IX - TP 4

Parcours de graphe

On utilisera dans ce tp les classes `Graphe`, `Pile`, `File` définies dans le fichier `ds.py`, ainsi que la fonction `randint` du module `random` qui étant donné deux entiers `a` et `b` renvoie un entier aléatoire compris entre `a` (inclus) et `b` (inclus). On rappelle que la classe `Graphe` permet de représenter des graphes orientés.

Code

```
from ds import Graphe, Pile, File
from random import randint
```

Quelques modifications ont été apportées par rapport aux classes écrites en cours :

- la méthode `log` des variables de classe `Pile` (resp. `File`) affiche toutes les opérations d'empilement (resp. enfilement) et de dépilement (resp. défilement) auxquelles elles ont été soumises.

Code

```
p = Pile()
p.empiler(1)
p.depiler()
p.empiler(2)
p.log()
```

Résultat

```
Empile 1
Dépile 1
Empile 2
```

- la méthode `sommets` et la méthode `voisins` de la classe `Graphe` renvoient une liste de sommets **triée par ordre croissant**.

Afin de faciliter l'écriture des codes python, on s'autorisera **dans ce tp uniquement** l'utilisation du mot-clé python `in`. L'instruction `val in collection` renvoie `True` si `val` est un élément de la `collection`, `False` sinon. Cette instruction est valide lorsque `collection` est une liste, un dictionnaire (dans ce cas on teste si `val` est une clé du dictionnaire)...

Code

```
l = [0, 1, 2]
d = {0:'a', 1:'b', 2:'c'}
print(1 in l)
print(42 in l)
print(1 in d)
print('b' in d)
```

Résultat

```
True
False
True
False
```

1 Exemples de graphes non orientés

Écrire une fonction `arcs_vers_graphe_no` qui étant donnée une liste d'arêtes construit le **graphe non orienté** correspondant. On représentera un graphe non orienté à l'aide d'un graphe orienté dans lequel tous les arcs `(s, d)` ont été doublés avec leur arc retour `(d, s)`. La valeur renvoyée par la méthode `taille` est alors le double de la taille du graphe *non orienté*.

On commencera par instancier une variable `g` de type `Graphe` que l'on modifiera de la façon suivante : pour toutes les arêtes `(s, d)` présentes dans la liste `aretes`,

- on créé si besoin les sommets `s` et `d` dans le graphe ;
- on ajoute les arcs `(s, d)` et `(d, s)`.

Code

```
def arcs_vers_graphe_no(aretes):
    """ [(Sommet, Sommet)] -> Graphe
    Sommet peut être de type int ou str
    Construit le graphe non orienté dont on donne la liste des arêtes. """
    pass
```

Code

```
aretes = [(0, 1), (0, 3),
          (0, 6), (1, 4), (1, 6),
          (2, 5), (3, 6), (3, 8),
          (4, 7), (5, 6), (5, 7)]
g1 = arcs_vers_graphe_no(aretes)
g1.decrire()
```

Résultat

```
Graphe d'ordre 9 et de taille 22
Sommet 0 -> 1 3 6
Sommet 1 -> 0 4 6
Sommet 2 -> 5
Sommet 3 -> 0 6 8
Sommet 4 -> 1 7
Sommet 5 -> 2 6 7
Sommet 6 -> 0 1 3 5
Sommet 7 -> 4 5
Sommet 8 -> 3
```

Code

```
aretes = [(0, 1), (0, 5),
    (1, 2), (1, 6), (1, 5), (2, 3),
    (2, 7), (2, 6), (3, 7), (3, 4),
    (4, 5), (4, 6), (4, 7), (5, 6),
    (6, 7)]
g2 = arcs_vers_graphe_no(aretes)
g2.decrire()
```

Résultat

```
Graphe d'ordre 8 et de taille 30
Sommet 0 -> 1 5
Sommet 1 -> 0 2 5 6
Sommet 2 -> 1 3 6 7
Sommet 3 -> 2 4 7
Sommet 4 -> 3 5 6 7
Sommet 5 -> 0 1 4 6
Sommet 6 -> 1 2 4 5 7
Sommet 7 -> 2 3 4 6
```

Code

```
aretes = [(0, 8), (0, 6), (0, 1),
    (0, 2), (0, 7), (1, 2), (1, 6),
    (1, 8), (1, 4), (2, 4), (2, 5),
    (2, 7), (3, 4), (3, 8), (4, 5),
    (4, 7), (5, 7), (6, 8)]
g3 = arcs_vers_graphe_no(aretes)
g3.decrire()
```

Résultat

```
Graphe d'ordre 9 et de taille 36
Sommet 0 -> 1 2 6 7 8
Sommet 1 -> 0 2 4 6 8
Sommet 2 -> 0 1 4 5 7
Sommet 3 -> 4 8
Sommet 4 -> 1 2 3 5 7
Sommet 5 -> 2 4 7
Sommet 6 -> 0 1 8
Sommet 7 -> 0 2 4 5
Sommet 8 -> 0 1 3 6
```

Code

```
g4 = Graphe()
aretes = [(0, 1), (0, 2),
    (1, 2), (1, 3), (2, 3), (4, 5)]
g4 = arcs_vers_graphe_no(aretes)
g4.decrire()
```

Résultat

```
Graphe d'ordre 6 et de taille 12
Sommet 0 -> 1 2
Sommet 1 -> 0 2 3
Sommet 2 -> 0 1 3
Sommet 3 -> 1 2
Sommet 4 -> 5
Sommet 5 -> 4
```

Question 1. Représenter graphiquement les graphes non orientés G_1, G_2, G_3 et G_4 représentés respectivement par les variables g1, g2, g3 et g4.

2 Parcours aléatoire

2.1 Parcours aléatoire

Écrire une fonction `parcours_aleatoire` qui étant donné un graphe `G`, un sommet `depart` et un entier `nombre_sauts` renverra le nombre de fois où chaque sommet du graphe a été visité si on le parcourt depuis le sommet `depart` en choisissant à chaque étape une destination aléatoire parmi les voisins du sommet courant. Plus précisément :

- On utilisera un dictionnaire `visites` pour compter le nombre de fois où le parcours est passé par chaque sommet. Les clés de ce dictionnaire sont les sommets du graphe `G`, et la valeur associée au sommet `s` le nombre de fois où le parcours est passé par le sommet `s`. On initialise le dictionnaire en associant à chaque clé la valeur `0`.
- On utilisera une variable `sommet_courant` pour mémoriser le sommet en cours de visite par le parcours. À chaque itération, on incrémentera de 1 la valeur associée à `sommet_courant` dans le dictionnaire `visites`.
- On choisira le prochain sommet à explorer à partir du sommet courant (un "saut") aléatoirement à l'aide de la fonction `randint`. Si `voisins` est la liste des voisins du sommet courant, on tire un indice aléatoire `i` compris entre `0` et `len(voisins) - 1` et on choisit le sommet `voisins[i]` comme nouveau sommet courant.

Code

```
def parcours_aleatoire(G, depart, nombre_sauts):
    """ Graphe, Sommet, int -> {Sommet: int}
    Parcourt aléatoirement le graphe G depuis le sommet depart
    en effectuant nombre_sauts choix de sommets. """
    pass
```

Code

```
print(parcours_aleatoire(g1, 0, 100))
```

Résultat

```
{0: 16, 1: 12, 2: 13, 3: 6, 4: 8, 5: 22, 6: 16, 7: 7, 8: 0}
```

2.2 Fréquence d'occupation d'un sommet

On appelle **histogramme** un dictionnaire dont les valeurs sont des entiers non tous nuls. Si $k_1, \ldots k_n$ sont les clés du dictionnaire et $v_1, \ldots, v_n$ les entiers correspondant, on construit le dictionnaire des fréquences en associant à chaque clé k_i la valeur $\dfrac{v_i}{v_1 + \ldots + v_n}$.

Question 2. **1.** Écrire une fonction `frequences` qui prend en entrée un histogramme et renvoie le dictionnaire des fréquences associé. On arrondira toutes les fréquences au centième.

Code

```
def frequences(histo):
    """ {Sommet: int} -> {Sommet: float}
    Renvoie le dictionnaire des fréquences (arrondies au centième)
    associé au dictionnaire du nombre d'occurrences des histo du
    ↪ graphe """
    pass
```

2. Pour chaque graphe G de l'énoncé, calculer la fréquence du temps passé en chaque sommet lorsque l'on parcours aléatoirement le graphe pendant 10 000 sauts. Que constate-t-on ?

3 Parcours en profondeur

3.1 Version récursive

Le **parcours en profondeur** d'un graphe correspond à une exploration commençant par un sommet de `depart` du graphe, suivie d'une exploration récursive de chacun de ses voisins. Il est nécessaire de marquer les sommets du graphe par lesquels l'exploration est passée, sans quoi l'algorithme ne termine pas lorsque le graphe possède un cycle.

Dans le code donné ci-dessous, on ajoute à la liste `vus` les sommets dans l'ordre dans lequel ils sont visités lors d'un parcours en profondeur. On n'explore pas (récursivement) de sommet qui aurait déjà été visité.

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
```

Question 3. 1. Pour chacun des graphes G_1, G_2, G_3 et G_4.

   a. Dresser l'arbre d'appel de l'instruction `parcours_profondeur(G, 0, [])`. On indiquera sur les nœuds de l'arbre d'appel uniquement le sommet correspondant du parcours. La liste `vus` est modifiée en place par tous les appels récursifs : on pourra considérer qu'il s'agit d'une variable globale.

   b. En déduire la valeur renvoyée par `parcours_profondeur(G, 0, [])`.

2. Vérifier vos réponses aux questions précédentes en exécutant la fonction `parcours_profondeur`.

3.2 Version itérative

Le parcours en profondeur récursif peut poser problème dans le cas de graphe de très grande taille. En effet, il est (par défaut) impossible d'empiler plus de 1000 appels récursifs lorsque l'on exécute un programme python.

On modifie l'algorithme de parcours en profondeur afin de ne plus utiliser d'appel récursif : on utilise une `Pile` pour stocker les sommets à explorer.

- on initialise une pile `à_explorer` dans laquelle on empile le sommet `depart`
- tant que la pile des sommets `à_explorer` n'est pas vide :
  - on dépile le `sommet` courant que l'on marque comme exploré en l'ajoutant à la liste `vus`.
  - on empile **dans le bon ordre** les voisins non explorés de `sommet` à la pile des sommets `à_explorer`.

Pour le graphe `g1` avec pour sommet de départ `0`, on créé une pile vide `à_explorer` dans laquelle on empile `0`. Lors du premier passage dans la boucle, on dépile `0`, puis on ajoute `0` à la liste `vus`. À la fin du premier passage dans la boucle la pile est constituée des éléments `6 3 1` (l'élément au sommet de la pile est `1`). Le prochain sommet à explorer est donc `1`.

Code

```
def parcours_profondeur_pile(G, depart):
    """ Graphe, Sommet, [Sommet] -> [Sommet]
    Parcours le graphe G depuis le sommet depart en profondeur
    (sachant que la liste des sommets présents dans vus ont déjà été
    visités). """
    vus = []
    à_explorer = ...
    ...
    while not ...:
        sommet = ...
        vus.append(sommet)
        # ajout des voisins non visités
        for destination in reversed(G.voisins(sommet)):
            if ...:
                ...
    return vus
```

Question 4. **1.** Donner un exemple de graphe pour lequel le parcours en profondeur récursif soulève l'erreur `RecursionError`.

2. **a.** Compléter le code de la fonction `parcours_profondeur_pile`.

b. Exécuter `parcours_profondeur_pile(g, 0)` pour chaque graphe `g` de l'énoncé. Que constate-t-on ?

3. **a.** Exécuter pas à pas l'instruction `parcours_profondeur_pile(g1, 0)`. Vous suivrez l'évolution des variables `vus` et `à_explorer` et noterez `EX` (resp. `DX`) lorsque l'on empile `X` (resp. dépile `X`) dans (resp. depuis) `à_explorer`.

b. Modifier la fonction `parcours_profondeur_pile` de telle sorte qu'elle renvoie également la pile `à_parcourir`. À l'aide de la méthode `log` de la classe `Pile`, vérifier votre réponse à la question précédente.

c. Lorsque l'on utilise le mot-clé `continue` dans le corps d'une boucle, ceci a pour effet de sauter directement à la prochaine itération.

Par exemple :

Code

```
for i in [1, 2, 3]:
    if i == 2:
        continue
    print(i)
```

Résultat

```
1
3
```

Expliquer pourquoi il est nécessaire d'ajouter les lignes

Code

```
if sommet in vus:
    continue
```

entre les lignes 9 et 10 de la fonction `parcours_profondeur_pile` pour que celle-ci renvoie effectivement la même liste que la fonction `parcours_profondeur` ?

4 Parcours en largeur

L'algorithme de parcours en largeur d'un graphe est très similaire à l'algorithme de parcours en largeur d'un arbre :

- On maintient la liste des éléments à parcourir dans une file : initialement la file est constituée d'un élément, correspondant au sommet de départ du parcours.
- Tant que la file des éléments à parcourir n'est pas vide :
  1. Défiler le premier sommet A de la file et vérifier qu'il n'a pas été déjà vu.
  2. Traiter le sommet A.
  3. Ajouter à la file d'attente les successeurs de A.

Code

```
def parcours_largeur(G, depart):
    """ Graphe, Sommet -> [Sommet]
    Parcours le graphe G depuis le sommet depart en largeur
    (sachant que la liste des sommets présents dans vus ont déjà été
    visités). """
    vus = []
    à_explorer = ...
    ...
    while ...
        sommet = ...
        if sommet in vus:
            continue
        # opération de traitement
        vus.append(sommet)
        # ajout des voisins non visités
        for destination in G.voisins(sommet):
            if ...:
                ...
    return vus
```

Question 5. **1.** Compléter le code de la fonction `parcours_largeur`.

2. **a.** Exécuter pas à pas l'instruction `parcours_largeur(g1, 0)`. Vous suivrez l'évolution des variables `vus` et `à_explorer` et noterez `EX` (resp. `DX`) lorsque l'on enfile `X` (resp. défile `X`) dans (resp. depuis) `à_explorer`.

b. Modifier la fonction `parcours_largeur` de telle sorte qu'elle renvoie également la file `à_parcourir`. À l'aide de la méthode `log` de la classe `Pile`, vérifier votre réponse à la question précédente.

3. **a.** Pour chacun des graphes de l'énoncé, indiquer pour chaque sommet à quelle distance il se trouve du sommet `0`.

Rappel. Dans un graphe, la **distance** entre deux sommets est la plus petite longueur d'un chemin reliant les deux sommets dans le graphe. Elle n'est pas définie s'il n'existe pas de chemin entre les deux sommets.

b. Exécuter et faire afficher la valeur renvoyée par l'instruction `parcours_largeur(g, 0)` pour chaque graphe de l'énoncé.

Quel est le lien entre le parcours en largeur d'un graphe à partir du sommet `0` et la distance de chaque sommet par rapport au sommet `0` ?

5 Intermède complexité : autour du mot-clé `in`

Dans toutes les fonctions écrites jusqu'à présent, nous avons utilisé le mot-clé `in` afin de tester l'appartenance d'un élément `elt` à une **liste**. On se propose dans cette partie de détailler le fonctionnement en python du mot-clé `in`.

5.1 Test d'appartenance d'un élément à un tableau

On propose le code de la classe `MonTableau` ci-dessous. Celle-ci est munie d'un unique attribut `contenu` de type `list`. On pourra supposer que les éléments de `contenu` sont tous de type `int`. L'instruction `0 in tab` soulève une erreur lorsque `tab` est de type `MonTableau`.

Code

```
class MonTableau:
    def __init__(self, tab):
        """ MonTableau, [int] -> Nonetype """
        self.contenu = tab
```

Code

```
tab = MonTableau([0, 1, 2, 3])
print(tab)
print(tab.contenu)
print(0 in tab)
```

Résultat

```
<__main__.MonTableau object at 0x7f16d2acc7c0>
[0, 1, 2, 3]
---------------------------------
TypeError Traceback (most recent call last)
Input In [141], in <cell line: 4>()
      2 print(tab)
      3 print(tab.contenu)
----> 4 print(0 in tab)

TypeError: argument of type 'MonTableau' is not iterable
```

Question 6. **1.** **a.** Écrire une méthode `__contains__` de la classe `MonTableau` qui étant donné un élément `elt` renvoie `True` si et seulement si l'élément `elt` se trouve dans la liste `contenu` de `self`.

Code

```
def __contains__(self, elt):
    """ MonTableau, int -> bool
    Détermine si elt appartient à self.contenu """
    pass
```

b. Quelle est la complexité de la méthode `__contains__` en fonction de la taille de la liste `contenu` ?

2. Exécuter les instructions `0 in tab`, `4 in tab`. Que constate-t-on ?

5.2 Test d'appartenance d'un élément à un ensemble

En python, le type `dict` est implémenté à l'aide de tables de hachage. Ainsi, lorsque `d` est un dictionnaire, on peut considérer que l'instruction `e in d` s'exécute en **temps constant** (une opération élémentaire). On peut utiliser les dictionnaires pour implémenter une classe `Ensemble` dont l'interface est la suivante :

- `Ensemble()` : créer un ensemble vide ;
- `s.ajouter(e)` : ajouter l'élément `e` à l'ensemble `s` ;
- `s.supprimer(e)` : supprimer l'élément `e` de l'ensemble `s` (ne soulève pas d'erreur si `e` n'est pas un élément de `s`) ;
- `s.__contains__(e)` : renvoie `True` si et seulement si `e` est un élément de l'ensemble `s`.

Question 7. **1.** On donne ci-dessous le code d'une classe `Ensemble`. Celle-ci est munie d'un unique attribut de type `dict`. **Un élément `e` appartient à l'ensemble `s` si et seulement si `e` est une clé de `s.contenu` associé à la valeur `True`.**

Code

```
class Ensemble:
    def __init__(self):
        """ Ensemble -> Nonetype """
        self.contenu = dict()

    def ajouter(self, elt):
        """ Ensemble, elt -> Nonetype """
        pass

    def supprimer(self, elt):
        """ Ensemble, elt -> Nonetype """
        pass

    def __contains__(self, elt):
        """ Ensemble, elt -> bool """
        pass
```

Compléter le code de l'énoncé afin que celui-ci implémente correctement l'interface du type `Ensemble` tel que décrit dans l'énoncé.

2. Quelle est la complexité de la méthode `__contains__` de la classe `Ensemble` en fonction du nombre d'éléments de l'ensemble ?

6 Conclusion

Question 8. Modifier le code des fonctions `parcours_profondeur_pile` et `parcours_largeur` afin d'en améliorer la complexité. On utilisera un objet `vus` de type `Ensemble` pour mémoriser les sommets déjà visités lors du parcours. On renverra une liste `ordre_visite` contenant la liste de tous les sommets visités sans répétition dans l'ordre chronologique du parcours.

Code

```
def parcours_profondeur_mieux(G, depart):
    """ Graphe, Sommet -> [Sommet]
    Parcours le graphe G depuis le sommet depart en profondeur """
    pass

def parcours_largeur_mieux(G, depart):
    """ Graphe, Sommet -> [Sommet]
    Parcours le graphe G depuis le sommet depart en largeur """
    pass
```

Code

```
print(parcours_profondeur_mieux(g1, 0))
print(parcours_largeur_mieux(g1, 0))
```

Résultat

```
[0, 1, 4, 7, 5, 2, 6, 3, 8]
[0, 1, 3, 6, 4, 8, 5, 7, 2]
```