Terminale - Spécialité NSI

TP - Les graphes

Graphes orientés

Dans la première partie de ce TP, nous allons commencer par implémenter la classe qui permettra de manipuler des **graphes orientés**.
Les sommets et les arcs seront gérés grâce à **une liste d'adjacence** représentée, en python, par **un dictionnaire**.

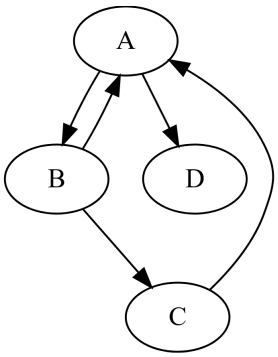
Exercice 1 : Création de la classe GrapheOriente *

Dans ce premier exercice, vous allez créer la classe **GrapheOriente**. Nous viendrons améliorer cette classe avec différentes implémentations de méthodes dans les prochains exercices.
La classe **GrapheOriente** est définie par un seul attribut que nous nommerons **dico_adjancence**, représentant la liste d'adjacence du graphe. Cet attribut est un **dictionnaire**.

Chaque clé de ce dictionnaire sera un sommet, associé à une liste de sommets, qui représentera les sommets adjacents (voisins) à ce sommet.
Par exemple, si on regarde le dictionnaire suivant :

```
d = {"A":["B", "D"], "B":["A", "C"], "C":["A"], "D":[]}
```

Cela représente le graphe orienté suivant :



Définissez la classe GrapheOriente et implémentez son constructeur qui n'aura pas de paramètres (en dehors de self). Le constructeur devra **initialiser** l'attribut **dico_adjancence** avec un dictionnaire vide.

```
class GrapheOriente:
    def __init__(self):
        ......
```

Exercice 2 : Les sommets *

Vous allez maintenant implémenter les méthodes permettant de gérer les **sommets** du graphe.

1. Implémentez la méthode "ajouter_sommet(self, sommet)" de la classe **GrapheOriente** qui ajoute, dans l'attribut **dico_adjancence** de l'objet, une nouvelle clé représentée par le paramètre **sommet**, en l'associant à une nouvelle liste vide. Attention, il faut ajouter le sommet **seulement si dico_adjancence ne contient pas déjà la clé correspondante** (en d'autres termes, seulement si le sommet n'existe pas déjà dans le graphe).

```
def ajouter_sommet(self, sommet):
    ......
    ......
    ......
```

2. Implémentez la méthode "sommets(self)" de la classe **GrapheOriente** qui renvoie **la liste des sommets** du graphe, c'est à dire, la liste des clés de **dico_adjancence**, et cela, **triée dans l'ordre croissant**. Pour cela, vous pouvez utiliser la fonction **"sorted"** de python qui, appliquée sur une liste, renvoie la liste triée.

```
def sommets(self):
    ......
    ......
    ......
```

3. Implémentez la méthode "contient_sommet(self, sommet)" de la classe **GrapheOriente** qui renvoie **True** si le graphe contient le sommet passé en paramètre et **False** si ça n'est pas le cas. Il faut donc vérifier que **dico_adjancence** contient (ou non) une clé qui correspond à **sommet**.

```
def contient_sommet(self, sommet):
    ......
    ......
    ......
```

Tests

```
>>> test_graphe_oriente = GrapheOriente()
>>> test_graphe_oriente.ajouter_sommet("A")
>>> test_graphe_oriente.ajouter_sommet("B")
>>> test_graphe_oriente.ajouter_sommet("C")
>>> test_graphe_oriente.ajouter_sommet("D")
>>> test_graphe_oriente.sommets()
["A", "B", "C", "D"]
>>> test_graphe_oriente.contient_sommet("C")
True
>>> test_graphe_oriente.contient_sommet("E")
False
```

Exercice 3 : Les arcs *

Vous allez maintenant implémenter les méthodes permettant de gérer les **arcs** (lien entre les sommets) du graphe.

1. Implémentez la méthode "ajouter_arc(self, sommet_1, sommet_2)" de la classe **GrapheOriente** qui ajoute **sommet_1** et **sommet_2** au graphe puis crée un **arc** dirigé depuis le premier sommet vers le second. Pour cela, il faut simplement ajouter **sommet_2** à la liste des voisins de **sommet_1** dans l'attribut **dico_adjancence** de l'objet.

```
def ajouter_arc(self, sommet_1, sommet_2):
    ......
    ......
    ......
```

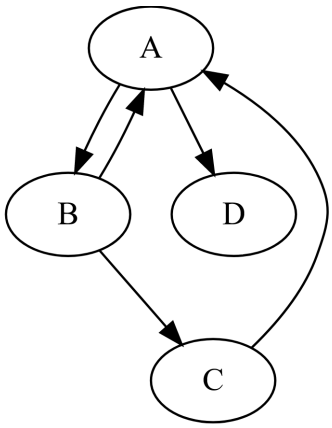
2. Implémentez la méthode "arc_existe(self, sommet_1, sommet_2)" de la classe **GrapheOriente** qui renvoie **True** si l'arc allant de **sommet_1** vers **sommet_2** existe dans le graphe et **False** si ce n'est pas le cas. En d'autres termes, il faut vérifier que **sommet_2** apparaît bien dans la liste des voisins de **sommet_1**.

```
def arc_existe(self, sommet_1, sommet_2):
    ......
    ......
    ......
```

Tests

Afin de facilitez vos tests, vous allez importez une fonction (afficher_graphe(graphe)) qui vous permettra de visualiser facilement votre graphe. Veillez à **bien avoir respecté le nom de la classe GrapheOriente et de ses attributs**. Placez le fichier **"afficheur_graphe.py"** dans le même dossier que votre fichier python courant.
Importez le module au début de votre fichier ainsi :

```
from afficheur_graphe import afficher_graphe
#Tests
>>> mon_graphe = GrapheOriente()
>>> mon_graphe.ajouter_arc("A", "B")
>>> mon_graphe.ajouter_arc("A", "D")
>>> mon_graphe.ajouter_arc("B", "A")
>>> mon_graphe.ajouter_arc("B", "C")
>>> mon_graphe.ajouter_arc("C", "A")
>>> mon_graphe.arc_existe("B", "C")
True
>>> mon_graphe.arc_existe("C", "B")
False
>>> afficher_graphe(mon_graphe)
```



Pour la suite des exercices de cette section, nous nous baserons sur le graphe orienté suivant, que nous nommerons **"graphe_oriente_tp"** :

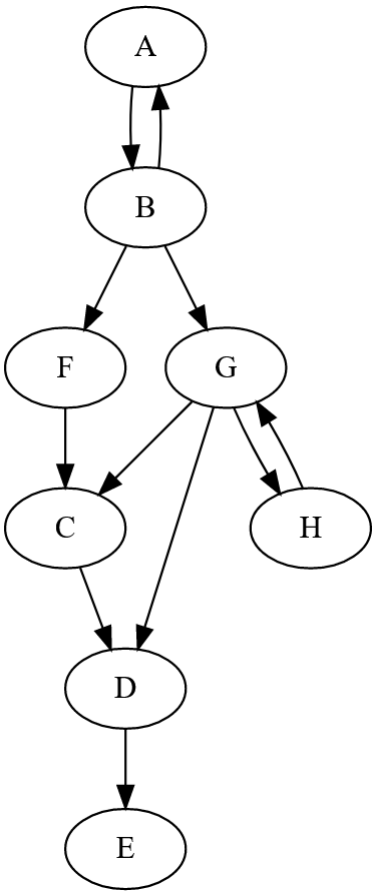


FIGURE 1 – Un graphe oriente "graphe_oriente_tp"

Exercice 4 : Création d'un graphe orienté *

A l'aide des lignes de codes utilisées pour tester votre classe **GrapheOriente** lors de l'exercice précédent, **créez le graphe ci-dessus (graphe_oriente_tp) en python**.
Gardez le dans votre fichier python en le stockant dans une variable **graphe_oriente_tp** (donc pas dans la console) car vous le réutiliserez plus tard.
Enfin, **visualisez le résultat** pour valider en utilisant (cette fois-ci dans la console) la fonction **afficher_graphe**.

Exercice 5 : Voisins d'un sommet *

Implémentez et testez (sur **graphe_oriente_tp**) la méthode **"voisins(self, sommet)"** de la classe **GrapheOriente** qui renvoie la liste des sommets voisins de **sommet** dans le graphe, c'est à dire, les sommets **accessibles** depuis ce sommet. La liste devra, là aussi, être triée dans l'ordre croissant (utilisez **sorted**).

```
def voisins(self, sommet):
    ......
    ......
```

```
#Tests
>>> graphe_oriente_tp.voisins("A")
["B"]
>>> graphe_oriente_tp.voisins("B")
["A", "F", "G"]
>>> graphe_oriente_tp.voisins("G")
["C", "D", "H"]
>>> graphe_oriente_tp.voisins("E")
[]
```

Exercice 6 : Degré d'un sommet **

Implémentez et testez (sur **graphe_oriente_tp**) la méthode **"degre(self, sommet)"** de la classe **GrapheOriente** qui renvoie le degré du **sommet** passé en paramètre. Pour rappel, **pour un graphe orienté**, le degré d'un sommet correspondant au nombre d'arcs **sortants** (arcs ayant pour origine ce sommet) additionné au nombre d'arcs **entrants** (ayant pour destination ce sommet). Par exemple, G a 3 arcs sortants et 2 arcs entrants, son degré est donc de 5.

```
def degre(self, sommet):
    ......
    ......
```

```
#Tests
>>> graphe_oriente_tp.degre("G")
5
>>> graphe_oriente_tp.degre("A")
2
>>> graphe_oriente_tp.degre("B")
4
>>> graphe_oriente_tp.degre("E")
1
```

Graphes non orientés

Dans cette seconde partie du TP, nous allons implémenter la classe qui permettra de manipuler des **graphes non orientés**. Pour rappel, dans un **graphe non orienté**, on parle **d'arête** et non plus **d'arc** car les liens entre les sommets ne sont plus orientés. Ainsi, si un sommet "A" est relié à un sommet "B", le sommet "B" est lui aussi relié au sommet "A". Il y a donc une **arête** entre "A" et "B" et/ou entre "B" et "A".

La classe GrapheNonOriente est définie de la même manière que la classe GrapheOriente, c'est à dire, avec un dictionnaire permettant de gérer la liste des voisins de chaque sommet. Seul l'ajout d'une arête et le calcul du degré d'un sommet diffère. On peut donc dire que GrapheNonOriente est un sous-type de GrapheOriente.

Pour ne pas à avoir à récrire tout le code déjà défini dans GrapheOriente, nous allons utiliser le mécanisme **d'héritage** que nous avons rapidement abordé lors du TP sur les arbres. Nous n'aurons même pas besoin d'écrire de constructeur, car celui-ci est identique à celui de GrapheOriente.

Définissez la classe GrapheNonOriente comme suit :

```
class GrapheNonOriente(GrapheOriente):
```

Exercice 7 : Gestion des arêtes *

Afin de gérer les arêtes, vous allez redéfinir la méthode **"ajouter_arc(self, sommet_1, sommet_2)"** pour pouvoir ajouter un arc. Pour donner plus d'explications sur ce mécanisme, lorsqu'une classe "hérite" depuis une autre classe, c'est comme si tout le code de la première classe était copié dans la nouvelle. Le terme "redéfinir" signifie "récrire" le code d'une méthode dans la nouvelle classe pour l'adapter à un comportement voulu. Il suffit de récrire la fonction correspondante.

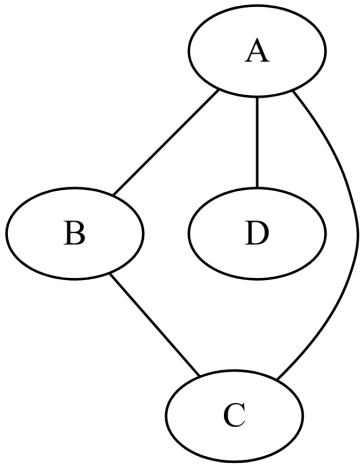
Ici, le but va être de redéfinir la méthode de telle sorte que quand on ajoute un arc depuis **sommet_1** vers **sommet_2**, cela crée aussi un arc depuis **sommet_2** vers **sommet_1**. En résumé, **sommet_2** est ajouté à la liste des voisins de **sommet_1** et inversement. Ainsi, on obtient alors bien **une arête**.

Implémentez la méthode "ajouter_arc(self, sommet_1, sommet_2)" de la classe **GrapheNonOriente** qui ajoute **sommet_1** et **sommet_2** au graphe puis crée une **arête** entre les deux sommets.

```
def ajouter_arc(self, sommet_1, sommet_2):
    ......
    ......
    ......
```

Tests

```
>>> mon_graphe = GrapheNonOriente()
>>> mon_graphe.ajouter_arc("A", "B")
>>> mon_graphe.ajouter_arc("A", "D")
>>> mon_graphe.ajouter_arc("B", "C")
>>> mon_graphe.ajouter_arc("C", "A")
>>> mon_graphe.arc_existe("B", "C")
True
>>> mon_graphe.arc_existe("C", "B")
True
>>> mon_graphe.arc_existe("A", "C")
True
>>> afficher_graphe(mon_graphe)
```



Pour la suite des exercices de cette section, nous nous baserons sur le graphe non orienté suivant, que nous nommerons **"graphe_non_oriente_tp"** :

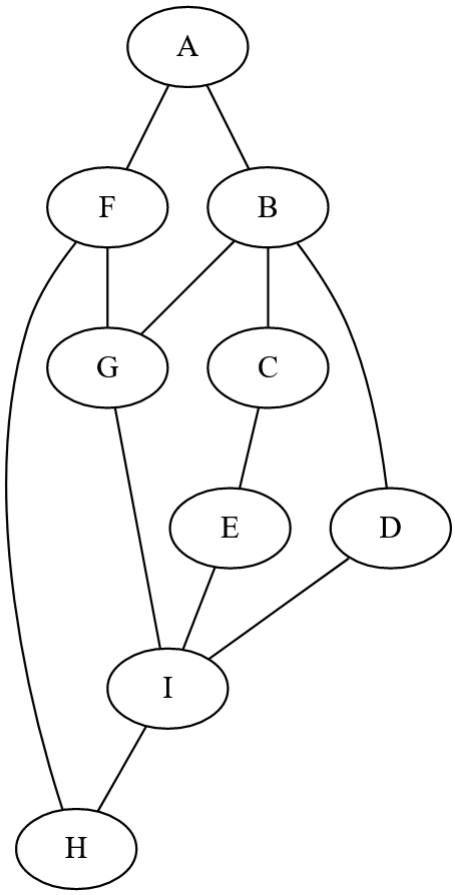


FIGURE 2 – Un graphe oriente "graphe_non_oriente_tp"

Exercice 8 : Création d'un graphe non orienté *

A l'aide des lignes de codes utilisées pour tester votre classe **GrapheNonOriente** lors de l'exercice précédent, **créez le graphe ci-dessus (graphe_non_oriente_tp) en python**. Gardez le dans votre fichier python en le stockant dans une variable **graphe_non_oriente_tp** (donc pas dans la console) car vous le réutiliserez plus tard.
Enfin, **visualisez le résultat** pour valider en utilisant (cette fois-ci dans la console) la fonction **afficher_graphe**.

Exercice 9 : Degré d'un sommet pour un graphe non orienté *

Pour **un graphe non orienté** comme il n'y a plus d'arcs sortants et entrants, le calcul du degré d'un sommet est plus simple : Il s'agit du nombre d'arêtes auxquelles est relié le sommet. Il s'agit donc du nombre de voisins du sommet sur le graphe.

Implémentez la méthode "degre(self, sommet)" de la classe **GrapheNonOriente** qui renvoie le degré du sommet passé en paramètre.

```
def degre(self, sommet):
    ......
```

```
#Tests
>>> graphe_non_oriente_tp.degre("A")
2
>>> graphe_non_oriente_tp.degre("B")
4
>>> graphe_non_oriente_tp.degre("G")
3
>>> graphe_non_oriente_tp.degre("H")
2
```

Algorithmes sur les graphes

Dans la section suivante, vous implémenterez les algorithme dans la classe **GrapheOriente**. Grâce à **l'héritage**, la classe **GrapheNonOriente** aura aussi accès à ces nouvelles méthodes.

Exercice 10 : Parcours en largeur **

Pour réaliser le **parcours en largeur** d'un graphe, à partir d'un sommet donné, on utilise une **file**. La méthode est la suivante :

1. On initialise une file puis on y ajoute le sommet de départ.
2. On initialise une structure qui permettra de mémoriser les sommets déjà rencontrés. en python, on utilisera un dictionnaire. On ajoute également le sommet de départ à la structure.
3. Tant que la file n'est pas vide :
   (a) On retire le sommet courant depuis la file (on le stocke dans une variable).
   (b) On **affiche** le sommet courant (print).
   (c) Pour chaque voisin du sommet courant, si le voisin n'a pas déjà été rencontré (vérification avec le dictionnaire), on l'ajoute à la file et on l'ajoute aussi comme clé du dictionnaire pour mémoriser qu'on a déjà rencontré ce sommet.

Comme expliqué, pour vérifier si un sommet a déjà été rencontré, on utilise un **dictionnaire**. On peut l'utiliser comme suit, en associant le sommet à la valeur **True**, dans le dictionnaire, pour indiquer qu'il a déjà été rencontré. L'utilisation d'un dictionnaire (au lieu d'une liste, par exemple) est justifié pour des raisons de **complexité temporelle**.

```
sommets_rencontres = {}
#Quand on rencontre un nouveau "sommet" :
sommets_rencontres[sommet] = True
#Pour vérifier si on a déjà rencontré un "sommet"
if sommet in sommets_rencontres:
    ...
```

Importez la classe **File** (codée lors d'un précédent TP) dans votre programme. La classe doit se nommer "File" et doit se trouver dans un fichier **"File.py"**. Vous pouvez utiliser le fichier **"File.py"** corrigé et fourni par le professeur.

```
from File import File
```

Implémentez et testez (sur **graphe_oriente_tp** puis **graphe_non_oriente_tp**) la méthode **"parcours_largeur(self, sommet)"** de la classe **GrapheOriente** qui effectue le **parcours en largeur** du graphe, à partir du sommet passé en paramètre, en affichant, successivement, les sommets visités.

```
def parcours_largeur(self, sommet):
    ......
    ......


#Tests
>>> graphe_oriente_tp.parcours_largeur("A")
A
B
F
G
C
D
H
E
>>> graphe_non_oriente_tp.parcours_largeur("A")
A
B
F
C
D
G
H
E
I
```

Exercice 11 : Parcours en profondeur **

Le **parcours en profondeur** d'un graphe, à partir d'un sommet, a quasiment la même méthode que le **parcours en largeur** sauf qu'on utilise une pile (et donc, on empile / depile, au lieu d'ajouter / retirer de la file...). Le seul autre changement, se situe au niveau de l'ordre des voisins du sommet courant que l'on traite. Comme on les empile, leur ordre sera inversé. Il faut donc appliquer la fonction **reversed** sur la liste des voisins parcourue lorsque, dans la boucle, on ajoute les prochains voisins à traiter.

Importez la classe **Pile** (codée lors d'un précédent TP) dans votre programme. La classe doit se nommer "Pile" et doit se trouver dans un fichier **"Pile.py"**. Vous pouvez utiliser le fichier **"Pile.py"** corrigé et fourni par le professeur.

```
from Pile import Pile
```

Implémentez et testez (sur **graphe_oriente_tp** puis **graphe_non_oriente_tp**) la méthode **"parcours_profondeur(self, sommet)"** de la classe **GrapheOriente** qui effectue le

parcours en profondeur du graphe, à partir du sommet passé en paramètre, en affichant, successivement, les sommets visités.

```
def parcours_profondeur(self, sommet):
    ......
    ......

#Tests
>>> graphe_oriente_tp.parcours_profondeur("A")
A
B
F
C
D
E
G
H
>>> graphe_non_oriente_tp.parcours_profondeur("A")
A
B
C
E
I
H
D
G
F
```

Exercice 12 : Trouver un chemin ***

On aimerais maintenant avoir une méthode qui permettrait de trouver un chemin allant d'un sommet à un autre dans le graphe (et ne passant pas deux fois par le même sommet).

Le but est de procéder de manière **récursive** : A partir du sommet de départ, pour chacun de ses voisins, je regarde si je peux trouver un chemin allant du voisin jusqu'au sommet désiré. Le **cas de base** est donc quand le sommet de départ est égal au sommet de destination.

La méthode (**récursive**) est la suivante :

1. On va gérer une liste "chemin" qui sera modifiée et transmisse lors des appels récursifs.
2. Au début, on ajoute le sommet "de départ" dans la liste "chemin"
3. Si le sommet de départ et le sommet de destination sont identiques, on retourne la liste **cas de base**.
4. Sinon, pour tous les voisins du sommet de départ :
   (a) Si le voisin n'est pas déjà dans le chemin, on calcule un nouveau chemin (que l'on stocke donc dans une variable) en faisant **un appel récursif** avec comme paramètre, le voisin en question comme sommet de départ, le sommet de destination qui

reste inchangé, et enfin, la liste représentant le chemin, et contenant donc le chemin courant, actuellement en cours de construction.

(b) Si ce nouveau chemin (liste) calculé n'est pas nul (None), cela signifie qu'on a donc un chemin allant du départ à la destination, on le retourne donc.

5. Quand on sort de la boucle, cela signifie que, à partir du sommet de départ donné en paramètre, on n'a pas pu trouver de chemin allant jusqu'au sommet désiré. On retire donc le sommet de départ de la liste, et on retourne None.

Implémentez et testez (sur **graphe_oriente_tp** puis **graphe_non_oriente_tp**) la méthode **"trouver_chemin_recursif(self, sommet_depart, sommet_destination, chemin)"** de la classe **GrapheOriente** qui, à partir d'un sommet de départ, un sommet de destination et d'une liste "chemin" (qui est modifiée lors des appels récursifs) **renvoie une liste** représentant le parcours à effectuer (sommet par sommet) pour aller du sommet de départ jusqu'à destination.

On se donne une deuxième fonction "trouver_chemin" qui permettra d'initialiser le paramètre "chemin" avec une liste vide, et ne pas à avoir à utiliser directement la fonction récursive.

```
def trouver_chemin_recursif(self, sommet_depart, sommet_destination, chemin):
    ......
    ......

def trouver_chemin(self, sommet_depart, sommet_destination):
    return trouver_chemin_recursif(sommet_depart, sommet_destination, [])

#Tests
>>> graphe_oriente_tp.trouver_chemin("A", "E")
['A', 'B', 'F', 'C', 'D', 'E']
>>> graphe_oriente_tp.trouver_chemin("G", "E")
['G', 'C', 'D', 'E']
>>> graphe_oriente_tp.trouver_chemin("E", "A") #Ne renvoie rien (None)
>>> graphe_non_oriente_tp.trouver_chemin("A", "I")
['A', 'B', 'C', 'E', 'I']
>>> graphe_non_oriente_tp.trouver_chemin("D", "G")
['D', 'B', 'A', 'F', 'G']
```

Exercice 13 : Détection de cycles

Dans un graphe, un cycle est un chemin qui commence et termine par le même sommet (plus précisément, on utilise le terme "cycle" pour les graphes non orientés, et "circuit" pour les graphes orientés).

Pour détecter un cycle, dans un graphe, à partir d'un sommet de départ, le but va être de faire une sorte de parcours en profondeur du graphe à partir de ce sommet en mémorisant les sommets rencontrés. Si jamais on rencontre de nouveau un sommet déjà rencontré, on a donc la présence d'un cycle.

La méthode est la suivante :

1. On initialise une **pile** et on empile le sommet de départ.
2. On initialise un **dictionnaire** qui stockera les sommets rencontrés (comme dans les parcours en profondeur et en largeur).
3. Tant que la pile n'est pas vide :
   (a) On dépile un sommet, depuis la pile (et on le stocke dans une variable) qui constituera donc **le sommet courant**.
   (b) Si le sommet a déjà été rencontré (on regarde avec le dictionnaire), cela veut donc dire qu'un cycle est présent, on retourne **True**.
   (c) Sinon, on mémorise, avec le dictionnaire, qu'on a rencontré le sommet.
   (d) Enfin, pour chaque voisin du sommet courant (dépilé), si le voisin n'a pas déjà été rencontré, on l'empile.
4. Quand on sort de la boucle, cela signifie qu'on a parcouru le graphe depuis le sommet de départ sans détecter de cycle, on retourne donc **False**.

1. Implémentez et testez (sur **graphe_oriente_tp** puis **graphe_non_oriente_tp**) la méthode **"detecte_cycle(self, sommet)"** de la classe **GrapheOriente** qui, à partir d'un sommet, retourne **True** si un cycle est détecté dans le graphe en partant de ce point et **False** si cela n'est pas le cas.

```
def detecte_cycle(self, sommet):
    ......
    ......

#Tests
>>> graphe_oriente_tp.detecte_cycle("A")
False
>>> graphe_oriente_tp.detecte_cycle("B")
False
>>> graphe_non_oriente_tp.detecte_cycle("A")
True
>>> graphe_non_oriente_tp.detecte_cycle("B")
True
```

Néanmoins, cette fonction à elle seule ne permet pas de déterminer si le graphe entier ne contient pas de cycle, notamment si tous les points du graphe ne sont pas reliées, ou bien pour un graphe orienté, selon la disposition des arcs. Il faut donc exécuter l'algorithme à partir de tous les sommets du graphe pour pouvoir conclure.

2. Implémentez et testez la méthode **"contient_cycle(self)"** de la classe **GrapheOriente** qui, en vérifiant pour chaque sommet, à l'aide de la méthode **"detecte_cycle(self, sommet")** renvoie **True** si le graphe contient un cycle et **False** si cela n'est pas le cas.

```
def contient_cycle(self):
    ......
    ......
```

```
#Tests
>>> graphe_test = GrapheOriente()
>>> graphe_test.ajouter_arc("A", "B")
>>> graphe_test.ajouter_arc("B", "C")
>>> graphe_test.ajouter_arc("A", "C")
>>> graphe_test.ajouter_arc("C", "D")
>>> graphe_test.ajouter_arc("A", "E")
>>> graphe_test.ajouter_arc("E", "B")
>>> graphe_test.ajouter_arc("B", "A")
>>> graphe_test.detecte_cycle("C")
False
>>> graphe_test.detecte_cycle("A")
True
>>> graphe_test.contient_cycle()
True

>>> graphe_test_bis = GrapheNonOriente()
>>> graphe_test_bis.ajouter_arc("A", "B")
>>> graphe_test_bis.ajouter_arc("C", "D")
>>> graphe_test_bis.ajouter_arc("C", "E")
>>> graphe_test_bis.ajouter_arc("D", "E")
>>> graphe_test_bis.detecte_cycle("A")
False
>>> graphe_test_bis.detecte_cycle("C")
True
>>> graphe_test_bis.contient_cycle()
True
>>> graphe_test_bis.ajouter_arc("A", "C")
>>> graphe_test_bis.detecte_cycle("A")
True
>>> graphe_test_bis.contient_cycle()
True
```

Exercice 14 : Conversion en matrice d'adjacence **

Nous aimerions maintenant disposer d'une méthode permettant d'obtenir la **matrice d'adjacence** à partir du graphe. En python, la matrice sera représentée par une **liste de liste**, c'est-à-dire, une liste où chaque élément de la liste est une liste (qui contiendra des valeurs de la matrice). Chaque liste contenue dans la liste principale représente **une ligne** de la matrice, et chaque élément des sous-listes représentent **les colonnes**.

Pour un numéro de ligne et de colonne donnés, la matrice contiendra **1** si il existe un arc allant du sommet correspondant à la ligne vers la sommet correspondant à la colonne. Si cet arc n'existe pas, la matrice contiendra 0 à cet emplacement.

Par exemple, si on prend la matrice suivante :

```
matrice = [[1,0,1,0],
           [0,1,0,1],
           [1,1,0,1],
           [0,1,1,0]]
```

On a donc un graphe contenant 4 sommets. Si on prend $matrice[0, 2]$, on obtient **1**. cela signifie donc qu'il existe un arc entre le sommet 0 et le sommet 2.

Implémentez et testez (sur **graphe_oriente_tp** puis **graphe_non_oriente_tp**) la méthode **"conversion_vers_matrice(self)"** de la classe **GrapheOriente** qui renvoie une **matrice** (liste de liste) correspondant à la **matrice d'adjacence** du graphe.

```
def conversion_vers_matrice(self):
    ......
    ......

#Tests
>>> graphe_oriente_tp.conversion_vers_matrice()

[[0, 1, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 1, 1, 0],
 [0, 0, 0, 1, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 1, 0, 0, 0, 0, 0],
 [0, 0, 1, 1, 0, 0, 0, 1],
 [0, 0, 0, 0, 0, 0, 1, 0]]

>>> graphe_non_oriente_tp.conversion_vers_matrice()

[[0, 1, 0, 0, 0, 1, 0, 0, 0],
 [1, 0, 1, 1, 0, 0, 1, 0, 0],
 [0, 1, 0, 0, 1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0, 0, 0, 0, 1],
 [0, 0, 1, 0, 0, 0, 0, 0, 1],
 [1, 0, 0, 0, 0, 0, 1, 1, 0],
 [0, 1, 0, 0, 0, 1, 0, 0, 1],
 [0, 0, 0, 0, 0, 1, 0, 0, 1],
 [0, 0, 0, 1, 1, 0, 1, 1, 0]]
```