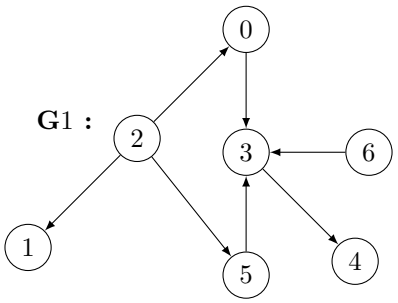
TP 12 - Parcours de graphes

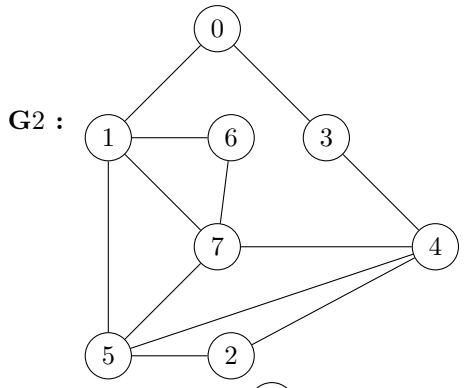
Dans ce TP, on aborde l'implémentation des parcours de graphes et quelques unes de leurs applications. Dans la suite, on suppose que le graphe $G = (S, A)$ est représenté par listes d'adjacence et que ses sommets sont les entiers de 0 à $n - 1$ où $n = |S|$.

Question 1. Définir des variables représentant les graphes suivants :

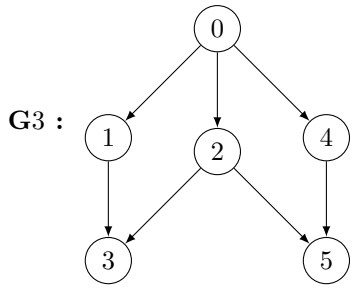
G1 :



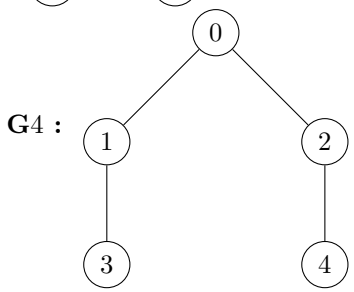
G2 :



G3 :



G4 :



1 Algorithmes de parcours

On commence par coder les parcours en largeur et profondeur d'un graphe (orienté ou non). On rappelle que le module `deque`, à importer depuis la librairie `collections`, permet une implémentation des piles et des files à l'aide des opérations `append`, `appendleft`, `pop` et `popleft`.

Question 2. Appliquer à la main le parcours en largeur à chacun de ces graphes en partant de 0. Présenter la trace des structures de données `atraiter` et `dejavu`. C'est-à-dire décrire `atraiter` et `dejavu` à chaque étape du parcours.

Question 3. Ecrire une fonction `parcoursLargeur` qui étant donné un graphe affiche (en utilisant `print`) les sommets visités depuis le sommet $s0$ lors d'un parcours en largeur de ce graphe.

Question 4. Appliquer à la main le parcours en profondeur à chacun de ces graphes en partant de 0. Présenter la trace des structures de données `atraiter` et `dejavu`.

Question 5.
1. Que suffit-il de modifier dans l'algorithme de parcours en largeur précédent pour obtenir un parcours en profondeur ?
2. Ecrire une fonction `parcoursProfondeur` qui étant donné un graphe affiche (en utilisant `print`) les sommets visités depuis le sommet $s0$ lors d'un parcours en profondeur de ce graphe.

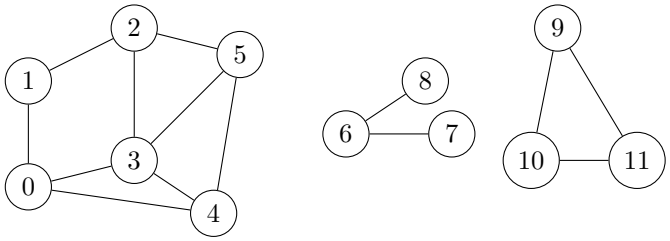
Il est possible de coder le parcours en profondeur de manière récursive. De cette manière, la pile `atraiter` de la version itérative est alors remplacée par la pile des appels récursifs. Cela permet en autre d'éviter d'empiler plusieurs fois un même sommet. Pour ce faire, on utilise une fonction auxiliaire récursive, c'est-à-dire une fonction récursive définie à l'intérieur de l'algorithme de parcours en profondeur, permettant de parcourir les voisins d'un sommet.

Question 6. Ecrire une version récursive du parcours en profondeur `parcoursProfondeurRec`.

2 Composantes connexes

On considère un graphe $G = (S, A)$ non orienté dont on veut calculer les composantes connexes.

Question 7.
1. Définir une variable représentant le graphe suivant :



2. Ce graphe est-il connexe ? Combien possède-t-il de composantes connexes ? Lesquelles ?
3. Lancer les parcours précédents sur cet arbre à partir de 0, de 7 et 11. Que constate-t-on ?

On souhaite connaître le nombre de composantes connexes ainsi que la répartition des sommets du graphe au sein de ces composantes connexes. Pour ce faire, on propose la stratégie suivante :

- on lance un parcours en profondeur du graphe à partir d'un sommet $s0$ (arbitraire),
- on associe à chaque sommet visité lors de ce parcours la composante connexe numéro 1,
- puis si tous les sommets n'ont pas été visités, on lance un nouveau parcours à partir d'un sommet $s1$ non déjà visité en associant à tous les sommets visités par ce nouveau parcours la composante connexe numéro 2 etc ...

La fonction doit renvoyer un entier désignant le nombre total de composantes connexes et une liste de taille $n = |S|$ qui associe à chaque sommet, le numéro de la composante connexe dans laquelle il se trouve.

Question 8. Ecrire une fonction `compConnexesNO` qui renvoie un entier indiquant le nombre total de composantes connexes du graphe ainsi qu'une liste telle que la case d'indice `i` contienne le numéro de la composante connexe du sommet `i`.

Le calcul des composantes fortement connexes d'un graphe orienté est plus délicat, il peut se faire à l'aide de l'algorithme de Kosaraju impliquant deux parcours en profondeur : le premier dans le graphe `G` et le second dans son graphe transposé, c'est-à-dire graphe dans lequel le sens de tous les arcs a été inversé.

3 Plus court chemin : affaire Cicéron

On vient d'apprendre que le maître espion allemand Ludwig Carl Moyzisch a récemment reçu des documents top secrets des alliés. Ce n'est pas une surprise, car ces documents sont des faux. Ils ont été créés de toutes pièces par les services de renseignements alliés pour confondre un espion présumé à l'ambassade de Grande Bretagne à Ankara, en Turquie. Cela faisait un moment que l'un des domestiques de l'ambassade, Elyesa Bazna, était soupçonné d'exfiltrer des documents de l'ambassadeur à destination des allemands. Or, seul ce dernier a pu avoir en sa possession les documents truqués, cela prouve sa culpabilité.

Il reste toutefois une dernière chose à déterminer : le réseau par lequel Bazna parvient à faire parvenir à Moyzisch les documents volés. Cette tâche vous revient, en tant qu'analyste. Heureusement, vous bénéficiez d'une documentation abondante sur les relations des deux individus sus-cités, mais également de tous les occidentaux présents à Ankara, sous forme d'un fichier indiquant que deux personnes se connaissent. Dans ce fichier, les prénoms et noms de deux personnes se connaissant apparaissent sur la même ligne séparés par un tiret `-`, par exemple une ligne peut être : `Paul Lefèvre - Hubert Bonisseur de La Bath`. Cela devrait être suffisant pour déterminer le cheminement possible des documents entre Elyesa Bazna et Ludwig Carl Moyzisch.

Un dernier point important : compte tenu de la rapidité avec laquelle les documents ont abouti dans les services allemands, il paraît certain que parmi les différents chemins possibles entre les deux individus, c'est le plus court qui représente le réseau à démanteler.

L'objectif de cette partie est d'indiquer les noms des différents intermédiaires par qui les documents ont circulé, entre Bazna et Moyzisch dans le bon ordre, séparés par des virgules. Les noms des deux protagonistes connus ne doivent pas être mentionnés.

Par exemple : `Austin Powers, Paul Lefèvre, Hubert Bonisseur de La Bath`.

Question 9. Ecrire une fonction `extraireDonnees` qui prend en argument l'adresse du fichier et renvoie la liste des listes de taille 2 d'individus se connaissant. Par exemple, si les individus Paul Lefèvre et Hubert Bonisseur de La Bath sont des occidentaux présents à Ankara et qui se connaissent, le fichier comporte une ligne contenant `Paul Lefèvre - Hubert Bonisseur de La Bath` et la liste obtenue contient la liste de taille deux `[Paul Lefèvre, Hubert Bonisseur de La Bath]`.

Pour démanteler le réseau d'espions allemands, on souhaite modéliser les liens entre individus à l'aide d'un graphe dont les sommets sont les individus présents dans le fichier et les arêtes indiquent les liens de connaissance.

Dans ce TP, comme les sommets de ce graphe sont étiquetés par des chaînes de caractères (et non des entiers), on le représente par un dictionnaire de listes d'adjacence. Autrement dit, un sommet s du graphe, i.e. une chaîne de caractère, est une clé du dictionnaire et sa valeur associée est sa liste d'adjacence i.e. la liste de ses voisins i.e. une liste de chaînes de caractères. Par exemple, le premier graphe de la question 1 (composé d'entiers) est représenté par le dictionnaire suivant

`dico1 = {0:[3], 1:[], 2:[0,1,5], 3:[4], 4:[], 5:[3], 6:[3]}`.

Question 10. Donner la représentation sous forme de dictionnaire d'adjacences des trois autres graphes de la question 1.

Question 11. Ecrire une fonction `creer_graphe` qui prend en argument une liste de listes de taille 2 et qui renvoie le dictionnaire de listes d'adjacence représentant le graphe correspondant.

Question 12. Ecrire une fonction `parcourslargeur` qui prend en entrées un graphe `g` représenté par dictionnaire de listes d'adjacence et un sommet source `s0` et qui renvoie le dictionnaire des prédécesseurs obtenu lors d'un parcours en largeur du graphe `g` depuis `s0`.

Question 13. Ecrire une fonction `afficher` qui prend entrée une liste et affiche ses éléments un par un séparés par des virgules.

Question 14. Ecrire une fonction `filiere` qui prend en entrées un graphe `g`, un sommet source `s0` et un sommet de fin `sfin` et qui affiche, à l'aide de la fonction `afficher`, les sommets composants le plus court chemin allant de `s0` à`sfin` dans le graphe `g`. Les sommets `s0` et `sfin` ne doivent pas être affichés.

4 Ordre topologique

Soit $G = (S, A)$ un graphe orienté sans cycle. On appelle **ordre topologique** sur G une relation d'ordre totale sur les sommets, notée $<$, telle que pour tout $s, t \in S$, $(s, t) \in A \Rightarrow s < t$. Comme l'ordre est total, le graphe est connexe. On suppose que cet ordre est cohérent i.e. qu'il existe un sommet s minimale pour cet ordre.

L'ordre topologique permet, notamment, d'implémenter un tri topologique permettant de définir l'ordre de réalisation de tâches ayant des dépendances entre elles. Par exemple, lors de la compilation des modules d'un programme, certains modules doivent être exécutés après certains autres. On peut modéliser le problème par un graphe dont les sommets sont les modules du programme en créant un arc de x vers y quand un module x dépend d'un module y. Ainsi lors de la compilation, il faudra lancer x après y car $(x, y) \in A$. Autrement dit, il faudra exécuter les modules dans l'ordre décroissant d'un ordre topologique.

Question 15. Ecrire une fonction `ordreTopo` prenant en entrée un graphe orienté sans cycle et renvoyant une liste énumérant les sommets du graphe dans un ordre topologique par ordre décroissant. On pourra effectuer un parcours en profondeur récursif avec un "bon" post-traitement des sommets (i.e. traiter un sommet après son parcours).

Par exemple, sur le graphe orienté numéro 1, un ordre topologique est `[4, 3, 0, 1, 5, 2, 6]`. Pour le graphe orienté numéro 3, on peut obtenir `[3, 1, 5, 2, 4, 0]`.

5 Détection de cycles

On souhaite détecter la présence d'un cycle dans un graphe orienté. Une stratégie consiste à effectuer un parcours en profondeur tout en coloriant les sommets du graphe comme suit :

- blanc pour les sommets non découverts ;
- gris pour un sommet découvert dont le parcours en profondeur est en cours
- et noir pour un sommet dont le parcours en profondeur est terminé.

Question 16. Ecrire une fonction `aCircuit` qui étant donné un graphe orienté renvoie un booléen indiquant si le graphe contient un circuit.

Question 17. Adapter la fonction précédente pour qu'elle renvoie une liste contenant les sommets d'un circuit quand il y en a un et `None` sinon. La liste doit contenir les sommets dans l'ordre de gauche à droite.

Dans le cas d'un graphe non orienté, s'ajoute la difficulté que pour l'arête $\{s, t\}$ l'enchaînement $s \to t \to s$ ne doit pas être pris en compte comme étant un cycle. Il faut donc ajouter un paramètre `parent` permettant de savoir si s a permis la découverte de t.

Question 18. Ecrire une fonction `aCycle` qui étant donné un graphe orienté renvoie un booléen indiquant si le graphe contient un circuit.

Question 19. Adapter la fonction précédente pour qu'elle renvoie une liste contenant les sommets d'un circuit quand il y en a un et `None` sinon. La liste doit contenir les sommets dans l'ordre de gauche à droite.