TP #7 — Graphes #1

Lecture/écriture de graphes. Si vous le souhaitez, vous pouvez (notamment lors des tests !) utiliser les fonction de lecture et écriture de graphe du précédent TP. Si besoin, une proposition de correction de ces fonctions est disponible ici.

Exercice 1. *Premiers pas*

Dans tout cet exercice, on considère des graphes (possiblement orientés) dont les sommets sont représentés par des entiers entre 0 et $\#S - 1$, avec $\#S$ le nombre de sommets.

1. Écrivez une fonction `list2matrix(G)` qui prend en entrée un graphe G représenté par liste de listes d'adjacence et qui en renvoie une représentation par matrice d'adjacence.
2. Écrivez une fonction `matrix2list(G)` qui réalise l'opération inverse.
3. Testez.
4. Écrivez une fonction `gen_complete(n)` qui construit et renvoie une représentation sous forme de matrice d'adjacence de K_n, le graphe complet non-orienté à n sommets
5. Testez.
6. Écrivez une fonction `gen_cycle(n, directed)` qui construit et renvoie une représentation sous forme de liste de listes d'adjacence de C_n, « le » graphe cycle à n sommets. Ce graphe devra être non-orienté si l'argument *directed* vaut `False`, et orienté sinon (dans un sens au choix).
7. Testez.
8. Écrivez une fonction `degree(G)` qui prend en entrée un graphe orienté représenté par liste de listes d'adjacence et en renvoie le degré (max).
9. Testez.

On définit le graphe *miroir* (ou *transposé*) d'un graphe orienté $G = (S, A)$ comme le graphe $G^{\leftarrow} = (S, A^{\leftarrow})$ où $A^{\leftarrow} := \{(u, v) \mid (v, u) \in A\}$. Autrement dit, $G^{\leftarrow}$ possède un arc orienté u, v ssi. G possède un arc orienté v, u.

10. Pourquoi définir le miroir d'un graphe non-orienté aurait peu d'intérêt ?
11. Écrivez une fonction `mirror_list(G)` qui prend en entrée un graphe G représenté par liste de listes d'adjacence et construit et renvoie une représentation (toujours par liste de listes d'adjacence) de son miroir.
12. Testez.
13. Écrivez une fonction `mirror_matrix(G)` qui prend en entrée un graphe G représenté par matrice d'adjacence et construit et renvoie une représentation (toujours par matrice d'adjacence) de son miroir.

14. Testez.

Exercice 2. *Coloration*

Dans tout cet exercice, on considère des graphes non orientés sans boucle dont les sommets sont représentés par des entiers entre 0 et $\#S - 1$, avec $\#S$ le nombre de sommets.
Une *coloration* d'un graphe est une fonction $S \to \mathbb{N}$ qui associe une couleur (représentée par un entier) à chaque sommet.

1. Proposez une façon de représenter une coloration en Python. Que pourriez-vous changer dans le cas où les sommets ne seraient pas représentés par des entiers (mais par exemple, par des chaînes de caractère ?)

Une coloration est dite valide si aucun sommets adjacents ne sont coloriés en la même couleur.

2. Écrivez une fonction `is_valid(G, C)` qui prend en entrée un graphe G représenté par liste de listes d'adjacence et une coloration C représentée de façon appropriée, et qui renvoie `True` si la coloration est valide, et `False` sinon.
3. Testez (par exemple avec le graphe C_4 et une coloration de tous les sommets à `0` ; de tous les sommets de couleur différente ; où les sommets alternent de couleur dans le cycle).

Un graphe colorié est souvent utilisé pour représenter une allocation de ressource : les couleurs représentent des ressources disponibles, les sommets des consommateurs de ressource, et le fait que deux sommets soient adjacents le fait qu'ils ne peuvent pas utiliser la même ressource (en même temps).
Par exemple, les sommets peuvent représenter des cours, les couleurs des salles de classe, et une arête le fait que deux cours ont lieu en même temps (et ne peuvent donc pas occuper la même salle).
Pour ce type de modélisation, l'objectif est donc souvent de trouver une coloration valide qui *minimise* le nombre de couleurs utilisées. Malheureusement, c'est un problème difficile en général…

4. Écrivez une fonction `greedy_col(G)` qui prend en entrée un graphe G représenté par liste de listes d'adjacence et qui renvoie une coloration valide pour G. On pourra pour cela utiliser l'algorithme « glouton » suivant, qui colorie les sommets de façon incrémentale : on parcourt les sommets dans un certain ordre (à préciser), et chaque sommet est colorié avec la plus petite (en tant qu'entier) couleur avec laquelle il peut être colorié sans invalider la coloration (c'est à dire, avec la plus petite couleur qui n'est pas déjà utilisée pour colorier ses éventuels voisins déjà coloriés).
Vous pouvez éventuellement vous aider de fonctions intermédiaires, par exemple une fonction qui renvoie pour un sommet la liste (éventuellement

vide) des couleurs de ses voisins déjà coloriés, ainsi qu'une fonction qui renvoie le plus petit entier naturel qui n'est pas présent dans une liste d'entiers naturels. On peut aussi être plus direct en utilisant un dictionnaire à bon escient…

5. Testez (par exemple en coloriant C_4, C_5, C_6, K_4…).

On dit d'une coloration d'un graphe qu'elle est *optimale* si elle utilise le nombre minimum de couleurs nécessaires pour le colorier. Ce minimum est appelé *nombre chromatique* du graphe.

6. Montrez que la notion de nombre chromatique est bien définie.
7. Donnez un encadrement du nombre chromatique pour un graphe à N sommets.
8. Montrez sur un exemple d'exécution (éventuellement « à la main ») que votre fonction *greedy_col* ne renvoie pas forcément une coloration optimale.

On peut montrer que pour tout graphe, il existe toujours (au moins) *un* ordre de parcours (qui dépend du graphe) tel que *greedy_col* renvoie une coloration optimale.

9. Décrivez (sans nécessairement l'implémenter) un algorithme qui prend un graphe G en entrée et en calcule et renvoie une coloration optimale. Quel est le coût de cet algorithme en fonction du nombre de sommets de G ? Pouvez-vous dire que cet algorithme est « efficace » ?

N.B. On ne connaît pas d'algorithme efficace pour le problème de coloration de graphe en général. On sait par contre colorier efficacement des graphes possédant une certaine structure, par exemple les graphes bipartis (cf. prochain TP ?).

Exercice 3. *Cycle hamiltonien*

Dans tout cet exercice, on considère des graphes non orientés sans boucles dont les sommets sont représentés par des entiers entre 0 et $\#S - 1$, avec $\#S$ le nombre de sommets.

1. Écrivez une fonction `is_cycle(G, C)` qui prend en entrée un graphe G représenté par matrice d'adjacence et un tuple de sommets deux à deux distincts C, et qui renvoie `True` si G possède un cycle `C[0]` → `C[1]` → `...` → `C[len(C) - 1]` → `C[0]`, et `False` sinon.
   (On rappelle que dans le cas non orienté, on ne considère pas que $u \to v \to u$ est un cycle.)
2. Testez

On veut maintenant écrire une fonction qui décide si un graphe possède un cycle « hamiltonien », c'est à dire un cycle passant exactement une fois par tous ses sommets. Pour cela, on ne propose pas mieux qu'une approche par recherche exhaustive, qui consiste à énumérer tous les cycles possible et tester si l'un deux est présent dans le graphe.

3. Pourquoi peut-on se contenter de tester les cycles qui « commencent » au sommet 0 ? Combien de cycles cela fait-il ?

Le module Python itertools contient une fonction permutations permettant d'itérer sur toutes les permutations de son argument dans l'ordre lexicographique :

```
from itertools import permutations
for p in permutations([1,2,3]):
    # p vaudra successivement
    # (1,2,3) ; (1,3,2) ; (2,1,3)
    # (2,3,1) ; (3,1,2) ; (3,2,1)
```

4. Écrivez une fonction `is_hamiltonian(G)` qui prend en entrée un graphe G représenté par matrice d'adjacence et qui renvoie `True` si celui-ci possède un cycle hamiltonien, et `False` sinon. Cette fonction ne devra pas tester plus de permutations que nécessaire. (Vous pouvez aussi dévier de ces spécifications pour renvoyer un cycle hamiltonien quand il en existe un.)
5. Testez.