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 et dans notre cas c'est la classe *FragmentAssembler* qui 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 un fragment et un ensemble d'arc représentant la similarité entre les fragments. Ce graphe fait intervenir les classes suivantes de notre programme : *Graph* et *Edge* principalement. Certains problèmes ont été rencontrés lors de la conception de celui-ci 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*. Et en résulte de cet algo une liste contenant un objet *Edge* contenant le chemin Hamiltonien.

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 par majorité afin de pouvoir recréer notre séquence finale qui est enregistrée dans un fichier *.fasta* .

2 Récupération des fragments

3 Construction de l' *Overlap Multigraph*

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 ecxepté euxmeme 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 poid qui est la similitude entre le noeud de départ et le noeud d'arrivée. Cette similitude est calculée à partir de la chaine de caractère commune maximale entre la fin du fragment de départ et le début du fragment d'arrivée.

3.2.1 Calcul de poid d'un arc

Pour calculer ce poid 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'aligment 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 poid de notre arc.

3.3 Optimisation apportée

4 Calcul du chemin Hamiltonien

Lors de la présentation du projet, il nous a été proposé d'utilisé 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 poid de leur arc (poid 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.

5 Gestion des gaps

Pour gérer les gaps c'est à dire 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 fonction *gaps* se trouvant dans la classe *FragmentAssembler* et 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 fagment, 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 décrémentons le score de 1.

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

6 Consensus

7 Résultats obtenus

8 Remarques

8.1 Points forts

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 couteux en temps. Une manière de remédier à ce problème pourrait être de parralléliser le calcul des poids des arcs afin de réduire drastiquement le temps d'exécution.

8.3 Répartition des tâches

La répartion 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 complèt, les points importants ont été réfléchis en commun.

9 Conclusion