UE BIF : projet “Compression de lectures de séquençage”

Téo Lemane, Claire Lemaitre

22/10/2019

Ce projet est à réaliser en **binôme**, durant les séances de TP et pendant votre temps libre.
Il devra être rendu au plus tard le **10 décembre 2019**, par mail aux adresses :
`teo.lemane@inria.fr` et `claire.lemaitre@inria.fr` avec comme sujet `[UE BIF] Projet` suivi de vos deux noms.
Vous trouverez à l'adresse `http://bioinformatique.irisa.fr/BIF2019/` des informations complémentaires tout au long de la réalisation du projet (données tests, précisions, ajustement du sujet, . . .).

1 Pitch

Proposer un compresseur/décompresseur de fichiers de lectures issues d'un séquençage d'ADN. L'idée de base du compresseur est **1/** de créer un graphe de *de Bruijn* à partir des données de séquençage **2/** de coder chaque lecture par son premier k-mer puis coder son chemin emprunté dans le graphe (voir la Figure 1).

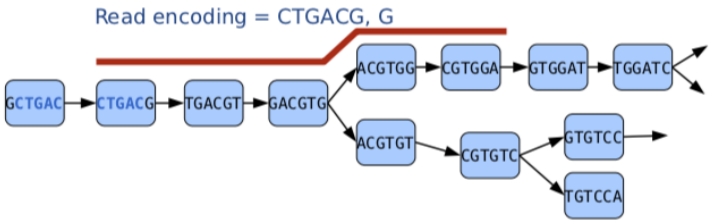

FIGURE 1 – Exemple d'encodage de la lecture "CTGACGTGGA" sur un graphe de *De Bruijn* construit pour $k = 6$.

1.1 Rappels sur le séquençage et le graphe de *de Bruijn*

Une sortie typique d'un séquençage d'ADN est un fichier contenant des millions, voire des centaines de millions de courtes séquences d'ADN (typiquement 100 nucléotides), appelées *lectures* ou *read* en anglais. Chaque séquence provient d'une position aléatoire et inconnue dans le génome de l'organisme qu'on cherche à séquencer. Le génome est séquencé avec une certaine redondance, appelée couverture ou profondeur de séquençage. Typiquement si un génome est séquencé à 30x, chaque position du génome est couverte en moyenne par 30 lectures. Ce type de fichier est donc par nature redondant et peut facilement se compresser.

Le graphe de *de Bruijn* est une structure de données très utilisée en assemblage de génomes. C'est un graphe dirigé dans lequel, les noeuds sont l'ensemble des mots distincts de taille k (k-mer) présents dans un ensemble de séquences. On définit un arc du noeud x au noeud y si le suffixe de taille $k-1$ de x est le préfixe de taille $k-1$ de y.

Cette structure de données permet de représenter de manière compacte l'information de séquences contenue dans un fichier de séquençage, puisque chaque k-mer ne sera représenté qu'une seule fois même s'il est présent dans plusieurs lectures. La séquence d'une lecture peut être retrouvée comme un chemin dans ce graphe.

Pour une valeur de k suffisamment grande ($k > 20$), la très grande majorité des k-mers est unique dans un génome donné, et le graphe a une topologie assez "linéaire", avec la majorité des noeuds n'ayant qu'un seul successeur et un seul prédécesseur. Par exemple, dans un cas idéal où le génome ne possède que des k-mers uniques (aucune répétition de taille $> k$), le plus grand chemin dans ce graphe représente la séquence complète du génome séquencé.

Dans le cas de la compression, une lecture sera représentée par son noeud de départ dans le graphe (un k-mer), sa taille (en général toutes les lectures sont de même taille) et la liste des bifurcations qu'il faut choisir dans le graphe pour reconstruire la lecture. Si les paramètres du graphe sont bien choisis, on espère que cette liste de bifurcations sera très petite (voire vide) pour la majorité des lectures.

2 Entrées/sorties

2.1 Compression

Entrées

— Un fichier de lectures $\mathcal{L}$ au format `fasta` (voir ci-après). Toutes les lectures seront de taille strictement identique.

Sorties

— Un fichier `.graph.pgz` contenant une version compressée du graphe de *de Bruijn* (on utilisera la sérialisation avec le package `pickle`, voir un exemple dans la section 7).
— Un fichier texte `.comp` étant une version compressée des lectures de $\mathcal{L}$ (les séquences d'ADN seulement), voir le format ci-dessous.

2.2 Decompression

Entrées

— Un fichier `.graph.pgz` contenant une version compressée du graphe de *de Bruijn*
— Un fichier `.comp`

Sorties

— Un fichier de lectures $\mathcal{L}'$ au format `fasta`. Ce fichier de lectures ne devra pas différer du fichier original $\mathcal{L}$ (compression sans perte, au minimum pour les séquences d'ADN) :
  — Les lectures devront être encodées dans l'ordre original des lectures dans $\mathcal{L}$
  — Pour les *headers*, on fera l'hypothèse qu'ils seront toujours sous la forme "`>read i`" (avec i commençant à 0 et s'incrémentant pour chaque lecture). Ainsi, on ne cherchera pas à encoder, ni stocker l'information des *headers* dans les fichiers compressés, on pourra les reconstruire à la volée lors de la décompression.

2.3 Formats des fichiers

Le format `fasta` est le format standard en biologie et bioinformatique pour écrire des séquences biologiques dans un fichier. C'est un fichier texte, avec deux lignes par séquence. La première ligne d'une

séquence commence toujours par le caractère ">", suivi du nom ou d'informations sur la séquence, on appelle cette ligne le *header*. La deuxième ligne contient la séquence d'ADN.

Exemple avec 2 séquences :

```
>read 0
ATTTCGGGAAAAAATCGAGCCCTAATT
>read 1
CGGGAAAAAATCGAGCCCTAATTATTT
```

Le format attendu `.comp` pour représenter ces 2 lectures (pour k=8) est un fichier texte, dont la première ligne contient la taille des lectures, puis chaque ligne représente une lecture (dans le même ordre que le fichier original) avec le kmer de départ, un espace et la liste des bifurcations :

```
27
ATTTCGGG TC
CGGGAAAA TCT
```

3 Algorithmes

3.1 Structure de données pour le graphe de *de Bruijn*

Nous représenterons le graphe de *De Bruijn* à partir d'un filtre de Bloom.

3.1.1 Rappels sur le filtre de Bloom

Un filtre de Bloom est une structure de données probabiliste permettant de représenter de manière très compacte un ensemble d'éléments, et de requêter très rapidement la présence ou l'absence d'un élément quelconque.

C'est un tableau de bits $B[0, ..., m-1]$ associé à L fonctions de hachage avec $h_i : U \rightarrow \{0, ..., m-1\}$, où $1 \leq i \leq L$ et U est l'univers des objets qui peuvent être insérés dans le filtre. Une fonction de hachage est une fonction qui va, à partir d'une entrée, renvoyer une signature qui correspond ici à une position dans le filtre, c'est-à-dire un entier p tel que $0 \leq p \leq m-1$. Pour insérer un élément x, le filtre règle $B[h_i(x)] \leftarrow 1$ pour $i = 1, ..., L$. Pour vérifier la présence d'un élément, on vérifie les valeurs des bits correspondant à chaque fonction de hachage : $CONTAIN(B, x) = \bigwedge_{i=1}^{L} B[h_i(x)]$.

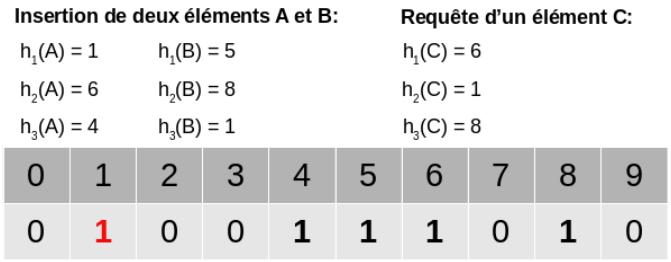

FIGURE 2 – Exemple d'insertion de deux éléments dans un filtre de Bloom de taille 10 avec 3 fonctions de hachage. Pour chaque élément, 3 positions sont retournées par les 3 fonctions de hachage et les bits correspondant à ces positions sont réglés à 1. Collision entre $h_1(A)$ et $h_3(B)$. Les collisions peuvent aboutir à des faux positifs lors de la requête comme c'est le cas pour l'élément C. Les positions correspondantes aux valeurs de hachage de C sont à 1 alors que cet élément n'a pas été indexé.

Les fonctions de hachage n'étant pas bijectives (typiquement quand $m < |U|$), il est possible d'avoir plusieurs éléments avec la même signature (Figure 2). Cette particularité entraîne la possibilité d'avoir des faux positifs lors du requêtage. Le taux de faux positifs dépend de m, $|U|$ et L.

3.1.2 Représentation et parcours du graphe de *de Bruijn*

Il est possible de représenter **implicitement** un graphe de *de Bruijn* par un filtre de Bloom. Dans ce cas, on insère uniquement les k-mers (noeuds du graphe) dans le filtre. Puis pour parcourir le graphe à partir d'un k-mer donné, on teste la présence de ces 4 voisins possibles.

Exemple pour des k-mers de taille 6 :
Pour parcourir le graphe à partir du k-mer ACGTCA, on va interroger le filtre pour ces 4 voisins :
- CGTCA**A**
- CGTCA**C**
- CGTCA**G**
- CGTCA**T**

Chaque élément (k-mer) sera alors représenté par L bits (pour les L fonctions de hachage) dans le pire des cas, contre 8*31=248 bits pour une représentation sous forme de texte et des k-mers de taille 31.

3.1.3 Implémentation

L'implémentation du filtre de Bloom devra être réalisée à partir de la classe python *bytearray* et de la fonction de hachage *MurmurHash3* qui sera donnée (package `https://pypi.org/project/mmh3/`, voir section 7.2). On se limitera à des valeurs de $k < 32$.

Important : un TD et un TP seront consacrés à l'implémentation du filtre de Bloom (semaine 47, 18-20 novembre).

3.2 Encodage des lectures

Pour la **compression** il s'agit simplement de créer le graphe de *de Bruijn* à partir de tous les kmers des lectures, et finalement de stocker uniquement pour chaque lecture son kmer et les éventuels choix à faire aux embranchements. Pour la **décompression**, il suffit de recharger le graphe de *de Bruijn* en mémoire et de partir du k-mer de départ de chaque lecture et de suivre le graphe de *de Bruijn* en choisissant les bons embranchements le cas échéant.

3.3 Reverse-complément et k-mer canonique

Les molécules d'ADN sont composées deux brins antiparallèles, où chacun de ces brins est un polymère constitué de nucléotides (A, C, G, T). Ces deux brins sont complémentaires, chaque nucléotide est capable de se lier à un autre nucléotide spécifique sur le brin complémentaire (A :T, C :G).

Lors du séquençage les brins d'ADN sont lus dans les deux sens ce qui aboutit à la présence de k-mers portant la même information (l'un peut être déduit de l'autre). Comme il n'est pas nécessaire de stocker deux fois l'information, seules les *formes canoniques* des k-mers seront représentées dans votre graphe de De Bruijn. On définit la *forme canonique* comme la séquence la plus petite lexicographiquement entre un k-mer et son reverse-complément.

Exemple pour le k-mer CGTACGT, la représentation en double brin de la molécule est :

```
5' - CGTACGT - 3'
3' - GCATGCA - 5'
```

Le reverse-complément correspond ici à **ACGTACG**, qui est la séquence du brin complémentaire lue de la droite vers la gauche (sens $5'->3'$). Observer CGTACGT revient alors à observer ACGTACG, dans ce cas, c'est la forme ACGTACG qui sera stockée dans le graphe.

3.4 [Bonus] Améliorer le taux de compression

A minima, nous vous demandons d'implémenter la version "basique" telle que décrite dans les sections précédentes. Mais le taux de compression de cette version n'est pas optimal et peut être encore amélioré.

Plusieurs pistes d'amélioration peuvent être envisagées :

— Encoder de manière plus efficace le premier k-mer
— Ré-ordonner les lectures. Afin d'éviter de stocker le premier k-mer de chaque lecture, on peut regrouper toutes les lectures qui débutent par le même k-mer et ainsi ne stocker qu'une fois ce k-mer.
— Optimiser le nombre de faux positifs dans le filtre de Bloom, en jouant sur le nombre de fonctions de hachage.
— ... ou toute autre idée lumineuse !

La compétition au meilleur taux de compression est lancée !

Note : si vous implémentez une ou plusieurs optimisations du taux de compression, la version "basique" devra rester opérationnelle (et exécutée par défaut), et ces optimisations seront executées si l'utilisateur renseigne l'option `-optim`.

4 Aspects pratiques

4.1 code

— Langage de programmation : python3.
— Bibliothèques extérieures type *BioPython* interdites.
— L'utilisation d'un package type `getopt` ou `argparse` pour gérer les options du programme est vivement conseillée.
— En cas d'erreur lors de l'appel de votre programme, affichage d'un message détaillé des options possibles. [anglais]
— Votre code sera largement commenté. [anglais]
— Vos noms de variable et de fonction auront un sens. [anglais]

Les implémentations du compresseur et du décompresseur seront **bien séparées** : votre dossier contiendra au moins 2 fichiers python : `compress.py` et `decompress.py`. Cependant les modules communs à ces deux fichiers ne seront pas dupliqués, mais implémentés dans des fichiers .py ad-hoc.

4.2 Rapports

Vous fournirez **1 rapport "développeur"** et **1 rapport "utilisateur"**, en français ou anglais.

Développeur

Un court rapport (max. 3 pages) développeur présentera l'organisation du code de chacun des deux programmes (compresseur et decompresseur) et l'enchainement des divers appels de fonctions clefs.

Utilisateur

Un rapport destiné aux utilisateurs présentera le compresseur/décompresseur et son cadre d'utilisation. Une importante partie du rapport sera consacrée à l'évaluation du compresseur en fonction des paramètres

utilisés (valeur de k, taille du filtre de Bloom, ...) (temps de calcul, mémoire utilisée, taux de compression). À l'image d'un article scientifique, cette partie présentera des résultats d'expérimentations qui seront ensuite discutés, conduisant à une conclusion générale guidant les utilisateurs de votre outil. En comptant les éventuelles figures et tableaux, votre rapport utilisateur ne devra pas excéder 6 pages.

5 Données

Divers jeux de lectures vous seront fournis. Dans un premier temps, un petit jeu sera disponible. Des jeux plus conséquents seront ensuite proposés sur la page web du projet, pour tester les performances de votre outil.

6 Notation

La notation de vos projets (rendus à temps et respectant le format du mail de rendu) prendra en compte les aspects suivants :

— Choix algorithmiques et succès de l'implémentation,
— Organisation, lisibilité et commentaires du code,
— Qualité des rapports et des analyses effectuées.

Ne sous-estimez pas l'importance des rapports et la forme du code. Un projet "qui marche" n'est pas nécessairement synonyme de bonne note.

7 Utilitaires

7.1 Sérialisation

Il est possible d'écrire le contenu d'un objet entier dans un fichier en utilisant le paquet *pickle* (documentation ici : `https://docs.python.org/3/library/pickle.html`) :

```
import pickle
O = MY_OBJECT()
pickle.dump(O, open("myobjectO.dumped","wb"))
```

Plus tard (lors d'un autre appel du programme, vous pouvez lire 'O' à partir du fichier *myobjectO.dumped* :

```
O = pickle.load(open("myobjectO.dumped","rb"))
```

Pour réduire l'espace disque nécessaire au stockage d'un objet, vous pouvez utiliser pickle avec des fichiers gzip :

```
import pickle
import gzip

#Ecriture
pickle.dump(O, gzip.open("myobjectO.dumped.gz", "wb"))

#Lecture
O = pickle.load(gzip.open("myobject0.dumped.gz", "rb"))
```

7.2 Fonction de hachage MurmurHash3

7.2.1 Installation

Pour installer un paquet python, vous pouvez utiliser le gestionnaire de paquet python *pip* (pip3 pour python3) :

```
pip3 install mmh3 --user
```

Si pour des raisons de droits vous ne pouvez pas installer ce paquet, il est possible de faire un environnement virtuel qui va permettre une installation de python3 isolée du système.

```
virtualenv -p /usr/bin/python3 venv
```

Cette commande va créer un dossier *venv* avec une installation de python3. Il faut ensuite sourcer cet environnement :

```
source ./venv/bin/activate
```

Après avoir sourcé votre environnement, vous pouvez utiliser *pip3* qui installera les paquets dans venv/lib/python3. Si vous utilisez un IDE, pensez à changer l'interpréteur qui est classiquement celui du système (/usr/bin/python3) par l'interpréteur de votre envrionnement (/chemin_vers_venv/venv/bin/python3).

7.2.2 Utilisation

```
import mmh3

def kmer2hash(kmer, seed, m):
    return mmh3.hash(kmer, seed=seed, signed=False) % m
```