Projet - Algorithmique et Bioinformatique
Assemblage de fragments d'ADN

UMONS
Faculté des Sciences
Ma1 Sciences informatiques.

Delplanque Julien
Charlier Maximilien

Année académique
2015-2016

1 Introduction

Dans le cadre du cours “Algorithmique et Bioinformatique”, nous sommes amenés à implémenter un programme d’assemblage de fragments d’ADN. Ce programme utilise une approximation de type Greedy.

Le programme développé prend en paramètre un fichier fasta contenant un ensemble de fragments et retourne un fichier fasta qui contient le fragment résultant.

2 Représentation des Fragments

Les fragments sont représentés de deux façon différentes durant l’exécution du programme.

Tout d’abord, les fichiers “fasta” sont “parsés” et des objets *Fragment* sont générés. Ces objets *Fragment* stockent la suite de nucléotide sous la forme d’un tableau de caractères (2 octets / caractère). À l’origine, l’utilisation de l’objet *String* avait été considérée mais après quelques comparaisons de performances entre un tableau de caractères et un objet *String*, il a été décidé d’utiliser la représentation décrite précédemment car celle-ci est plus performante dans le cas présent. Cette représentation sera utilisée durant les étapes d’alignement semi-global, de construction du graphe et de l’algorithme “Greedy”.

Lors de l’assemblage, la représentation utilisant l’objet *Fragment* est abandonnée au profit d’une représentation utilisant une liste-chainée (classe *LinkedList* de l’API Java). Ce choix est justifié par le besoin de faire des insertions successives de “gaps” dans le fragment. Si la représentation utilisant l’objet *Fragment* avait été utilisée lors de l’insertion de “gaps”, des copies superflues du tableau auraient été nécessaires. Dans le cas d’une liste chainée, il n’y a pas de copie requise. Cette deuxième représentation sera utilisée jusqu’à l’étape du consensus.

Lors du consensus, un objet *Fragment* est généré. Celui-ci est ensuite écrit sur le disque sous forme textuelle (fichier fasta).

2.1 Alignement semi-global

L’alignement semi-global est implémenté dans la classe *Fragment* par la méthode *semiGlobalAlignmentWith*. Cette méthode peut soit prendre un autre objet *Fragment* en paramètre, soit prendre un objet *Fragment* et la matrice calculée par programmation dynamique. Cette matrice peut elle-même être calculée par un objet *Fragment* via la méthode *computeMatrixWith*. Cette deuxième méthode qui prend le fragment et la matrice en paramètre est utile afin de pouvoir réutiliser la matrice calculée lors de l’alignement semi-global entre deux fragments f et g pour calculer l’alignement semi-global entre g et f. En effet, appelons la matrice calculée pour l’alignement semi-global entre f et g, M_{fg} et celle calculée pour l’alignement semi-global entre g et f, M_{gf}. Alors, on a la relation suivante :

$$M_{fg} = M_{gf}^T$$

Afin d'exploiter cette propriété, l'objet *Matrix* représentant une telle matrice dispose d'une méthode *setReverseAccess* qui prend un booléen en paramètre et permet d'inverser les accès à des éléments de la matrice via la méthode *dataAt*. En fait, lorsque l'élément aux l'indices (x, y) sera accédé et que la méthode *setReverseAccess* avec comme paramètre *true* aura été appelée précédemment, c'est l'élément d'indices (y, x) qui sera retourné. Ce procéder permet de simuler la transposition de la matrice sans devoir créer un second objet *Matrix*.

D'autre part, lors du calcul du maximum de la dernière ligne ou colonne, si le maximum apparait plusieurs fois dans la dernière ligne ou colonne, le maximum qui a la position i, j la plus proche de l'indice $\text{height}(M), \text{width}(M)$ est choisi. Choisir ce maximum permet de limiter le nombre de gaps insérés au début d'un alignement.

3 Algorithme Greedy

Pour la réalisation de l'algorithme Greedy, plusieurs structures de données ont été utilisées. Notamment un graphe (sommets et arcs) et une notion d'ensemble pour les sommets. L'algorithme sera appliqué sur un graphe et le modifiera via un effet de bord. Ensuite, celui-ci retournera la liste des arcs choisies. Il faudra ensuite placer cette liste dans un second algorithme afin d'extraire une liste ordonnée de fragments qui représente un chemin hamiltonien dans le graphe.

3.1 Constructions du graphe

En partant d'une liste de fragments, nous construisons les sommets du graphe. Chaque fragment sera présent deux fois : le fragment original et son complémentaire inversé. Afin d'améliorer le comportement de l'algorithme, les sommets contenant des complémentaires inversés sont liés entre eux. Avant d'ajouter un sommet et donc un nouveau fragment, on va vérifier si celui-ci n'est pas déjà présent dans un autre sommet. La même vérification est faite pour son complémentaire inversé. Si le fragment est déjà présent dans le graphe, on ne l'ajoute pas. Cette démarche permet de ne pas avoir des doublons dans le graphe en présence de fragment complémentaire inversé dans le fichier source `.fasta`.

Pour sélectionner un sommet, il faut que son complémentaire n'ai pas déjà été choisi par l'algorithme. Une fois un sommet choisi, le sommet contenant le fragment complémentaire inversé de celui-ci sera "bloqué" en passant `in` et `out` à `true`.

Les arcs du graphe représentent le score entre la source et la destination. Nous avons constaté que le calcul des scores était très long et nécessitait d'être parallélisé. Il est donc possible d'utiliser des threads dont le nombre est paramétrable par l'utilisateur. Un gain de temps significatif a été observé en parallélisant la construction du graphe.

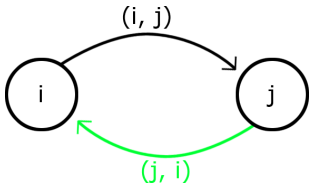
3.2 Optimisation du calcul des scores et du multi-threading

L'algorithme Greedy calcule les alignements entre deux fragments dans les deux sens. Cela revient donc a calculer deux alignements semi-globaux, un entre f et g et un autre entre g et f.

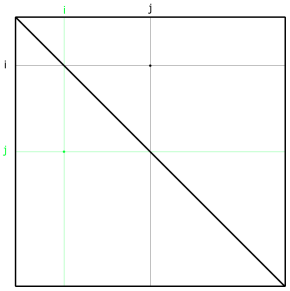
En représentant les calculs sous forme de graphe (illustré en FIGURE 1a), cela revient à calculer l'arc entre i et j et l'arc entre j et i. Nous avons illustré dans la SECTION 2 la possibilité d'utiliser une méthode *setReverseAccess* pour améliorer les performances lors d'u calcul de ce type.

Cette méthode est utilisée lors de la construction des arcs. **Cela nous a permis de diminuer le nombre de calculs d'un facteur deux ce qui divise par deux le temps d'exécution de l'algorithme !**

Le calcul des arcs se fait via deux boucles imbriquées. Au niveau des calculs, cela peut être représenté par une matrice carrée. On constate que calculer l'arc (i, j) permet de calculer l'arc (j, i). On peut alors intuitivement déduire via la FIGURE 1b que seule la moitié de la matrice doit être parcourue.



(a) Arc du graphe



(b) Position arc dans la matrice

FIGURE 1 – Illustration d'un score dans le graphe.

Lors du multithreading, on parcourt donc que le triangle supérieur à la diagonale de la matrice. Le travaille des threads doit néanmoins être réparti de façon homogène. Chaque thread doit donc parcourir la même surface de la matrice afin d'avoir une bonne division du travail.
La FIGURE 2a, met en avant le fait que la division de la matrice peut former des parallélogrammes.

Contrairement à une parallélisation standard sur un tableau, il ne suffit pas de diviser la boucle extérieur en x intervalles de même taille. Dans notre cas, on chercher a trouver les indices de la boucle extérieur qui divisent la demi matrice en x surface égale. Une illustration pour deux surfaces égale est donné en FIGURE 2b, on remarque que l'intervalle $[0, x]$ et $[x, \text{size}]$ n'ont pas la même longueur.

(a) Surface contenue dans un trapèze rectangle **(b)** Matrice carrée divisée en deux surfaces égale

FIGURE 2 – Illustration de la division de la matrice.

Nous avons choisi de calculer cette répartition grâce à une fonction récursive. Celle-ci permet de diviser une matrice de taille $size$ en n surfaces égale où n sera une puissance de deux. La contrainte que n doit être une puissance de deux, coïncide au nombre d'appels récursifs, cela n'est pas contraignant car le nombre de cœurs sur un processeur récent est toujours une puissance de deux.

La fonction de division de surface, a deux types d'appel récursif, soit pour diviser un triangle rectangle, soit trapèze rectangle en deux.

Nous allons illustrer les calculs effectués pour diviser un trapèze rectangle en deux surface égale. En partant du trapèze défini par les variables x, $size$, $size'$, x' illustré en FIGURE 3, on cherche y tel que l'air du polygone rose soit égale à l'air du polygone jaune qui doit être égale à l'air du trapèze rectangle x, $size$, $size'$, x' divisé par deux. On obtient donc le système d'équation suivant :

$$\begin{cases} \dfrac{x(size-x)+\frac{(size-x)^2}{2}}{2} & (1) \\ x(y-x)+\dfrac{(y-x)^2}{2} = (1) & (2) \\ y(size-y)+\dfrac{(size-y)^2}{2} = (1) & (3) \end{cases}$$

Il n'y qu'une inconnue y, elle peut s'extraire de l'équation (2) facilement du fait que c'est une équation du second degré. L'algorithme récursif est donc basé sur cette équation.

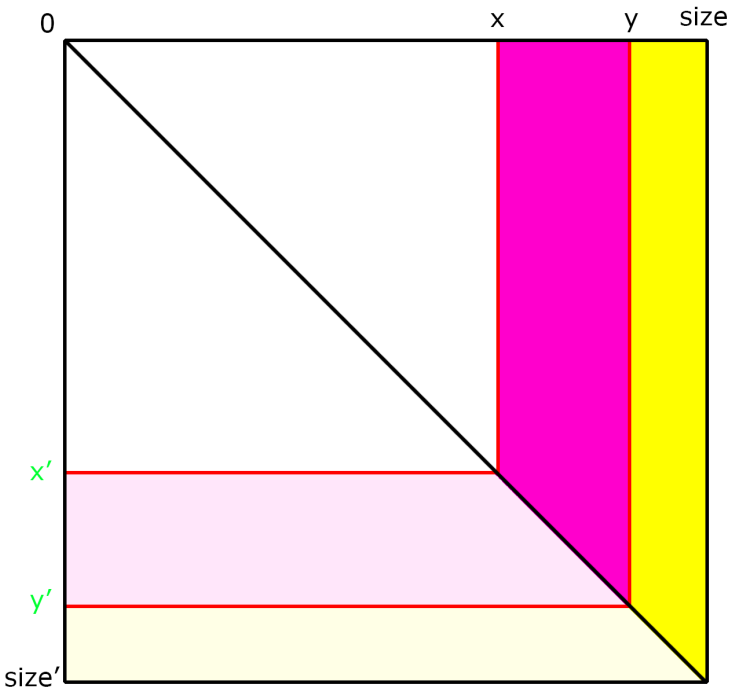


FIGURE 3 – Illustration de la division d'un trapèze rectangle en deux.

La même logique a été suivie dans le cas des triangles rectangle à diviser en 2.

4 Assemblage

4.1 Alignement global

Calcul des alignements semi-globaux

Afin de réaliser l'alignement global, la première étape est de recalculer les alignement semi-globaux entre chaque paire successive de fragments du chemin hamiltonien (les alignements calculés précédemment ayant été "oubliés" dès qu'il ne fussent plus utiles afin d'économiser de la mémoire).

Formellement, soit f_i un fragment et A_i un alignement semi-global entre deux fragments, pour un ensemble ordonné de n fragments $\{f_0, f_1, \ldots, f_{n-1}\}$, un ensemble ordonné de $n-1$ alignements semi-globaux est calculé : $\{A_0, A_1, \ldots, A_{n-2}\}$ où A_i correspond à l'alignement semi-global de f_i et f_{i+1}.

À l'implémentation, ce calcul est réalisé par la méthode *computeAlignments* de la classe *MultiAlignment*. Pour rappel, un fragment est représenté par un objet *Fragment* et un alignement par un objet *Alignment*.

Calcul des offsets

Une fois ces alignements calculés, "l'offset" de chaque fragment nécessaire à l'alignement global est calculé.

Formellement, soit o_i "l'offset" de f_i, un ensemble ordonné de n "offsets" est calculé à partir des alignements : $\{o_0, o_1, \ldots, o_{n-1}\}$.

À l'implémentation, un tableau d'entiers est utilisé afin de stocker les "offsets". Ce tableau d'entier est lui-même stocké dans la variable d'instance *offsets* de la classe *MultiAlignment*.

Calcul et insertion des gaps

Enfin, les "gaps" à insérer dans chaque fragment et à propager dans le chemin sont calculés.

À chaque fragment f_i sera associé deux ensembles de "gaps" : un ensemble contenant les "gaps" à propager aux fragments suivant le fragment f_i dans le chemin hamiltonien et un ensemble contenant les "gaps" à propager aux fragments le précédent dans le chemin hamiltonien.

Formellement, soit Gf_i et Gb_i les "gaps" à propager respectivement aux fragments suivant i ("forward") et aux fragments précédents i ("backward") alors un ensemble ordonné de tuples contenants les "gaps" à propager vers l'avant et vers l'arrière est calculé : $\{(Gf_0, Gb_0), (Gf_1, Gb_1), \ldots, (Gf_{n-1}, Gb_{n-1})\}$.

Une fois ces "gaps" connus, pour chaque fragment d'indice i dans le chemin, les "gaps" de Gf_i sont insérés dans f_i et propagés à tous fragments f_j où $j > i$ et les "gaps" de Gb_i sont insérés dans f_i et propagés à tous fragments f_h où $i > h$.

À l'implémentation, le calcul des "gaps" est réalisé par la méthode *computeGapsToPropagate* de la classe *MultiAlignment* et la propagation de ceux-ci est réalisée par la méthode *propagateGaps* de cette même classe. La classe *GapsToPropagate* (interne à la classe *MultiAlignment* est utilisé pour stocker deux listes d'entiers. L'une contient les indices des "gaps" à insérer dans le fragment associé et ses prédécesseurs et l'autre ceux à insérer dans le fragment associé et ses successeurs.

Résultat

Le résultat de ces trois étapes est une liste de fragments contenant des gaps et une liste d'offsets qui sont prêtes à être utilisées afin de réaliser le consensus.

À l'implémentation, les fragments sont stockés dans la variable d'instance *fragments* de la classe *MultiAlignment* et les offsets dans la variable d'instance *offsets* de cette même classe.

4.2 Consensus

Afin d'expliquer le fonctionnement du consensus fait sur la liste de fragments contenant des "gaps" et la liste "d'offsets" associés aux fragments, nous allons commencer par faire une abstraction des données d'entrée de l'algorithme. Au lieu de penser en terme de fragments et "d'offsets", nous allons imaginer une matrice dont les lignes représentent les fragments (avec éventuellement un décalage ("offset") au début représenté par des cases contenant le symbole "_"). La largeur de cette matrice sera déterminée par le fragment dont la valeur de la taille additionné à l'offset est la plus grande. La fin des lignes de la matrice contenant un fragment dont la valeur de sa taille additionnée à son offset est inférieure à la largeur de la matrice sont comblée par des symboles "_".

À partir de cette abstraction, le consensus est très facile à réaliser. Il suffit de parcourir chaque colonne et de choisir le nucléotide qui apparait le plus de fois. Si plusieurs nucléotide apparaissent un nombre égale de fois, le choix du nucléotide pour le consensus peut être fait aléatoirement car chacun a une probabilité égale d'apparaitre.

Exemple :

$$\begin{bmatrix} \_ & \_ & a & c & g & a & c & g & t & \_ \\ a & c & a & t & g & - & c & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & g & a & t & c & g & a \end{bmatrix} \rightarrow \begin{bmatrix} a & c & a & c & g & a & c & g & t & a \end{bmatrix}$$

Cette approche n'est néanmoins pas utilisée à l'implémentation et ce pour plusieurs raisons d'optimisation :

- Stocker l'offset encodé dans plusieurs cases d'une matrice est inutile car celui-ci ne contient qu'une seule information : le décalage en question. Il sera donc stocké sous forme d'entier lors du consensus.
- Il est inutile de stocker l'espace non utilisé après les fragments dans une matrice car il est possible de calculer si l'indice de la colonne traitée est inclue à un fragment en utilisant "l'offset" et la taille de celui-ci.
- Lorsque les fragments sont traités pour réaliser le consensus, ils sont représentés sous forme d'une liste chainée. Construire une matrice à partir de ces listes chainée serait couteux en temps.

Le consensus devra donc être réalisé sous les deux contraintes suivantes :

1. Un fragment est représenté sous forme d'une liste chainée.
2. Un offset est représenté sous forme d'un entier.

Puisque les fragments sont des listes chainées et que l'algorithme dois progresser dans chacun des fragments afin de compter les nucléotides, des itérateurs seront utilisés. Ces itérateurs permettent de réduire le temps d'accès à un élément de la liste chainée. En effet ceux-ci permettent de manipuler finement le déplacement sur celle-ci. Si la méthode *get* de la classe *LinkedList* était utilisée, à chaque accès à l'élément i de la liste, tous les noeuds avant le noeud i devraient être parcourus.

L'algorithme utilisera donc la méthode *next* de l'itérateur afin de faire progresser chaque itérateur sur sa liste respective.

Une représentation schématique de la progression des itérateurs sur les listes chainées est disponible en FIGURE 4.

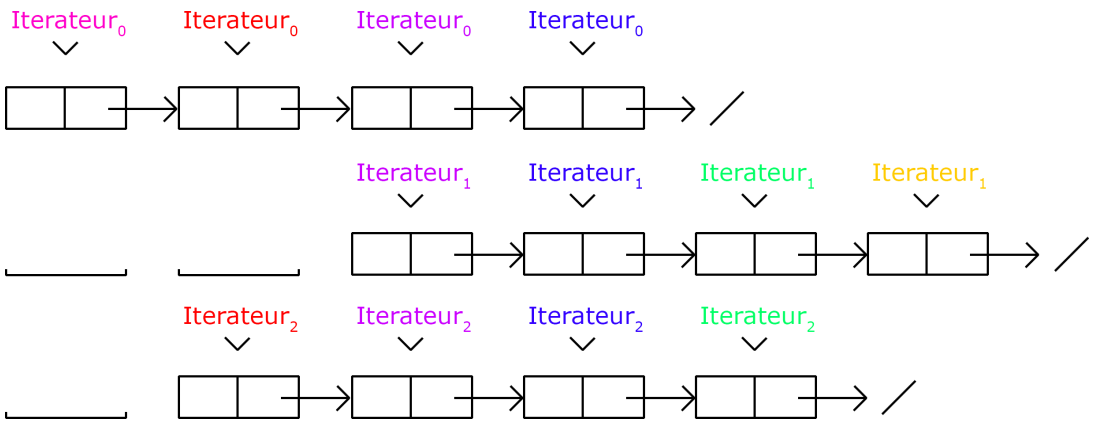


FIGURE 4 – Illustration des itérateurs parcourant les fragments.

C'est la classe *ListIterator* de l'API de Java qui sera utilisée.

Avant de réaliser le consensus l'itérateur de chaque liste chainée est récupéré et stocké dans un tableau. Ensuite, on va calculer la valeur maximal de l'indice global. Cette valeur correspond au maximum des tailles des fragments auxquelles est ajoutée la valeur de l'offset correspondant.

Une boucle va alors itérer de 0 à la valeur maximale de l'indice global et calculer le nucléotide qui apparait le plus de fois à chaque itération. Pour une itération i, la liste des fragments va être parcouru et les fragments dont $o_i \leq i$ et $i < o_i + \text{size}(f_i)$ seront considérés pour l'étape i (les autres ne sont pas concerné par l'étape i et donc ignoré). Dans les fragments considérés, le nucléotide apparaissant le plus de fois sera selectionné pour être celui en position i dans le super-fragment créé.

À l'implémentation, la classe *ConsensusCounter* se charge de compter les occurences de chaque nucléotide via la méthode *incrementScoreOf* et de décider quel nucléotide sera utilisé dans le super-fragment via la méthode *chooseCharacter*.

La classe *ConsensusMaker* est chargée de calculer le consensus via la méthode *doConsensus*. D'autre part, la méthode *computeMaximalGlobalIndex* est utilisé pour, comme son nom l'indique, calculer l'indice global maximal.

5 Interface en ligne de commande

Une interface en ligne de commande a été créée afin de pouvoir paramétrer le programme. L'aide du programme est accessible en utilisant l'option "**--help**".

Il est possible de paramétrer le nombre de threads à utiliser lors de la construction du graphe ("**--threads** <**n**>"), spécifier si la collection passée en paramètre est une collection simple ("**--simple**") ou normale ("**--normal**"), de spécifier les fichiers à écrire en sortie ("**--out** <**file**>" et "**--out-ic** <**file**>") ou de ne simplement pas écrire le résultat dans un fichier ("**--no-out**"), afficher le progrès lors de la construction du graphe ("**--progress** <**nbr-of-iterations**>") et enfin choisir le paramètre diviseur à utiliser pour le score des fragments ("**--divider** <**n**>").

Cette interface en ligne de commande implémente donc, en particulier, l'interface commune à tous les groupes définie lors d'une séance de travaux pratiques.

6 Résultats obtenus

Pour chaque collection, nous allons procéder à une analyse basée sur le dotmatcher obtenu en comparant le fichier cible fourni pour la collection avec le consensus généré par notre algorithme. Nous avons constaté que Greedy pouvait être amené à choisir un fragment inverse complémenté lors de son exécution qui n'a pas été choisi par l'algorithme utilisé pour concevoir le fichier cible. Il peut en résulter des "trous" dans la visualisation avec dotmatcher de notre consensus final par rapport au fichier cible du fait qu'un "mauvais choix" de Greedy peut entrainer d'autres choix différents en cascade.

De ce fait, par rapport au fichier cible fourni, certaines parties identiques peuvent être présentes sous leur forme d'inverse complémenté. Lors de la visualisation des résultats avec dotmatcher, nous avons donc décidé de prendre en compte l'inverse complémenté du résultat généré par notre algorithme. Nous considérons comme résultat principal le fragment d'ADN reconstitué par l'algorithme, mais superposeront les résultats obtenus dans dotmatcher avec les résultats obtenus avec son inverse complémenté. Si le fichier principal obtenu a des "trous" l'inverse complémenté pourrait les remplir si l'on tombe dans le scénario décrit plutôt.

Lors des premières exécutions nous avons constaté des problèmes de concensus trop court par rapport à la cible fournie. Cela provenait du fait que le choix des arcs dans Greedy ne favorisait par l'apparition d'un "escalier" pour le chemin hamiltonien. On entend par là, le fait que l'on pouvait choisir un arc A $\longrightarrow$ B sur deux fragments où B serait à gauche de A plutôt qu'a droite. Pour allonger le fragment résultant, nous avons favorisé les "gaps" à gauche de B, pour cela, nous avons ajouté des "points" au score pour chaque "gaps" présent à gauche de B. De ce fait, les arcs A $\longrightarrow$ B avec des "gaps" à gauche de B obtiennent de meilleurs scores et sont donc choisi prioritairement.

6.1 Collection 1

Simplifiée

Sur cette collection, nous avons obtenu de très bon résultats. Comme on peut le voir sur la FIGURE 5, une diagonale presque parfaite est affichée dans le dotmatcher.

Cela signifie que la correspondance entre le fichier cible et le fichier généré par notre algorithme est très grande.

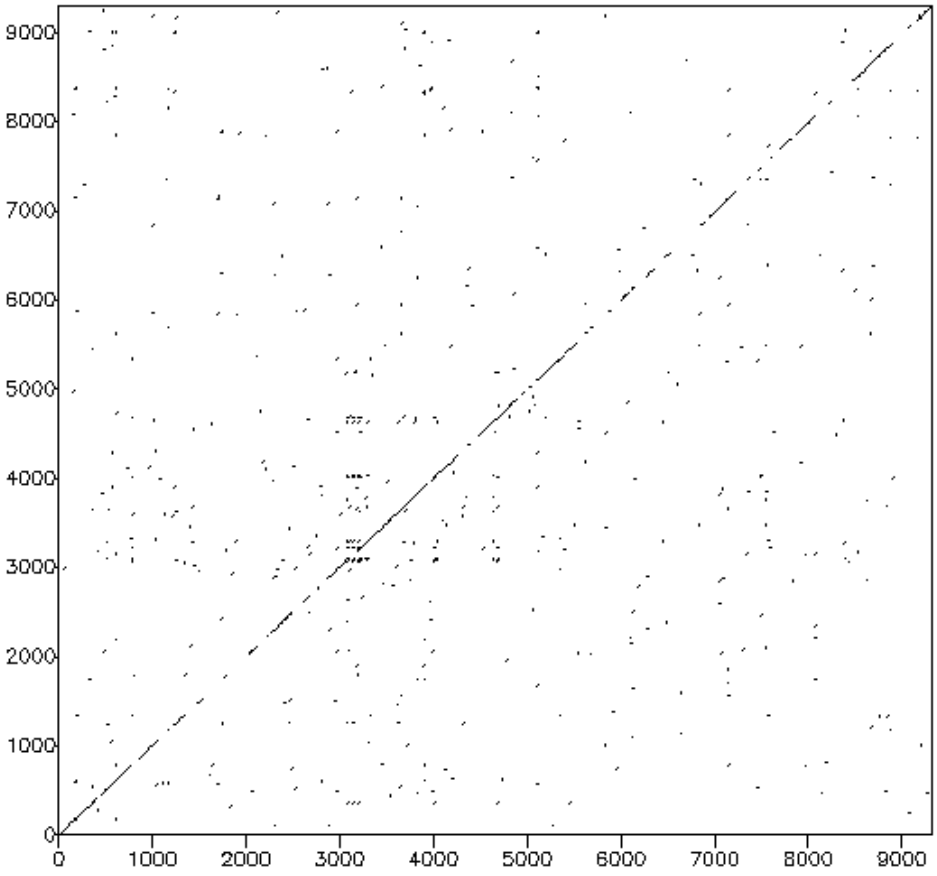
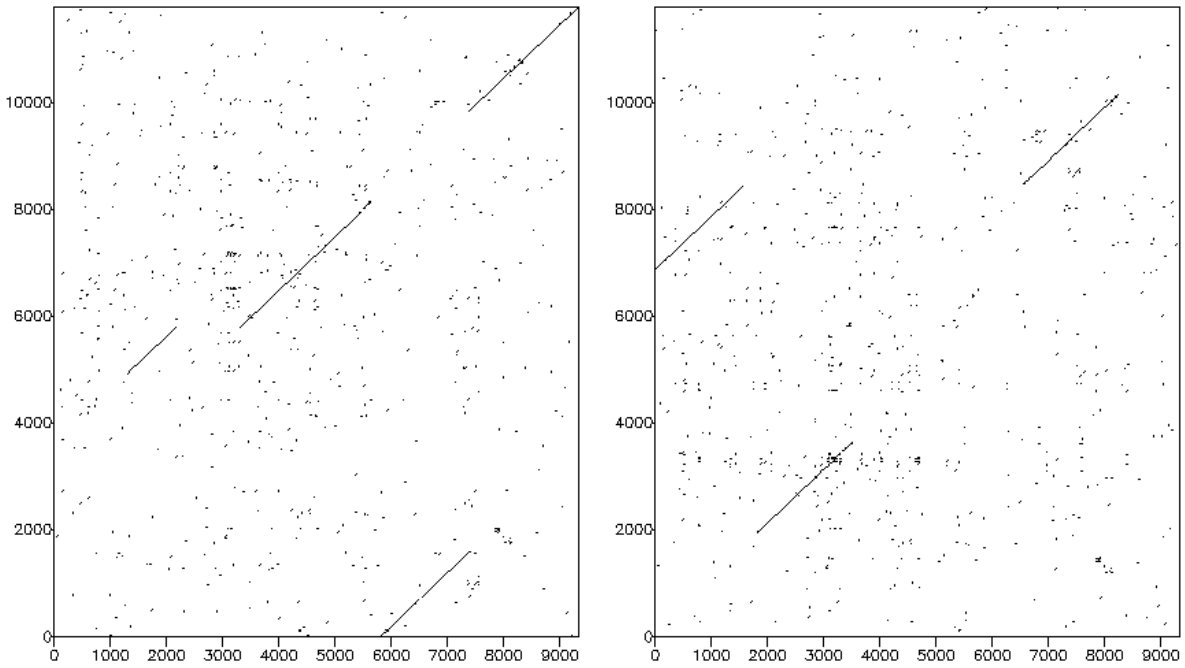


FIGURE 5 – Illustration du dotmatcher obtenu pour la collection 1 simplifiée .

Normale

La collection 1 "normale" est l'une des collections qui doivent être supportées par notre algorithme. Sur la FIGURE 6a, on constate que notre consensus est plus long que le consensus cible. En axe Y est représenté le fichier cible, en axe X le fichier obtenu par notre algorithme. Le fichier cible est donc un fragment d'ADN de longueur 11000, alors que celui généré par Greedy fait 9000 de longs. Pour avoir une correspondance parfaite, il faut donc qu'une diagonale ou des parties de diagonale fusionnée entre elles traversent le graphique de haut en bas. Le fichier obtenu ne traverse pas totalement le graphique de haut en bas, on constate des "trous" de 2000 à 5000 et de 8000 à 1000. Par contre, si l'on regarde l'inverse complémenté dans dotmatcher en FIGURE 6b , on constate que ces "trous" sont justement presque totalement comblés. Celui de 2000 à 5000 est comblé de 2000 à 4000. Quant à celui de 8000 à 1000, il est presque totalement comblé. Nous avons donc illustré en figure 6, la superposition des deux fichiers générée par notre algorithme. On constate que la pratique confirme l'interprétation donnée en début de section concernant le fonctionnement de Greedy. On constate aussi que notre fragment d'ADN reconstitué grâce a Greddy recouvre presque complètement la verticale

ce qui indique un très bon résultat.



(a) Obtenu **(b)** Inverse complementé

FIGURE 6 – Illustration du dotmatcher obtenu pour la collection 1.

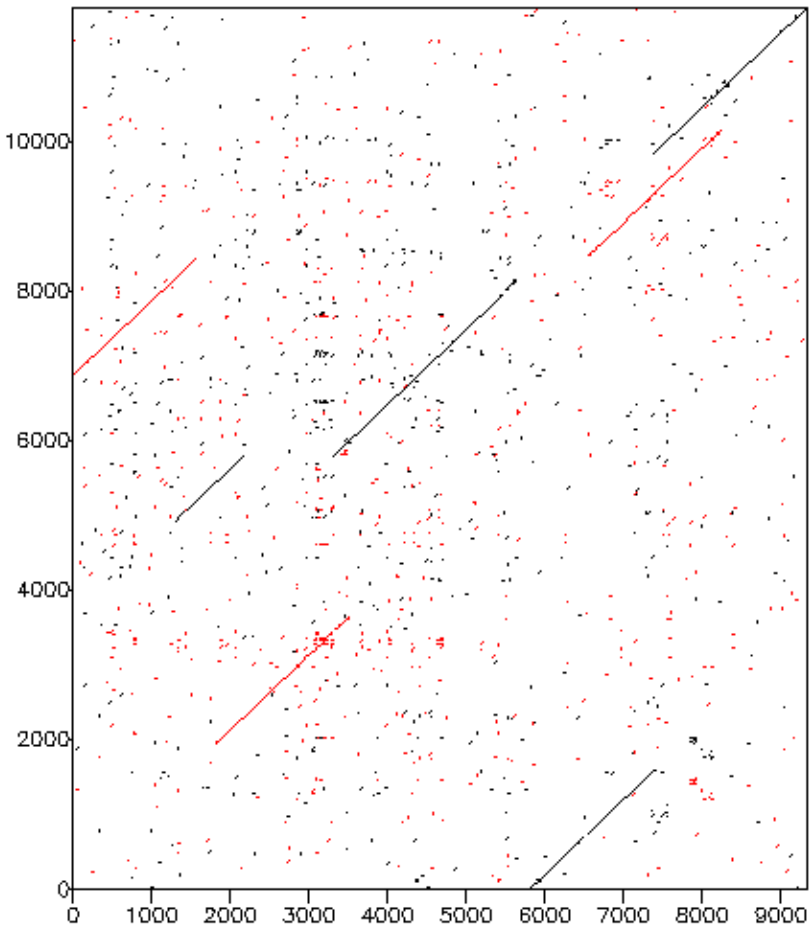


FIGURE 7 – Illustration du dotmatcher obtenu pour la collection 1 superposée avec l'inverse complimenté.

6.2 Collection 2

Simplifiée

La FIGURE 8 illustre les résultats obtenu en comparant notre fragment d'ADN reconstitué avec le fragment cible de la collection 2. On remarque que si l'on traverse horizontalement le dotmatcher de haut en bas, on n'a pas de "trous" dans celui-ci. Cela signifie que le résultat est très bon et que la similitude entre notre résultat et celui attendu est bonne. On a le même contenu mais pas le même ordre.

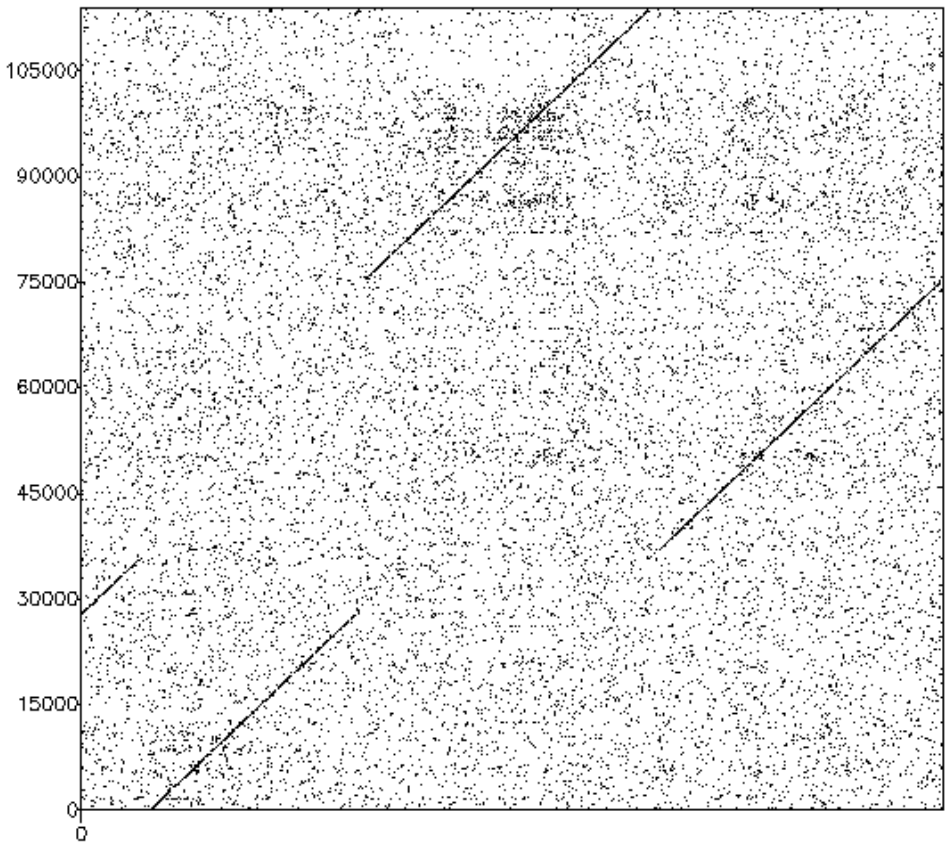
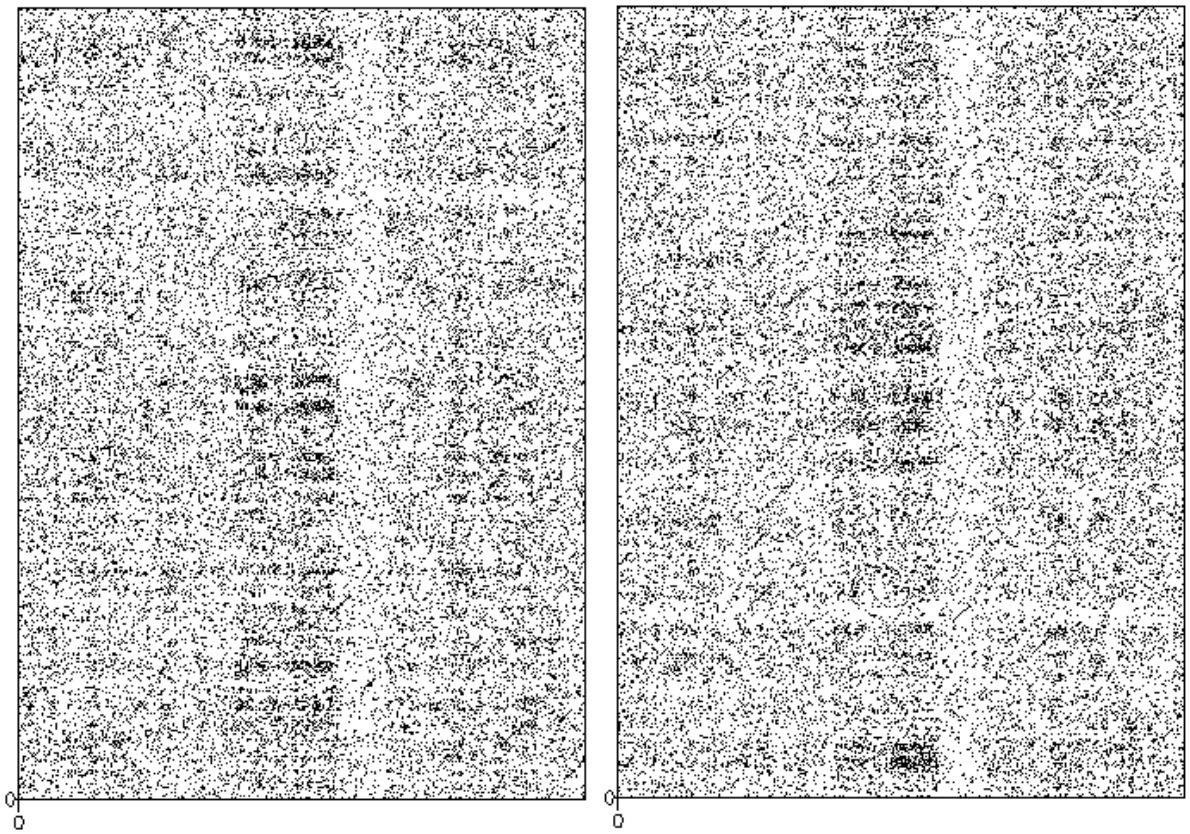


FIGURE 8 – Illustration du dotmatcher obtenu pour la collection 2 simplifiée .

Normale

Les résultats de la collection 2 normale ne sont pas bon. En effet on peut voir en FIGURE 9 qu'aucune ligne ne se dégage du plot. Par manque de temps, nous n'avons pas su trouver la source de ce problème dans les différents algorithmes implémentés dans le cadre de ce projet.



(a) Obtenu **(b)** Inverse complementé

FIGURE 9 – Illustration du dotmatcher obtenu pour la collection 2 normale.

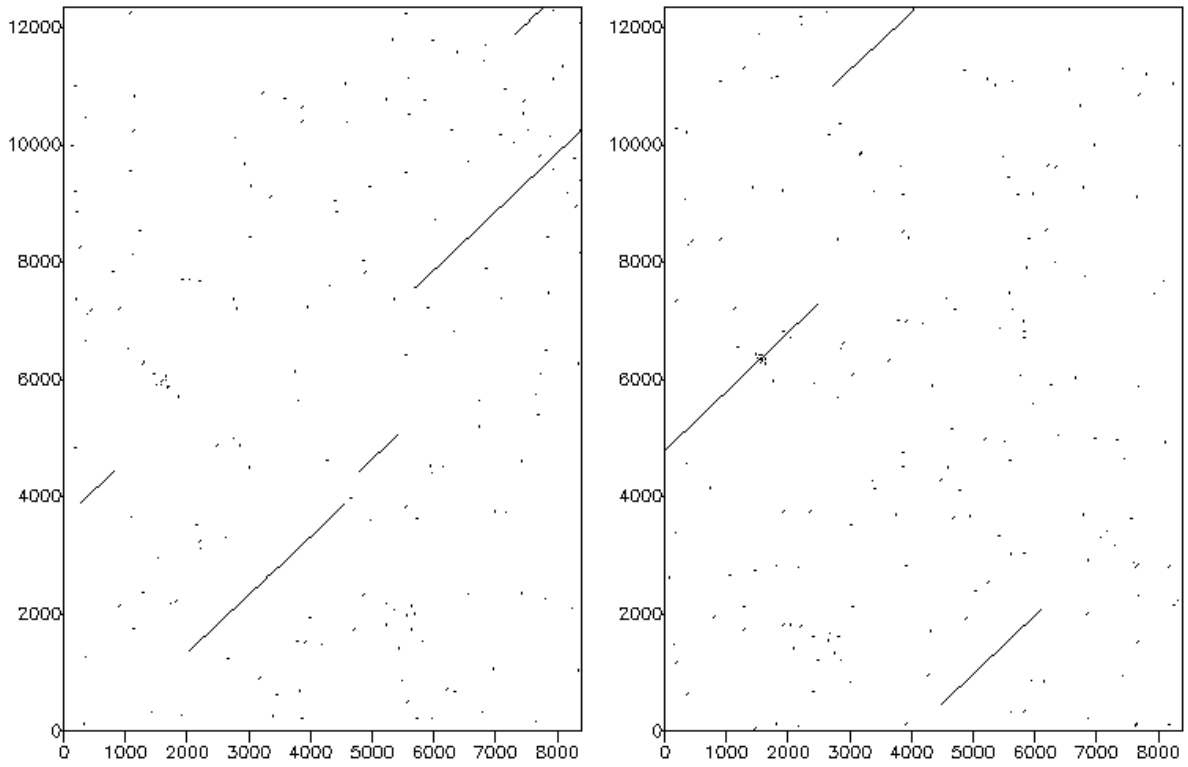
6.3 Collection 3

La collection n'est pas traitable par notre logiciel (du moins en moins de 20 heures, après ces 20 heures de calculs le graphe n'était toujours pas construit en entier le programme a donc été arrété) car celle-ci est très grande et prends donc beaucoup de temps de calcul. Puisque aucun des membres du groupe ne dispose d'une machine qu'il peut laisser calculer pendant plusieurs journées (et donc la rendre inutilisable durant ce temps), aucun résultat n'a été obtenu pour cette collection.

6.4 Collection 4

Pour cette collection, on obtient un fragment reconstitué plus court que le fragment cible. Illustré en FIGURE 11 la superposition du fragment d'ADN reconstitué et de son complémentaire inversé par rapport au fichier cible permet tout de même de limiter les "trous". Néanmoins, comme notre fragment reconstitué est plus court que le fragment cible, on a une perte d'information. De ce fait tout les "trous" ne peuvent pas être rempli.

La FIGURE 10, illustre quand à elle la comparaison entre le fragment reconstitué et le fragment cible ainsi que celle avec l'inverse complémentaire de façon séparé.



(a) Obtenu **(b)** Inverse complementé

FIGURE 10 – Illustration du dotmatcher obtenu pour la collection 4.

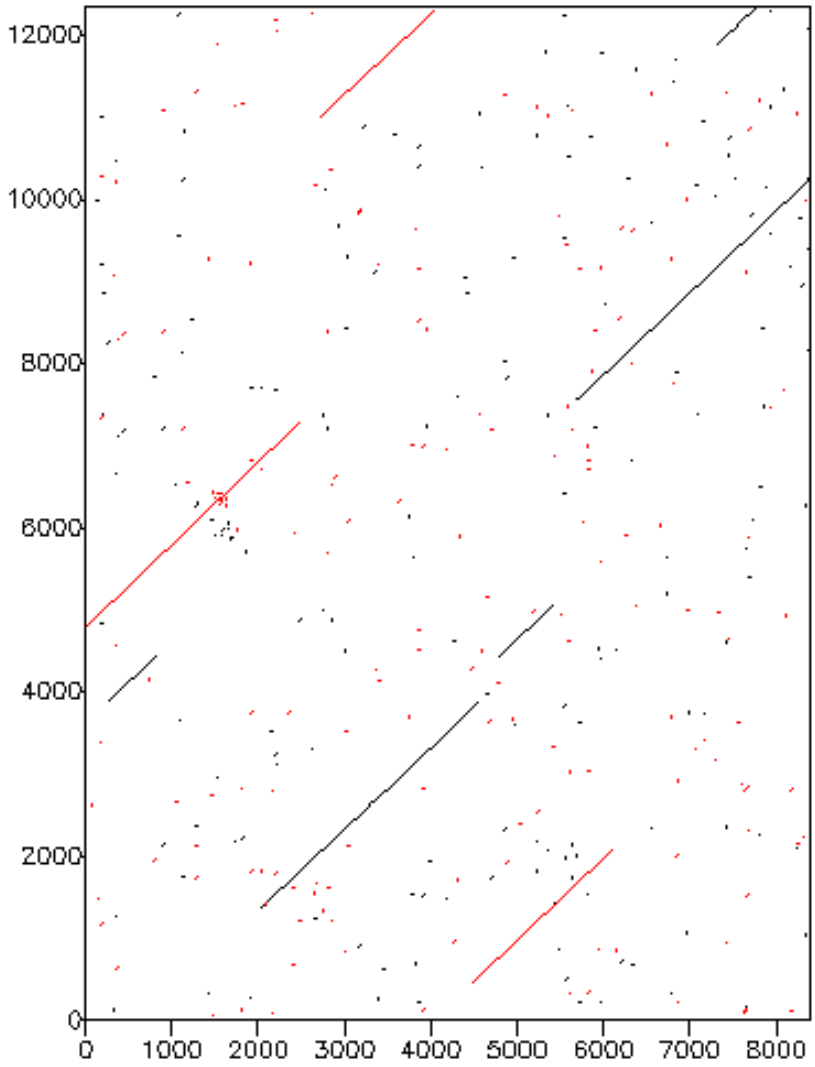


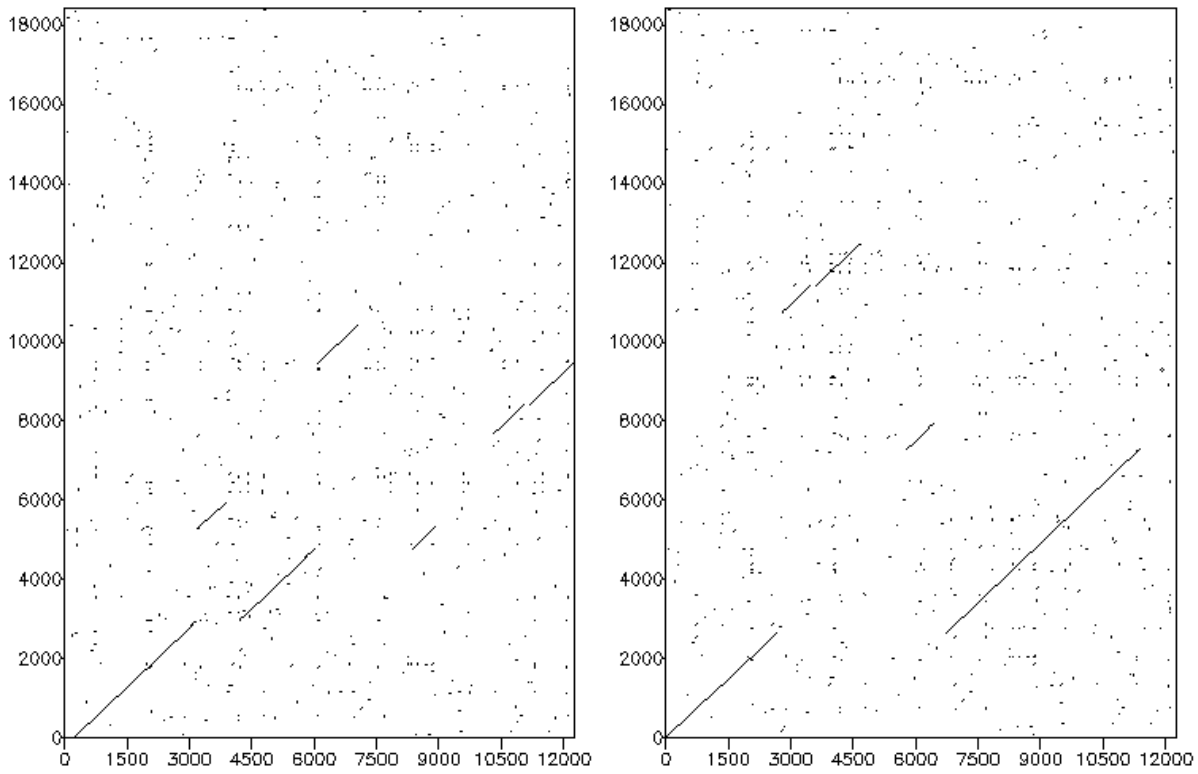
FIGURE 11 – Illustration du dotmatcher obtenu pour la collection 4 superposée avec l'inverse complimenté.

6.5 Collection 5

Pour cette collection, on obtient un fragment reconstitué plus court que le fragment cible. 12000 acides aminé contre 18000 pour le fragment cible. Cette différence de longueur est plus marquée que pour la collection 4. Elle est de 6000 acides aminé en moins, le fragment reconstitué est un tiers plus court que le fragment cible. Cela signifie une plus grande perte d'information et donc ne sera pas possible de ne pas avoir de trou dans les résultats.

Illustré en FIGURE 13, la superposition du fragment d'ADN reconstitué et de son complémentaire inversé par rapport au fichier cible permet tout de même de limiter les "trous". Néanmoins, comme notre fragment reconstitué est plus court que le fragment cible, on a une perte d'information. De ce fait tout les "trous" ne peuvent pas être rempli, environ un tiers n'est pas recouvert. Cela conforte l'observation théorique fait plus tôt.

La FIGURE 12, illustre quand à elle la comparaison entre le fragment reconstitué et le fragment cible ainsi que celle avec l'inverse complémentaire de façon séparé.



(a) Obtenu **(b)** Inverse complementé

FIGURE 12 – Illustration du dotmatcher obtenu pour la collection 5.

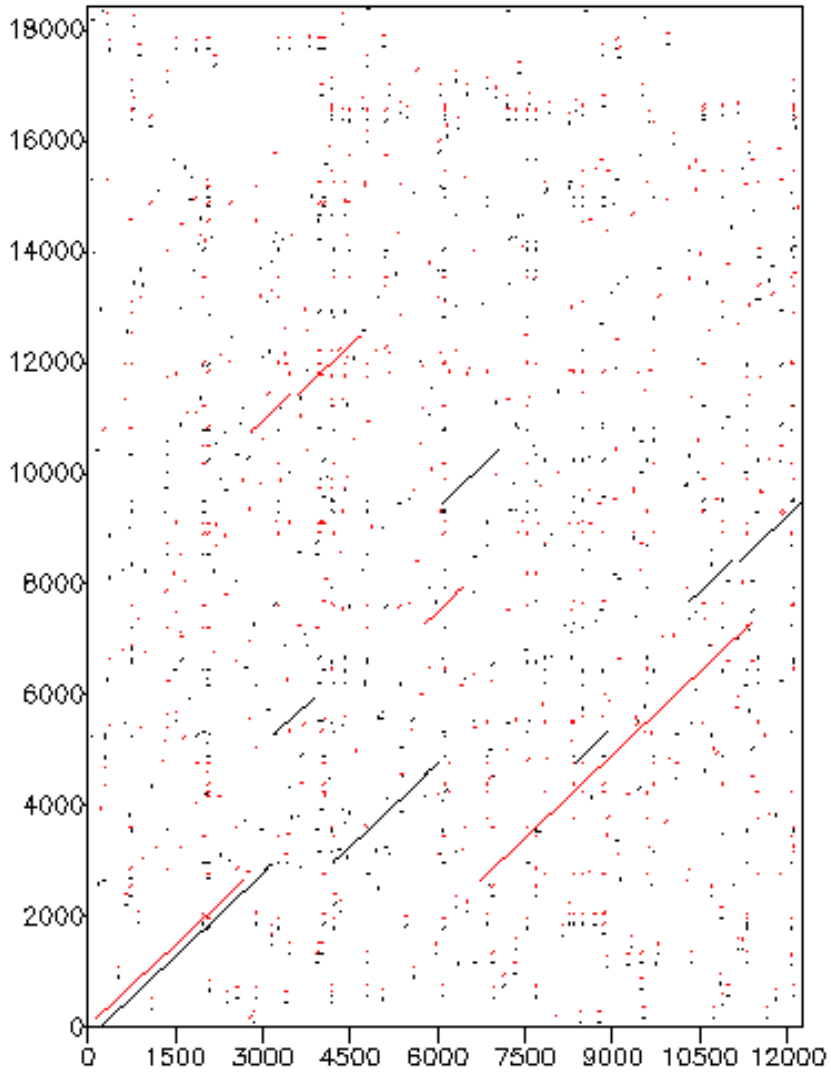


FIGURE 13 – Illustration du dotmatcher obtenu pour la collection 5 superposée avec l'inverse complimenté.

7 Points forts et faibles

7.1 Points forts

Du fait de la division du travail, le projet a été conçu de façon modulable. Il est donc possible de remplacer l'une des étapes (alignement semi-global, construction du graphe, propagation des gaps, consensus) facilement. D'autre part, le projet dispose de tests unitaires qui peuvent être lancés via la commande "ant test" dans le dossier "src". Finalement, le programme dispose d'une interface en ligne de commande permettant de le paramétrer de façon assez avancée.

7.2 Points faibles

Le programme n'est pas capable de traiter la collection 3 en un temps raisonnable et ses résultats sur la collection 2 ne sont pas de bonne qualité.

8 Répartition des tâches

Maximilien Charlier	Julien Delplanque
Construction du graphe	"Parsing" des fichiers fasta
Algorithme Greedy	Alignement semi-global
	Assemblage

Le rapport a été écrit par les deux membres du groupe.

9 Conclusion

Ce projet nous a permis d'implémenter des algorithmes vu aux cours et de nous rendre compte à quel point l'optimisation est importante lors de l'application de ceux-ci sur des problèmes concrets. Aucune difficulté notable n'a été remarquée lors de l'implémentation.