Université de Lille 1 SCIENCES ET TECHNOLOGIES

CRIStAL
Centre de Recherche en Informatique, Signal et Automatique de Lille

# RAPPORT DE STAGE

## Recombinaison VDJ et amélioration de VIDJIL

---

Cyprien Borée

Avril 2018 à Juin 2018

**Entreprise** : Fondation INRIA, Domaine du Voluceau – Rocquencourt, 78153, Le Chesnay Cedex
**Université** : Université de Lille 1, Cité Scientifique, 59650, Villeneuve d'Ascq
**Formation initiale :** Licence 3 Informatique, parcours Info
**Tuteur à l'université :** Pierre Allegraud
**Tuteurs dans l'entreprise** : Mikaël Salson et Mathieu Giraud

## Remerciements

Je souhaite remercier **Hélène Touzet**, directrice de l'équipe de recherche *Bonsai* pour m'avoir permis de faire ce stage au sein de l'équipe dont elle est la responsable.

En second lieu je tiens à remercier l'Université de Lille 1, ainsi que son personnel m'ayant grandement aidé lors de l'élaboration de la convention de stage.

Mes remerciements vont plus particulièrement à **Mathieu Giraud** et **Mikaël Salson** pour m'avoir supervisé pendant ce stage ainsi que durant mon label Recherche portant sur le même sujet. Leur pédagogie et leur sagacité m'aura aidé à développer une rigueur scientifique dans mon travail et d'y apposer un point de vue critique.

Aussi je souhaite remercier **Pierre Allegraud** pour avoir été mon professeur tuteur et m'avoir fait profiter de son savoir.

Enfin, plus généralement je tiens à remercier mes collègues de bureau, les membres de l'équipe *Bonsai* et des autres équipes de recherche que j'ai côtoyés au long de ces trois mois.

Merci à tous

# Résumé


## Français

Ce stage prend place au sein de l'équipe de recherche en bio-informatique *Bonsai*, membre du laboratoire *CRIStAL* et partenaire de l'Université de Lille 1 et du *CNRS*. Plus précisément, ce stage s'articule autour du projet *VIDJIL*, un logiciel d'analyse de séquences ADN recombinées.

Néanmoins pour analyser une séquence, l'algorithme principal du logiciel la compare à toutes les autres séquences enregistrées afin de déduire celle qui a le plus de chance de lui ressembler. Mais ce temps de comparaison est fonction du nombre de séquences déjà enregistrées. Comme ce dernier tend à grandir, il est critique d'implémenter une solution optimisant cette partie.

Puisqu'il est difficile d'agir sur les comparaisons (déjà optimisées par des méthodes de programmation dynamique), la solution envisagée consistait à réduire ce nombre de comparaisons. Dans ce but, j'ai utilisé l'automate d'*Aho-Corasick*, déjà implémenté dans le logiciel mais encore inexploité.

Enfin après avoir mis en place cet algorithme j'ai pu observer les résultats et les comparer à ceux initiaux et il s'est avéré que en moyenne cet automate pouvait réduire le temps de comparaisons de 30 %.

## English

This internship takes place in the bioinformatics research team *Bonsai*, member of *CRIStAL* laboratory, and in a partnership with the *Université de Lille 1* and the *CNRS*. More precisely, this internship revolves around the *VIDJIL* software, a DNA sequences analysis software.

However to analyze a sequence, the main algorithm of the software needs to compare it to every other reference genes in order to get the most resembling one. But this comparison can take time depending on the number of reference genes. Since this number tends to grow, it was critical to implement a solution to optimize this part.

Due to the difficulty to operate on the comparisons themselves (already optimized by dynamic programming), the considered solution was to reduce the number of comparisons. In order to reduce it, I used the *Aho-Corasick* automaton already implemented in the software but still unused.

Finally, after set up this algorithm, I could observe the results and compare them to the original ones. It was found that the comparisons time could be reduced by 30 % in average.

# Sommaire

# Introduction

Ayant déjà effectué un stage l'année précédente dans une université étrangère dans les domaines de la reconnaissance d'image et des réseaux de neurones, j'étais intrigué par la manière dont était traité les technologies avancées en France. J'ai donc réalisé un *label Recherche* (option de la licence informatique) au sein de l'équipe *Bonsai* et j'ai poursuivi sur le même sujet avec ce stage.

Ce dernier s'articule autour du logiciel *VIDJIL*, un logiciel d'analyse de séquences ADN. Plus précisément, ce logiciel permet d'identifier une séquence ADN en entrée et de déduire à quelle catégorie de lymphocytes cette séquence appartient (grâce à la recombinaison de séquences nommée *VDJ*, propre aux lymphocytes et détaillée dans la partie 1.2.1).

Cet outil constitue un support majeur pour les équipes médicales afin de suivre la population de lymphocytes chez un patient. Néanmoins, ce logiciel a une complexité quadratique puisque la séquence en entrée est comparée à toutes les autres séquences connues et que pour chacune de ces comparaisons, les caractères sont évalués un à un. Par conséquence, il a été décidé que le logiciel soit écrit en C++, permettant une certaine rapidité.

Mais, le choix de ce langage ne suffit pas à palier le temps que met l'algorithme de comparaisons, et comme ce dernier est difficilement améliorable (déjà optimisé par des méthodes de programmation dynamique), la solution restante consistait à réduire le nombre de comparaisons.

Pour réduire ce nombre, j'ai utilisé l'automate d'*Aho-Corasick*. Cet algorithme était déjà implémenté dans le logiciel mais utilisé à d'autres fins. C'est pourquoi il a fallu créer un cas d'utilisation spéciale dans lequel cet automate est utilisé exclusivement pour comparer les séquences.

Cet automate se construit sur l'ensemble des séquences connues et prend en entrée la séquence dont on souhaite connaître à quel lymphocyte elle appartient. En sortie de l'automate on récupère l'ensemble des séquences ressemblant de près ou de loin à celle en entrée.

Ma partie du projet consistait donc à construire l'automate et à en récupérer les informations pour obtenir les séquences propices à la comparaison. Subsidiairement, il était également question de fixer un seuil (pour récupérer les « N » meilleurs séquences, ou bien les séquences apparaissant au moins « N » fois) pour limiter le nombre de comparaisons.

Le développement de cette solution a été majoritairement pilotée par des tests unitaires et fonctionnels, tests vérifiant que le résultat fourni par *VIDJIL* n'est pas modifié et que seul le temps d'exécution change.

# 1 Contexte

## 1.1 Contexte du stage

### 1.1.1 Le laboratoire *CRIStAL*

*CRIStAL* (Centre de Recherche en Informatique, Signal et Automatique de Lille), est un institut de recherche né le 1er janvier 2015 sous la tutelle du *CNRS*, de l'Université Lille 1 et de Centrale Lille en partenariat avec d'autres organismes.[1]

Aujourd'hui, le laboratoire *CRIStAL* est composé de près de 430 membres (228 permanents et plus de 200 non permanents) dont 24 permanents *CNRS* et 31 permanents *INRIA*. L'institut reprend les thématiques du *LIFL* et du *LAGIS* pour les adapter aux enjeux scientifiques et sociétaux du moment tels que : *Big-data*, logiciel, image et ses usages, interactions homme-machine, robotique, commande et supervision de grands systèmes, systèmes embarqués intelligents, bio-informatique…[1]

C'est au sein de ce laboratoire qu'existe l'équipe de recherche en bio-informatique *Bonsai*, appartenant anciennement à *LIFL*.[2][3]

### 1.1.2 Bonsai

*Bonsai* est une équipe de recherche en bio-informatique affiliée à *l'INRIA* et sous la tutelle du laboratoire *CRIStAL*. C'est au sein de cette équipe que s'est déroulé le travail couvert par ce rapport. L'équipe est composée d'une quinzaine de membres dont des enseignants-chercheurs, des chercheurs et des doctorants.[4]

L'objectif principal de l'équipe *Bonsai* est de développer des algorithmes et des modèles pour l'analyse des séquences biologiques. Cela comprend les domaines d'application suivants : l'annotation des génomes, l'analyse des données de séquençage à haut débit, la métagénomique, la structure des gènes et des génomes, les ARN non-codants, les peptides non ribosomiques. Les méthodes employées viennent de l'algorithmique discrète, empruntant à l'algorithmique du texte, la théorie des graphes, les structures d'index… Ce travail se matérialise par la diffusion de logiciels libres, et la validation sur des données biologiques.[2]

Parmi les logiciels libres on peut citer *YASS* (outil de recherche de similarité de génomes), *NORINE* (une plateforme contenant une base de données de peptides non ribosomiques), *GATB* (boîte à outils d'analyse de génomes) ou encore *VIDJIL*, logiciel sur lequel s'est déroulé le stage.[5]

### 1.1.3 *VIDJIL*

*VIDJIL* est un logiciel d'analyse de séquences ADN des lymphocytes produites par séquençage à haut-débit, en identifiant dans les séquences les clones avec des recombinaisons *VDJ*[1]. Le logiciel décrit quantitativement et qualitativement le répertoire lymphocytaire, ce qui permet d'améliorer le diagnostic et le suivi de certaines leucémies, et plus généralement, d'aider à des avancées en hématologie et en immunologie.

Ce logiciel est développé depuis 2011 et développé depuis lors par des membres de l'équipe *Bonsai*. Il est majoritairement écrit en C++11, permettant vitesse de calcul et optimisation mémoire.

VIDJIL est soutenu et financé par la fondation *INRIA*.

### 1.1.4 La fondation *INRIA*

*INRIA* (Institut National de Recherche Informatique et Automatique) est un établissement public crée en 1967 dans le cadre du « plan Calcul »[2]. Il s'agit du seul institut public de recherche consacré entièrement aux sciences du numérique.

Le 6 février 2017 est lancé la *fondation INRIA*, une fondation partenariale[7] dont le but est de créer et de financer des consortiums de logiciels libres et open-sources. Depuis le 1er janvier 2018, le consortium *VIDJILNET* est incubé au sein de l'action *INRIASOFT* de la fondation INRIA Ce consortium a pour but de :

- Pérenniser et soutenir le développement open-source de la plateforme web *VIDJIL*.
- Fournir du support et des services bio-informatiques aux adhérents.[8]

### 1.1.5 Le stage

La mission de stage a été commencé lors de mon label recherche et complété au long de ces trois mois.

L'objectif principal consistait à développer une solution pour réduire le nombre de comparaisons qu'effectue *VIDJIL*. La solution proposée consistait à utiliser l'algorithme (ou automate) d'*Aho-Corasick*.

Durant mon label Recherche, un travail de documentation et de veille technologique a été nécessaire pour me familiariser avec des thématiques tels que la programmation dynamique, l'alignement de séquence ou encore l'automate d'Aho-Corasick[3]. En plus de cette veille technologique j'ai pu apprivoiser VIDJIL et comprendre sa structure de développement[4].

1 La recombinaison VDJ est expliquée partie 1.2.1.
2 Plan gouvernemental lancé en 1966 par Charles de Gaulle destiné à assurer l'autonomie du pays dans les technologies de l'information.[6]
3 Ces thématiques sont abordées respectivement dans les parties 1.2.3 à 1.2.6 et 2.1.1.
4 La structure de développement est expliquée partie 1.2.7 et 1.2.8.

## 1.2 Contexte scientifique

Comme il m'a été nécessaire de réaliser un travail de documentation avant le stage, il me paraît évident d'introduire les concepts nécessaires à la bonne compréhension du sujet. Aussi je tiens à aborder la méthode de développement que suit *VIDJIL* qui aura été le fer de lance de ces trois mois.

### 1.2.1 Recombinaison VDJ

La recombinaison VDJ est une recombinaison de séquences ADN. Elle apparaît chez l'être humain et tous les autres vertébrés. Mais parmi l'entièreté des cellules du corps, elle n'est présente que chez les lymphocytes.[5]

Les lymphocytes sont des cellules participant à la défense immunitaire. Leur rôle est de reconnaître un antigène et de produire un anticorps qui le neutralisera. Néanmoins, un lymphocyte ne peut produire d'anticorps que pour un antigène particulier. Ainsi pour reconnaître X antigènes différents il faudra autant d'anticorps différents (voir Annexe A.1, Illustration n°1).

Cette diversité chez le lymphocyte est obtenue par la recombinaison de son génome. Elle s'opère dans une zone de l'ADN qui est appelée *VDJ*. C'est cette zone qui sera en contact direct avec les antigènes via les récepteurs du lymphocyte.

La zone *VDJ* est elle-même séparée en trois sous-zones : V, D et J, même si la zone D n'est pas toujours présente.[6]Lorsque cette dernière est absente on parle de chaîne *légère*, dans le cas contraire il s'agît d'une chaîne *lourde* (Voir Annexe A.1, Illustration n°2).

### 1.2.2 Alignement de séquences

En bio-informatique on parle d'alignement de séquences lorsqu'on superpose deux séquences biologiques (le plus souvent des séquences ADN), dans le but d'en faire ressortir les régions homologues ou similaires. L'objectif d'un alignement est de déterminer à quel point deux séquences sont proches (voir Annexe A.2, Illustration n°3).[7]

C'est ce que réalise le logiciel *VIDJIL* en prenant une séquence en entrée et en la comparant aux autres séquences déjà connues. Néanmoins, l'alignement de séquences tel qu'il est réalisé au sein du programme fait appel à différents types d'alignement et à des méthodes de programmation dynamique.[8]

5 Toutes les cellules du corps humain contiennent le génome humain. Ce génome est identique dans toutes les cellules à l'exception des lymphocytes et des gamètes.
6 C'est pour cette raison que parfois le « D » est mis entre parenthèses dans « V(D)J ».
7 En mathématique on parle de distance de *Levenshtein* pour donner une mesure de différences entre deux chaînes de caractères.
8 Il existe également des méthodes heuristiques comme la méthode *BLAST*.

### 1.2.3 Programmation dynamique

Malgré un nom qui peut porter à confusion, le terme de programmation dynamique signifie en réalité le fait de résoudre algorithmiquement des problèmes d'optimisation. Plus concrètement il s'agit de décomposer un problème en sous-problèmes plus simples à résoudre et dont la résolution de l'ensemble des sous-problèmes amène à la résolution du problème initiale.[9]

Ce principe de décomposition intervient directement lors d'un alignement de séquences. En effet, pour comparer deux séquences entre elles, l'usage de la programmation dynamique veut que les séquences soient comparées caractère à caractère.

De ces comparaisons de caractères on peut en tirer un coût, qui additionné au coût de toutes les comparaisons des caractères constituant les deux séquences, formera un coût total.

Ce coût total servira alors « *d'indice de ressemblance* » entre les deux séquences, il ne restera ainsi qu'à faire de même entre toutes les séquences et à déterminer quelle est la séquence la plus semblable à notre séquence initiale.

Pour comparer un caractère à un autre on dispose de trois opérations : la substitution, la délétion et l'insertion. La première consiste à remplacer un caractère par un autre (voir Annexe A.2, Illustration n°4) tandis que les deux autres servent à retirer ou à ajouter un caractère (dans le cas où les deux séquences n'ont pas la même longueur) (voir Annexe A.2, Illustration n°5).

Généralement, on ajoute un coût plus élevé à la délétion et à l'insertion qu'à la substitution. Dans la substitution on peut même différencier deux cas : substituer un caractère par un autre et substituer un caractère par lui-même (si les deux caractères sont identiques). Dans le second cas le coût peut être moins élevé que dans le premier, voire nul.

Néanmoins l'appréciation des coûts est laissée à l'algorithmicien et aux priorités qu'il définit. En ce qui concerne *VIDJIL* les coûts sont définis Annexe A.2, Illustration n°6.

Ils signifient que lors d'un alignement de séquences, le logiciel privilégiera la substitution d'un caractère par lui-même et l'insertion de caractères plutôt que le remplacement d'un symbole par un autre et la délétion (coût le plus faible).

Une fois les coûts définis, il ne reste qu'à les appliquer à tous les caractères d'une séquence et à les additionner pour en tirer le coût global. Ce dernier sera comparé au coût global de tous les alignements et ainsi il sera possible de trouver le meilleur, car il s'agît de celui qui a le plus grand coût global.

9 Parmi les applications algorithmiques de la programmation dynamique on y retrouve des algorithmes tels que l'algorithme *Bellman-Ford*, algorithme de *Floyd-Warshall*, l'algorithme *CYK* et bien d'autres encore.

### 1.2.4 Alignement global

Il existe plusieurs alignements de séquences, le plus général étant l'alignement *global*. Il consiste à calculer la distance entre l'ensemble d'une séquence X et l'ensemble d'une seconde séquence Y. L'algorithme « *Needleman-Wunsch* » permet de réaliser des alignements globaux de manière optimale.

Selon cet algorithme, pour calculer un alignement de séquences, on définit en premier lieu une matrice de similarité. Cette matrice porte en abscisse et en ordonnée les deux séquences à aligner (voir Annexe A.2, Illustration n°7).

En suite la première ligne et la première colonne sont initialisées avec une valeur quelconque, en général 1 ou 0. Puis Le calcul des coefficients par case s'effectue du haut vers le bas et de la droite vers la gauche. Ainsi la première case à être calculé est bordée par des cellules arborant les valeurs initiales (en haut, à droite et en haut à droite).

Lorsqu'on passe d'une case à une autre en effectuant un mouvement de translation horizontal vers la droite, il s'agit d'une délétion. Une translation verticale vers le bas indique une insertion. Enfin une translation en diagonale vers le bas à droite indique une substitution. Le coût de la case est obtenu en prenant le meilleur coût des trois opérations (voir Annexe A.2, Illustration n°8).

Une fois que la matrice est remplie, on s'intéresse à la dernière case en bas à droite, elle contient le coût total de l'alignement des deux séquences. Pour obtenir l'alignement correspondant il suffit de remonter la matrice à partir de la dernière case jusqu'à la case initiale en passant par les cases ayant le meilleur coût (il y a plusieurs chemins possibles) (Voir Annexe A.2, Illustration n°9).[9]

L'algorithme opérant sur une matrice N x M, sa complexité en temps dans le pire des cas est O(NM).

Il existe également une méthode où on définit une matrice des opérations, cette matrice met en relation un caractère avec tous les autres caractères qu'il est possible de rencontrer et définit pour chacun un coût. On peut donc définir autant de matrices d'opérations qu'il y a d'opérations.

### 1.2.5 Alignement local

À la différence de l'alignement global, l'alignement local se contente de calculer la distance entre un sous-mot d'une séquence X et un sous-mot d'une séquence Y. Autrement dit l'alignement local est conçu pour rechercher dans la séquence X des régions semblables à Y (ou à des parties de Y) (voir Annexe A.2, Illustration n°10).[10]

Cet alignement est généralement réalisé par l'algorithme « *Smith-Waterman »*. Cette fois-ci dans la matrice d'alignement on ne cherche plus à rejoindre les deux coins opposés de la matrice, mais à aligner des sous-chaînes des séquences entre elles.

### 1.2.6 Alignement semi-global

L'alignement semi-global est basé sur le même principe que celui de l'alignement local. Le calcul de la distance s'effectue entre une séquence X et un sous-mot d'une séquence Y. L'algorithme reste le même.

### 1.2.7 Génie logiciel

Le logiciel est rigoureusement développé à l'aide de la plateforme *GitLab*, qui permet d'inclure à un dépôt GIT un système d'intégration continue via des *pipelines*[11]. Cela permet que les tests unitaires, fonctionnels et de mémoires soient lancés sur chaque nouveau *commit* poussé sur la branche principale.

En plus de cela, la manière de développer le logiciel repose sur l'utilisation d'*issues*, des espaces de discussion où les développeurs peuvent s'exprimer sur un bug rencontré ou une nouvelle fonctionnalité à ajouter. Dans un cas comme dans l'autre, une nouvelle branche est créée au nom de l'*issue*. Lorsque cette dernière fixe la problématique initiale et qu'elle passe les tests d'intégration continue alors la branche est *mergé* à celle d'origine.

En dehors de *GitLab*, l'équipe de développement utilise également Jenkins. Un outil permettant de tester que la compilation et l'exécution du logiciel se déroule correctement sur de nombreuses plateformes différentes (Debian, CentOS, FreeBSD…).

### 1.2.8 Structure de *VIDJIL*

Le logiciel définit ses propres *types*[10] pour faciliter le traitement de séquences, notamment avec des objets tels que des conteneurs de séquences, des sous-chaînes de caractères ou encore des analyseurs de fichiers *FASTA*[11].

Ainsi dans la suite de ce rapport, *BioReader* fera référence à un objet contenant des séquences biologiques, *Germline* désignera un conteneur de *BioReader* et *KmerAffect*[12] une sous-chaîne de caractère contenant une information et un label.

La fonction réalisant l'alignement de séquences dans *VIDJIL*, *align_against_collection* prend en paramètre un *BioReader* et la séquence à aligner. À l'origine, cette dernière était alignée à toutes les séquences du *BioReader*.

Or, aligner des séquences est ce qui prend le plus de temps dans le programme et, donc il est critique d'améliorer ce point. Comme vu précédemment il n'est pas possible d'améliorer l'alignement de séquences en elles-mêmes. Néanmoins il est possible de réduire le nombre de séquences dans l'objet *BioReader* en entrée pour ne pas avoir à aligner toutes les séquences.

10 Contrairement au C, le C++ est orienté objet et il est possible de définir des types de données contenant des fonctions et des attributs qui leur sont propres.

11 En bio-informatique, le format FASTA est un format de fichier texte représentant des séquences biologiques. Chaque séquence ayant un en-tête et un corps constitué de lettres représentant la séquence en question. Pour une séquence ADN, on retrouve les lettres A, C, G et T décrivant les bases azotées.

12 Un K-mer représente une sous-chaîne d'une séquence.

# 2 Contribution

## 2.1 Durant le label recherche

Durant les trois mois précédant le stage, je me suis familiarisé avec l'automate d'*Aho-Corasick*. Ce dernier était déjà implémenté dans le logiciel (pour isoler les gènes *V*, *D* et *J* d'une séquence) mais inexploité pour la réduction du nombre de comparaisons.

### 2.1.1 Automate d'*Aho-Corasick*

L'automate (ou algorithme) d'*Aho-Corasick*[13] est un algorithme qui consiste à avancer dans une structure de données abstraite appelée dictionnaire. Ce dictionnaire contient les mots recherchés. Cet algorithme est adapté efficacement, garantissant que chaque lettre n'est lue qu'une seule fois. Une fois implanté, l'algorithme adopte une complexité linéaire.

La construction d'un tel automate est réalisée de la manière suivante :

- Le point d'entrée se fait à partir d'un état vide.
- Ajout d'un mot connu à partir de l'état vide où chaque état forme une sous-chaîne et où chaque transition comporte la lettre suivant la sous-chaîne. Ainsi l'état acceptant d'une branche correspond au mot initial. Réitérer l'opération autant de fois qu'il y a de mots (voir Annexe A-3, Illustration n°11).
- Associer à chaque état un état de repli. Un état de repli étant un état correspondant au plus long suffixe d'un sous-mot *v*, différent de *v* et qui soit également préfixe de la feuille de la branche de l'état. (voir Annexe A-3, Illustration n°12). S'il existe un état final défini en tant qu'état de repli alors l'état initial devient lui-même état de repli.

Une fois l'automate construit de la sorte, on fait passer notre séquence à aligner en entrée de celui-ci et pour chaque état final atteint on incrémente son score. À la fin lorsque la séquence est entièrement parcourue il suffit de regarder les états et leurs nombres d'occurrences pour déterminer quelles sont les séquences les plus probables de correspondre à notre séquence en entrée (voir Annexe A-3, Illustration n°13).[12]

13 Son nom vient d'Alfred Aho et Margaret Corasick. Publié en 1975, l'algorithme a été utilisé initialement dans le programme utilitaire grep sous UNIX.

### 2.1.2 Fonction *getMultiResults*

Initialement, pour utiliser les informations renvoyées par l'automate il fallait utiliser la méthode *getResults*[14] (voir Annexe B.1). Mais cette dernière ne renvoyait qu'une portion des informations. Il a donc fallu définir une nouvelle fonction chargée de renvoyer l'intégralité de ces données. C'est de cette nécessité qu'a été créée *getMultiResults* (voir Annexe B.2).

Là où la fonction initiale récupérait une information par état de l'automate, *getMultiResults* retourne toutes les données. Alors que *getResults* renvoyait un *vecteur* [15] d'informations, la nouvelle méthode renvoie un tableau associatif, ayant pour *clefs*[16] les K-mers présents dans les états et en *valeurs* le nombre de fois que ces K-mers apparaissent à travers ces derniers.

L'algorithmie de cette routine repose sur deux boucles. La première parcourant les états de l'automate et la seconde qui pour un état donné parcourt les informations qu'il contient. Si pour un état on rencontre plusieurs fois le même K-mer alors on ne l'ajoute qu'une seule fois à notre tableau associatif et on augmente son nombre d'occurrences en tant que *valeur*.

Si pour la première boucle il y a **N** états et que chaque état contient au maximum **M** informations alors on peut calculer que la complexité en temps de cette méthode dans le pire des cas est :

Il y a 2NM affectations.
(L6 + L7 + L10 + NL11 + NM(L14 + L17))
Il y a 6NM lectures.
(L7 + L10 + N(L10 + 3L11 + 2L12) + NM(L14 + 3L16 + 3L17))
Il y a NM opérations mathématiques.
(NL10 + NML17)
Il y a NM instructions.
(L4 + L5 + L6 + L7 + N(2L11 + L12 + L13) + NM(2L20) + L 24)

Texte 1: Calcul de la complexité de *getMultiResults* par étapes

Soit au total 10 N x M dans le pire des cas. La constante étant ignorée dans les calculs de complexité on peut se rapporter à O(N x M). Cette complexité quadratique diffère de la complexité constante de la méthode *getResults* où dans le pire des cas on obtenait O(N).

Même si l'augmentation de la complexité peut sembler défavorable, la résolution de cette méthode prend un temps moindre comparé à l'alignement de séquences. On peut donc se permettre d'augmenter sensiblement le temps de cette partie de l'algorithme pour réduire considérablement le temps prit par le reste.

14 Dans la suite du document, les noms des objets et des fonctions respectent la convention de nommage « CamelCase ». Ainsi les fonctions commenceront par une minuscule et les objets par une majuscule.
15 En C++ les vecteurs sont des tableaux dynamiques dont la taille peut varier. Il ne faut pas les confondre avec les listes, structures de données similaires mais stockées sur des zones de mémoires discontinues (et donc plus lentes à manipuler).
16 Dans un tableau associatif, les clefs sont supposées uniques mais les valeurs peuvent être les mêmes.

## 2.2 Durant le stage

L'objectif du stage était de traiter les données renvoyées par *getMultiResults* et les utiliser pour filtrer le nombre de séquences à aligner. Tout cela en respectant le principe *TDD*[17] pour garantir le bon comportement du logiciel.

### 2.2.1 Fonction *buildACAutomatonToFilterBioReader*

Comme son nom l'indique, cette fonction a pour but de construire l'automate d'*Aho-Corasick* dans le but de filtrer un *BioReader*. Cette méthode prend en paramètre le *BioReader* que l'on souhaite filtrer, ainsi qu'une *graine*.

L'idée principale est d'insérer les gènes connus dans l'automate et que dans la seconde méthode *filterBioReaderWithACAutomaton*, *getMultiResults* renvoie un tableau associatif contenant le nom des gènes et leur nombre d'occurrences. Mais comme l'automate peut s'avérer grand (plus d'un million d'états), il est nécessaire que chaque information soit enregistrée sur un espace réduit.

C'est là qu'intervient le *KmerAffect*, un objet *Kmer* mais qui contient en plus un label stocké sur un octet (c'est la plus petite taille manipulable en C++), c'est-à-dire un objet de type *char*.

L'idée est que chaque label de *KmerAffect*, chaque caractère, représente un gène du *BioReader* d'origine. De cette manière il est possible de savoir à quels gènes appartiennent les *KmerAffect* en sortie de *getMultiResults*.

Mais comme il est impossible de dire si un gène contient ou non des allèles (séquences de la même famille qui sont proches de lui), on utilise également un vecteur d'entiers dans lequel est enregistrée les positions des gènes dans l'objet *BioReader*.

De cette manière, chaque indice du vecteur représente un *KmerAffect* et la valeur à cet indice correspond au nombre d'allèles pour le gène en question.

Cette graine est utilisée pour découper toutes les séquences du *BioReader* lors de leur insertion dans l'automate.

17 Le « TDD », Test Driven Development ou en français développement piloté par les tests, est une technique de développement logiciel qui préconise d'écrire les tests unitaires avant le code source d'un logiciel.

### 2.2.2 Fonction *filterBioReaderWithACAutomaton*

C'est cette fonction qui réalise le filtrage à proprement parler. En effet, elle prend en paramètres l'automate créé, le vecteur d'index correspondant au *BioReader* initial, le *BioReader* initial et la séquence à aligner.

Dans un premier temps, cette routine va récupérer les résultats de *getMultiResults* en provenance de l'automate et va itérer sur le tableau associatif renvoyé. Ce tableau contenant en clef un objet *KmerAffect* et en valeur son nombre d'occurrences peut être trié selon un paramètre de la fonction.

Si ce dernier est précisé, alors le tableau associatif est trié selon le nombre d'occurrences des K-mers et le paramètre correspond au nombre de K-mers les plus représentés qui seront gardés pour le filtrage. Si le paramètre n'est pas précisé alors tous les K-mers du tableau associatif sont utilisés.

Dans les deux cas, l'algorithme de filtrage récupère les K-mers du tableau associatif, et grâce à leur label il détermine à quel gène appartient ce K-mer. Et, donc il peut insérer depuis le *BioReader* d'origine, les gènes retrouvés dans un nouveau *BioReader* qui sera résultat du filtrage.

### 2.2.3 Intégration dans le logiciel

Durant l'intégralité du stage, le travail a été réalisé sur une branche différente de la branche principale (*VIDJIL* utilisant le logiciel de gestion de versions *GIT*). De ce fait il a fallu fusionner les deux branches.

Néanmoins, de base le logiciel ne filtre pas les séquences et ne construit pas l'automate sur celles connues. C'est pourquoi il a fallu ajouter ce cas-là comme un cas optionnel. Le logiciel utilisant des options d'exécution traités par *getopt*.[18]

Ce qui a été proposé fût de considérer une option « **-Z** » et un argument entier. Cet argument spécifiant la spécificité de filtrage de l'automate. Ainsi pour les différentes valeurs **N** avec l'option **-Z** nous obtenons :

- « **-Z all** » ou « **-Z -1** » : Ne pas construire l'automate et ne pas effectuer de filtrage. C'est ce qui est réalisé par défaut quand rien n'est spécifié.
- « **-Z 0** »: Effectuer le filtrage sur tous les K-mers apparaissant au moins une fois dans *getMultiResults*.
- « **-Z N** » : Effectuer le filtrage sur les N meilleurs K-mers (Avec N > 0).

18 « getopt » est une fonction de la librairie C, utilisée pour analyser les options d'une ligne de commande.

### 2.2.4 Résultats

À l'aide d'un script shell, j'ai lancé la commande time sur différentes lignes de commande *VIDJIL* lorsque le logiciel fait appel au filtrage. Sur les petits jeux de données le gain de temps est minime. Mais pour de grands nombres de séquences on observe un gain significatif.

```
./vidjil-algo -c segment -g germline/homo-sapiens.g:IGH -x 10000 demo/Stanford_S22.fasta
```

Cette commande signifie que 10 000 séquences du fichier *demo/Stanford_S22.fasta* seront lancées avec comme gènes de comparaisons *germline/homo-sapiens.g:IGH*. Comme le paramètre **-Z** est absent, cela signifie que le filtrage n'a pas lieu.

Le temps d'exécution de cette commande est d'environ 7000 secondes.

En incluant le filtrage dans la commande :

```
./vidjil-algo -c segment -g germline/homo-sapiens.g:IGH -x 10000 -Z 1 demo/Stanford_S22.fasta
```

Ici on filtre en ne gardant que la séquence la plus représentative. Le temps d'exécution est réduit à environ 300 secondes.

Dans le meilleur des cas qui a été observé, le gain de temps le plus significatif est de 96 % (le temps mis par le logiciel a été réduit de 96 %). De manière général, le gain de temps tourne autour de 30 %.

## Conclusion

L'objectif principal du stage, qui était de réduire le nombre de comparaisons dans *VIDJIl*, a été entièrement réalisé au bout de ces trois mois. Le cahier des charges a été complété par l'intégration d'une ligne de commande propre au filtrage, une approche orientée objet pour la classe de filtrage, validés par de nombreux tests unitaires et fonctionnels.

Aussi l'exécution de cette partie du programme a été méthodiquement analysé, notamment par des mesures de vitesse. Il a été révélé que grâce à l'implémentation d'un filtre pour réduire le nombre de comparaisons de VIDJIL, on gagnait un temps moyen de 30 %, pouvant grimper jusqu'à 95 % dans certains cas.

Ce stage m'aura apporté de nombreuses connaissances, à commencer par la découverte de nouvelles thématiques (programmation dynamique et algorithmique textuelle). Puis j'ai également appris à développer et à documenter ce que je développais avec plus d'efficacité (dans les messages de commits).

J'ai aussi appris à faire preuve d'un certain sens critique concernant les résultats des mesures de temps et à toujours remettre en perspective les gains potentiels qu'apportent les méthodes de filtrage.

Enfin j'ai pu mettre en application les connaissances enseignées en licence informatique. En effet, j'ai eu à traiter de tests unitaires, de complexité algorithmique, de programmation orientée objet et de principes de développement. Toutes ces notions qui sont au cœur de la licence informatique témoignent de leur utilité dans un contexte professionnel.

En dehors des compétences techniques pures que requiert le monde du développement, j'ai également appris à participer à des réunions d'équipe où chacun fait part de ses avancées. Cela est nécessaire pour avoir le point de vue des autres membres sur notre propre travail et échanger sur la cohérence, l'amélioration et des détails le concernant.

## Bilan

D'un point de vue personnel je suis satisfait des contributions que j'ai pu apporter au projet *VIDJIL*. Les thématiques pouvant sembler compliquer au premier abord me paraissent bien plus simples maintenant. Je suis également satisfait de l'intégration du filtrage au sein du logiciel.

D'un point de vue professionnel, j'ai beaucoup appris de ce stage. J'ai découvert le monde de la recherche bien plus que je ne l'ai fait à travers mon label Recherche. J'ai pu développer un regard critique et prendre du recul sur les résultats obtenus par mon algorithme. Plus généralement je pense avoir acquis une plus grande rigueur scientifique.

Enfin, je pense que ce stage constitue un premier pas vers le métier de chercheur que je souhaite faire après mes études. De la même manière les thématiques abordées m'ont aidé à me positionner sur ma poursuite d'études à travers mon application pour un master *MOCAD*.

## Bibliographie

[1] :COLOT Olivier. *Page d'accueil CRIStAL*. [en ligne]. 2015. [consulté le 18 juin 2018]. Disponible à l'adresse :https://www.cristal.univ-lille.fr/?rubrique1
[2] :TISON Sophie. *Page LIFL de Bonsai*. [en ligne]. 2012. [consultation le 18 juin 2018]. Disponible à l'adresse :http://www.lifl.fr/Recherche/Equipes/Presentationafe5.html?eid=4
[3] :Hélène Touzet. *site LIFL de Bonsai*. [en ligne]. Début 2000. [consulté le 18 juin 2018]. Disponible à l'adresse :http://www.lifl.fr//bonsai/
[4] :Hélène Touzet. *site Bonsai, membres de l'équipe*. [en ligne]. 2018. [consulté le 18 juin 2018]. Disponible à l'adresse :http://www.lifl.fr/bonsai/people
[5] :Hélène Touzet. *site Bonsai projects*. [en ligne]. date inconnue. [consulté le 18 juin]. Disponible à l'adresse :http://www.lifl.fr/bonsai/start#some_of_our_projects
[6] :Pierre Mounier-Kuhn .L'informatique en France de la seconde guerre mondiale au Plan Calcul..2010.. Disponible à l'adresse: .
[7] :Antoine Petit. *La Fondation INRIA - A propos*. [en ligne]. 2017. [consulté le 18 juin 2018]. Disponible à l'adresse :http://www.inria-foundation.fr/about/
[8] :Antoine Petit. *Le Consortium VidjilNet incubé au sein de l'action InriaSoft*. . 2018. [consulté le 18 juin 2018]. Disponible à l'adresse :http://www.inria-foundation.fr/incubation/inriasoft/2017/12/11/VidjilNet-incub%C3%A9-par-InriaSoft.html
[9] :BLOCH Laurent. *ALIGNEMENT DE SÉQUENCES - Algorithme de Needleman et Wunsch*. . 2008. . Disponible à l'adresse : https://www.laurentbloch.net/MySpip3/Algorithme-de-Needleman-et-Wunsch.
[10] :GAUTHERET Daniel. *Alignement local ou global*. [en ligne]. 2012. [consulté le 18 juin 2018]. Disponible à l'adresse :http://rna.igmors.u-psud.fr/gautheret/cours/localglobal.html
[11] :GitLab company. *GitLab Continuous Integration & Deployment*. [en ligne]. . [consulté le 20 juin 2018]. Disponible à l'adresse :https://about.gitlab.com/features/gitlab-ci-cd/
[12] :CIURLIK Yohann. *Recherche de mots dans un texte – Aho-Corasick*. [en ligne]. 2007. [consulté le 18 juin 2018]. Disponible à l'adresse :https://www.spawnrider.net/2007/01/17/aho_corasick/

# Annexes

## Annexes A

### A.1 Principe immunitaire

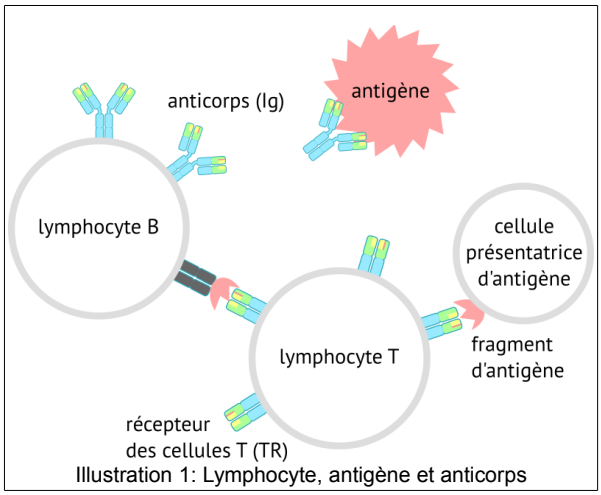

Illustration 1: Lymphocyte, antigène et anticorps

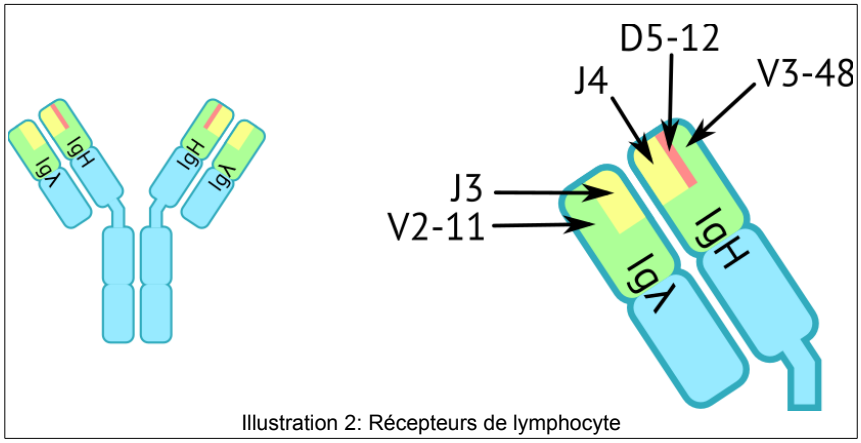

Illustration 2: Récepteurs de lymphocyte

## A.2 Alignement de séquences

```
AAB24882        TYHMCQFHCRYVNNHSGEKLYECNERSKAFSCPSHLQCHKRRQIGEKTHEHNQCGKAFPT 60
AAB24881        --------------------YECNQCGKAFAQHSSLKCHYRTHIGEKPYECNQCGKAFSK 40
                                    ****: .***:  * *:** * :****.:* *******..

AAB24882        PSHLQYHERTHTGEKPYECHQCGQAFKKCSLLQRHKRTHTGEKPYE-CNQCGKAFAQ- 116
AAB24881        HSHLQCHKRTHTGEKPYECNQCGKAFSQHGLLQRHKRTHTGEKPYMNVINMVKPLHNS 98
                 **** *:***********:***:**.: .***************     :  *.: :
```

Illustration 3: alignement de séquence réalisé par le logiciel *ClustalW* entre deux protéines humaines

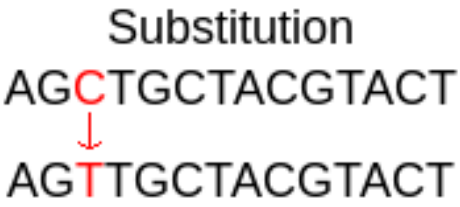


Illustration 4: Substitution d'un caractère par un autre

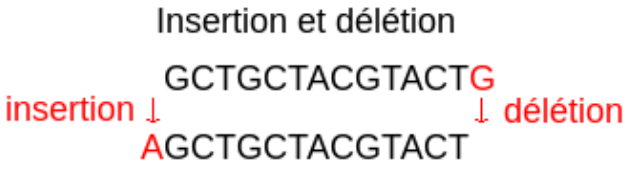


Illustration 5: Insertion et suppression de caractère

Substitution d'un caractère par un autre : -6

Substitution d'un caractère par lui-même : +4

Délétion d'un caractère : - 10

Insertion d'un caractère : - 1

Illustration 6: Coûts des opérations dans l'algorithme de *VIDJIL*

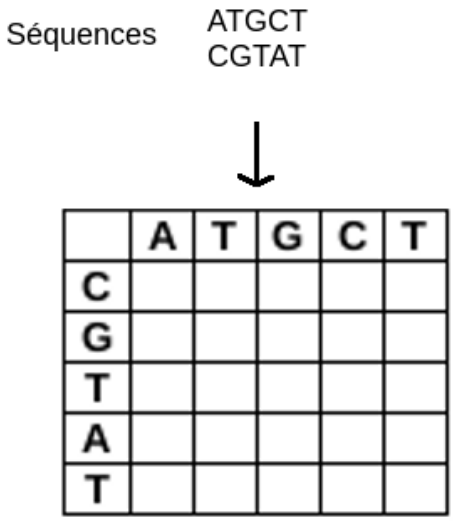


Illustration 7: Insérer les séquences dans une matrice d'alignement

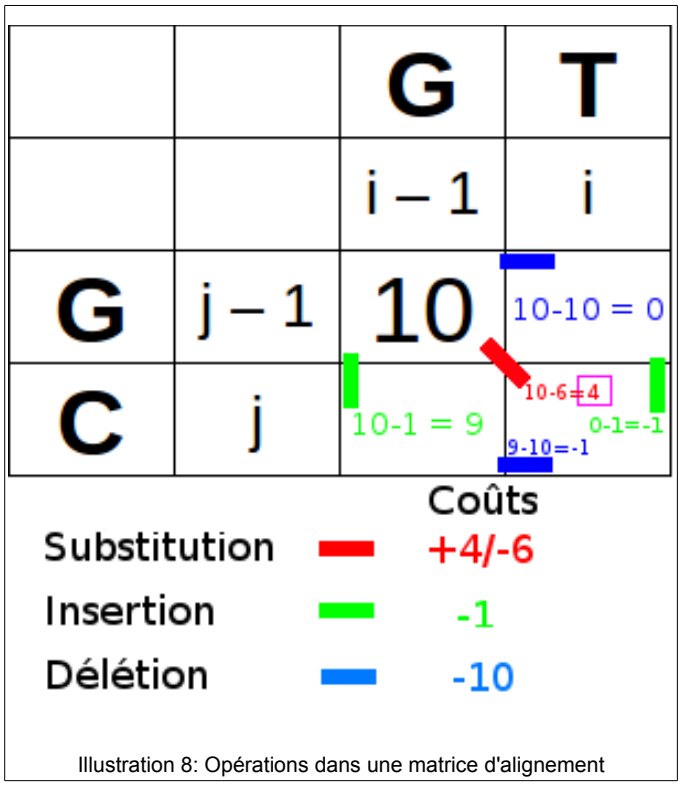


Illustration 8: Opérations dans une matrice d'alignement

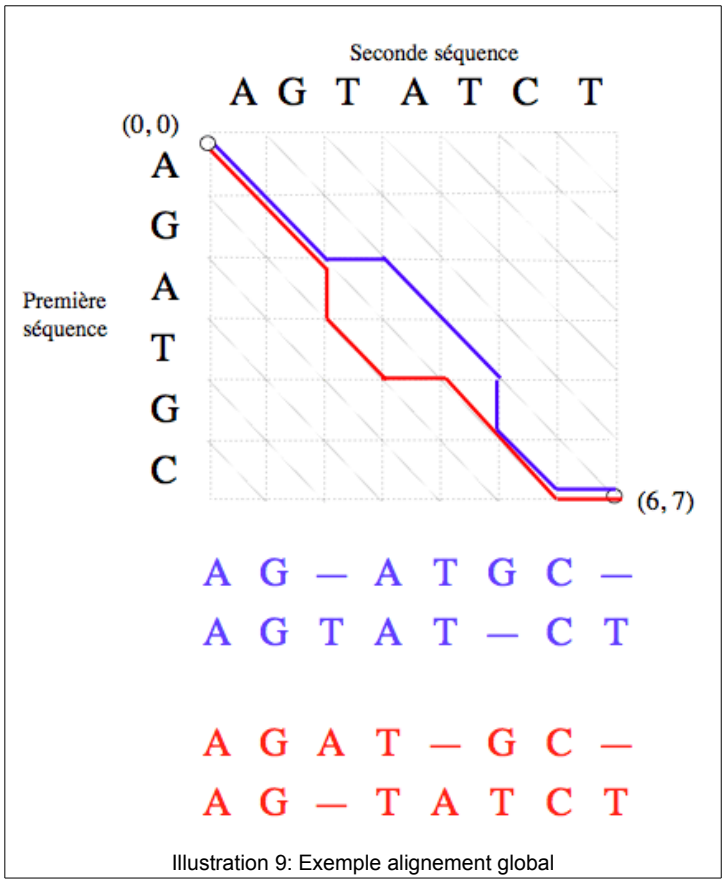


Illustration 9: Exemple alignement global

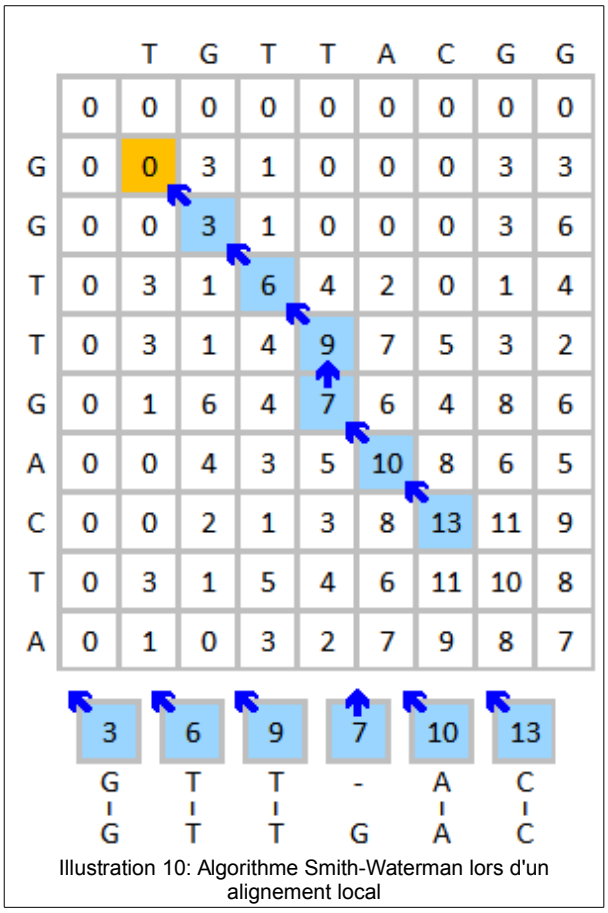


Illustration 10: Algorithme Smith-Waterman lors d'un alignement local

## A.3 Automate d'*Aho-Corasick*

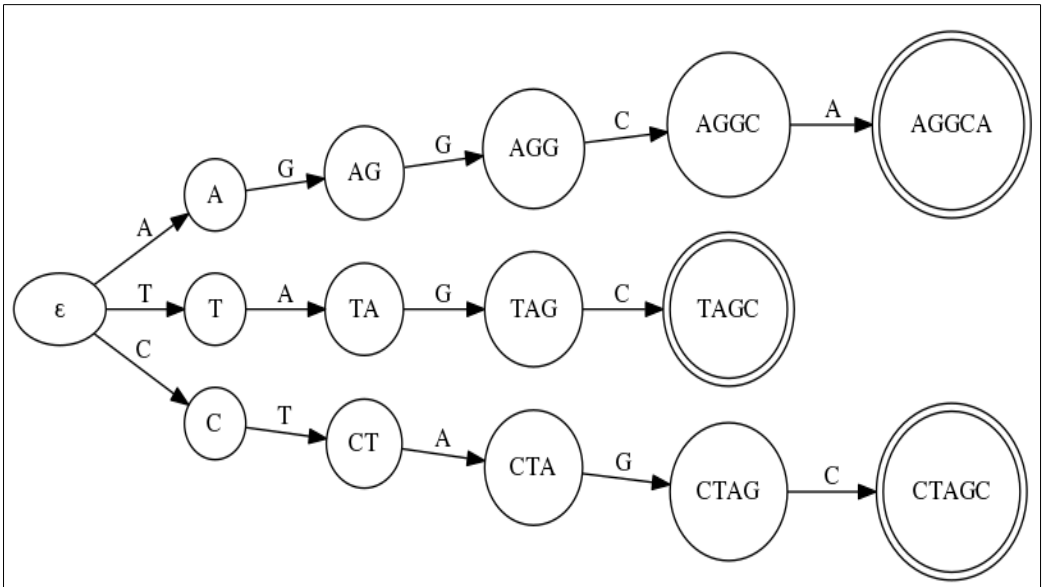


Illustration 11: Construction de l'automate d'Aho-Corasick sur les mots "AGGCA", "TAGC" et "CTAGC"

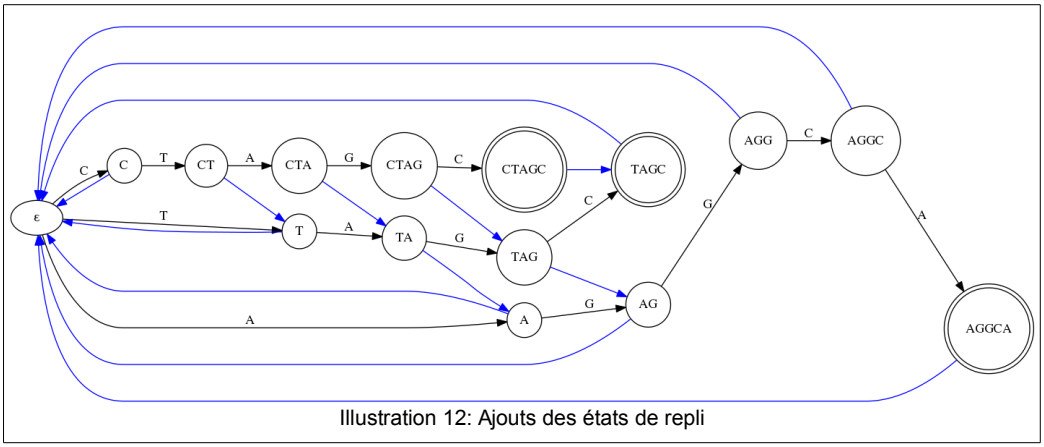


Illustration 12: Ajouts des états de repli

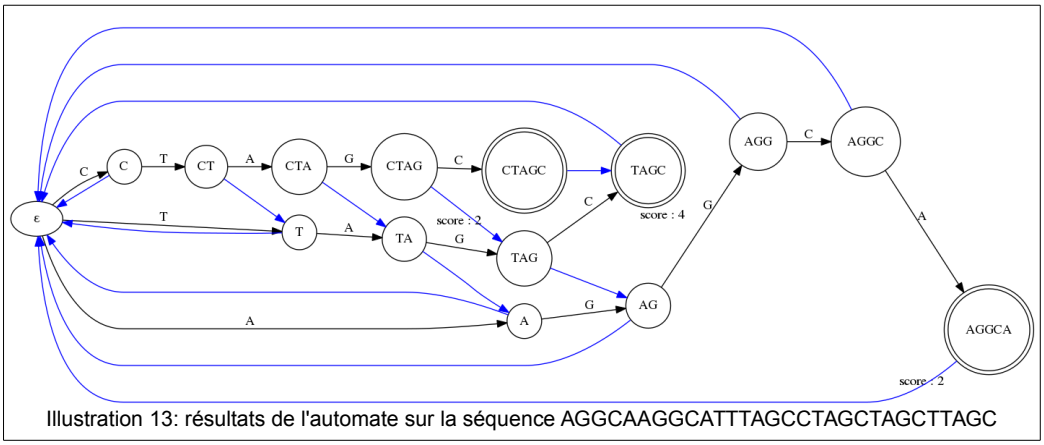


Illustration 13: résultats de l'automate sur la séquence AGGCAAGGCATTTAGCCTAGCTAGCTTAGC

# Annexes B

## B.1 Fonction *getMultiResults*

```
1 : template <class Info>
2 : map<Info, int> PointerACAutomaton<Info>::getMultiResults(const seqtype &seq, bool
3 : no_revcomp, string seed) {
4 :  UNUSED(no_revcomp);
5 :  UNUSED(seed);
6 :  pointer_state<Info>* current_state = getInitialState();
7 :  size_t seq_len = seq.length();
8 :  map<Info, int> results;
9 :
10 :  for(size_t i = 0;i < seq_len;++i) {
11 :    current_state = (pointer_state<Info> *)next(current_state, seq[i]);
12 :    set<Info> informations(current_state->informations.begin(),
13 :      current_state->informations.end());
14 :    for(auto const& info : informations){
15 :      /* If map contain info, increase its occurence. */
16 :      if(results.count(info) > 0){
17 :        results[info] = results[info] + 1;
18 :      }
18 :      /* Otherwise add info into map with a value of 1. */
19 :      else{
20 :        results.insert(pair<Info,int>(info,1));
21 :      }
22 :    }
23 :  }
24 :  return results;
25 :}
```

## B.2 Fonction *getResults*

```
1 :template <class Info>
2 :vector<Info> PointerACAutomaton<Info>::getResults(const seqtype &seq, bool no_revcomp,
3 :string seed) {
4 :  UNUSED(no_revcomp);
5 :  UNUSED(seed);
6 :
7 :  pointer_state<Info>* current_state = getInitialState();
8 :  size_t seq_len = seq.length();
9 :  vector<Info> result(seq.length());
10 :
11 :  for (size_t i = 0; i < seq_len; i++) {
12 :    current_state = (pointer_state<Info> *)next(current_state, seq[i]);
13 :    Info info = current_state->informations.front();
14 :    result[i - info.getLength()+1] = info;
15 :  }
16 :  return result;
17 :}
```