Université de Mons
Faculté des sciences
Département d'Informatique

Algorithmique et bioinformatique
Rapport de projet

Professeurs :
Olivier Delgrange
Clément Tamines

Auteurs :
Laurent Bossart
Guillaume Proot

Année académique 2021-2022

Table des matières

1 Introduction

Dans le cadre du cours de bioinformatique, il nous été proposé comme projet d'implémenter un programme permettant de reconstituer une séquence ADN à partir d'un ensemble de fragments. Ce programme se divise en plusieurs parties et celles-ci seront expliquées tout le long du rapport.

Tout d'abord, il nous faut récupérer un ensemble de fragments d'ADN contenu dans un fichier. Dans notre cas, la classe *FragmentAssembler* s'en occupe en chargeant un fichier *.fasta* et en instanciant un objet *Collection* contenant l'ensemble des fragments d'ADN.

Ensuite, on construit un graphe à partir de l'ensemble de ces fragments. Ce graphe est caractérisé par un ensemble de noeuds représentant chacun des fragments et un ensemble d'arcs représentant la similarité entre ces fragments. Ce graphe fait intervenir la classe *Graph*.

Une fois le graphe obtenu, nous avons appliqué un algorithme afin de calculer le chemin hamiltonien de ce graphe à l'aide d'une heuristique de type *Greedy* (c'est à dire une approche gloutonne). Pour cette recherche de chemin, tout se déroule dans la classe *Graph*. Il en résulte une liste de fragments classés suivant le chemin hamiltonien.

Une fois que le chemin est obtenu, il a fallut réaligner les fragments correctement tout en suivant le chemin. Pour celà, l'approche semi-globale a été utilisée pour aligner les fragments deux par deux. Tout se déroule dans la classe *Graph*.

Les fragments alignés, il nous a fallu ensuite propager les gaps et les décalages aux différents fragments. Cela ce fait dans la classe *FragmentAssembler*.

Pour finir, nous avons appliqué un consensus qui se traduit par un vote à la majorité afin de pouvoir recréer notre séquence finale qui est enregistrée dans un fichier *.fasta* .

2 Récupération des fragments

La récupération des fragments consiste en une série d'étapes :
- Identifier et ouvrir le fichier *fasta*
- Identifier le début d'un fragment (un fragment pouvant être représenté sur plusieurs lignes du fichier)
- Récupérer le contenu du fragment

L'identification et l'ouverture du fichier *fasta* se fait de manière triviale, il n'a donc pas été jugé nécessaire de s'y attarder.

Une expression régulière a été nécessaire afin d'identifier le commencement d'un fragment. Cette expression régulière vérifie la présence du mot "fragment" dans le chaîne actuellement vérifiée. Si "fragment" est présent, alors il s'agit du point de départ d'un fragment.

Pour la récupération du contenu du fragment, il a fallut s'assurer que nous n'étions pas en présence d'un nouveau fragment (i.e. la ligne actuelle du fichier ne contient pas le mot "fragment"). Pour se faire, on vérifie si chaque ligne est la dernière ligne du fichier (i.e. on vérifie la propriété "hasNextLine()"), et si c'est effectivement le cas, on vérifie si l'on est en présence d'un nouveau fragment. Si ce n'est pas le cas, la partie du fragment lue est ajoutée à un *StringBuilder*. Si nous somme en présence d'un nouveau fragment, la chaîne de caractères nouvellement formée est ajoutée à un *ArrayList* qui contient tous les fragments extraits du fichier *fasta*.

3 Fonctionnement de l'algorithme *greedy*

Lors de la présentation du projet, il nous a été proposé d'utiliser une heuristique *greedy* afin de trouver le chemin hamiltonien dans notre graphe. En effet, nous avons vu dans nos cours de bachelier que l'algorithme du calcul exact d'un chemin hamiltonien est un problème dit *NP-complet*. C'est pourquoi, au vu de la taille de notre problème à résoudre, un algorithme d'approximation s'avère nécessaire. Le principe de cet algorithme est d'ajouter chaque paire de noeuds selon le poids de leur arc (poids triés dans l'ordre décroissant) jusqu'à ce que le chemin contienne tous les noeuds possibles (c'est à dire si un chemin contient un certain noeud il ne peut pas contenir son complémentaire-inversé).

Dans notre code, cet algorithme se trouve dans la classe *Graph* et se base sur l'ensemble de noeuds créé précédemment, et va nous ressortir une liste de fragments qui composent le chemin hamiltonien.

3.1 Déroulement de l'algorithme

Dans un premier temps, un graphe comprenant l'entièreté des fragments, ainsi que leurs complémentaires-inversés, représentants les noeuds est créé afin de pouvoir calculer le chemin hamiltonien. Ensuite, on crée deux tableau de *boolean*, *in* et *out*, de la taille du nombre de noeuds dont chaue entrée contient respectivement *false* ou *true* à l'indice i, si l'on est déjà entré dans le noeud i et déjà ressorti du noeud i. Nous créons aussi une liste *listOfSets* qui va contenir les ensembles résultants des *makeset*. Les *makeset(i)* sont effectués sur le i^{eme} noeud de la liste des noeuds, et consiste à créer une liste ne contenant que ce noeud.

Une fois cela fait, les arcs entre les noeuds sont créés et triés par ordres décroissants de poids (voir sous-section 3.3). Une fois les arcs triés, on applique l'algorithme *Greedy* afin de trouver le chemin hamiltonien. Une liste de fragments ordonnés représentant le chemin hamiltonien est retourné à la fin de l'algorithme.

3.2 Création de la liste des noeuds

Comme dit précédemment, les noeuds du graphe sont les fragments récupérés dans l'étape précédente. Il ne faut pas non plus oublier de considérer

les complémentaires-inversés de chaque fragment. C'est pourquoi pour k fragments, il y a $n = k * 2$ noeuds dans le graphe.

3.3 Tri des arcs

Un nombre de *threads* a été déterminé ainsi qu'une liste de taille fixe d'arcs. Connaissant le nombre de noeud contenu dans le graphe, il est trivial de calculer le nombre d'arc présent dans le graphe à l'aide de la formule : $arc = noeud * (noeud - 2)$. Les *threads* sont ensuite lancés sur des partitions contenant un nombre équitable d'arcs afin de pouvoir calculer le poids des arcs à l'aide de la fonction *semiGlobalAlignmentScore* de la classe *Graphe*. Une fois tout les arcs créés et leurs poids calculés, la fonction sort() de Java a été utilisé en prenant comme argument un *Comparator* afin de pouvoir trié selon le paramètre de poids.

3.4 Remarques

Afin de gérer le cas où un fragment serait inclus à un autre, nous avons effectué une vérification visant à s'assurer qu'un noeud d'un arc ne soit pas inclus aux noeuds auxquels il est relié. Cette vérification est effectuée dans l'algorithme *greedy*, avant de faire l'union entre deux ensembles.

3.5 Optimisations apportées

Tout d'abord, nous avons décidé de ne pas créer d'objet noeud. Un tel objet ne contiendrait que le fragment. De ce fait, avec pour objectif d'utiliser un minimum de mémoire, nous avons préféré l'utilisation d'un *record Edge* qui sera utilisé par la suite dans l'identification du chemin hamiltonien. Cet objet, étant donné qu'il va contenir plusieurs informations telles que : le noeud source, le noeud d'arrivée et le score d'alignement entre les deux fragments va pouvoir simuler la présence de noeuds tout en minimisant l'utilisation d'objets.

Nous avons aussi pris le parti prix de calculer le poids de l'arc à la création de celui-ci. C'est un choix qui nous fait perdre du temps lors de la création de la liste d'arc, mais qui nous en fait gagner lors du calcul du chemin hamiltonien, étant donné que l'on pourra récupérer le poids de l'arc directement, sans devoir le recalculer à chaque fois.

Nous avons choisis de faire une liste contenant l'ensemble des fragments ce qui est, certes, assez coûteux en mémoire, mais étant donné que nous

n'avons pas d'objet noeud, ce coût est balancé et permet ainsi d'accéder à chaque fragment en temps constant par la suite.

Une autre optimisation apportée a été d'ajouter du *multithreading* dans la création et tri de nos arcs. En effet, ayant $nn-2$ arcs (où n est le nombre de noeud) il est impératif de pouvoir faire du calcul en parallèle pour déterminer leurs poids. Chaque *thread* va s'occuper de calculer le poids des arcs pour une série de noeuds, qui sont répartis équitablement entre les *threads*.

4 Alignement des fragments

Une fois le chemin hamiltonien trouvé, une liste de fragment est obtenu. De cette liste, il faut recalculer les tables de similarités par deux fragments et dans l'ordre. Donc le premier avec le second, le second avec le troisième, etc.

A partir de cette table, il nous faut reconstruire l'alignement des deux fragments correspondants. La fonction *alignements()* de la classe *Graph.java* intervient pour réaliser cette opération. Cette fonction ressort une liste d'objet *Alignement* composés de 3 éléments : le premier fragment (nommé source), le deuxième fragment(nommé destination) et ainsi qu'un entier contenant les *shifts* qui seront utilisés par la suite pour la propagation des gaps suivant l'approche semi-globale.

4.1 Calcul de l'alignement semi-global

Tout d'abord, deux *StringBuilder* sont créés pour sauvegarder les résultats de l'alignement entre deux fragments. L'alignement est retourné sous la forme d'un *record* contenant le score de l'alignement, la matrice de l'alignement qui nous servira à retrouver l'alignement optimal, l'index de la position de la plus grande valeur dans la matrice, et un *byte* indiquant si l'index se trouve sur la colonne ou la ligne.

Une fois l'index de la plus grande valeur de la matrice identifié, on va remonter dans les valeurs de la matrice jusqu'à arriver à un 0 (l'index $i = 0$ ou l'index $j = 0$). On ajoute les caractères dans les *StringBuilder* suivant que c'est un *match*, un *missmatch* ou un *gap*. Dans le cas où un des deux index se trouve à 0, mais que l'autre à toujours une valeur > 0, il faut ajouter les derniers caractères du fragment non complété, en prenant soin d'appliquer des décalages à l'autre fragment.

5 Gestion des gaps

Une fois l'alignement entre les fragments refait, il faut maintenant propager les *shifts* afin que tous les fragments soient correctement alignés. Pour ce faire, la méthode *shifts* se trouvant dans la classe *Graph* à été créé.
Cette méthode modifie chaque alignement se trouvant dans la liste des alignements (voir section 4) afin de lui ajouter un certain nombre de *shifts* représenté par la variable $shifts$ dans chaque alignement. Ainsi, les *shifts* seront reportés à tous les alignements suivants.

Pour ce faire, on va instancier une variable avec la destination de notre noeud actuel et calculer son nombre de *shifts*. On va ensuite regarder l'alignement suivant et on va lui ajouter les *shifts* de la destination précédente à ceux déjà possédé. On répète cette opération pour chaque alignement de la liste et l'on obtient ainsi une suite d'alignement décalé.

5.1 Optimisation apportée

Une optimisation trouvée à cet gestion de gaps en début de chaîne a été de stocker, dans une variable, le nombre de gaps que l'on doit ajouter à un alignement. En effet, par ce procédé, nous arrivons facilement à voir le nombre de *shifts* à effectuer sur un alignement lors de l'étape du consensus (voir section 6) tout en économisant de la mémoire, car nous n'agrandissons pas la taille de nos fragments dans l'alignement.

6 Consensus

La dernière étape de notre projet consiste à déterminer le nucléotide majoritaire par rapport à l'alignement effectué, ainsi qu'aux décalages ajoutés. Étant donné que nous avons en notre possession une liste des alignements ordonnés selon le chemin hamiltonien, il est facile de déterminer le nucléotide étant le plus représenté.

Le consensus consiste à parcourir tous les fragments qui se chevauchent, et à comparer le nombre de nucléotide pour un indice donné. Le nucléotide le plus présent sera celui sélectionné pour le résultat final. En cas d'égalité, l'ordre préféré est : a > c > t > g > -.

Pour ce faire l'algorithme *concensus* de la classe *FragmentAssembler* va prendre la liste des alignements précédemment décalés, et va ressortir un

String contenant la chaîne de caractère final représentant notre séquence ADN.

L'algorithme fonctionne de la manière suivante :
Tout d'abord, la méthode *findSomething* permet de déterminer l'index du premier alignement ne présentant pas de *shifts*. Pour se faire, quand il rencontre un alignement avec des *shifts*, ce nombre est réduit de un. Quand un alignement sans *shifts* est trouvé, l'index de cet alignement est retourné.
Ensuite, pour chaque alignement dans la liste des alignements qui suivent l'index trouvé précédemment, si le nombre de *shifts* est plus grand que zéro, on réduit ce nombre de un et on passe au suivant. Par contre, si le nombre de *shifts* est égal à zéro, on récupère le premier caractère des fragments composants l'alignement, et on incrémente les compteurs correspondants. Les fragments se trouvant dans cet alignement "perdent" alors leur premier élément. De cette façon, il n'est pas nécessaire de sauvegarder d'index, ou des fragments modifiés, car seul le premier caractère sera toujours observé.

7 Résultats obtenus

Une première observation porte sur le nombre de nucléotides présentent dans notre séquence finale. Effectivement, nous avons remarqué que notre séquence était toujours plus grande que la séquence cible.

Ensuite, en utilisant l'outil *dotmatcher*, nous avons généré les diagrammes présentés dans les sous-sections suivantes. La documentation fournie par *dotmatcher* nous indique que notre séquence se trouve sur l'axe des y, et que la séquence cible se trouve sur l'axe des x. Ainsi, lorsqu'une similarité est observée entre les deux, un point est placé sur le graphe. De plus, deux régions importantes de similarité vont entraîner l'apparition d'une diagonale sur le graphe.

7.1 Collection 1

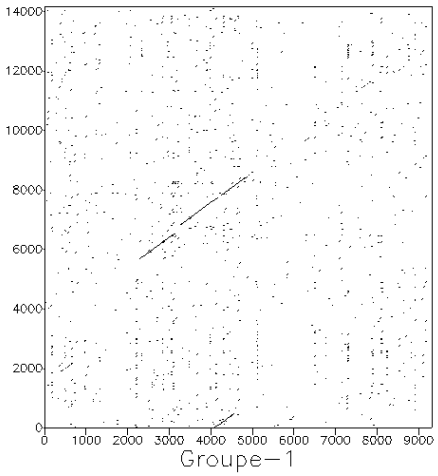

FIGURE 1 – Dotmatcher - output pour la première collection.

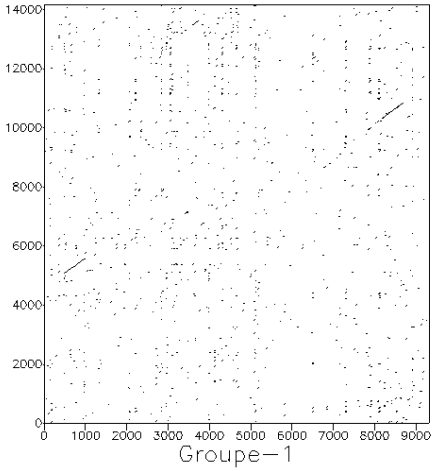

FIGURE 2 – Dotmatcher - output (complémentaire inversé) pour la première collection.

La Figure 1 présente le résultat de notre alignement pour la première collection. Sur cette figure, nous observons plusieurs similarités (représentées par les diagonales), ainsi qu'une bonne couverture, représentée par le nombre de points.

De plus, nous observons sur la Figure 2 que moins de similarité sont présentent, par contre, il semblerait que nous avons obtenu une meilleure

couverture.

7.2 Collection 2

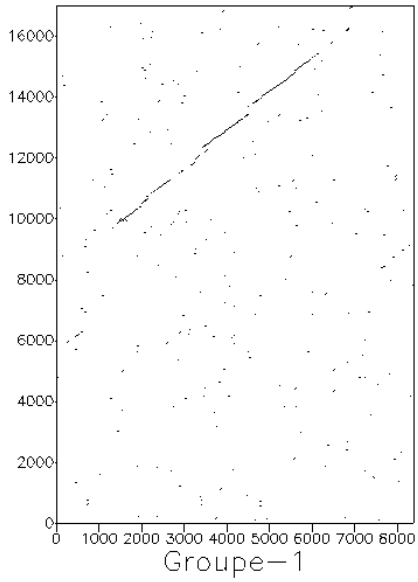

FIGURE 3 – Dotmatcher - output pour la deuxième collection.

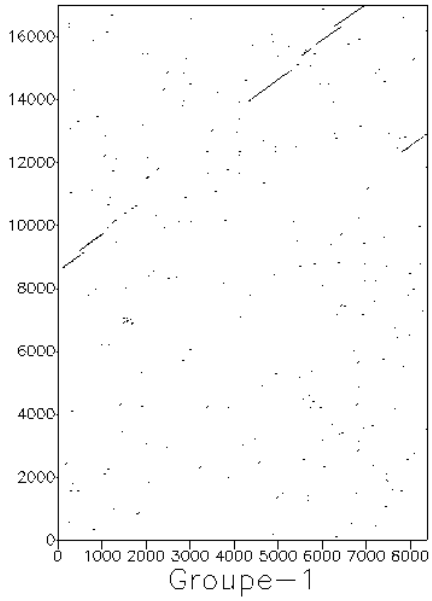

FIGURE 4 – Dotmatcher - output (complémentaire inversé) pour la deuxième collection.

La Figure 3 présente le résultat de notre alignement pour la deuxième collection. Sur cette figure, que les similarités avec la séquence cible sont importantes, et que la couverture globale est un peu faible.

De plus, nous observons sur la Figure 4 que le nombre de similarités reste foncièrement le même. Finalement, la couverture ne semble pas être meilleure.

7.3 Collection 3

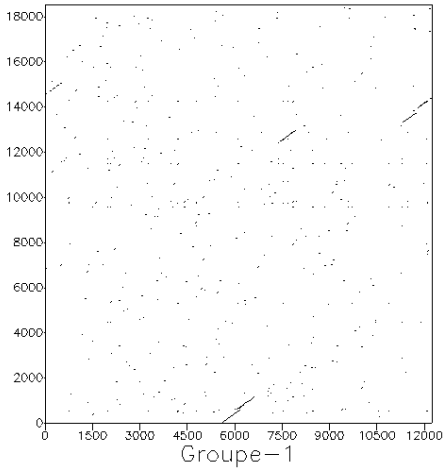

FIGURE 5 – Dotmatcher - output pour la troisième collection.

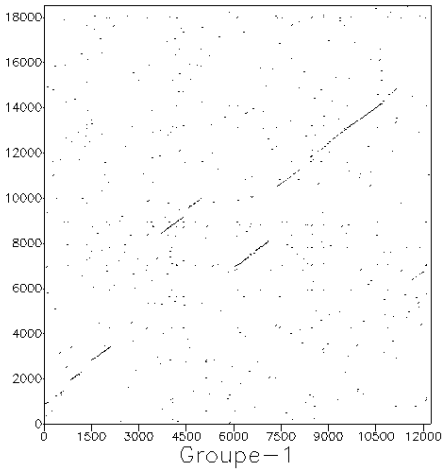

FIGURE 6 – Dotmatcher - output (complémentaire inversé) pour la troisième collection.

La Figure 5 présente le résultat de notre alignement pour la troisième collection. Nous observons quelques similarités avec la séquence cible. Par contre, la couverture reste assez faible.

Dans le cas de la Figure 6, nous observons que les similarités sont plus importantes bien que la couverture reste faible.

7.4 Collection 4

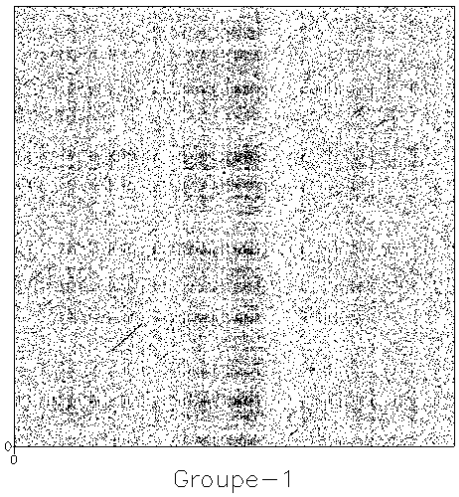

FIGURE 7 – Dotmatcher - output pour la quatrième collection.

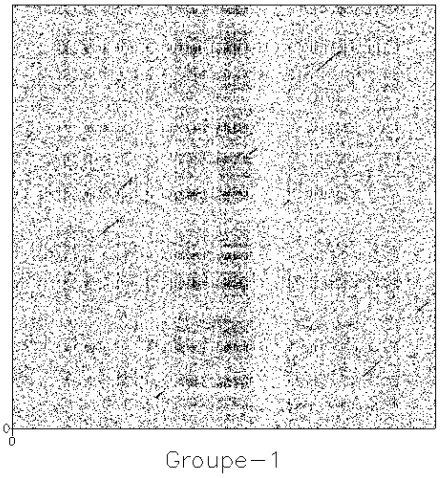

FIGURE 8 – Dotmatcher - output (complémentaire inversé) pour la quatrième collection.

Finalement, la Figure 7 présente le résultat de l'alignement pour la quatrième collection. On peut voir que la couverture est plutôt importante, et que quelques similarités sont présentent.

Pour le complémentaire inversé de notre séquence, la Figure 8 présente une légèrement moins bonne couverture, avec un nombre de similarités qui semble être plus important. Du fait du nombre de points sur le graphique, il est difficile d'avoir une observation très précise.

8 Remarques

L'exécution du programme se fait avec Java 17. Pour exécuter notre code, il suffit d'utiliser la commande "java -jar FragmentAssembler.jar <fichier.fasta>–out <sortie.fasta> –out-ic <sortie-ic.fasta>".

8.1 Points forts

Le programme ne se base pas seulement sur les performances en temps d'exécution mais aussi en temps de mémoire. Nous avons essayé de garder un équilibre entre ces deux éléments pour éviter une "explosion" du temps d'exécution ou de la mémoire utilisée.

De plus, l'utilisation du *multithreading* permet d'améliorer le temps de calcul pour l'étape la plus gourmande.

Nous avons constaté que, sur une machine récente, le temps d'exécution de programme est de moins de cinq minutes pour les trois premières collections. Cependant, pour la quatrième, nous avons observé un temps d'exécution de plus ou moins une heure.

Finalement, chaque fonctionnalité du programme a été implémentée de façon modulaire, ce qui permet un travail bien réparti, et une facilité pour corriger les éventuels bugs rencontrés lors de l'implémentation.

8.2 Points faibles

Une zone d'ombre reste sur la partie "report de gaps". Cette partie du projet peu présenter quelques imperfections et erreurs de logique.

Aussi, les résultats obtenus, bien que couvrant parfois une bonne partie de la cible, ne sont pas excellents et des améliorations devraient pouvoir permettre d'atteindre plus effacement la cible.

8.3 Amélioration par rapport à janvier

— Tout d'abord une refonte complète de l'approximation du chemin hamiltonien a été effectuée, étant donné que celui-ci ne fonctionnait pas lors de la première session.

— Une optimisation du tri des arc a aussi été faite pour simplifier le code et le rendre plus performant. Cette optimisation à été possible grâce, entre-autres, à l'utilisation du *multithreading*.
— Le code dans sa globalité subit un *refactoring* dans le but d'y apporter des améliorations en terme de logique et de performance.

8.4 Répartition des tâches

La répartition des tâches de ce projet c'est fait de la manière suivante :
Laurent :
- Lecture et écriture des fichiers
- Alignement semi-global
- Algorithme Greedy
- Consensus

Guillaume :
- Graphe
- Report de Gaps
- Algorithme Greedy
- Rapport

Cette répartition est assez approximative, en effet, pour de nombreux points, les deux parties se sont entraidées lors de leurs parties respectives. De plus pour permettre un projet plus cohérent et complet, les points importants ont été réfléchis en commun.

9 Conclusion

Ce projet représentait un challenge intéressant compte tenu des techniques de bioinformatique à comprendre et découvrir, ainsi qu'aux algorithmes à mettre en place.

Certaines parties du projet n'étant pas clairement décrites dans le cours, ni sur les moteurs de recherche, et il a parfois été nécessaire de prendre des décisions subjectives sur les méthodes à adopter pour résoudre la problématique.

Cette prise de décisions a entraîné des problèmes d'implémentation par rapport à des parties du projet qui étaient, elles, mieux décrites. Il a donc fallut accorder nos choix par rapport aux parties les mieux décrites.

Cependant, ce projet nous aura permis de montrer l'application que peut avoir l'informatique dans une autre science, dans des conditions qui auraient pu être réelles.

Finalement, ce projet à été l'opportunité d'améliorer nos connaissances en Java, et de réfléchir à des problèmes de performances qui ne sont pas toujours le centre de nos préoccupations pour les autres projets.