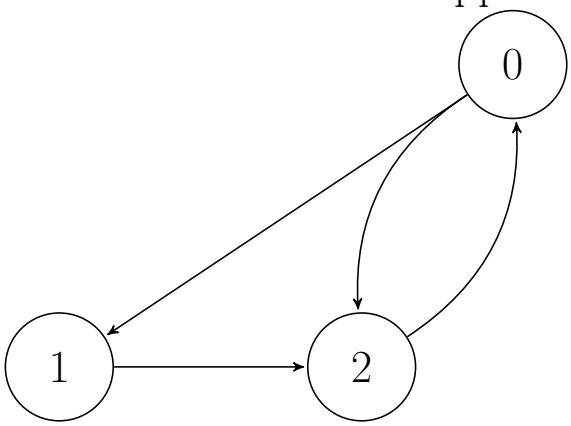
Graphes orientés

Parfois le lien entre deux sommets d'un graphe nécessite une orientation, par exemple lorsque ce lien représente une rue en sens unique.

Voici une carte d'un quartier de Palaiseau où il y a des rues en sens unique :

que l'on peut schématiser par un graphe **orienté**

Les liens orientés sont appelés des **arcs**



On peut parler ici du **chemin** $0 \to 1 \to 2$

1 Vocabulaire

1. Le **degré entrant** d'un sommet s, $d^+(s)$ est le nombre d'arcs arrivant à ce sommet s.

   Par exemple dans le graphe précédent $d^+(2) = 2$

2. Le **degré sortant** d'un sommet s, $d^-(s)$est le nombre d'arcs partant de ce sommet.

   Par exemple dans le graphe précédent $d^-(0) = 2$

3. Un sommet est **équilibré** si son degré sortant est égal à son degré entrant.

   Le sommet 1 est équilibré, les sommets 0 et 2 ne le sont pas.

4. Un **chemin** d'un sommet u vers un sommet v est une succession d'arcs partant de u et arrivant à v.

   Par exemple dans le graphe précédent il existe au moins un **chemin** pour aller de 0 vers 2, par xemple $0 \to 1 \to 2$

5. La **longueur d'un chemin** est le nombre d'arcs de ce chemin.

   Le chemin précédent a pour longueur 2.

6. Un **cycle** est un chemin partant et arrivant au même sommet.

   $0 \to 2 \to 0$ est un cycle dans le graphe précédent.

7. Un **DAG** (pour Directed Acyclic Graph) est un graphe orienté acyclique.

8. Un **cycle eulérien** est un cycle utilisant une fois et une seule chaque arc du graphe.

   Il n'y a pas de cycle eulérien dans le graphe précédent (Théorème d'Euler)

9. Un **chemin eulérien** de u vers v est un chemin de u vers v utilisant une fois et une seule chaque arc du graphe.

   Il y a un chemin eulérien de 0 vers 2 :

   $0 \to 2 \to 0 \to 1 \to 2$

10. Deux sommets u et v sont **fortement connectés** s'il existe un chemin pour aller de u vers v et un autre pour aller de v vers u.

11. Un graphe orienté est **fortement connexe** si tout couple de sommets sont fortement connectés.

    Tout graphe orienté non fortement connexe peut être décomposé en union disjointe de **composantes fortement connexes**.

12. Théorèmes d'Euler

    Théorème 1. *Tout graphe orienté fortement connexe dont chaque sommet est équilibré a un cycle eulérien*

    Théorème 2. *Tout graphe orienté fortement connexe dont chaque sommet est équilibré sauf deux sommets u et v, tels que*

    (a) $d^+(u) - d^-(u) = 1$

    (b) $d^+(v) - d^-(v) = -1$

    a un chemin eulérien de v vers u

Voici une interface pour un graphe orienté

Interface d'un graphe orienté
Opérations : •`creer_graphe: entier` $\rightarrow$ `Graphe` `creer_graphe(nb_sommets)` crée un graphe à `nb_sommets` sans arcs •`ajoute_arc: Couple d'Entiers` $\rightarrow \phi$ `ajoute_arc(i,j)` mémorise l'arc $i \rightarrow j$ •`nb_arcs: Graphe` $\rightarrow$ `Entier` `nb_arcs(G)` retourne le nombre d'arcs du Graphe G •`voisins: Entier` $\rightarrow$ `Liste d'Entiers` `voisins(i)` retourne la liste des voisins de i

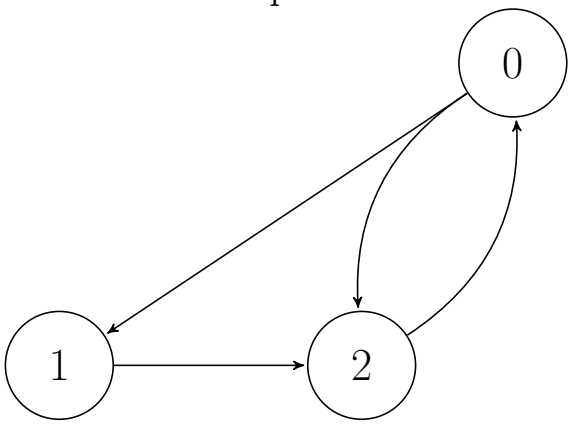
2 Implémentation en Python

Dans la structure de donnée suivante un graphe sera représenté par des **listes de voisinages** plutôt que par **une matrice d'adjacence**

```
class Graphe_O:
    def __init__(self,nb_sommets):
        self.nb_sommets = nb_sommets
        self.nb_arcs = 0
        self.arcs =  \
        [[]for i in range(nb_sommets)]

    def ajoute_arcs(self,i,j):
        self.arcs[i] .append( j)
        self.nb_arcs += 1

    def voisins(self,i):
        return self.arcs[i]

    def __str__(self):
```

```
        ch = str(self.nb_sommets)+" sommets "+str(s
        for i in range(self.nb_sommets):
            ch = ch + str(i) + " --> "
            for sommet in self.arcs[i]:
                ch = ch + str(sommet) + " "
            ch = ch +"\n"
        return ch
```

Voici un exemple d'utilisation pour le graphe orienté suivant




```
g = Graphe_O(3)
g.ajoute_arc(0,1)
g.ajoute_arc(0,2)
g.ajoute_arc(2,0)
g.ajoute_arc(1,2)
print(g)
```

On obtient dans la console

```
3 sommets 4 arcs
0 --> 1 2
1 --> 2
2 --> 0
```

3 Problèmes

Voici quelques problèmes qui peuvent être posés concernant un graphe orienté :

1. Existe-t-il un chemin reliant deux sommets i et j donnés ?
2. Quels sont tous les sommets accessibles à partir de i ?
3. Existe-t-il un cycle dans le graphe orienté ?
4. Quels sont les longueurs des chemins entre i et j donnés ?

3.1 Parcours en profondeur d'un graphe orienté

L'algorithme récursif est le même que celui d'un graphe non orienté

Algorithme 1 : Parcours en profondeur d'un graphe G au départ d'une source s

```
   parcours_profondeur (G,s)
   début
   |  Données : Une source s
   |  Résultat : Rien
1  |  marquer(s)
2  |  afficher(s)
3  |  pour chaque voisin de s non marqué faire
4  |  |  parcours_profondeur (G,voisin)
5  |  fin
   fin
```

Voici une implémentation en Python (PP pour parcours en profondeur) qui est valable à la fois pour les graphes orientés et non orientés.

```
class PP:
    """
    valable pour G orienté ou non orienté
```

```
    """
    def __init__(self,graphe,source):
        self.visites = [False]*graphe.nb_sommets
        self.ancetres = [0]*graphe.nb_sommets
        self.source = source
        self.pp(graphe,source)

    def pp(self,graphe,v):
        self.visites[v] = True
        for w in graphe.voisins(v):
            if not self.visites[w]:
                self.ancetres[w] = v
                self.pp(graphe,w)

    def pp_iter(self,graphe,v):
        pass

    def a_un_chemin_vers(self,v):
        return self.visites[v]

    def chemin_vers(self,v):
        """
        retourne le chemin s'il existe de s vers v
        forme d'une liste d'entiers
        s'il n'existe pas retourne une liste vide
        """

        if not(self.visites[v]):
            return []
        chemin = [v]
        sommet = v
```

```
        while sommet != self.source:
            sommet = self.ancetres[sommet]
            chemin.insert(0,sommet)
        return chemin
```

Applications

1. **Détection d'un cycle dans un graphe orienté :**

   On veut définir une fonction qui détecte le premier cycle rencontré dans un graphe orienté et retourne Vrai

   On ne peut plus utiliser un tableau pour marquer les sommets visités

   Par contre au lieu de considérer qu'un sommet a deux états, visité ou non visité on associe à un sommet trois états ou couleurs :

   (a) BLANC ou non visité

   (b) GRIS , visité mais l'exploration des voisins est en cours

   (c) NOIR visité et l'exploration des voisins est terminée

   On détecte un cycle quand dans un parcours en profondeur, un sommet w NOIR a un de ses voisins v GRIS (cela signifie que l'on est en train de revenir sur ses pas (le sommet est NOIR) et on a un voisin qui est GRIS d'où on est parti plus tôt autrement dit c'est un cycle)

   On a donc une classe `Cycle_0`

```
class Cycle_0:
    """
    utilise dfs pour détecter la présence d'un c
    dans un graphe (sans boucle, sans arêtes par
    on colorie les sommets d'un graphe
    BLANC = 0 = sommet non visité
```

```
    GRIS = 1 = sommet visité mais dont on n'a pa
                tous les voisins
    NOIR = 2 = sommet déjà visité ainsi que tous

    cycle = sommet noir "voit dans son voisinage
sommet gris
    """
    def __init__(self,graphe):
        self.visites = [False]*graphe.nb_sommets
        self.couleurs = [0]*graphe.nb_sommets
        self . ancetres = [0]* graphe . nb_somme
        self.a_cycle = False
        #on parcourt tous les sommets du graphe
        #en partant de 0
        for sommet in range(graphe.nb_sommets) :
            #si le sommet s n'est pas visité
            #on lance un dfs en partant de s
            if  not self.visites[sommet]:
                self.pp(graphe,sommet)
                # à la sortie si cycle = True a
                if self.a_cycle:
                    break


    def pp(self,graphe,source):
        #marquer source
        self.visites[source] = True
        #mettre à jour la couleur (source passe
gris = 1)
        self.couleurs[source] = 1
        #pour chaque voisin v de source non vis
```

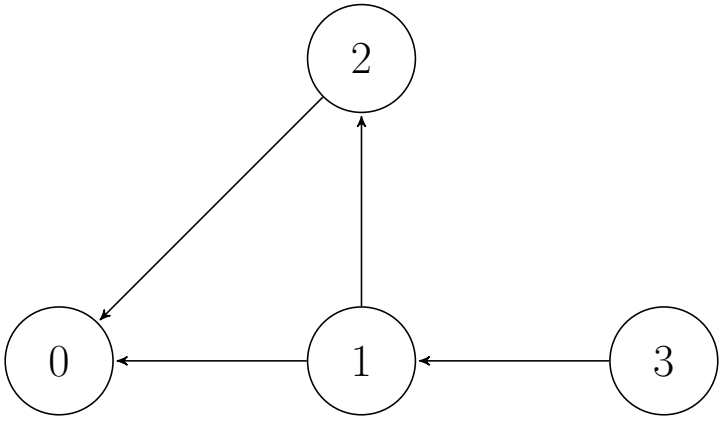
```
            for voisin in graphe.voisins(source):
                if not self.visites[voisin]:
                    #memoriser l'ancetre de voisin
                    self.ancetres[voisin] = source
                    self.pp(graphe, voisin)
            #mettre à jour la couleur (source passe
    noir = 2)
            self.couleurs[source] = 2
            #si un de mes voisins v est gris alors
            for voisin in graphe.voisins(source):
                if  self.couleurs[voisin] == 1:
                    self.a_cycle = True
                    #attribut permettant de constru
                    self._cycle = [source,voisin]
                    break

        def a_un_cycle(self):
            return self.a_cycle

        def cycle(self):
            if self.a_un_cycle():
                #reconstruire le cycle en partant de
                while self._cycle[0] != self._cycle
                    self._cycle.insert(0,
                    self.ancetres[self._cycle[0]])
                return self._cycle
            else:
                return []
```

2. **Tri topologique**

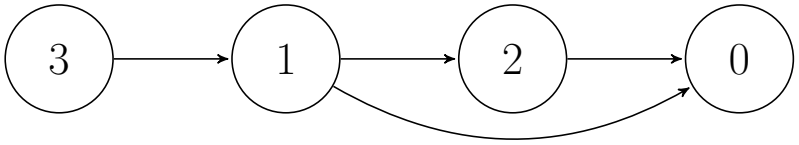
   Dans un graphe orienté acyclique (DAG) on peut ordon-

ner les arcs de telle sorte que les sommets du graphe sont placés de la gauche vers la droite sur un axe horizontal et si il y a un arc u -> v alors u est à la gauche de v.

On parle alors du tri topologique d'un graphe.



Après le tri topologique du DAG



Théorème 3. *Un tri topologique d'un graphe orienté n'est possible si et seulement si ce graphe est acyclique*

Reprenons l'outil des couleurs pour la détection de cycle dans un graphe.

(a) BLANC ou non visité

(b) GRIS , visité mais l'exploration des voisins est en cours

(c) NOIR visité et l'exploration des voisins est terminée

Si on fait un parcours en profondeur **globalement** sur un DAG en visitant tous les sommets du graphe, tous les sommets vont devenir noirs dans un certain ordre $s_1, s_2, ..., s_n$, alors le tri topologique du DAG est simplement :

$s_n, s_{n-1}, ..., s_2, s_1$, car

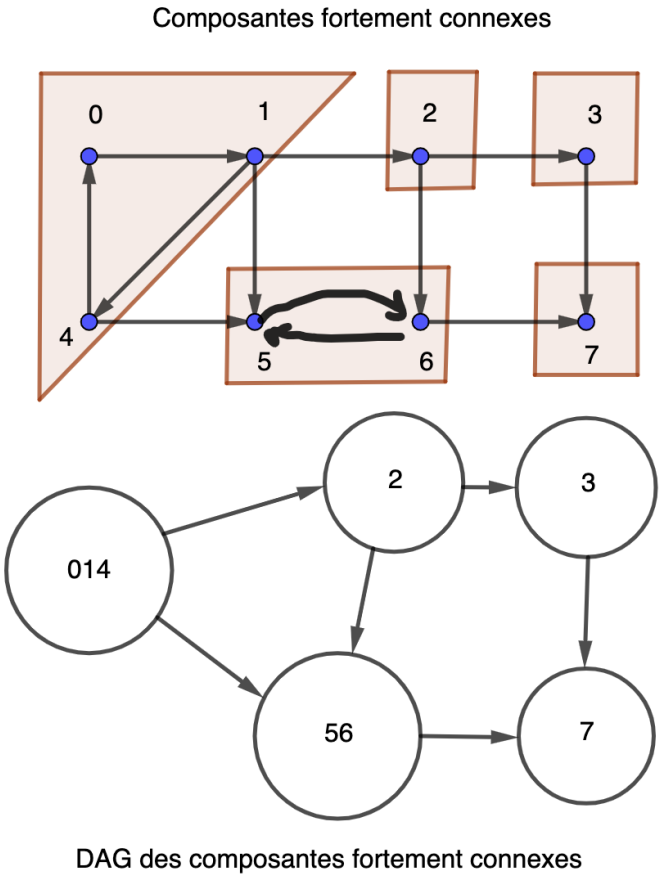
Théorème 4. *Si s1 et s2 sont deux sommets noirs où s2 est devenu noir après s1 au cours du parcours en profondeur global sur G, alors s'il existe un chemin entre s1 et s2 il est dans le sens de s2 vers s1*

D'où l'algorithme du tri topologique d'un DAG

```
tri_topologique(G)

faire un dfs global sur G
chaque fois qu'un sommet prend la couleur NOIR
l'insérer au début d'une liste
renvoyer la liste
```

Dans la classe `Cycle_O` ajouter les méthodes `est_un_DAG()` est `tri_top()`.

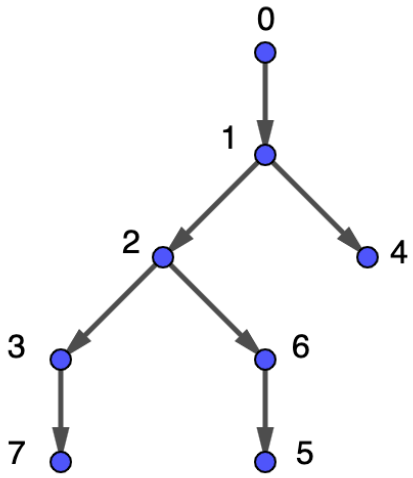
3. Composantes fortement connexes



On suppose que la liste des voisins du graphe ci-dessus est :

```
arcs = [[1],[2,5,4],[3,6],[7],[0,5],[6],[5],[]]
```

On fait un parcours en profondeur global sur ce graphe, voici l'arbre des différents appels récursifs

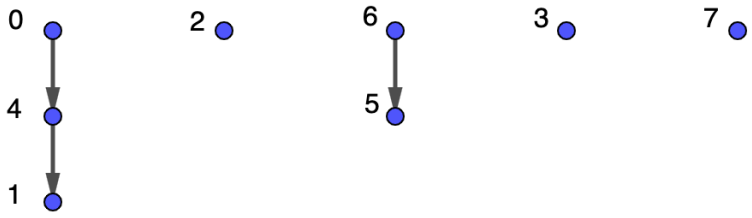


Comme pour le tri topologique chaque fois qu'un sommet prend la couleur NOIR on le met au début d'une liste, ce qui donne :

0,1,4,2,6,5,3,7

On va se servir de cet ordre sur les sommets pour lancer un deuxième parcours en profondeur cette fois ci sur le graphe inverse de G.

Voici la **forêt des appels récursifs** obtenu :



On obtient ainsi les composantes fortement connexes, d'où l'algorithme pour obtenir les composantes fortement connexes d'un graphe orienté

```
CFC(G)

*faire un dfs global sur G
chaque fois qu'un sommet prend la couleur NOIR
l'insérer au début d'une liste L
*Construire le graphe inverse de G
*Faire un dfs global sur le graphe inverse de G
en suivant l'ordre de L
```

Voici les propriétés qui justifient cet algorithme :

Théorème 5. *Les composantes fortement connexes d'un graphe orienté et son inverse sont les mêmes*

Théorème 6. *Soit C1 et C2 deux composantes fortement connexes, s'il existe un chemin partant d'un sommet s1 de C1 vers un sommet s2 de C2 alors il ne peut pas avoir de chemin allant de s2 vers s1*

Théorème 7. *Soit C1 et C2 deux composantes fortement connexes, s'il existe un arc partant d'un sommet s1 de C1 vers un sommet s2 de C2 alors le dernier sommet de C1 ayant pris la couleur NOIR a pris cette couleur après le dernier sommet de C2 ayant pris la couleur NOIR*

3.2 Parcours en largeur dans un graphe orienté

Algorithme 2 : Parcours en largeur d'un graphe à partir d'une source

```
parcours_largeur (G,s)
début
    Données : Un graphe G, une source s un entier
    Résultat : Rien
1   f = file_vide()
2   enfiler(f,s)
3   marquer(s)
4   tant que non (est-vide(f)) faire
5       x = defiler(f)
6       afficher(x)
7       pour chaque voisin de x non marqué faire
8           enfiler(f,voisin)
9           marquer(voisin)
10      fin
11  fin
fin
```

Voici une implémentation en Python (PL pour parcours en largeur) valable à la fois pour les graphes orientés et non orientés

```
class PL:
    """
    pour graphe orienté et non orienté
    """
    def __init__(self,graphe,source):
        self.visites = [False]*graphe.nb_sommets
        self.peres = [source]*graphe.nb_sommets
        self.source = source
        self.pl(graphe,source)
```

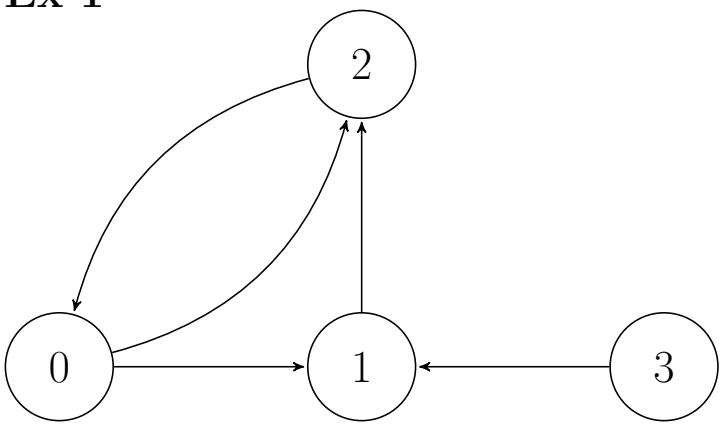
```
def pl(self,graphe,v):
    self.visites[v] = True
    file = [v]
    while len(file) != 0:
        x = file.pop()
        for w in graphe.voisins(x):
            if not self.visites[w]:
                self.visites[w] = True
                self.peres[w] = x
                file.insert(0,w)

def a_un_chemin_vers(self,v):
    return self.visites[v]

def chemin_vers(self,v):
    """
    retourne le chemin s'il existe de s vers v
    forme d'une liste d'entiers
    s'il n'existe pas retourne une liste vide
    """
    if not(self.visites[v]):
        return []
    chemin = [v]
    sommet = v
    while sommet != self.source:
        sommet = self.peres[sommet]
        chemin.insert(0,sommet)
    return chemin

def distance(self,v):
```

```
        """
        Le parcours en largeur détermine les chemin
        entre la source et un sommet visité
        """
        ch = self.chemin_vers(v)
        if len(ch) == 0:
            return inf
        return len(ch) - 1
```

Le parcours en largeur donne le chemin le plus court de la source à un sommet donné (si ce chemin existe)

4 Exercices

Ex 1



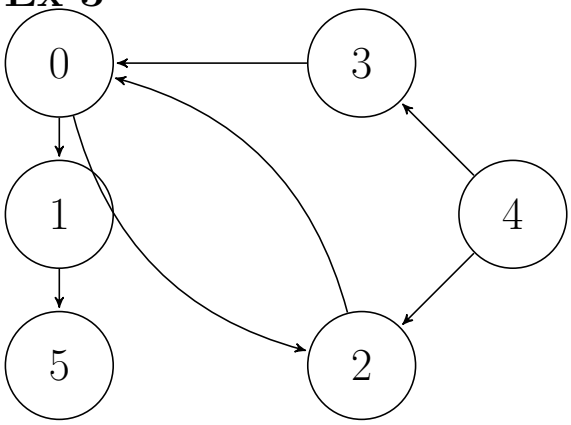
1. Quels sont les degrés entrants et sortants de chaque sommet ?
2. Quels sont les sommets équilibrés ?
3. Est ce que les sommets 0 et 2 sont fortement connectés ?
4. Est ce que les sommets 1 et 2 sont fortement connectés ?
5. Est ce que les sommets 1 et 3 sont fortement connectés ?
6. Est ce que le graphe est fortement connexe ?
7. Existe-t-il un cycle eulérien ? un chemin eulérien ?

Ex 2

Si G est un graphe orienté, le graphe inverse de G noté $\overline{G}$ est le graphe ayant les même sommets que G mais dont les arcs sont orientés dans le sens inverse que ceux de G

Implémenter une méthode `inverse()` dans la classe `Graphe_O` qui crée le graphe inverse de self

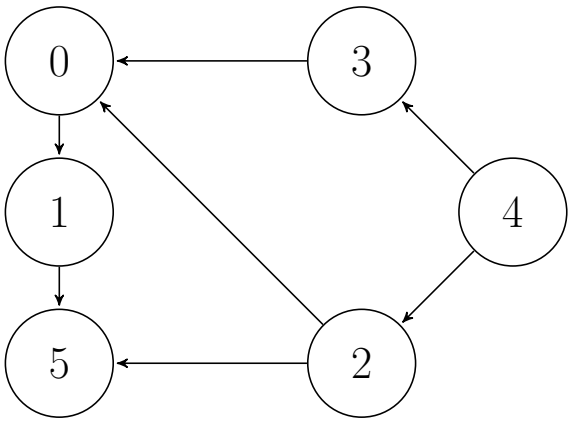
Ex 3



1. Donner la succession des sommets visités par un parcours en profondeur en partant de la source, où :
   (a) source = 4
   (b) source = 2
2. Donner la succession des sommets visités par un parcours en largeur en partant de la source, où :
   (a) source = 4
   (b) source = 2

Ex 4

Faire le tri topologique du graphe orienté suivant



Ex 5

Implémenter une classe `Ordre_PP` dans laquelle on obtient les trois ordres de visite des sommets lors d'un parcours en profondeur du graphe orienté

Ex 6

Un graphe orienté g est représenté par une liste d'adjacence. (celui de la classe `Graphe_O` vue en cours)

1. Ajouter une méthode `deg_ent` à la classe `Graphe_O` qui renvoie un tableau `tab` où `tab[i]` est le degré entrant du sommet `i`
2. Justifier par un argument en Français que l'algorithme suivant pour détecter un cycle dans un graphe orienté est correct : "Tant que c'est possible supprimer du graphe un sommet sans prédécesseur (degré entrant nul). Si on réussit à supprimer tous les sommets, le graphe est sans cycle"
3. Comment implémenter cet algorithme dans la classe `Graphe_O` sous la forme d'une méthode `a_un_cycle` sans supprimer les sommets de `self` ?(Indication : partir du tableau des degrés entrants, s'il n'y a pas de degré entrant nul alors il y a un cycle sinon partir d'un sommet s ayant un degré entrant nul dans le tableau mettre -1 à la place de 0 pour dire qu'il est supprimé puis enlever 1 au degré entrant de chaque sommet v voisin de s (car il faut supprimer les arcs

associés au sommet s qu'on supprime) etc... à finir)

Ex 7

Le tri topologique d'un graphe orienté acyclique consiste à ordonner linéairement tous ses sommets de sorte que si G contient un arc $u \to v$, u apparaisse avant v dans le tri.

On a vu en cours comment avec une file et des parcours en profondeurs on fait le tri topologique du graphe orienté acyclique (DAG).

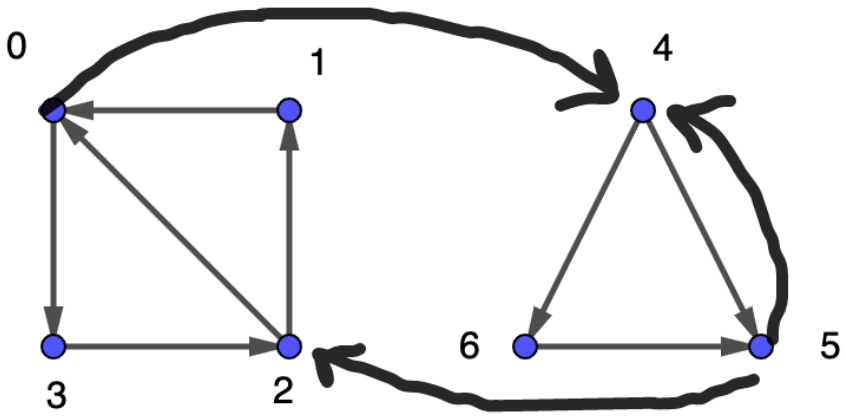
Voici un autre algorithme dans le prolongement de celui de l'exercice précédent (on améliore l'algorithme précédent)

"Lorsqu'on supprime un sommet s sans prédecesseur l'insérer en fin d'une liste Python. Si on a réussi à supprimer tous les sommets le graphe est un DAG et dans la liste Python à la fin on a le tri topologique du graphe, sinon il y a un cycle dans le graphe"

Ecrire une méthode `tri_top` de la classe `Graphe_O` qui renvoie soit la liste des sommets triés selon le tri topologique (dans ce cas il n'y a pas de cycle) soit la liste vide dans ce cas il y a un cycle.

Ex 8

On va travailler sur le graphe orienté G suivant :



1. Justifier que G est fortement connexe.

2. Justifier que tous les sommets de G sont équilibrés.

Donc, d'après le théorème d'Euler G admet un cycle eulérien.

On va donner ci-après une preuve **constructive** de ce théorème sous forme d'un algorithme.

Voici l'idée de l'algorithme (aléatoire).

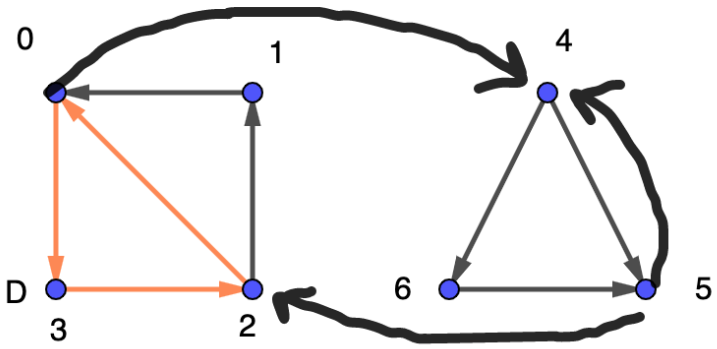
On part de n'importe quel sommet, par exemple le sommet 3. (la lettre D sur le schéma ci-dessous)

Puis on avance de manière "aléatoire" jusqu'à ce qu'on obtienne un cycle. Si ce cycle est eulérien on arrête l'algorithme sinon on recommence voir plus bas :

PREMIERE ETAPE

Au début on n'a pas le choix donc on va de 3 vers 2, mais quand on arrive en 2 on peut aller soit vers 1 ou 0, prenons par exemple la sortie vers 0, arrivé en 0 on a le choix on peut aller soit vers 4 soit vers 3 et on a obtenu un cycle qui n'est pas eulérien 3-2-0-3. (voir dessin en orange).

Arrivé en 3 on est coincé on ne peut plus utiliser les arcs restants en noir. Si on avait commencé en 2 ou en 0 et en faisant le même cycle on aurait pu continuer d'utiliser les arcs restants.



On peut s'en sortir en faisant une **permutation circulaire** sur le cycle 3-2-0-3.

A partir de 3-2-0-3 on a trois permutations circulaires possibles (on part d'un autre sommet du cycle et on fait le même

cycle) possibles :

1. 2-0-3-2
2. 0-3-2-0
3. 3-2-0-3

On observe que les cycles 2-0-3-2 et 0-3-2-0 nous permettent de continuer le cheminement.

Justifier la propriété suivante :

Propriété

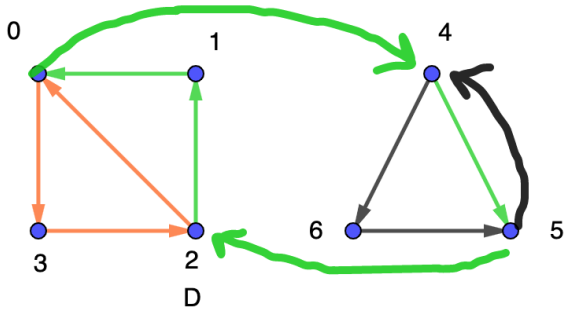
Pourquoi est on sûr qu'il existe au moins une permutation circulaire nous permettant de nous débloquer le chemin (Rappel : G est fortement connexe et tous les sommets sont équilibrés) ?

DEUXIEME ETAPE

On part de 2 et on refait le cycle donc 2-0-3-2.

Cette fois-ci on n'est pas coincé en 2 donc on continue notre promenade aléatoire et supposons qu'on fasse à partir de 2 (en vert) 2-1-0-4-5-2.

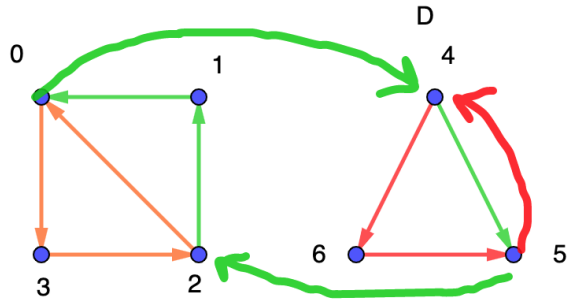
On a donc un cycle 2-0-3-2-1-0-4-5-2 qui n'est pas eulérien, il reste encore des arcs et on est coincé en 2.



TROISIEME ETAPE

La propriété ci-dessus nous assure encore qu'il existe une permutation circulaire nous permettant de débloquer le chemin, tant qu'on n'a pas utilisé tous les arcs autrement dit tant qu'on n'a pas obtenu ce que l'on cherche un cycle eulérien.

En effet 4-5-2-0-3-2-1-0-4 nous permet de continuer le chemin en faisant 4-6-5-4 (en rouge) et on a obtenu un cycle eulérien possible 4-5-2-0-3-2-1-0-4-6-5-4



Questions

1. Poursuivre le cheminement qui démarre en 4 et tel qu'on obtienne le premier cycle 4-5-4
2. Faire un ou deux autres cheminements personnels.
3. Pourquoi n'est il pas nécessaire de vérifier que le graphe est fortement connexe et que tous les sommets sont équilibrés avant de lancer l'algorithme ?

   Autrement dit pourquoi pas lancer l'algorithme sur n'importe quel graphe orienté quitte ensuite à interpréter correctement le résultat de l'algorithme ?
4. En pseudo-code on a une fonction `perm(C,s)` qui réécrit un cycle C en partant d'un sommet s de C.

   Par exemple `perm([4,5,6,4],5)` renvoie [5,6,4,5].

   L'union de deux cycles notée U vérifie :

   [2,0,3,2] U [2,1,0,4,5,2] donne [2,0,3,2,1,0,4,5,2]

   Voici une façon de mettre en pseudo code l'algorithme de création d'un cycle eulérien.

   ```
   cycle_eulerien(G)
   Faire un cycle C en voyageant
   ```

```
au hasard dans G
Tant qu'il existe des arcs non visités de G fair
    Choisir un sommet s de C ayant
    des arcs de sortie non visités
    En partant de s en voyageant au hasard
    faire un cycle C'
    C <- perm(C ,s) U C'
renvoyer C si C eulérien liste vide sinon
```

Ecrire une méthode `cycle_eulerien()` dans la classe `Graphe_O` qui renvoie un cycle eulérien sous la forme d'une liste de sommets parcourus s'il en existe un, et qui renvoie une liste vide sinon.

(Dans un premier temps ne pas se poser le problème du coût mais il existe une implémentation qui a un coût linéaire en le nombre d'arcs du graphe orienté.)