IX - TP 5 Application des parcours de graphe

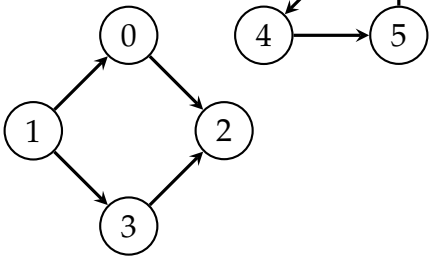
1 Introduction

On s'intéresse dans ce tp à différentes applications directes des parcours (en profondeur, en largeur) de graphes. On utilisera de manière judicieuse les classes et fonctions du fichier `ds.py` fourni, que l'on pourra importer à l'aide de l'instruction :

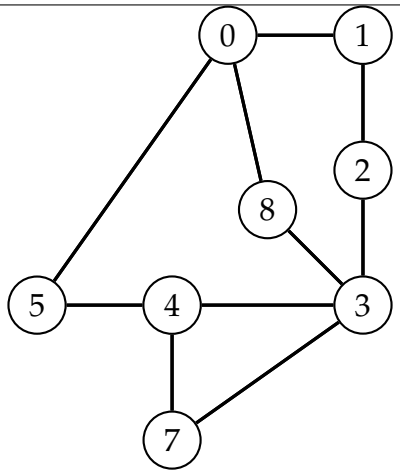
Code

```
from ds import *
```

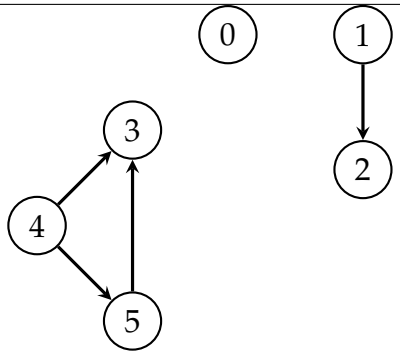
Les variables g1, g2, g3 et g4 sont des objets de type `Graphe` qui représentent les graphes ci-dessous. On les utilisera de manière judicieuse afin de tester les fonctions écrites.



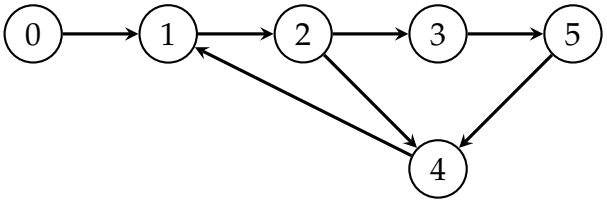
Graphe G_1



Graphe G_2



Graphe G_3



Graphe G_4

2 Existence de chemin entre deux sommets distincts

Écrire une fonction `existe_chemin` qui étant donné un graphe `G` (orienté ou non) et deux sommets `source` et `destination` distincts détermine s'il est possible de trouver un chemin dans le graphe `G` entre le sommet `source` et le sommet `destination`. Vous utiliserez pour cela une des fonctions de parcours de graphe écrite précédemment.

Code

```
def existe_chemin(G, sommet, destination):
    """ Graphe, Sommet, Sommet -> bool
    Renvoie True si et seulement si il existe un chemin entre source et
    ↪ destination dans G. """
    pass
```

Code

```
print(existe_chemin(g1, 1, 2))
print(existe_chemin(g2, 7, 1))
print(existe_chemin(g3, 2, 1))
print(existe_chemin(g4, 0, 4))
print(existe_chemin(g4, 4, 0))
```

Résultat

```
True
True
False
True
False
```

3 Composantes connexes d'un graphe non orienté

Écrire une fonction `composantes_connexes` qui étant donné un graphe `G` *non orienté* en détermine les composantes connexes. Cette fonction renverra un dictionnaire définit de la manière suivante :

- `k` est une clé du dictionnaire si et seulement si `k` est un sommet de `G`
- on associera à la clé `k` un entier représentant le numéro de la composante connexe associée à `k`. L'entier choisit peut-être arbitraire, mais deux clés `k1` et `k2` doivent être associé au même entier si et seulement si les sommets `k1` et `k2` appartiennent à la même composante connexe.

Indication. On pourra utiliser l'algorithme suivant :

- on initialise un dictionnaire `composantes` ainsi qu'un entier `composante_num` qui permettront respectivement de mémoriser le numéro de la composante connexe de chaque sommet et le numéro de la dernière composante connexe découverte.
- pour chaque `sommet` du graphe, on détermine la liste des sommets `atteignables` depuis `sommet` à l'aide d'un parcours du graphe depuis `sommet`.
  - pour chaque sommet `atteignable` présent dans la liste `atteignables`, on attribue à `atteignable` dans le dictionnaire `composantes` le numéro de la composante connexe de `sommet`.
  - on prendra soin de mettre à jour convenablement le numéro de la dernière composante connexe découverte ainsi que de ne pas explorer inutilement le graphe depuis les sommets dont on a déjà trouvé la composante connexe.

Code

```
def composantes_connexes(G):
    """ Graphe -> {Sommet:int}
    Détermine les composantes connexes du graphe G sous la forme d'un
    ↪ dictionnaire """
    pass
```

Code

```
print(composantes_connexes(oublie_orientation(g1)))
print(composantes_connexes(oublie_orientation(g2)))
print(composantes_connexes(oublie_orientation(g3)))
print(composantes_connexes(oublie_orientation(g4)))
```

Résultat

```
{0: 1, 1: 1, 3: 1, 2: 1, 4: 2, 5: 2, 6: 2}
{0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 7: 1, 8: 1}
{0: 1, 1: 2, 2: 2, 3: 3, 4: 3, 5: 3}
{0: 1, 1: 1, 2: 1, 3: 1, 5: 1, 4: 1}
```

4 Repérer la présence d'un cycle

On cherche dans cette question à détecter si un cycle est présent dans un graphe orienté `G`. On rappelle qu'un cycle (ou circuit) dans un graphe $G = (S, A)$ est une suite d'arcs consécutifs $a_0, \ldots, a_{n-1}$ de telle sorte que le sommet source de l'arc a_0 soit le sommet destination de l'arc a_{n-1} ($n \geq 1$ est la longueur du cycle).

L'algorithme de détection de cycle dans un graphe se base sur l'algorithme de parcours en profondeur récursif, que l'on rappelle ci-dessous.

Code

```
def parcours_profondeur(G, depart, vus):
    """ Graphe, Sommet, [Sommet] -> [Sommet]
    Parcourt le graphe G depuis depart en profondeur (sachant que
    la liste des sommets présents dans vus ont déjà été visités).
    On suppose que depart n'est pas dans vus. """
    vus.append(depart)
    # appels récursif sur les voisins non visités
    for destination in G.voisins(depart):
        if not destination in vus:
            parcours_profondeur(G, destination, vus)
    return vus
```

Plutôt que d'ajouter les sommets visités à l'ensemble `vus` on va **marquer** les sommets avec un code couleur :

- les sommets en `BLANC` sont les sommets non atteints par le parcours en profondeur ;
- les sommets en `GRIS` sont les sommets atteints par le parcours en profondeur mais dont l'exploration récursive des voisins n'est pas encore terminée ;
- les sommets en `NOIR` sont les sommets atteints par le parcours en profondeur et dont l'exploration récursive des voisins est terminée.

4.1 Détection d'un cycle depuis un sommet donné

On remplace l'ensemble `vus` de l'algorithme récursif du parcours en profondeur par un dictionnaire `couleurs` définit de la manière suivante :

- les clés `k` sont les sommets du graphe ;
- la valeur associée à la clé `k` dans le dictionnaire `couleurs` est un entier représentant la couleur actuelle du sommet `k` dans le parcours en profondeur du graphe. On prendra comme convention que l'entier `0` représente la couleur blanche, l'entier `1` représente la couleur grise, et l'entier `2` la couleur noire.

Lors de l'appel récursif `parcours_cycle(G, depart, couleurs)` le sommet `depart` (dont on suppose qu'il n'a pas encore été exploré, il est donc en `BLANC`) :

- on commence par marquer en `GRIS` le sommet départ ;
- pour chaque `destination` voisine de `depart` :
  - si la `destination` n'a pas encore été explorée, alors on l'explore récursivement et on renvoie `True` dans le cas où cette exploration détecte un cycle, `False` sinon.
  - sinon, si la `destination` est déjà en cours d'exploration, alors cela veut dire l'on vient de détecter un cycle. On renvoie donc `True`.
- on marque le sommet en `NOIR` pour indiquer qu'il a été complètement exploré et que l'on n'a pas trouvé de cycle. On renvoie donc `False`.

Code

```
BLANC, GRIS, NOIR = 0, 1, 2

def parcours_cycle(G, depart, couleurs):
    """ Graphe, Sommet, {Sommet:Couleur} -> bool
    Renvoie True si et seulement si  il existe un cycle dans le graphe G
 ↪  dans la composante connexe de depart.
    On suppose que depart est en BLANC. """
    couleurs[...] = GRIS
    for destination in G.voisins(depart):
        if ...:
            return ...
        elif ...
            return ...
    couleurs[...] = ...
    return ...
```

Code

```
for g in [g1, g2, g3, g4]:
    print(parcours_cycle(g, 0, {s:BLANC for s in g.sommets()}))
```

4.2 Détection d'un cycle dans un graphe

En déduire une fonction `contient_cycle` qui étant donné un graphe orienté `G` détermine si celui-ci contient un cycle. On fera pour cela appel à la fonction précédente, en initialisant un dictionnaire `couleurs` dans lequel tous les sommets du graphe `G` sont coloriés en blanc. On parcourra le graphe à l'aide de la fonction `parcours_cycle` depuis les sommets du graphe, et on renverra `True` si on détecte un cycle depuis un sommet de cette façon. On prendra garde à ne pas explorer de manière inutile le graphe à partir de sommets déjà parcourus en utilisant astucieusement les `couleurs` des sommets.

Code

```
def contient_cycle(G):
    """ Graphe -> bool
    Renvoie True ssi le graphe orienté G contient un cycle """
    pass
```

Code

```
for g in [g1, g3, g4]:
    print(contient_cycle(g))
```

Résultat

```
True
False
True
```

5 Distance entre deux sommets

Écrire une fonction `distance` qui étant donné un graphe `G` ainsi que deux sommets `source` et `arrivee` détermine la plus petite longueur d'un chemin de `source` vers `arrivee`. Si `arrivee` n'est pas accessible depuis `source`, la fonction renverra `-1`.

Indication. On reprendra l'algorithme du parcours en largeur d'un graphe en le modifiant. On enfilera dans la file `à_explorer` le couple `(sommet, distance)` où `distance` correspond à la

longueur d'un plus petit chemin entre `source` et `sommet`. Ainsi, si l'on défile le couple `(sommet, distance)` et que `sommet` est égal à `arrivee`, alors on pourra renvoyer la valeur `distance`.
En particulier :

- on commencera par enfiler le couple `(depart, 0)` dans la file `à_explorer` ;
- si le `sommet` courant se trouve à `distance` du sommet `source`, alors ses voisins non encore explorés se trouvent à `distance + 1` du sommet `source`, lorsque le graphe est parcouru en largeur.

Code

```
def distance(G, depart, arrivee):
    """ Graphe, Sommet, Sommet -> int
    Renvoie la longueur du plus petit chemin entre depart et arrivee dans G """
    pass
```

Code

```
for source in g4.sommets():
    dist = " | ".join([str(distance(g4, source, destination)).center(4)
                       for destination in g4.sommets()])
    print(dist)
```

Résultat

```
 0   |  1   |  2   |  3   |  3   |  4
-1   |  0   |  1   |  2   |  2   |  3
-1   |  2   |  0   |  1   |  1   |  2
-1   |  3   |  4   |  0   |  2   |  1
-1   |  1   |  2   |  3   |  0   |  4
-1   |  2   |  3   |  4   |  1   |  0
```

6 Diamètre d'un graphe

Écrire une fonction `diametre` qui étant donné un graphe `G` en détermine le diamètre `d` ainsi que les deux sommets `(source, destination)` les plus éloignés du graphe. On rappelle que le diamètre d'un graphe est la plus grande distance possible entre deux sommets `source` et `destination` quelconques du graphe.

Code

```
def diametre(G):
    """ Graphe -> int
    Renvoie la distance maximale entre deux sommets quelconques du graphe
    """
    pass
```

Code

```
for g in [g1, g2, g3, g4]:
    print(diametre(g))
```

Résultat

```
(2, (1, 2))
(3, (0, 7))
(1, (1, 2))
(4, (0, 5))
```