Master Bioinformatique

HAU803I
Développement opérationnel avancé :
application aux gros volumes de données

Rapport de projet : mapping de données issues de Nouvelles Générations de Séquenceurs (NGS) sur un génome de référence.

Auteur :
Florent Marchal 22008079

Équipe pédagogique :
Alban Mancheron

[Version du 16 avril 2024|Version finale]

1 Introduction

Dans le cadre du Master de bio-informatique de Montpellier, les étudiants ont suivi un module ayant pour but de les initier au développement opérationnel (HAU803I). Pour cela, il leur a été demandé de réaliser un projet visant à créer un mapper pour des séquences génomiques courtes tout en faisant attention aux complexités en temps et en espace. Ce module se conclut par un oral et un rapport (que vous êtes en train de lire) visant à évaluer, non pas la qualité du code mais le degré de compréhension des concepts abordés tout au long de ce module.

Connaissant les règles d'évaluation et sachant que ce genre d'API existe déjà, je n'ai pas concentré mes efforts sur l'accomplissement du projet mais sur l'apprentissage du C++ et des sujets abordés en cours. J'expliquerai les stratégies que j'ai / j'aurai mis en place pour réaliser ce projet. Vous trouverez le git du projet ici.

Table des matières

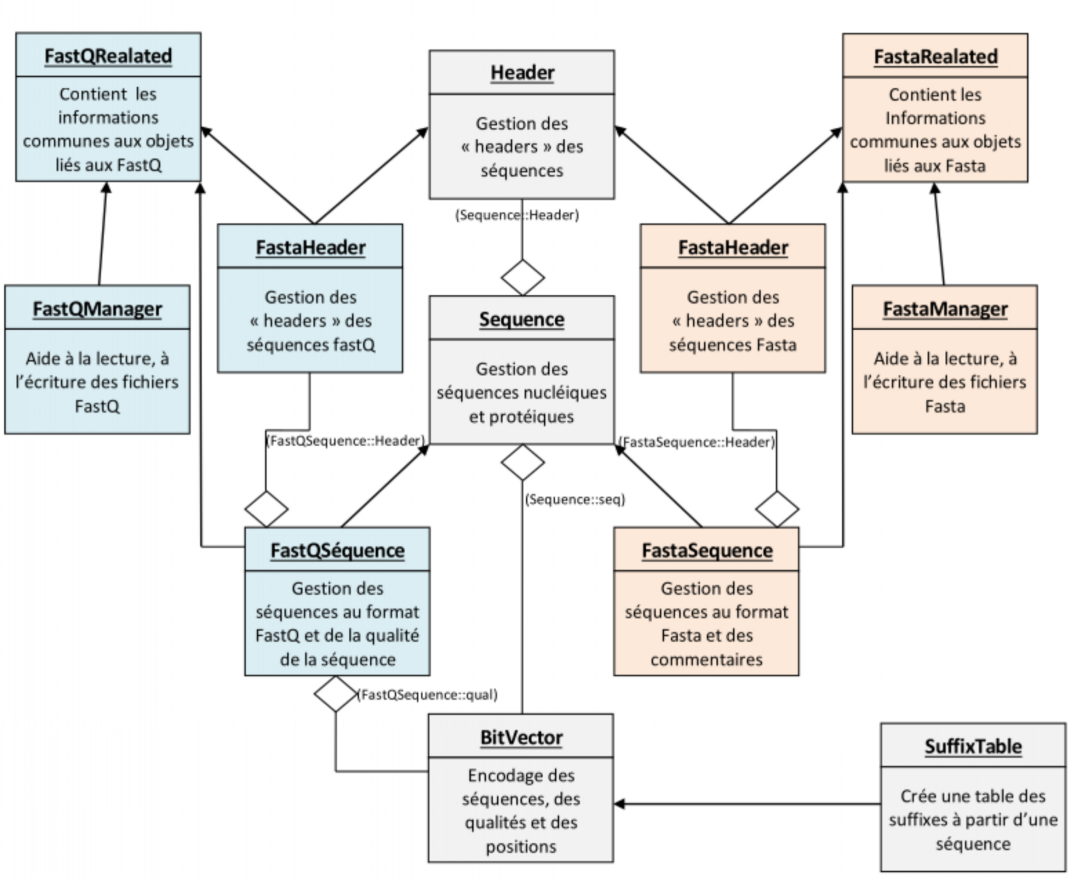

FIGURE 1 – Diagramme pseudo-UML présentant les classes du projet

2 Réalisation du projet

Pour réaliser ce projet, j'ai choisi de profiter de la partie « objet » du *c++* pour segmenter au mieux mon projet (cf 1).

Je présenterai ici les différentes classes en expliquant pour chacune leur conception et les problèmes rencontrés.

2.1 BitVector

2.1.1 Utilité dans le projet

Cette classe est probablement la classe la plus importante du projet d'un point de vue réduction de l'usage de la mémoire vive. Celle-ci permet d'enregistrer dans un tableau de caractères une suite de bits. De ce fait, cette classe permet de « ranger » efficacement les symboles contenus dans les séquences (2.3).

2.1.2 Objectifs lors de la construction

Dans le cadre de ce projet, j'aurais pu me contenter d'une classe capable de ne ranger que des éléments de taille « 2 ». Cependant, j'ai opté pour une classe capable de ranger des éléments de taille n. En effet, il me paraît plus pertinent de créer une classe capable de stocker des éléments de toutes tailles. De ce fait, il me sera possible de la réutiliser dans d'autres situations. Ce choix est aussi motivé par mon envie de pouvoir stocker efficacement

à la fois l'alphabet réduit des acides nucléiques (4 éléments → 2 bits) et l'alphabet complet des acides aminés (28 éléments → 5 bits).

2.1.3 Description de l'objet

La classe $BitVector$ est conçue pour contenir des éléments de taille n bits. Ces éléments sont stockés dans un tableau de caractères. Les éléments sont accessibles grâce à leur index dans le vecteur tandis que les bits sont accessibles par leur position dans le tableau en utilisant la classe interne $BitVector :: Coords$. Celle-ci représente la position des bits en utilisant le numéro d'octet ($size\_t$) et le numéro de bit ($unsignedshortint$ compris entre 0 et 7).

La classe interne $BitVector :: Coords$ sert aussi de base au $BitVector :: Iterator$. Ce dernier permet d'itérer dans le $BitVector$ tout en lui permettant de supporter la syntaxe suivante :

$$for(char * unObjet : unBitVector)$$

2.1.4 Problèmes rencontrés

Position des éléments dans le vecteur Par défaut, $BitVector$ utilise les coordonnées ($BitVector :: Coords$) pour interagir avec le tableau de caractères qui contient les éléments. Il est donc nécessaire de trouver un moyen de convertir les coordonnées des bits en position d'éléments. Pour cela, plusieurs techniques ont été envisagées :

Utilisation d'une boucle *while* permettant de compter le nombre d'éléments pouvant être contenus entre la coordonnée $(0, 0)$ et une coordonnée donnée C. Cette technique a l'avantage de ne pas limiter le nombre d'éléments rangeables dans le $BitVector$. Cependant, sa complexité en temps est en $O(n)$ où n est le nombre de bits de C, ce qui est affreusement lent.

Utilisation des opérateurs arithmétiques en multipliant le numéro d'octet et le nombre d'éléments pouvant être présents dans un seul octet. Cette technique est peu complexe en temps ($O(1)$). En revanche, elle est fortement limitée par la précision des *double* .

Création d'un $size\_t$ permettant de stocker à la fois la valeur de l'octet et la valeur des bits :

- les 3 premiers digits sauvegardent la position du bit (position dans le caractère) : $bit/8 * 1000$
- les autres digits sauvegardent la position de l'octet (position dans le tableau)
- au total :

$$Valeur\_du\_Size\_t = position\_dans\_le\_tableau * 1000 + \frac{position\_dans\_le\_car * 1000}{8}$$

Cette technique rend les calculs simples et peu complexes ($O(1)$) au prix de la réduction du nombre d'octets utilisables par trois puissances de dix. C'est cette dernière méthode qui a été choisie (cf $BitVector :: Coords :: tosize\_t()$).

Opérations Mathématiques sur $size\_t$ Lors de l'ajout d'un élément, la limite de place du $BitVector$ peut être dépassée. J'ai donc ajouté des garde-fous dans mes fonctions pour éviter cela. ($BitVector :: maxElementLimit()$, $BitVector :: maxOctetLimit()$).

Aussi, en $c++$, les $size\_t$ et autres $unsigned\ int$ ont une fâcheuse tendance : ils retournent à une valeur antérieure lorsqu'une addition ou une multiplication dépasse leur valeur maximale. J'ai donc essayé de prendre cela en compte ($BitVector :: Coords :: canBeAddedBy$, $BitVector :: Coords :: canBeSubtracted$, $ByBitVector :: Coords :: canMultiplyBy$).

2.1.5 Ce que j'ai appris

Voici une liste non exhaustive des éléments que les classes $BitVector$, $BitVector :: Coords$ et $BitVector :: Iterator$ m'ont permis de comprendre et d'apprendre.

- $BitVector$
  - manipulation des bits
  - manipulation des pointeurs
  - utilisation des tableaux de caractères
  - création de classes internes
  - manipulation des $size\_t$ sur de grands nombres
  - manipulation des Variadics
- $BitVector :: Coords$
  - utilisation des méthodes *operator* (tests logiques, conversions...)
  - manipulation des $size\_t$ sur de grands nombres
- $BitVector :: Iterator$
  - conception d'une classe permettant d'itérer dans un autre objet
  - fonctionnement « caché » des boucles for ayant la syntaxe $for(char * unObjet : unBitVector)$

2.2 Header

La classe $Header$ est très simple. Elle sert de base aux classes $FastaHeader$ et $FastQHeader$. Ces deux classes représentent respectivement les « headers » liées aux séquences Fasta et aux séquences FastQ. Elles sont censées pouvoir être initialisées à partir d'un texte représentant un « header » afin d'en extraire le format, le numéro d'accession et les éventuels commentaires associés.

Je n'ai pas terminé de travailler sur les classes $Header$ mais voici comment je souhaite m'y prendre dans la classe $FastaHeader$:

1. découpe du texte au niveau des « | » et potentiellement au niveau des espaces
2. extraction du format (ex : gb, bbs, gim...), il s'agit normalement la première partie du header.
3. passage du texte dans une fonction de « parsing » correspondant au format pour extraire les autres informations. Cela implique de créer soit un dictionnaire associant un format avec une fonction, soit une grande liste de if / $else$ associant le format et la fonction.

Dans l'hypothèse où le texte ne contiendrait pas de « | », on considèrera que le premier « mot » correspond au numéro d'accession.

Ces classes sont censés être associées avec des séquences de même type.

2.3 Sequences

2.3.1 Utilité dans le projet

La classe *Sequence* est l'objet censé pouvoir représenter les séquences biologiques. De fait, elle sera utilisée par le mapper (2.6) et la table des suffixes (2.5).

Sa classe interne $Sequence :: SequenceSymbol$ sert à représenter les caractères que la séquence peut contenir. Elle permet aussi de choisir un caractère de remplacement en fonction de l'alphabet choisi (iupac ou basique) (2.3.3)

2.3.2 Objectifs lors de la construction

La classe *Sequence* est conçue pour pouvoir contenir des séquences biologiques protéiques ou nucléiques (ADN ou ARN). Elle peut contenir tous les caractères de l'alphabet étendu si nécessaire mais peut aussi se limiter à l'alphabet basique (A, T/U, C, G). Bien sûr elle se sert de $BitVector$ pour limiter la consommation de mémoire. Afin de faciliter l'analyse des séquences, elle est capable d'identifier au fur et à mesure le type qu'elle représente (protéine, ADN, ARN ou ARN + ADN).

2.3.3 Fonctionnement

Définition de l'ensemble des caractères valides J'ai choisi de créer une $std :: map$ par jeu de caractères (ADN, ARN, acides aminés). Chaque dictionnaire contient l'ensemble des caractères légaux pour un type donné couplé à un $Sequence :: SequenceSymbol$.

Dans l'idéal, j'aurais aimé utiliser un $std :: set < Sequence :: SequenceSymbol >$. Cependant j'avais besoin de pouvoir rechercher dans l'ensemble des *char* en plus des $Sequence :: SequenceSymbol$. Je n'ai pas réussi à implémenter cela. Je me suis donc rabattu sur un $std :: map < char, Sequence :: SequenceSymbol >$. En effet, il est plus simple de convertir $Sequence :: SequenceSymbol$ grâce à l'$operatorchar()$.

Note : il aurait peut-être été possible d'utiliser une fonction $friend$ pour pouvoir convertir les *char* en $Sequence :: SequenceSymbol$ et ainsi pouvoir chercher un *char* dans un $std :: set < Sequence :: SequenceSymbol >$. C'est une option qu'il serait intéressant de tester ultérieurement.

Auto-détermination du type L'auto-détermination du type se fait en utilisant un tableau de booléens représentant chacun un aspect possible de la séquence. Lorsque la recherche de types est activée, chaque caractère est « analysé » pour identifier son groupe d'origine et modifier en conséquence le type de la séquence. Le plus complexe est de réussir à différencier les séquences ADN des séquences ARN.

Encodage des caractères Pour l'encodage des caractère dans un $BitVector$, deux solutions ont été envisagées :

1. le définir en dur dans le code (Dans les $Sequence :: SequenceSymbol$)
2. le créer à chaque exécution en fonction de l'alphabet considéré

J'ai opté pour la deuxième solution car c'est, de loin, la solution la plus intéressante d'un point de vue apprentissage du *c++*. L'encodage est donc créé à chaque exécution en fonction des trois alphabets (ADN, ARN, acides aminés) et des trois modes d'alphabet (2.3.3). Les tables de conversions ne sont pas générées à chaque usage, elles sont stockées comme des variables *static* dans la fonction $Sequence :: translationTab$.

Alphabet iupac Les séquences peuvent choisir de supporter :
- l'alphabet basique : 4 acides nucléiques ou les 20 acides aminés principaux
- l'alphabet quasi complet : tout l'alphabet iupac à l'exception des « gap » et des acides aminés / nucléiques rares
- l'alphabet complet : tout l'alphabet iupac

Lorsque l'alphabet iupac choisi ne supporte pas un symbole, il est replacé aléatoirement par un symbole de substitution en fonction de sa signification. L'alphabet choisi à une grande influence sur la mémoire consommée par la séquence. (cf 2.6.3)

2.3.4 *FastQSequence* et *FastaSequence*

Deux classes sont dérivées de *Sequence* : *FastQSequence* et *FastaSequence*.

FastQSequence est une *Sequence* possédant un second *BitVector* permettant de stocker les qualités de chaque symbole de la séquence. Les fonctions d'insertion doivent être modifiées pour accepter à la fois les nouveaux caractères et leur qualité.

FastQSequence est une *Sequence* possédant un *std* :: *map* permettant de stocker les commentaires liés aux différentes positions de la séquence. Les fonctions d'insertion doivent être modifiées pour déplacer les commentaires en fonction du nombre de symboles insérés.

2.3.5 Ce que j'ai appris

- tests sur héritage (mot clef *virual*),
- opérateurs de conversion de type (*operator char*(), *operator string*(), *operator bool*())
- usage du mot clef *static* (classe et fonction)
- usage des opérateurs de conversion (classe et fonction)
- utilisation des *std* :: *maps*
- utilisation des *std* :: *set*
- utilisation des strings

2.4 Managers

FastaManager et *FastQManager* sont deux classes « outils » ne servant qu'à faciliter la lecture et l'écriture des *FastaSequence* et des *FastQSequence* depuis des fichiers ou des textes. Elles ont vocation à segmenter la lecture des fichiers en trois parties :
- identification de la partie « header » et création d'un *Header*
- identification de la partie « séquence » et création d'un objet *Sequence*
- identification de la partie « qualité » et modification de la partie associée de l'objet *Sequence*
- attachement des objets *Sequence* avec leurs headers

A noter : les « Manager » ne s'occupent pas directement du « parsing », ce sont les classes *Sequence* et *Header* qui s'en occupent pour respecter l'encapsulation des classes.

Les classes « Manager » n'ont pas encore été implémentées.

2.5 Table des suffixes

2.5.1 Utilité dans le projet

Cette classe permet de grandement limiter la complexité en temps de la recherche de motifs à travers une séquence. Celle-ci « range » par ordre alphabétique l'ensemble des

suffixes dans un vecteur de taille $sequence.size() + 1$. Les suffixes sont représentés par leur index dans la séquence. Cette structure permet d'effectuer des recherches par dichotomie ($O(log(n))$ au lieu de $O(n)$).

2.5.2 Usage de la mémoire

Le principal problème de la table de suffixes est la place en mémoire d'une telle structure. Compte tenu du fait que la taille d'une *Sequence* est représentée par un $size\_t$ (8 octets sur un système 64 bits), la table des suffixes du génome humain ($\approx 10^9 bp$) consommerait environ $10^9 * 8$ octets soit environ 8 Go. Pour limiter ce problème je pense représenter la table des suffixes avec un $BitVector$ dont la taille des éléments serait $E(log_2(Sequence.size())) + 1$ bits. Cela rendrait la consommation de mémoire proportionnelle à la taille de la séquence. Dans le cas du génome humain, l'usage de mémoire passerait de 64 bits par élément à $E(log_2(10^9)) + 1 = 30$ bits par élément ce qui diminuerait l'usage de la mémoire à $\frac{30}{8} * 10^9 coctet = 3,75 Go$ soit 47 % de la mémoire occupée originellement. Plus d'exemples de la taille mémoire de la table des suffixes ici : (2.6.3)

A noter : la table des suffixes a besoin que la séquence qu'elle représente reste en mémoire. Si l'on utilise notre classe *Sequence*, le génome humain peut peser entre $\frac{2}{8} * 10^9$ $octet = 0,25 Go$ et $\frac{5}{8} * 10^9 octet = 0,625$ Go pour un total de mémoire vive utilisée compris entre 4 Go et 4,375 Go.

2.5.3 Compression

Si la table des suffixes doit être conservée mais que de la mémoire doit être libérée, il est possible de construire une transformée de Burrows-Wheeler qui permet de combiner la séquence et la table des suffixes en un seul élément. L'usage mémoire de la transformée correspond à celui de la *Sequence* initiale.

2.6 Mapper

2.6.1 Création

La partie mapper est relativement simple à construire à partir des éléments précédents. Il suffit de créer la table des suffixes du génome que l'on souhaite étudier et d'y chercher par dichotomie le motif que l'on souhaite identifier.

2.6.2 Recherche avec l'alphabet étendu

Les recherches utilisant l'alphabet étendu sont faciles à mettre en place. En couplant $Sequence :: getSymbolDNA$ et $Sequence :: SequenceSymbol :: correspondTo$ lors des comparaisons de caractères, on peut déterminer si les deux caractères se correspondent.

2.6.3 Consommation globale de mémoire

Dans ce projet, la consommation totale de mémoire est très majoritairement liée à la table des suffixes. La consommation totale de mémoire en fonction des différents paramètres est : $Taille\_en\_mmoire\_de\_la\_squence$ (2.6.3) + $Taille\_en\_mmoire\_de\_la\_table\_des\_suffixe$ (2.6.3)

Especce	Taille du génome (Mpb)	Consommation de mémoire de la séquence en fonction de l'alphabet 2.3.3 (Go)								
		Basique			Semi complet			Complet		
		ADN	ARN	AA	ADN	ARN	AA	ADN	ARN	AA
Polychaos dubium (amibe)	675 000	169	169	422	338	338	422	422	422	422
Zea mays (maïs)	5 000	1,25	1,25	3,13	2,50	2,50	3,13	3,13	3,13	3,13
Hommo sapiens (Homme)	3 4000	0,85	0,85	2,13	1,70	1,70	2,13	2,13	2,13	2,13
Hommo sapiens (exemple 2.5)	1 000	0,25	0,25	0,63	0,50	0,50	0,63	0,63	0,63	0,63
Mus musculus (souris)	119	0,03	0,03	0,07	0,06	0,06	0,07	0,07	0,07	0,07
Nombre de bits par symbole		2	2	5	4	4	5	5	5	5

TABLE 1 – Exemple de Consommation d'espace mémoire par les séquences. ADN=ADN, ARN=ARN, AA=Acides aminés (A titre indicatif, les protéines ne sont pas aussi grande). Source taille des genomes : Wikipedia.org - Taille du génome

Especce	Taille du génome (Mpb)	Nombre de bit par position	Consommation totale (Go)
Polychaos dubium (amibe)	675 000	40	3 375
Zea mays (maïs)	5 000	33	20,63
Hommo sapiens (Homme)	3 4000	32	13,60
Hommo sapiens (exemple 2.5)	1 000	30	3,75
Mus musculus (souris)	119	27	0,40

TABLE 2 – Exemple de Consommation d'espace mémoire par les tables des suffixes. Source taille des genomes : Wikipedia.org - Taille du génome

3 Réalisation des objectifs du cours

3.1 Développement (c++)

L'apprentissage du *c++* a été réalisé tout au long du semestre en essayant de mener à bien le projet. J'ai essayé de toucher à autant d'aspects que possible de ce langage comme montré dans les parties 2.3 et 2.1 et comme visible sur le git du projet.

3.2 Automatisation (Makefile)

Je me suis servi d'un $MakeFile$ pour faciliter la compilation et l'exécution de mon programme et pour lancer *doxygen*. Celui-ci se trouve sur le git du projet.

3.3 Documentation (Docxygen)

Mon apprentissage de *doxygen* est passé par une phase de découverte dans le *wizard* de l'application (*doxywizard*) suivie par la découverte du *Doxyfile* et une étude plus approfondie des paramètres utilisables. Une partie de la documentation de ce projet a été générée par *Doxygen*. (cf git)

3.4 Déploiement et tests

Dans le futur, j'aimerais construire un fichier $.gitlab-ci.yml$ pour créer ma « pipeline » de « tests » et de « déploiement ». Des tests ont été réalisés tout au long du développement pour exécuter les fonctions principales de chaque classe dans différentes situations. Malheureusement, je n'ai pas conservé les tests.

3.5 gestion de versions/maintenance (Git)

Le git du projet est hébergé ici. J'ai essayé de manipuler, autant que possible, les branches et les merges.

4 Conclusion

Ce projet, très formateur, m'a permis de découvrir de très nombreux aspects du *c++* (2.3.5, 2.1.5,git), du logiciel *Git* et du développement opérationnel en général. Je pense le prolonger à l'avenir par des tests plus approfondis sur les différentes fonctions déjà réalisées dans le but de créer une « pipeline » de « tests » 3.4.