TP assemblage

La création et simplification d'un graphe de de Bruijn est une tâche complexe qui ne peut être résolue simplement. Nous ne chercherons pas l'optimalité, ni la meilleur implémentation (nous n'en avons pas le temps), mais à résoudre des problèmes de topologie de graphe, tout en améliorant votre maîtrise de la programmation Python. Afin de réaliser cette tâche, vous devrez résoudre successivement les problèmes posés dans ce tp.

L'objectif de ce TP sera d'assembler le génome de l'entérovirus A71. Ce génome présente l'intérêt d'être très court: 7408 nucléotides, linéaire et non segmenté.

Clonez le projet:

git clone https://github.com/aghozlane/debruijn-tp

Et ajoutez ce travail dans votre dépôt github (à créer en ligne):
git remote add myrepo https://github.com/mylogin/debruijn-tp
git push myrepo

Dans le dossier debruijn-tp/data/, vous trouverez:

- eva71.fna : génome du virus d'intérêt
- eva71_plus_perfect.fq: lectures

Exécutez les commandes suivantes dans ce dossier les commandes :
head -n 8 eva71_plus_perfect.fq > eva71_two_reads.fq
head -n 400 eva71_plus_perfect.fq > eva71_hundred_reads.fq

Le fichier fastq dont vous disposez a été généré à l'aide du programme ART [Huang 2011] via la commande:
art_illumina -i eva71.fna -ef -l 100 -f 20 -o eva71 -ir 0 -dr 0 -ir2 0 -dr2 0 -na -qL 41 -rs 1539952693
Les lectures ont une qualité maximale (41) et ne présentent pas d'insertion. Seuls les lectures correspondant aux brins 5' -> 3' vous sont ici fournies.
Par soucis de simplicité, vous créerez un programme Python3 nommé debruij.py qui prendra en argument:

-i fichier fastq single end
-k taille des kmer (optionnel - default 21)
-r Reference genome (optionnel)
-o fichier config

Vous utiliserez les librairies networkx, pytest et pyllint de Python:

`pip3 install --user networkx pytest pylint pytest-cov`

Vous **testerez** vos fonctions à l'aide de la commande `pytest --cov=debruijn` à exécuter dans le dossier debruijn-tp/. En raison de cette contrainte les noms des fonctions ne seront pas libre. Il sera donc impératif de respecter le nom des fonctions "imposées", de même que leur caractéristique et paramètres.
Vous vérifierez également la qualité syntaxique de votre programme en exécutant la commande: `pylint debruijn.py`
Chaque étape sera ponctuée de **commit** et **push** sur votre dépôt.

1. Création du graphe de de Bruijn:

Pour cette première étape, vous utiliserez le fichier: eva71_two_reads.fq.

a. Identification des k-mer unique

Créez un dictionnaire contenant les k-mer uniques présent dans notre ensembre de reads. Nous aurons besoin de connaître le nombre d'occurrence de chaque k-mer. Trois fonctions sont ici imposées:

- **read_fastq** qui prend un seul argument correspondant au fichier fastq et retourne un itérateur de séquences
- **cut_kmer** qui prend une séquence, une taille de k-mer et retourne un itérateur de k-mer
- **build_kmer_dict** qui prend un fichier fastq, une taille k- mer et retourne un dictionnaire ayant pour clé le k-mer et pour valeur le nombre d'occurrence de ce k-mer

b. Construction de l'arbre de de Bruijn

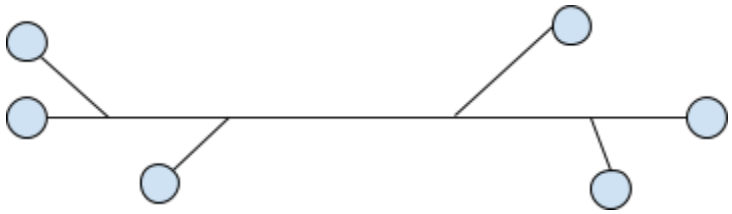
Créez un arbre orienté et pondéré représentant tous les k-mers préfixes et suffixes existant.
Par exemple:

Le kmer: TCAGAGCTCTAGAGTTGGTTC (vu 10 fois)
|
weight :10
Donne: TCAGAGCTCTAGAGTTGGTT ----> CAGAGCTCTAGAGTTGGTTC

La fonction **build_graph** prendra en entrée un dictionnaire de k-mer et créera l'arbre de k-mer prefixes et suffixes décrit précédemment. Les arc auront pour paramètre obligatoire un poids nommé "**weight**".

Validez à l'aide de **pytest** le test de construction, testez votre implémentation de **pylint** et n'oubliez pas de commiter et pusher.

2. Parcours du graphe de de Bruijn



Vous implémenterez une fonction qui parcourt le graphe de de Bruijn et identifie tous les chemins présents entre des noeuds d'"entrée" et des noeuds de "sorties". Chaque chemin constituera un contig potentiel.

Trois fonctions sont ici imposées:

- **get_starting_nodes** prend en entrée un graphe et retourne une liste de noeuds d'entrée
- **get_sink_nodes** prend en entrée un graphe et retourne une liste de noeuds de sortie
- **get_contigs** prend un graphe, une liste de noeuds d'entrée et une liste de sortie et retourne une liste de tuple(contig, taille du contig)
- **save_contigs** qui prend un tuple (contig, taille du contig) et un nom de fichier de sortie et écrit un fichier de sortie contenant les contigs selon le format:

Testez votre implémentation avec le fichier eva71_hundred_reads.fq

Vous écrirez les contigs en respectant le format fasta à l'aide de la fonction fill:

```
def fill(text, width=80):
    """Split text with a line return to respect fasta format"""
    return os.linesep.join(text[i:i+width] for i in range(0, len(text), width))
```

Et indiquerez dans l'intitulé:

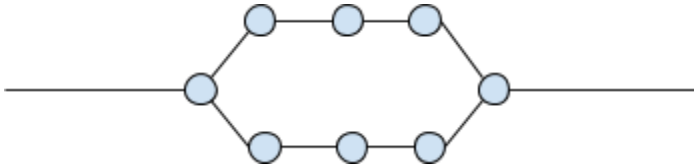
- >contig_Numéro len=longueur du contig

Validez votre implémentation à l'aide de **pytest** et de **pylint** et n'oubliez pas de commiter et pusher.

3. Simplification du graphe de de Bruijn

    a. Résolution des bulles

On considère deux chemins redondant s'ils démarrent et finissent au même noeud (formant une bulle) (et contenant des séquences similaires).



Ces bulles peuvent être le résultat d'erreur, de variant biologiques, ou de répétition dans le génome. L'augmentation taille k peut les faire disparaître mais rendre également l'utilisation du graphe de de Bruijn moins efficace.

Par simplicité, nous chercherons ici les noeuds ayant plusieurs prédécesseurs et déterminerons s'ils ont un ancêtre commun. Le meilleur chemin (liste de noeuds consécutif et acyclique) sera identifié par 3 critères:

- Un chemin est plus fréquent
- Un chemin est plus long
- Le hasard, vous imposerez une seed à 9001

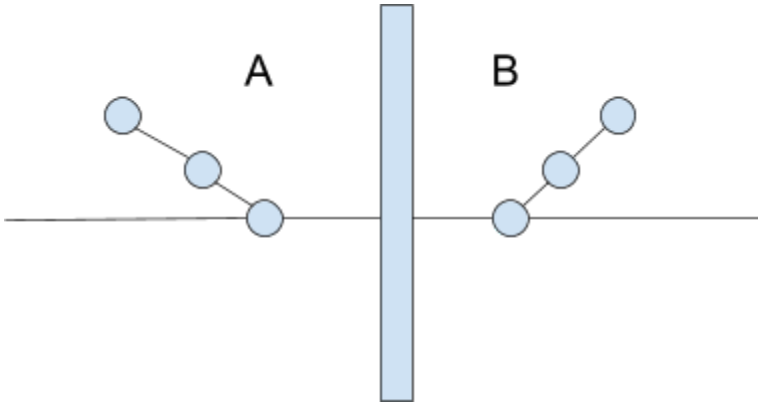
Vous implémenterez les fonctions:

- **std** qui prend une liste de valeur, qui retourne l'écart type.
- **path_average_weight** qui prend un graphe et un chemin et qui retourne un poids moyen.
- **remove_paths** qui prend un graphe et une liste de chemin, delete_entry_node pour indiquer si les noeuds d'entrée seront supprimés et delete_sink_node pour indiquer si les noeuds de sortie seront supprimés et retourne un graphe nettoyé des chemins indésirables.
- **select_best_path** qui prend un graphe, une liste de chemin, une liste donnant la longueur de chaque chemin, une liste donnant le poids moyen de chaque chemin, delete_entry_node pour indiquer si les noeuds d'entrée seront supprimés et delete_sink_node pour indiquer si les noeuds de sortie seront supprimés et retourne un graphe nettoyé des chemins indésirables.
- **solve_bubble** qui prend un graphe, un noeud ancêtre, un noeud descendant et retourne un graph nettoyé de la bulle se trouvant entre ces deux noeuds.
- **simplify_bubbles** qui prend un graphe et retourne un graphe sans bulle

Validez votre implémentation à l'aide de **pytest** et de **pylint** et n'oubliez pas de commiter et pusher.

b. Détection des pointes (tips)

Une pointe est une chaîne de noeud qui est déconnectée d'un côté. Elle se distingue du chemin "principal" par deux critères: sa longueur et son poids inférieur.



Deux cas doivent être ici considérés:

- Dans le cas A, un noeud aura plusieurs prédécesseurs
- Dans le cas B, un noeud aura plusieurs successeurs

Vous implémenterez en conséquence les fonctions:

- **solve_entry_tips** qui prend un graphe et une liste de noeuds d'entrée et retourne graphe sans chemin d'entrée indésirable
- **solve_out_tips** qui prend un graphe et une liste de noeuds de sortie et retourne graphe sans chemin de sortie indésirable

Validez votre implémentation à l'aide de **pytest** et de **pylint** et n'oubliez pas de commiter et pusher.

4. Soumission

Connectez vous à l'adresse https://c3bi.pasteur.fr/m2-p7-resultat/
Indiquez votre nom / prénom / dépôt et soumettez votre travail.

Attendez le mail de confirmation