Master Bioinformatique

[HAU803I] : Développement opérationnel avancé (DevOps)

Rapport de projet de DevOps : API de mapping

Auteur :
Tiziri TAMANI

Professeur :
Alban MANCHERON

Version du 9 avril 2025

Table des matières

Table des figures

Liste des tableaux

Introduction

Le séquençage haut débit (Illumina, 454,..) a révolutionné la génomique en permettant l'analyse massive de données biologiques. Cependant, cette quantité colossale de données pose d'importants défis informatiques, notamment pour le mapping des reads sur un génome de référence. Ce projet vise à développer une solution efficace pour localiser précisément des millions de reads sur un génome de référence, une étape cruciale pour des applications comme l'étude des variations génétiques, l'analyse transcriptomique ou le diagnostic médical.

Les algorithmes comme ceux utilisés dans BWA, Bowtie ou encore SOAP ont été conçus pour répondre à ce besoin, en combinant des structures d'indexation efficaces — comme les tables de hachage, les arbres suffixes ou le FM-index basé sur la transformation de Burrows-Wheeler — avec des heuristiques permettant de réduire le temps de recherche tout en maintenant une précision acceptable. Ces outils ont largement démontré leur robustesse, mais présentent parfois des limites, notamment en termes de flexibilité dans la gestion des erreurs, de consommation mémoire ou de complexité du code source, souvent écrit en C, de manière peu modulaire. Certains intègrent aussi des approches basées sur les k-mers, mais rarement combinées avec une analyse spatiale des positions, ce qui laisse un espace pour explorer de nouvelles stratégies d'alignement.

Concrètement, l'algorithme commence par lire et valider les fichiers d'entrée au format FASTA ou FASTQ, puis à construire une structure d'indexation efficace permettant de localiser efficacement les occurrences de k-mers dans le génome. À partir de cette indexation, les k-mers extraits des reads, qui sont ensuite analysés en prenant en compte la cohérence des positions pour établir un score d'alignement pertinent. Cette logique permet de détecter les alignements les plus probables.

Enfin, une attention particulière est portée à l'optimisation des performances, en utilisant des structures de données contiguës et une gestion fine de la mémoire, ce qui rend notre solution particulièrement adaptée aux exigences de la bioinformatique moderne. L'ensemble du développement s'appuie sur des pratiques rigoureuses issues du DevOps, telles que le versionnage avec Git ou les tests automatisés, offrant ainsi un équilibre optimal entre performance et maintenabilité.

1. Généralités

1.1 Caractéristiques et performances du séquençage Illumina (short-reads)

Le séquençage Illumina, est le leader de la technologie des short-reads, il produit des fragments de 50 à 300 pb avec une précision élevée (>99,9%) et un débit impressionnant (des milliards de reads en un seul run). Ses principaux avantages résident dans son coût réduit par base, sa reproductibilité et sa capacité à détecter efficacement les variants simples (SNPs, petites indels). Cependant, cette approche présente des limites notables pour les insertions et délétions (indels) supérieures à 10-15 pb, en raison de la difficulté à aligner précisément de courts reads autour de ces variations structurelles. La courte taille des fragments complique également l'assemblage de novo et le mapping dans les régions répétitives, tandis que les artefacts PCR (duplications et biais de couverture) peuvent fausser les analyses. Bien que moins performant que le long-read pour les variations structurales complexes, Illumina reste la référence pour les études nécessitant une quantification précise (RNA-seq, épigénétique) ou un séquençage ciblé hautement précis.

1.2 Optimisation des algorithmes de séquençage : Programmation orientée objet en C++ pour une analyse bio-informatique

L'application de la programmation orientée objet en C++ dans le traitement des données de séquençage haut débit Illumina permet d'allier hautes performances, modularité, flexibilité et maintenabilité. Ce paradigme est particulièrement adapté pour gérer des projets bioinformatiques complexes, en offrant une solution robuste, évolutive et efficace face aux défis posés par l'analyse de volumes massifs de données.

L'utilisation de la POO en C++ offre aussi un cadre propice à l'intégration de nouvelles fonctionnalités sans perturber l'architecture existante. Les classes et objets permettent d'ajouter de nouveaux modules (comme un module d'analyse spatiale ou un module de scoring d'alignement) de manière transparente, en exploitant les principes du polymorphisme et de l'héritage pour étendre les fonctionnalités tout en préservant la structure du code.

Enfin, un autre avantage non négligeable de la POO en C++ réside dans son interopérabilité avec d'autres langages et outils bioinformatiques, permettant de tirer parti des bibliothèques existantes et de les intégrer dans des solutions plus grandes et plus complexes. Ainsi, la POO en C++ offre non seulement des performances de haut niveau, mais également une souplesse et une adaptabilité accrues pour s'aligner avec les besoins changeants de la bioinformatique moderne.

1.2.1 Programmation orientée objet en biologie computationnelle

Dans ce contexte, la programmation orientée objet (POO) en C++ offre une approche particulièrement adaptée grâce à sa capacité à gérer des projets complexes, tout en fournissant une flexibilité, une réutilisabilité et une maintenabilité accrues.

L'un des avantages majeurs de la programmation orientée objet réside dans la modularité qu'elle permet. Les structures de données, telles que les arbres suffixes, les tables de hachage ou encore les index basés sur la transformation de Burrows-Wheeler, peuvent être encapsulées dans des classes, permettant une gestion plus claire et une évolution plus facile des algorithmes au fil du développement. La possibilité d'implémenter des structures complexes sous forme d'objets distincts, chacun avec des responsabilités définies, facilite le test et la maintenance du code.

1.2.2 Avantages de la programmation orientée objet en C++

C++ offre de nombreux avantages pour le traitement des données de séquençage, en particulier grâce à sa flexibilité et à sa gestion fine de la mémoire. Voici les principaux atouts de ce langage dans le contexte bioinformatique :

— **Templates :** La possibilité de créer des structures de données génériques permet d'adapter les algorithmes à différents types de données de séquençage, réduisant ainsi la duplication de code et assurant une meilleure évolutivité du système.

— **Contrôle de la mémoire :** C++ permet une gestion fine de la mémoire, essentielle pour manipuler de grandes quantités de données génomiques. Cela permet d'optimiser les performances en ajustant l'utilisation des ressources selon les besoins.

— **Programmation de bas niveau :** C++ offre un contrôle direct sur la gestion des ressources matérielles, ce qui permet de maximiser les performances, surtout dans des contextes de traitement intensif de données (par exemple, lors du séquençage).

— **Compatibilité avec les outils bioinformatiques :** C++ est compatible avec de nombreux outils bioinformatiques et bibliothèques populaires (comme BWA, Bowtie), ce qui permet d'intégrer efficacement de nouveaux algorithmes de séquençage dans des systèmes existants.

— **Flexibilité et évolutivité :** Grâce à sa nature orientée objet et à sa gestion des exceptions, C++ permet de développer des solutions qui évoluent facilement avec l'avancement des technologies et des méthodologies de séquençage.

Cette combinaison de contrôle bas niveau,de flexibilité et compatibilité avec les outils bioinformatiques permet de concevoir des systèmes de séquençage hautement performants et adaptables aux besoins futurs.

2. Phases de réalisation du projet

2.1 Matériel et méthodes

- **Assistance IA** : Utilisation de **ChatGPT** et **DeepSEKK** pour :
  - La compréhension des concepts avancés (Suffix Array, K-mer indexing).
  - La génération et la correction des fichiers sources (orthographe, reformulation).
  - La recherche de méthodes d'optimisation et le débogage des erreurs de programmation.

- **Langage et Outils de Développement** :
  - **C++ orienté objet** pour la programmation et la structuration modulaire du code.
  - **VS Code** comme environnement de développement intégré (IDE) sous Linux Mint pour l'édition de code.

- **Gestion de Version** :
  - **GitHub** pour le suivi des modifications et la collaboration en version contrôlée. Le projet est accessible via le lien suivant : cliqure ici le lien de projet.
  - **Makefile personnalisé** pour automatiser la compilation et l'exécution du projet.

- **Documentation Technique** :
  - **Doxygen** pour générer automatiquement la documentation, y compris les diagrammes UML des classes et les descriptions des fonctions.

2.2 Étapes clés de la mise en œuvre

- **Étape 01 :** Développement des outils d'analyse pour les fichiers FASTA et FASTQ.
- **Étape 02 :** Indexation de texte avec la table des suffixe.
- **Étape 03 :** Découpage des reads en k-mers et algorithme de recherche en batch.
- **Étape 04 :** Étape de mapping sur un génome de référence.

2.3 Diagramme des classes

L'architecture des classes reflète une séparation des responsabilités, avec des composants dédiés à chaque tâche.Les détails de ces classes et de leur fonctionnement seront abordés plus en détail dans la suite de ce rapport. Le diagramme présenté ci-dessous illustre l'architecture modulaire d'un système de traitement de séquences biologiques, chacune ayant un rôle spécifique dans le traitement des données. Voici un aperçu des composants principaux :

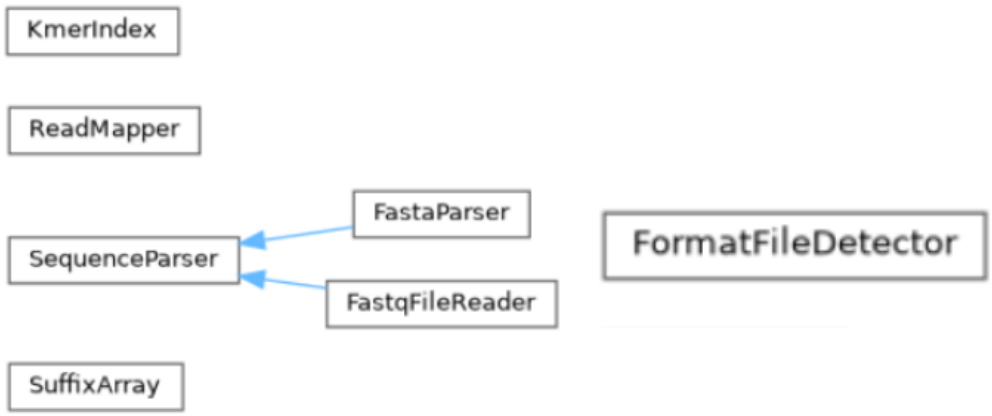

FIGURE 2.1 – Diagramme de dépendance entre les classes du programme.

2.4 Étape 01 : Développement des outils d'analyse pour les fichiers FASTA et FASTQ

2.4.1 Le format FASTA

Le format FASTA permet de représenter des séquences nucléiques ou protéiques, en respectant le code IUPAC. Il a été introduit en 1985 par Lipman et Pearson. Une séquence au format FASTA commence par une ligne de description (en-tête) précédée du symbole ">", parfois ";", suivie de lignes contenant la séquence [1].

2.4.1.1 Implémentation

Afin de faciliter la lecture et l'analyse des fichiers FASTA, j'ai conçu une classe `FastaParser`. Cette classe permet de lire, d'analyser et de manipuler des fichiers au format FASTA. Elle est conçue pour être simple à utiliser, et le tableau ci-dessous présente les principales méthodes de la classe `FastaParser` :

TABLE 2.1 – Fonctions de la classe `FastaParser` et leurs rôles.

Fonction	**Rôle principal**
`explicit FastaParser(const std::string& filePath);`	Constructeur : initialise l'objet avec le chemin du fichier FASTA
`bool loadFile() override;`	Charge le fichier FASTA en mémoire (headers + séquences)
`bool processSequences(const std::function<void(const std::string&, const std::string&)>& callback);`	Mode streaming : lit ligne par ligne et applique une fonction à chaque séquence
`bool validate() const override;`	Vérifie la validité du contenu : headers valides + caractères autorisés
`size_t countSequences() const override { return sequences_size(); };`	Retourne le nombre de séquences dans le fichier
`const std::vector<std::string>& getSequences() const override { return sequences; };`	Accès direct à toutes les séquences chargées
`std::vector<size_t> getSequenceSizes() const;`	Donne la taille (en nucléotides/acides aminés) de chaque séquence
`const std::vector<std::string>& getHeaders() const override { return headers; };`	Accès direct à tous les headers

— Elle repose sur plusieurs méthodes clés, structurées autour de deux modes de lecture : un chargement complet en mémoire (`loadFile`) et une lecture en flux ou streaming (`processSequences`). Ces méthodes visent à fournir à l'utilisateur des outils flexibles pour analyser, valider, et extraire des informations essentielles à partir des fichiers FASTA.

— La méthode `loadFile()` lit le fichier entier, stockant les en-têtes `headers()` et les séquences dans des vecteurs en mémoire. C'est la toute première méthode que j'ai implémentée, et elle est robuste sur des petits fichiers de tests unitaires. Quant à la méthode `processSequences()`, elle permet de traiter le fichier ligne par ligne via une fonction de rappel `callback`, ce qui évite de charger tout le fichier en mémoire – particulièrement utile pour les fichiers volumineux de taille en giga-octets. Cette méthode est conçue pour être utilisée dans des scénarios où la mémoire est limitée ou lorsque l'on souhaite traiter les données au fur et à mesure de leur lecture.

2.4.1.2 Cas traités dans l'implémentation

— Support des en-têtes commençant par ">" ou ";".
— Gestion des espaces dans les séquences, avec un système d'avertissement.
— Vérification des caractères valides selon les types d'acides (ADN, ARN, protéines).
— Détection d'erreur si le fichier ne commence pas par un en-tête.
— Support de séquences sur plusieurs lignes (concaténées lors de la lecture).
— Gestion multi-séquences, avec appariement correct des en-têtes et des séquences.
— Accès direct aux données à l'aide de "const &" pour éviter les copies inutiles.

— Limitation des messages d'avertissement redondants pour éviter l'encombrement de la sortie standard.

2.4.2 Le format FASTAQ

Le format FASTQ est une extension du format FASTA, conçu pour intégrer des informations de qualité associées à chaque base nucléotidique. Chaque entrée comprend quatre lignes : une ligne d'identifiant débutant par '@', la séquence nucléotidique elle-même, une ligne séparatrice commençant par '+', et une ligne contenant les scores de qualité encodés en ASCII. Ces scores reflètent la fiabilité de l'identification de chaque base, selon un encodage issu du modèle Phred [1]. Ce format est largement utilisé dans les technologies de séquençage à haut débit, notamment celles d'Illumina. Dans mon implémentation, j'ai utilisé le code de Phred33 encodé avec les caractères ASCII compris entre "!" et "~".

2.4.2.1 Implémentation

La structure de la classe ainsi que les méthodes utilisées sont résumées dans le tableau suivant :

TABLE 2.2 – Résumé des fonctions de la classe `FastqFileReader` et de leur rôle.

Fonction	**Rôle**
`bool loadFile()`	Charge complètement un fichier FASTQ en mémoire : stocke les headers, séquences et scores de qualité dans des vecteurs.
`bool processSequences(const std::function<void(...)>& callback)`	Parcourt les séquences sans les stocker (mode stream) et applique une fonction de rappel (callback) sur chaque trio (header, séquence, qualité).
`bool validate() const`	Valide la structure des séquence du fichier.
`const std::vector<std::string>& getSequences()`	Accesseur aux séquences.
`const std::vector<std::string>& getQualityScores()`	Accesseur aux scores de qualité.
`size_t countSequences() const`	Retourne le nombre de séquences chargées.
`bool isSequenceStart(const std::string& line) const`	Vérifie si une ligne marque le début d'une séquence commence par `@`.
`bool isQualitySeparator(const std::string& line) const`	Vérifie si une ligne est un séparateur de qualité valide commence par `+` et optionnellement suivi du entête.
`void parseQualityScores(std::ifstream& file, size_t seqLength)`	Méthode interne : extrait une ligne de qualité.

2.4.2.2 Cas traités dans l'implémentation

— La séquence de qualité est de même longueur que la séquence et est tronquée si elle est trop longue, avec un avertissement pour l'utilisateur.
— Si la séquence de qualité est trop petite, je lève une exception.

— Vérification de la séquence de qualité (de '!' à '~') et de la validité de la séquence moléculaire.
— Vérification des séparateurs de qualité, du caractère de début de séquence, ainsi que des en-têtes.
— Vérification de l'absence d'une séquence, qu'il s'agisse de la séquence de qualité ou de la séquence moléculaire.

2.4.3 Détecteur de format de fichier

La méthode de détection automatique du format de fichier, implémentée dans la classe `FormatFileDetector`, offre une solution efficace pour automatiser le processus de détection. Grâce à l'utilisation de l'énumération `Format`, elle permet de déterminer facilement le format du fichier (`FASTA`, `FASTQ` ou `UNKNOWN`) en analysant le chemin du fichier. Cette approche est particulièrement avantageuse si l'on souhaite étendre la classe à d'autres formats à l'avenir, en ajoutant simplement de nouvelles valeurs à l'énumération `Format` et en adaptant la méthode `detect` en conséquence. Le constructeur de la classe étant statique, il n'est pas nécessaire d'instancier la classe dans le `main`, ce qui rend le code plus clair et plus simple à utiliser.

```
class FormatFileDetector {
public:
    enum Format { FASTA, FASTQ, UNKNOWN };

    // detecter le format d'un fichier
    static Format detect(const std::string& filePath);

    // convertir le format enum en nom lisible
    static std::string formatToString(Format format);
};
```

Listing 2.1 – Détection du format de fichier

En termes de complexité, cette méthode présente une complexité temporelle de $\mathcal{O}(n)$, où n est la taille du fichier, car la détection repose sur la lecture du fichier ligne par ligne jusqu'à la détection d'un format valide. En termes de complexité spatiale, elle est également de $\mathcal{O}(1)$, car l'algorithme n'utilise qu'un espace constant supplémentaire pour les variables temporaires, indépendamment de la taille du fichier analysé. Cette solution est donc efficace, tant en termes de temps que de mémoire.

2.4.4 Architecture orientée objet et héritage

Le système de `parsers` que j'ai développé repose sur une hiérarchie de classes structurée (voir figure 2.2 et figure 2.3), avec `SequenceParser` comme classe de base abstraite, et `FastaParser` / `FastqFileReader` comme implémentations concrètes. Cette conception offre plusieurs avantages :

— Réutilisabilité du code : Les fonctionnalités communes, comme le stockage des séquences et des en-têtes, sont centralisées dans la classe parente.
— Polymorphisme : Les méthodes virtuelles pures (`loadFile()`, `validate()`, etc.) garantissent une interface cohérente.

— Extensibilité : Ajouter un nouveau format serait simple en créant une nouvelle classe pour un nouveau type de fichier.

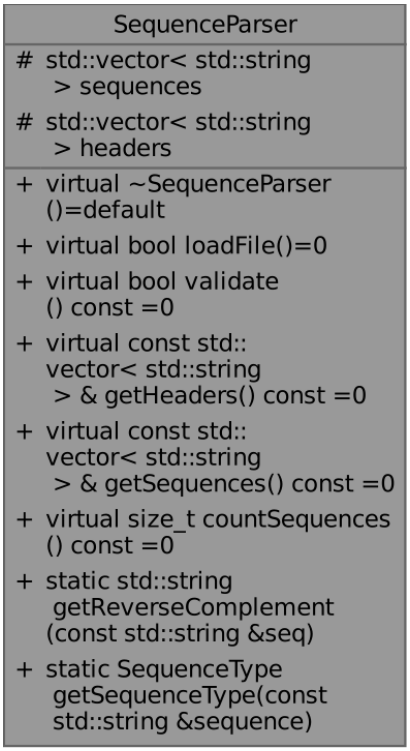

FIGURE 2.2 – Structure de la classe SequenceParser

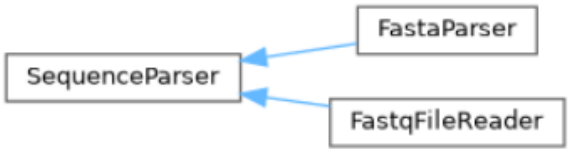

FIGURE 2.3 – Schéma de l'héritage des les classes des parseurs des fichiers.

2.4.4.1 Implémentation

La classe `SequenceParser` sert de base abstraite pour tous les parseurs de séquences biologiques, offrant des fonctionnalités essentielles et des outils communs pour le traitement des séquences ADN, ARN et protéiques, comme suit :

1. **Détection du Type de Séquence** (`getSequenceType`)
   — Cette méthode statique détermine automatiquement le type de séquence en analysant ses caractères constitutifs.

```
1        enum SequenceType { DNA, RNA, AA, UNKNOWN };
2        static SequenceType getSequenceType(const std::string&
             sequence);
```

   — Vérifie les caractères autorisés pour chaque type (ADN, ARN, acides aminés) inclut les codes d'ambiguïté IUPAC (R, Y, S, W, etc.).
   — Utilise `find_first_not_of` pour une détection efficace.
   — Retourne `UNKNOWN` si la séquence contient des caractères non reconnus.
   — Les jeux de caractères valides sont définis comme constantes.
   — Analyse en `O(n)` avec un arrêt au premier caractère invalide.
2. **Calcul du Complément Inverse** (`getReverseComplement`)
   — Méthode spécialisée pour les séquences nucléiques.

```
1         static std::string getReverseComplement(const std::
              string& sequence);
```

— Gère à la fois l'ADN (A-T) et l'ARN (A-U).
— Préserve la casse originale (majuscules/minuscules).
— Inclut les codes d'ambiguïté IUPAC (R, Y, S, W, etc.).
— Afin d'optimiser cette fonction, j'ai choisi l'utilisation d'une table statique, comme suit :

```
1         static constexpr auto buildComplementTable = []() {
2             std::array<char, 128> table{};
3             table['A'] = 'T'; table['T'] = 'A'; table['U'] = 'A
                  ';
4             table['C'] = 'G'; table['G'] = 'C';
5             // ... tous les codes d'ambiguite
6             return table;
7         };
8         static constexpr auto complementTable =
              buildComplementTable();
```

Listing 2.2 – Détection du Type de Séquence

— Cette table de lookup est générée à la compilation.
— Pré-allocation mémoire avec `reserve()`.
— Inversion en place avec `std::reverse`.

3. **Méthodes Virtuelles Pures**
— L'interface standardisée permet une utilisation polymorphique. Ainsi, ces fonctions seront implémentées dans les classes filles.

```
1         virtual bool loadFile() = 0;
2         virtual bool validate() const = 0;
3         virtual const std::vector<std::string>& getHeaders()
              const = 0;
4         virtual const std::vector<std::string>& getSequences()
              const = 0;
5         virtual size_t countSequences() const = 0;
```

2.4.4.2 Résumer de performance des fonctions

Ce tableau présente les complexités algorithmiques des principales fonctions de traitement de séquences biologiques. La colonne 'Temporelle' indique le temps d'exécution en fonction de la taille des données (n = nombre de séquences, m = longueur moyenne d'une séquence, $\mathcal{O}(1)$ = mémoire constante), tandis que la colonne 'Spatiale' montre la mémoire requise. Les justifications expliquent brièvement le comportement de chaque fonction.

TABLE 2.3 – Complexités des fonctions principales.

Fonction	Temporelle	Spatiale	Justification
loadFile	O(n)	O(n)	Lit et stocke toutes les séquences
processSequences	O(n)	O(1)	Traitement ligne par ligne sans tout stocker
validate	O(n*m)	O(1)	Vérifie chaque caractère des séquences
getReverseComplement	O(m)	O(m)	Crée une nouvelle séquence complémentaire

2.5 Étape 02 : Indexation de texte avec la table des suffixe

La classe `SuffixArray` permet de construire et d'exploiter un tableau de suffixes (SA) et un tableau des plus longs préfixes communs (LCP) pour une chaîne de caractères donnée. Elle est principalement utilisée pour rechercher efficacement un motif dans un texte et compter les occurrences ainsi que leur position. La figure ci-dessus montre le diagramme et l'organisation de la classe `SuffixArray`.

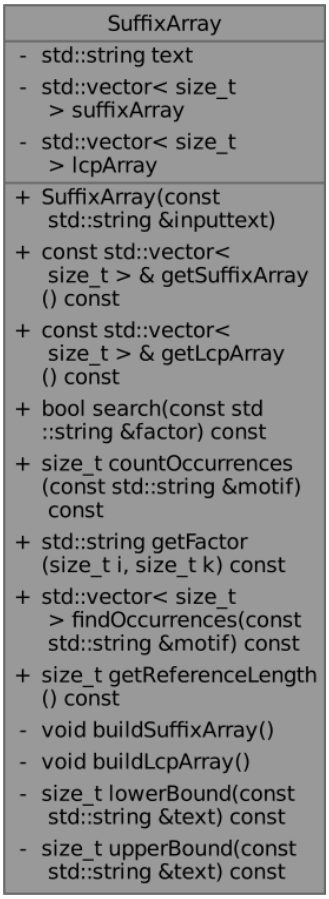

FIGURE 2.4 – Diagramme de la classe `SuffixArray`.

Table 2.4 : Tableau des performances et complexités

Opération	Complexité	Explication
Construction du SA	$O(n^2 \log n)$	Tri standard
Construction du LCP	$O(n)$	Parcours linéaire avec optimisation
Recherche (search)	$O(m \log n)$	Dichotomie + comparaison caractère par caractère
Comptage d'occurrences	$O(\log n)$	lowerBound/upperBound

2.5.1 Implémentation

— **Parallélisation** : Pour optimiser ma fonction, j'ai opté pour l'utilisation de `#pragma omp parallel for` afin de paralléliser le calcul du `rank` dans la construction du tableau `LCP`.

— **Optimisation mémoire** : Aucun stockage explicite de tous les suffixes ; seul un tableau d'indices est maintenu, évitant un coût mémoire en $O(n^2)$.

2.6 Étape 03 : Découpage des reads en k-mers et algorithme de recherche en batch

La classe `KmerIndex` permet d'indexer efficacement un read en le découpant en k-mers (sous-chaînes de longueur fixe k). Elle est utilisée pour mapper rapidement des reads sur un génome de référence. L'optimisation des recherches locales permet de réduire la complexité des opérations de matching. La classe est organisée comme le montre l'image ci-dessous :

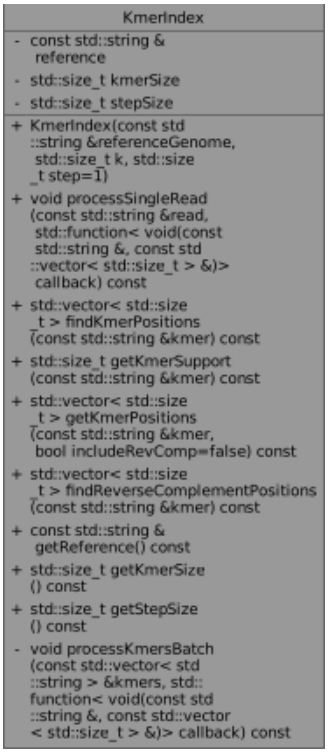

FIGURE 2.5 – Diagramme de la classe `kmerIndex`.

2.6.1 Implémentation

— Initialisation avec un génome de référence, une taille de k-mer (k) et un pas configurable (`step`) pour les k-mers chevauchants.
— Construction d'un *SuffixArray* pour le génome de référence.
— `processSingleRead()` : Découpe un read en k-mers avec un pas configurable.
— `processKmersBatch()` : Traite plusieurs k-mers en un seul passage pour optimiser les performances.
— Recherche des positions des k-mers dans le génome via le `SuffixArray` (`findKmerPositions`).

— Tri des k-mers pour regrouper les recherches identiques et minimiser les accès mémoire.
— Utilisation de la fonction `Callback` pour un traitement personnalisé des résultats.

2.6.2 Performance et complexité

Le tableau suivant résume la complexité temporelle des fonctions les plus pertinentes dans la classe, en tenant compte de la complexité spatiale. Ces fonctions sont optimisées au maximum et dépendent uniquement de la table `SuffixArray`, dont la taille est proportionnelle à celle du génome.

TABLE 2.4 – Tableau des complexités des fonctions

Opération	**Complexité**	**Description**
Construction de l'index	$O(n^2 \log n)$	Due au SuffixArray.
Recherche d'un k-mer	$O(m \log n)$	m = taille du k-mer, n = taille du génome.
Traitement par batch	$O(b \log b)$	b = nombre de k-mers (tri préalable).

2.7 Étape 04 : Étape de mapping sur un génome de référence

La classe `ReadMapper` permet de mapper des séquences (reads) sur un génome de référence en utilisant un index de k-mers. Elle combine la recherche locale via `KmerIndex` et l'évaluation de la qualité (score de confiance, distance d'édition), puis la détection automatique du brin (forward/reverse). Cette classe est structurée comme suit :

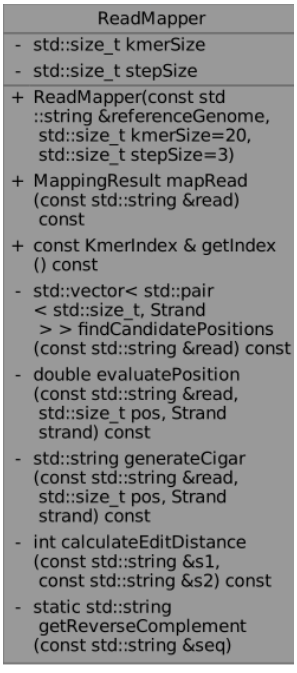

FIGURE 2.6 – Diagramme de la classe `ReadMapper`.

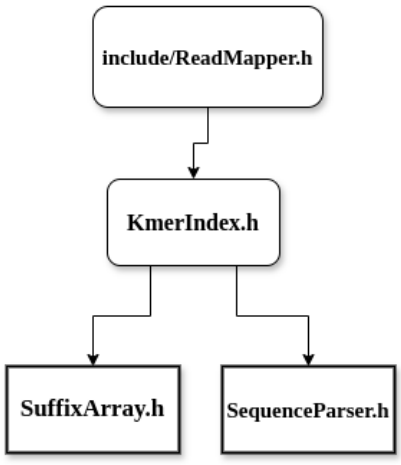

FIGURE 2.7 – Les classes incluses et appelées par `ReadMapper`.

2.7.1 Implémentation

Cette classe est conçue de manière simple, avec quelques astuces d'optimisation pour cette tâche de mapping. Les cas traités dans la recherche d'un read dans un génome de référence sont les suivants :

- Les reads les plus courts que k sont ignorés (retourne un `MappingResult` vide).
- Le cas des multiples positions candidates est traité par la sélection de la meilleure en fonction du score de confiance.
- La recherche se fait dans le brin direct et complémentaire.
- Les collisions de k-mers sont résolues par comptage des occurrences.
- Les alignements partiels sont basés sur un score de confiance qui dépend lui-même des k-mers matchés.
- Gestion des bords par vérification des débordements (`pos + read.length()`).

2.7.2 Performances et optimisations

Pour optimiser les performances, les `k-mers` sont triés afin de regrouper les recherches identiques, ce qui permet de réduire les accès mémoire. Les *compléments inverses* ne sont pas pré-calculés, mais générés dynamiquement plutôt que pré-calculés et stockés, grâce à la fonction `getReverseComplement`. Un `score de confiance`, normalisé entre 0 et 1, permet de comparer facilement les différents candidats. Enfin, une `terminaison anticipée` (early termination) est appliquée pour ignorer directement les reads trop courts, évitant ainsi un traitement inutile. Les complexités spatiale et temporelle sont résumées dans le tableau suivant :

TABLE 2.5 – Complexités temporelles et spatiales des principales fonctions.

Opération	**Temporelle**	**Spatiale**
`mapRead`(mappage complet)	$O(m \cdot k \cdot n)$	$O(m + n)$
`findCandidatePositions`	$O(m \cdot k)$	$O(p)$
`evaluatePosition`	$O(m)$	$O(1)$

Légende : m = longueur du read, k = taille des k-mers, n = taille du génome, p = nombre de positions candidates.

3. Évaluation et validation de l'implémentation

À présent, ce code conçu durant ce projet tient compte de plusieurs cas de figure suivants :

- Séparation claire des responsabilités entre les classes (`ReadMapper`, `KmerIndex`, `SuffixArray`, ...).
- Utilisation pertinente de l'héritage (`SequenceParser` comme classe de base pour `FastaParser` et `FastqFileReader`).
- `SuffixArray` pour une recherche rapide de `k-mers` : $O(\log n)$ par motif.
- Calcul du reverse-complément à la volée, évitant un stockage redondant.
- Tri des *k-mers* pour regrouper les recherches identiques et limiter les accès mémoire.
- Validation des fichiers FASTA/FASTQ (vérification des en-têtes, séquences, scores de qualité).
- Gestion des exceptions avec des messages explicites pour les erreurs de format ou de paramétrage.
- Détection automatique du format des fichiers (FASTA ou FASTQ).
- Stratification des résultats selon :
  - le brin (forward / reverse),
  - le score de confiance,
  - la distance d'édition et le CIGAR simplifié.
- Fonction `explainCIGAR()` intégrée pour expliquer les alignements.
- Paramètres configurables (taille de k-mer, pas) via la ligne de commande.

Ci-dessous un exemple de résultats d'exécution sur un test unitaire :

```
=== Résultat pour @Read1:ExactMatchForward ===
Position: 0 | Brin: Forward
Confidence: 100.00%
Distance d'édition: 0
CIGAR: 20M
Mapping unique: Oui

=== Résultat pour @Read2:ExactMatchReverse ===
Position: 1 | Brin: Reverse
Confidence: 100.00%
Distance d'édition: 12
CIGAR: 20M
Mapping unique: Oui

=== Résultat pour @Read3:MultiPosKmers ===
Position: 54 | Brin: Forward
Confidence: 100.00%
Distance d'édition: 0
CIGAR: 20M
Mapping unique: Non

=== Résultat pour @Read4:MiddleMutation ===
Position: 55 | Brin: Reverse
Confidence: 30.77%
Distance d'édition: 16
CIGAR: 19M
Mapping unique: Non

=== Résultat pour @Read5:EndMutation ===
Position: 0 | Brin: Forward
Confidence: 85.71%
Distance d'édition: 2
CIGAR: 20M
Mapping unique: Oui
```

FIGURE 3.1 – Exemple d'exécution sur une petite séquence de test avec k=8 et un pas=2

NOTE : Les paramètres k et le pas de chevauchement des k-mers peuvent être modifiés selon l'objectif biologique.

3.1 Limites et points faibles de l'implémentation

— Construction du `SuffixArray` en $O(n^2 \log n)$: coûteux pour les grands génomes de taille 10^9 GO.
— Pas de méthode probabiliste pour filtrer les k-mers inexistants.
— Recherche brute parmi les candidats : absence d'alignement local/global (type Needleman-Wunsch ou Smith-Waterman).
— Gestion naïve des répétitions : risque accru de faux positifs pour les k-mers non uniques.
— Stockage complet des reads en mémoire : problématique avec des fichiers $>$ 1 Go.
— Pas de parallélisation du mapping (sauf pour le LCP) : le processus reste essentiellement séquentiel.
— L'absence de gestion adéquate de la taille des k-mers par rapport à celle des reads peut conduire à des résultats incohérents.
— L'absence de fonction de nettoyage pour les fichiers FASTA et FASTQ, qui se contentent de lever des exceptions dès la première erreur rencontrée, sauf dans le cas des espaces dans une séquence FASTA.

Conclusion générale et Perspectives

Ce projet a constitué une immersion concrète dans le développement d'outils bioinformatiques à haute performance, allant de la lecture de fichiers FASTA/FASTQ à l'implémentation d'algorithmes de mapping optimisés. Il m'a permis de mettre en pratique les principes de la programmation orientée objet en C++, tout en intégrant une architecture modulaire, évolutive et maintenable. Chaque composant – du parser aux index de suffixes et k-mers, jusqu'au module de mapping – a été conçu avec une séparation claire des responsabilités, permettant une grande réutilisabilité du code.Ce travail reflète une compréhension des besoins spécifiques de la bioinformatique moderne, notamment en termes de gestion mémoire, de performance algorithmique et de traitement massif de données.

Parmi les pistes d'optimisation envisagées, l'encodage des nucléotides sur 4 bits (1/2 octet) représente une solution innovante et prometteuse. Cet encodage binaire permettrait une réduction significative de l'espace mémoire tout en facilitant les comparaisons via des opérations bit-à-bit. Couplée à d'autres approches comme la transformation de Burrows-Wheeler (BWT), cette optimisation pourrait faire évoluer le système vers une architecture encore plus performante et adaptée au traitement des données omiques modernes.

Ce projet illustre aussi l'importance des bonnes pratiques DevOps dans la production de code scientifique : gestion de version avec Git, documentation automatique avec Doxygen, modularité et tests unitaires. À travers ce travail, j'ai pu consolider des compétences techniques clés (indexation, parsing, complexité, optimisation) tout en développant une approche du développement logiciel en bioinformatique.

Enfin, cette UE m'a permis de mieux appréhender les enjeux de performance, de modularité et de scalabilité dans les systèmes bioinformatiques, ouvrant la voie vers des applications concrètes en recherche génomique, en analyse transcriptomique ou en médecine personnalisée.

References

[1] Alban MANCHERON. support de cours et travaux pratiques.