Projet Réseaux Biologiques en Python

rédigé par Emmanuelle Becker & Olivier Dameron

Semestre automne-hiver 2022

Table des matières

Ce projet est dédié à l'étude des graphes d'interactions entre protéines. Vous allez être amené·e à écrire des outils permettant de manipuler ces graphes, et à introduire au sein de ces graphes une notion d'interaction entre domaines protéiques.

Chapitre 1

Lecture d'un graphe d'interactions entre protéines

1.1 Préambule

Les graphes d'interactions qui vous seront fournis se trouvent dans des fichiers dont le format est le suivant : la première ligne indique le nombre d'interactions (le nombre de lignes du fichier -1), et chaque ligne suivante décrit une interaction. Ces interactions n'ont pas d'orientation ; le graphe d'interaction est donc un graphe non orienté.

1.1.1 Question de compréhension

Dessinez un petit graphe d'interactions (une dizaine), et écrivez le fichier qui y sera associé. Échangez votre dessin avec vos collègues, écrivez le fichier correspondant à leur graphe et déterminez si vous avez compris la même chose.

1.2 Structure de données pour stocker le graphe

Cette section a pour objectif de comparer différentes stratégies pour représenter un graphe d'interactions.

Une première manière de stocker ce graphe est d'utiliser un dictionnaire où les clés sont les sommets et les valeurs associées aux clés sont les voisins des sommets. On peut aussi imaginer de stocker ce graphe dans une liste de couples (X, Y) qui représentent toutes les interactions.

1.2.1 Question structure 1

Écrire une fonction qui lise un graphe d'interactions entre protéines dans un fichier tabulé et le stocke dans un dictionnaire. Le nom de la fonction sera `read_interaction_file_dict` et la fonction prendra en unique argument le nom du fichier à lire. Le dictionnaire créé sera retourné par la fonction.

1.2.2 Question structure 2

Écrire une fonction qui lise un graphe d'interactions entre protéines dans un fichier et le stocke dans une liste de couples. Le nom de la fonction sera `read_interaction_file_list` et la fonction prendra en unique argument le nom du fichier à lire. La liste créée sera retournée par la fonction.

1.2.3 Question structure 3

Écrire une fonction qui lise un graphe d'interactions entre protéines dans un fichier et le stocke dans une matrice d'adjacence (vous pouvez utiliser `numpy` pour les matrices en python, et wikipédia pour la définition de matrice d'adjacence).
Le nom de la fonction sera `read_interaction_file_mat` et la fonction prendra en unique argument le nom du fichier à lire. La matrice créée sera retournée par la fonction, ainsi que la liste ordonnée des sommets (parce que, évidemment, l'ordre des sommets est crucial pour lire correctement la matrice d'adjacence !).

1.2.4 Question structure 4

Écrire une fonction nommée `read_interaction_file`, qui a partir d'un fichier d'interactions, retourne un triplet `(d_int, l_int, m_int, l_som)` dont le premier élément est le dictionnaire représentant le graphe, le second élément est la liste d'interactions représentant le même graphe, le troisième élément est la matrice d'ajacence correspondant à ce graphe, et le dernier élément est la liste ordonnée des sommets.

1.2.5 Question structure 5

Pour un gros graphe d'interactions, quelle(s) stratégie(s) adopteriez-vous pour ne pas trop dégrader les performances de la fonction `read_interaction_file` ?

1.2.6 Question test des structures 1

Il est fondamental de tester le bon comportement de vos fonctions de lecture avant de poursuivre plus avant dans l'étude des réseaux d'interactions protéine-protéine. Vos fonctions ne sont pas utilisables si nous n'avons pas moyen de nous assurer qu'elles font effectivement ce pour quoi elles ont été conçues.
Dans un fichier à part, pour lequel vous choisirez un nom explicite, préparez toute une série de tests unitaires pour vérifier que vos fonctions ont le comportement que nous attendons d'elles. C'est le moment de vous souvenir des tests unitaires que nous avons employés l'an dernier [1].

1.2.7 Question test des structures 2

Écrire une fonction nommée `is_interaction_file` dont l'objectif est de vérifier que le fichier est bien au format attendu pour être lu correctement. Cette fonction prend en

1. `https://docs.python.org/fr/3/library/unittest.html`

argument un fichier d'interaction, et renvoie le booléen `true` is le format est correct et `false` sinon. Travaillez à partir de plusieurs fichiers tests, certains respectant les spécifications du format demandés, d'autres non. Par exemple :

1. fichier ne comportant pas la première ligne qui compte le nombre d'interactions ;
2. fichier vide ;
3. fichier dont la première ligne contient un nombre qui n'est pas le nombre d'interactions ;
4. fichier contenant une ligne qui ne comporte pas le bon nombre de colonnes...

1.3 Chapitre 1 : Structurez, commentez et déposez votre travail

1.3.1 Charge de travail chapitre 1

Pour cette première partie de projet, nous estimons la charge de travail à entre 2h et 4h de programmation (selon votre aisance), et 2h de nettoyage de code et commentaires. La création du projet git et son dépôt pourront vous prendre un quart d'heure de plus.

1.3.2 Consignes générales

1. Vous travaillerez en binôme en choisissant votre binôme au sein de votre parcours. Vous ne changerez pas de binôme au cours du semestre.
2. Vous vous organiserez pour envoyer la liste exhaustive de tous les binômes à votre enseignante avant la fin de la semaine.
3. Votre code doit être déposé sous git. Idéalement pas en une fois à la fin, mais de façon incrémentale au fur et à mesure que vous ajoutez de nouvelles fonctionnalités ou que vous corrigez des bugs.
4. Votre enseignante doit avoir accès à votre projet sous git, dès cette semaine (merci de m'envoyer le lien).
5. Quelque part de très visible, sur votre dépôt git, vous indiquerez les membres du binôme et votre identifiant sous git.
6. La gestion de projet fera partie intégrante de la notation.

1.3.3 Consignes de programmation

1. Vos fonctions devront être écrites propremment.
2. Vos noms de variables doivent être cohérents entre eux (pas de mélange de langue français - anglais, et comme le code est commencé en anglais, tous les noms de variables sont en anglais)
3. Vos noms de fonctions et de variables ont la même syntaxe (ici tout en minuscule avec des tirets du bas pour séparer les mots).
4. Vos noms de variables doivent contenir un suffixe indiquant leur type.

5. Vos noms de variables doivent faire moins de 20 caractères de long.
6. Toutes vos fonctions doivent être préfixées avec des commentaires qui indiquent le but de la fonction et sa signature. C'est sans doute le bon moment pour enfin lire la documentation sur les docstrings [2].
7. Vos fonctions ne doivent pas faire plus de 30 lignes (et les lignes doivent faire une taille raisonnable).

2. `https://www.python.org/dev/peps/pep-0257/`

Chapitre 2

Exploration du graphe d'interactions protéine-protéine

Dans les questions suivantes, vous aurez besoin de lire, voire d'écrire, des graphes d'interactions entre protéines. Il ne vous sera jamais précisé comment lire votre fichier d'interaction et quelle structure de données utiliser. Vous devrez trouver par vous-même, en fonction de la question posée, laquelle sera la plus adaptée parmi celles vues au chapitre précédent.

2.1 Exploration du graphe global

2.1.1 Question exploration 1

Écrire une fonction `count_vertices(file)` qui compte le nombre de sommets d'un graphe.

2.1.2 Question exploration 2

Écrire une fonction `count_edges(file)` qui compte le nombre d'arêtes d'un graphe.

2.1.3 Question nettoyage

Écrire une fonction `clean_interactome(filein, fileout)` qui lit un fichier contenant un graphe d'interactions protéine-protéine et y enlève (i) toutes les interactions redondantes, et (ii) tous les homo-dimères. Le graphe obtenu sera écrit dans un nouveau fichier au même format que le format de départ (posez-vous la question de savoir si ça ne vaut pas le coup d'écrire une ou plusieurs fonctions d'écriture d'un graphe dans un fichier).

2.1.4 Question test

Il est fondamental de tester le bon comportement de vos fonctions avant de poursuivre plus avant dans l'étude des réseaux d'interactions protéine-protéine. Vos fonctions ne sont pas utilisables si nous n'avons pas moyen de nous assurer qu'elles font effectivement ce

pour quoi elles ont été concues. Préparez toute une série de tests pour vérifier que vos fonctions ont le comportement que nous attendons d'elles.

2.2 Travail autour du degré des sommets

On nomme degré d'un sommet le nombre d'arêtes incidentes au sommet. Par exemple, dans le graphe de départ, si A est connecté à B et F, le degré du sommet A est 2.

2.2.1 Question degré 1

Écrire une fonction `get_degree(file, prot)` qui prend en argument un fichier contenant un graphe d'interactions protéine-protéine et le nom d'une proteine, et qui renvoie le degré de cette protéine dans le graphe.

2.2.2 Question degré 2

Écrire une fonction `get_max_degree(file)` qui renvoie le nom de la protéine de degré maximal ainsi que le degré de cette protéine.

2.2.3 Question degré 3

Écrire une fonction `get_ave_degree(file)` qui calcule le degré moyen des protéines du graphe.

2.2.4 Question degré 4

Écrire une fonction `count_degree(file, deg)` qui calcule le nombre de protéines du graphe dont le degré est exactement égal à `deg`.

2.2.5 Question degré 5

Écrire une fonction `histogram_degree(file, dmin, dmax)` qui calcule, pour tous les degrés `d` compris entre `dmin` et `dmax`, le nombre de protéines ayant un degré `d`. Essayer d'afficher le résultat sous la forme d'un histogramme en comme par exemple :

```
1 **
2 **
3 **
```

Que constatez-vous ? Quelle analyse pouvez-vous faire au vu de cette distribution ?

2.3 Structurez, commentez et déposez votre travail

2.3.1 Charge de travail

Pour ce chapître de projet, nous estimons la charge de travail à entre 2h et 4h de programmation (selon votre aisance), et 1h de nettoyage de code et commentaires. La

compréhension des notions nouvelles dans le projet (sommets, arêtes, degrés) pourrait vous predre une demi-heure de lecture en plus (maximum).

2.3.2 Consignes

Les consignes générales et de programmation du chapitre précédent continuent de s'appliquer. Vos mises à jour du code doivent être visible sur l'architecture du projet git, à l'adresse que vous avez communiquée à l'enseignante la semaine passée.

Chapitre 3

Modification des spécifications, python orienté objet

3.1 Problème actuel

Vous aurez peut-être noté que notre organisation n'est pas optimale. En effet, le graphe d'interaction protéine-protéine peut-être lu sous forme de liste, de dictionnaire, voire de matrice. Vous avez aussi vu que certaines structures de données sont plus adaptées à certaines fonctions d'exploration que d'autres. C'est la raison pour laquelle, jusqu'à présent, dans chaque fonction d'exploration, on commence par lire le fichier avec la méthode de stockage adaptée. Mais ce n'est pas optimal, car du coup, on passe son temps à relire le même fichier...

3.1.1 Question de compréhension

Lors de l'execution de la fonction `histogram_degree`, combien de fois a t-on lu le fichier d'interactions ?

3.2 Créons un objet interactome

Nous proposons dans cette section de créer un objet (comme en programmation orientée objet l'année dernière). Cet objet représentera un réseau d'interactions protéine-protéine (appelé interactome), et les attributs de la classe seront la liste des interactions, le dictionnaire des interactions, et la matrice d'adjacence des interactions. De cette façon, on pourra lire le fichier un nombre raisonnable de fois pour construire l'objet, et ensuite manipuler la structure de données qui nous convient sans avoir à relire le fichier à chaque fois...

3.2.1 Lecture...

Allez lire le chapitre 19 du livre Python pour les sciences de la vie.

3.2.2 Modification

Créez une nouvelle branche dans votre dépôt git. Dans cette nouvelle branche, créez une classe Interactome :

- avec un attribut `int_list` qui stockera l'interactome sous la forme d'une liste d'interaction ;
- avec un attribut `int_dict` qui stockera l'interactome sous la forme d'un disctionnaire d'adjacence
- avec un attribut `proteins` qui stockera la liste des interactants.

3.2.3 Constructeur

Ajoutez un constructeur, qui prend en entrée un fichier d'interactions, et remplit les différents attributs.

3.2.4 Méthodes...

Transformez toutes les méthodes que vous avez écrites précédemment en membres de la classe `Interactome`.

3.3 Métriques simples sur l'interactome : travail sur la densité

3.3.1 Question densité

Dans le cas d'un graphe non orienté simple $G = (V, E)$, la densité est le rapport :

$$D = \frac{2\,|E|}{|V| \cdot (|V| - 1)}$$

Il représente le nombre d'arêtes présentes, par rapport au nombre d'arêtes total qu'il pourrait théoriquement y avoir. Ajoutez une méthode `density()` qui renvoie la densité de l'interactome.

3.3.2 Question coefficient de clustering

En théorie des graphes et en analyse des réseaux sociaux, le coefficient de clustering d'un graphe (aussi appelé coefficient d'agglomération, de connexion, de regroupement, d'agrégation ou de transitivité), est une mesure du regroupement des nœuds dans un réseau. Plus précisément, ce coefficient est la probabilité que deux nœuds soient connectés sachant qu'ils ont un voisin un commun. C'est l'un des paramètres étudiés dans les réseaux sociaux (par exemple) : les amis de mes amis sont-ils mes amis ?

Le coefficient de clustering local d'un nœud `i` est défini comme :

$$C_i = \frac{|\text{triangles de sommet } i|}{|\text{paires de voisins distincts de } i|}$$

C'est la fraction de ses paires de voisins connectés, égale à 0 si $d_i \leq 1$ par convention. Ajoutez une méthode `clustering(prot)` qui renvoie le coefficient de clustering du sommet `prot` au sein de l'interactome.

3.4 Testez vos méthodes

Dans un second fichier, écrivez un programme principal où vous testez vos nouvelles méthodes. Pensez à bien travailler vos tests : ils doivent nous convaincre que votre code fonctionne...

3.5 Structurez, commentez et déposez votre travail

3.5.1 Charge de travail

Nous estimons la charge de travail correspondant au chapitre 3 à 4h de programmation, voire moins.

3.5.2 Consignes

Les consignes générales et de programmation du chapitre précédent continuent de s'appliquer. Vos mises à jour du code doivent être visible sur l'architecture du projet git, à l'adresse que vous avez communiquée à l'enseignante la semaine passée.

Chapitre 4

Graphes aléatoires

4.1 Graphes aléatoires de Erdös-Renyi

Dans ce modèle de graphe aléatoire, souvent noté $G(n, p)$, chacune des $n(n-1)/2$ arêtes est présente avec probabilité p et absente avec probabilité $1-p$. Le nombre N_p d'arêtes de $G(n, p)$ suit la loi binomiale de paramètres $n(n-1)/2$ et p.

4.1.1 Question graphe aléatoire

Ajoutez une méthode $graph_er(p)$ à votre classe `Interactome` qui génère un graphe aléatoire de $Erds-Renyi$ de probabilité p et le stocke dans l'objet `Interactome` courant.

4.1.2 Test

Afin de vérifier votre méthode, construisez un graphe aléatoire de paramètre $p = 0.3$ et calculez ensuite la distribution des degrés des sommets. Pour ce faire, vous devriez vous servir de méthodes implémentées au chapitre 2...

4.2 Graphes aléatoires de Barabasi-Albert

Le modèle de Barabási–Albert (BA) est un algorithme pour la génération aléatoire de réseaux sans échelle à l'aide d'un mécanisme d'attachement préférentiel. Le détail de l'algorithme a été vu en cours.

4.2.1 Question graphe aléatoire

Ajoutez une méthode $graph_ba()$ à votre classe `Interactome` qui génère un graphe aléatoire de $Barabasi-Albert$ de et le stocke dans l'objet `Interactome` courant.

4.2.2 Test

Afin de vérifier votre méthode, construisez un graphe aléatoire et calculez ensuite la distribution des degrés des sommets. Pour ce faire, vous devriez vous servir de méthodes implémentées au chapitre 2...

4.3 Structurez, commentez et déposez votre travail

4.3.1 Charge de travail

Pour ce chapitre de projet, nous estimons la charge de travail à entre 2h et 4h de programmation (selon votre aisance).

4.3.2 Objectifs

1. Continuez sur votre lancée : on reste en python orienté objet.
2. Vos méthodes devront être écrites propremment.
3. Vos noms de variables doivent être cohérents entre eux (tous les noms de variables sont en anglais)
4. Vos noms de méthodes et de variables ont la même syntaxe (ici tout en minuscule avec des tirets du bas pour séparer les mots).
5. Vos noms de variables doivent contenir un suffixe indiquant leur type.
6. Vos noms de variables doivent faire moins de 20 caractères de long.
7. Toutes vos méthodes doivent être préfixées avec des commentaires qui indiquent le but de la fonction et sa signature.
8. Aucune fonction ne doit faire plus de 30 lignes.
9. Votre code doit être déposé sous git.
10. Votre mise à jour du code doit être visible sur l'architecture du projet git.

Chapitre 5

Calcul des composantes connexes d'un graphe d'interactions entre protéines

5.1 Travail sur les composantes connexes

On dit qu'un graphe est connexe si quels que soient les sommets X et Y, il existe un chemin qui relie X à Y. Malheureusement, les graphes d'interactions entre protéines sont rarement connexes. On cherche alors à savoir combien de composantes connexes composent ces graphes... Les questions suivantes listent des méthodes à écrire et leur spécification. Soyez rusés, lisez le sujet jusqu'au bout et réfléchissez à l'ordre optimal pour implémenter vos méthodes...

5.1.1 Question préliminaire

Recherchez sur internet la définition de « composante connexe ».

5.1.2 Question composantes connexes

Écrivez une fonction `count_CC()` qui calcule le nombre de composantes connexes d'un graphe, et donne pour chacune d'elle sa taille (c'est à dire son nombre de protéines).

5.1.3 Question composantes connexes

Écrivez une fonction `write_CC()` qui écrive dans un fichier les différentes composantes connexes d'un graphe. Vous utiliserez le format suivant :

1. une ligne par composante connexe
2. le premier élément de la ligne est la taille de la composante connexe,
3. ensuite vous ajouterez la liste des sommets qui composent cette composante connexe.

5.1.4 Question composantes connexes

Écrivez une fonction `extract_CC(prot)` qui renvoie tous les sommets de la composante connexe de `prot`.

5.1.5 Question composantes connexes

Écrivez une fonction `compute_CC()` qui renvoie une liste `lcc` dont chaque élément `lcc[i]` correspond au numéro de composante connexe de la protéine à la position `i` dans la liste des protéines du graphe (qui est un attribut de notre classe).

5.2 Structurez, commentez et déposez votre travail

5.2.1 Charge de travail

Pour ce chapitre de projet, nous estimons la charge de travail à entre 4h et 8h de programmation (selon votre aisance), et 2h de nettoyage de code et commentaires.

5.2.2 Objectifs

1. Continuez sur votre lancée : on reste en python orienté objet.
2. Vos méthodes devront être écrites propremment.
3. Vos noms de variables doivent être cohérents entre eux (tous les noms de variables sont en anglais)
4. Vos noms de méthodes et de variables ont la même syntaxe (ici tout en minuscule avec des tirets du bas pour séparer les mots).
5. Vos noms de variables doivent contenir un suffixe indiquant leur type.
6. Vos noms de variables doivent faire moins de 20 caractères de long.
7. Toutes vos méthodes doivent être préfixées avec des commentaires qui indiquent le but de la fonction et sa signature.
8. Aucune fonction ne doit faire plus de 30 lignes.
9. Votre code doit être déposé sous git.
10. Votre mise à jour du code doit être visible sur l'architecture du projet git.