Université de Mons
Faculté des sciences
Département d'Informatique

Algorithmique et bioinformatique
Rapport de projet

Professeur :
Olivier Delgrange
Clément Tamines

Auteur :
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 récuperer 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 les classes suivantes : *Graph* et *Edge* principalement. Certains problèmes ont été rencontrés lors de la conception de nos algorithmes et seront détaillés par la suite.

Une fois le graphe obtenu, nous avons appliqué un algorithme *Greedy* (c'est à dire une approche gloutonne) afin de calculer le chemin Hamiltonien de ce graphe. Pour cette recherche de chemin, tout se déroule dans la classe *Graph*. Il en résulte une liste contenant un objet *Edge* qui contient le chemin Hamiltonien recherché.

Une fois que le chemin est obtenu, 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

2.1 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.

Pour ce qui est de l'identification du début d'un fragment, il a été nécessaire d'utiliser une expression régulière. 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 Construction de l' *Overlap Graph*

3.1 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.

3.2 Création de la liste d'arc

Les arcs du graphe font le lien entre chaque noeud du graphe excepté eux-même et leur complémentaire inversé c'est pourquoi pour un graphe contenant n noeuds, on va pouvoir trouver m arcs tel que $m = n*(n-2)$. De plus

chaque arc possède un poids qui est la similitude entre le noeud de départ et le noeud d'arrivée. Cette similitude est calculée à partir de la chaîne de caractère commune maximale entre la fin du fragment de départ et le début du fragment d'arrivée.

3.2.1 Calcul de poids d'un arc

Pour calculer ce poids c'est à dire le chevauchement entre les deux noeuds que l'arc courant relie, nous avons implémenté une matrice d'assemblage (voir *getOverlapGraph* dans la classe *Graph*). Dans cette algorithme, nous avons utilisé comme demandé dans les consignes l'alignent semi-global c'est à dire que nous avons créé une matrice où notre première ligne et première colonne sont des 0 et que l'alignement ce trouve par backtracking. Cet algorithme nous retourne donc le score du meilleur alignement et ce score sera donc le poids de notre arc.

3.3 Optimisation apportée

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 de n'utiliser qu'un minimum de mémoire, nous avons préféré l'utilisation d'un objet *Edge*. Cet objet, étant donné qu'il va contenir plusieurs informations telles que : le noeud source, le noeud d'arrivée et le poids (ainsi qu'un "chemin" mais cet élément sera expliqué par la suite) 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 qu'il faudra seulement récupérer le poids et non le recalculer à chaque fois.

Nous avons choisis de faire une liste contenant l'ensemble des arcs 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 arc en temps constant par la suite.

4 Calcul du chemin Hamiltonien

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 était 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 noeud selon le poids de leur arc (poids 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 le graphe créé précédemment (voir section 3) et va nous ressortir une liste contenant un arc spécial qui contient le chemin hamiltonien.

Tout d'abord, nous avons créé deux tableau de byte *in* et *out* de la taille du nombre de noeuds qui contienne respectivement un 0 ou un 1 à l'indice i si l'on est déjà entré dans le noeud i et déjà ressorti du noeud i.

Nous avons ensuite trié notre liste de chemin par poids décroissant (voir sous-section suivante) et ainsi pu boucler sur chaque arc afin de créer un chemin hamiltonien.

Pour chaque arc, on vérifie tout d'abord si l'on est pas déjà sorti de sa source ni déjà rentré dans son noeud d'arrivée et si ce n'est pas le cas alors on modifie son *in* et *out* correspondant (ainsi que ceux des noeuds complémentaires-inversés afin d'être sur de ne pas passer dans le noeud contenant le fragment f et le noeud contenant le fragment $\bar{f}$.

Après cela, on va vérifier si on ne peut pas unir l'arc avec un autre arc déjà visité afin de créer un chemin au fur et à mesure. Pour ce faire nous avons implémenté une fonction *union* qui va parcourir chaque arc déjà visité et vérifié si la source de notre arc à ajouter est identique à une arrivée d'un arc de la liste et si l'arrivée de notre arc est identique à la source d'un arc de la liste. Si c'est le cas alors les arcs vont fusionner et le chemin sera dans la variable chemin de ce nouvel arc.

A la fin de la visite de chaque arc un chemin est alors créé sous la forme d'une liste contenant un arc qui a pour variable un chemin qui est le chemin

hamiltonien de notre graphe.

4.1 Optimisation pour le calcul du chemin Hamiltonien

Lors du calcul du chemin, il est nécessaire de trier la liste des arcs de manière décroissante. Pour effectuer ce tri, nous avons choisi le *Bubble sort* qui, en moyenne et dans le pire des cas, va effectuer n^2 comparaisons et n^2 permutations. Dans le pire des cas, la complexité en mémoire de ce genre d'algorithme de tri est en O(n).
Le compromis fait entre la rapidité de son exécution et la complexité en mémoire nous a semblé intéressant et c'est pourquoi nous l'avons intégré à notre projet.

Nous avons choisi de stocker le chemin directement dans l'arc afin de gagner en mémoire étant donné que l'objet *Edge* existait déjà et ce chemin est en fait une liste de fragment ordonné qui va de la source vers la destination de cet arc.

4.2 Problèmes rencontrés

Un problème a été rencontré lors de cette partie et n'a pas pu être résolu ce qui fausse la suite du programme, en effet lors du passage dans *greedy* seul un fragment qui était la fusion des deux premiers fragments à la place de l'ensemble des fragments qui forment le chemin hamiltonien. Malgré de nombreuses recherches et essai de modification, il ne nous a pas été possible de résoudre ce bug et donc l'output de cet algorithme fausse les résultats de la suite de notre programme et donc les algorithmes suivants sont fonctionnels seulement de manière théorique.

5 Gestion des gaps

Pour gérer les gaps, i.e. ajouter un certain nombre de gaps en début de fragment afin que le fragment soit le mieux aligné possible avec le précédent, nous avons créé une méthode *gaps* se trouvant dans la classe *FragmentAssembler*. Cette méthode modifie chaque fragment se trouvant dans le chemin hamiltonien (voir section 4) afin de lui ajouter un certain nombre de gaps représenté par la variable $shift$.

Pour ce faire, On va parcourir l'ensemble des fragments en le comparant avec le fragment suivant. Pour chaque paire de fragment, on va comparer

chaque nucléotide entre eux afin d'obtenir un score (voir sous-section calcul de score) et on retiendra le décalage ayant obtenu le meilleur score afin de déterminer le nombre de gaps à ajouter à chaque fragment.

5.1 Calcul de score

Lors du calcul du score d'alignement entre deux fragments, nous allons parcourir les deux fragments nucléotide par nucléotide sur le minimum de la longueur des deux fragments et pour chaque comparaison, trois cas sont possibles :

1. Dans le premier cas, soit le nucléotide du fragment 1 ou le nucléotide du fragment est un gaps. Dans ce cas là nous avons choisi d'ajouter 0 au score.
2. Il est possible que les deux nucléotides comparés soient différents dans ce cas là nous avons affaire à un *missmatch* et donc nous soustrayons 1 au score.
3. Enfin, si les deux nucléotides comparés sont égaux, nous allons incrémenter le score de 1.

Au final, nous aurons un entier qui sera le score de chaque comparaison de l'alignement actuel des deux fragments.

5.2 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 en début de fragment. En effet, par ce procédé nous arrivons facilement à voir de combien le fragment doit être décalé pour le consensus (voir section suivante) tout en économisant de la mémoire, car nous n'agrandissons pas la taille de notre fragment.

6 Consensus

La dernière étape de notre projet consiste à déterminer le nucléotide majoritaire par rapport à l'alignement effectué, ainsi qu'aux gaps ajoutés. Étant donné que nous avons en notre possession une liste des fragments alignés et ordonnés selon le chemin hamiltonien, la calcul du consensus peut se faire en plusieurs étapes :

- Itérer sur tous les fragments du chemin

- Déterminer les fragments suivants qui *overlap* avec le fragment actuellement considéré
- Déterminer l'indice à partir duquel il faut commencer la comparaison. Effectivement, lors d'une itération précédente un partie de du fragment actuel peut déjà avoir été traité, et ne doit donc plus l'être
- Effectuer le consensus avec le vote à la majorité

Pour trouver les fragments qui *overlap* avec le fragment actuel, nous prenons le longueur de ce fragment et l'on y soustrait le nombre de *shifts* calculés lors de la recherche du meilleur alignement. Tant que la longueur reste positive, les fragments *overlap* avec le fragment initial. Quand cette valeur devient négative, c'est que les fragments suivants ne vont pas avoir d'*overlap* avec le fragment actuel.

Pour déterminer l'indice à partir duquel il faut commencer le consensus, nous avons calculé le nombre d'éléments qui n'ont pas encore été traités sur le fragment actuel par rapport aux alignements précédents. Avec cette valeur, nous pouvons facilement trouver l'indice de départ pour le consensus suivant.

Le consensus consiste à parcourir tous les fragments qui *overlap* 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.

7 Résultats obtenus

Nous n'avons pas été capable de comparer nos résultats avec les résultats souhaités. Cette incapacité est liée à des erreurs dans certaines parties de notre programme qu'il ne nous a pas été possible de résoudre car nous n'avons pas réussis à les identifier clairement (i.e. nous avons compris où elles se trouvent, mais nous n'avons pas identifié pourquoi elles sont présentent).

8 Remarques

L'exécution du programme se fait avec Java 15. Pour lancer 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.

8.2 Points faibles

Un point faible de notre algorithme se trouve dans le calcul des arcs. En effet, lors de la création de notre graphe nous calculons tous les poids de tous les arcs à l'aide de l'alignement semi-global ce qui est très coûteux en temps. Une manière de remédier à ce problème pourrait être de paralléliser le calcul des poids des arcs afin de réduire drastiquement le temps d'exécution.

8.3 Répartition des tâches

La répartition des tâches de ce projet c'est fait de la manière suivante :
Laurent :

- Lecture et écriture des fichiers
- Report de Gaps
- Consensus

Guillaume :

- Graphe
- Alignement semi-global
- Algorithme Greedy

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, il a parfois été nécessaire de prendre des décisions personnelles 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.