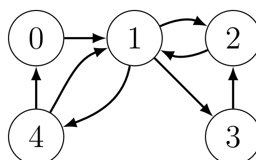
Graphes

Exercice 1

1. Dessiner tous les graphes non orientés (simples) ayant exactement 3 sommets.
2. Dessiner au moins neuf graphes orientés ayant exactement 3 sommets.

Exercice 2
On considère le graphe suivant :

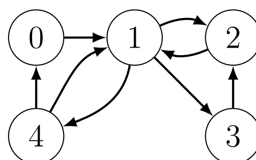


1. Donner le degré de chacun des sommets.
2. Donner 3 cycles différents de ce graphe.

Exercice 3
Tracer les graphes associés aux matrices d'adjacence données, les sommets étant numérotés à partir de 0.

$$M_1 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{matrix} 0 & 1 & 2 & 3 \end{matrix} \\ \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \end{array} \qquad M_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix} \qquad M_3 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

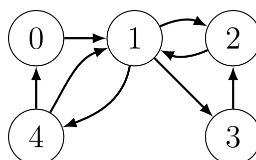
Exercice 4
Écrire la matrice d'adjacence du graphe ci-dessous :



Exercice 5
Donner le graphe associé à liste d'adjacence suivante :

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

Exercice 6
Écrire la liste d'adjacence du graphe ci-dessous :



Exercice 7 (classe Graphe pour les graphes orientés définis par matrice d'adjacence)
Définir, pour les graphes orientés, dont on considérera les sommets nommés par les indices de `0` à `n-1` (où `n` est l'ordre du graphe), une classe `Graphe` contenant :

- La fonction d'initialisation `__init__(self,n)` qui crée le graphe avec un attribut `adj` qui est la matrice d'adjacence (nulle au départ) et un attribut `n`, ordre du graphe donné en argument ;
- La méthode `ajouter_arc(self,s1,s2)` qui permet d'ajouter un arc entre le sommet `s1` et le sommet `s2` ;
- La méthode `arc(self,s1,s2)` qui retourne `True` s'il y a un arc depuis `s1` vers `s2`, `False` sinon ;
- La méthode `ordre(self)` qui retourne l'ordre du graphe ;
- La méthode `sommets(self)` qui retourne la liste des sommets du graphe
- La méthode `voisins(self,s)` qui retourne la liste des sommets adjacents au sommet `s`
- La méthode `degre(self,s)` qui retourne le degré du sommet `s`.
- La méthode `liste_adjacence(self)` qui retourne la liste d'adjacence du graphe (sous forme de dictionnaire) ;
- La méthode `matrice_adjacence(self)` qui retourne la matrice d'adjacence du graphe (sous forme de liste de listes) ;

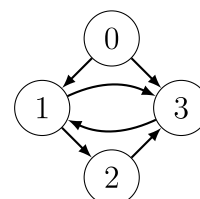
Effectuer quelques tests pour vérifier le fonctionnement de ces diverses méthodes.
Facultatif : Refaire l'exercice en permettant d'avoir des noms quelconques pour les sommets (de type quelconque, que ce soit `str` ou `int` par exemple). On pourra gérer cela à l'aide d'un dictionnaire et utiliser celui-ci pour traduire entre nom et indice, la liste (supposée ordonnée) des noms de sommets (supposés tous uniques) étant donnée en paramètre lors de l'initialisation au lieu de `n`.

Exercice 8 (classe Graphe pour les graphes non orientés définis par liste d'adjacence)
Définir (puis tester là encore), pour les graphes non orientés, dont on considérera les sommets nommés de manière quelconque, une classe `Graphe` contenant :

- La fonction d'initialisation `__init__(self)` qui crée le graphe avec seulement un attribut `adj` qui est la liste d'adjacence, de type dictionnaire (vide au départ) ;
- La méthode `ajouter_sommet(self,s)` qui permet d'ajouter le sommet `s`, si celui-ci n'existe pas déjà, au graphe ;
- La méthode `ajouter_arete(self,s1,s2)` qui permet d'ajouter une arête entre le sommet `s1` et le sommet `s2` ;
- La méthode `arete(self,s1,s2)` qui retourne `True` s'il y a une arête entre `s1` et `s2`, `False` sinon ;
- Les méthodes `ordre(self)`, `sommets(self)`, `voisins(self,s)`, `degre(self,s)`, `liste_adjacence(self)` et `matrice_adjacence(self)` définies dans l'exercice précédent.

Exercice 9 (Facultatif)
Ajouter, dans les classes `Graphe` définies précédemment, les méthodes suivantes :

1. `nb_arcs(self)` (resp. `nb_aretes(self)`) qui retourne le nombre d'arcs (resp. d'arêtes) du graphe ;
2. `supprimer_arc(self,s1,s2)` (resp. `supprimer_arete(self,s1,s2)`) qui supprime l'arc (resp. l'arête) entre `s1` et `s2` s'il existe, qui ne fait rien sinon.
3. `__str__(self)` qui retourne une chaîne de caractère décrivant le graphe. Par exemple, si le graphe `G` est celui donné ci-contre, l'instruction `print(G)` doit alors afficher :

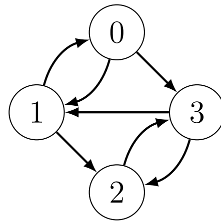



```
0 [1, 3]
1 [2, 3]
2 [3]
3 [1]
```

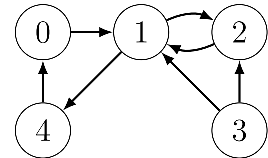
Ou bien :

```
0 -> 1 3
1 -> 2 3
2 -> 3
3 -> 1
```

Exercice 10
Pour chacun des sommets du graphe ci-dessous, dérouler à la main le parcours en profondeur. Donner à chaque fois la valeur finale de la liste `vus`.



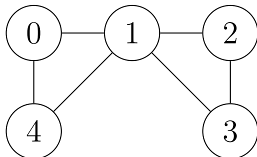
Exercice 11
On considère la fonction `mystere` ci-dessous et le graphe `g` ci-contre.



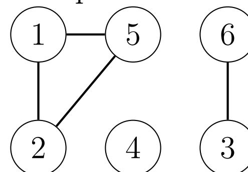

```
def mystere(g, u, v):
    vus = parcours_profondeur(g, u)
    return v in vus
```

1. Quelle est le résultat de `mystere(g, 0, 4)` ?
2. Même question pour `mystere(g, 0, 3)`.
3. Décrire en une phrase le résultat de `mystere(g, u, v)` pour `u` et `v` deux sommets de `g`.

Exercice 12
On peut se servir d'un parcours en profondeur pour déterminer si un graphe *non orienté* est *connexe*, c'est à dire si tous ses sommets sont reliés entre eux par des chemins.



Graphe connexe



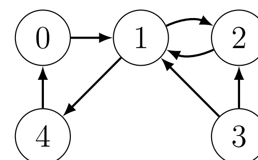
Graphe non connexe

Pour cela, il suffit de faire un parcours en profondeur et de vérifier que tous les sommets ont bien été visités par ce parcours.
En utilisant de la méthode `sommets` de la classe `Graphe` et de la fonction `parcours_profondeur`, écrire écrire une fonction `est_connexe` qui réalise cet algorithme.

Exercice 13
Dans cet exercice, on se propose d'utiliser un parcours en profondeur pour *construire* un chemin entre deux sommets lorsque c'est possible.
On le fait avec deux fonctions :

1. La fonction `parcours_chemin` est très semblable à la fonction `parcours_profondeur`. Elle prend un paramètre supplémentaire `org` (pour origine) qui est le sommet qui permis d'atteindre `s` et l'argument `vus` n'est plus une liste mais un *dictionnaire* qui associe à chaque sommet visité le sommet qui a permis de l'atteindre.

   Pour le graphe ci-contre, un tel parcours d'origine le sommet `1` donne le dictionnaire `vus` suivant :


   ```
   {1 : None, 2 : 1, 4 : 1, 0 : 4}
   ```

   (a) Pour le graphe ci-dessus, déterminer le dictionnaire `vus` obtenu par `parcours_chemin` avec pour origine le sommet `2`.

   (b) Compléter le programme ci-dessous à fin qu'il réalise ce parcours.

```
def parcours_chemin(g, vus, org, s):
    '''parcours depuis le sommet s, en venant de org'''
    if s not in vus:
        vus[s] = ........
        for v in g.voisins(s):
            ........
```

2. La deuxième fonction `chemin(g, u, v)` permet de construire le chemin entre `u` et `v` s'il existe. Pour cela on « remonte » le dictionnaire `vus` obtenu avec `parcours_chemin`.

   Par exemple, si l'on veut trouver un chemin du sommet `1` au sommet `0` et que notre dictionnaire obtenu à partir d'un parcours d'origine le sommet `1` est

   `{1 : None, 2 : 1, 4 : 1, 0 : 4}`, « remonter » le dictionnaire donnera `0 4 1 None`, ce qui donne le chemin `1 -> 4 -> 0`.

   Compléter la fonction ci-dessous pour qu'elle réalise cet algorithme.

```
def chemin(g, u, v):
    '''un chemin de u a v, le cas échéant, None sinon'''
    vus = {}
    parcours_chemin(g, vus, None, u)
    # s'il n'existe pas de chemin
    if ........
        return None
    # sinon on construit le chemin
    ch = []
    s = v
    while ........
        ch.append(s)
        s = ........
    ch.reverse()
    return ch
```

Exercice 14

Un parcours en profondeur permet de déterminer s'il existe un cycle dans un graphe donné. Si, lors de ce parcours, le sommet qui vient d'être découvert a un voisin en cours de traitement (gris), c'est qu'un cycle existe. Le principe plus en détail est le suivant :

Lorsque l'on visite un sommet `s`,

- S'il est gris, c'est qu'on vient de découvrir un cycle :
- s'il est noir, on ne fait rien ;
- sinon c'est qu'il est blanc, et on procède ainsi :
  1. on colore le sommet `s` en gris ;
  2. on visite tous ses voisins récursivement ;
  3. enfin, on colore le sommet `s` en noir.

Comme on peut le remarquer, les voisins du sommet `s` sont examinés après le moment où le sommet `s` est coloré en gris, et avant le moment où il est colorié en noir. Ainsi, s'il existe un cycle nous ramenant à `s`, on le trouvera comme étant gris et le cycle sera signalé.

Compléter le programme suivant de sorte qu'il renvoie un booléen indiquant s'il y a un cycle dans le graphe *orienté* `g` avec le principe que l'on vient d'énoncer.

```python
def parcours_cycle(g, couleur, s):
    '''parcours en profondeur depuis le sommet s'''
    if couleur[s] == 'gris':
        return ........
    if couleur[s] == 'noir':
        return ........
    couleur[s] = 'gris'
    for v in g.voisins(s):
        if ........
            return True
    couleur[s] = 'noir'
    return False

def existe_cycle(g):
    '''détermine la présence d'un cycle dans le graphe g'''
    couleur = {}
    for s in g.sommets():
        couleur[s] = 'blanc'
    for s in g.sommets():
        if ........
            return True
    return False
```

Exercice 15
Reprendre l'exercice 10 mais avec le parcours en largeur.

Exercice 16
Un arbre binaire peut être vu comme un graphe non orienté. Le parcours en profondeur correspond alors au parcours préfixe.
Écrire une fonction `largeur(arb)` qui prend un arbre binaire `arb` en argument et affiche les valeurs de ses nœuds dans un ordre donné par un parcours en largeur.
Pour cela adapter le programme de parcours en largeur du cours.

Exercice 17
Cet exercice reprend l'exercice 13 mais on se propose cette fois d'utiliser un parcours en largeur pour *construire* un chemin entre deux sommets lorsque c'est possible.
On utilise encore une fois deux fonctions :

1. La fonction `parcours_largeur_ch` qui est très semblable à la fonction `parcours_largeur`, l'argument `vus` n'est plus une liste mais un *dictionnaire* qui associe à chaque sommet visité le sommet qui a permis de l'atteindre.
   Compléter le programme ci-dessous à fin qu'il réalise ce parcours.

   ```python
   def parcours_largeur_ch(g, s):
       vus = {}
       vus[s] = None
       ...
   ```

2. La deuxième fonction `chemin(g, u, v)` permet de construire le chemin entre `u` et `v` s'il existe. Pour cela on « remonte » le dictionnaire `vus` obtenu avec `parcours_largeur_ch`.
   Compléter la fonction ci-dessous pour qu'elle réalise cet algorithme.

   ```python
   def chemin(g, u, v):
       '''un chemin de u a v, le cas échéant, None sinon'''
       ...
   ```

Exercice 18

Le parcours en largeur permet de déterminer la *distance* entre deux sommets `s1` et `s2` d'un graphe, c'est à dire le nombre minimal d'arcs à emprunter pour aller de `s1` à `s2`.

La fonction `parcours_distance` est très semblable à la fonction `parcours_largeur`. La liste `vus` est remplacée par un dictionnaire `dist` des distances du sommet `s` aux autres sommets accessibles du graphe `g`. Ce dictionnaire contient initialement le sommet `s` avec une distance `0` et à chaque fois qu'un sommet `v` est découvert depuis le sommet `u`, la distance de `s` à `v` est égale à la distance de `s` à `u` plus 1 pour l'arc que l'on vient d'emprunter.

Compléter la fonction Python ci-dessous de manière à ce qu'elle réalise ce parcours.

```
def parcours_distance(g, s):
    '''dictionnaire des distances entre s et les autres sommets
    accessibles de g'''
    dist = {}
    ........
    file = File()
    file.ajouter(s)
    while not file.est_vide() :
        u = file.consulter()
        for v in g.voisins(u):
            if ........:
                ........
                file.ajouter(v)
        file.retirer()
    return dist
```

Compléter alors la fonction `distance(g, u, v)` qui donne la distance entre les sommets `u` et `v` si un chemin entre ces deux sommets existe et `None` sinon.

```
def distance(g, u, v):
    '''distance de u a v et None si pas de chemin'''
    dist = parcours_distance(g, u)
    if ........
        return None
    else:
        return ........
```