condition d'invariant est respectée de manière immédiate, d'après la définition de la fonction de transition. Seul le dernier cas ($\psi(q, a) =< X_U \cup X_V, 0 >$) est plus difficile à comprendre et analyser. X_U désigne l'ensemble des préfixes restreints qui ne couvrent pas la dernière lettre différente de `1` (la longueur du préfixe est alors inférieure à t), X_V représente l'ensemble des préfixes restreints qui couvrent la dernière substitution (de longueur supérieure ou égale à t). L'union de ces deux ensembles disjoints donne alors l'ensemble des préfixes remis à jour après l'ajout d'une lettre différente de `1`. □

EXEMPLE 26 : *automate S_π de la graine sous-ensemble #−@#*



Nous reprenons le cadre des définitions introduites dans les deux exemples 24 et 25. Soit la graine $\pi = \#-@\#$. L'ensemble des longueurs des préfixes restreints de π est $R_\pi = \{2,3\}$. Nous en déduisons, d'après la définition 22, que les états non-finaux potentiels de S_π sont les 8 paires $< X,t >= \{ < \varnothing,0 >, < \varnothing,1 >, < \varnothing,2 >, < \varnothing,3 >, < \{2\},0 >, < \{2\},1 >, < \{3\},0 >, < \{2,3\},0 > \}$. Tous ces états sont accessibles et donc sont présents dans S_π. Les transitions entre états sont calculées à partir de la définition 23.

Lemme 11 (nombre d'états de l'automate S_π) *Le nombre d'états de l'automate S_π est au plus $(w+1)2^{s-w}$.*

Preuve (nombre d'états de l'automate S_π) R_π est l'ensemble des longueurs des préfixes restreints $R_\pi = \{r_1, r_2, \ldots, r_{s-w}\}$ avec $r_1 < r_2 \cdots < r_{s-w}$. Soit Q_i l'ensemble des états non-finaux $< X,t >$ dont le plus long préfixe restreint appartenant à X est de taille r_i ($i \in [1..s-w]$).

Remarquons que pour un état $q =< X,t >\in Q_i$, il existe au plus 2^{i-1} choix possibles pour X et $s - r_i$ choix possibles pour t, puisque $max\{X\} + t < s$.

Ainsi

$$|Q_i| \leq 2^{i-1}(s - r_i) \leq 2^{i-1}(s-i), \text{ et} \tag{4.13}$$

$$\sum_{i=1}^{s-w} |Q_i| \leq \sum_{i=1}^{s-w} 2^{i-1}(s-i) = (w+1)2^{s-w} - s - 1. \tag{4.14}$$

En plus des états Q_i, Q possède également s états de la forme $< \varnothing, t >$ (avec $t \in [0..s-1]$) ainsi qu'un état final. Donc $|Q| \leq (w+1)2^{s-w}$. □

Remarquons que si une graine π possède comme première lettre un # (ce qui est le cas pour des graines espacées classiques), alors $r_i \geq i+1$, $i \in [1..s-w]$, et la borne de l'équation (4.13) se réduit à $2^{i-1}(s-i-1)$: le nombre d'états est alors le même que pour l'automate de la section 3.7.7.1. Une propriété intéressante de cet automate est qu'il atteint la borne maximale du nombre d'états.

Lemme 12 (borne supérieure sur le nombre d'états de S_π) *Soit, pour un entier j donné, la graine $\# \cdot -^j \cdot \#$: L'automate associé S_π sera alors minimal, en ce sens que tout état est accessible depuis l'état initial q_0, et que deux états non-finaux sont non-équivalents.*

Preuve (borne supérieure sur le nombre d'états de S_π) Nous montrons que l'automate S_π est réduit, c'est-à-dire qu'il vérifie les propriétés suivantes

(i) tout état est accessible depuis l'état initial $< \varnothing, 0 >$,

(ii) deux états non-finaux q, q' sont non-équivalents, i.e. il existe un mot α tel que seul un des états $\psi(q,\alpha), \psi(q',\alpha)$ est un état final.

preuve de (i) soit $q =< X, t >$ un état de l'automate S_π, et soit $X = \{x_1, \ldots, x_k\}$ avec $x_1 < \cdots < x_k$. $x_k + t < j+2$. Soit l'alignement $\alpha \in \mathcal{A}^{x_k}$ avec $\mathcal{A} = \{0,1\}$ tel que pour tout $i \in [1, x_k]$, $\alpha_i = \mathtt{1}$ si et seulement si $\exists j \in [1,k]$, $\quad i = x_k - x_j + 1$. Remarquons que comme $\pi_1 = \#$, alors $\mathtt{1} \notin X$ et $\alpha_{x_k} = 0$. En conclusion $\psi(< \varnothing, 0 >, \alpha \cdot 1^t) = q$.

preuve de (ii) soit $q_1 =< X_1, t_1 >$ et $q_2 =< X_2, t_2 >$ deux états non-finaux de S_π. Soit $X_1 = \{y_1, \ldots, y_a\}, X_2 = \{z_1, \ldots, z_b\}$, et $y_1 < \cdots < y_a$, $z_1 < \cdots < z_b$.
Supposons que $max\{X_1\} + t_1 > max\{X_2\} + t_2$ et soit $d = (j+2) - (max\{X_1\} + t_1)$. Alors $\psi(q_1, 1^d)$ est un état final mais pas $\psi(q_2, 1^d)$. Maintenant, supposons que $max\{X_1\} + t_1 = max\{X_2\} + t_2$. Pour un ensemble $X \subseteq \{1, \ldots, j+1\}$ et un entier t, soit $X\{t\} = \{v + t | v \in X \text{ et } v + t < j+2\}$. Soit $g = max\{v | (v + t_1 \in X_1 \text{ et } v + t_2 \notin X_2) \text{ ou } (v + t_2 \in X_2 \text{ et } v + t_1 \notin X_1)\}$ et soit $d = j + 1 - g$. Alors $\psi(q_1, 0^d \cdot 1)$ est un état final et pas $\psi(q_2, 0^d \cdot 1)$, ou inversement.

□

Nous avons proposé une définition d'un automate S_π compact pour une graine sous-ensemble π et donné son principe de construction. Cet automate est compact ce qui permet de l'utiliser sans nécessairement le minimiser. Cependant pour que le calcul de la sensibilité soit suffisamment rapide, une implémentation efficace de la construction de ce dernier est requise. En particulier, il faut que chaque état, et surtout chaque transition, puisse être calculé en temps constant. Nous allons dans la partie suivante donner plus explicitement l'algorithme de construction qui permet de respecter ces deux conditions.

4.4.2.2 Construction

Nous explicitons dans cette partie une implémentation de l'automate et donnons un algorithme de construction efficace. Il est assez facile de vérifier qu'une génération de la table de transitions peut être faite en temps $\mathcal{O}\big((s-w) \cdot w \cdot 2^{s-w} \cdot |\mathcal{A}|\big)$. Si l'on suppose que le nombre maximal d'états $w \cdot 2^{s-w}$ peut être encodé sur un entier, ce qui est une hypothèse raisonnable, alors un algorithme plus complexe atteint une borne en temps de $\mathcal{O}(w \cdot 2^{s-w} \cdot |\mathcal{A}|)$.

Soit une graine π de #-poids w et d'étendue s. Nous souhaitons calculer l'automate S_π. Nous supposerons que la table de transitions est contenue dans un tableau dont chaque case représente un état $< X, t >$ et ses transitions sortantes associées.

Nous allons considérer deux problèmes :

- le premier consiste à calculer, pour tout état $q =< X, t >$ non-final, un indice $Ind(q)$ dans le tableau.
- le deuxième consiste à évaluer rapidement, pour un état q donné et une lettre $a \in \mathcal{A}$, la fonction de transition $\psi(q, a)$.

Encodage des indices Le principe de calcul des indices reste le même que celui utilisé pour l'énumération du nombre d'états faite lors de la preuve du lemme 11. Pour une graine sous-ensemble π, soit $R_\pi = \{r_1, \ldots, r_{s-w}\}$ l'ensemble des longueurs des préfixes restreints ($r_1 < r_2 < \cdots < r_{s-w}$) . Pour $X \subseteq R_\pi$, nous définissons un vecteur binaire $v(X) = v_1 \ldots v_r \in \{0,1\}^{s-w}$ tel que $v_i = 1$ si et seulement si $r_i \in X$.

On représentera $v(X)$ sous la forme d'un encodage entier naturel : $n(X)$ est l'entier correspondant à la représentation binaire de $v(X)$ c'est-à-dire

$$n(X) = \sum_{j=1}^{s-w} 2^{j-1} \cdot v_j.$$

Soit $p(t) = max\{p \mid r_p < s - t\}$. On remarquera que, pour tout état non-final $< X, t >$, $X \subseteq \{r_1, \ldots, r_{p(t)}\}$, et donc que $n(X) < 2^{p(t)}$.

Il est donc possible de donner un indice unique pour un état $\{< X, t >\}$ à l'aide de la fonction suivante :

$$Ind(< X, t >) = n(X) + 2^{p(t)}.$$

En conséquence, la taille maximale du tableau sera de $(w + 1) \cdot 2^{s-w}$.

Calcul de la fonction de transition $\psi(q, a)$ Le calcul de la fonction de transition $\psi(q, a)$ repose sur deux principes qui accélèrent l'implémentation :

- un pré-calcul de certains éléments du résultat est réalisé.
- les valeurs $\psi(q', a)$ pour certains états q' sont réutilisées afin de calculer $\psi(q, a)$ pour d'autres états q.

Nous nous intéressons au deuxième problème, et donnons les dépendances entre états q et q'. Soit $q =< X, t >$ un état non-final accessible de S_π, avec $X = \{x_1, \ldots, x_k\}$ ($x_1 < x_2 \cdots < x_k$). Soit $X' = X \setminus \{x_k\} = \{x_1, \ldots, x_{k-1}\}$ et $q' =< X', t >$. Nous montrons dans un premier temps que q' existe (est accessible) et que sa fonction de transition a été calculée antérieurement à celle de q.

Lemme 13 (précédence et accessibilité des états de l'automate S_π) *Si $q =< X, t >$ est accessible, alors $q' =< X', t >$ est accessible et sa fonction de transition a été calculée auparavant.*

Preuve (précédence et accessibilité des états de l'automate S_π) Montrons d'abord que $< X', t >$ est accessible. Il est assez facile de voir que si $< X, t >$ est accessible, alors $< X, 0 >$ l'est également à cause de la définition de la fonction de transition pour t positif. Il est donc possible de trouver une séquence $\alpha \in \mathcal{A}^{r_k}$ telle que $\forall i \in [1..s-w]$, $r_i \in X$ si et seulement si le préfixe associé $\pi_1 \cdots \pi_{r_i}$ détecte $\alpha_{r_k - r_i + 1} \cdots \alpha_{r_k}$. Si l'on prend alors un préfixe de α, $\alpha' = \alpha_{r_k - r_{k-1} + 1} \cdots \alpha_{r_k}$, la séquence associée permet d'atteindre l'état $< X', 0 >$. On en déduit finalement que s'il existe un mot $\alpha \cdot 1^t$ pour atteindre l'état $< X, t >$, il existe également un mot $\alpha' \cdot 1^t$ pour atteindre l'état $< X', t >$. Comme $|\alpha' \cdot 1^t| < |\alpha \cdot 1^t|$, un parcours en largeur d'abord des états de S_π calculera toujours l'état $< X', t >$ avant $< X, t >$. □

Nous savons qu'il est possible d'utiliser des états antérieurs dans le processus de calcul de la fonction $\psi(< X, t >, a)$. Nous montrons désormais comment obtenir les valeurs de $\psi(< X, t >, a)$. Ce processus de calcul est réalisé par l'algorithme 9, que nous expliquons ici.

Un état q sera représenté par un triplet $q = \langle X, k_X, t\rangle$, avec $k_X = max\{i | r_i \in X\}$.

- Si $a = 1$, la fonction de transition $\psi(q, a)$ peut être calculée en temps constant à cause de sa définition (partie a. de l'algorithme 9).
- Si $a \neq 1$, l'algorithme doit
  1. retrouver la position de q' dans la table connaissant $q = \langle X, k_X, t\rangle$ (partie c. de l'algorithme 9),
  2. calculer la valeur de $\psi\big(\langle X, k_X, t\rangle, a\big)$ en utilisant celle de $\psi\big(\langle X', k_{X'}, t\rangle, a\big)$. (partie d. de l'algorithme 9).

Nous décrivons et proposons une solution pour les problèmes 1. et 2. dans la partie suivante :

1. Remarquons que $Ind\big(\langle X, k_X, t\rangle\big) = Ind\big(\langle X', k_{X'}, t\rangle\big) - 2^{k_X}$, ce qui peut être calculé en temps constant puisque k_X est mémorisé de manière explicite pour chaque état.

2. Soit

$$V_X(k, t, a \neq 1) \quad = \quad \begin{cases} r_i & \text{si } r_i = r_k + t + 1 \text{ et } \ a \in \pi_{r_i} \\ \varnothing & \text{sinon} \end{cases}$$

et

$$V_k(k, t, a \neq 1) \quad = \quad \begin{cases} i & \text{si } r_i = r_k + t + 1 \text{ et } \ a \in \pi_{r_i} \\ 0 & \text{sinon} \end{cases}$$

Les fonctions $V_X(k, t, a)$ et $V_k(k, t, a)$ peuvent être pré-calculées en temps et espace $\mathcal{O}(|\mathcal{A}| \cdot s^2)$ (partie d. de l'algorithme 8). Soit $\psi(\langle X, k_X, t\rangle, a) = \langle Y, k_Y, 0\rangle$ et $\psi(\langle X', k_{X'}, t\rangle, a) = \langle Y', k_{Y'}, 0\rangle$. L'ensemble Y diffère de Y' par au plus un élément. Cet élément peut être calculé en temps constant en utilisant les fonctions V_X, V_k. En effet $Y = Y' \cup V_X(k_X, t, a)$ et $k_Y = max(k_{Y'}, V_k(k_X, t, a))$.

Un cas limite apparaît lorsque $X = \varnothing$. (partie b. de l'algorithme 9). Il est dans ce cas possible de pré-calculer deux fonctions U_X, U_k (partie c. de l'algorithme 8) définies par :

$$\begin{aligned} U_X(t, a \neq 1) &= \cup\{x | x \leq t + 1 \text{ et } a \in \pi_x\} \\ U_k(t, a \neq 1) &= max\{x | x \leq t + 1 \text{ et } a \in \pi_x\} \end{aligned}$$

Ce qui conduit au lemme 14 et 15.

Lemme 14 (calcul de la fonction de transition $\psi(q, a)$) *La fonction de transition $\psi(q, a)$ peut être calculée en temps constant pour chaque état accessible q et chaque $a \in \mathcal{A}$.*

Lemme 15 (calcul de la table de transitions de S_π)
La table de transitions de l'automate S_π peut être construite en temps proportionnel à sa taille, qui est en $\mathcal{O}(w \cdot 2^{s-w} \cdot |\mathcal{A}|)$.

Algorithme 8 : *pré-calcul de S_π*

Données : une graine π d'étendue s, #-poids w, et nombre de symboles non-# $r = s - w$
Résultat : fonctions p, U_X, U_k et V_X, V_k utilisées dans le calcul de S_π

```
   /* pré-calcul des longueurs des préfixes restreints R */
   R[0] ← 0;
   r ← 0;
   pour i ← 1 à s faire
a      si π_i ≠ # alors
           r ← r + 1;
           R[r] ← i;
   /* pré-calcul de p(t) */
   p[0] ← 0;
   c ← r;
   pour t ← 1 à s faire
b      si R[c] = s − t − 1 alors
           c ← c − 1;
       p[t] ← p[t − 1] + 2^c;
   /* pré-calcul de U_X et U_k */
   pour t ← 0 à s et a ∈ A faire
       U_X[t][a] ← 0, U_k[t][a] ← 0;
   pour t ← 0 à s faire
c      i ← 1;
       tant que i ≤ r et R[i] ≤ t faire
           pour a ∈ A faire
               si a ∈ π_{R[i]} alors
                   si t ≠ 0 alors
                       U_X[t][a] ← U_X[t − 1][a] + 2^{i−1};
                   sinon
                       U_X[t][a] ← 2^{i−1};
                   U_k[t][a] ← i
           i ← i + 1;
   /* pré-calcul de V_X et V_k */
   pour t ← 0 à s et k ← 0 à r et a ∈ A faire
       V_X[k][t][a] ← 0, V_k[k][t][a] ← 0;
   pour i ← 1 à r faire
d      pour a ∈ A faire
           si a ∈ π_{R[i]} alors
               pour k ← 1 à i faire
                   t ← R[i] − R[k] − 1;
                   V_X[k][t][a] ← 2^{i−1};
                   V_k[k][t][a] ← i;
```

Algorithme 9 : *calcul de* S_π

Données : une graine π d'étendue s, #-poids w, et nombre de symbole non-# $r = s - w$
Résultat : un automate $S_\pi =< Q, q_0, q_F, \mathcal{A}, \psi >$
$Q.ajoute(q_F)$;
$q_0 \leftarrow \langle X = \varnothing, k = 0, t = 0\rangle$;
$Q.ajoute(q_0)$;
$queue.ajoute(q_0)$;
tant que $queue \neq \varnothing$ **faire**
  $\langle X, k_X, t_X\rangle = queue.enleve()$;
  pour $a \in \mathcal{A}$ **faire**
    /* *calcul de* $\psi(< X, t_X >, a) = \langle Y, k_Y, t_Y\rangle$ */
    si $a = 1$ **alors**
      $t_Y \leftarrow t_X + 1$;
a $k_Y \leftarrow k_X$;
      $Y \leftarrow X$;
    sinon
      si $X = \varnothing$ **alors**
b $Y \leftarrow U_X(t_X, a)$;
        $k_Y \leftarrow U_k(t_X, a)$;
      sinon
        /* *utiliser* $\psi(< X', t_{X'} >, a)$ ... */
c $X' \leftarrow X \setminus \{l_{k_X}\}$;
        $\langle Y', k_{Y'}, t_{Y'}\rangle \leftarrow \psi(< X', t >, a)$;
        /* ... *pour calculer* $\psi(< X, t_X >, a)$ */
d $k_Y \leftarrow max\big(k_{Y'}, V_k(k_X, t_X, a)\big)$;
        $Y \leftarrow Y' \cup V_X(k_X, t_X, a)$;
      $t_Y \leftarrow 0$;
    si $R[k_Y] + t_Y \geq s$ **alors**
      /* $< Y, t_Y >$ *est un état final* */
      $\psi(< X, t_X >, a) \leftarrow q_F$;
    sinon
      si $\langle Y, k_Y, t_Y\rangle \notin Q$ **alors**
        $Q.ajoute(\langle Y, k_Y, t_Y\rangle)$;
        $queue.ajoute(\langle Y, k_Y, t_Y\rangle)$;
      $\psi(< X, t_X >, a) \leftarrow \langle Y, k_Y, t_Y\rangle$;

Nous montrons dans la partie suivante qu'en pratique, l'automate S_π est très compact. Il est ainsi possible, en utilisant l'algorithme de calcul de la partie 4.2 en combinaison avec S_π, d'obtenir une méthode efficace pour le calcul des graines sous-ensemble ainsi que des graines espacées classiques.

4.5 Expériences

Plusieurs expériences ont été menées, à la fois sur les automates et sur le modèle des graines sous-ensemble. Le modèle utilisé concerne la recherche de similarités sur l'ADN : l'alphabet d'alignement $\mathcal{A}$ est $\{1, h, 0\}$ et représente les substitutions de type { match, transition, transversion}. L'alphabet des graines sous-ensemble $\mathcal{B}$ est défini par $\{\#, @, -\}$, avec $\# = \{1\}, @ = \{1, h\}, - = \{1, h, 0\}$. Le poids d'une graine est calculé en donnant respectivement les poids 1, 0.5 et 0 aux lettres #, @ et −.

4.5.1 Taille de l'automate

Nous avons comparé la taille de l'automate des graines sous-ensemble S_π avec celui de Aho-Corasick [2], à la fois pour des graines espacées traditionnelles (alphabet des alignements $\mathcal{A}$ binaire) et pour des graines sous-ensemble pour l'ADN. L'automate de Aho-Corasick a été construit selon la méthode utilisée dans [23]. Dans le cas des graines sous-ensemble, une généralisation directe a été faite : l'automate de Aho-Corasick a été construit pour l'ensemble des alignements détectés par la graine.

Les tables 4.2(a) et 4.2(b) donnent les résultats respectifs pour des graines espacées classiques et des graines sous-ensemble. Pour chaque poids w, nous avons énuméré les graines et calculé le nombre moyen d'états (μ_{taille}) de l'automate de Aho-Corasick et de notre automate S_π. Le ratio ρ avec le nombre moyen d'états pour l'automate minimisé est également donné. Les graines énumérées sont toutes les graines espacées d'étendue au plus $w + 8$, ainsi que les graines sous-ensemble d'étendue au plus $w + 5$ avec deux symboles de type @. Il est intéressant de remarquer

(a) Graines espacées	Aho-Corasick		S_π		Minimisé
w	μ_{taille}	ρ	μ_{taille}	ρ	μ_{taille}
9	345.94	3.06	146.28	1.29	113.21
10	380.90	3.16	155.11	1.29	120.61
11	415.37	3.25	163.81	1.28	127.62
12	449.47	3.33	172.38	1.28	134.91
13	483.27	3.41	180.89	1.28	141.84

(b) Graines sous-ensemble	Aho-Corasick		S_π		Minimisé
w	μ_{taille}	ρ	μ_{taille}	ρ	μ_{taille}
9	1900.65	15.97	167.63	1.41	119.00
10	2103.99	16.50	177.92	1.40	127.49
11	2306.32	16.96	188.05	1.38	135.95
12	2507.85	17.42	198.12	1.38	144.00
13	2709.01	17.78	208.10	1.37	152.29

TAB. 4.2 – *Comparaison du nombre moyen d'états de l'automate de Aho-Corasick , de l'automate S_π et de l'automate minimisé.*

que l'automate S_π est plus compact, non seulement lorsque l'alphabet des alignements $\mathcal{A}$ n'est pas binaire (ce qui était attendu), mais également dans le cas binaire (voir la table 4.2(a)). Dans tous les cas, pour une graine donnée, il est possible de définir une correspondance surjective des états de l'automate de Aho-Corasick vers les états de S_π. Ceci implique que S_π a *toujours* une

taille inférieure ou égale à celle de l'automate de Aho-Corasick.

4.5.2 Conception de graines

Dans cette partie, nous allons sélectionner les graines espacées et les graines sous-ensemble les plus sensibles à l'aide de l'approche proposée ci-dessus. L'ensemble des alignements cibles est constitué de tous les alignements de longueur 64 sur l'alphabet $\mathcal{A} = \{\mathtt{1}, \mathtt{h}, \mathtt{0}\}$. Quatre automates probabilistes ont été considérés (ces derniers sont semblables à ceux proposés dans [15]). Il s'agit des modèles suivants :

- B : modèle de Bernoulli.
- $DT1$: automate probabiliste déterministe donnant les probabilités de $\{\mathtt{1}, \mathtt{h}, \mathtt{0}\}$ à chaque position d'un codon. Il est équivalent à 3 modèles de Bernoulli phasés et est une extension du modèle $M^{(3)}$ proposé dans [15] pour un alphabet à 3 lettres.
- $DT2$: automate probabiliste déterministe donnant les probabilités de chacun des 27 codons pris dans $\{\mathtt{1}, \mathtt{h}, \mathtt{0}\}^3$. Il s'agit d'une extension du modèle $M^{(8)}$ proposé dans [15] pour un alphabet de trois lettres.
- NT : automate probabiliste non-déterministe qui combine quatre copies du modèle $DT2$ pour 4 niveaux de conservation des codons. Ces quatre copies du modèle $DT2$ sont couplées à un modèle de Markov Caché à 4 états. Le modèle est similaire à celui proposé dans [15].

Les quatre modèles ont été appris sur des alignements provenant d'une comparaison de 40 génomes de bactéries.

Pour chacun des modèles donnés, et pour chaque poids de graine w dans l'intervalle [9..12], nous avons recherché la meilleure graine pour le modèle des graines espacées, et pour le modèle des graines sous-ensemble en autorisant deux symboles @ par graine sous-ensemble. Les graines espacées classiques ont été énumérées de manière exhaustive jusqu'à une certaine étendue, et pour chaque graine, la sensibilité a été calculée en utilisant l'algorithme de la partie 4.2 et la construction de l'automate S_π de la partie 4.4. Les expériences ont été réalisées à l'aide du logiciel HEDERA décrit à la partie 4.6.2. Pour une graine, le calcul prend entre 10 et 500 ms sur un Pentium™ IV de fréquence 2.4Ghz, selon le poids et l'étendue de la graine, et le modèle utilisé. Pour chaque expérience, la meilleure graine trouvée a été gardée. L'ensemble des résultats est présenté dans les tables 4.3–4.6.

Dans tous les cas, les graines sous-ensemble sont plus sensibles que les graines espacées classiques. La sensibilité est améliorée jusqu'à 0.04 ce qui est assez remarquable. En outre, le gain observé en utilisant des graines sous-ensemble augmente de manière significative lorsque la probabilité d'une transition est plus grande que celle d'une transversion, ce qui est très souvent le cas dans des génomes proches.

w	graine espacée	sensibilité	graine sous-ensemble	sensibilité
9	`###---#-#-##-##`	0.4183	`###-#--#@#-@##`	0.4443
10	`##-##---##-#-###`	0.2876	`###-@#-@#-#-###`	0.3077
11	`###-###-#--#-###`	0.1906	`##@#--##-#-#-@###`	0.2056
12	`###-#-##-#--##-###`	0.1375	`##@#-#-##--#@-####`	0.1481

TAB. 4.3 – *Meilleures graines et sensibilité pour l'automate probabiliste B*

w	graine espacée	sensibilité	graine sous-ensemble	sensibilité
9	`###---##-##-##`	0.4350	`##@---##-##-##@`	0.4456
10	`##-##----##-##-##`	0.3106	`##-##---@##-##@#`	0.3173
11	`##-##----##-##-###`	0.2126	`##@#@-##-##--###`	0.2173
12	`##-##----##-##-####`	0.1418	`##-@###--##-##@##`	0.1477

TAB. 4.4 – *Meilleures graines et sensibilité pour l'automate probabiliste DT1*

w	graine espacée	sensibilité	graine sous-ensemble	sensibilité
9	`#-##----##-##-##`	0.5121	`#-#@-##-@--##-##`	0.5323
10	`##-##-##----##-##`	0.3847	`##-@#-##--@-##-##`	0.4011
11	`##-##--#-#---#-##-##`	0.2813	`##-##-@#-#---#-#@-##`	0.2931
12	`##-##-##-#---#-##-##`	0.1972	`##-##-#@-##-@--##-##`	0.2047

TAB. 4.5 – *Meilleures graines et sensibilité pour l'automate probabiliste DT2*

w	graine espacée	sensibilité	graine sous-ensemble	sensibilité
9	`##-##-##----##-#`	0.5253	`##-@@-##----##-##`	0.5420
10	`##-##----##-##-##`	0.4123	`##-##----##-@@-##-#`	0.4190
11	`##-##----##-##-##-#`	0.3112	`##-##----##-@@-##-##`	0.3219
12	`##-##----##-##-##-##`	0.2349	`##-##----##-@@-##-##-#`	0.2412

TAB. 4.6 – *Meilleures graines et sensibilité pour l'automate probabiliste NT*

4.5.3 Performances des graines espacées et des graines sous-ensemble

De manière à valider l'approche sur le plan expérimental, un ensemble de comparaisons a été réalisé sur des génomes entiers de manière à évaluer l'efficacité des graines espacées et des graines sous-ensemble. Huit génomes bactériens ont été comparés par paires en utilisant le logiciel YASS. La comparaison a été réalisée à l'aide d'une graine espacée et d'une graine à transition du même poids.

Le seuil de *E-value* pour les alignements de sortie a été fixé à 10, et pour chaque comparaison, le nombre *d'alignements exclusifs* (*nb.align.ex.*) trouvés par chaque graine est donné. Un alignement exclusif est un alignement de *E-value* inférieure à un seuil de 10^{-3} qui ne recouvre pas tout alignement trouvé par l'autre graine. Afin de prendre en considération un biais possible dû à l'algorithme de *X-drop* (voir partie 3.4.4.2) qui peut éventuellement séparer un alignement en sous-alignements plus courts, nous avons également calculé la somme des longueurs des *alignements exclusifs* (*lg.align.ex*).

Les résultats sont exposés dans la table 4.7 pour des graines de poids 9 et 10 générées sur les modèles probabilistes $DT2$ et NT. Chaque paire de lignes correspond à une recherche faite à l'aide des graines données dans les tables 4.5 et 4.6, selon le modèle probabiliste choisi. Les résultats expérimentaux montrent que la meilleure graine de type sous-ensemble détecte entre 1% et 8% d'alignements significatifs en plus par rapport à la meilleure graine espacée de poids équivalent.

graine	*temps*	*nb.align*	*nb.align.ex.*	*lg.align.ex*
$DT2$, $w = 9$, graine espacée	15 :14	19101	1583	130512
$DT2$, $w = 9$, graine sous-ensemble	14 :01	20127	1686	141560
$DT2$, $w = 10$, graine espacée	8 :45	18284	1105	10174
$DT2$, $w = 10$, graine sous-ensemble	8 :27	18521	1351	12213
NT, $w = 9$, graine espacée	42 :23	20490	1212	136049
NT, $w = 9$, graine sous-ensemble	41 :58	21305	1497	150127
NT, $w = 10$, graine espacée	11 :45	19750	942	85208
NT, $w = 10$, graine sous-ensemble	10 :31	21652	1167	91240

TAB. 4.7 – *Tests comparatifs des graines sous-ensemble et des graines espacées. Les temps donnés (min :sec) ont été obtenus sur un Pentium™ IV de fréquence 2.4Ghz.*

4.6 Les logiciels YASS et HEDERA

Dans cette partie, nous décrivons les logiciels qui ont été développés dans le cadre de cette thèse. Il s'agit du logiciel YASS et du logiciel annexe HEDERA.

4.6.1 Le logiciel YASS

Le logiciel YASS est un outil de recherche de similarités basé sur l'utilisation de *graines à transitions*. Il donne toutes les similarités «acceptables» trouvées à l'aide de l'heuristique de filtrage proposée en partie 4.4.

4.6.1.1 Présentation

Le logiciel YASS est accessible depuis http://www.loria.fr/projects/YASS, mais également accessible en ligne depuis un serveur dédié à l'adresse http://yass.loria.fr/interface.php. Le logiciel permet de comparer une ou deux séquences au format FASTA ou Multi-FASTA.

La version en ligne de commande permet de traiter les séquences en considérant éventuellement la séquence complémentaire inversée. Les matrices de score et les coûts des *indels* peuvent être, soit choisis dans une présélection, soit modifiés au libre choix de l'utilisateur. Les alignements sont ordonnés selon leur score et un calcul de l'E-value est réalisé pour chacun. D'autres options pratiques allant de la sélection des éléments dans les fichiers Multi-FASTA, aux diagonales «anti-tandem» permettent d'affiner les résultats. L'utilisateur peut également librement modifier la graine et les paramètres de recherche associés, ainsi que les filtres proposés afin par exemple d'adapter la recherche en éliminant les répétitions de faible complexité.

Le format de sortie peut être également choisi parmi quatre formats possibles allant de l'affichage complet des alignements à un mode d'affichage imitant celui du format tabulaire de BLAST. Ce dernier est directement utilisable par de nombreux *parseurs*, et est d'ailleurs utilisé pour réaliser les comparaisons avec d'autres logiciels lors de tests de sensibilité.

Dans l'exemple de la figure 4.6, le résultat est donné sous forme d'un en-tête suivi d'un alignement. La première ligne de sortie de YASS précise les paires de positions , la E-value (voir la partie 3.4.4.2), les tailles des deux séquences et le brin sur lequel elles ont été trouvées (f ou r selon que le brin soit direct ou complémentaire inversé). La deuxième ligne donne les identifiants FASTA (noms normalisés des séquences), la troisième le score et le score normalisé (bit-score) de l'alignement trouvé. Enfin la quatrième ligne donne les mutations par triplet, et la probabilité

```
*(297663-298012)(1408324-1408670) Ev: 6.38864e-46 s: 351/348 f
*  (forward strand) / gi|12057208|gb|AE002098.1|AE002098 (forward strand)
* score = 382 : bitscore = 191.10
* mutations per triplet 18, 71, 25 (1.22e-11) | ts : 58 tv : 56

       |297670   |297680   |297690   |297700   |297710   |297720   |297730
AAGGTCATAGAGCATTATACACATCCAAGAAACGTCGGCTCATTAGATAAAAAATTGCCCAACGTCGGCACTGGTCTAGT
||.||:||:||:||.|||.:|:||||.:|:||||||||:||||:||.||...|::::::.::|||||||||.||.:|.||
AAAGTAATCGACCACTATGAAAATCCGCGCAACGTCGGCACATTCGACAAGGGAGACGATTCCGTCGGCACCGGCATGGT
      |1408330  |1408340  |1408350  |1408360  |1408370  |1408380  |1408390

       |297750   |297760   |297770   |297780   |297790   |297800   |297810
GGGTGCGCCAGCGTGCGGTGATGTGATGAGGTTGCAGATCAAAGTCAACGACTCTACTGGCGTTATTGAAGATGTCAAAT
:||.|||||:|||:|||||.||.||:|||:|:.||||.||||||||:||||||   .::|||.|.||.|||||||.:||||
CGGCGCGCCCGCCTGCGGCGACGTCATGCGCCTGCAAATCAAAGTGAACGAC---GAGGGCATCATCGAAGATGCGAAAT
      |1408410  |1408420  |1408430  |1408440  |1408450     |1408460  |1408470

       |297830   |297840   |297850   |297860   |297870   |297880   |297890
TCAAAACTTTTGGATGTGGCTCCGCCATTGCCTCCTCTTCATATATGACTGAATTGGTACAGGGGATGACCTTGGACGAT
|.|||||||:.||:||.|||||:|||||.||.||:||.:::.::||:||.||.|:|||::|.||:|:.|:|.||||.||.
TTAAAACTTACGGCTGCGGCTCGGCCATCGCTTCGTCCAGCCTGATTACCGAGTGGGTTAAAGGCAAAAGCCTGGATGAC
        |1408490  |1408500  |1408510  |1408520  |1408530  |1408540  |1408550

       |297910   |297920   |297930   |297940   |297950   |297960   |297970
GCGGCAAAAATTAAGAACACTGAAATTGCTAAGGAGTTGAGCTTGCCCCCAGTCAAGTTGCATTGCTCTATGTTAGCGGA
|||:...:|||.||.||||:.|||||.||..|||||||..:|||||:||.||:||.:|:||.|||||.||:||.||:||
GCGCTGGCAATCAAAAACAGCGAAATCGCCGAGGAGTTGGAATTGCCGCCGGTAAAAATCCACTGCTCCATCTTGGCTGA
        |1408570  |1408580  |1408590  |1408600  |1408610  |1408620  |1408630

       |297990   |298000
AGATGCGATCAAGGCAGCTATTAAGGACTA
|||||||.|:||.||.||..||.::||||
AGATGCGGTAAAAGCGGCCGTTGCCGACTA
        |1408650  |1408660
```

FIG. 4.6 – *Exemple de sortie donnée par le logiciel* YASS

d'obtenir ce biais en mutations, ainsi que le nombre de transitions et transversions. Lors de l'affichage de l'alignement, les substitutions de type transition sont représentées par un point, les transversions étant représentées par deux points.

La version en accès libre sur la page web http://yass.loria.fr/interface.php dispose également d'une représentation sous forme de dot-plot permettant de visualiser plus facilement la disposition des alignements entre les deux séquences, si ces dernières possèdent de nombreuses similarités.

Les temps de calcul dépendent bien entendu du taux de redondance sur les séquences, du taux de *GC* ainsi que de la sur-représentation de certains mots ou groupes de mots. Ainsi une comparaison entre deux séquences dont la taille est de 2 Méga-bases peut prendre des temps différents : 15 secondes si les séquences sont générées de manière i.i.d et par exemple plus d'une minute sur deux génomes issues de famille *Neisseria Menengitidis* (forte redondance du génome). Bien entendu le temps de calcul peut être bien plus important : il est bon de rappeler que, quoi que puisse être l'efficacité de l'étape de filtrage, la taille de la solution développée sera en $\mathcal{O}(m \times n)$ avec m et n données comme tailles des séquences à comparer. Il suffit par exemple de comparer une séquence contenant une très longue répétition en tandem, composée d'une copie répétée de très nombreuses fois, pour obtenir une solution dont la taille sera critique.

4.6.1.2 Implémentation

Nous décrivons dans cette partie l'implémentation de YASS.

Le programme YASS comporte environ 10000 lignes de code en langage C ANSI. Il est sous licence GPL. Une description de l'algorithme initial et de ses spécificités est donnée dans l'article [88], inclus en annexe. Citons notamment, en plus des graines espacées, le calcul de paramètres statistiques pour regrouper les *hits* appartenant à des alignements potentiels.

L'interface web L'interface web a dû faire l'objet de la création d'un serveur dédié, à cause des coûts engendrés par le programme et les scripts servant à afficher les résultats. Le serveur fonctionne actuellement sur un ordinateur de l'équipe Adage. Il s'agit d'un serveur Apache 2.0 dont les modules PHP et Perl-CGI ont été installés. La majorité des traitements est faite en utilisant des scripts Perl-CGI, seul le dot-plot et l'interface sont générés à l'aide du langage PHP.

4.6.2 Le logiciel HEDERA

Le logiciel HEDERA est un outil qui réalise la conception des graines utilisées par YASS.

4.6.2.1 Présentation

Le logiciel HEDERA est téléchargeable sous forme de binaire (Windows^{TM}, Linux) depuis `http://www.loria.fr/projects/YASS/hedera.html`. Le logiciel permet de sélectionner les graines efficaces selon différents modèles d'alignements. Il propose des modèles probabilistes issus en particulier de [15] et adaptés à l'approche des automates probabilistes. Il implémente également les nouveaux automates et les nouveaux modèles concernant les graines à transitions, les graines sous-ensemble, ainsi que les séquences homogènes.

Il permet de tester des graines, soit de manière exhaustive, soit par génération aléatoire. Les critères pour la génération des graines à transition sont leur poids, leurs étendues minimale et maximale, le nombre d'éléments autorisant des transitions, ainsi que le nombre de graines à optimiser en même temps. Il est possible d'afficher soit la meilleure graine, soit toutes les graines dont la sensibilité est supérieure à un certain seuil. D'autres options permettent d'activer ou de désactiver certaines fonctions internes au programme. Par exemple la minimisation de Hopcroft n'est pas nécessaire sur des familles de graines composées d'une seule graine relativement courte, et peut dans ce cas être désactivée ; elle est par contre recommandée lorsque plusieurs graines sont optimisées en même temps, car elle allège le produit d'automates.

```
hedera -W 9,9 -S 9,15 -t 0.736 -T 2,2
# Match Prob    Seed()s
0.73657         ###_#__#@#_@##
0.73657         ##@_#@#__#_###
0.736186        ###@_#_#_@#__##
0.737453        ###@_#_#__#@_##
0.737107        ###@_#_#__@#_##
0.73608         ##_##__#_#_@@##
0.73608         ##@@_#_#__##_##
0.737107        ##_#@__#_#_@###
0.737453        ##_@#__#_#_@###
0.736186        ##__#@_#_#_@###
```

FIG. 4.7 – *Exemple de sortie donnée par le logiciel* HEDERA

Dans l'exemple de la figure 4.7, HEDERA recherche les graines de poids 9 et d'étendue au plus 15 dont la sensibilité est supérieure à 0.736 sur le modèle de Bernoulli par défaut. Le résultat

est donné sous forme d'une liste de graines au fur et à mesure que le calcul progresse. Lorsque la recherche est lancée de manière exhaustive, le temps de calcul est relativement important : 5 min 30 sec sur un Pentium™ IV de fréquence 2.4Ghz pour l'exemple de la figure 4.7.

De nombreux tests ont été réalisés, les temps de calcul pour une seule graine varient entre 10 ms sur les modèles les plus simples comme celui de Bernoulli et plus de 500 ms sur les modèles complexes à base d'automates probabilistes non-déterministes.

4.6.2.2 Implémentation

Il est basé sur les travaux décris à la partie 4.2 et totalise environ 3000 lignes de code en langage C++ ANSI. Un ensemble de classes, servant à représenter les automates, est particulièrement développé et étendu. Le programme inclut :

- l'algorithme de construction de l'automate de Aho Corasick (décrit à la partie 2.2.4 et adapté à la partie 3.7.7.1), deux algorithmes de construction d'automates déterministes des graines sous-ensemble (la construction est une variante de celle décrite à la partie 4.4.2.2, algorithme 9).
- les algorithmes de construction d'automates probabilistes pour 5 modèles allant du modèle de Bernoulli aux modèles de Markov cachés (automates non-déterministes).
- les algorithmes de construction d'automates déterministes pour représenter les séquences homogènes.
- le produit et l'union d'automates.
- la minimisation d'automates déterministes (algorithme de Hopcroft, présenté à la partie 2.2.3.2, algorithme 3).
- l'algorithme permettant de calculer la probabilité d'atteindre un état donné sur un automate probabiliste.
- des méthodes de génération et d'énumération des graines espacées et à transitions.

Le programme a été testé sous Linux et Windows™. D'un point de vue purement technique, nous avons remarqué que le programme est plus rapide compilé sous Linux avec le compilateur `gcc` (version 3.3.1) qu'avec le compilateur `icc` (version 8.0) d'IBM. Dans les deux cas, les options d'optimisation associées au Pentium™ IV ont été utilisées.

4.6.3 Utilisation de YASS : clusters de répétitions

Le logiciel YASS a permis d'analyser plusieurs génomes bactériens afin d'y trouver des répétitions. Une idée sous-jacente est d'analyser les répétitions présentes entre une ou plusieurs espèces afin d'y isoler et d'examiner des groupes de répétitions intéressants [105].

En collaboration avec A. Vitreschak, nous avons travaillé sur une approche permettant de rechercher de manière rapide les groupes de répétitions selon les recouvrements entre les paires d'occurrences. Cette méthode est divisée en deux étapes :

- Une étape dite de pré-regroupement : cette étape a pour but de rassembler tous les fragments de séquence qui ont une forte similarité à l'aide d'une heuristique permettant de rechercher des quasi-cliques [1] (des cliques presque parfaites) dans un graphe où les fragments de séquence sont des nœuds et les arêtes représentent les alignements. La structure de données principale de cette étape est un arbre d'intervalles modifié afin de mémoriser les paires d'intervalles représentant les alignements.
- Une deuxième étape de regroupement a alors lieu sur les groupes formés, afin d'affiner le résultat. Elle s'appuie sur une méthode mise au point par A. Vitreschak, qui utilise la notion de blocks communs conservés nommés *core blocks* [104].

A. Vitreschak dans [104] a traité différentes souches du génome de bactérie pathogène *Neisseria Meningitidis*, et obtenu des répétitions déjà connues comme des éléments mobiles, des terminateurs ρ-indépendants. Il a également pu extraire des éléments inconnus dont un a attiré son attention. Il s'agit d'un élément de 120 paires de bases, présent en plus de 100 copies dans le génome. Cet élément a une structure palindromique complexe laissant présager un ARN avec structure tridimensionnelle complexe, ou un élément mobile d'un nouveau type. Un autre élément a également été détecté, ce dernier étant très souvent présent devant les gènes pathogènes de la bactérie.

Chapitre 5

Recherche de motifs approchés et filtrage sans perte

Sommaire

Nous allons étudier dans ce chapitre un problème de recherche de motifs approchés. Il s'agit de retrouver les occurrences d'un motif donné dans un texte en autorisant une certaine distance entre les copies du motif.

Nous définirons dans un premier temps le problème (partie 5.1), donnerons les approches précédemment utilisées pour le résoudre (parties 5.1.1-5.1.3) avant de nous intéresser à une méthode particulière : le filtrage sans perte à l'aide de graines multiples (partie 5.2). Nous analyserons cette méthode de manière détaillée, donnerons des algorithmes de programmation dynamique permettant de calculer certaines caractéristiques (partie 5.2.1), analyserons certaines propriétés de la conception des graines simples et multiples (partie 5.3), et donnerons enfin quelques résultats associés à la conception des graines (partie 5.4).

Nous considérerons dans un second temps un problème biologique relatif à la recherche d'oligonucléotides (partie 5.5), poserons le problème (partie 5.5.2) et les solutions précédentes (partie 5.5.2.1) avant de proposer notre approche (partie 5.5.2.2).

Les travaux présentés dans la partie 5.2–5.3 ont été réalisés en collaboration avec G. Kucherov et M. Roytberg.

5.1 Recherche de motifs approchés

Nous allons considérer dans cette partie le problème de recherche de motifs approchés selon la *distance de Hamming* (définie en partie 2.1.2.1).

Problème 1 (recherche de motifs approchés) *Étant donné un texte T de taille n, un motif P de taille m, et un entier k ($k < m$), la recherche de motifs approchés consiste à trouver toutes les positions i sur T telles que $dist_{\mathcal{H}}(P, T[i..i+m-1]) \leq k$.*

Nous obtenons ainsi les positions i de toutes les similarités du motif P vérifiant le critère $dist_{\mathcal{H}}(P, T[i..i+m-1]) \leq k$. De manière à modéliser ces similarités, nous utiliserons la représentation précédemment introduite à l'aide de mots sur un alphabet binaire $\mathcal{A} = \{1, 0\}$. Toute similarité sera représentée par un mot binaire de taille m ayant au plus k lettres 0 pour modéliser les substitutions.

EXEMPLE 27 : *recherche de motifs approchés*

Soit T = CGCTGCAGTG, P = CACT ($m = 4$), et $k = 1$. Les positions des similarités détectées sont alors $\{1, 6\}$ et les mots binaires associés sont respectivement $\{1011, 1101\}$ d'après les similarités suivantes :

```
CACT              CACT
1011      et      1101   .
CGCTGCAGTG    CGCTGCAGTG
```

Définition 24 (problème (m, k)) *Étant donné deux entiers m et k ($k < m$), nous considérons l'ensemble des similarités représentées par des mots sur $\mathcal{A}$ ayant k lettres 0 et $(m-k)$ lettres 1. Ces similarités sont appelées* (m, k)-similarités *et constituent un problème (m, k).*

Résoudre le problème (m, k) consiste à trouver un filtre qui détecte toutes les (m, k)-similarités.

Définition 25 (filtre résolvant un problème (m, k)) *Étant donné deux entiers m et k, un filtre résout le problème (m, k) si et seulement s'il génère au moins un* hit *sur chacune des (m, k)-similarités.*

Bien entendu, le but est ici de réaliser un *filtrage sans perte*, c'est-à-dire trouver *toutes* les (m, k)-similarités.

5.1.1 Approche PEX

Les méthodes de filtrage classiques pour un distance de Hamming (ou de Levenstein) sont en général basées sur la recherche de fragments contigus (nommé PEX dans [82]). En effet, si deux fragments de texte de taille m ont une distance de Hamming de k, alors, en utilisant le principe des pigeons et des nids (ou des tiroirs et des chaussettes), ils ont en commun au moins un fragment de taille $\lceil \frac{m-k}{k+1} \rceil = \lfloor \frac{m}{k+1} \rfloor$. Ainsi une graine contiguë, c'est-à-dire uniquement composée de symboles # et dont le motif est de longueur $\lfloor \frac{m}{k+1} \rfloor$ résoudra un problème (m, k) donné.

Lemme 16 (filtrage par graines contiguës) *La graine* $\#^{\lfloor \frac{m}{k+1} \rfloor}$ *résout le problème* (m, k) *car elle détecte toutes les* (m, k)*-similarités.*

Preuve (**filtrage par graines contiguës**) Les k erreurs de substitution séparent une (m, k)-similarité en $k+1$ intervalles de taille positive ou nulle composés uniquement de lettres `1`. Comme la somme de la taille de ces intervalles, c'est-à-dire le nombre de lettres `1`, est $m - k$, alors par distribution de ces lettres, éléments discrets, il existe au moins un intervalle de taille $\lceil \frac{m-k}{k+1} \rceil$. □

EXEMPLE 28 : *recherche de* (m, k)*-similarités à l'aide de graines contiguës*

Considérons le problème $(m = 12, k = 3)$. Les similarités issues de ce problème possèdent au moins un fragment commun de taille $\lfloor \frac{m}{k+1} \rfloor = 3$. En conséquence la graine `###` peut résoudre le problème $(m = 12, k = 3)$. Par exemple, la graine `###` détecte les (m, k)-similarités `110110110111` et `110111011011` :

```
          ###                  ###
110110110111      ou     110111011011
```

Cette approche est traditionnelle, elle possède d'ailleurs l'avantage de pouvoir s'appliquer à la distance de Levenstein.

5.1.2 Approche proposée par Pevzner et Waterman

L'idée que Pevzner et Waterman ont proposée et analysée dans l'article [91] consiste à combiner deux filtres. Le premier est basé sur l'approche de la partie précédente, c'est-à-dire la recherche à l'aide de graines contiguës. Le deuxième est un filtre plus particulier : il utilise une *graine espacée*. Le motif de cette graine espacée est complètement défini pour un problème (m, k) donné : il est donné par le mot $(\#(-)^k)^r\#$ avec $r = \lfloor \frac{m}{k+1} \rfloor - 1$.

EXEMPLE 29 : *recherche de* (m, k)*-similarités à l'aide d'une graine espacée définie*

Si l'on reprend le problème $(m = 12, k = 3)$ de l'exemple 28, la graine espacée a comme motif `#---#---#`. Cette graine à elle seule résout le problème $(m = 12, k = 3)$. Par exemple `#---#---#` détecte les (m, k)-similarités `110011110111` et `110111110011` :

```
      #---#---#                  #---#---#
110011110111      ou     110111110011
```

La raison du fonctionnement de ce filtre n'est pas expliqué ici, mais une généralisation est donnée plus loin au lemme 18. Les deux graines résolvent le problème (m, k), et sont utilisées conjointement pour obtenir un filtre composé efficace : si les deux graines respectives génèrent un *hit* dans un voisinage proche, alors le critère de *hit* du filtre composé est vérifié.

5.1.3 Filtrage sans perte à l'aide de graines espacées simples

Burkhardt et Kärkkäinen ont été les premiers à rechercher des graines espacées génériques (le terme employé était *Gapped Q-grams*) pour résoudre le problème de filtrage sans perte. Ici la conception de la graine est libre : il s'agit de trouver le motif d'une graine espacée sans précondition sur la forme de ce motif, le but étant uniquement de résoudre le problème (m, k). Le

choix se porte bien entendu sur les graines dont le *poids* est maximal.

EXEMPLE 30 : *recherche de (m,k)-similarités à l'aide d'une graine espacée générale*
Si l'on considère le problème $(m = 14, k = 3)$, une graine contiguë résolvant le problème a un poids maximal de 3 (motif `###`). Il est cependant possible de résoudre le même problème à l'aide d'une graine espacée de poids 4 comme par exemple `##-##`, ou bien `#-###`, ou encore `#-#---#-#`.

5.2 Filtrage sans perte à l'aide de graines espacées multiples

Le principe de filtrage sans perte à l'aide de graines simples permet des gains par rapport aux approches traditionnelles à l'aide de graines contiguës. Ce gain reste cependant limité, en particulier lorsque le ratio entre le nombre d'erreurs autorisées et la longueur de la (m,k)-similarité recherchée $\frac{k}{m}$ est relativement élevé (> 0.1).

Nous avons donc proposé une approche à l'aide de graines multiples. Une *famille* de graines (ensemble de graines) est conçue de manière à résoudre un problème (m,k) donné. Une famille de graines $F =< \pi_l >_{l=1}^{L}$ *résout le problème* (m,k) si et seulement si pour toutes les $\binom{m}{k}$ (m,k)-similarités w, il existe au moins une graine $\pi_l \in F$ qui détecte w.

Ainsi, les graines d'une famille sont utilisées de manière complémentaire (disjonctive) : une (m,k)-similarité est détectée dès qu'une graine de la famille génère un *hit*. Cette approche diffère de la méthode proposée dans la partie 5.1.2 où une (m,k)-similarité n'est détectée que si les deux graines qui composent le filtre génèrent un *hit* dans un voisinage proche.

De manière à illustrer le principe et son intérêt, nous prenons l'exemple du problème $(25,2)$. La graine de poids maximal qui résout le problème $(25,2)$ est `###-#--###-#--###-#`, son poids est de 12. Il est également possible de résoudre le même problème $(25,2)$ à l'aide d'une famille de deux graines de poids 14 composée de `#####-##---#####-##` et `#-##---#####-##---####`.

Dans cet exemple, la sélectivité de la famille de graines est meilleure que celle de la graine unique. En effet la probabilité d'un *hit* sur des séquences générées par un modèle de Bernoulli uniforme varie. Dans le cas du filtre composé de deux graines, la probabilité est d'environ $\frac{2}{|\Sigma|^{14}}$ alors qu'elle est de $\frac{1}{|\Sigma|^{12}}$ dans le cas du filtre à une seule graine. Le ratio $|\Sigma|^2/2$ donne le gain en sélectivité, de l'ordre de 8 sur des séquences générées par un modèle de Bernoulli uniforme sur un alphabet à 4 lettres comme l'ADN. En contrepartie il est nécessaire de créer un index pour chacune des graines de la famille, ce qui se traduit en général par un léger coût supplémentaire en temps lors du pré-traitement mais surtout par un coût éventuel en mémoire pour le stockage des index.

5.2.1 Mesure de propriétés des graines

Burkhardt et Kärkkäinen [27] ont proposé une approche par programmation dynamique pour évaluer le seuil de détection d'une graine π, c'est-à-dire le nombre minimal de *hits* générés par π sur chacune des (m,k)-similarités. Nous décrivons dans cette partie une extension de cet algorithme pour des graines multiples, ainsi que pour le calcul d'autres caractéristiques utiles des graines multiples.

Fixons un problème (m,k), et une famille de graines $F =< \pi_l >_{l=1}^{L}$. Nous utilisons les notations suivantes :

- $s(\pi)$ désigne l'étendue d'une graine π donnée.
- $s_{max} = max\{s(\pi_l)\}_{l=1}^{L}$, $s_{min} = min\{s(\pi_l)\}_{l=1}^{L}$,

- pour un mot binaire w et une graine π_l, $term(\pi_l, w)$ est une fonction qui retourne 1 si π_l détecte le mot binaire w à la position $(|w| - s(\pi_l) + 1)$: (i.e. π_l détecte le suffixe de w), sinon $term(\pi_l, w) = 0$,
- $last(w)$ donne la dernière lettre du mot w,
- $zeros(w)$ donne le nombre de lettres 0 dans le mot w.

5.2.2 Seuil optimal

Étant donné un problème (m, k), une famille de graines $F =< \pi_l >_{l=1}^{L}$ a le seuil optimal $T_F(m, k)$ si toute (m, k)-similarité génère au moins $T_F(m, k)$ *hits* de graines $\in F$. Deux *hits* d'une même graine π à des positions différentes ou deux graines distinctes d'une famille générant un *hit* à la même position sont comptés séparément.

EXEMPLE 31 : *seuil optimal d'une graine*

La famille $F = \{$`###-##`$\}$ composée d'une seule graine a un seuil optimal de 2 pour le problème $(m = 15, k = 2)$. Par exemple, si l'on considère la (m, k)-similarité `111111011011111`, deux *hits* sont observés en $1^{ère}$ et $4^{ème}$ positions :

```
###-##                 ###-##
111111011011111   et   111111011011111
```

F résout donc un problème (m, k) si et seulement si $T_F(m, k) > 0$. Il est possible de renforcer le critère du filtre à deux *hits* si par exemple $T_F(m, k) > 1$.

De manière à évaluer ce seuil, nous décrivons un algorithme de programmation dynamique pour le calcul du seuil optimal $T_F(m, k)$, extension de l'algorithme proposé par [27] selon un principe similaire à [59]. Pour un mot binaire w, soit $T_F(m, k, w)$ le nombre de *hits* minimal d'une famille F sur toutes les (m, k)-similarités qui ont comme suffixe le mot w.

L'évaluation de $T_F(m, k)$ se ramène alors à estimer $T_F(m, k, \varepsilon)$. Supposons que nous pré-calculions les valeurs $\mathcal{T}_F(j, w) = T_F(s_{max}, j, w)$ pour $j \leq \max\{k, s_{max}\}$, $|w| = s_{max}$.

L'algorithme de programmation dynamique est basé sur les relations de récurrence suivantes sur $T_F(i, j, w)$ lorsque $i \geq s_{max}$.

$$T_F(i, j, w[1..n]) = \begin{cases} \mathcal{T}_F(j, w), & \text{si } i = s_{max}, \\ T_F(i-1, j-1, w[1..n-1]), & \text{si } last(w) = 0, \\ T_F(i-1, j-1+last(w), w[1..n-1]) + \sum_{l=1}^{L} term(\pi_l, w), & \text{si } n = s_{max}, \\ \min\{T_F(i, j, \mathtt{1}.w), T_F(i, j, \mathtt{0}.w)\}, & \text{si } zeros(w) < j, \\ T_F(i, j, \mathtt{1}.w), & \text{si } zeros(w) = j. \end{cases} \tag{5.1}$$

La première ligne de la relation sert de condition pour l'initialisation de la récurrence. La deuxième ligne utilise le principe suivant : si la dernière lettre de w est 0 alors aucune graine ne peut détecter la (m, k)-similarité à la dernière position (puisque le premier et dernier symbole d'une graine sont #).

La troisième ligne réduit la taille du problème en comptant le nombre de graines détectant le mot w à sa dernière position.

Les deux derniers cas se déduisent d'une extension du mot w sur sa gauche : si w contient déjà k zéros, alors seul une lettre 1 peut être placée à la gauche de w, autrement l'extension par ajout de la lettre 0 ou 1 est possible. Une implémentation de la récurrence permet de calculer $T_F(m, k, \varepsilon)$ en appliquant les relations dans l'ordre donné.

Une programmation directe donnerait un coût en temps et en espace de $\mathcal{O}(m \cdot k \cdot 2^{s_{max}})$. Il est cependant possible, pour tout i et tout j de ne considérer que les mots binaires de taille s_{max}

qui contiennent au plus j zéros. Ils sont au nombre de $g(j, s_{max}) = \sum_{a=0}^{j} \binom{s_{max}}{a}$. Pour chaque i, le nombre de valeurs possibles de paires $j \times w$ sera donc de $f(k, s_{max}) = \sum_{j=0}^{k} g(j, s_{max}) = \sum_{j=0}^{k} \binom{s_{max}}{j} \cdot (k-j+1)$. La complexité est alors la même que celle proposée pour le calcul du seuil optimal pour une seule graine [27]. Les valeurs $\sum_{l=1}^{L} suff(\pi_l, w)$ peuvent être pré-calculées pour tous les mots considérés en temps $\mathcal{O}(L \cdot g(k, s_{max}))$ et en espace $\mathcal{O}(g(k, s_{max}))$, si l'on suppose que la vérification d'un *hit* de graine est faite en temps constant sur un mot binaire.

La complexité totale en temps est alors de $\mathcal{O}(m \cdot f(k, s_{max}) + L \cdot g(k, s_{max}))$ avec en général comme terme principal $m \cdot f(k, s_{max})$ (L est dans la plupart des cas plus petit que m et $g(k, s_{max})$ est plus petit que $f(k, s_{max})$).

5.2.3 Nombre de (m, k)-similarités non-détectées

Nous décrivons dans cette partie un algorithme de programmation dynamique qui calcule une autre caractéristique d'une famille de graines. Cet algorithme est utilisé dans la partie 5.3.4. Étant donné un problème (m, k) et une famille de graines $F =< \pi_l >_{l=1}^{L}$, nous sommes intéressés par le nombre $U_F(m, k)$ de (m, k)-similarités qui ne sont pas détectées par F. Pour un mot binaire w, on définit $U_F(m, k, w)$ comme le nombre de (m, k)-similarités non-détectées qui ont comme suffixe w.

La relation de récurrence suivante permet de calculer $U_F(i, j, w)$ pour tout $i \leq m$, $j \leq k$, et pour $|w| \leq s_{max}$.

$$U_F(i, j, w[1..n]) = \begin{cases} \binom{i-|w|}{j-zeros(w)}, & \text{si } i < s_{min}, \\ U_F(i-1, j-1, w[1..n-1]), & \text{si } last(w) = 0, \\ 0, & \text{si } \exists l \in [1..L], term(\pi_l, w) = 1, \\ U_F(i, j, \mathtt{1}.w) + U_F(i, j, \mathtt{0}.w), & \text{si } zeros(w) < j, \\ U_F(i, j, \mathtt{1}.w), & \text{si } zeros(w) = j. \end{cases}$$

La première condition précise que si $i < s_{min}$, alors aucune graine ne détectera un mot de longueur i, donc tous les mots vérifiant ce critère ne seront pas détectés. La deuxième et les deux dernières conditions sont similaires à l'équation 5.1. La troisième condition provient de la définition de $U_F(i, j, w)$.

Le temps de calcul est également en $\mathcal{O}(m \cdot f(k, s_{max}) + L \cdot g(k, s_{max}))$.

5.2.4 Contribution d'une graine

Un aspect intéressant d'une famille de graines est de connaître la contribution respective de chaque graine dans la résolution potentielle d'un problème (m, k).

En utilisant une approche similaire il est possible de calculer le nombre de (m, k)-similarités détectées par une et une seule graine d'une famille. Ce paramètre est également utile à l'algorithme de la partie 5.3.4.

Étant donné un problème (m, k) et une famille de graines $F =< \pi_l >_{l=1}^{L}$, nous définissons $S_F(m, k, l)$ comme le nombre de (m, k)-similarités détectées par la graine π_l uniquement (un ou plusieurs *hit* sont possibles), et de manière similaire $S_F(m, k, l, w)$ comme le nombre de (m, k)-similarités qui terminent avec le suffixe w.

Une récurrence similaire à celle de l'équation 5.1 peut être donnée pour le calcul associé à

chacune des graines de la famille.

$$S_F(i,j,w[1..n],l) = \begin{cases} 0 & \text{si } i < s_{min} \text{ ou } \exists l' \neq \ term(\pi_{l'},w) = 1 \\ S_F(i-1,j-1,l,w[1..n-1]) & \text{si } last(w) = 0 \\ S_F(i-1,j-1+last(w),l,w[1..n-1]) & \text{si } n = |\pi_l| \text{ et } term(\pi_l,w) = 0 \\ S_F(i-1,j-1+last(w),l,w[1..n-1]) & \text{si } n = s_{max} \text{ et } term(\pi_l,w) = 1 \\ \quad + U_F(i-1,j,w[1..n-1]) & \text{et } \forall l' \neq l, term(\pi_{l'},w) = 0, \\ S_F(i,j,l,\mathtt{1}.w[1..n]) & \\ \quad + S_F(i,j,l,\mathtt{0}.w[1..n]) & \text{si } zeros(w) < j \\ S_F(i,j,l,\mathtt{1}.w[1..n]) & \text{si } zeros(w) = j \end{cases}$$

La troisième et la quatrième relations servent de moteur à l'algorithme : si π_l ne détecte pas le suffixe de $w[1..n]$, alors la dernière lettre de w est retirée. Si π_l détecte le suffixe de w, et est la seule graine à le faire, alors nous comptons le nombre de préfixes uniquement détectés par π_l (terme $S_F(i-1,j,l,w[1..n-1])$).

5.3 Conception des graines

Dans la partie précédente, nous avons montré comment calculer des caractéristiques utiles d'une famille de graines. Un problème plus difficile est de trouver une famille de graines efficace qui résout un problème (m,k) donné. Il existe par exemple des solutions évidentes comme des familles composées de toutes les $\binom{m}{k}$ graines associées aux $\binom{m}{k}$ combinaisons de substitutions mais ce type de solutions est peu appropriée en pratique à cause du nombre élevé de graines impliquées dans la famille. Notre but est de trouver des familles de graines de taille raisonnable (par exemple dont le nombre de graines est inférieur à 10), possédant une bonne efficacité au niveau du filtrage.

Dans cette partie, nous présentons des résultats pour contribuer à ce problème. Dans la partie 5.3.1, nous considérons le cas d'une seule graine avec un nombre fixé de jokers. Nous montrons en particulier que pour un seul joker, il existe une graine optimale (selon un critère d'optimalité tout à fait raisonnable). Nous montrons ensuite dans la partie 5.3.2 qu'une solution pour un grand problème (en terme de taille m) peut être obtenue à partir d'un problème plus petit, à l'aide d'une opération d'expansion sur les graines de la famille résolvant le problème. Dans la partie 5.3.3, nous nous intéressons aux graines qui ont une structure périodique et montrons comment ces graines peuvent être construites par répétitions de motifs de graines particulières plus courtes. Nous donnons un méthode pour construire des familles de graines périodiques efficaces. Enfin, dans la partie 5.3.4, nous présentons une approche heuristique pour construire des familles de graines efficaces dont certaines sont utilisées dans la partie 5.4 de ce travail.

5.3.1 Graines simples avec un nombre fixé de jokers

Supposons que nous définissons une classe de graines qui nous intéressent (par exemple des graines de poids minimal donné). Une manière possible de définir le problème de conception est de fixer la longueur de similarité m, et trouver une graine de poids fixé w dans la classe qui résout le problème (m,k) en maximisant alors k. Une définition complémentaire est de fixer k et de minimiser m dans la mesure où le problème (m,k) est résolu par une graine de poids w de la classe. Dans cette partie nous choisissons la deuxième définition et donnons une solution générale optimale pour un cas particulier.

Nous donnons dans un premier temps quelques définitions.

Définition 26 (longueur k-critique d'une graine) *Pour une graine π et un nombre d'erreurs k, la* longueur k-critique *pour une graine π est la valeur minimale de m telle que π puisse résoudre le problème (m, k).*

Définition 27 (k-optimalité dans une classe de graines) *Pour une classe de graines $\mathcal{C}$ et un entier k, une graine est dite k-optimale dans $\mathcal{C}$ si π possède la longueur k-critique minimale de toutes les graines de $\mathcal{C}$.*

Il nous reste à définir la classe de graines que nous souhaitons considérer. Une classe intéressante peut être obtenue en fixant une borne supérieure sur le nombre de jokers contenus dans la graine, soit sur la quantité $(s(\pi) - w(\pi))$. Nous proposons une solution générale pour le problème de conception de graines pour la classe $\mathcal{C}_1(n)$, qui comprend l'ensemble des graines de poids d ne contenant qu'un joker, c'est-à-dire des graines de la forme $\#^r\text{-}\#^{d-r}$ pour tout $r \in [1..d-1]$.

Nous considérons dans un premier temps des (m, k)-similarités avec une erreur, c'est-à-dire $k = 1$.

Lemme 17 (graine 1-optimale dans la classe $\mathcal{C}_1(d)$) *Une graine 1-optimale pour $\mathcal{C}_1(d)$ est de la forme $\#^r\text{-}\#^{d-r}$ avec $r = \lfloor d/2 \rfloor$.*

Preuve (graine 1-optimale dans la classe $\mathcal{C}_1(d)$) prenons une graine arbitraire dans $\mathcal{C}_1(d)$ $\pi = \#^p\text{-}\#^q$, $p+q = d$. Supposons que $p \geq q$. Alors la plus longue similarité du problème $(m, 1)$ non détectée par π sera de la forme $1^{p-1}01^{p+q}$ et de longueur $(2p + q)$. Nous devons donc minimiser $2p + q = d + p$, et comme $p \geq \lceil d/2 \rceil$, le minimum est atteint pour $p = \lceil d/2 \rceil$, $q = \lfloor d/2 \rfloor$. □

Dans le cas où le nombre d'erreurs k est supérieur ou égal à deux, une graine optimale de $\mathcal{C}_1(d)$ aura une structure asymétrique que nous donnons dans le théorème suivant :

Théorème 2 (graine k-optimale dans la classe $\mathcal{C}_1(d)$) *Soit d un entier désignant le poids de la graine et $r = \lfloor d/3 \rfloor$ (avec $\lfloor x \rfloor$ donnant l'arrondi entier le plus proche de x). Pour tout $k \geq 2$, la graine de la forme $\pi(d) = \#^d\text{-}\#^{d-r}$ est k-optimale parmi les graines de $\mathcal{C}_1(d)$.*

Preuve (graine k-optimale dans la classe $\mathcal{C}_1(d)$) Soit une graine $\pi = \#^p\text{-}\#^q$ avec $p+q = d$, et supposons que $p \geq q$. Soit $S(k)$ la plus longue similarité de type $(1^*0)^k1^*$, avec $k \geq 1$, qui n'est pas détectée par π. $L(k)$ est la longueur de la similarité $S(k)$. D'après le lemme 17, $S(1) = 1^{p-1}01^{p+q}$ et $L(1) = 2p + q$.

Il est possible de voir que pour tout k, $S(k)$ a comme préfixe, soit $1^{p-1}0$, soit $1^{p+q}01^{q-1}0$. Soit $L'(k)$ la longueur maximale d'une similarité de la forme $(1^*0)^k1^*$ dont le préfixe est $1^{q-1}0$ et qui n'est pas détectée par π. Puisque le préfixe à lui seul empêche la détection par π, nous avons $L'(k) = q + L(k-1)$. Notons que $L'(1) = p + 2q$ (à cause de la similarité $1^{q-1}01^{p+q}$).

Les relations de récurrence suivantes sont établies pour $k \geq 2$:

$$L'(k) = q + L(k-1), \tag{5.2}$$

$$L(k) = \max\{p + L(k-1), p + q + 1 + L'(k-1)\}, \tag{5.3}$$

avec comme conditions initiales $L'(1) = p + 2q$, $L(1) = 2p + q$.

Deux cas peuvent être distingués. Si $p \geq 2q + 1$, alors une induction directe montre que le premier terme de (5.3) est plus grand que le second, ce qui implique

$$L(k) = (k + 1)p + q, \tag{5.4}$$

avec comme similarité non-détectée associée

$$S(k) = (1^{p-1}0)^k 1^{p+q}. \tag{5.5}$$

Si $q \leq p \leq 2q+1$, alors par induction, nous obtenons

$$L(k) = \begin{cases} (\ell+1)p + (k+1)q + \ell & \text{si } k = 2\ell, \\ (\ell+2)p + kq + \ell & \text{si } k = 2\ell+1, \end{cases} \tag{5.6}$$

avec

$$S(k) = \begin{cases} (1^{p+q}01^{q-1}0)^\ell 1^{p+q} & \text{si } k = 2\ell, \\ 1^{p-1}0(1^{p+q}01^{q-1}0)^\ell 1^{p+q} & \text{si } k = 2\ell+1. \end{cases} \tag{5.7}$$

Par définition de $L(k)$, la graine $\#^p\text{-}\#^q$ détecte toute similarité de $(1^*0)^k1^*$ de longueur $(L(k)+1)$ ou plus, ce qui est la borne la plus précise. Nous devons donc trouver des couples p, q qui minimisent $L(k)$. Comme $p+q=d$, on peut voir que pour $p \geq 2q+1$, $L(k)$ (défini par (5.4)) est une fonction croissante selon p, alors que pour $p \leq 2q+1$, $L(k)$ (défini par (5.6)) est une fonction décroissante selon p. On peut donc en conclure que les fonctions atteignent leur minimum commun lorsque $p = 2q+1$. Ainsi, lorsque $d \equiv 1 \pmod 3$, alors $q = \lfloor d/3 \rfloor$ et $p = d-q$. Si $d \equiv 0 \pmod 3$, un calcul montre que le minimum est atteint pour $q = d/3$, $p = 2d/3$. Sinon pour $d \equiv 2 \pmod 3$, le minimum est atteint pour $q = \lceil d/3 \rceil$, $p = d-q$. Lorsque l'on assemble les trois cas, le résultat est alors $q = [d/3]$, $p = d-q$. □

EXEMPLE 32 : *graine k-optimale dans la classe $\mathcal{C}_1(d)$*

Afin d'illustrer le théorème 2, la graine `####-##` est optimale parmi toutes les graines de poids 6 possédant un seul joker. Ceci signifie que cette graine résout le problème $(m, 2)$ pour tout $m \geq 16$, et que cette borne est la plus petite pour toute les graines de cette classe. De manière similaire, cette graine résout le problème $(m, 3)$ pour tout $m \geq 20$, ce qui est également la meilleure borne pour la classe.

La classe $\mathcal{C}_1(d)$ est une classe de graines intéressante car elle correspond aux graines expérimentales optimales de l'article [26]. Dans cet article, les auteurs proposent l'utilisation d'une graine de poids $d = p+q$ de la forme $\#^p\text{-}^j\#^q$ (avec $j \geq 1$) afin de détecter les similarités, non plus selon la distance de Hamming, mais selon la distance de Levenstein. Cette approche est en effet possible en utilisant deux graines complémentaires données par les motifs $\#^p\text{-}^{j-1}\#^q$ et $\#^p\text{-}^{j+1}\#^q$ afin de tolérer un *indel* en plus des substitutions.

Nous nous sommes également intéressé à la classe $\mathcal{C}_2(d)$, en obtenant un résultat asymptotique proche de la conjecture énoncée ci-dessous.

Conjecture 1 (graine k-optimale dans la classe $\mathcal{C}_2(d)$) *Soit d un entier désignant le poids de la graine. Pour tout $k \geq 4$, la graine de la forme $\pi(d) = \#^{d-2r-1}\text{-}\#^r\text{-}\#^{r+1}$ avec $r = \lfloor d/5 \rfloor$ est k-optimale parmi les graines de $\mathcal{C}_2(d)$.*

5.3.2 Expansion et contraction régulières des graines

Nous montrons dans cette partie que des graines qui résolvent des problèmes dont m est grand peuvent être obtenues à partir de graines résolvant des problèmes plus petits. La réciproque est également vraie. Les opérations de transformation associées sont des expansions et des contractions de graines.

Définition 28 (expansion régulière d'une graine ou d'une famille de graines) *Pour une graine π donnée, son* expansion i-régulière $i \otimes \pi$ *est obtenue en multipliant chacune de ses indices $\in I_\pi$ par i. Cette opération est équivalente à l'insertion de $i-1$ jokers entre deux positions successives le long de la graine. Pour une famille de graines F, l'expansion i-régulière de la famille $i \otimes F$ sera la famille des graines i étendues.*

EXEMPLE 33 : *expansion régulière d'une graine*

Si la graine π est donnée par $\{0,2,3,5\}$ (ou `#-##-#`), alors son expansion 2-régulière est donnée par $2 \otimes \pi = \{0,4,6,10\}$ (ou `#---#-#---#`).

Lemme 18 (résolution d'un problème (m,k) par expansion) *Si une famille de graines F résout un problème (m,k), alors le problème $(im,(i+1)k-1)$ est résolu par la famille F, et également résolu par la famille F_i, expansion i-régulière de F ($F_i = i \otimes F$).*

Preuve (**résolution d'un problème (m,k) par expansion**) Soit une similarité w du problème $(im,(i+1)k-1)$. Par le principe des pigeons et des nids, cette similarité contient au moins une sous-chaîne de longueur m avec au plus k erreurs : F résout donc le problème $(im,(i+1)k-1)$. Autrement, prenons i sous séquences disjointes de w, chacune étant composée des m lettres de positions égales modulo i. De la même manière, par le principe des pigeons et des nids, au moins une de ces sous séquences possède au plus k erreurs : $i \otimes F$ résout donc le problème $(im,(i+1)k-1)$. □

Le lemme suivant propose l'opposé du lemme 18, il établit que si une graine résout un problème relativement grand tout en ayant une structure régulière alors une solution pour un problème plus petit peut être obtenue par contraction régulière

Lemme 19 (résolution d'un problème (m,k) par contraction) *Si une famille $F_i = i \otimes F$ résout un problème (im,k), alors F résout le problème (im,k) ainsi que le problème $(m,\lfloor k/i \rfloor)$.*

Preuve (**résolution d'un problème (m,k) par contraction**) Il est même possible de montrer que F résout le problème (im,k) avec la restriction supplémentaire pour F de détecter la similarité par un *hit* d'une graine entièrement contenue dans un des intervalles $[1..m],[m+1..2m],\ldots,[(i-1)m+1..im]$. Nous utilisons pour cela la correspondance bijective du lemme 18 : étant donné une similarité w d'un problème (im,k), nous considérons les i sous-séquences disjointes w_j $(0 \leq j \leq i-1)$ du mot w construites en prenant les m positions égales à j modulo i, et en considérant la concaténation $w' = w_1 w_2 \ldots w_{i-1} w_0$.

Pour toute similarité w' du problème (im,k), son image inverse est détectée par F_i, et donc F détecte w' sur un des intervalles $[1..m],[m+1..2m],\ldots,[(i-1)m+1..im]$.

Pour tout similarité v du problème $(m,\lfloor k/i \rfloor)$, soit $w' = v^i$ et son image inverse w. Comme w' est détecté par F_i, v est détecté par F. □

EXEMPLE 34 : *illustration des lemmes 18-19*

Afin d'illustrer les deux lemmes, nous donnons les exemples suivants (repris de [25]). Les graines suivantes sont les seules de poids 12 à résoudre le problème $(m=50,k=5)$: `#-#-#---#-----#-#-#---#-----#-#-#---#` et `###-#--###-#--###-#`. La première est l'expansion 2-régulière de la deuxième. La deuxième est la seule graine de poids 12 qui résout le problème $(25,2)$

L'expansion régulière permet, dans certains cas, d'obtenir des solutions efficaces pour un problème avec m grand, en le réduisant à un problème plus petit pour lequel une solution optimale ou proche de l'optimal est connue.

5.3.3 Graines périodiques

Dans cette partie, nous étudions des graines particulières dont la structure est périodique : il s'agit de graines obtenues en itérant une graine plus petite. De telles graines apparaissent quelquefois comme des graines de poids maximal permettant de résoudre un problème (m, k). Cette périodicité de la structure dans le cadre du filtrage sans perte contraste avec l'aspect «aléatoire» de la structure des graines optimales dans le cadre du filtrage avec perte.

Notation (graine périodique) *Soit deux graines π_1,π_2 représentées par des mots sur un alphabet {#,-}. Nous oublions dans cette partie l'hypothèse qu'une graine doive avoir un caractère # en première et dernière position. Nous noterons $[\pi_1,\pi_2]^i$ la graine définie par $(\pi_1\pi_2)^i\pi_1$.*

EXEMPLE 35 : *graine périodique*

La graine proposée à la partie peut être écrite sous la forme `###-#--###-#--###-#` = $[$`###-#`,`--`$]^2$.

Nous considérerons également dans cette partie un nouveau type de problème (m, k) dit *cyclique* : les (m, k)-similarités de ce problème seront considérés modulo une permutation cyclique des lettres.

EXEMPLE 36 : *(m, k)-similarités linéaires/cycliques*

Soit un problème $(m = 7, k = 2)$ linéaire. Les (m, k)-similarités linéaires données par les mots `1101101`, `1110110`, `0111011`, `1011101`, `1101110`, `0110111` et `1011011` représentent une et une seule (m, k)-similarité dans la cas du problème (m, k) cyclique. Une graine détecte la (m, k)-similarité cyclique si elle détecte au moins l'une des (m, k)-similarités linéaires associées.

Définition 29 (problème (m, k) cyclique) *Une famille F résout un problème cyclique, si pour toute similarité w d'un problème (m, k), au moins l'une des graines de F détecte une des permutations cycliques de w.*

De manière naturelle, si F résout un problème (m, k), alors F résout le problème cyclique. Nous appellerons problème *linaire* le problème (m, k) classique de manière à le distinguer du problème cyclique.

Nous nous intéressons au cas d'une seule graine. Le lemme suivant montre qu'une graine plus petite répétée et concaténée peut permettre d'obtenir des solutions pour des problèmes plus larges ayant le même nombre d'erreurs

Lemme 20 (graine périodique et problème cyclique/linéaire) *Si une graine π résout un problème (m, k)* cyclique*, alors pour tout $i \geq 0$, la graine $\pi_i = [\pi, -^{(m-s(\pi))}]^i$ résout le problème linéaire $(m \cdot (i + 1) + s(\pi) - 1, k)$. Si $i \neq 0$, la réciproque est également vraie.*

Preuve (graine périodique et problème cyclique/linéaire) $\Rightarrow$ Soit u un similarité d'un problème $(m \cdot (i + 1) + s(\pi) - 1, k)$. Nous allons transformer u en une similarité u' associée au problème (m, k) cyclique de la manière suivante : Pour chaque position d'erreur ℓ sur u, nous

créons une erreur à la position ($\ell \mod m$) sur u', les autres positions de u' étant des matches . Il y a donc au plus k erreurs sur la similarité u'.

Comme π résout le problème cyclique (m, k), il est donc possible de trouver au moins une position j, $1 \leq j \leq m$, tel que π détecte u' de manière cyclique.

Nous montrons désormais que π_i détecte u à la position j (cette position est en effet valide car $1 \leq j \leq m$ et $s(\pi_i) = im + s(\pi)$). Comme les positions de matches sur u sont projetées modulo m sur des positions de # sur π, il n'y a donc pas d'erreurs sous les symboles # de π_i, et donc π_i détecte u.

$\Leftarrow$

Prenons la graine $\pi_i = [\pi, -^{(m-s(\pi))}]^i$ résolvant le problème $(m{\cdot}(i{+}1){+}s(\pi){-}1, k)$. Comme $i > 0$, nous prenons toutes les similarités du problème dont les erreurs sont placées dans l'intervalle $[m, 2m-1]$. Pour une similarité w ayant cette propriété, il existe une position j, $1 \leq j \leq m$, telle que π_i détecte w. Comme l'étendue de la graine est d'au moins $m + s(\pi)$, il y a donc une occurrence entière de π présente dans la fenêtre $[m, 2m-1]$, à une permutation cyclique associée près. Ceci implique que π résout le problème (m, k) cyclique. □

EXEMPLE 37 : *graine périodique*

Prenons la graine `###-#` qui résout le problème cyclique $(m = 7, k = 2)$. D'après le lemme 20, cela implique que pour tout $i \geq 0$, le problème linéaire $(m = 11 + 7i, k = 2)$ est résolu par la graine $[$`###-#`,`--`$]^i$ d'étendue $5 + 7i$. Il est intéressant de remarquer que pour $i = 1, 2, 3$, cette graine est optimale en ce sens que sont poids est maximal parmi toutes les graines résolvant ce problème.
De manière similaire, en appliquant le lemme 20, la graine périodique $[$`#####-##`,`---`$]^i$ résout le problème linéaire $(m = 18+11i, k = 2)$. Remarquons que son poids croît en $\frac{7}{11}m$, comparé à la graine du paragraphe précédent dont le poids était en $\frac{4}{7}m$. Ce n'est cependant pas la borne asymptotique optimale lorsque $m \to \infty$, comme nous allons le voir plus tard.
Le problème linéaire $(18 + 11i, 3)$ est résolu par $[$`###-#--#`,`---`$]^i$, puisque la graine `###-#--#` résout le problème cyclique $(11, 3)$. Pour $i = 1, 2$, la graine précédente est également de poids maximal pour l'ensemble des graines résolvant le problème $(18 + 11i, 3)$.

Une question soulevée par ces exemples est de savoir si le fait d'itérer une graine donnée peut apporter une solution asymptotique optimale, c'est-à-dire une graine de poids asymptotique optimal. Le théorème 3 établit une borne précise sur le poids de la graine optimale, pour un nombre fixé d'erreurs. Il donne également une réponse négative à la question précédente, puisqu'il montre que le poids maximal croît plus rapidement que toute fraction linéaire de la taille de la similarité.

Théorème 3 (comportement asymptotique sur un problème cyclique)
Soit k un entier fixé, et soit $w(m)$ le poids maximal d'une graine résolvant un problème cyclique (m, k). Alors $(m - w(m)) = \Theta(m^{\frac{k-1}{k}})$.

Preuve (**comportement asymptotique sur un problème cyclique**) Remarquons d'abord que toutes les graines résolvant un problème cyclique (m, k) peuvent être considérées comme des graines d'étendue m. Le nombre de jokers dans toute graine π est alors $n = m - w(\pi)$. Le théorème précise que le nombre minimal de jokers pour une graine résolvant un problème cyclique (m, k) est en $\Theta(m^{\frac{k-1}{k}})$ pour tout k.

Borne inférieure

Soit un problème cyclique (m, k) donné. Le nombre $D(m, k)$ de similarités cyclique distinctes associées au problème vérifie alors

$$\frac{\binom{m}{k}}{m} \leq D(m, k), \tag{5.8}$$

puisque chaque similarité linéaire a au plus m équivalents cycliques.

Soit π une graine et soit n le nombre de jokers de π. Soit $J_\pi(m, k)$ le nombre de similarités cycliques distinctes détectées par π. Remarquons que $J_\pi(m, k) \leq \binom{n}{k}$ et si π résout le problème cyclique (m, k), alors

$$D(m, k) = J_\pi(m, k) \leq \binom{n}{k} \tag{5.9}$$

A partir de (5.8) et (5.9), nous obtenons

$$\frac{\binom{m}{k}}{m} \leq \binom{n}{k}. \tag{5.10}$$

ce qui donne, en utilisant la formule de Stirling, $n(k) = \Omega(m^{\frac{k-1}{k}})$.

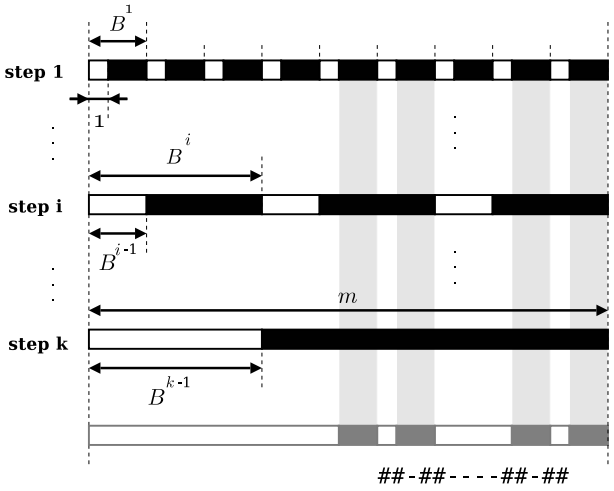


FIG. 5.1 – *Construction des graines π_i. Les jokers sont représentés en blanc.*

Borne supérieure

Pour prouver la borne supérieure, nous allons construire une graine π qui n'a pas plus de $k \cdot m^{\frac{k-1}{k}}$ jokers et qui résout le problème cyclique (m, k). Cette graine va être construite en prenant une graine composée uniquement de # et en substituant des jokers à certaines positions.

La graine de départ est nommée π_0, elle est d'étendue m et composée uniquement de #. La substitution de jokers est réalisée en k étapes, et après l'étape i, la graine alors obtenue est nommée π_i, la graine résultat étant donnée par $\pi = \pi_k$.

Soit $B = \lceil m^{\frac{1}{k}} \rceil$. π_1 est obtenue en substituant à π_0 des # par des jokers individuels avec une périodicité B, c'est-à-dire en les substituant aux positions $1, B+1, 2B+1, \ldots$. A l'étape 2, nous substituons dans π_1 des # par des intervalles de jokers contigus de longueur B, ayant une périodicité B^2 : des jokers sont substitués aux intervalles de positions $[1 \ldots B], [B^2+1 \ldots B^2+B], [2B^2+1 \ldots 2B^2+B], \ldots$.

En général, à l'étape i ($i \leq k$), nous substituons dans π_i des intervalles de B^{i-1} jokers avec une périodicité B^i aux positions $[1 \dots B^{i-1}], [B^i + 1 \dots B^i + B^{i-1}], \dots$ (voie la figure 5.1).

Remarquons que π_i est périodique et que la périodicité est B^i. Remarquons aussi que, pour chaque étape i, nous introduisons au plus $\lfloor m^{1-\frac{i}{k}} \rfloor$ intervalles de B^{i-1} jokers. Les recouvrements avec les jokers précédemment ajoutés font que chaque intervalle n'ajoute au plus que $(B-1)^{i-1}$ nouveaux jokers. Cela implique que le nombre total de jokers ajoutés à l'étape i est au plus de $m^{1-\frac{i}{k}} \cdot (B-1)^{i-1} \leq m^{1-\frac{i}{k}} \cdot m^{\frac{1}{k} \cdot (i-1)} = m^{\frac{k-1}{k}}$. Le nombre total de jokers dans π est donc inférieur ou égal à $k \cdot m^{\frac{k-1}{k}}$.

Afin de montrer que la graine ainsi construite résout le problème (m, k) cyclique, nous utilisons l'induction suivante sur i : pour toute similarité du problème (m, i) ($i \leq k$), π_i détecte u, c'est-à-dire qu'il existe un décalage cyclique de π_i tel que toutes les erreurs de u sont couvertes avec des jokers placés aux étapes $1, \dots, i$.

Pour $i = 1$, il est toujours possible de couvrir la seule erreur et décalant la graine π_1 d'au plus $(B-1)$ positions. En supposant que la propriété soit vraie pour $i-1$, nous montrons maintenant qu'elle est également vraie pour i.

Soit une similarité u du problème (m, i). Nous sélectionnons une erreur de u, et considérons les $i-1$ erreurs restantes : celles ci peuvent être recouvertes par π_{i-1} (hypothèse d'induction). Remarquons que π_{i-1} a une période de B^{i-1} et π_i diffère de π_{i-1} puisqu'il a au moins un intervalle contiguë de B^{i-1} jokers. Il est donc possible de décaler π_i de $j \cdot B^{i-1}$ positions de manière à recouvrir l'erreur sélectionnée avec l'intervalle contiguë de B^{i-1} jokers, tout en conservant la solution de π_{i-1} grâce à la structure périodique. On peut donc en conclure que π_i détecte u, et donc par induction que π résout le problème cyclique (m, k). □

En utilisant le résultat du théorème 3, il est possible d'obtenir la borne suivante sur le nombre de jokers dans le cas d'un problème *linéaire* (m, k).

Lemme 21 (comportement asymptotique des graines sur un problème linéaire) *Soit un entier k fixé. Soit $w(m)$ le poids maximal de la graine résolvant le problème linéaire (m, k). Alors $(m - w(m)) = \Theta(m^{\frac{k}{k+1}})$.*

Preuve (comportement asymptotique des graines sur un problème linéaire) Afin de montrer la borne supérieure, nous construisons une graine π qui résout le problème linéaire (m, k). Nous allons pour cela construire une graine à partir d'un motif périodique P . Soit un entier $l < m$ et soit P une graine qui résout le problème cyclique (l, k). Nous supposerons que l'étendue $s(P)$ est égale à l.

Pour un nombre réel $e \geq 1$, on définit P^e comme la graine linéaire de poids maximal et d'étendue au plus l^e obtenue en concaténant le motif P : la graine sera donc de la forme $P' \cdot P \cdots P \cdot P''$, avec P' et P'' correspondent respectivement à un préfixe et un suffixe de P. On peut en conclure, d'après la maximalité du poids de P^e, que $w(P^e) \geq e \cdot w(P)$.

Nous définissons $\pi = P^e$ pour un réel e à fixer : remarquons que si $e \cdot l \leq m - l$, alors π résout le problème linéaire (m, k). Nous choisirons de fixer la valeur $e = \frac{m-l}{l}$.

A partir de la preuve du théorème 3, nous avons sur le problème cyclique (l, k) la propriété $l - w(P) \leq k \cdot l^{\frac{k-1}{k}}$. Nous pouvons en déduire

$$w(\pi) = e \cdot w(P) \geq \frac{m-l}{l} \cdot (l - k \cdot l^{\frac{k-1}{k}}). \tag{5.11}$$

Si nous fixons

$$l = m^{\frac{k}{k+1}}, \tag{5.12}$$

nous obtenons alors

$$m - w(\pi) \leq (k+1)m^{\frac{k}{k+1}} - km^{\frac{k-1}{k+1}}, \tag{5.13}$$

et puisque k est considéré comme constante,

$$m - w(\pi) = \mathcal{O}(m^{\frac{k}{k+1}}). \tag{5.14}$$

La borne inférieure est obtenue de manière semblable à celle du théorème 3. Soit π une graine résolvant un problème linéaire (m, k), et soit $n = m - w(\pi)$. A partir de considérations de comptage, nous avons

$$\binom{m}{k} \leq \binom{n}{k} \cdot (m - s(\pi)) \leq \binom{n}{k} \cdot n, \tag{5.15}$$

ce qui implique que $n = \Omega(m^{\frac{k}{k+1}})$ pour k constant. □

Nous allons utiliser les résultats de graines simples, au cas de familles de graines composées de plusieurs graines. Nous introduisons tout d'abord une notation.

Notation (graine par décalage cyclique) *Nous noterons pour une graine π, $\pi_{[i]}$ comme son décalage cyclique à gauche de i symboles.*

EXEMPLE 38 : *graine par décalage cyclique*
Si π =`####-#-##--`, alors $\pi_{[5]}$ =`#-##--####-`.

Le lemme suivant donne une méthode pour construire les graines à partir d'un motif de graine permettant de résoudre un problème cyclique plus petit

Lemme 22 (famille de graines périodiques) *Supposons qu'une graine résolve un problème cyclique (m, k), et supposons que l'étendue de π soit égale à m (dans le cas contraire, nous complétons π à droite avec des jokers). Fixons un rationnel $i > 1$, et pour un $L > 0$ donné, considérons une liste de L entiers $0 \leq j_1 < \cdots < j_L < m$.*

Nous définissons une famille de graines $F =< \|\pi^i_{[j_l]}\| >_{l=1}^{L}$, avec $\|\pi^i_{[j_l]}\|$ représentant la graine $\pi^i_{[j_l]}$ en effaçant les symboles jokers *préfixes et suffixes. Soit $\delta(l)$ défini par $\delta(l) = (j_{l-1} - j_l) \mod m$ (ou, alternativement, $\delta(l) = (j_l - j_{l-1}) \mod m$) pour tout $l \in [1..L]$. Soit $m' = \max\{etendue(\|(\pi_{[j_l]})^i\|) + \delta(l)\}_{l=1}^{L} - 1$.*

Alors F résout le problème linéaire (m', k).

Preuve (**famille de graines périodiques**) La preuve est une extension de la preuve du lemme 20. Dans ce cas, les graines de la famille sont construites de telle façon que pour chaque instance d'un problème (m', k) linéaire, il existe au moins une graine qui satisfasse la propriété requise dans le lemme 20 et donc détecte cette instance □

Lorsque le lemme 22 est utilisé, l'intérêt est de prendre des entiers régulièrement espacés sur l'intervalle $[0..m]$, de telle sorte que les valeurs $s(||(\pi[j_l])^i||) + \delta(l)$ soient proches les unes des autres. Nous illustrons l'utilisation du lemme 22 par les deux exemples qui suivent.

EXEMPLE 39 : *famille de graines périodiques*
Prenons $m = 11$, $k = 2$. Soit le motif de graine π =`####-#-##--` permettant de résoudre le problème cyclique $(m = 11, k = 2)$. Si l'on choisit $i = 2$, $L = 2$, $j_1 = 0$, $j_2 = 5$, le théorème donne deux graines $\pi_1 = \|(\pi_{[0]})^2\|$ = `####-#-##--####-#-##` et $\pi_2 = \|(\pi_{[5]})^2\|$ =`#-##--####-#-##--####` d'étendue 20 et 21, avec $\delta(1) = 6$ et $\delta(2) = 5$. $\max\{20+6, 21+5\} - 1 = 25$. La famille $F = \{\pi_1, \pi_2\}$ résout ainsi le problème linéaire $(25, 2)$.

EXEMPLE 40 : *famille de graines périodiques*

Prenons $m = 11$, $k = 3$. Soit le motif π =`###-#--#---` permettant de résoudre le problème cyclique $(m = 11, k = 3)$. Si l'on choisit $i = 2$, $L = 2$, $j_1 = 0$, $j_2 = 4$, les deux graines obtenues sont $\pi_1 = \|(\pi_{[0]})^2\|$ =`###-#--#---###-#--#` (d'étendue 19) et $\pi_2 = \|(\pi_{[4]})^2\|$ =`#--#---###-#--#---###` (d'étendue 21), avec $\delta(1) = 7$ et $\delta(2) = 4$. $\max\{19+7, 21+4\} - 1 = 25$. Ainsi la famille $F = \{\pi_1, \pi_2\}$ résout le problème $(25, 3)$.

5.3.4 Conception heuristique des graines

Les résultats de la partie 5.3.1-5.3.3 permettent de construire des familles de graines lorsque k est petit comparé à m. Cependant, ils ne permettent pas de réaliser une conception systématique des familles.

Des approches par programmation linéaire ont été proposées pour réaliser une conception de familles de graines dans [108] et dans [18], respectivement dans le cadre de la recherche sur l'ADN et sur les protéines. Cependant aucune de ces approches n'était destinée au problème de filtrage sans perte.

Dans cette partie, nous décrivons une heuristique basée sur un algorithme génétique destiné à concevoir des familles qui résolvent des problèmes (m, k). Cette méthode est utilisée dans la partie 5.3.4 où des expériences sont réalisées, et emploie en particulier les méthodes de programmation dynamique proposées dans la partie 5.2.1.

L'algorithme utilise un algorithme génétique dont le principe est schématisé à la figure 5.2. L'algorithme améliore de manière continue les caractéristiques d'une population de graines (figure

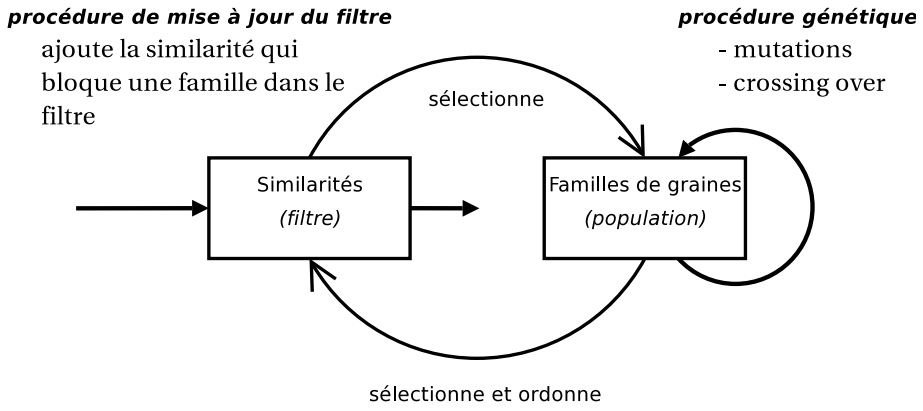


FIG. 5.2 – *Algorithme génétique de conception des familles de graines*

5.2 partie droite) jusqu'à ce qu'il trouve une famille de graines qui détecte toutes les similarités d'un problème (m, k).

La première étape de chaque itération consiste à sélectionner les familles de la population en les testant sur un ensemble de similarités considérées comme *difficiles* qui constitue un filtre (figure 5.2 partie gauche). Il s'agit de similarités qui ont été détectées par très peu de familles. Cet ensemble de similarités difficiles est réordonné et remis à jour de manière permanente selon le nombre de familles qui ne détectent pas ces similarités.

Pour cela, l'ensemble est mémorisé dans une liste dont le ré-ordonnancement est réalisé en utilisant le principe de *list-as-a-tree* [90] : chaque fois qu'une similarité de l'ensemble n'est pas détectée par une famille, elle est déplacée vers la racine, et son rang est alors divisé par deux.

Les familles qui parviennent à passer ce premier filtre sont évaluées. En particulier, le nombre de similarités non-détectées est calculé à l'aide de l'algorithme de la partie 5.2.3. La famille est conservée si elle produit un nombre plus petit de similarités non détectées que les familles actuellement connues. En général, une similarité non-détectée obtenue lors de ce calcul est ajouté en temps que feuille de l'arbre des similarités difficiles (deuxième moitié de la liste).

Afin de déterminer les graines à améliorer à l'intérieur d'une famille, nous calculons la contribution de chaque graine à l'aide de l'algorithme de programmation dynamique de la partie 5.2.4. Les graines qui ont la contribution la plus faible sont alors modifiées avec une plus grande probabilité. En général, la population de graines évolue par *mutation* et *crossing over* entre familles selon l'ensemble de similarités qu'elles ne détectent pas. Des familles aléatoires sont introduites dans la population afin d'essayer d'éviter les optima locaux.

La méthode heuristique décrite permet d'obtenir, sur des petites instances de problèmes, des familles de graines efficaces voire même quelquefois optimales en un temps de calcul raisonnable. Par exemple, en 10 essais de l'algorithme génétique, nous avons trouvé 3 des 6 familles existantes de deux graines de poids 14 résolvant le problème $(m = 25, k = 2)$. Le calcul total a pris moins d'une heure, à comparer à la semaine de calcul nécessaire pour tester de manière exhaustive toutes les paires de graines.

Il est intéressant de remarquer qu'un algorithme glouton randomisé (les familles sont complétées par ajout à chaque étape de la meilleure graine trouvée pour la famille) sur le même problème ne donne que des familles de trois et quelquefois quatre graines.

5.3.5 Travaux ultérieurs

L'optimisation de graines simples ou multiples pour le filtrage sans perte a fait récemment l'objet de nouvelles études sur le plan théorique. En particulier, deux articles ont été publiés.

F. Nicolas et E. Rivals [85] analysent la complexité du problème de décision suivant.

Problème 2 (Non Détection) *Soit une graine espacée π et deux entiers m et k vérifiant $0 \geq k \geq m$. Existe-t-il une similarité du problème (m, k) non détectée par π ?*

En particulier, ils prouvent que Non Détection est NP-complet [45], donc qu'il y a peu d'espoir d'améliorer les algorithmes de programmation dynamique de la partie 5.2.1.

D. Tsur et coauteurs [43] s'intéressent à différentes propriétés associées aux graines et au problème (m, k). Parmi les nombreux résultats proposés, une amélioration du lemme 21 est réalisée. Les auteurs montrent en particulier que, pour tout $k < \frac{1}{2}log(m)$, le poids maximal de la graine $w(k)$ pour un problème (m, k) donné vérifie :

$$m - w(k) = \begin{cases} \mathcal{O}(k \cdot m^{\frac{k}{k+1}}) & \text{si } k < log(log(m)), \\ \mathcal{O}(m^{\frac{k}{k+1}}) & \text{si } k \geq log(log(m)). \end{cases} \tag{5.16}$$

5.4 Résultats associés à la conception

Nous décrivons dans cette partie une expérience de conception de familles de graines réalisée sur deux problèmes (m, k). Pour chacun des problèmes, et pour une taille de famille donnée, nous avons calculé les familles qui résolvent le problème tout en atteignant le poids maximal possible (nous supposons que toutes les graines d'une famille doivent avoir le même poids). Nous notons également les méthodes utilisées pour la conception des graines (graines périodiques, algorithme génétique, recherche exhaustive).

taille	poids	graines de la famille	δ
1	$12^{e,p,g}$	`###-#--###-#--###-#`	$5.96 \cdot 10^{-8}$
2	$14^{e,p,g}$	`####-#-##--####-#-##` `#-##--####-#-##--####`	$7.47 \cdot 10^{-9}$

3	15^{p}	`#--##-#-######--##-#-##`
		`#-######--##-#-#####`
		`####--##-#-######--##`
			$2.80 \cdot 10^{-9}$

4	16^{p}	`###-##-#-###--#######`
		`##-#-###--#######-##-#`
		`###--#######-##-#-###`
		`#######-##-#-###--###`
			$9.42 \cdot 10^{-10}$

6	17^{p}	`##-#-##--#######-####-#`
		`#-##--#######-####-#-##`
		`#######-####-#-##--###`
		`###-####-#-##--#######`
		`####-#-##--#######-###`
		`##--#######-####-#-##--#`
			$3.51 \cdot 10^{-10}$

TAB. 5.1 – *Familles de graines pour le problème* $(25, 2)$

Les tables 5.1 et 5.2 donnent des résultats obtenus respectivement pour les problèmes $(25, 2)$ et $(25, 3)$. Les familles de graines périodiques sont marquées avec le symbole ^{p}, celles issues d'un algorithme génétique sont marquées avec le symbole ^{g}, et enfin celles obtenues lors d'une recherche exhaustive sont marquées avec le symbole ^{e}. Bien entendu seul le dernier cas garantit l'optimalité des familles. Les familles de graines périodiques sont présentées avec un décalage selon leur principe de construction (lemme 22).

Afin de comparer la sélectivité des différentes familles de graines pour un problème (m, k) donné, nous avons estimé la probabilité δ qu'au moins l'une des graines de la famille génère un *hit* sur une séquence de Bernoulli uniforme sur un alphabet de quatre lettres. Ce calcul a été réalisé en utilisant les formule d'inclusion-exclusion. Il est intéressant de voir que le simple passage d'une seule graine à une famille de deux graines permet d'obtenir un gain important

taille	poids	graines de la famille	δ
1	$8^{e,p,g}$	`###-#-----###-#`	$1.53 \cdot 10^{-5}$
2	10^{p}	`####-#-##--#---##` `##--#---####-#-##`	$1.91 \cdot 10^{-6}$

3	11^{p}	`#---####-#-##--#---##`
		`###-#-##--#---####`
		`##--#---####-#-##--#`
			$7.16 \cdot 10^{-7}$

4	12^{p}	`#---####-#-##--#---###`
		`###-#-##--#---####-#`
		`#-##--#---####-#-##--#`
		`##--#---####-#-##--#---#`
			$2.39 \cdot 10^{-7}$

TAB. 5.2 – *Familles de graines pour le problème* $(25, 3)$

concernant la sélectivité δ.

5.5 Sélection d'oligonucléotides à l'aide de famille de graines

Il existe un certain nombre de problèmes pour lesquels la garantie de trouver *toutes* les similarités vérifiant un critère donné est requise. Certains problèmes demandent, à l'inverse, une garantie de l'unicité d'un motif donné dans une séquence ou un ensemble de séquences. Ce motif ne doit donc pas, par exemple, avoir une occurrence sur toute autre séquence d'une base de données. Le problème de l'unicité d'un motif peut être facilement traité dans le cadre de la recherche de motifs exacts, mais si l'on veut obtenir une garantie avec une certaine distance « de sécurité », le problème alors posé est un problème de recherche de motifs approchés, plus difficile et coûteux.

Dans cette partie, nous sommes intéressés par une application particulière, instance du problème précédemment posé. Il s'agit de la sélection et de la conception d'oligonucléotides. Nous allons définir et expliquer les motivations biologiques liées aux oligonucléotides (partie 5.5.1), pour ensuite nous intéresser au problème de la conception d'oligonucléotides (partie 5.5.2). Nous décrirons le problème ainsi que la méthode proposée pour le résoudre à l'aide d'un filtrage sans perte par familles de graines (partie 5.5.2.1).

5.5.1 Oligonucléotides

Un oligonucléotide est un fragment d'ADN simple brin en général assez court (taille variant de 10 à 50 bases). Ce fragment d'ADN est conçu de manière à *s'hybrider*, c'est-à-dire se fixer sur une séquence complémentaire (voir partie 1.2, figure 1.2).

Les oligonucléotides sont par exemple utilisés dans les puces à ADN. Il s'agit de surface de verre sur lesquelles des milliers de copies d'un oligonucléotide simple brin sont fixées. Le motif de l'oligonucléotide copié est conçu de manière à s'hybrider avec une séquence d'ARN cible produite lors de l'expression d'un gène particulier. Divers procédés permettent de mesurer la quantité d'ARN fixée (fluorescence, marquage), ce qui donne une estimation de la concentration de l'ARN cible, et donc du degré d'activation du gène associé.

5.5.2 Problème de la conception d'oligonucléotides

Concevoir un oligonucléotide consiste à rechercher un motif adapté pour s'hybrider à un fragment d'un d'ARN cible. Il faut en particulier respecter la complémentarité du fragment pour que l'hybridation soit suffisamment solide. Mais le réel problème pour la conception d'un oligonucléotide est sa *spécificité* : un oligonucléotide spécifique doit s'hybrider avec un fragment d'un ARN cible, tout en ne s'hybridant pas avec tout autre ARN produit par la cellule.

Le problème de la conception d'un oligonucléotide est assez complexe, car il fait intervenir de nombreux paramètres dans le calcul de l'hybridation (température, salinité, etc.). Le calcul est généralement réalisé selon des paramètres empiriques permettant d'obtenir une approximation affine de l'énergie libre d'hybridation. Un modèle fréquemment utilisé et des paramètres associés ont été proposés dans [95] pour l'ADN, et dans [94, 112] concernant l'ARN.

Des filtres supplémentaires servent à éliminer les oligonucléotides qui possèdent une forte complémentarité palindromique. En effet, un oligonucléotide qui est un palindrome peut par hybridation se replier sur lui-même (phénomène d'épingle à cheveux) : il est donc inutilisable en pratique.

Nous associerons ici plus simplement la propriété d'hybridation à la notion de similarité sans considérer les propriétés annexes. Un motif d'oligonucléotide spécifique devra donc être similaire à un fragment de l'ARN cible, tout en étant suffisamment éloigné du reste des ARN produits par la cellule. Cette étape de conception basée sur la distance s'applique à un ensemble d'ARN présents dans la cellule qui représente, en terme de séquences, plusieurs dizaines de méga-bases. Elle est donc très coûteuse et doit donc être la plus efficace possible.

5.5.2.1 Précédentes approches

L'étape de présélection des oligonucléotides potentiels repose sur une notion de distance. Cette distance permet d'établir un critère de ressemblance entre un oligonucléotide potentiel et son ARN cible, ainsi qu'avec tout autre ARN susceptible d'être présent dans la cellule.

Cette distance est à spécifier, et de nombreux auteurs ont proposé des méthodes de conception d'oligonucléotides basées à la fois sur une distance particulière, une structure de données et un algorithme originaux. Les méthodes utilisant des tableaux de suffixes [76, 71, 93, 92] ou des arbres de suffixes [54] ont été développées. Elles permettent d'isoler les sous-mots exacts peu fréquemment répétés, susceptible d'être facteurs d'un oligonucléotide spécifique, et se basent en général sur la distance du plus long fragment commun, c'est-à-dire la *distance facteur* vue en partie 2.1.2.

Une méthode basée sur la distance de Hamming a été également proposée dans [111] pour la sélection d'oligonucléotides. Un des problèmes considérés dans [111] est la recherche d'oligonucléotides uniques : le problème est formulé de la manière suivante :

Problème 3 (sélection d'oligonucléotides candidates selon la distance de Hamming) *Étant donné un ensemble E de séquences, et deux entiers m, k ($m > k$), il s'agit de trouver l'ensemble des fragments f de taille m d'une séquence $s \in E$ qui sont* uniques, *ceux pour lesquels :*

$$\forall s' \in E \backslash s \text{ et } \forall i \in [1..|s'| - m + 1] \qquad dist_{\mathcal{H}}(f, s'[i..i+m-1]) > k$$

La solution proposée [111] est basée sur un algorithme combinatoire utilisant des k-mots autorisant des erreurs. Nous allons dans la partie suivante traiter le même problème à l'aide d'une famille de graines espacées.

5.5.2.2 Approche proposée

Les expériences que nous décrivons ici montrent que le filtrage sans perte à l'aide d'une famille de graines peut être une méthode efficace pour sélectionner des oligonucléotides candidates.

Dans cette partie, nous considérons le problème 3, et recherchons donc les fragments de taille m uniques selon un distance de Hamming d'au plus k. Dans l'expérience suivante la taille des fragments m a été fixée à 32 et le nombre d'erreurs k est fixé à 5. Pour le problème $(m = 32, k = 5)$, différentes familles de graines ont été conçues, et leur sélectivité a été estimée. Cet ensemble de graines est résumé dans la table de la figure 5.3, en utilisant les même conventions que dans les tables 5.1 et 5.2. Une famille, composée de 6 graines de poids 11 (figure 5.3 à droite) a été sélectionnée pour les expériences de filtrage.

Le procédé de filtrage a été appliqué à une base de données d'EST du riz, composée de 100015 séquences pour une longueur totale de 42.845.242 paires de bases [1].

Tous les fragments de longueur m qui détectent au moins un autre fragment avec au plus 5 erreurs de substitution ont été calculés. Le calcul a pris un peu plus d'une heure sur un

[1] source : `http ://bioserver.myongji.ac.kr/ricemac.html`, The Korea Rice Genome Database

taille de la famille	poids	δ
1	7^e	$6.10 \cdot 10^{-5}$
2	8^e	$3.05 \cdot 10^{-5}$
3	9^e	$1.14 \cdot 10^{-5}$
4	10^g	$3.81 \cdot 10^{-6}$
6	11^g	$1.43 \cdot 10^{-6}$
10	12^g	$5.97 \cdot 10^{-7}$

```
{ ####---#---------#---#--#### ,
  ###--#--##--------#-#### ,
  ####----#--#--##-### ,
  ###-#-#---##--#### ,
  ###-##-##--#-#-## ,
  ####-##-#-#### }
```

FIG. 5.3 – *familles de graines pour le problème* $(32, 5)$ *et famille utilisée (6 graines de poids 11)*

Pentium^{TM} IV de fréquence 3GHz. Avant d'appliquer le filtrage à l'aide de la famille associée, un pré-filtrage rapide a été réalisé à l'aide d'une graine espacée de poids 16 afin de détecter avec une forte sélectivité toutes les régions quasi-identiques.

65% de la base de données ont été éliminés après ce pré-filtrage. La famille de graines a ensuite filtré respectivement 22% de la base de données, laissant les 13% restants comme oligonucléotides candidats.

Conclusion

Nous nous sommes intéressés dans cette thèse à la recherche de similarités dans les séquences d'ADN, à l'aide de filtres basés sur différents modèles de graines. Le but était de réaliser, soit un filtrage avec perte qui soit à la fois sensible et sélectif, soit un filtrage sans perte le plus sélectif possible.

Dans un premier temps, nous avons recensé les méthodes de recherche de similarités existantes avant de considérer le filtrage à l'aide de graines espacées. Nous avons proposé une méthode permettant de calculer la sensibilité de graines espacées sur des similarités (alignements) dites *homogènes* afin de prendre en compte la spécificité des algorithmes heuristiques de recherche. Nous avons ensuite proposé une méthode permettant de modéliser les différentes composantes qui entrent dans le calcul de la sensibilité à l'aide d'automates déterministes et d'automates probabilistes. Cette approche a été utilisée sur un nouveau modèle de graine appelé *graine sous-ensemble*. Dans le cas des séquences génomiques, ce modèle correspond aux *graines à transitions*. Le logiciel HEDERA a été développé dans ce cadre afin de concevoir des graines efficaces selon les différents modèles. Le logiciel YASS a été développé afin de donner la possibilité d'utiliser en pratique ces graines sur de vraies séquences génomiques.

Dans un second temps, nous nous sommes intéressés à une méthode de filtrage sans perte à l'aide de *familles de graines espacées*. Nous avons proposé dans ce cadre des méthodes de conception des familles, des algorithmes pour mesurer leurs caractéristiques. Nous avons également obtenu des résultats sur le poids asymptotique des graines lorsque la taille des similarités tend vers l'infini, ainsi que certaines propriétés d'optimalité sur la classe des graines à un joker.

* *
*

Les perspectives de ce travail concernent aussi bien des aspects théoriques que pratiques.

D'un point de vue pratique, nous voudrions utiliser YASS et le programme de clustering associé afin de réaliser une étude plus approfondie de certains génomes de manière à déterminer de nouveaux éléments partiellement conservés sur l'ADN. Ce travail a fait l'objet d'une première étude [104] sur différentes souches de la bactérie *Neisseria Meningitidis* mais son champ d'application est vaste et doit donc être poursuivi sur d'autres espèces, soit en considérant distinctement une espèce et en analysant ses souches, soit en considérant des familles d'espèces. Les éléments potentiellement recherchés peuvent par exemple être des ARN de régulation encore non-identifiés.

Nous voudrions également adapter le modèle des graines sous-ensemble à l'alphabet des protéines, tout en conservant une certaine souplesse d'utilisation, en particulier une tolérance aux erreurs moins localisée sur la graine : les graines vecteurs possèdent cet avantage mais elles n'ont pas de bonne propriétés quant à leur algorithme de mesure de la sensibilité, ou de création d'index. Les graines sous-ensemble multiples pourraient être un bon compromis. Ce travail a déjà

fait l'objet d'une première étude de DEA effectué par V. Deleplace [41] dans l'équipe ADAGE. Les problèmes qui se posent alors sont liés à la définition des alphabets d'alignements et des alphabet des graines : de nombreuses combinaisons sont en effet possibles, mais le choix ne peut actuellement pas être déterminé **avant** de concevoir les meilleures familles de graines.

Concernant les problèmes plus théoriques, la conception des graines reste encore peu comprise et analysée, et l'étude des corrélations des graines se révèle délicate [109], même pour l'obtention d'algorithmes d'approximation efficaces. Différentes idées sont à étudier ici afin d'obtenir des méthodes de génération de graines plus efficaces que celles basées sur des principes de programmation linéaire [108]. A plus court terme, les graines sous-ensemble posent également un problème, de part l'explosion combinatoire qu'elles engendrent : se pose alors le choix de la composition des lettres qui entrent dans la construction de la graine optimale. Et ce, en prenant en compte la répartition des lettres de l'alphabet des alignements, et par exemple un cadre asymptotique en perspective.

Annexe A

Improved hit criteria for DNA local alignment

(*cet article a été publié dans le journal BMC Bioinformatics en octobre 2004 [88], il présente les améliorations apportées par le logiciel* YASS)

Background : The hit criterion is a key component of heuristic local alignment algorithms. It specifies a class of patterns assumed to witness a potential similarity, and this choice is decisive for the selectivity and sensitivity of the whole method.

Results : In this paper, we propose two ways to improve the hit criterion. First, we define the *group criterion* combining the advantages of the single-seed and double-seed approaches used in existing algorithms. Second, we introduce *transition-constrained seeds* that extend spaced seeds by the possibility of distinguishing transition and transversion mismatches. We provide analytical data as well as experimental results, obtained with the YASS software, supporting both improvements.

Conclusions : Proposed algorithmic ideas allow to obtain a significant gain in sensitivity of similarity search without increase in execution time. The method has been implemented in YASS software available at http://www.loria.fr/projects/YASS/.

A.1 Background

Sequence alignment is a fundamental problem in Bioinformatics. Despite of a big amount of efforts spent by researchers on designing efficient alignment methods, improving the alignment efficiency remains of primary importance. This is due to the continuously increasing amount of nucleotide sequence data, such as EST and newly sequenced genomic sequences, that need to be compared in order to detect similar regions occurring in them. Those comparisons are done routinely, and therefore need to be done very fast, preferably instantaneously on commonly used computers. On the other hand, they need to be precise, i.e. should report all, or at least a vast majority of interesting similarities that could be relevant in the underlying biological study. The latter requirement for the alignment method, called the *sensitivity*, counterweights the speed requirement, usually directly related to the *selectivity* (called also *specificity*) of the method. The central problem is therefore to improve the trade-off between those opposite requirements.

The Smith-Waterman algorithm [97] provides an exact algorithmic solution to the problem of computing optimal local alignments. However, its quadratic time complexity has motivated the creation of rapid heuristic local alignments tools. A basic idea behind all heuristic algorithms is to focus only on those regions which share some patterns, assumed to witness (or to *hit*) a potential similarity. Those patterns are formed by *seeds* which are small strings (usually up to 25 nucleotides) that appear in both sequences. FASTA [74] and BLAST [5, 6] are well-known examples of such methods. BLAST is currently the most commonly used sequence alignment tool, and is a kernel of higher-level search tools, such as PSI-BLAST [6] for instance.

More recently, several new alignment methods have been proposed, such as BLAT [60], PATTERNHUNTER [75], LAGAN [20], or BLASTZ [96], to name a few. The improvement brought by all those tools results from new more efficient *hit criteria* that define which pattern shared by two sequences is assumed to witness a potential alignment. Two types of improvements can be distinguished. On the one hand, using two or more closely located smaller seeds instead of one larger seed has been shown to improve the sensitivity/selectivity trade-off [6, 60, 75], especially for detecting long similarities. On the other hand, new seed models have been proposed, such as spaced seeds [75], seeds with errors [60], or vector seeds [16].

In this paper, we propose further improvements in both those directions. In the first part (Section *Group hit criterion*A.2.1), we propose a new flexible and efficiently computable hit criterion, called *group criterion*, combining the advantages of the single-seed ([5]) and multi-seed ([74, 6, 60, 75]) criteria. In the second part (Section *Generalized seed models*A.2.2), we propose a new more expressive seed model which extends the spaced seed model of PATTERNHUNTER [75] by the possibility of distinguishing transition and transversion mismatches. We show that this allows to obtain a further gain in sensitivity on real genomic sequences, usually rich in transition mutations. Both proposed improvements have been implemented in YASS software [86], used in the experimental part of this work.

A.2 Results

A.2.1 Group hit criterion

The first preparatory step of most heuristic alignment algorithms consists of constructing a hash table of all *seeds* occurring in the input sequences. In this section, we assume that a seed of *weight* k is a word consisting of k contiguous nucleotides (k-word), more general notions of seed will be considered in the next Section.

In the simplest case, implemented in the early version of BLAST [5], an individual seed occurring in both sequences acts as a *hit* of a potential alignment. It triggers the *X-drop* algorithm trying to extend the seed to a so-called High-scoring Segment Pair (HSP), used then to obtain a larger final alignment. GAPPED-BLAST [6] proposes a *double-seed* criterion that defines a hit as two non-overlapping seeds occurring at the same dotplot diagonal within a fixed-size window. This allows to considerably increase the selectivity with respect to the single-seed approach, and at the same time to preserve, and even to improve, the sensitivity on large similarities. On the other hand, GAPPED-BLAST is less sensitive on short and middle-size similarities of weak score. (We will show this more formally at the end of this Section.) Most of the existing alignment programs [60, 75] use either a single-seed or a double-seed hit criterion.

Here we propose a new flexible hit criterion defining a hit as a group containing an arbitrary number of possibly overlapping seeds, with an additional constraint on the minimal number of matching nucleotides. The seeds of the same group are assumed to belong to the same similarity, and therefore should be proximate to each other. In contrast to other multi-seed hit criteria

[6, 60, 75], we don't require seeds to occur at the same dotplot diagonal but at close diagonals, to account for possible indels. The possible placement of seeds is controlled by parameters computed according to statistical models that we describe now.

A.2.1.1 Group criterion

A hit criterion defines a pattern which is considered as an evidence of a potential similarity. Every time this pattern is found, its extension is triggered to compute a potential larger alignment. The extension is usually done via a dynamic programming algorithm and is a costly step. The hit criterion should be *selective* enough to avoid spurious extensions and, on the other hand, should be *sensitive* to detect as many relevant similarities as possible.

The hit criterion we propose is based on a *group of seeds* verifying conditions (A.1), (A.2) (see Section *Methods*). By the considered statistical analysis, this ensures a good sensitivity. However, many groups will contain a single seed or two strongly overlapping seeds, that either belong to a similarity with a low score, or do not belong to any similarity at all (i.e. don't belong to an alignment with a sufficiently high score).

To cope with this problem, we introduce an additional criterion that selects groups that will be actually extended. The criterion, called *group criterion*, is based on the *group size* defined as the *minimal number of matching individual nucleotides in all seeds of the group*. The group size can be seen as a parameter specifying the maximal overlap of the seeds of a group. For example, if the group size is $k+1$, then no constraint on the overlap is imposed, i.e. any group containing two distinct seeds forms a hit. If the group size is $2k$, then the group must contain at least two non-overlapping seeds, etc.

Allowing overlapping seeds is an important point that brings a flexibility to our method. Note that other popular multi-seed methods [6, 60] consider only non-overlapped seeds. Allowing overlapped seeds and controlling the overlap with the group size parameter offers a trade-off between a single-seed and a multi-seed strategies. This increases the sensitivity of the usual multi-seed approach without provoking a tangible increase in the number of useless extensions. In the next section, we will provide quantitative measures comparing the sensitivity of the Yass group criterion with Blast and Gapped-Blast.

A.2.1.2 Some comparative and experimental data

In this section, we adopt the following experimental setup to estimate the sensitivity of the Yass group criterion compared to other methods. We first set a match/mismatch scoring system, here fixed to +1/-3 (default `NCBI-BLAST` values). The main assumption is that the sensitivity is estimated as the probability of hitting a random *gapless alignments of a fixed score*. Moreover, to make this model yet more close to reality, only *homogeneous alignments* are considered, i.e. alignments that don't contain proper sub-alignments of bigger score (see [63]). For a given alignment length, all homogeneous alignments are assumed to have an equal probability to occur.

In this setting, we computed the hit probability of a single-seed criterion with seed weight 11 (default for Blast) and compared it with multi-seed criteria of Gapped-Blast and Yass for seed weight 9 (default for Gapped-Blast). For Yass, the group size was set to 13. Figure 1 shows the probability graphs for alignment score 25.

Comparing Blast and Gapped-Blast, the former is more sensitive on short similarities (having higher identity rate), while the latter is more sensitive on longer similarities, in which two close non-overlapping runs of 9 matches are more likely to occur than one run of 11 matches.

The YASS group criterion combines the advantages of both : it is more sensitive than the single-seed criterion even for short similarities, and than the non-overlapping double-seed criterion for large similarities (Figure 1).

Note, however, that for the chosen parameters, the YASS criterion is slightly less selective than that of GAPPED-BLAST as it includes any two non-overlapping seeds but also includes pairs of seeds overlapped by at most 5 bp. The selectivity can be estimated by the probability of a hit at a given position in a random uniform Bernoulli sequence (see [60]). For YASS, this probability is $2.1 \cdot 10^{-8}$, which improves that of BLAST ($2.4 \cdot 10^{-7}$) by more than ten. For GAPPED-BLAST, this probability is $7.3 \cdot 10^{-9}$. To make an accurate sensitivity comparaison of YASS and GAPPED-BLAST, parameters should be set so that both algorithms have the same selectivity level.

To compare the sensitivity of YASS and GAPPED-BLAST for an equal selectivity level, we chose a parameter configuration such that both algorithms have the same estimated selectivity (10^{-6}). This is achieved with seed weight 8 for GAPPED-BLAST and group size 11 for YASS (while keeping seed weight 9). In this configuration, and for sequences of score 25, YASS turns out to be considerably more sensitive on sequences up to 80 bp and is practically as sensitive as GAPPED-BLAST on longer sequences (data not shown). At the same time, YASS is more time efficient in this case, as GAPPED-BLAST tends to compute more spurious individual seeds that are not followed by a second hit, which takes a considerable part of the execution time. This is because the YASS seed is larger by one nucleotide, and the number of spurious individual seeds computed at the first step is then divided by 4 on large sequences.

Compared to the single-seed criterion of BLAST, the YASS group criterion is both more selective (group size 13 vs single-seed size 11) and more sensitive *for all alignment lengths*, as soon as the alignment score is 25 or more. If the score becomes smaller, both criteria yield an unacceptably low sensitivity, and the seed weight has then to be decreased to detect those similarities.

Finally, we point out another experiment we made to bring more evidence that the group criterion captures a good sensitivity/selectivity trade-off. We monitored the partition of the execution time between the formation of groups and their extension by dynamic programming (data not shown). It appeared that YASS spends approximately equal time on each of the two stages, which gives a good indication that it provides an optimal distribution between the two complementary parts of the work.

A.2.2 Generalized seed models

So far, we defined seeds as k-words, i.e. short strings of *contiguous* nucleotides. Recently, it has been understood that using *spaced seeds* allows to considerably improve the sensitivity. A spaced seed is formed by two words, one from each input sequence, that match at positions specified by a fixed *pattern* – a word over symbols `#` and `_` interpreted as a match and a don't care symbol respectively. For example, pattern `##_#` specifies that the first, second and fourth positions must match and the third one may contain a mismatch.

PATTERNHUNTER [75] was the first method that used carefully designed spaced seeds to improve the sensitivity of DNA local alignment. In [27], spaced seeds have been shown to improve the efficiency of *lossless* filtration for approximate pattern matching, namely for the problem of detecting all matches of a string of length m with q possible substitution errors (an (m, q)-problem). The use of some specific spaced seeds for this problem was proposed earlier in [91]. Yet earlier, random spaced seeds were used in `FLASH` software [28] to cover sequence similarities, and the sensitivity of this approach was recently studied in [21].

The advent of spaced seeds raised new questions : How to choose a good seed for a local

alignment algorithm ? How to make a precise estimation of the seed goodness, or more generally, of a seed-based local alignment method ? In [59], a dynamic programming algorithm was proposed to measure the hit probability of a seed on alignments modeled by a Bernoulli model. In the lossless case, an analogous algorithm that allows to test the seed completeness for an (m, q)-problem was proposed in [27]. The algorithm of [59] has been extended in [15] for hidden Markov models on order to design spaced seeds for comparing homologous coding regions. Another method based on finite automata was proposed in [59]. A complementary approach to estimate the seed sensitivity is proposed in [63]. Papers [32, 31] present a probabilistic analysis of spaced seeds, as well as experimental results based on the Bernoulli alignment model.

Other extensions of the contiguous seed model have been proposed. BLAT [60] uses contiguous seeds but allows one error at any of its positions. This strategy is refined in BLASTZ [96] that uses spaced seeds and allows one transition substitution at any of match positions. An extension, proposed in [16], enriches the PATTERNHUNTER spaced seeds model with a scoring system to define a hit.

Here we propose a new *transition-constrained seed* model. Its idea is based on the well-known feature of genomic sequences that transition mutations (nucleotide substitutions between purins or between pyrimidins) occur relatively more often than transversions (other substitutions). While in the uniform Bernoulli sequence transitions are twice less frequent than transversions, in real genomic sequences this ratio is often inverted. For example, matrices computed in [30] on mouse and human sequences imply that the transition/transversion rate (hereafter ti/tv) is greater than one on average. Transitions are much more frequent than transversions in coding sequences, as most of silent mutations are transitions. ti/tv ratio is usually greater for related species, as well as for specific DNA (mitochondrial DNA for example).

Transition-constrained seeds are defined on the ternary alphabet {`#`,`@`,`_`}, where `@` stands for a match or a transition mismatch (`A` $\leftrightarrow$ `G`, `C` $\leftrightarrow$ `T`), and `#` and `_` have the same meaning as for spaced seeds. The *weight of a transition-constrained seed* is defined as the sum of the number of `#`'s plus half the number of `@`'s. This is because a transition carries one bit of information while a match carries two bits.

Note that using transition-constrained seeds is perfectly compatible with the group criterion described in Section *Group criterion*A.2.1. The only non-trivial algorithmic issue raised by this combination is how to efficiently compute the group size during the formation of groups out of found seeds. In YASS, this is done via a special finite automaton resulting from the preprocessing of the input seed.

A.2.2.1 Transition-constrained seeds for Bernoulli alignment model

To estimate the detection capacity of transition-constrained seeds, we first use the Bernoulli alignment model, as done in [75, 32, 31]. We model a gapless alignment by a Bernoulli sequence over the ternary match/transition/transversion alphabet with the match probability 0.7 and the probabilities of transition/transversion varying according to the ti/tv ratio. The sequence length is set to 64, a typical length of a gapless fragment in DNA alignments. We are interested in seed weights between 9 and 11, as they represent a good sensitivity/selectivity compromise.

Table 1 compares the sensitivity of the best spaced seeds of weight 9, 10 and 11, reported in [31], with some transition-constrained seeds, assuming that transitions and transversions occur with equal probability 0.15. The transition-constrained seeds have been obtained using a stepwise Monte-Carlo search, and the probabilities have been computed with an algorithm equivalent to that of [59]. The table shows that transition-constrained seeds are more sensitive with respect to this model.

A natural question is the efficiency of transition-constrained seeds depending on the ti/tv ratio. This is shown in Figure 2. The left and right plots correspond to the seeds from Table 1 of weight 9 and 10 respectively. The plots show that the sensitivity of transition-constrained seeds greatly increases when the ti/tv ratio is over 1, which is typically the case for real genomic sequences.

A.2.2.2 Transition-constrained seeds for Markov alignment model

Homologous coding sequences, when aligned, usually show a regular distribution of errors due to protein coding constraints. In particular, transitions often occur at the third codon position, as these mutations are almost always silent for the resulting protein. Markov models provide a standard modeling tool to capture such local dependencies. In the context of seed design, papers [59, 15] proposed methods to compute the hit probability of spaced seeds with respect to gapless alignments specified by (Hidden) Markov models.

To test whether using transition-constrained seeds remains beneficial for alignments specified by Markov models, we constructed a Markov model of order 5 out of a large mixed sample of about 100 000 crossed alignments of genomic sequences of distantly related species (*Neisseria Meningitidis, S.Cerevisiae, Human X chromosome, Drosophila*). The alignments were computed with different seeds of small weight, to avoid a possible bias caused by a specific alignment method. We then designed optimal spaced and transition-constrained seeds of weight 9-11 with respect to this Markov model. Table 2 shows the results of this computation providing evidence that transition-constrained seeds increase the sensitivity with respect to this Markov model too.

A.2.3 Experiments

A.2.3.1 Seed experiments

In order to test the detection performance of transition-constrained seeds on real genomic data, we made experiments on full chromosomic sequences of *S.Cerevisiae* (chromosomes IV, V, IX, XVI) and *Neisseria meningitidis* (strains MC58 and Z2491). The experiments were made with our YASS software [86] that admits user-defined transition-constrained seeds and implements the group criterion described in Section *Group criterion*A.2.1. The experiments used seeds of weight 9 and 11, obtained on Bernoulli and Markov models (reported in Tables 1 and 2). The search was done using group size 10 and 12 respectively for seed weight 9 and 11 (option `-s` of YASS). This means that at least two distinct seeds were required to trigger the extension, with no additional constraint on their overlap, which is equivalent to the double-seed criterion of PATTERNHUNTER. The scoring system used was $+1/-1$ for match/mismatch and $-5/-1$ for gap opening/extension. Both strands of input chromosomes has been processed in each experiment (`-r 2` option of YASS).

For each comparison, we counted the number of computed alignments with E-value smaller than 10^{-3}. Table 3 reports some typical results of this experiment. They confirm that using transition-constrained seeds does increase the search sensitivity. A side (non-surprising) observation is that, in all tests, the seed designed on the Markov model turns out to be more efficient than the one designed on the Bernoulli model.

Note that the similarity search can be further improved by using transition-specific scoring matrices (for example, PAM Transition/ Transversion matrices or matrices designed for specific comparisons [30]) rather than uniform matches/mismatch matrices, and transition-constrained spaced seeds are more likely to detect alignments highly scored by those matrices.

Another advantage of transition-constrained seeds comes from the fact that they are more robust with respect to the GC/AT composition bias of the genome. This is because purins and pyrimidins remain balanced in GC- or AT-rich genomes, and one match carries less information (is more likely to occur "by chance") than two match-or-transition pairs.

A.2.3.2 Program experiments

A series of comparative tests has been carried out to compare the sensitivity with traditional approaches. Several complete bacterial genomes ranging from 3 to 5 Mb have been processed against each other using both YASS and the `bl2seq` programs (`NCBI BLAST` package 2.2.6.). The tests used the scoring system +1/-1 for match/mismatch and -5/-1 for gap opening/extension. The threshold E-value for the output was set to 10 (default BLAST value), and the sequence filtering was disabled. YASS was run with its default seed pattern `#@#__##__#_##@#` of weight 9 which provides a good compromise in detecting similarities of both coding and non-coding sequences.

For each test, the number of alignments with E-value less than 10^{-6} found by each program, and the number of exclusive alignments were reported. By "exclusive alignment", we mean every alignment of E-value less than 10^{-6} that does not share a common part (do not overlap on both sequences) with any alignment found by the other program. To take into account a possible bias caused by splitting alignments into smaller ones (X-drop effect), we also computed the total length of exclusive alignments, found by each program.

Experiments are summarized in Table 4 and show that within a generally smaller execution time, YASS detects more exclusive similarities that cover about twice the overall length of those found by `bl2seq`. The gain in execution time increases when the sequence length gets larger.

A.3 Conclusions

In this paper, we introduced two independent improvements of hit criteria for DNA local alignment. The *group criterion*, based on statistical DNA sequence models, combines the advantages of both single-seed and double-seed criteria. *Transition-constrained seeds* account for specificities of real DNA sequences and allow to further increase the search sensitivity with respect to spaced seeds. Both options have been implemented in YASS software available at `http://www.loria.fr/projects/YASS/`.

Transition-constrained seeds could be further extended using the idea of vector seeds [16], i.e. by assigning weights to each seed position, but also to each type of mutation. This would give a more general mechanism to account for the information brought by different mutations. But the model is also more flexible, an thus involves a larger search space to design seeds.

Another new direction for further improving the efficiency is a simultaneous use of several seed patterns [72, 99, 64], complementing the sensitivity of each other. However, designing such families is also hard problem, due to the involved search space.

A.4 Methods

A.4.1 Statistical analysis

We first introduce some notations used in this section. Let S_1 and S_2 be the input sequences of length m and n respectively. Each of them can be considered as a succession of $m-k+1$ (respectively $n-k+1$) substrings of length k, called *k-words*. If a k-word of S_1 matches another

k-word of S_2, i.e. $S_1[i..i+k-1] = S_2[j..j+k-1]$ for some $i \leq m$ and $j \leq n$, then these two k-words form a *seed* denoted $<i, j>$. Two functions on seeds are considered : For a seed $<i, j>$, the *seed diagonal* $d(<i, j>)$ is $m+j-i$. It can be seen as the distance between the k-words $S_1[i..i+k-1]$ and $S_2[j..j+k-1]$ if S_2 is concatenated to S_1, For two seeds $<i_1, j_1>$ and $<i_2, j_2>$, where $i_1 < i_2$ and $j_1 < j_2$, the *inter-seed* distance $D(<i_1, j_1>, <i_2, j_2>)$ is the maximum between $|i_2 - i_1|$ and $|j_2 - j_1|$. The problem considered in this Section is to derive conditions under which two seeds are likely to be a part of *the same* alignment, and therefore should be grouped together. More precisely, we want to be able to compute parameters ρ and δ such that two *seeds* $<i_1, j_1>$ and $<i_2, j_2>$ have a probability $(1 - \varepsilon)$ to belong to the same similarity iff

$$D(<i_1, j_1>, <i_2, j_2>) \leq \rho, \tag{A.1}$$

$$|d(<i_1, j_1>) - d(<i_2, j_2>)| \leq \delta. \tag{A.2}$$

The first inter-seed condition insures that the seeds are close enough to each other. The second seed diagonal condition requires that in both seeds, the two k-words occur at *close diagonals*.

We now describe statistical models used to compute parameters ρ and δ.

Bounding the inter-seed distance Consider two homologous DNA sequences that stem from a duplication of a common ancestor sequence, followed by independent individual substitution events. Under this assumption, the two sequences have an equal length and their alignment is a sequence of matched and mismatched pairs of nucleotides. We model this alignment by a Bernoulli sequence with the probability p for a match and $(1 - p)$ for a mismatch. To estimate the inter-seed shift D_k, we have to estimate the distance between the starts of two *successive runs of at least k matches* in the Bernoulli sequence. It obeys the geometric distribution of order k called the *Waiting time distribution* [3, 10] :

$$\mathcal{P}[D_k = x] = \begin{cases} 0 & \text{for } 0 \leq x < k \\ p^k & \text{for } x = k \\ (1-p)p^k\big(1 - \sum_{i=0}^{x-k-1} \mathcal{P}[D_k = i]\big) & \text{for } x > k \end{cases}$$

Using this formula, we compute ρ such that the probability $\mathcal{P}[D_k \leq \rho]$ is $(1 - \varepsilon)$ for some small ε.

Note that the Waiting time distribution allows us to estimate another useful parameter : the number of runs of matches of length at least k inside a Bernoulli sequence of length x. In a Bernoulli sequence of length x, the probability of the event $I_{p,x,r}$ of having exactly r *non-overlapping* runs of matches of length at least k is given by the following recursive formula :

$$\mathcal{P}[I_{p,x,r}] = \mathcal{P}[I_{p,x-1,r}] + p^k(1-p)(\mathcal{P}[I_{p,x-k-1,r-1}] - \mathcal{P}[I_{p,x-k-1,r}])$$

This gives the probability of having exactly r *non-overlapping* seeds of length at least k inside a repeat of size x. The recurrence starts with $r = 0$, in which case $\mathcal{P}[I_{p,x,0}] = \mathcal{P}[D_k > x - k]$ and is computed through the Waiting time distribution.

The distribution $\mathcal{P}[I_{p,x,r}]$ allows us to infer a lower bound on the number of non-overlapping seeds expected to be found inside a similarity region. In particular, we will use this bound as a first estimate of the group criterion introduced later.

Bounding the seed diagonal variation *Indels* (nucleotide insertions/deletions) are responsible for a diagonal shift of seeds viewed on a dotplot matrix. In other words, they introduce a

possible difference between $d(<i_1, j_1>)$ and $d(<i_2, j_2>)$. To estimate a typical shift size, we use a method similar to the one proposed in [10] for the search of tandem repeats.

Assume that an indel of an individual nucleotide occurs with an equal probability q at each of l nucleotides separating two consecutive seeds. Under this assumption, estimating the diagonal shift produced by indels is done through a discrete one-dimensional *random walk* model, where the probability of moving left or right is equal to q, and the probability of staying in place is $1-2q$. Our goal is to bound, with a given probability, the deviation from the starting point.

The probability of ending the random walk at position i after l steps is given by the following sum :

$$\sum_{j=0}^{(l-i)/2} \frac{l!}{j!(j+i)!(l-(2j+i))!} q^{2j+i}(1-2q)^{l-(2j+i)}$$

A direct computation of multi-monomial coefficients quickly leads to a memory overflow, and to circumvent this, we use a technique based on generating functions. Consider the function $P(x) = qx+(1-2q)+\frac{q}{x}$ and consider the power $P^l(x) = a_l.x^l+\cdots+a_{-l}.x^{-l}$. Then the coefficient a_i computes precisely the above formula, and therefore gives the probability of ending the random walk at position i after l steps. We then have to sum up coefficients a_i for $i = 0, 1, -1, 2, -2, \ldots, l, -l$ until we reach a given threshold probability $(1-\varepsilon)$. The obtained value l is then taken as the parameter δ used to bound the maximal diagonal shift between two seeds.

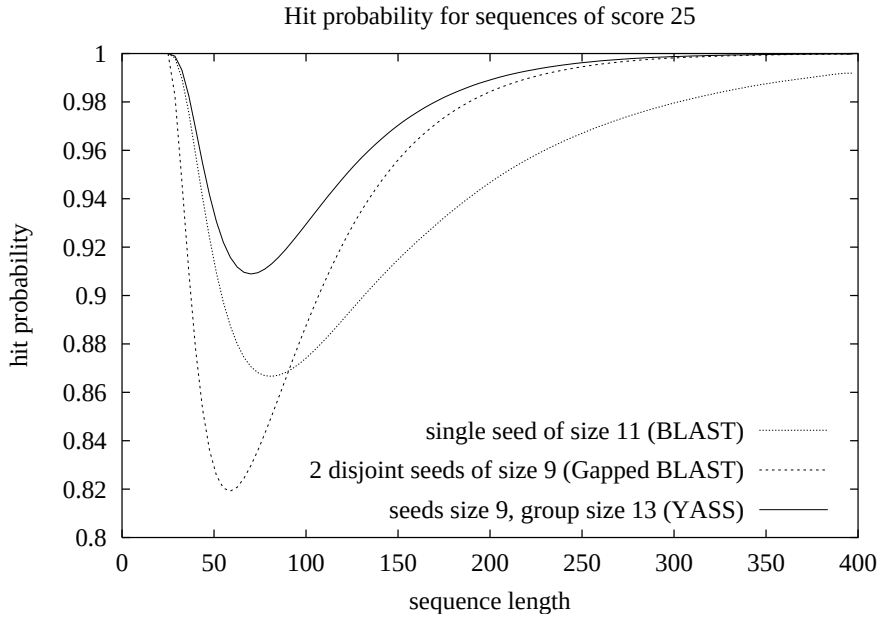
A.5 Acknowledgments

We are grateful to Mikhail Roytberg for enlightening discussions, and to Marie-Pierre Etienne, Roman Kolpakov, Gilles Schaeffer and Pierre Valois for their helpful comments at early stages of this work.

A.6 Figures

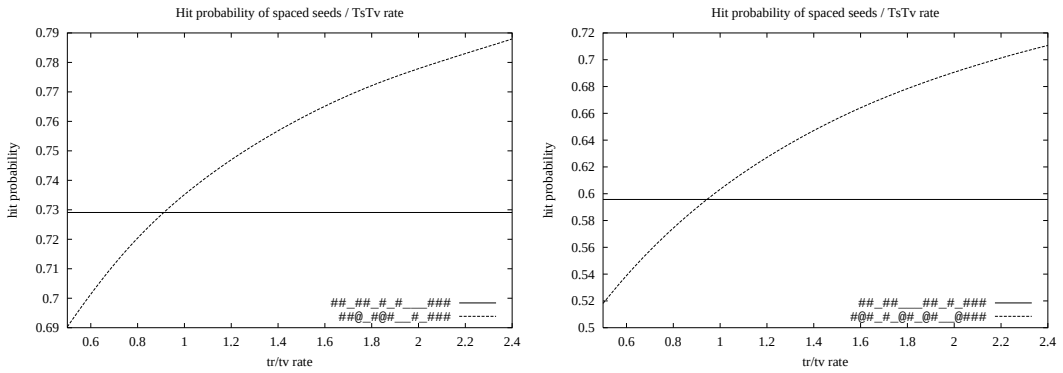
A.6.1 Figure 1 - Hit Probability

Hit probability as a function of length of fixed-score alignments



A.6.2 Figure 2 - Seed Probability

Hit probability of seed models on Bernoulli sequences as a function on ti/tv ratio



(a) (b)

A.7 Tables

A.7.1 Table 1 - Bernoulli Model

Hit probability of seeds on Bernoulli sequences of length 64 with match probability 0.7 and transition/transversion probabilities 0.15

weight	spaced seed	hit proba	transition-constrained seed	hit proba
9	B_9 = `##_##_#_#___###`	0.7291	B_9^{tr} = `###@_#@#__#_###`	0.7366
10	B_{10} = `##_##___##_#_###`	0.5957	B_{10}^{tr} = `#@#_#_@#_@#__@###`	0.6056
11	B_{11} = `###_#__#_#__##_###`	0.4671	B_{11}^{tr} = `###@#_#@__##_#_@@##`	0.4784

A.7.2 Table 2 - Markov Model

Hit probability of seeds on a Markov model of order 5 trained on a large mixed sample of cross-species alignments

weight	spaced seed	hit proba	transition-constrained seed	hit proba
9	M_9 = `##_##_##_###`	0.822	M_9^{tr} = `##@##_##@##`	0.845
10	M_{10} = `##_##_##___##_##`	0.716	M_{10}^{tr} = `#@@##_##_##@#@`	0.746
11	M_{11} = `##_##_##_##_###`	0.603	M_{11}^{tr} = `##@##_##_##@##`	0.632

A.7.3 Table 3 - Seed experiments

Number of high-scoring similarities found with different seed patterns

sequences	B_9	B_9^{tr}	B_{11}	B_{11}^{tr}	M_9	M_9^{tr}	M_{11}	M_{11}^{tr}
IX /V	323	336	275	279	312	325	274	293
IX /XVI	342	354	271	280	349	357	280	295
XVI /IV	1314	1361	1124	1172	1309	1348	1180	1235
MC58/Z2491	361896	380028	341113	364792	385444	392164	359348	366759

A.7.4 Table 4 - Comparative Tests

Comparative tests of YASS vs `bl2seq` (NCBI BLAST 2.2.6). Reported execution times have been obtained on a Pentium IV 2.4GHz computer.

sequence 1		sequence 2		time (sec)		# align.		# ex. align.		ex. align. length	
name	size	name	size	Y.	B.	Y.	B.	Y.	B.	Y.	B.
S.sp.	3.6	*M.t.*	4.4	122	148	494	310	130	27	29145	7970
S.sp.	3.6	*C.g.*	3.3	161	163	578	369	168	63	37310	30138
S.sp.	3.6	*Y.p.*	4.6	156	253	901	617	186	54	39354	19994
S.sp.	3.6	*V.p.*	3.3	164	167	940	465	349	60	65788	28883
M.t.	4.4	*C.g.*	3.3	211	542	1851	1265	397	160	102103	80012
M.t.	4.4	*Y.p.*	4.6	168	255	738	515	197	86	44348	23361
M.t.	4.4	*V.p.*	3.3	72	69	498	295	171	30	36474	12021
C.g.	3.3	*Y.p.*	4.6	130	161	962	640	186	45	34538	11277
C.g.	3.3	*V.p.*	3.3	95	93	1109	687	197	72	42009	21575
Y.p.	4.6	*V.p.*	3.3	149	217	2900	1953	622	264	186585	110352

C.g : *Corynebacterium glutamicum ATCC 13032* ,

M.t : *Mycobacterium tuberculosis (CDC1551)* ,

S.sp. : *Synechocystis sp. PCC 6803* ,

S.sp. : *Vibrio parahaemolyticus RIMD 2210633 chr I* ,

Y.p. : *Yersinia pestis KIM*

Glossaire

état récurrent : état d'un automate par lequel il est possible de transiter un nombre infini de fois lors de la lecture d'une entrée.
état transient : état d'un automate par lequel il n'est possible de transiter qu'une seule fois lors de la lecture d'une entrée.

ADN : Acide DésoxyriboNucléique : molécule portant l'information génétique, constituée de deux brins recourbés en forme de double hélice, liés entre eux pas des liaisons hydrogènes.
algorithme de Needleman-Wunsch : algorithme d'alignement global de deux séquences.
algorithme de Smith-Waterman : algorithme d'alignement local de deux séquences recherchant les alignements maximaux de score positif.
alignement : représentation d'une similarité par superposition des deux copies.
alignement global : comparaison de deux séquences sur toute leur étendue.
alignement local : comparaison de deux séquences seulement sur des portions de leur étendue.
ARN : Acide RiboNucléique : molécule similaire à l'ADN, mais constituée en général d'un seul brin et composée de nucléotides différentes (l'Uracile de l'ARN remplace la Thymine de l'ADN).
automate : machine à états finis qui, étant donné une entrée représentée par un mot, se déplace dans une série d'états donnés par une fonction de transition.
automate complet : automate pour lequel tout état possède au moins une transition pour chaque lettre de l'alphabet d'entrée.
automate de Aho-Corasick : automate qui accepte un ensemble de facteurs donnés dans une entrée.
automate minimal : automate reconnaissant un langage donné et dont le nombre d'états est minimal.
automate total : voir automate complet.

cellule : unité de structure de base de tous les organismes vivants contenant entre autres une copie complète du génome de l'organisme.
centromère : désigne la portion du chromosome qui normalement sépare celui-ci en deux bras.
chromosome : agrégat condensé composé de deux brins d'ADN.
code génétique : système de correspondance permettant aux acides nucléiques d'être traduit en protéine.
codon : facteur de trois nucléotides qui encode un acide aminé ou une terminaison de traduction.
contig : groupes de fragments de séquences d'ADN assemblés selon leurs recouvrements
crossing-over : Echange de matériel génétique entre chromosomes homologues durant la méiose

diagramme de transitions : représentation d'un automate sous forme de graphe.

distance de Hamming : distance entre deux chaînes de caractères qui n'autorise que les opérations de substitution de lettres.
distance de Levenstein : distance entre deux chaînes de caractères qui autorise que les opérations de substitution de lettres ainsi que les ajouts et suppressions de lettres.
dot-plot : représentation de l'ensemble des similarités entre deux séquences sous forme planaire.

eucaryote : organisme possédant une cellule avec un noyau portant le matériel génétique.
exon : fragment de séquence d'ADN qui est transcrite en ARN messager et traduite en protéine.

génome : ensemble complet de matériel génétique d'un être vivant.
graphe d'édition : graphe représentant les différentes opérations d'édition possibles entres deux chaînes de caractères.

hit : événement généré par un filtre lorsqu'une paire de position est susceptible de contenir une similarité.
homologie : ressemblance héritée d'un ancêtre commun.

in silico : signifie *réalisé sur ordinateur* ou *réalisé par simulations sur ordinateur*. Ce terme provient des phrases latines *in vivo* et *in vitro*, qui sont utilisées en biologie pour désigner des expériences faites sur des organismes vivants et sur des solutions chimiques
indel : *insertion/deletion event*, mutation produite par ajout ou suppression d'une ou plusieurs nucléotides
intron : fragment de séquence d'ADN qui est transcrite en ARN messager mais non traduite en protéine.

langage régulier : langage obtenu par union, concaténation et fermeture de Kleene d'ensembles

méiose : processus de division cellulaire qui forme les gamètes et réduit le nombre de chromosomes en ne selectionnant qu'un chromosome pour chaque paire.
matrice de scores : matrice associant un score à chacune des substitutions possibles entre deux lettres.
minimisation de Hopcroft : algorithme sous quadratique de minimisation d'automate.
minimisation de Moore : algorithme quadratique de minimisation d'automate.
mitose : Processus de division cellulaire qui crée deux cellules filles identiques à la celulle mère du point de vue de la séquence chromosomique.
mutation ponctuelle : mutation ne portant que sur sur une seule base, il s'agit en général de substitutions.
mutation silencieuse : mutation d'un exon qui n'affecte pas la protéine produite, à cause de la redondance du code génétique.

nucléotide : molécule composée d'un pentose (sucre à cinq carbone) lié à une base nitrogène (adénine, cytosine, guanine, thymine, or uracile) et d'un groupe phosphate.

pathogène : qui génère une maladie dans un autre organisme.
procaryote : organisme unicellulaire ne possédant pas de noyau, le matériel génétique est dispersé dans la cellule.
protéine : macro-molécule complexe composée d'une ou plusieurs chaînes d'acides aminés. Les protéines sont les constituants majeurs des cellules biologiques

répétition distante : répétition sur la même séquence dont les copies ne sont pas juxtaposées.

répétition en tandem : répétition sur la même séquence dont les copies sont juxtaposées les une aux autres.
relation d'équivalence : relation binaire réflexive, symétrique, et transitive sur une ensemble.

sélectivité : capacité à ne détecter que la réalité biologique et rien de plus.
score : système de mesure permettant d'estimer le degrés de confiance d'une similarité
sensibilité : capacité à détecter tout ce qui est intéressant sur le plan biologique.
spéciation : évolution d'une espèce en plusieurs autres, due par exemple à des isolements géographiques, ou une adaptation à un environnement particulier

télomère : extrémité des chromosomes eucaryotes.
traduction : synthèse d'une chaîne d'acides aminés (protéine) à partir d'un ARN messager.
transcription : synthèse d'une molécule d'ARN messager à partir d'une séquence d'ADN.
transition : mutation ponctuelle substituant une purine (A,G) à une purine, ou une pyrimidine (T,C) à une pyrimidine.
transposon : fragment de chromosome mobile.
transversion : mutation ponctuelle substituant une purine (A,G) à une pyrimidine (T,C), ou une pyrimidine à une purine.

Index

Bibliographie

[1] J Abello, M Resende, and S Sudarsky. Massive quasi-clique detection. In *Proceedings of the 5th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 598–612, London, UK, 2002. Springer-Verlag. 4.6.3

[2] A. V. Aho and M. J. Corasick. Efficient string matching : An aid to bibliographic search. *Communications of the ACM*, 18(6) :333–340, 1975. 2.2.4, 3.7.7.1, 4.5.1

[3] S. Aki, H. Kuboki, and K. Hirano. On discrete distributions of order k. *Annals of the Institute of Statistical Mathematics*, 36 :431–440, 1984. A.4.1

[4] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle : a new structure for pattern matching. In J. Pavelka, G. Tel, and M. Bartosek, editors, *SOFSEM'99, Theory and Practice of Informatics*, volume 1725 of *Lecture Notes in Computer Science*, pages 291–306. Springer-Verlag, 1999. 3.4.1.2

[5] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215 :403–410, 1990. (document), 3.4.4.2, 4.1.2, A.1, A.2.1

[6] S. Altschul, T. Madden, A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. Gapped BLAST and PSI-BLAST : a new generation of protein database search programs. *Nucleic Acids Research*, 25(17) :3389–3402, 1997. (document), 3.4.4.2, 4.1.1.4, A.1, A.2.1, A.2.1.1

[7] A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985. 3.4.1.1

[8] G. Bejerano, D. Haussler, and M. Blanchette. Into the heart of darkness : large-scale clustering of human non-coding DNA. *Bioinformatics*, 20(Suppl. 1) :40–48, 2004. (document)

[9] G. Benson. An algorithm for finding tandem repeats of unspecified pattern size. In *Proceedings of the 2nd Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 20–29, 1998. 3.5.2

[10] G. Benson. Tandem repeats finder : a program to analyse DNA sequences. *Nucleic Acids Research*, 27(2) :573–580, 1999. 3.5.2, A.4.1

[11] G. Benson. Composition alignment. In *Proceedings of the Third International Workshop on Algorithms in Bioinformatics (WABI), Budapest, Hungary, September 15-20, 2003, Budapest (Hungary)*, volume 2812 of *Lecture Notes in in Bioinformatics*, pages 447–461. Springer-Verlag, 2003. 2.1.1

[12] S. Bérard and E. Rivals. Comparison of minisatellites. *Journal of Computational Biology*, 10(3/4) :357–372, 2003. 2.1.1

[13] J. Berstel and D. Perrin. Algorithms on words. In J. Berstel and D. Perrin, editors, *Applied combinatorics on words*, Lothaire books. Cambridge University Press, 2005. 2.2, 2.2.3

[14] B. E. Blaisdell, C. Burge, and S. Karlin. An efficient algorithm for identifying matches with errors in multiple long molecular sequences. *Journal of Molecular Biology*, 221 :1367–1378, 1991. 3.4.3

[15] B. Brejova, D. Brown, and T. Vinar. Optimal spaced seeds for Hidden Markov Models, with application to homologous coding regions. In M. Crochemore R. Baeza-Yates, E. Chavez, editor, *Proceedings of the 14th Symposium on Combinatorial Pattern Matching (CPM), Morelia (Mexico)*, volume 2676 of *Lecture Notes in Computer Science*, pages 42–54. Springer-Verlag, June 2003. 3.7.5, 4.5.2, 4.6.2.1, A.2.2, A.2.2.2

[16] B. Brejova, D. Brown, and T. Vinar. Vector seeds : an extension to spaced seeds allows substantial improvements in sensitivity and specificity. In G. Benson and R. Page, editors, *Proceedings of the 3rd International Workshop in Algorithms in Bioinformatics (WABI), Budapest (Hungary)*, volume 2812 of *Lecture Notes in Computer Science*, pages 39–54. Springer-Verlag, September 2003. 3.8, 4.4, A.1, A.2.2, A.3

[17] B. Brejova, D. Brown, and T. Vinar. Optimal spaced seeds for homologous coding regions. *Journal of Bioinformatics and Computational Biology*, 1(4) :595–610, Jan 2004. 4.2.1.2

[18] D. Brown. Multiple vector seeds for protein alignment. In I. Jonassen and J. Kim, editors, *Proceedings of the 4th International Workshop in Algorithms in Bioinformatics (WABI), Bergen (Norway)*, volume 3240 of *Lecture Notes in Computer Science*, pages 170–181. Springer-Verlag, September 2004. 3.8, 5.3.4

[19] D. Brown. Optimizing multiple seeds for protein homology search. *IEEE Transactions on Computational Biology and Bioinformatics (IEEE TCBB)*, 2(1) :29–38, january 2005. 3.8

[20] M. Brudno, Chuong Do, G. Cooper, M. Kim, E. Davydov, E. Green, A. Sidow, and S. Batzoglou. LAGAN and Multi-LAGAN : Efficient tools for large-scale multiple alignment of genomic DNA. *Genome Research*, 13 :1–11, 2003. A.1

[21] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5) :419–428, 2001. 3.6, 3.7, A.2.2

[22] J. Buhler. Provably sensitive indexing strategies for biosequence similarity search. In *RECOMB, Washington, DC (USA)*, pages 90–99. ACM Press, April 2002. (document)

[23] J. Buhler, U. Keich, and Y. Sun. Designing seeds for similarity search in genomic DNA. In *Proceedings of the 7th Annual International Conference on Computational Molecular Biology (RECOMB), Berlin (Germany)*, pages 67–75. ACM Press, April 2003. 3.7, 3.7.7, 3.7.7.1, 3.7.7.2, 3.7.7.2, 3.8, 4.2.1.2, 4.2.2, 4.5.1

[24] J. Buhler and M. Tompa. Finding motifs using random projections. In *Proceedings of the 5th Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 69–76. ACM Press, 2001. 3.7

[25] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. In *Proceedings of the 12th Symposium on Combinatorial Pattern Matching (CPM)*, volume 2089 of *Lecture Notes in Computer Science*, pages 73–85. Springer-Verlag, July 2001. 3.7.1, 34

[26] S. Burkhardt and J. Karkkainen. One-gapped q-gram filters for Levenshtein Distance. In *Proceedings of the 13th Symposium on Combinatorial Pattern Matching (CPM)*, volume 2373 of *Lecture Notes in Computer Science*, pages 225–234. Springer-Verlag, 2002. 32

[27] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. *Fundamenta Informaticae*, 56(1-2) :51–70, 2003. Preliminary version in Combinatorial Pattern Matching 2001. 5.2.1, 31, 31, A.2.2

[28] 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 (ISMB)*, pages 56–64, July 1993. (document), 3.6, A.2.2

[29] A. Carbone and M. Gromov. Mathematical slices of molecular biology. *Société Mathématique de France*, Supplement Number 88 :11–80, 2000.

[30] F. Chiaromonte, V.B. Yap, and W. Miller. Scoring pairwise genomic sequence alignments. In *Pacific Symposium on Biocomputing*, volume 7, pages 115–126, 2002. A.2.2, A.2.3.1

[31] K. P. Choi, F. Zeng, and L. Zhang. Good spaced seeds for homology search. *Bioinformatics*, 20(7) :1053–1059, 2004. 4.2.1.2, A.2.2, A.2.2.1

[32] K.P. Choi and L. Zhang. Sensitivity analysis and efficient method for identifying optimal spaced seeds. *Journal of Computer and System Sciences*, 68(1) :22–40, 2004. 3.7.6, 4.2.1.2, A.2.2, A.2.2.1

[33] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithmique du texte*. Vuilbert, 2001. 2.1.2, 2.1.4.1

[34] M. Crochemore, G. Landau, and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In David Eppstein, editor, *Proceedings of the Thirteen Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 679–688. ACM-SIAM, 2002. (document)

[35] M. Crochemore and T. Lecroq. *Handbook of Computer Science and Engeneering*. CRC Press, 1997. 2.2

[36] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2002. 2.2.4

[37] M. Csürös. Performing local similarity searches with variable lenght seeds. In S.C. Sahinalp, S. Muthukrishnan, and U. Dogrusoz, editors, *Proceedings of the 15th Annual Combinatorial Pattern Matching Symposium (CPM), Istanbul (Turkey)*, volume 3109 of *Lecture Notes in Computer Science*, pages 373–387. Springer-Verlag, 2004. 3.8

[38] M. Dayhoff, R.M. Schwartz, and B.C. Orcutt. Atlas of protein sequence and structure. In *A model of evolutionary change in proteins.*, volume 5, pages 345–352. National Biomedical Research Foundation, Silver Spring, Maryland, 1978. 16

[39] A. Delcher, S. Kasif, R. Fleischmann, J. Peterson, O. White, and S. Salzberg. Alignement of whole genomes. *Nucleic Acids Reseach*, 27(11) :2369–2376, 1999. 3.4.1.1

[40] A. Delcher, A. Phillippy, J. Carlton, and S. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Reseach*, 30(11) :2478–2483, 2002. 3.4.1.1

[41] V. Deleplace. Recherche de similitudes de protéines à l'aide de graines sous-ensemble. Rapport de DEA, ESIAL/UHP/LORIA, June 2005. 5.5.2.2

[42] A. Denise. Génération aléatoire et uniforme de mots de langages rationnels (Uniform random generation of words of regular languages). *Theoretical Computer Science*, 159(1) :43–63, 1996. 4.1.1.3

[43] M. Farach-Colton, G. M. Landau, S. Cenk Sahinalp, and D. Tsur. Optimal spaced seeds for faster approximate string matching. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP'05), Lisboa (Portugal)*, volume 3580 of *Lecture Notes in Computer Science*, pages 1251–1262. Springer-Verlag, 2005. 5.3.5

[44] A.V. Finkelstein and M.A. Roytberg. Computation of biopolymers : A general approach to different problems. *BioSystems*, 30(1-3) :1–19, 1993. 4.3.2

[45] M. R. Garey and D. S. Johnson. *Computer and Intractability : A Guide to the Theory of NP-Completeness.* W H Freeman, november 1979. 5.3.5

[46] E. Glémet and J. Codani. LASSAP, a LArge Scale Sequence compArison Package. *CABIOS*, 13(2) :137–143, 1997. (document)

[47] E Gumbel. *Statistics of extremes.* Columbia University Press, New York, 1958. 3.4.4.2

[48] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997. 3.4.1.1

[49] S. Henikoff and J.G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. USA*, 89 :10915–10919, 1992. 16

[50] T. Hickey and J. Cohen. Uniform random generation of strings in a context-free language. *SIAM. J. Comput*, 12(4) :645–655, 1983. 4.1.1.3

[51] J. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computation*, pages 189–196. Academic Press, New York, 1971. 2.2.3.2

[52] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison Wesley, 1979. 2.2

[53] P. Indyk and R. Motwani. Approximate nearest neighbors : towards removing the curse of dimensionality. In *Proc. 30th Symp. Theory of Computing*, pages 604–613. ACM, May 1998. 3.6

[54] L. Kaderali and A. Schliep. Selecting signature oligonucleotides to identify organisms using DNA arrays. *Bioinformatics*, 18(10) :1340–1349, 2002. 5.5.2.1

[55] S. Karlin and S. Altschul. Methods for assessing the statistical significance of molecular sequence feature by using general scoring schemes. In *Proc. Natl. Acad. Sci.*, volume 87, pages 2264–2268, 1990. 3.4.4.2

[56] S. Karlin and V. Brendel. Chance and significance in protein and DNA sequence analysis. *Science*, 257 :39–49, 1992. 3.4.4.2

[57] S. Karlin and F. Ost. Counts of long aligned word matches among random letter sequences. *Advances in Applied Probability*, 19 :293–351, 1987. 3.4.4.2

[58] S. Karlin and F. Ost. Maximal length of common words among random letter sequences. In *Ann. Prob.*, volume 16, pages 535–563, 1988. 3.4.4.2

[59] U. Keich, M. Li, B. Ma, and J. Tromp. On spaced seeds for similarity search. *Discrete Applied Mathematics*, 138(3) :253–263, 2004. preliminary version in 2002. (document), 3.7.1, 3.7.4, 3.8, 4.1.1.4, 31, A.2.2, A.2.2.1, A.2.2.2

[60] W. J. Kent. BLAT–the BLAST-like alignment tool. *Genome Research*, 12 :656–664, 2002. 3.4.4.3, 4.1.1.4, A.1, A.2.1, A.2.1.1, A.2.1.2, A.2.2

[61] R. Kolpakov, G. Bana, and G. Kucherov. mreps : efficient and flexible detection of tandem repeats in DNA sequences. *Nucleic Acid Research*, 31 :3672–3678, 2003. 3.5.1

[62] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Symposium on Foundations of Computer Science (FOCS)*, pages 596–604, New-York (USA), 1999. IEEE Computer Society. 3.5.1

[63] G. Kucherov, L. Noé, and Y. Ponty. Estimating seed sensitivity on homogeneous alignments. In *Proceedings of the IEEE 4th Symposium on Bioinformatics and Bioengineering (BIBE), May 19-21, 2004, Taichung (Taiwan)*, pages 387–394. IEEE Computer Society Press, April 2004. (document), A.2.1.2, A.2.2

[64] G. Kucherov, L. Noé, and M. Roytberg. Multi-seed lossless filtration. In S.C. Sahinalp, S. Muthukrishnan, and U. Dogrusoz, editors, *Proceedings of the 15th Annual Combinatorial Pattern Matching Symposium (CPM), July 5-7, 2004, Istanbul (Turkey)*, volume 3109 of *Lecture Notes in Computer Science*, pages 297–310. Springer-Verlag, 2004. (document), A.3

[65] G. Kucherov, L. Noé, and M. Roytberg. Multiseed lossless filtration. *IEEE Transactions on Computational Biology and Bioinformatics (TCBB)*, 2(1) :51–61, january 2005. (document)

[66] G. Kucherov, L. Noé, and M. Roytberg. A unifying framework for seed sensitivity and its application to subset seeds. In R Casadio and G Myers, editors, *Proceedings of the 5th International Workshop in Algorithms in Bioinformatics (WABI), Mallorca (Spain)*, volume 3692 of *Lecture Notes in Computer Science*, pages 251–263. Springer Verlag, 2005. (document)

[67] S. Kurtz. Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29(13) :1149–1171, 1999. 3.4.1.1

[68] S. Kurtz, E. Ohlebusch, C. Schleiermacher, and J. Stoye. Reputer : the manifold applications of repeat analysis. *Nucleic Acids Research*, 29(22) :4633–4642, 2001. 3.4.1.1

[69] S. Kurtz and C. Schleiermacher. REPuter : Fast Computation of Maximal Repeats in complete genome. *Bioinformatics*, 15(5) :426–427, 1999. 3.4.1.1

[70] A. Lefebvre, T. Lecroq, H. Dauchel, and J. Alexandre. FORRepeats : detects repeats on entire chromosomes and between genomes. *Bioinformatics*, 19(3) :319–326, 2003. 3.4.1.2

[71] F. Li and G.D. Stormo. Selection of optimal DNA oligos for gene expression arrays. *Bioinformatics*, 17 :1067–1076, 2001. 5.5.2.1

[72] M. Li, B. Ma, D. Kisman, and J. Tromp. PatternHunter II : Highly sensitive and fast homology search. *Journal of Bioinformatics and Computational Biology*, 2(3) :417–439, 2004. (early version in GIW 2003). 3.7.1, 3.7.3, 3.8, A.3

[73] D. Lipman and W. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227 :1435–1441, 1985. (document), 3.4.4.1

[74] D. Lipman and W. Pearson. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. USA*, 85 :2444–2448, 1988. (document), 3.4.4.1, A.1

[75] B. Ma, J. Tromp, and M. Li. PatternHunter : Faster and more sensitive homology search. *Bioinformatics*, 18(3) :440–445, 2002. (document), 3.7.1, 3.7.3, 4.1.1.4, 4.1.2, 4.2.1.2, A.1, A.2.1, A.2.2, A.2.2.1

[76] U. Manber and E. Myers. Suffix arrays : A new method for on-line string searches. *SIAM Journal on Computing*, 22(935–948), 1993. 5.5.2.1

[77] E. Margulies, M. Blanchette, D. Haussler, and E. Green. Identification and characterization of multi-species conserved sequences. *Genome Research*, 13 :2507–2518, 2003. (document)

[78] J. Martin. *Introduction to Languages and the Theory of Computation.* McGraw-Hill, Inc., 1990. 1

[79] E. McCreight. A space economical suffix tree construction algorithm. *Journal of the Association of Computing Machinery*, 2 :262–272, 1976. 3.4.1.1

[80] E.F. Moore. Gedanken-experiments on sequential machines. Technical report, Automata studies, Princeton university, Princeton, 1956. 2.2.3.1

[81] B. Morgenstern, A. Dress, and T. Werner. Multiple DNA and protein sequence alignment based on segment-to-segment comparison. In *Applied Mathematics*, volume 93, pages 12098–12103, 1996. (document)

[82] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages. 5.1.1

[83] 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. (document), 3.4.4.2

[84] P. Nicodème, B. Salvy, and P. Flajolet. Motif statistics. *Theoretical Computer Science*, 287(2) :593–617, 2002. 3.7.7

[85] F. Nicolas and E. Rivals. Hardness of optimal spaced seed design. In A. Apostolico, M. Crochemore, and K. Park, editors, *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM), Jeju Island (Korea)*, volume 3537 of *Lecture Notes in Computer Science*, pages 144–155. Springer-Verlag, 2005. 3.8, 5.3.5

[86] L. Noé and G. Kucherov. YASS : Similarity search in DNA sequences. Research Report RR-4852, INRIA, Jun 2003. A.1, A.2.3.1

[87] L. Noé and G. Kucherov. Improved hit criteria for DNA local alignment. In *Proceedings of the 5th Open Days in Biology, Computer Science and Mathematics (JOBIM), June 28-30, 2004, Montréal (Canada)*, June 2004. (document)

[88] L. Noé and G. Kucherov. Improved hit criteria for DNA local alignment. *BMC Bioinformatics*, 5(149), october 2004. (document), 2.1.6, 4.1.1.4, 4.6.1.2, A

[89] L. Noé and G. Kucherov. YASS : enhancing the sensitivity of DNA similarity search. *Nucleic Acids Research*, 33 (web-server issue) :W540–W543, Jul 2005. (document)

[90] J. Oommen and J. Dong. Generalized swap-with-parent schemes for self-organizing sequential linear lists. In *Proceedings of the 1997 International Symposium on Algorithms and Computation (ISAAC'97), Singapore*, volume 1350 of *Lecture Notes in Computer Science*, pages 414–423. Springer-Verlag, December 17-19 1997. 5.3.4

[91] P. Pevzner and M. Waterman. Multiple filtration and approximate pattern matching. *Algorithmica*, 13 :135–154, 1995. Preliminary version in Combinatorial Pattern Matching 1993. 3.7.1, 5.1.2, A.2.2

[92] S. Rahmann. Fast and sensitive probe selection for DNA chips using jumps in matching statistics. In *IEEE Computer Society Bioinformatics Conference (CSB)*, pages 57–64, 2003. 5.5.2.1

[93] S. Rahmann. Fast large scale oligonucleotide selection using the longest common factor approach. *Journal of Bioinformatics and Computational Biology*, 1(2) :343–361, 2003. 5.5.2.1

[94] J. SantaLucia. A unified view of polymer, dumbbell, and oligonucleotide DNA nearest-neighbor thermodynamics. *Biochemistry*, 95 :1460–1465, February 1998. 5.5.2

[95] J. SantaLucia, H. Allawi, and A Seneviratne. Improved nearest-neighbor parameters for predicting DNA duplex stability. *Biochemistry*, 35 :3555–3562, 1996. 5.5.2

[96] S. Schwartz, J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. Hardison, D. Haussler, and W. Miller. Human–mouse alignments with BLASTZ. *Genome Research*, 13 :103–107, 2003. 3.8, A.1, A.2.2

[97] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(195–197), March 1981. (document), 3.2, A.1

[98] D. J. States and P. Agarwal. Compact encoding strategies for DNA sequence similarity search. In AAAI press, editor, *In Proceedings, Fourth International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 211–217, 1996. (document)

[99] Y. Sun and J. Buhler. Designing multiple simultaneous seeds for DNA similarity search. In *Proceedings of the 8th Annual International Conference on Computational Molecular Biology (RECOMB), San Diego (California)*, March 2004. 3.8, 4.2.1.2, A.3

[100] F. Touzain, S. Schbath, I. Debled-Rennesson, B. Aigle, P. Leblond, and G. Kucherov. SIGffRid : Programme de recherche des sites de fixation des facteurs de transcription par approche comparative. In *JOBIM*, 2005. 1.5

[101] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14 :249–260, 1995. 3.4.1.1

[102] J. Ullman, A. Aho, and J. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, 1974. 4.2.1.2, 4.3.2

[103] P. Vincens, L. Buffat, C. André, J. Chevrolat, J. Boisvieux, and S. Hazout. A strategy for finding regions of similarity in complete genome sequences. *Bioinformatics*, 14(8) :715–725, 1998. (document), 3.4.4.3

[104] A. Vitreschak, L. Noé, and G. Kucherov. Computer analysis of multiple repeats in bacteria. In *Proceedings of the 4th International conference on Bioinformatics of Genome Regulation and Structure (BGRS), July 25-30, 2004, Novosibirsk (Russia)*, volume 2, pages 297–299. Institute of Cytology and Genetics, July 2004. 4.6.3, 5.5.2.2

[105] N. Volfovsky, B. J Hass, and S. Salzberg. A clustering method for repeat analysis in DNA sequences. *Genome Biology*, 2(8), 2001. 4.6.3

[106] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 4th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973. 3.4.1.1

[107] H. Wilf. A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects. *Advances in Mathematics*, 24 :281–291, 1977. 4.1.1.3

[108] J. Xu, D. Brown, Ming Li, and Bin Ma. Optimizing multiple spaced seeds for homology search. In S.C. Sahinalp, S. Muthukrishnan, and U. Dogrusoz, editors, *Proceedings of the 15th Symposium on Combinatorial Pattern Matching (CPM), Istambul (Turkey)*, volume 3109 of *Lecture Notes in Computer Science*, pages 47–58. Springer-Verlag, 2004. 5.3.4, 5.5.2.2

[109] I. Yang, S. Wang, Y. Chen, P. Huang, L. Ye, X. Huang, and K. Chao. Efficient methods for generating optimal single and multiple spaced seeds. In *Proceedings of the IEEE 4th Symposium on Bioinformatics and Bioengineering (BIBE), Taichung (Taiwan)*, pages 411–416. IEEE Computer Society Press, 2004. 5.5.2.2

[110] Z. Zhang, S. Schwartz, . Wagner, and W. Miller. A greedy algorithm for aligning DNA sequences. *Journal of Computational Biology*, 7(1/2) :203–214, 2000. 3.4.4.3

[111] J. Zheng, T. Close, T. Jiang, and S. Lonardi. Efficient selection of unique and popular oligos for large EST databases. In *Proceedings of Symposium on Combinatorial Pattern Matching (CPM), Morelia, Mexico*, volume 2676 of *Lecture Notes in Computer Science*, pages 273–283. Springer-Verlag, June 2003. 5.5.2.1, 5.5.2.1

[112] M. Zuker, D. Mathews, and D. Turner. Algorithms and thermodynamics for RNA secondary structure prediction : A practical guide. In J. Barciszewski & B.F.C. Clark, editor, *RNA Biochemistry and Biotechnology*, NATO ASI Series. Kluwer Academic Publishers, 1999. 5.5.2

Résumé

Les méthodes de recherche de similarités les plus fréquemment utilisées dans le cadre de la génomique sont heuristiques. Elles se basent sur un principe de filtrage du texte qui permet de localiser les régions potentiellement similaires. Dans cette thèse, nous proposons de nouvelles définitions de filtres pour la recherche de similarités sur les séquences génomiques et des algorithmes associés pour mesurer leurs caractéristiques.

Plus précisément, nous avons étudié le modèle des *graines espacées*, et proposé un algorithme d'évaluation de l'efficacité des graines sur des similarités d'une classe particulière (similarités dites *homogènes*). Nous avons également développé un algorithme général pour la mesure de l'efficacité des graines, ainsi qu'un nouveau modèle de graine appelé *graine sous-ensemble*, extension du modèle des *graines espacées*. Enfin nous donnons, dans le cadre du *filtrage sans perte*, une extension à l'aide de graines multiples, que nous analysons et appliquons au problème de la conception d'oligonucléotides.

Nous avons réalisé et donnons accès à des outils pour la conception des filtres, ainsi que pour la recherche de similarités.

Mots-clés: bio-informatique, alignement local de séquences génomiques, graines espacées, graines sous-ensemble

Abstract

Most commonly used similarity search methods in genomic sequences are heuristic ones. These are based upon text filtering that allows one to infer potential regions of similarity. This thesis proposes new filter definitions to search for similarities in genomic sequences, and fast algorithms to measure the efficiency of these filters.

More precisely, we study the *spaced seed* model and propose an algorithm to measure the seed efficiency on similarities of a certain kind, called *homogeneous similarities*. A generic algorithm has also been developed to measure the seed efficiency, together with an extension of the spaced seed model called *subset seed*. Finally, we propose and analyze a multi-seed approach in the framework of lossless filtration, and apply it to the problem of oligonucleotide design.

Several software tools have been developed to search for similarities as well as to design seed-based filters.

Keywords: bioinformatics, genomic local alignment, spaced seeds, subset seeds