TP13.1 - Graphes, manipulations

→ *Q.1) Récupérez le fichier* `TP13.py` *sur cahier de prépa dans lequel vous pourrez sauvegarder votre travail.*

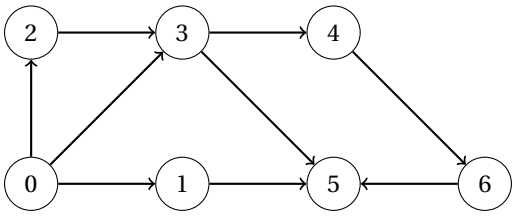
A) Dictionnaires et matrices

On a vu dans le cours qu'on peut représenter un graphe de deux façons différentes :

— avec un dictionnaire de listes d'adjacence : les clés sont les sommets du graphe et la valeur associée à une clé est la liste des voisins du sommet correspondant ;

— avec une matrice d'adjacence : c'est une liste de listes de booléens.

→ *Q.2) Créer une variable* `glist` *contenant la représentation par listes d'ajacence du graphe suivant, puis une variable* `gmat` *contenant la matrice d'adjacence du même graphe.*



→ *Q.3) Écrire une fonction* `sont_voisins(A, i, j)` *qui étant donnés une matrice d'adjacence* `A`*, et deux sommets* `i` *et* `j` *renvoie* `True` *si les sommets sont voisins et* `False` *sinon.*

→ *Q.4) Écrire une fonction* `voisins(A, i)` *qui étant donnés une matrice d'adjacence* `A`*, et un sommet* `i` *renvoie une liste composée de tous les voisins de* `i`*.*

→ *Q.5) Écrire une fonction* `listes_adjacence(A)` *qui étant donnée une matrice d'adjacence* `A` *renvoie le dictionnaire des listes d'adjacence du graphe associé.*

→ *Q.6) Écrire une fonction* `matrice_adjacence(G)` *qui étant donnée un dictionnaire* `G` *composé des listes d'adjacence d'un graphe renvoie sa matrice d'adjacence.*
On pourra supposer pour simplifier que les clés de `G` *sont des entiers de* `0` *à* `len(G)-1`*.*

→ *Q.7) Écrire une fonction* `ajout_sommet_mat(A)` *qui prend en paramètre une matrice d'adjacence et modifie* `A` *pour ajouter un sommet isolé au graphe.*

→ *Q.8) Écrire une fonction* `ajout_arc_mat(A, a)` *qui prend en paramètre une matrice d'adjacence et modifie* `A` *pour ajouter l'arc* `a` *au graphe ; l'arc* `a` *est représentée par la liste des deux indices des sommets dans la matrice.*

→ *Q.9) Écrire une fonction* `supprime_arc_mat(A, a)` *qui prend en paramètre une matrice d'adjacence et modifie* `A` *pour supprimer l'arc* `a` *du graphe.*

→ *Q.10) Écrire une fonction* `ajout_sommet_list(G, s)` *qui prend en paramètre un dictionnaire d'adjacence et modifie* `G` *pour ajouter un sommet isolé au graphe.*
Attention, si le sommet `s` *est déjà dans* `G`*, la fonction doit renvoyer une erreur.*

→ *Q.11) Écrire une fonction* `ajout_arc_list(G, a)` *qui prend en paramètre un dictionnaire d'adjacence et modifie* `G` *pour ajouter l'arc* `a` *au graphe ; l'arc* `a` *est représentée par la liste des sommets à joindre.*
Attention, si un des sommets de `a` *n'est pas un sommet du graphe, ou bien si* `a` *est déjà dans le graphe, la fonction doit renvoyer une erreur.*

→ *Q.12) Écrire une fonction* `supprime_arc_list(G, a)` *qui prend en paramètre un dictionnaire d'adjacence et modifie* `G` *pour supprimer l'arc* `a` *du graphe.*
Attention, si `a` *n'est pas dans le graphe, la fonction doit renvoyer une erreur.*

B) Coloration de graphes

Une coloration d'un graphe consiste à associer à chaque sommet du graphe une couleur de sorte que deux sommets voisins n'aient pas la même. Nous allons représenter une couleur par un entier naturel et travailler avec des listes d'adjacence. Nos graphes ne sont pas orientés.

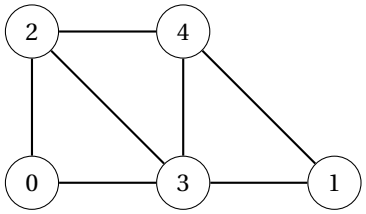
On souhaite écrire une fonction `coloration(G:dict) -> dict` qui prend en paramètre un graphe sous forme d'un dictionnaire de listes d'adjacence et renvoie un dictionnaire dont les clés sont les sommets du graphe et la valeur associée à chaque sommet est un entier représentant une couleur. On va pour cela utiliser un algorithme glouton :

— on récupère la liste des sommets de `G` ;

— pour chaque sommet dans la liste, on choisit le plus petit entier qui n'est pas utilisé par ses voisins.

→ *Q.13) Écrire une fonction* `plus_petit(L)` *qui renvoie le plus petit entier naturel non présent dans* `L`.
On essayera d'avoir une complexité linéaire en la taille de la liste `L`.

→ *Q.14) Écrire la fonction* `coloriage(G)` *voulue puis tester cette fonction sur le graphe :*



La fonction précédente ne renvoie pas le coloriage optimal : elle va parfois utiliser trop de couleurs par rapport au minimum nécessaire. Pour améliorer l'algorithme glouton, on commence par trier la liste des sommets du graphe par ordre décroissant de degré.

→ *Q.15) Écrire une fonction* `degre(G, s)` *qui renvoie le degré du sommet* `s` *dans le graphe* `G`.

Le mot clef `lambda` permet de créer à la volée des fonctions anonymes. La syntaxe est : `lambda arguments:expression`.
Il peut y avoir plusieurs arguments (séparés par une virgule) mais une seule expression qui est renvoyée par la fonction.
Par exemple `(lambda x,y : x+y)(1,1)` renvoie 2.
La fonction `sorted` permet de trier une liste : elle prend en argument une liste et en argument optionnel une fonction qui s'applique sur les éléments de la liste à trier. Le tri est fait en utilisant les valeurs de retour de la fonction.
Par exemple, `sorted([2, 0, 4, 9, 5], key = (lambda n : n%3))` trie la liste en utilisant la valeur du reste modulo 3 des éléments.

→ *Q.16) Écrire une fonction* `tri_degre(G)` *qui renvoie la liste des sommets de* `G` *triée par ordre décroissant de degré.*

→ *Q.17) Écrire une fonction* `coloriage_bis(G)` *qui implémente le nouvel algorithme de coloration puis comparer le résultat obtenu sur le graphe précédent.*