Algorithmique et Bioinformatique

Rapport de projet

Activité d'Apprentissage S-INFO-022

LAVEND'HOMME Thomas FLORIANI Laurence

Année Académique 2020-2021
Première année de Master en Sciences Informatiques

Faculté des Sciences, Université de Mons

18 décembre 2020

Table des matières

1 Introduction

Dans le cadre de cours d'algorithmique et bioinformatique, il nous est proposé d'implémenter une application permettant de reconstituer une séquence d'ADN à partir d'un ensemble de fragments. Plusieurs étapes sont nécessaires afin d'y parvenir.

Dans un premier temps, il faut lire le fichier et récupérer l'ensemble des fragments d'ADN. La classe `FastaIO` s'occupe de charger le fichier au format *fasta* et instancie des objets de la classe `Fragment`.

Dans un second temps, un graphe est construit à partir de cet ensemble de fragments. Il est caractérisé par une liste de noeuds, les fragments, et une liste d'arcs représentant la similarité entre deux fragments. Ceci fait intervenir les classes `Graph`, `Similarity` et `Edge`. Les premières difficultés se trouvent à cet endroit, nous en parlerons plus en détails dans la section dédiée.

Dans un troisième temps, un chemin hamiltonien doit être calculé dans le graphe précédemment construit. Pour se faire, une approche gloutonne est employée. À ce stade, les classes `GraphSet` et `FragmentPair` interviennent. Il en résulte une liste de fragments ordonnée selon le chemin hamiltonien choisi.

Dans un quatrième temps, il faut reconstruire les alignements pour toutes les paires de fragments dans l'ordre précédemment décidé. Ensuite, il faut propager les *gaps* et les décalages à tous les fragments.

Enfin, l'application d'une méthode de vote à la majorité permet d'obtenir une séquence finale qui sera enregistrée dans un fichier *.fasta*.

2 Récupération des fragments

Le fichier passé en entrée est au format *fasta* c'est-à-dire que les fragments y sont présentés avec une première ligne identifiée par "$>$" et contenant le numéro de fragment et la collection à laquelle il appartient suivi des caractères du fragments, à raison de soixante par lignes.

Un objet de type `Fragment` a plusieurs caractéristiques :

- *id* : Un entier donnant un identifiant à un fragment, celui-ci est généré de manière itérative dans le code afin d'éviter un risque de doublon ou d'erreur du fichier. Il est utile pour la suite afin de déterminer si deux fragments proviennent d'un même fragment.
- *sequence* : Tableau de *byte* qui contient effectivement la séquence du fragment correspondant.
- *variant* : Booléen qui indique si la séquence a été complémentée et inversée (*true*) ou pas (*false*)
- *shift* : Entier représentant le décalage appliqué à un fragment par rapport à un autre, il est initié à 0. Utile à partir de l'étape d'alignement entre deux fragments.

2.1 Gestion de la mémoire pour les fragments

Le principal point posant problème pour la gestion en mémoire des fragments est la séquence du fragment. En effet, elle peut contenir un nombre important de nucléotides.

D'abord, l'idée a été d'utiliser simplement des *String*s car il s'agit d'une structure existante de base dans Java. Le tableau 1 permet de constater qu'il ne s'agit pas de la plus efficace manière de procéder.

Ensuite, pour s'affranchir de la surcharge de 28 octets imposée par l'utilisation de *String*s, il a été décidé d'utiliser un tableau de caractère. Comme il est possible de le constater dans le tableau 2, un tableau est moins lourd en mémoire que les autres structures de données.

En parallèle, un essai avec des *Bit Set*s a été réalisé, par contre il s'est rapidement avéré que leur utilisation représentait une surcharge de calculs n'entraînant que peu d'améliorations au point de vue de la mémoire.

Enfin, dans un soucis d'optimisation, les *char*s du tableau précédent ont été remplacés par des *byte*, gagnant ainsi encore un octet par nucléotide de la séquence.

Type de donnée	Quantité de mémoire employée
byte	1 octet
short	2 octets
int	4 octets
char	2 octets
String	28 octets + *Array* de *char*

TABLE 1 – Comparaison des types de données en *Java*

Structure de donnée	Quantité de mémoire employée
Array	Pour n données de taille s : n * s + 32 octets
List, *Set*, etc.	Surcharge minimum de 0.3MB

TABLE 2 – Comparaison de structures de données en *Java* [ibm]

3 Construction du graphe

3.1 Noeuds

Les noeuds du graphe sont les fragments récupérés précédemment. Il faut considérer les fragments originaux ainsi que leurs complémentaires-inversés. C'est pourquoi pour k fragments, il y a $n = 2 \times k$ noeuds dans le graphe.

3.2 Arcs

Les arcs du graphe représentent la similarité existante entre deux fragments c'est-à-dire entre deux noeuds du graphe. Pour établir ce score, l'application utilise un alignement semi-globale, comme imposé par l'énoncé.

Soit n le nombre de noeuds du graphe, alors m le nombre d'arcs sera :

$$m = n \times (n - 2)$$

Ce résultat a été obtenu grâce à la logique présentée dans le tableau 3.

L'augmentation importante du nombre d'arcs par rapport au nombre de fragments fait penser qu'il s'agit du point le plus lourd pour l'application. Dès lors, il a été particulièrement important de réfléchir à différentes optimisations.

Nombre de fragments k	**Nombre de noeuds n**	**Nombre d'arcs m**
1	2	0
2	4	8
3	6	24
...		
k	$2 \times k$	$n \times (n-2)$

TABLE 3 – Calcul du nombre de noeuds et d'arcs en fonction du nombre de fragments

3.2.1 Optimisation du nombre de tables de similarité à calculer

Sans optimisation, il y a autant de tables de similarité à calculer qu'il y a d'arcs c'est-à-dire $n \times (n \times 2)$. Une première relation a été établie afin de diviser le nombre de ces tables par deux.

Soient deux fragments f et g, la figure 1 représente la table de similarité entre ces deux fragments. Pour obtenir le score de similarité de f vers g, il faut prendre le nombre maximum situé dans la dernière ligne, mise en évidence en bleu. Ensuite, le score de g vers f est obtenu de manière similaire dans la dernière colonne, mise en évidence en rouge.

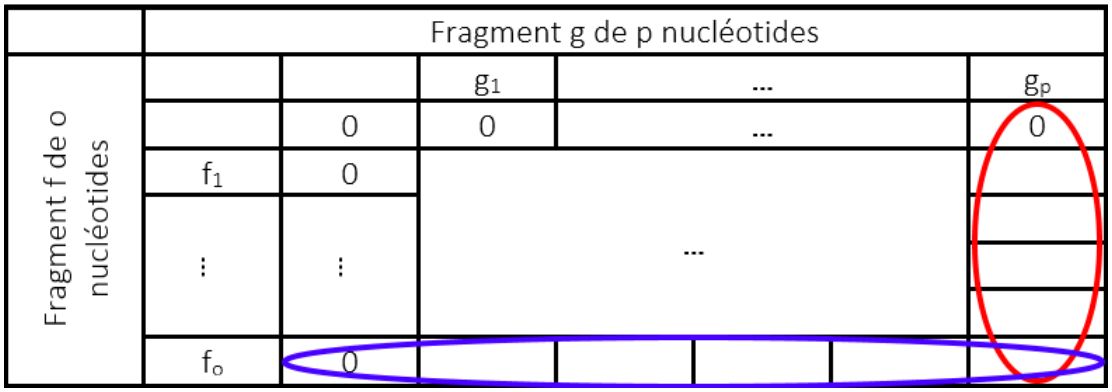

FIGURE 1 – Représentation tableau de similarités

Une seconde relation a été mise en évidence afin de diviser à nouveau le nombre de ces tables par deux :

$$sim(f, g) = sim(\overline{g}, \overline{f})$$

De manière semblable :

$$sim(\overline{f}, g) = sim(\overline{g}, f)$$

Cette relation n'est pas valable dans le cas où l'un des fragments est inclus à l'autre.

En raison de ces deux relations, il est possible d'affirmer que le nombre total de tables de similarités à calculer est de $\frac{m}{4}$. En effet, l'application génère quatre arcs à partir d'une table de similarité.

3.2.2 Optimisation de la mémoire utilisée

Une première approche a été de construire des objets `Edge` contenant un fragment source, un fragment destination et un score. Par contre, l'instanciation d'autant d'objets est très lourd en mémoire. En effet, ceci a provoqué des exceptions de type `OutOfMemory` où la taille maximum du `Heap` de Java était dépassée.

Après réflexion, il a été établi que pour identifier le fragment source et celui de destination, il n'est pas nécessaire d'avoir le pointeur vers l'objet correspondant. En effet, il suffit de connaître leur numéro d'identification ainsi que le fait d'être complémentaire-inversé ou non. En raison de quoi, il a été décidé de remplacer les objets `Edge` par des tableaux d'entiers. Les tables 1 et 2 permettent de confirmer que cette décision semble la bonne du point de vue de la mémoire.

Donc les arcs du graphe sont stockés dans un tableau de tableaux d'entiers. Le tableau 4 permet de visualiser le tableau représentant un arc.

0	**1**	**2**	**3**	**4**
`ID` Source	`Variant` Source	`ID` Destination	`Variant` Destination	Score de similarité

TABLE 4 – Représentation du tableau d'entier représentant un arc

Remarque : Nous sommes conscients qu'il ne s'agit pas d'une approche de programmation orientée objets mais au vu des résultats nous sommes convaincus qu'il s'agit du meilleur compromis auquel nous avons pensé.

3.2.3 Optimisation du temps de calcul

Le calcul des tables de similarité peut être assez long. L'exécution du programme avec le *Java Flight Recorder* activé a permit remarquer que ce calcul prendre plus de 90% du temps d'exécution du programme. Comme vu dans section 3.2.1, la première étape est d'essayer de réduire la quantité de calcul à faire. Une fois cette étape réalisée, il est pertinent de s'intéresser au parallélisme.

Étant donné que calculer la similarité entre deux fragments ne dépend que de ces derniers et qu'il n'y a pas d'effet de bord, il est tout à fait possible d'effectuer plusieurs calculs de similarité simultanément et de remplir le tableau des arcs au fur et à mesure.

Java ne réalise aucun calcul en parallèle par défaut, car un seul *thread* est crée. Le **Stream API** de Java permet de mieux tirer parti des processeurs multi-coeurs. Une fois qu'un *Stream* est créé, il est possible d'activer l'exécution sur plusieurs *threads* avec la fonction `.parallel()`. Java se charge lui-même de répartir chaque calcul du *Stream* sur un *thread* de manière relativement équilibrée.

Un point négatif des *Streams* parallèles est que l'ordre des itérations n'est plus garantie. Ce n'est pas un problème car chaque itération sert à ajouter un arc à une liste qui sera ensuite triée. L'ordre des arcs avant le tri n'a donc pas d'importance.

Suite à ce changement, l'application a été testée sur des processeurs à quatre coeurs, l'utilisation de ceux-ci lors de cette tâche a dépassé les 95% sur chaque coeur. Il s'agit d'une bonne indication que le parallélisme fonctionne correctement.

4 Calcul du chemin Hamiltonien

Comme proposé par la présentation du projet, le calcul du chemin hamiltonien est réalisé par une heuristique gloutonne. En effet, le calcul exact d'un chemin hamiltonien est un problème *NP-Complet*. Étant donné la taille du problème à résoudre, un algorithme d'approximation est nécessaire. Le principe de l'algorithme glouton est d'ajouter chaque paire de nœud selon le poids de leur arc jusqu'à ce que le chemin contienne tous les nœuds.

En pratique, l'algorithme `greedy()`, situé dans la classe `Graph`, se base sur le graphe décrit dans la section 3.

Les variables `in` et `out` sont des tableaux de booléens. Ils permettent de conserver le fait d'être déjà entré ou sorti d'un noeud. L'indice dans ces tableaux correspond au numéro d'identification d'un fragment moins un. Ces tableaux ont autant de cases qu'il y a de fragments donc $n/2$ ou k, comme définis à la section 3.

De plus, il est nécessaire de pouvoir garder en mémoire le lien existant entre un fragment et son complémentaire-inversé. A ces fins, la classe `FragmentPair` a été implémentée. Elle comprend deux fragments, un original et son complémentaire-inversé ou vice-versa, ainsi qu'une variable entière indiquant lequel des fragments est sélectionné. Ce dernier attribut peut prendre trois valeurs : 0 pour aucun fragment sélectionné, 1 pour le premier est sélectionné et 2 pour le second est sélectionné. Le but principal de la classe est d'éviter d'avoir le cas où le chemin rentre dans un fragment puis ressort par son complémentaire-inversé, problème qui a été rencontré lors d'une version intermédiaire. Deux méthodes principales y sont implémentées :

— `use()` permet d'utiliser le premier ou le second fragment
— `getSelected()` renvoie l'identifiant pour le fragment qui a été sélectionné, celui-ci est identifié de manière similaire que dans la section 3.2.2.

En outre, il a été également nécessaire de représenter les ensembles de nœuds, c'est pourquoi, la classe `GraphSet` a été créée. Cette classe est juste une encapsulation par dessus un objet *LinkedHashSet*. Cette encapsulation permet de définir des opérations utiles telles que l'union d'ensembles ou la recherche d'ensembles.

La figure 2 donne une bonne idée des étapes de l'algorithme :

1. Initialisation : Chaque paire de fragments est seule dans un ensemble
2. Sélection de l'arc c vers $\overline{a}$: Les deux ensembles correspondants sont unis et les c et $\overline{a}$ sont marqués.
3. Sélection de l'arc b vers d : Les deux ensembles correspondants sont unis et les b et d sont marqués.
4. Sélection de l'arc $\overline{a}$ vers b : Les deux ensembles correspondants sont unis. Il s'agit

de la fin de l'algorithme car il ne reste qu'un ensemble.

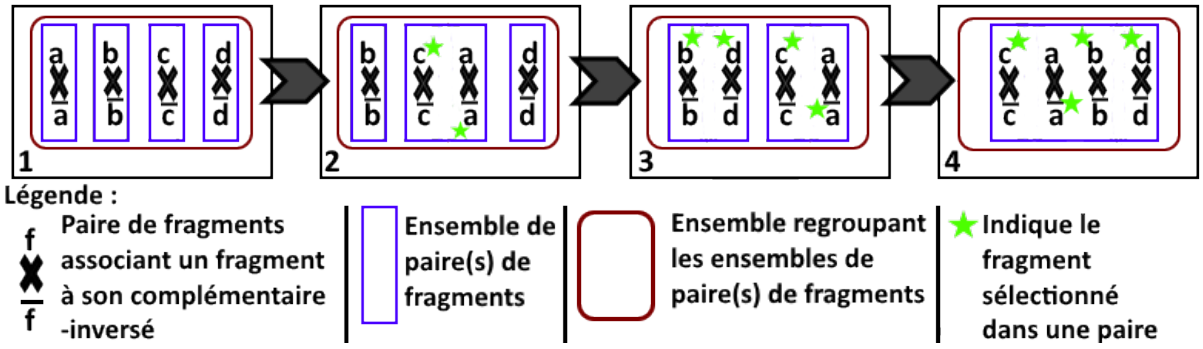

FIGURE 2 – Fonctionnement de l'algorithme `greedy`

4.1 Optimisations réalisées

Étant donné que le graphe enregistre seulement l'ID du fragment ainsi que s'il est complémentaire inversé ou non, une **table de hachage** `HashTable<Integer, FragmentPair> pairsDico` a été ajoutée afin d'obtenir la paire associée à un fragment et son complémentaire inversé avec une complexité moyenne en $\mathcal{O}(1)$.

Le `GraphSet` est implémenté en utilisant un `LinkedHashSet`. En effet, l'ordre d'ajout des éléments dans le `GraphSet` est très important, vu qu'il contiendra l'ordre final retourné par l'algorithme. Il faut donc également garder l'ordre lors d'une union de `GraphSet`. Il est donc nécessaire d'utiliser une collection qui garde l'ordre d'ajout des éléments, telle qu'une liste. Cependant, vérifier si une paire est dans le `GraphSet` ou non est une opération très courante. Étant donné que la liste ne peut pas être triée, la recherche devrait se faire en $\mathcal{O}(n)$ où n est la taille de la liste. Pour améliorer la recherche, un `LinkedHashSet` est utilisé car il combine l'ordre d'arrivée d'une liste chaînée, avec une opération `contains` en $\mathcal{O}(1)$, comme dans un `HashSet` (grâce à une table de hachage des paires). L'ajout d'un élément dans le `LinkedHashSet` reste en $\mathcal{O}(1)$ en moyenne car il suffit d'ajouter un nœud en fin de liste et une entrée dans la table de hachage, en supposant qu'il n'y a pas de collision.

5 Construction des alignements

De l'algorithme décrit à la section 4 résulte une liste de fragments dans un ordre précis. De celle-ci, 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, ...

A partir de cette table il faut reconstruire l'alignement des deux fragments correspondants. Dans la classe `Similarity`, qui a déjà été utilisée précédemment, il y a deux tableaux d'entiers. Le premier contient les scores et est utile pour la construction des arcs du graphe, voir la section 3.2. Quant au second tableau, il contient les directions, c'est-à-dire pour chaque case la direction qui a permis de l'atteindre. La direction est un entier, 0 pour diagonale, 1 pour à gauche et 2 pour à droite.

Il est, par ailleurs, maintenant possible de reconstruire l'alignement de deux fragments.

Phase d'initialisation :
Plusieurs variables sont nécessaires :

- Deux piles, une pour récupérer les nucléotides de la source et une pour ceux de la destination.
- L'objet `Similarity` correspondant aux deux fragments
- Deux indices, i et j, pour suivre l'évolution dans le fragment source et celui de destination. Ils sont initialisés pour pointer sur le dernier nucléotide du fragment respectif.
- Deux entiers pour tenir compte du décalage entre les fragments. Ils sont initialisés à 0.

La dernière étape d'initialisation est de récupérer les nucléotides finaux du fragment destination et de les placer dans la pile correspondante.

Phase d'itération :
Cette phase est à répéter tant que i et j sont plus grands ou égaux à 0. C'est ici que la table contenant les directions a tous son intérêt pour ne pas à devoir refaire des comparaisons et des calculs. La direction courante est récupérée dans le tableau et l'une des trois actions suivantes est effectuée en fonction de sa valeur :

- En diagonale : Récupérer le nucléotide à la position i dans le fragment source et le placer dans la pile correspondante. Faire de même avec la destination et j. i et j sont décrémentés. Cette situation correspond à un *match* ou un *mismatch* entre les nucléotides.
- A gauche : Récupérer le nucléotide à la position j dans le fragment de destination et le placer dans la pile correspondante. Placer un '-' dans la pile correspondant à la source. Seul j est décrémenté. Cette situation correspond à un *gap* dans le fragment source.
- En haut : Récupérer le nucléotide à la position i dans le fragment de la source et le placer dans la pile correspondante. Placer un '-' dans la pile correspondant à

la destination. Seul i est décrémenté. Cette situation correspond à un *gap* dans le fragment destination.

Phase finale :
La précédente phase s'est achevée car soit i soit j vaut 0, ce qui signifie que potentiellement l'un des deux ne vaut pas 0. Il faut donc récupérer les nucléotides restants dans la source ou la destination. C'est pourquoi deux nouvelles boucles ont été rajoutées à ce niveau là.

— Tant que i ne vaut pas 0 : Récupérer les nucléotides de la source et les placer dans la pile correspondante, décrémenter i et incrémenter le décalage correspondant à la destination.

— Tant que j n'est pas égal à 0 : Récupérer les nucléotides de la destination et les placer dans la pile correspondante, décrémenter j et incrémenter le décalage correspondant à la source.

5.1 Optimisations et améliorations effectuées

Dans un premier temps, le décalage existant entre deux fragments était représenté à l'aide d'un nucléotide neutre, représenté par N. Or cette façon de procéder est vite devenue trop lourde au moment de la propagation des *gaps*, voir la section 6. C'est pourquoi, le décalage a tout simplement était ajouté à l'objet `Fragment`. Ce qui s'avère en effet beaucoup moins lourd pour le système et aussi intuitif à l'utilisation quoique peut-être un peu plus dur à déployer.

Ensuite, pour construire les fragments alignés, la décision a été prise d'utiliser des piles. Cette structure de données présente plusieurs avantages dans ce cas. En effet, elle permet d'inverser les nucléotides de manière transparentes car la première nucléotide poussée dans la pile est toujours la dernière nucléotide du fragment. De manière analogue, la dernière nucléotide placée dans la pile est la première de la séquence résultante. Donc, au moment de reconstruire la séquence du fragment, les nucléotides sont retirés de la pile directement dans le bon ordre.

6 Report des *Gaps*

A la fin de la phase d'alignement, l'application a à sa disposition une liste ordonnée selon le chemin hamiltonien d'alignement effectué entre deux fragments. Dans cette liste, il faut propager les gaps et les décalages à l'ensemble des fragments. Ceci se fait en deux itérations : du début de la liste vers la fin (du haut vers le bas) puis dans l'autre sens (du bas vers le haut). Dans les deux cas la même fonction est utilisée afin d'éviter la duplication de code.

`reportGap` prend trois fragments et un booléen en arguments. Le premier fragment est celui dont il faut copier les *gaps* et le décalage. Le second est le même fragment mais qui avait été aligné à l'étape précédente avec un autre fragment, le troisième. Le booléen permet d'indiquer à la méthode si elle est appelée en descendant ou en montant car le décalage ne doit être propagé qu'en descendant.

Phase d'initialisation
Soient f_1, le fragment à copier, f_1' et f_2 les fragments dans lesquels il faut propager les *gaps*.

- Récupérer les décalages de f_1' et de f_2, y propager celui de f_1 dans le cas où la fonction est appelée en descendant la liste.
- Initier trois pointeurs, i, j et k, à 0, un pour chaque fragment
- Initier deux *StringBuilder* afin de construire les nouvelles séquences pour f_1' et f_2.

Pour finir la phase d'initialisation, il faut prendre en considération la situation où un décalage est présent dans f_1' ou dans f_2. Alors il est nécessaire récupérer les nucléotides du fragment n'ayant pas de décalage avant de passer à la suite.

Phase d'itération
Il faut boucler tant que i est inférieur à la taille de f_1, que j est inférieur à la taille de f_1' et que k est inférieur à la taille de f_2. Le processus de décision suit les étapes suivantes : Est-ce un *Gap* à l'indice i de f_1 ?

- Oui : Est-ce également un *Gap* à l'indice j de f_1' ?
  - Oui : Récupérer les nucléotides de f_1' à l'indice j et de f_2 à l'indice k puis les ajouter au *builder* correspondant et incrémenter i, j et k.
  - Non : Ajouter un *gap* dans les deux *builder* et incrémenter i.
- Non : Récupérer les nucléotides de f_1' à l'indice j et de f_2 à l'indice k puis les ajouter aux *builder*s correspondant et incrémenter j et k. Par contre, n'incrémenter i que s'il n'y a pas de *gap* à l'indice j de f_1.

Phase finale
Il faut récupérer les derniers nucléotides présents soit dans f_1 et f_1' soit dans f_2. La fonction retourne une paire de fragments instanciés avec les *builder*s et les décalages obtenus.

7 Consensus

À la fin du report des *gaps*, l'application a à sa disposition une liste des fragments alignés et ordonnés selon le chemin hamiltonien. Comme vu dans le chapitre 6, les fragments ont maintenant les *gaps* nécessaires au consensus et le décalage par rapport au début de la chaîne de nucléotides. Ensuite, pour chaque indice i du fragment final, il faut effectuer un vote à la majorité avec les nucléotides correspondant de chaque fragment.

D'abord, la position d'un nucléotide au sein d'un fragment est calculé par rapport à l'indice i. En effet, l'indice i correspond à la position courante dans la séquence finale. C'est pourquoi, il faut le convertir pour pouvoir récupérer le nucléotide au sein de la séquence du fragment considéré :

$$Position(Nuclotide) = i - \texttt{shift} \text{ du fragment courant}$$

De plus, il faut veiller à ne pas essayer de récupérer un nucléotide alors que la séquence correspondante est finie. C'est pourquoi la relation suivante doit être respectée :

Si ($i <$ `shift` ou $i \geq$ `shift` + Taille du fragment) alors il n'y a pas de nucléotide dans le fragment correspondant à l'indice i.

Maintenant qu'il est possible de récupérer tous les nucléotides à un indice donné de la séquence, il faut choisir le plus fréquent. Tous les nucléotides à l'indice i sont ajoutés à une pile (`Stack`) en $\mathcal{O}(1)$. Une fois que toute la liste des fragments a été parcourue, il suffit de dépiler toute la pile tout en enregistrant le nombre d'occurrence de chaque nucléotide dans un `HashMap`. Enfin, la clé de cette `HashMap` correspondant à la plus grande des valeurs est le résultat du vote.

Quand le nucléotide majoritaire pour l'itération i a été trouvé, il est ajouté à la séquence solution et l'itération est répété pour $i + 1$. Cet algorithme final s'arrête quand tous les indices i possibles ont été parcourus, c'est-à-dire de $i = 0$ jusqu'à $i =$ last.length() + last.shift() où last est le dernier élément de la liste des fragments alignés.

```
A T A G T A
      A G - A C - A T C C
        G - A C T A T - G      ⟹ A T A G - A C T A T - C
      T A - A C T A T - -
```

Étant donné que cette implémentation du consensus prend un temps d'exécution négligeable par rapport à la construction du graphe et que chaque opération a une complexité raisonnable, il n'a pas semblé nécessaire d'optimiser les performances de cette partie du logiciel.

8 Résultats obtenus

Pour l'ensemble des résultats obtenus par l'application, les graphiques représentent la correspondance existante entre la séquence obtenue à la fin du programme et la séquence cible. L'axe des abscisses équivaut à la séquence cible et l'axe des ordonnées à la séquence obtenue par le programme.

Quand les échantillons fournis le permettaient, l'application a été testée sur la version simplifiée puis sur la version dite normale. Pour ce dernier cas, deux résultats sont fournis : le premier pour la séquence originale renvoyée à la fin du programme et le second pour son complémentaire inversé.

8.1 Collection 1

Version simplifiée

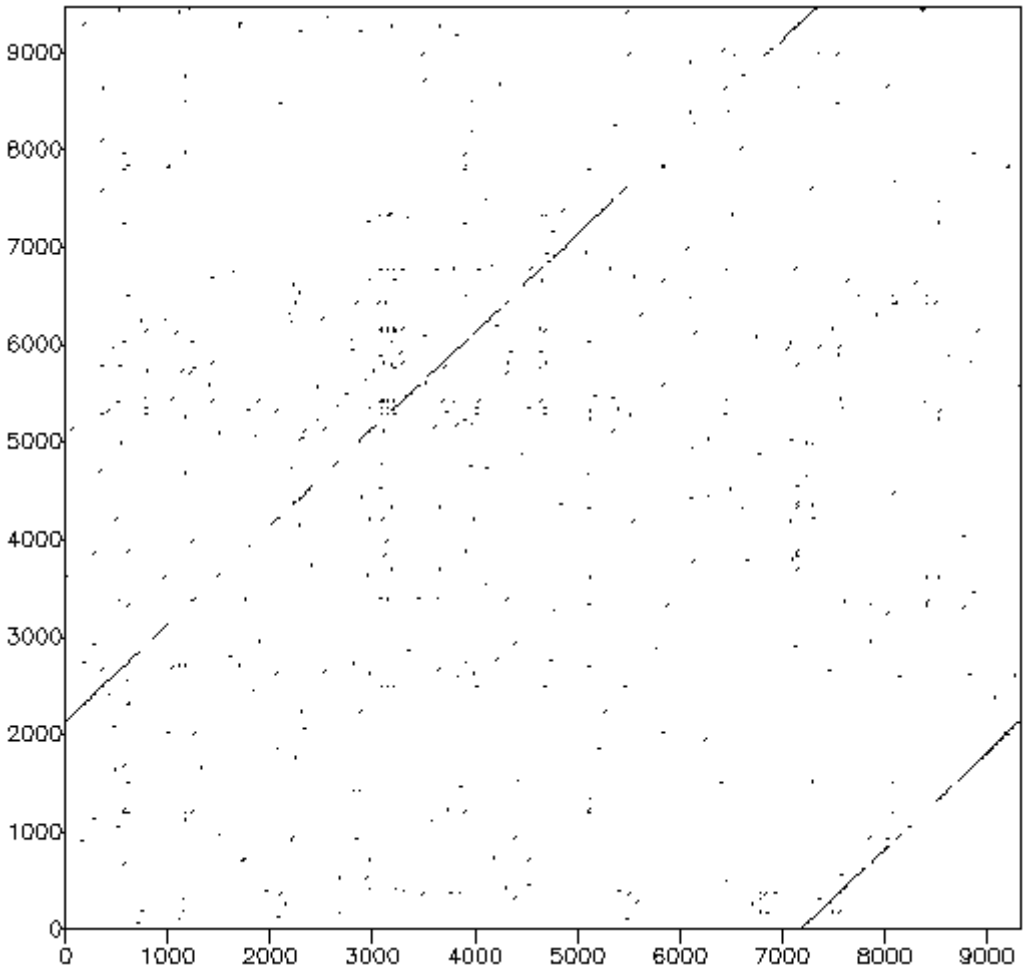

FIGURE 3 – Collection 1 version simplifiée

Le résultat obtenu à la figure 3 montre une diagonale assez complète. Ce qui signifie qu'il y a une bonne correspondance entre la séquence obtenue par l'application et la séquence cible. Il faut toutefois nuancer car certains pans de la séquence n'ont pas été récupérés, il s'agit des 'trous' de la diagonale. De plus, la séquence n'a pas été récupérée dans le bon ordre, le morceau de diagonale en bas à droite devrait être à gauche et l'autre pan devrait plus à droite.

Version normale

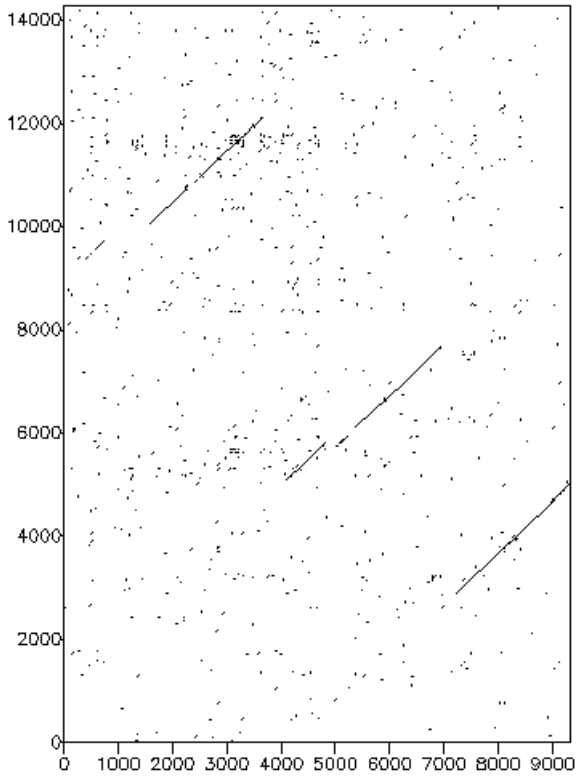

FIGURE 4 – Collection 1

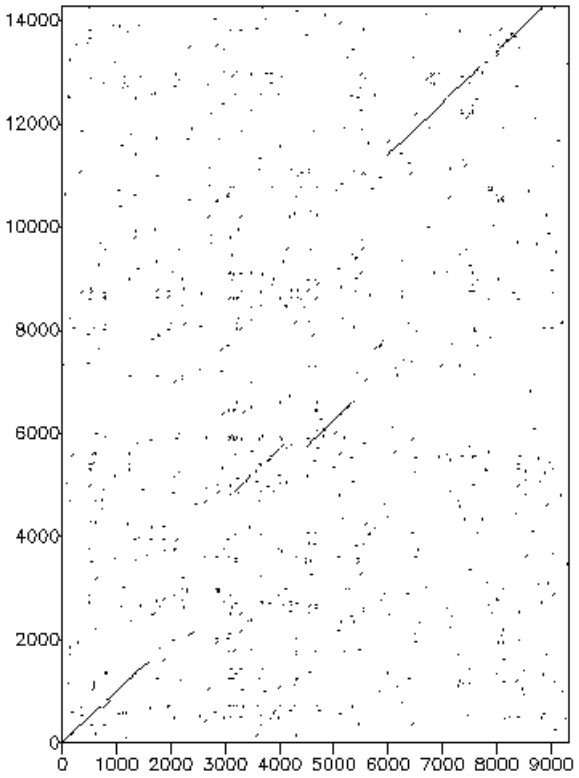

FIGURE 5 – Collection 1 IC

Dans cette sous-section, les résultats correspondant à la version dite normale de la collection 1 sont présentés. La figure 4 montre la correspondance entre la séquence obtenue et la séquence cible. De plus, la figure 5 montre la correspondance entre le complémentaire inversé de la séquence obtenue et la séquence cible. Dans les deux cas, la séquence obtenue est plus longue que la séquence cible ($\pm 33\%$)

La figure 4 montre qu'il y a une assez bonne couverture de la séquence cible même si les morceaux de diagonales ne sont pas dans le bon ordre. Près de trois quarts de la cible a été récupéré par l'application, ce qui représente un résultat correct.

La figure 5 fait penser que la cible a été récupérée dans un meilleur ordre mais elle est un peu moins bien couverte.

8.2 Collection 2

Version simplifiée

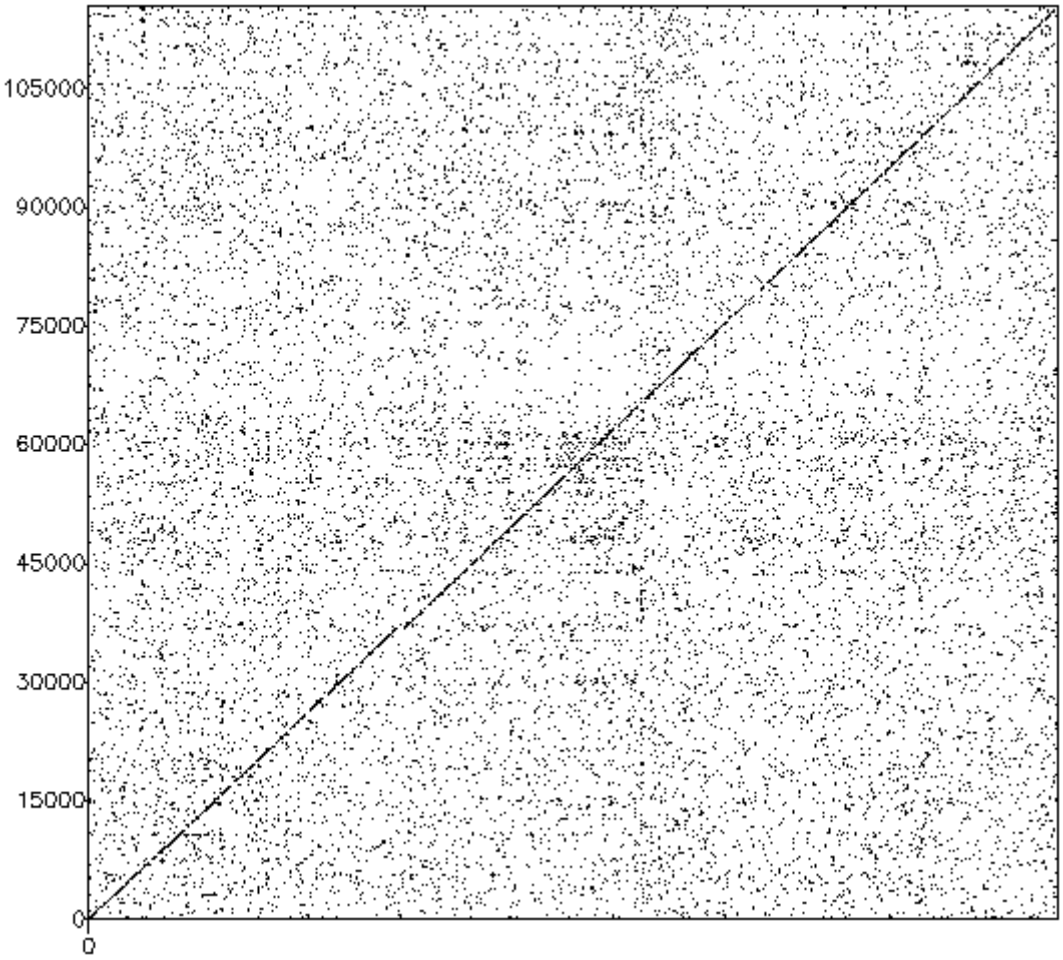

FIGURE 6 – Collection 2 version simplifiée

Le résultat obtenu pour la collection 2 en version simplifiée est visible à la figure 6. La diagonale est presque parfaite, en effet elle commence au coin inférieur gauche et finit au coin supérieur droite. De plus, la droite est presque totalement complète. Ces constations permettent d'affirmer que la couverture de la séquence cible est particulièrement bonne.

Version normale

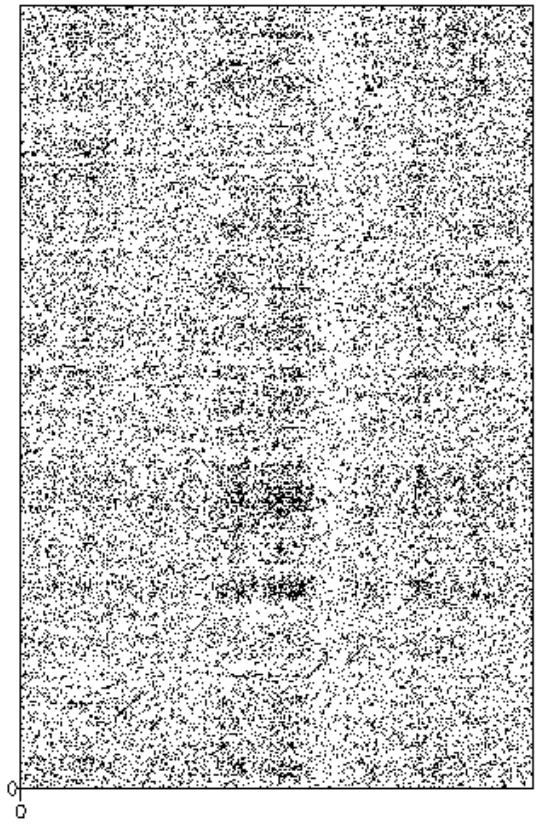

FIGURE 7 – Collection 2

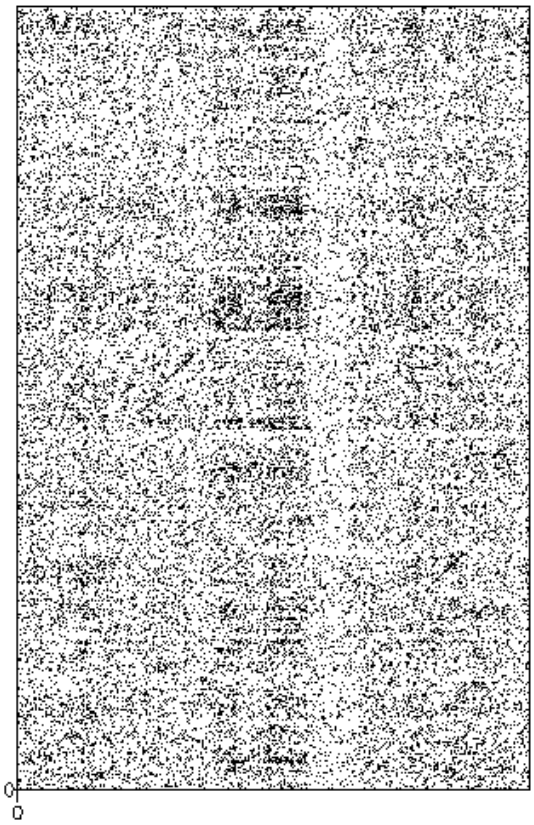

FIGURE 8 – Collection 2 IC

Les résultats pour la collection 2 normale, présentés aux figures 7 et 8, ne sont pas très bons car aucune diagonale ne ressort. Il n'a hélas pas été possible d'établir la cause de ce problème. En effet, plusieurs des algorithmes implémentés dans l'application peuvent en être à la base. Comme l'algorithme glouton qui fournit qu'un chemin hamiltonien parmi d'autres. Il y a aussi le vote à la majorité qui, en cas d'égalité, renvoie le premier caractère correspondant à la valeur maximum. D'autres algorithmes peuvent aussi être impliqués.

8.3 Collection 3

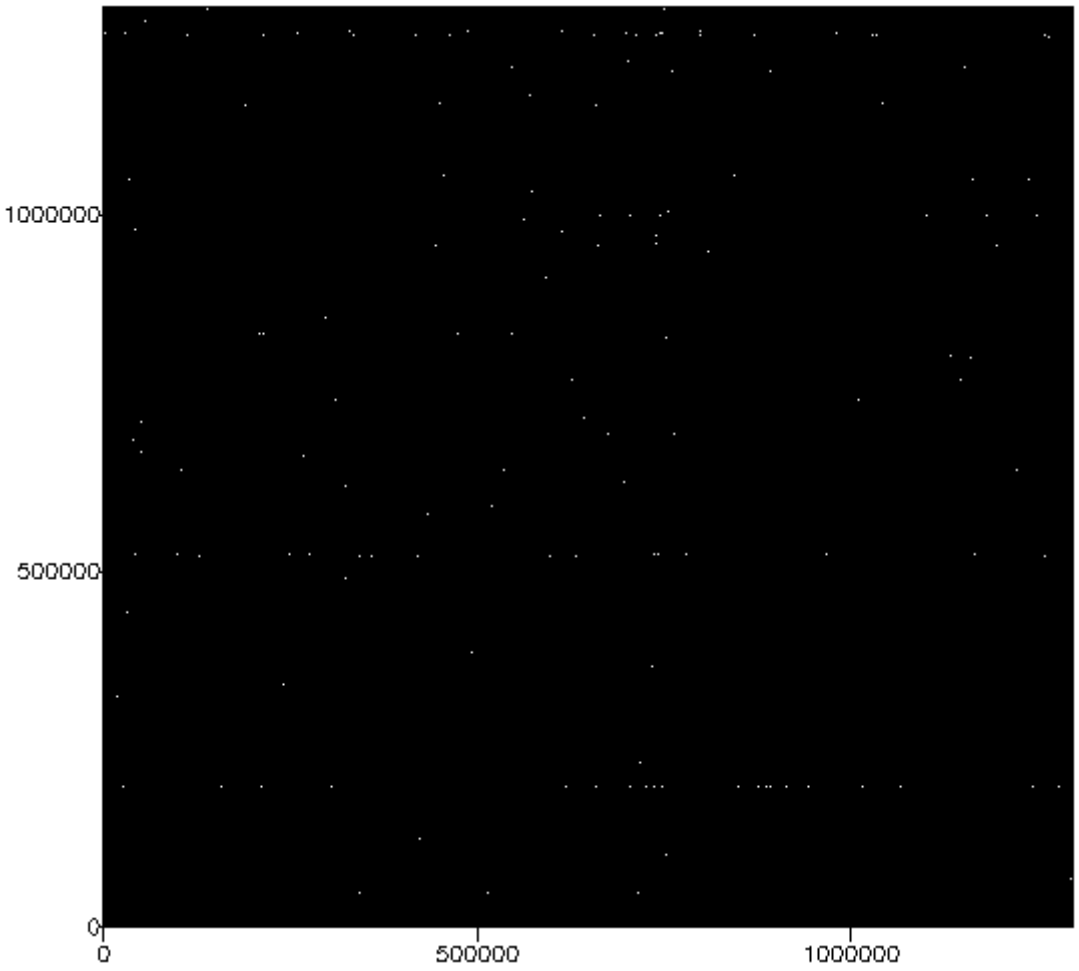

FIGURE 9 – Collection 3 version simplifiée

Le programme a permis d'obtenir un résultat pour la collection 3. Par contre, le graphique présenté à la figure 9 ne permet pas de tirer des conclusions claires. Afin de pouvoir mieux visualiser les résultats, il faudrait peut-être pouvoir afficher moins de points ou diminuer le nombres de nucléotides considérés.

Remarque : Le résultat proposé ici correspond à un résultat obtenu lors d'une version intermédiaire qui incluait les *gaps* dans la séquence. Étant donné le temps nécessaire pour obtenir ce résultat, il n'a pas été recalculé pour la version finale.

8.4 Collection 4

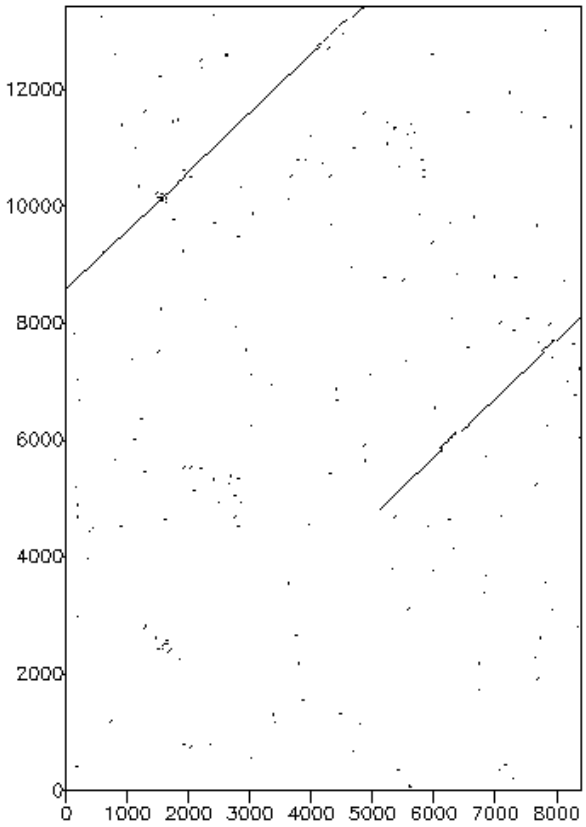

FIGURE 10 – Collection 4

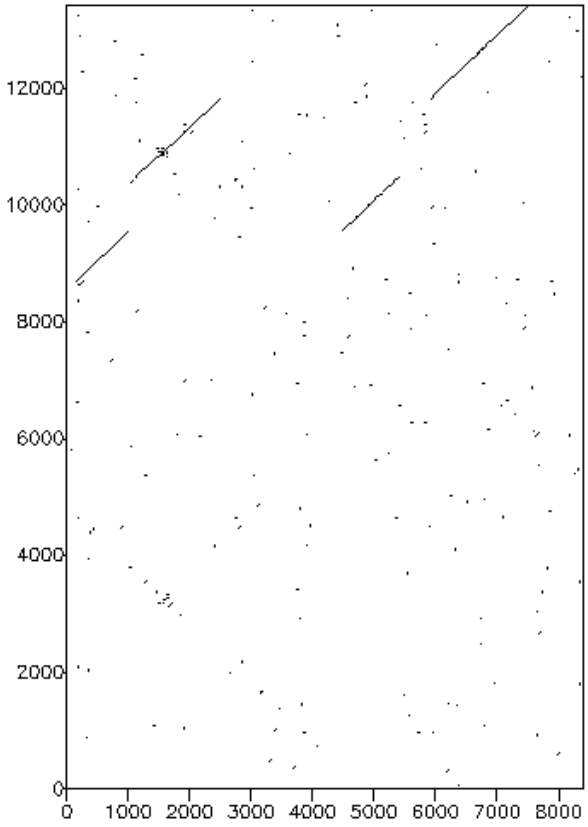

FIGURE 11 – Collection 4 IC

La séquence résultante de l'application est nettement plus longue que la séquence cible dans ce cas.

La figure 10 montre que la séquence obtenue couvre presque la totalité de la séquence cible. Par contre, elle n'est pas dans le bon ordre. De plus, un peu moins de la moitié des nucléotides de la séquence obtenue ne corresponde à rien dans la séquence cible.

Avec la figure 11, il est possible de constater que la couverture de la séquence cible est moins bonne qu'avec la séquence originale fournie par l'application. De plus, près de deux tiers des nucléotides de la séquence ne correspondent un rien dans la cible.

8.5 Collection 5

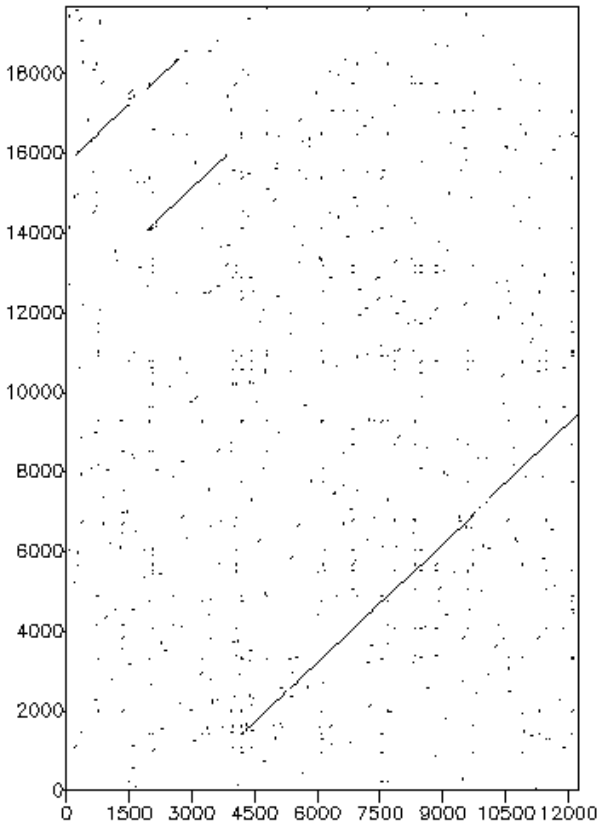

FIGURE 12 – Collection 5

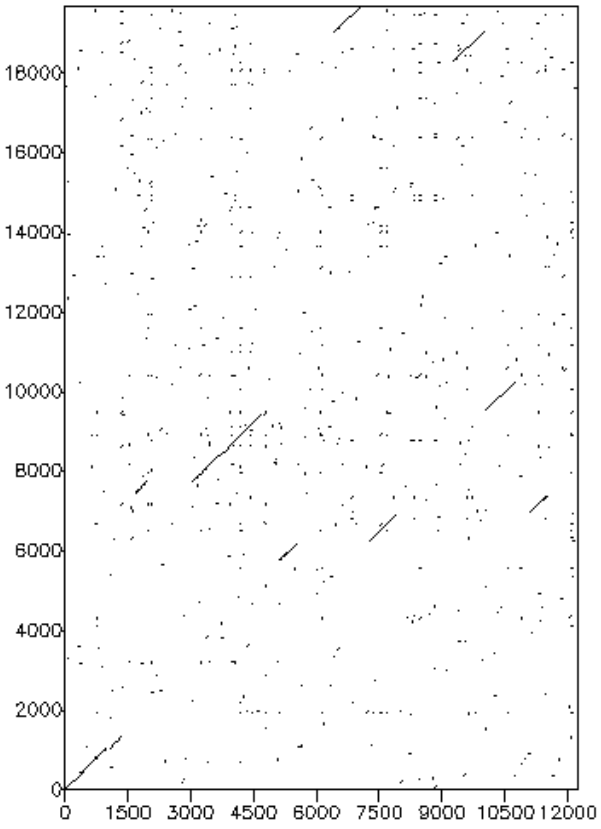

FIGURE 13 – Collection 5 IC

De nouveau, les graphiques présentés en figures 12 et 13 montrent que la séquence obtenue par le programme est plus longue que celle de la cible.

La figure 12 montre une couverture correcte mais désordonnée de la cible.

Par contre, la figure 13 est nettement moins pertinente. En effet, elle est composé de beaucoup de petites diagonales éparses sur le graphique.

9 Remarques

9.1 Exécution

Ce programme a été écrit en utilisant des nouveautés de Java 14, notamment certaines qui sont encore en *Preview*. Cela inclut les `record`, qui sont une implémentation en Java des *dataclasses*, les *Text Blocks* ainsi que une nouvelle syntaxe pour les `switch`, qui est très lisible et intuitive. Aucune de ces fonctionnalités n'est essentielle, le programme pourrait facilement être traduit en Java 8.

Pour lancer le programme, il faut donc utiliser un **JRE 14** ainsi que ajouter l'option `-enable-preview` lors de l'exécution avec la machine virtuelle `java`.

9.2 Points forts

Les calculs d'alignements se font en un temps raisonnable. En effet, le tableau 5 présente les temps approximatifs nécessaire pour chaque collection. Les résultats ont été obtenus en utilisant un ordinateur portable de la marque *Dell* ayant un processeur *Intel(R) Core(TM) i7-10510U @ 1.80GHz 2.30GHz* avec *16.0 Go* de mémoire RAM installée, dont *15.8 Go* utilisable. Il fonctionne sous *Windows 10* et les tests ont été réalisés via *IntelliJ* avec *azul-14 java version "14.0.2"*.

Collection	**Version**	**Temps d'exécution**
1	Simplifiée	$< 1min$
1	Normale	$< 1min$
2	Simplifiée	$< 1min$
2	Normale	$\pm 30min$
3	Simplifiée	$< 12heures$
4	Normale	$< 5min$
5	Normale	$< 5min$

TABLE 5 – Temps approximatif mesuré lors de l'exécution du programme

Remarque : Les temps présentés dans le tableau 5 n'ont pas été mesurés de manière formelle.

De plus, étant donné que les tâches ont été réparties de manière individuelles et indépendantes les unes des autres, le programme est relativement modulaire et un algorithme pourrait relativement facilement être remplacé par un autre.

Le programme contient également des tests *JUnit* qui se trouvent dans le module : `be.ac.umons.dnasequencer.tests`.

9.3 Points faibles

Lors de l'ajout du parallélisme, certains tests unitaires (supprimés suite à un *refactoring* du programme) échouent avec une certaine probabilité (environ une fois sur cinq). En effet, plusieurs exécutions successives ont montrées que les arcs du graphe n'étaient pas toujours générés dans leurs totalités. Il semblerait que le parallélisme entraîne une mauvaise évaluation de la condition permettant de construire ou non un arc entre deux noeuds.

De plus, les résultats obtenus ne sont pas parfaits et pourraient être améliorés. Comme vu dans la section 8, les résultats sont assez variables, allant de la diagonale presque complète au graphique noir tellement il y a de points.

9.4 Répartition des tâches

Laurence Floriani	**Thomas Lavend'Homme**
Lecture et écriture des fichiers	Alignement semi-global
Graphe	Report de gaps
Algorithme glouton	Consensus

Cette répartition est approximative, en réalité les deux parties prenantes se sont beaucoup entre-aidées lors de leurs parties respectives. De plus, les points importants du projet ont été réfléchis ensemble afin de mettre en commun les idées et d'amener à une réflexion plus complète.

10 Conclusion

Le premier et principal challenge de ce projet a été de gérer la mémoire du *Heap* de Java. En effet, il s'est avéré vite dépassé par la quantité importante d'éléments à calculer. Pour relever ce défi, nous avons choisi une approche non orientée objet pour certaines parties de l'application. C'est pourquoi, si le projet était à réimplémenter, notre choix se porterait vers des langages comme Go ou C++. En effet, comme vu lors de la section 3.2.2, utiliser une structure de donnée qui n'est pas un objet mais qui se rapproche d'une `struct` en C, nous a permis de diminuer la consommation de mémoire dans le *heap*.

Le second challenge, et pas des moindres, a été d'utiliser efficacement les *threads* à notre disposition. Les calculs à réaliser étant particulièrement nombreux, il nous a semblé inenvisageable de les réaliser à l'aide d'un seul *thread*. Afin de répondre à ce challenge, nous

avons choisi d'utiliser la *Stream API* de Java. Elle présente la possibilité de paralléliser les calculs de manière assez facile. En observant le moniteur de ressources, nous avons constatés que près de 100% de chaque coeur est utilisé. Il nous aurait été difficile d'obtenir d'aussi bons rendements en implémentant nous-même le *multi-threading*.

A part les challenges, nous avons essayé d'avoir une approche proche de la programmation fonctionnelle, avec le autant d'objets immuables que possible. Cette manière de procéder permet d'éviter les effets de bords "incontrôlés" qui nous ont déjà posés des problèmes lors de projets précédents. Nous comptions donc beaucoup utiliser la *Stream API* de Java pour avoir une meilleure lisibilité. Malheureusement, cette API ne fonctionne bien qu'avec des `Objects`, et pas avec des types primitifs et des tableaux. Comme, pour des raisons de mémoire et de performances, nous utilisons beaucoup les types primitifs, nous avons du abandonner cette approche en faveur d'une programmation Java un peu plus classique.

En outre, notre implémentation propose de bons résultats sur une partie des échantillons. Les problèmes de non couverture et d'ordre semblent s'accentuer lorsque la taille de la séquence à calculer augmente. Pour expliquer ces problèmes, plusieurs hypothèses peuvent être envisagées :

Au niveau de la construction de la table de similarité entre deux fragments, nous avons établi un ordre de priorité dans les déplacements. En cas d'égalité entre les scores, l'algorithme choisira en premier un *match* ou un *mismatch* puis un *gap* dans la destination et à défaut un *gap* dans la source.

Ensuite, pour construire un chemin hamiltonien dans le graphe, l'algorithme glouton fournit un chemin parmi d'autres. L'utilisation d'une métaheuristique à la place ou en complément pourrait permettre d'améliorer les résultats.

Un autre exemple de point pouvant être impliqué est le vote à la majorité car, lors du consensus, en cas d'égalité renvoie le caractère correspondant au premier maximum rencontré.

Enfin, nous sommes satisfaits du temps d'exécution de l'algorithme ainsi que de la gestion de la mémoire.

Références

[ibm] From java code to java heap. `https://www.ibm.com/developerworks/library/j-codetoheap/index.html`. Visité le : 2020-12-13.