Projet ALG 2025-2026 - Création et requête sur graphes de *de Bruijn* colorés

Notes de versions:

V1 (12 nov 2025): Version originale

Organisation

Ce projet est à effectuer en binômes.

Il devra être rendu au plus tard le **17 décembre 23h59 max**.

Il sera déposé sur un dépôt github ou gitlab privé sur lequel vous nous inviterez claire.lemaitre@inria.fr, leo.ackermann@inria.fr et pierre.peterlongo@inria.fr. Nous accuserons la bonne réception de l'invitation (vérifiez que c'est bien fait).

La notation considèrera la qualité du code et la qualité des résultats présentés dans un rapport. Nous effectuerons également une soutenance de projet durant laquelle vous présenterez votre code.

Dans les grandes lignes

L'objectif de ce projet est d'implémenter et d'analyser une structure d'indexation d'une collection de génomes, appelée le graphe de *de Bruijn* coloré. Cette structure de données permet de stocker de manière compacte et facilement requêtable l'ensemble des k-mers d'une collection de génomes, tout en gardant l'information de l'appartenance de chaque k-mer aux différents génomes. On peut donc se servir de cette structure de données pour calculer des mesures de similarités, dites "alignment-free", entre une séquence requête et chacun des génomes de la collection en calculant des ratio de k-mers partagés.

Le projet est divisé en deux phases.

1. Durant la première phase, nous considérons N génomes $\{G_1, \ldots, G_N\}$ en entrée, et nous créons $\mathcal{G}$, un graphe de *de Bruijn* coloré à partir de ces génomes. Nous sauvegardons $\mathcal{G}$ sur disque.
2. Durant la seconde phase, nous considérons une séquence requête Q et $\mathcal{G}$. La sortie consiste en une mesure de similarité entre Q et G_i pour tout $i \in \{G_1, \ldots, G_N\}$

Code

Le projet contiendra deux modules `build` et `query`. Ces deux modules seront appelés par une interface commune appelée `dbg_indexer.py` (cf fin de projet pour une aide au codage).

- `dbg_indexer.py build -i sequences_file_list -k kmer_size -o output_file [-h]`

  - Entrées:
    * `-i` un fichier texte contenant sur chaque ligne un chemin vers un génome G_i
    * `-k` la taille des k-mers
    * `-o` le fichier de sortie contenant le graphe de *de Bruijn* coloré (cDBG) *sérialisé* (cf ci-dessous)
    * `-h` affiche l'aide et ne fait rien.
  - Sortie:
    * Le graphe de *de Bruijn* coloré (cDBG) *sérialisé*
    * Des informations de performances sorties dans la console comme indiqué ci-dessous.

      ```
      OUT TIME_BUILD: temps pour créer le cDBG avant sérialisation
      OUT TIME_SERIALISATION: temps pour sérialiser le cDBG
      ```

      D'autres informations pourront être ajoutées tant que ces lignes apparaissent sous ce format.
- `dbg_indexer.py query -q query_file_name.fa -i cdbg -o output_file [-h]`,
  - Entrée:
    * `-q` un fichier fasta contenant une ou plusieurs séquences à requêter : un ensemble de séquences génomiques sur l'alphabet $\Sigma = \{A, C, G, T\}$.
    * `-i` un fichier qui contient le cDBG sérialisé
    * `-o` le fichier de sortie contenant les résultats de similarité entre chaque séquence requête et chaque génome G_i.
    * `-h` affiche l'aide et ne fait rien.
  - Sortie
    * Un fichier de sortie au format txt, avec une ligne par séquence requête composée du header de la séquence et de la valeur de similarité avec chaque génome G_i (ratio de k-mers partagés), séparés par une tabulation (exemple avec $N = 3$ génomes):

      ```
      afirstquery    0.975  0   0.12
      anotherquery   0.18   1   0.42
      ...
      ```

Deux versions du projet

Nous vous demandons de proposer et d'implémenter deux versions du projet.

- Une première version naïve, utilisant un dictionnaire python standard pour représenter le graphe de *de Bruijn* où chaque k-mer est associé à une liste d'identifiants de génomes dans lesquels il apparait (ce que l'on appelle les *couleurs* du k-mer).
- Une seconde version où l'on exploite le fait que pour un unitig, les couleurs des ses k-mers sont identiques. On évitera ainsi de stocker une information redondante pour tous les k-mers d'un même unitig.

Rapport et documentation

À écrire en français ou en anglais, selon votre préférence.

Court document README

Le fichier `README.md` permettra en quelques lignes d'indiquer à quoi sert votre outil, et comment utiliser le programme. Ce document est destiné à des utilisateurs non experts.

Rapport scientifique

L'idée est de présenter les résultats principaux obtenus avec vos outils. 8 pages maximum. On y attend un paragraphe où vous expliquerez votre approche exploitant le fait que les couleurs sont identiques au sein des k-mers d'un unitig pour limiter la taille de la structure proposée.

Autrement, il sera principalement composé de figures représentant:

- les temps de calcul principaux obtenus sur les deux phases (création du cDBG, et requête), en différenciant les temps de lecture ou écriture sur disque du cDBG, du reste.
- la taille des cDBG créés (mesurée par la taille du fichier créé sur disque)

Vous pourrez analyser le comportement de ces mesures en fonctions de différents paramètres (valeur de k, type de données).

Note : la valeur de k doit être ≥ 17.

Ce rapport proposera une discussion au regard de ces résultats (qualités et défauts de la version naïve comparée à la version avancée).

Misc.

Jeux de données proposés pour vos tests.

Des données vous seront fournies sur moodle pour vos tests.

Structure du dépôt git:

Votre dépôt git contiendra **au moins** la structure suivante:

```
.
|-- README.md
|-- report.pdf
`-- src
    |-- advanced
    |   |-- build.py
    |   |-- dbg_indexer.py
    |   `-- query.py
    `-- naive
```

```
|-- build.py
|-- dbg_indexer.py
`-- query.py
```

Note: cette structure a l'avantage de la simplicité mais elle a le désavantage de la redondance de code. Si vous êtes à l'aise pour proposer une autre solution sans redondance, n'hésitez pas à la proposer après l'avoir validé avec nous.

Merci d'éviter de stocker des données (fichiers fasta, d'index, ou de résultats) pour limiter la taille du dépôt.

Notez bien

- Respectez *a minima* les options et format d'affichage imposés. Vous pouvez ajouter d'autres options si vous le souhaitez, mais votre code sera utilisable avec ces seules options (correction automatique)
- Vos programmes ne seront pas interactifs. Une fois lancés avec leurs arguments, ils terminent sans intervention de l'utilisateur.
- Vos programmes seront largement commentés et les noms de classes, de variables et de fonctions devront avoir un sens.
- Nous fournirons au fil du temps divers jeux de données permettant de tester vos outils. Le sujet et les données seront disponibles sur moodle.
- L'utilisation de bibliothèques extérieures devra se limiter au strict minimum. À voir avec nous si vous avez un doute.
- En cas d'erreur utilisateur lors de l'appel de votre programme, affichage d'un message détaillé des options possibles.

Notation

La notation de vos projets (rendus à temps) prendra en compte les aspects suivants :

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

Ne sous-estimez pas l'importance du rapport et la forme du code. Ces deux aspects représentent environ 50% de la note finale. La soutenance de projet permettra de montrer votre compréhension du code et votre capacité à le modifier rapidement.

Aide au codage

Exemple de `dbg_indexer.py` permettant de "parser" les arguments avec sous-tâches (build ou query)

```
import argparse
```

```
def build_index(args): # main indexing function
    k = args.k
    ...

def main():
    parser = argparse.ArgumentParser()
    sub = parser.add_subparsers(dest='cmd')

    b = sub.add_parser('build')
    b.add_argument('--k', type=int, required=True)
    b.add_argument('--fasta', nargs='+', required=True,
                help='FASTA files, one per color')
    b.add_argument('--out', required=True)

    q = sub.add_parser('query')
    q.add_argument('--index', required=True)
    q.add_argument('--fasta', required=True, help='FASTA file with sequences to query')

    args = parser.parse_args()
    if args.cmd == 'build':
        build_index(args) # calling the main function for indexing
    elif args.cmd == 'query':
        query_index(args) # calling the main function for querying
    else:
        parser.print_help()

if __name__ == '__main__':
    main()
```

Sérialisation (écriture et lecture d'un objet sur disque) Il est possible d'écrire le contenu d'un objet entier dans un fichier en utilisant le paquet `pickle` : https://docs.python.org/3/library/pickle.html

```
import pickle
O = my_object()
pickle.dump(O, open("myobject.dumped","wb"))
# Plus tard (lors d'un autre appel du programme,
# vous pouvez lire 'O' à partir du fichier myobject.dumped :
O = pickle.load(open("myobject.dumped","rb"))
```