M1 - BIF
TP5 - Assemblage et graphe de De Bruijn

Victor Levallois, Claire Lemaitre, Pierre Peterlongo

2023-2024

Construction du Graphe

Pour implémenter un graphe de De Bruijn (dBG) et les méthodes de recherche associées, on utilisera une classe Python. Vous trouverez sur moodle un fichier `de_bruijn_graph.py` contenant une classe `DBG`, deux attributs et une méthode de visualisation de graph via le package *graphviz* (`pip install graphviz` peut être nécessaire à la première utilisation). Il vous reste à implémenter les méthodes nécessaires à la création du graphe, au parcours et à la recherche de motifs.

1 Préliminaires

Il existe plusieurs méthodes de stocker un graphe de De Bruijn (explicitement ou implicitement). Pour ce TP, nous utiliserons la manière explicite et plus précisément tableau associatif (dictionnaire en python) pour stocker les arêtes. A chaque clef K du dictionnaire sera associé un `set()` V contenant tous les prochains voisins de K.

Un `DBG` a donc 2 attributs :

— `k` : la taille des k-mers dans le dBG.

— `graph` : le `dict()` stockant les arrêtes.

Q 1. On construira le `DBG` en ajoutant une à une les lectures (reads). Implémentez une méthode `add_sequence` (n'oubliez pas le `self`) prenant en entrée une séquence S et permettant de mettre à jour le graph avec les nouveaux k-mer de S.

Reprenons l'exemple du TD :

```
AACCGTAT
 ACCGTGTA
  CCGTATAG
  CCGTGTCG
    GTGTAGCG
     TATAGCGT
```

Construisez un dBG `dbg1` à l'aide de ces séquences ($k = 5$) et affichez le. Construisez un deuxième dBG : `dbg2` toujours avec $k = 5$ avec deux petites modifications :

— retirer "GT" du read "GTGTAGCG" -> "GTAGCG"

— retirer le premier "T" du read "TATAGCGT" -> "ATAGCGT"

Affichez `dbg2`, si vous comptez plusieurs composantes connexes, demandez-vous si vous n'avez pas oublié des arêtes lors de la construction.

Q 2. Faites varier k, visualisez `dbg1` en le reconstruisant avec $k = 2$ ou $k = 3$ et avec $k = 7$, quelle différence notez-vous entre des valeurs de k plus ou moins grandes ?

2 Unitigs

Q 3. Implémentez deux méthodes `right_neighbours` et `left_neighbours` prenant en entrée un k-mer et donne en sortie un `set()` des voisins de droite et de gauche du k-mer.

Q 4. Implémentez une méthode `right_extension` qui renvoie la séquence de l'extension à droite sans branchement (unitig) d'un k-mer donné (incluant le k-mer)

3 Identification de *tips* dans le dBG

La 4ème lecture contient une erreur de séquençage en avant dernière position (C à la place d'un A).

Q 5. Implémentez une méthode `detect_tips` renvoyant une liste d'unitigs correspondant aux tips du dBG.

Q 6. Implémentez une méthode `remove_tips` prenant en entrée une liste de tips et permettant de les supprimer du graph. Vérifiez via `DBG.visualize()`.Vous pouvez ré-utiliser et/ou modifier la fonction `right_extension(kmer)`.

4 Identification de *bubbles* dans le dBG

Q 7. Implémentez une méthode `detect_bubbles` renvoyant la liste des *bubbles* du graphe. Une *bubble* sera représentée comme un tuple `(chemin1, chemin2)`. Chaque chemin étant un unitig de la *bubble* auquel on ajoute le caractère précédent et suivant dans le graphe.

5 Retour au monde réel

Q 8. Implémentez une méthode `add_seqs_from_fasta` prenant en entrée un nom/path de fichier et un nombre de séquences `n`. Cette méthode devra lire le fichier .fasta (dont vous pouvez étudier le format ici `https://fr.wikipedia.org/wiki/FASTA_(format_de_fichier)`) et ajouter les `n` premières séquences au dBG.

Q 9. Testez votre implémentation avec le fichier `ecoli_sequencing.fasta` (source : `https://www.ebi.ac.uk/ena/browser/view/SRX17134643` disponible sur moodle. Avec `n`$= 10$ et $k = 31$, essayez de visualiser. Avec `n`$= 100$, trouvez de potentielles bubbles. Avec `n`$= 1000$, notez le temps de calcul de votre premier algorithme de dBG dont la complexité est loin d'être optimale :-)