IFT2015 Travail pratique 2

Pour ce travail pratique, vous devrez créer un assembleur minimaliste en utilisant un graphe de Brujin.

Un graphe de Brujin est un graphe orienté dont les noeuds représentent des séquences de longueur k et dont les arcs indiquent les transitions entre les noeuds suite à l'ajout d'un caractère et de la prise du suffixe de longueur k de la séquence résultante.

Pour ce travail, nous considérerons seulment les graphes où les transition se font sur un symbol de l'alphabet génomique $\Sigma = \{A, T, C, G\}$ et les séquence de longueur $k = 21$.

Par exemple, la chaîne "ACTG", avec $k = 4$, a des arcs sortants vers "CTGA", "CTGT", "CTGC" et "CTGG" étiquetés par "A", "T", "C" et "G" respectivement.

Une stratégie adoptée par certains assembleurs est de segmenter les fragments de longueur l en sous-séquences de longueur k nommées k-mers et d'effectuer l'assemblage en parcourant les composantes disjointes du graphe de Brujin incomplet induit par ces k-mers.

Une façon directe d'implanter ce graphe est d'utiliser une table de hachage sur les chaîne de caractères représentant les noeuds et de considérer les arcs implicitement en énumérant les transitions possibles (i.e. celles qui aboutissent à un noeud existant). En exploitant les propriétés du domaine d'application, vous devrez construire une fonction de hachage efficace sur ces chaînes.

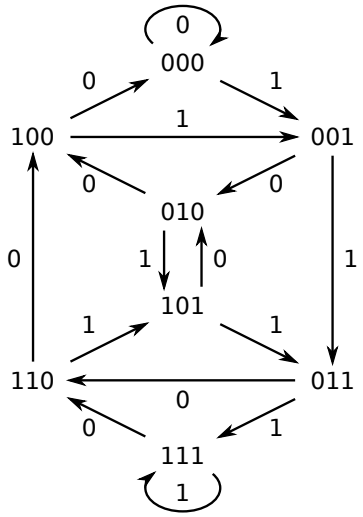

Figure 1: Exemple de graphe de Brujin avec $k = 3$ sur l'alphabet $\Sigma = \{0, 1\}$ et transition sur un seul symbole.

Instructions

Le travail peut être fait en équipe de deux (2). Assurez-vous d'inscrire vos noms et matricules. Vous devez également développer **VOTRE PROPRE CODE**; tout plagiat détecté entraînera automatiquement un échec.

Le code soumis devra être de bonne qualité, lisible et bien documenté. Jusqu'à dix (10) points pourront être déduits de la note finale. Vous êtes encouragé à consulter PEP8 qui est un guide de style pour Python à la page suivante: https://www.python.org/dev/peps/pep-0008/

Une pénalité de vingt (20) points sera appliquée pour chaque début de période de 24 heures suivant la date limite de remise.

Vous devez remettre un rapport d'au plus trois (3) pages décrivant vos implémentations et élaborant les questions théoriques en format PDF.

1. a) Implémenter la structure de données du graphe de Brujin à l'aide d'une table de hachage. Le constructeur prendra en entrée un itérable sur des chaînes de caractères de longueur k. /30

Puisque l'itérable est de longueur arbitraire, vous devez redimensionner dynamiquement la table de hachage pour maintenir le facteur de charge sous 0.75.

```python
class DeBrujinGraph:
    def __init__(self, nodes: Iterable[str], k=21):
        pass # initialise la structure de données

    def __contains__(self, N: str) -> bool:
        pass # détermine si le graphe de Brujin contient le noeud N

    def __iter__(self) -> Iterable[str]:
         return self.nodes() # retourne un itérable sur les noeuds du graphe

    def load_factor() -> float:
        pass # calcule le facteur de charge de la table de hachage sous-jacente

    def add(self, N: str):
        pass # ajoute le noeud N au graphe

    def remove(self, N: str):
        pass # enlève le noeud N du graphe

    def nodes(self) -> Iterable[str]:
        pass # retourne un itérable sur les noeuds du graphe

    def predecessors(self, N: str) -> Iterable[str]:
        pass # retourne tous les prédécesseur du noeud N

    def successors(self, N: str) -> Iterable[str]:
        pass # retourne tous les successeurs du noeud N
```

b) Discuter des détails de votre implémentation en répondant aux questions suivantes: /10

- Quelle fonction de hachage avez-vous utilisé? et dans quelle mesure est-elle spécifique au problème?
- Quelle stratégie d'adressage avez-vous utilisé?

c) Quel serait le principal gain d'utiliser une technique de hachage sensitif à la localité et de l'adressage linéaire pour ce problème? /5

2) a) Encoder toutes les k-mers des fragments fournis dans le fichier FASTQ dans une instance du graphe De Brujin. Chaque fragment possède $l - k + 1$ k-mers. /10

b) **Bonus:** Implanter une table de hachage plus rapide que celle de CPython (i.e. celle utilisée par `set` et `dict`) pour stocker des k-mers pour les opérations effectuées en 2a. /5

3) a) Parcourir le graphe pour obtenir des segments contiguës que vous devrez stocker dans un fichier FASTA nommé `contigs.fa` avec un identifiant unique pour chacun. /20

Commencer par identifier les noeuds qui n'ont pas de prédécesseurs pour débuter l'exploration. Alternativement, vous pouvez explorer les segments dans les deux directions.

b) Dans votre rapport, répondez aux questions suivantes: /5

- Quelle stratégie de parcours avez-vous adopté?
- Comment les boucles sont-elles gérées durant le parcours?

c) Quelle est l'influence du choix de k sur le résultat obtenu par le parcours? /3

4) a) Comparer les séquences assemblées avec les séquences de références contenus dans le fichier FASTA. /15

Vous pouvez utiliser `string.find` pour effectuer chaque paire de comparaison. Reporter chaque occurence dans un fichier `occurences.bed` contenant les quatre (4) colonnes suivantes séparées par des tabulations:

- `reference`: identifiant du segment de référence
- `start`: début de l'occurence basé sur 1 inclusif (une occurence à l'indice zéro devrait être en position 1)
- `end`: fin de l'occurence basé sur 1 inclusif
- `contig`: identifiant du contig

b) Peut-il exister des séquences qui ne peuvent pas être assemblée correctement dans la référence? /2

c) **Bonus:** Proposer une structure de données appropriée pour effectuer une comparaison efficace des séquences assemblées avec celles de référence. La complexité doit être strictement inférieure à la méthode proposée en 4a. /5

La remise doit être faite avant le **14 décembre 2018 à 23h55** sur StudiUM sous la forme d'une archive ZIP qui devra contenir les quatre fichiers suivants:

- `graph.py` contenant l'implémentation de la classe `DeBrujinGraph`
- `main.py` (ou `main.ipynb` si vous utiliser un notebook Jupyter)
- `contigs.fa` vos contigs assemblés
- `occurences.bed`
- `rapport.pdf`

Le fichier `main.py` doit contenir le code pour les sections d'application du devoir. Assurez-vous de bien identifier les questions.

Le code sera testé automatiquement, vous devez donc respecter les noms des fichiers, classes et méthodes proposées.

Format de fichiers et données à utiliser

Pour vous simplifier la tâche, un ensemble de lectures artificielles `reads.fastq.gz` a été généré qui sera assemblé sur une référence provenant du transcriptome de *C. elegans* `GCF_000002985.6_WBcel235_rna.fna.gz`.

FASTQ

Le format FASTQ représente des fragments et leur qualité lus par un appareil de séquençage à haut-débit. Chaque observation est constituée de quatre lignes:

```
@SEQID
SEQUENCE
+
QUALITY
```

La ligne `SEQUENCE` contient un fragment de 100 caractères qui a été observé dans la référence. Vous pouvez ignorer les autres lignes.

FASTA

Le format FASTA contient des séquences identifiées. Chaque entrée commence par un caractère `>` suivi d'un identifiant, d'une description et de la séquence en soit qui peut contenir des nouvelles lignes.

```
>ID DESCRIPTION
SEQUENCE
```

Il vous est recommendé de ne pas décompresser les fichiers founis, mais plutôt d'utiliser le module `gzip` de la librairie standard de Python.

```
import gzip
with gzip.open('reads.fastq.gz', mode='rt') as f:
    for seqid in f:
        seq, _, qual = f.readline(), f.readline(), f.readline()
```

Toutes les données sont disponibles sur StudiUM.

Amusez-vous bien!