IX - TP 3

Implémentation des graphes

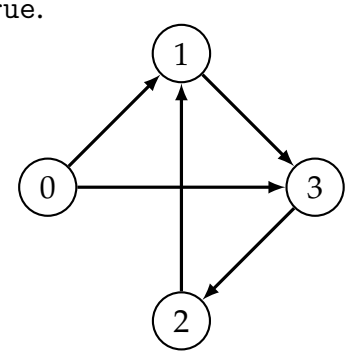
Il existe de multiples manières de représenter un graphe, selon la nature du graphe mais aussi selon la nature des opérations que l'on souhaite effectuer sur ce graphe. Quelque soit le choix retenu, on souhaite pouvoir :

- utiliser des opérations de construction du graphe. Il faut pouvoir ajouter/supprimer des sommets, des arcs…
- utiliser des opérations de parcours du graphe : parcourir la liste des sommets, des arcs, ou bien des arcs issus d'un sommet donné…

On commence dans un premier temps par implémenter **les graphes orientés**. On ne considérera que les 1-graphes, c'est à dire les graphes dans lesquelles l'arc $(u,v) \in S^2$ est présent une fois au plus.

1 Matrices d'adjacence

Les sommets du graphe sont les nombres $0, 1, \ldots, n-1$, pour un certain entier $n \in \mathbb{N}$. On représente le graphe à l'aide d'une **matrice** `adj` de taille $n \times n$ (un tableau constitué de n lignes et n colonnes). Il existe un arc entre les sommets i et j du graphe si et seulement si `adj[i][j]` vaut `True`.



Graphe G_1

indice de colonne j

indice de ligne i	0	1	2	3
0	FALSE	TRUE	FALSE	TRUE
1	FALSE	FALSE	FALSE	TRUE
2	FALSE	TRUE	FALSE	FALSE
3	FALSE	FALSE	TRUE	FALSE

Matrice d'adjacence `adj` de G_1.

On donne ci-dessous le code python du constructeur de la classe `Graphe`.

Code

```
class Graphe:
    """ Un graphe représenté par une matrice d'adjacence.
    Les sommets sont les nombres 0, 1, ..., n - 1. """
    def __init__(self, n):
        self.n = n
        self.adj = [[False]*n for _ in range(n)]

```

1. **a.** Initialiser une variable `g1` de type `Graphe`, représentant un graphe constitué des 4 somments 0, 1, 2, et 3. Initialement, aucun arc n'est présent dans le graphe `g1`.

b. Écrire une méthode `sommets` de la classe `Graphe` qui étant donné un graphe `self` renvoie la liste des sommets qui le constituent.

Code

```
def sommets(self):
    """ Graphe -> [int]
    Renvoie la liste des sommets du graphe self """
    pass
```

Code

```
print(g1.sommets())
```

Résultat

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

2. **a.** Écrire une méthode `ajouter_arc` de la classe `Graphe` qui étant donné deux sommets `source` et `destination` du graphe `self` ajoute l'arc `source` → `destination` au graphe `self`.

Code

```
def ajouter_arc(self, source, destination):
    """ Graphe, int, int -> int
    Ajoute l'arc source -> destination """
    pass
```

b. À l'aide de la méthode `ajouter_arcs`, construire le graphe `g1` de l'énoncé.

Code

```
for row in g1.adj:
    print(row)
```

Résultat

```
[False, True, True, True]
[False, False, True, True]
[False, True, False, False]
[False, False, True, False]
```

c. Écrire une méthode `est_voisin` de la classe `Graphe` qui étant donné deux sommets `source` et `destination` du graphe `self` renvoie `True` si et seulement si il existe un arc reliant `source` à `destination` dans le graphe `self`.

Code

```
def est_voisin(self, source, destination):
    """ Graphe, int, int -> bool
    Détermine si source et destination sont voisins """
    pass
```

Code

```
print(g1.est_voisin(0, 1))
print(g1.est_voisin(1, 0))
```

Résultat

```
True
False
```

3. **a.** Écrire une méthode `voisins` de la classe `Graphe` qui étant donné un `sommet` du graphe `self` renvoie la liste des voisins de `sommet`.

Code

```
def voisins(self, sommet):
    """ Graphe, int -> [int]
    Renvoie la liste des voisins de sommet """
    pass
```

Code

```
print(g1.voisins(0))
print(g1.voisins(1))
```

Résultat

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

b. Écrire une méthode `liste_arcs` de la classe `Graphe` qui renvoie la liste des couples `(source, destination)` correspondants à des arcs du graphe `self`.

Code

```
def liste_arcs(self):
    """ Graphe -> [(int, int)]
    Renvoie la liste des arcs du graphe """
    pass
```

Code

```
print(g1.liste_arcs())
```

Résultat

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

4. **a.** Écrire une méthode `ordre` de la classe `Graphe` qui renvoie l'ordre du graphe.

Rappel. L'ordre d'un graphe est le nombre de sommets qui composent le graphe.

Code

```
def ordre(self):
    """ Graphe -> int
    Renvoie l'ordre du graphe  """
    pass
```

Code

```
print(g1.ordre())
```

Résultat

```
4
```

b. Écrire une méthode `taille` de la classe `Graphe` qui renvoie la taille du graphe.

Rappel. La taille d'un graphe est le nombre d'arêtes qui composent le graphe.

Code

```
def taille(self):
    """ Graphe -> int
    Renvoie la taille du graphe """
    pass
```

Code

```
print(g1.taille())
```

Résultat

```
7
```

c. Écrire une méthode `decrire` qui affiche les informations du graphe `self` sur le modèle donné ci-dessous.

Code

```
def decrire(self):
    """ Graphe -> Nonetype
    Affiche une description du graphe self """
    pass
```

Code

```
g1.decrire()
```

Résultat

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

5. **a.** Écrire une méthode `ajouter_sommet` de la classe `Graphe` qui ajoute un sommet au graphe. Ainsi, si le graphe `g` est constitué de n sommets, alors après l'application de la méthode `ajouter_sommet` le graphe est constitué de $n+1$ sommets (le sommet ajouté est n) : la méthode `ajouter_sommet` modifiera à la fois l'attribut `n` et l'attribut `adj` du graphe `self`. Aucun arc ne sera ajouté par cette méthode.

Code

```
def ajouter_sommet(self):
    """ Graphe -> Nonetype
    Ajoute un sommet au graphe """
    pass
```

Code

```
g1.ajouter_sommet()
g1.decrire()
```

Résultat

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

b. Écrire une méthode `supprimer_arc` de la classe `Graphe` qui supprime (si présent) l'arc reliant `source` à `destination` dans le graphe `self`. Cette méthode ne produira pas d'erreur dans le cas où le graphe ne contient pas l'arc `(source)` $\to$ `(destination)`.

Code

```
def supprimer_arc(self, source, destination):
    """ Graphe, int, int -> Nonetype
    Supprime l'arc source -> destination """
    pass
```

c. Écrire une méthode `supprimer_sommet` de la classe `Graphe` qui supprime le sommet `i` du graphe `self`. La méthode `supprimer_sommet` modifiera l'attribut `n` et l'attribut `adj`. Si $j > i$ est un sommet du graphe, alors son numéro sera $j-1$ après application de la méthode `supprimer_sommet`, les sommets $j < i$ conserveront le même numéro.

Code

```
def supprimer_sommet(self, i):
    """ Graphe, int -> Nonetype
    Supprime le sommet i du graphe. Les sommets j > i sont
    renommés en j - 1. """
    pass
```

Code

```
g1.supprimer_sommet(2)
g1.decrire()
```

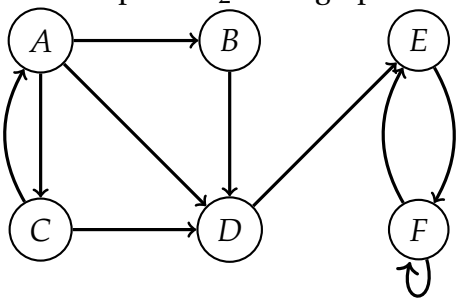
Résultat

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

2 Dictionnaire d'adjacence

Les sommets du graphe sont quelconques (des nombres, non forcément consécutifs, des lettres, des mots, etc). On représente un graphe à l'aide d'un **dictionnaire** d'adjacence, noté `adj`. Si $G = (S, A)$, alors les clés du dictionnaire `adj` seront les éléments de S. La valeur associée à la clé `source` sera la liste des sommets `destination` de telles que `(source, destination)` $\in A$.

Par exemple si G_2 est le graphe :



Alors le dictionnaire correspondant est :

Code

```
adj = {'A': ['B', 'C', 'D'],
       'B': ['D'],
       'C': ['A', 'D'],
       'D': ['E'],
       'E': ['F']
       'F': ['E', 'F']}
```

On donne ci-dessous le code python du constructeur de la classe `Graphe`.

Code

```
class Graphe:
    """ Un graphe représenté par un dictionnaire d'adjacence. """
    def __init__(self, n):
        self.n = n
        self.adj = [[False]*n for _ in range(n)]

```

Toutes les questions de cette partie concernent *cette* classe `Graphe`. Les sommets seront ici des chaînes de caractère.

1. **a.** Initialiser une variable g2 de type `Graphe` représentant un graphe initialement vide.

a. Écrire une méthode `ajouter_sommet` qui ajoute un sommet `s` au graphe `self`. Initialement, aucun arc ne sort du sommet ajouté. Si le sommet est déjà présent dans le graphe, alors cette méthode ne fait rien.

Code

```
def ajouter_sommet(self, s):
    """ Graphe, str -> Nonetype
    Ajoute le sommet s au graphe self """
    pass
```

b. Écrire une méthode `ajouter_arc` de la classe `Graphe` qui étant donné deux sommets `source` et `destination` du graphe `self` ajoute l'arc `source` $\rightarrow$ `destination` au graphe `self`.

Code

```
def ajouter_arc(self, source, destination):
    """ Graphe, str, str -> Nonetype
    Ajoute l'arc source -> destination au graphe self """
    pass
```

c. À l'aide des méthodes `ajouter_sommet` et `ajouter_arc`, construire le graphe `g2` de l'énoncé.

Code

```
print(g2.adj)
```

Résultat

```
{'A': ['B', 'C', 'D'], 'B': ['D'], 'C': ['A', 'D'], 'D': ['E'],
 ↪  'E': ['F'], 'F': ['E', 'F']}
```

2. Écrire les méthodes `est_voisin`, `sommets`, `voisins`, `liste_arcs`, `ordre`, `taille`, `decrire`, `supprimer_arc`, `supprimer_sommet` (dans cet ordre) de la classe `Graphe` (ainsi que leur signature et leur documentation).

VOUS TESTEREZ VOS MÉTHODES.

Code

```
g2.decrire()
```

Résultat

```
Graphe d'ordre 6 et de taille
 ↪  10
Sommet A -> B C D
Sommet B -> D
Sommet C -> A D
Sommet D -> E
Sommet E -> F
Sommet F -> E F
```

Code

```
g2.supprimer_sommet('D')
g2.decrire()
```

Résultat

```
Graphe d'ordre 5 et de taille 6
Sommet A -> B C
Sommet B ->
Sommet C -> A
Sommet E -> F
Sommet F -> E F
```

Rappels.

Si `l` est un objet de type `list`, `i` un indice compatible avec la taille de la liste, et `e1` un élément présent dans la liste `l`, `e2` un élément non présent dans la liste `l` :

- `l.copy()` renvoie une copie indépendante de `l`
- `e1 in l` s'évalue à `True` ; `e2 in l` s'évalue à `False`
- `l.pop()` supprime le dernier élément de la liste, `l.pop(i)` supprime l'élément d'indice `i` de la liste.
- `l.remove(e1)` supprime la première occurrence de `e1` dans `l` ; `l.remove(e2)` soulève une erreur
- `l.append(e2)` ajoute l'élément `e2` à la fin de la liste ; `l.insert(i, e2)` insère l'élément `e2` à l'indice `i`.
- `enumerate(l)` renvoie la liste `[(i, l[i]) for i in range(len(l))]`

Si `d` est un objet de type `dict`, `k1` une clé présente dans le dictionnaire, `v1` la valeur associée à la clé `k1`, `k2` une clé non présente dans le dictionnaire :

- `d[k1]` renvoie `v1` ; `d[k2]` soulève une erreur ; `d.get(k1)` renvoie `v1`, `d.get(k2)` renvoie `None` ;
- `k1 in d` s'évalue à `True` ; `k2 in d` s'évalue à `False`
- `d.pop(k1)` supprime `k1` de `d` et renvoie `v1` ; `d.pop(k2)` soulève une erreur ;
- `d.setdefault(k1, "valeur par defaut")` ne fait rien car `k1` est déjà présent dans le dictionnaire, `d.setdefault(k2, "valeur par défaut")` associe à `k2` la valeur `"valeur par défaut"`
- `d.keys()` renvoie la liste des clés du dictionnaire ; `d.values()` renvoie la liste des valeurs du dictionnaire ; `d.items()` renvoie la liste des couples `(clé, valeur)` du dictionnaire ;