Projet ALG 2021-2022 - V3 (22 nov 2021)

Notes de versions:

V1: Version originale

V2: correction type: utilisation d'un dictionnaire python pour compter les kmers canoniques (auparavant indiqué comme un set).

V3: ajout précision sur format du mail pour le rendu

Représentation compressée et indexation d'un ensemble de k-mers d'un jeu de données de séquençage.

Il est à effectuer en binômes.

Il devra être rendu au plus tard le Jeudi 6 janvier 23h59, par mail aux adresses : claire.lemaitre@inria.fr, pierre.peterlongo@inria.fr. Le sujet sera "*[UE ALG]Projet NOM1_NOM2*" ou vous remplacerez NOM1_NOM2 par vos noms de famille, dans l'ordre alphabétique.

Ce message contiendra un fichier compressé .gz qui contiendra le code et les rapports (ne pas envoyer de données).

Dans les grandes lignes

Nous cherchons à représenter et indexer des données de séquençage par tous ses k-mers solides.

Étant donné S l'ensemble des séquences issues d'un séquençage de génome, le projet consiste à

1. Générer le set $\mathcal{K}$ contenant les k-mers canoniques *solides* de S.
2. Générer $SPSS(\mathcal{K})$, une séquence contenant tous les k-mers (canoniques ou non) de $\mathcal{K}$.
3. Validation: Étant donné un génome G, votre programme indiquera
   - #FN : le nombre de faux négatifs : k-mers existant dans le génome G mais pas dans $SPSS(\mathcal{K})$
   - #FP : le nombre de faux positifs : k-mers existant dans $SPSS(\mathcal{K})$ mais pas dans le génome G.

   Le calcul de #FN nécessite de savoir rapidement si un k-mer de G appartient à $SPSS(\mathcal{K})$. Ceci se fera d'une part en utilisant l'appel `in` de python, et d'autre part via un FM-index construit sur la séquences $SPSS(\mathcal{K})$. Vous comparerez les temps de calculs de ces deux approches et vous vérifierez qu'elles renvoient bien toutes les deux le même résultat.

   Même remarque pour calculer #FP en indexant ou non le génome G avec un FM-index.

Précisions

- Point 1.: Afin de compter les k-mers nous utiliserons un dictionnaire python.
- Point 2.: Il existe diverses façons de générer $SPSS(\mathcal{K})$. À vous de trouver et de motiver une méthode qui vous semble convenable. Il sera important d'expliquer et de motiver vos choix au travers d'une étude qui mettra en relief les avantages et limitations de votre méthode.
- Point 3.: Simple utilisation de ce que nous avons implémenté en TP.

Code

Le projet contiendra ce module :

- `SequencesToSPSS.py -i sequences_file_name -g genome_fime_name -t solidity_threshold -k kmer_size [-h]`
  - Entrée:
    - `-i` un fichier fasta contenant S : un ensemble de séquences génomiques sur l'alphabet $\Sigma = \{A, C, G, T\}$.
    - `-g` le génome de référence. On suppose que ce fichier contient exactement deux lignes, une ligne de commentaire "`>genome...`" et une ligne unique contenant le génome codé sur le même alphabet Σ.
    - `-t` un seuil de solidité. Chaque k-mer dont la représentation canonique a moins de `solidity_threshold` occurrences n'est pas conservé dans $SPSS(\mathcal{K})$.
    - `-h` affiche l'aide et ne fait rien.
  - Sortie:
    - Aucun ficher demandé en sortie. Le programme affiche sur la sortie standard (en respectant le format demandé plus bas) :
      - La taille de $SPSS(\mathcal{K})$
        - En terme de nombre de caractères
        - En terme de nombre de séquences distinctes concaténées
      - Le nombre #FN
      - Le temps de calcul de #FN sans indexation de $SPSS(\mathcal{K})$
      - Le temps de calcul de #FN avec indexation de $SPSS(\mathcal{K})$
      - Le nombre #FP
      - Le temps de calcul de #FP sans indexation du génome
      - Le temps de calcul de #FN avec indexation du génome.
    - D'autre affichages sont autorisés, mais la sortie standard comportera au moins ces lignes avec ce format (en remplaçant ce qui est entre [] par les valeurs détectées). L'ordre n'importe pas.

```
OUT TIME_SELECTING_KMERS=[temps pour compter les kmers canonique et
stocker les kmers solides]
OUT |SPSS(K)|=[nombre de caractères total de la SPSS]
OUT #SPSS(K)=[nombre de séquences distinctes concaténées dans la SPSS]
OUT TIME_SPSS_CONSTRUCTION=[temps de création de la SPSS à partir des
kmers canoniques solides]
OUT #FN=[Nombre de faux négatifs]
OUT TIME_FN_WITHOUT_INDEX=[temps de calcul de FN sans index en secondes]
OUT TIME_FN_WITH_INDEX=[temps de calcul de FN avec fm-index en secondes]
OUT #FP
OUT TIME_FP_WITHOUT_INDEX=[temps de calcul de FP sans index en secondes]
OUT TIME_FP_WITH_INDEX=[temps de calcul de FP avec fm-index en secondes]
```

Bonus

Vous pourrez messurer et indiquer dans votre rapport:

- La quantité de mémoire utilisée par un set python pour stocker tous les kmers canoniques
- La quantité de mémoire utilisée pour stocker $SPSS(\mathcal{K})$
- La quantité de mémoire utilisée pour stocker $FM(SPSS(\mathcal{K}))$ le FM index utilisé pour indexer $SPSS(\mathcal{K})$

Rapports et documentation.

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

Court document README.

Le fichier README 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 développeur

Ce rapport explicite la structure du programme et précise le fonctionnement des fonctions clefs.

Il doit être court (max 2 pages par module, max 5 pages en tout).

Rapport scientifique

10 pages (très grand) max.
L'idée est de donner le contexte, d'exposer la technique utilisée pour construire $SPSS(\mathcal{K})$, et de donner les résultats en fonction de divers paramètres:

- valeur de k, valeur du seuil de solidité, type de jeux de données, ...

Vous pourrez également indiquer toutes les idées d'amélioration possible pour le calcul et la représentation de la SPSS.

Ce rapport mettra également en avant les différences de temps de traitement observées en utilisant ou non le FM-index lors du calcul de #FN et #FP.

Misc.

Précisions

- Respecter *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 bibliothèque extérieure devra se limiter au strict minimum (pas d'utilisation de bibliothèque de FMIndex par exemple). À voir avec nous si vous avez un doute.
- L'utilisation d'un package type `getopt` pour gérer les options du programme est vivement conseillée.
- 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 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.
- Bonus éventuels (si le reste est correctement finalisé)
- Ne sous-estimez pas l'importance des rapports et la forme du code. Ces deux aspects représentent environ 50% de la note finale.

Aide au codage

Vous pouvez utiliser ces suggestions pour faciliter la mesure du temps d'execution d'une partie de votre code et l'affichage d'une barre de progression lors de différents calculs.

Mesure du temps d'execution d'une portion de code:

timer.py:

```
import time


# This class allows us to monitor the time spent by a function
class Timer():
    # Function called right before the execution of the function
    def __enter__(self):
        self.t1 = time.perf_counter()
        return self

    # Function called right after the function
    def __exit__(self, type, value, traceback):
        self.t2 = time.perf_counter()
        self.t = self.t2 - self.t1

    # Function that prints on the shell the time spent by the instructions
    def print(self, template: str = "{}"):
        print(template.format(round(self.t, 2)))

if __name__ == "__main__":
    # Exemple how to use this class
    with Timer() as total_time:  # time all instructions in the 'with' statements
```

```
        for i in range(5):
            time.sleep(0.471)
    total_time.print("Durée = {} secondes")
```

Affichage d'une barre de progression:

progress_bar.py:

```
import sys
def update_progress(progress):
    barLength = 50  # Modify this to change the length of the progress bar
    status = ""
    if progress < 0:
        progress = 0
        status = "Halt...\r\n"
    if progress >= 1:
        progress = 1
        status = "Done...\r\n"
    block = int(round(barLength*progress))
    text = "\rPercent: [{0}] {1}% {2}".format( "#"*block + "-"*(barLength-block),
round(progress*100,2), status)
    sys.stderr.write(text)
    sys.stderr.flush()

if __name__ == "__main__":
    n = 10000000
    for i in range(n):
        if i%1000 == 0: # Avoid to update at each step (time consuming)
            update_progress(i/float(n))
            # your real code here
    update_progress(1) # Ends the line & writes "Done"
```

Exemple (avec importation et utilisation de barre de progression et de calcul de temps)

```
import time
from timer import Timer # should be in the same directory
from progress_bar import update_progress # should be in the same directory

if __name__ == "__main__":
    n = 60
    with Timer() as total_time:  # time all instructions in the 'with' statements
        for i in range(n):
            update_progress(i/float(60))
            time.sleep(0.1) # your code here
        update_progress(1)
    total_time.print("Time spent = {} seconds")
```