Rapport Projet

Rosset Paulin
Scalabrino Morgan
Baldous Jules

SLEVI502
2023/2024

UNIVERSITÉ CÔTE D'AZUR

Introduction :

Généralités :

Ce projet a pour but de créer un module et une interface graphique pour visualiser la synténie sur trois gènes entre 30 génomes de la bactérie E. coli.

En fournissant un identifiant de protéine et un génome de référence le programme rendra une visualisation de la synténie, et ceci sans avoir à coder quoi que ce soit.

Pour cela, nous utiliserons le langage Python et ses Notebook Jupyter ainsi que Tkinter. Nous utiliserons pyGenomeViz, un package récent de génomique comparative en Python.

Contexte biologique :

Comme dit plus haut, nous voulons visualiser la synténie entre 3 gènes, un gène qu'on nommera d'intérêt/central, car c'est celui sélectionné par l'utilisateur, et les deux gènes en amont et en aval de celui-ci.

La synténie est un concept de génétique comparative se définissant par la conservation de l'ordre et du voisinage d'un groupe de gènes sur plusieurs génomes. Dans notre cas, nous nous intéressons à des espèces de bactéries avec des génomes circulaires, nous allons donc linéariser leur génome.

Plus spécifiquement, nous nous penchons sur la microsynténie, c'est-à-dire, l'étude de la synténie sur un petit groupe de gènes, en l'occurrence 3.

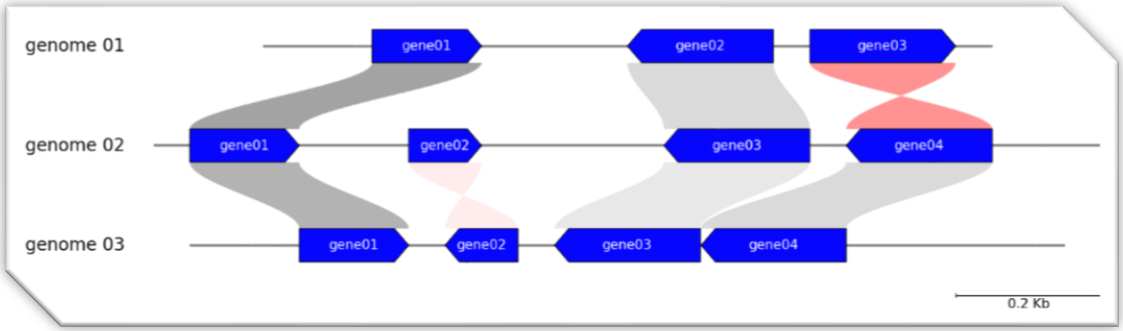

Figure 1: Microsynténie entre 3 génomes sur 4 gènes. Source : pyGenomeViz

Nous observons la synténie sur des génomes d'une même espèce, nous nous attendons donc à avoir une conservation forte entre les génomes. Dans la suite du projet, nous ferons l'abus de langage de confondre protéine et gène. Enfin, nous ferons aussi l'hypothèse qu'il y a un seul support génomique sans compter les éventuels plasmides bactériens.

Organisation du projet :

Nous avons choisi de regrouper tout le code dans un seul module (module.py) que vous pouvez importer ou directement lancer dans le fichier interface.ipnyb. Dans cette interface, il y a deux parties, une utilisant une fenêtre graphique Tkinter et l'autre, agissant en tant que backup vous permettant de tester rapidement les possibilités du code.

Nous avons découpé le projet en trois parties :
- Codage du module (récupération des informations) qui s'est étalé du 7 au 17/11 (Partie 1-5).
- Préparation de l'interface notebook et visualisation (ainsi que son intégration dans le module) qui s'est étalée du 17 au 24/11 (Partie 6).
- Codage de la fenêtre graphique Tkinter qui s'est étalé du 17/11 au 1/12 (Partie 6).

Légende : Fonction**() :** en gras / *Variable :* en italique

Description du module :

Le module se scinde en deux grandes parties : les fonctions et le main.
Il regroupe toute ce qui correspond à la partie 1 à 6 du projet excepté le code permettant de lancer la fenêtre Tkinter.

Nous allons faire une description de chaque partie du projet et comment nous avons abordé celle-ci, en scindant à chaque fois les fonctions crées du main exécuté.

Cette partie se complémente avec les commentaires présents dans le fichier module.py et les help intégrés dans chaque fonction crée. Nous décrirons rapidement le but de la fonction, le code en lui-même est analysé dans le module directement via les commentaires.

NB : Le main a dû être défini comme une fonction au lieu d'un test if _name_ = '_main_' pour faire marcher l'interface Tkinter mais nous le considérerons comme une partie distinctes des autres fonctions.

Partie 0 : Vérification des arguments en input

Cette partie concerne uniquement le main. Nous vérifions que l'utilisateur a fournit le bon nombre d'arguments et surtout la cohérence des informations (l'OS en argument correspond bien à son OS, le génome donné est bien l'un des 30 génomes de E. coli).

Partie 1 : Récupération de la liste des protéines dans un génome

Pour répondre à la problématique de cette partie nous avons défini 4 fonctions :
- **recup_data()** : cette fonction permet à partir d'un génome en argument de récupérer les informations de toutes les protéines de ce génome en les mettant dans un dataframe du module Pandas.
- **recup_info_prot()** : cette fonction récupère des informations sur une protéine dans un génome. Elle utilise pour cela **recup_data()** et gère l'erreur si jamais la protéine en argument n'est pas dans le génome donné.
- **recup_prot()** : cette fonction récupère la liste des protéines contenu dans un génome. Elle utilise pour cela **recup_data()**.

Ces trois fonctions répondent à la partie 1, une quatrième nous sera utile dans la suite :
- **recup_genome()** : cette fonction récupère la liste des 30 génomes disponibles dans le fichier data.

Dans le main :
Nous exécutons **recup_info_prot()** avec comme argument la protéine d'intérêt et le génome de référence choisis par l'utilisateur. Ainsi nous obtenons bien une liste d'informations sur la protéine d'intérêt stockée dans la variable *list_proteine_info* dans la suite.

Partie 2 : Récupération de la séquence d'une protéine

Pour répondre à la problématique de cette partie nous avons défini 1 fonction :
- **recup_seq()** : cette fonction permet de récupérer un objet seq_record à partir d'une protéine dans un génome, ceux-ci rentrés en argument. Pour cela nous parsons le fichier protein.faa pour le génome rentré en argument à l'aide de Seq.IO du module Biopython. L'objet retourné contient la séquence de la protéine ainsi que le header du fichier fasta parsé.

Dans le main :

Nous exécutons simplement la fonction **recup_seq()** avec en argument la protéine d'intérêt et le génome de référence choisis par l'utilisateur. Ainsi nous obtenons l'objet seq_record de la protéine d'intérêt que nous stockons dans la variable *record*.

Partie 3 : Lancer un blastp d'une protéine sur un génome

Pour répondre à la problématique de cette partie nous avons défini 2 fonctions :
- **blastp()** : cette procédure lance un blastp à l'aide du logiciel blast+ sur la protéine et le génome rentrés en argument. Comme blast+ est un logiciel nous avons du lancer la ligne de commande dans le terminal à l'aide du module subprocess. Le résultat du blastp est un fichier .xml que nous allons parser pour récupérer le meilleur hit grâce à la prochaine fonction.
- **best_hit()** : cette fonction récupère le meilleur hit du blastp exécuté par **blastp()** en parsant grâce à NCBIXML le fichier .xml crée auparavant.

Pour cela nous avons eu un peu de mal sur la méthode à employer. Après quelques recherches nous avons abandonné l'idée de trouver une fonction récupérant uniquement le premier hit d'un blastp. Nous avons alors mis en place un système de compteur qui permet aussi de gérer le cas où la protéine n'a pas eu de hit lors du blastp (ceci arrive si le gène n'a pas été conservé dans un génome).

Dans le main :
Nous exécutons naturellement le blastp sur l'ensemble des 29 autres génomes de E. coli que nous mettons dans *list_genome_autre* via la fonction **recup_genome()**. Pour cela nous effectuons une boucle for sur *list_genome_autre* et nous blastons à chaque tour.
Nous allons mettre dans le dictionnaire *dict_proteine_info*, en clé les génomes et en valeur les informations sur le résultat (ou 'Pas de résultat') du blastp sur ce génome à partir de la protéine d'intérêt.

Partie 4 : Trouver la protéine en amont et en aval

Pour répondre à cette problématique nous avons défini 1 fonction :
- **amont_aval()** : cette fonction donne à partir d'une protéine dans un génome sa protéine en amont et sa protéine en aval. Pour cela nous utilisons le fichier d'information .tsv qu'on récupère en dataframe via **recup_data()** puis nous allons récupérer la protéine sur la ligne n-1 et sur la ligne n+1. En effet, le dataframe est trié dans l'ordre et donc nous obtenons comme cela soit les protéines en amont et en aval. Pour décider laquelle est la protéine en amont et laquelle est celle en aval nous faisons un test en fonction du brin sur lequel se trouve la protéine d'intérêt.

Dans le main :
Nous exécutons la fonction **amont_aval()** sur la protéine d'intérêt rentré par l'utilisateur et nous stockons la protéine en amont dans *protéine_amont* et celle en aval dans *protéine_aval*.

Partie 5 : Lancer un blastp sur les protéines en amont/aval

Pour répondre à la problématique nous utilisons les fonctions **blastp()** et **best_hit()** sur les deux variables créées juste avant. Nous allons, comme pour la protéine d'intérêt, créer un dictionnaire que nous allons appeler cette fois ci *dict_proteine_amont_info* (resp. *dict_proteine_aval_info*) où les clés seront les génomes et les valeurs les informations du meilleur hit du blastp sur *protéine_amont* (resp. *protéine_aval*) dans le génome.
Dans cette partie nous avons hésité à passer par un autre moyen qui était d'appliquer la fonction **amont_aval()** sur les protéines du dictionnaire *dict_proteine_info*.

Partie 6 : Interface utilisateur

A partir des informations récoltées via les fonctions précédentes nous avions pour ambition de représenter la synténie de manière graphique en utilisant pyGenomeViz.
Le principe de la librairie GenomeViz que nous avons utilisé ici est d'afficher des gènes sur un brin en définissant à l'avance la longueur totale de ce brin (par exemple 5000pb) puis en définissant les CDS des gènes et leur sens (par exemple (1000, 2500, -1) = CDS de la position 1000 à 2500 sur le brin -), une fois les gènes affichés nous pouvons raffiner l'affichage avec des légendes ou même des liens entre les différents gènes.

Pour cela nous avons défini 2 fonctions :
- **strand_protein()** : cette fonction permet de retourner -1 ou 1 en fonction de si la protéine est sur le brin – ou +. Ceci sert dans la fonction d'après car comme vu dans l'exemple plus haut pyGenomeViz utilise -1 et 1 comme identifiant pour – ou +.
- **info_affichage_genome()** : cette fonction est la fonction centrale de notre visualisation. Elle va récupérer les positions des 3 gènes dans le génome donné et le sens de leur brin (via la fonction **strand_protein()**) via les trois dictionnaires (*dict_proteine_info, dict_proteine_aval_info, dict_proteine_amont_info*).

La fonction va également normaliser la position des différents gènes pour l'afficher sur un "track" d'une longueur correspondant au début du premier gène et la fin du dernier gène en rajoutant 500 paires de bases de chaque côté pour avoir de la marge.
La fonction récupère également l'ID des protéines associées à chaque gène grâce encore une fois aux trois dictionnaires.
La fonction s'occupe également des exceptions comme la possibilité que les gènes ne soient pas présents dans le génome ou bien qu'ils soient trop éloignés pour pouvoir être affichés à la même échelle que tous les autres gènes.
En effet, nous avons rencontré de fortes difficultés à afficher ces cas de déplacement d'un des gènes dans le génome de plusieurs milliers de paires de bases. Lorsque nous le représentions nous avions des visualisations complètement démesurées. Nous n'avons pas réussi à résoudre ce problème mais nous avons bon espoir de l'améliorer d'ici la présentation orale.

Dans le main :
La partie dans le main permet, à partir des informations récupérées par la fonction précédentes, d'afficher la visualisation. Pour cela nous créons le track pour chaque génome et via des boucles nous affichons chaque gène avec ses informations (données par la fonction **info_affichage_genome()**).

Description de l'interface utilisateur :

A] Interface Notebook :

Pour exécuter le programme nous avons créé une interface notebook nommée interface.ipnyb. Celle-ci contient tous les détails sur comment lancer le programme ou bien quels packages installer. Notamment, l'interface permet de lancer le programme à partir de sa partie backup (de secours) ou bien à partir de Tkinter. Dans les deux cas tout est expliqué sur le fichier notebook directement.

B] Interface Tkinter :

Nous avons choisi Tkinter pour sa modularité et sa facilité d'accès comparée à ses concurrents. Nous voulons créer une interface graphique interopérable affichant les résultats de l'analyse en renseignant uniquement la protéine, le génome où elle se trouve et le système d'exploitation de l'utilisateur. Ce script peut être lancé à partir d'un Jupyter Notebook ou du terminal.

Description du contenu du script :

classe Appli : classe principale du script. On y définit les attributs associés à la fenêtre graphique app ainsi que des fonctions, pour éviter les éventuels problèmes de variables globales et naviguer dans l'interface.
__init__ (de Appli) : on y définit les états initiaux des attributs lors de la création d'un objet Appli. Ici le seul objet de la classe sera app, la fenêtre graphique.
creer_widgets() : cette fonction définit les widgets de la fenêtre tels que les barres de saisie capturant ce que l'utilisateur écrit, le menu dépliant pour l'OS et le bouton pour lancer l'analyse. On affiche tous ces widgets dès la création de la fenêtre graphique par l'appel de la fonction dans **__init__**.
submit() : cette fonction permet de lancer l'analyse si les arguments ont été bien choisis. Cette fonction est associée au bouton « Submit ». Nous commençons par récupérer les arguments donnés par l'utilisateur et vérifier leur présence dans nos données.
Nous créons ensuite une zone de texte où nous voulons afficher nos résultats d'analyse. Pour cela nous redirigeons la sortie standard du script vers cette zone à l'aide de la classe « StdoutRedirector ».
classe StdoutRedirector : redirige la sortie standard vers le widget en argument de la classe lorsqu'elle est appelée. La fonction **write()** permet d'insérer les résultats à la fin du widget en argument, ici étant la zone de texte prévue à cet effet.
Thread() : fonction du module threading qui permet de faire tourner un script par un autre.
Nous avions initialement créé le module de sorte à ce qu'il affiche les résultats en le lançant simplement comme en TP. Seulement nous n'arrivions pas à le lancer via un autre script. Nous avons donc modifié le main du module en fonction, ses arguments étant ce qu'on renseignait lorsque l'on lançait le module. Pour le lancer depuis l'interface, nous avons dû utiliser **Thread()**.
image : on ouvre l'image illustrant la synténie à partir de notre protéine d'intérêt dans une autre fenêtre. Nous détaillons le processus d'obtention de l'image dans la partie suivante.

Dans le main du script de l'interface, nous affichons la fenêtre graphique par l'appel de la classe Appli avec tous ses widgets en attributs. Le mainloop d'une classe met à jour constamment l'état des attributs définis avant lui. Il sert à rendre dynamique les variables tels que **StringVar()** qui stockent les informations écrites en temps réel par l'utilisateur, ou encore les boutons.

Critiques sur le projet :

Voici quelques améliorations de notre projet que nous n'avons pas pu effectuer par faute de temps :
- Utiliser moins de fonctions.
- Faire un système de gestion de chemin pour ne pas obliger l'utilisateur à mettre le dossier data dans le même dossier que les modules et l'interface.
- Gérer les erreurs dans nos fonctions et pas dans le main.
- Améliorer la visualisation de la synténie et trouver une solution pour afficher les gènes déplacés très loin de leur position initiale.
- Laisser la possibilité à l'utilisateur de sélectionner le nombre de gènes en amont et aval à analyser pour fiabiliser l'observation de la synténie.

Répartition des rôles :

Paulin : Partie 3, Partie 6 : Interface Tkinter
Morgan : Partie 3, Partie 6 : Visualisation de la synténie
Jules : Partie 1,2,4,5, Partie 6 : Interface Notebook

Pour l'accès à notre répertoire de travail voyez ici.