Pierre TRETON

Compte Rendu :
Projet assembleur naïf

Introduction

La raison pour laquelle je fais ce projet en python est que cela me permet d'apprendre à programmer dans ce langage, en plus de pouvoir utiliser tout un tas de fonctionnalités qui simplifient grandement l'écriture du code.

Python offre la possibilité de comparer les chaînes avec l'opérateur « = », comme java le fait avec la méthode *equals()*. J'ai utilisé cet opérateur pour simplifier le code mais la complexité de cette opération entre dans l'équation finale. On a ainsi pour deux chaînes de taille n une complexité en O(n) dans le pire cas. Cela consisterait en une boucle qui compare 1 à 1 les caractères (dans un seul sens ou dans les deux sens de la chaîne).

Explications de l'algorithme

Tant que la liste de séquences passée en paramètres est de longueur supérieure à 1 :

- On cherche le nombre de caractères en commun (*cover*) entre la première séquence de la liste (*main_seq*) et chacune des autres séquences en parcourant toutes les séquences de l'indice 1 (seconde séquence dans la liste) jusqu'au dernier élément.

  L'idée est de trouver le chevauchement le plus grand entre la première séquence de la liste (prise arbitrairement) et toutes les autres séquences afin d'assembler ces deux chaînes par la suite.

  Pour simplifier on dira que la séquence principale (la première de la liste) est **A** et la séquence pointée par l'indice dans la liste est **B**.

  Afin de différencier un chevauchement au début **A** ou à la fin, on met un signe au chevauchement : un chevauchement négatif correspond à la fin de **B** qui chevauche le début de **A**, un chevauchement négatif correspond au début de **B** qui chevauche la fin de **A**.

  - On teste premièrement le chevauchement de **A** et **B** en comparant le début de **A** avec la fin de **B**. On part d'un chevauchement négatif de **- (taille mini entre A et B) + 1** :

    On slice les deux chaînes de manière à obtenir deux chaînes de même taille (chevauchement), **A** et **B** slicées respectivement à partir du début et de la fin.

    Tant que le chevauchement est différent de 0 et que **A** slicé ne chevauche pas **B** slicé :
    On diminue la valeur du chevauchement **(-chevauchement -> -chevauchement + 1)** puis on réduit (slice) **A** et **B** au nouveau chevauchement .

Si le chevauchement est différent de 0 et est supérieur au chevauchement maximum entre **A** et le dernier **B** comparé (par défaut ce max est 0) :
On sauvegarde le chevauchement et l'indice de **B**.

- De la même manière, on teste le chevauchement de **A** et **B** en comparant le début de **B** avec la fin de **A**. On part d'un chevauchement positif cette fois de **(taille mini entre A et B) - 1** :

  On slice les deux chaînes de manière à obtenir deux chaînes de même taille (chevauchement), **A** et **B** slicées respectivement à partir du début et de la fin.

  Tant que le chevauchement est différent de 0 et que **A** slicé ne chevauche pas **B** slicé :
  On diminue la valeur du chevauchement **(chevauchement -> chevauchement - 1)** puis on réduit (slice) **A** et **B** au nouveau chevauchement .

  Si le chevauchement est différent de 0 et est supérieur au chevauchement maximum entre **A** et le dernier **B** comparé (par défaut ce max est 0) :
  On sauvegarde le chevauchement et l'indice de **B**.

- Afin de réduire le temps de calcul, je pars du principe que si l'on trouve un chevauchement de **- (taille mini entre A et B) + 1** ou de **(taille mini entre A et B) – 1**, on n'est pas obligé de continuer à comparer les autres chaînes de la liste car le chevauchement maximal est bien de **|(taille mini entre A et B) – 1|**, de fait :

  Si le chevauchement est égal à **|(taille mini entre A et B) – 1|** :
  On sort de la boucle *for*.

- A la suite de la boucle *for*, on construit la chaîne à partir de la séquence à l'indice 0, de la séquence à l'indice de la chaîne avec chevauchement max à l'étape précédente ainsi que de la valeur de chevauchement également récupérée.

  Pour simplifier : **SEQ** = chaîne à l'indice récupéré précédemment (chaîne qui a le chevauchement maximal avec **A** parmi toutes les chaînes de la liste)

  - **Si** le chevauchement est différent de 0 :
    On construit la chaîne différemment en fonction du signe du chevauchement :
    - Négatif : début de **SEQ** avec **A** entière.
    - Positif : début de **A** avec **SEQ** entière.

    On remplace la valeur de la première séquence dans la liste (**A**) par la chaîne créée puis on supprime **SEQ** de la liste.

  - **Sinon** :
    Terminaison : On renvoie un message d'erreur car cela signifie qu'au moins qu'il sera impossible d'obtenir une unique chaîne à partir des séquences passées en paramètre.

Pierre TRETON

Exécution détaillée de l'algo d'assemblage

DEBUT :

reset max_overlap : 0 et index : -1

BOUCLE FOR :
comparaison de CTGTA -->
avec <-- ACCTG

CCTG
CTGT

CTG
CTG
overlap = -3

comparaison de <-- CTGTA
avec ACCTG -->

ACCT
TGTA

ACC
GTA

AC
TA

A
A
overlap = 1

> recouvrement (max_overlap) = -3

comparaison de CTGTA -->
avec <-- CCTGT

CTGT
CTGT
overlap = -4

> plus grand recouvrement possible (max_overlap) = -4, sortie de la boucle

FIN BOUCLE FOR

nouvelle séquence = C + CTGTA = CCTGTA

reset max_overlap : 0 et index : -1

BOUCLE FOR :
comparaison de CCTGTA -->
avec <-- ACCTG

CCTG
CCTG
overlap = -4

> plus grand recouvrement possible (max_overlap) = -4, sortie de la boucle

FIN BOUCLE FOR

nouvelle séquence = A + CCTGTA = ACCTGTA

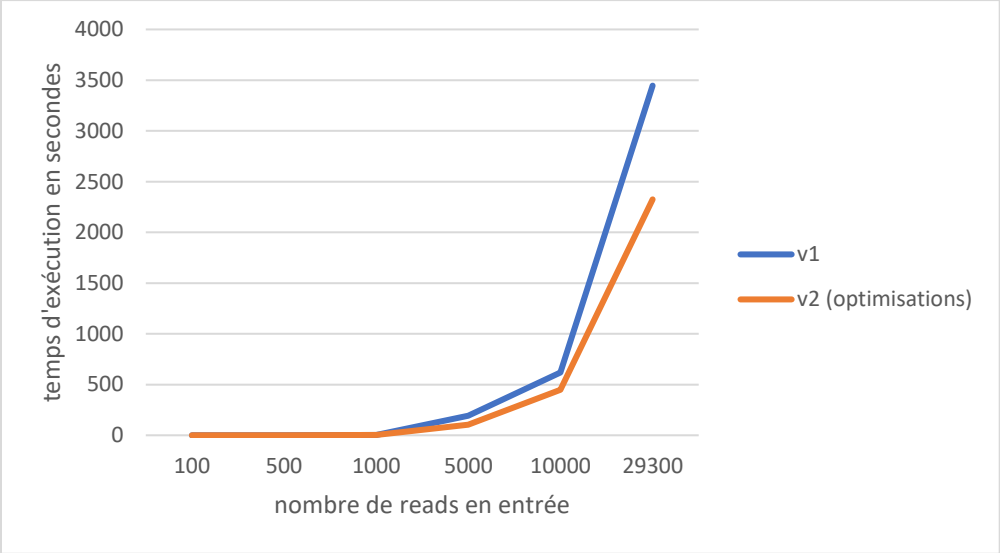
FIN

Pierre TRETON

Etude des performances

Mon algorithme met presque deux fois plus de temps à s'exécuter que ce qui est indiqué dans l'énoncé sur mon ordinateur, largement moins sur les pc de la fac. Les valeurs de temps relevées pour l'assemblage des différents groupes de bases sont les suivantes (sur les pc de la fac):

On voit que l'optimisation a beaucoup réduit le temps de l'assemblage, en moyenne de 36%.



Optimisations

- Création d'une nouvelle liste de séquences :

Plutôt que de retirer à chaque comparaison le read qui a le plus grand recouvrement, l'idée est d'ajouter à une nouvelle liste les éléments qui n'ont pas le plus grand recouvrement. En faisant cela, on évite d'utiliser la fonction *pop()* qui a une complexité de l'ordre de O(n) pour une approche moins complexe (mais plus coûteuse en mémoire).

- Ajout simultané à l'avant et à l'arrière de la chaîne :

L'idée est de trouver la meilleure chaîne à l'avant de la chaîne principale ainsi que la meilleure à placer à la suite. En théorie, on devrait pouvoir réduire le nombre de comparaisons par 2 dans le meilleur des cas.

En pratique, l'algorithme est moins efficace car il se pose un problème de décision, ajouter ou non la séquence. On peut imaginer qu'une séquence n'ayant pas le chevauchement maximal (c'est-à-dire égal à sa longueur – 1), cette séquence pourrait ne pas être placée à la bonne place si on l'ajoute trop top à la séquence principale.

J'ai fais des tests en limitant l'ajout des chaîne uniquement si le recouvrement est de la moitié de sa taille mais sur le génome E.Coli utilisé pour faire des tests de bon fonctionnement de l'algo, l'assembleur fait beaucoup d'erreurs. Ce n'est pas une solution fiable.

Pierre TRETON

- Utilisation d'un graphe de Bruijn

Sources utilisées :

https://dridk.me/assemblage.html

J'ai implémenté un assembleur basé sur l'utilisation de graphes + table de hachage.

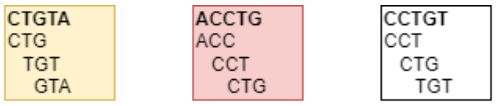
L'assemblage du graphe est très efficace car sa complexité est en O(n), n étant le nombre de nœuds, c'est-à-dire les séquences kmers – 1.

Pour stocker les nœuds et garantir leur unicité, j'utilise un dict python dont la clé est le kmer – 1 lui-même. Les opérations sur les tables de hachage sont très peu coûteuse, ce qui rend la création du graphe assez efficace.

Complexité des opérations dans dict : https://wiki.python.org/moin/TimeComplexity

L'algorithme de création du graphe fonctionne de cette manière :

- Pour chaque read : on extrait tous les k-mers pour k donné (étant entre la taille du read divisée par 2 et la taille du read – 1)
- Pour chaque k-mer, on l'ajoute au graphe :
  On ajoute alors un nœud de contenant les k premiers caractères et un autre nœud contenant les k derniers. Chaque nœud a connaissance de ses prédécesseurs et successeurs.



Construction d'une séquence à partir de l'exemple

Evaluation des performances

Dans l'exemple au-dessus, on a choisi k=3 et cela fonctionne très bien. Pour des séquences plus longues, choisir un k petit suppose la création des boucles dans le graphe, ce qui devient vite très complexe car l'utilisation de boucles suppose de devoir mettre plusieurs axes entre deux nœuds afin de garder un graphe Eulérien. Il serait bien trop complexe de savoir quand enlever les liens en trop et quand les garder, c'est pourquoi la façon la plus simple est d'augmenter la taille des k-mers pour que les nœuds est le moins de chance d'être réutilisé et de former plusieurs chemins.

L'assemblage avec utilisation du graphe de bruijn est en O(n), on voit bien que le temps d'exécution est linéaire (je n'ai pas réussi à faire un beau graphique).

Pierre TRETON

nombre reads	100	500	1000	5000	10000	29300
naïf opti	0,000005245	0,925	3,177	103,178	448,01	2325,009
graphe	1.978E-5	0.002	0.002	0.019	0.043	0.131

Les temps d'exécution des deux assembleurs ne sont pas du tout du même ordre de grandeur.

J'ai fait dans le notebook jupyter un bout de code pour tester le temps d'exécution de l'assembleur graphe de bruijn, en partant de la moitié de la taille de la chaine jusqu'à la taille moins un.

On observe que plus k est élevé, plus le temps d'exécution est court. Cela s'explique par la nature des données qui ont souvent un recouvrement optimal égal à la taille moins un.

Je n'ai pas eu le temps de faire un graphique avec ces données mais il est intéressant de les visualiser en lançant le code dans le notebook.