SDD_Graphes_Programmation

I/ Représentation d'un graphe en Python

1/ Matrice d'adjacence

Dans cette représentation, les sommets du graphe sont supposés être des entiers, notés de 0 à N-1 où N est son nombre de sommets.
On peut donc le traduire par une matrice carrée d'ordre N de **booléens** :

- **True** pour la présence d'un voisin,
- **False** pour l'absence de voisin.

Par défaut, on l'initialise à **False** puis, pour chaque chemin entre deux sommets, on modifie la matrice par une simple affectation à **True** aux coordonnées (s,t).

Un exemple d'initialisation ici :

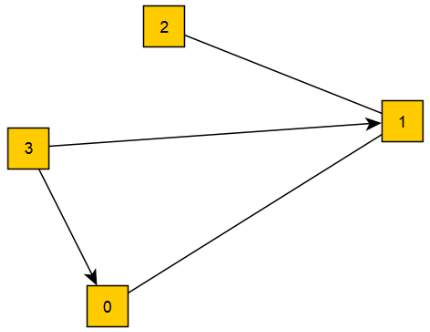
```python
# Initialisation de la matrice d'adjacence
ordre_matrice = 10   # Matrice carrée de 10 X 10

# Tout est initialisé à `False` par défaut
adj = [[False for i in range(ordre_matrice)] for j in range(ordre_matrice)]

# Création d'un chemin du sommt 0 au sommet 2
adj[0][2] = True
```

On peut écrire maintenant une **classe Graphe** :

```python
class Graphe :
    def __init__(self, ordre) :
        self.ordre = ordre
        self.adj = [[False for i in range(ordre_matrice)] for j in range(ordre_matrice)]

    def ajouter_arc(self, s1, s2) :
        self.adj[s1][s2] = True

    def est_arc(self, s1, s2) :
        return self.adj[s1][s2]

    # Renvoie les voisins d'un sommet
    def voisins(self, s) :
        voisins = []
        for i in range(self.ordre) :
            if self.adj[s][i] :
                voisins.append(i)

        return voisins
```

Voici un exemple de création de graphe :

```python
# Création du graphe
graph = Graphe(4)
graph.ajouter_arc(0,1)
graph.ajouter_arc(1,0)
graph.ajouter_arc(1,2)
graph.ajouter_arc(2,1)
graph.ajouter_arc(3,0)
graph.ajouter_arc(3,1)
```



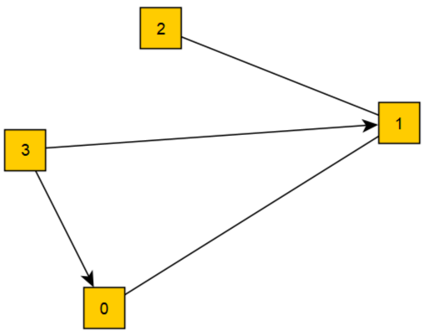
Remarque : on peut introduire un **système d'entiers** si le graphe est pondéré et initialiser la matrice par -1 par exemple si les poids sont positifs.

La **matrice d'adjacence** est une méthode simple pour implémenter mais est gourmande en mémoire : en effet, même si le graphe possède peu d'arcs, il faut le parcourir une ligne en entier pour déterminer les voisins d'un sommet. De plus, on ne peut qu'utiliser des nombres pour les noms des sommets.
Un graphe de 1000 sommets induira une matrice d'un million d'éléments par exemple.

2/ Dictionnaire d'adjacence

Dans cette représentation, chaque **nœud** est une **clé** d'un dictionnaire et **ses voisins sa valeur** associée, sous forme d'une **liste de nœuds**.

```python
# Représentation du graphe sous forme de dictionnaire
dic_adj = {"0":["1"],"1":["0", "2"],"2":["1"],"3":["0", "1"]}
```



On peut écrire maintenant une **classe Graphe** :

```python
class Graphe :
    def __init__(self, ordre) :
        self.adj = {}

    def ajouter_sommet(self, s) :
        # Vérification qu'il s'agit d'un nouveau sommet
        if s not in self.adj :
            self.adj[s] = []

    def ajouter_arc(self, s1, s2) :
        # Ajout des sommets (si nouveaux)
        self.ajouter_sommet(s1)
        self.ajouter_sommet(s2)
        # Ajout du voisin s2 à s1
        self.adj[s1].append(s2)
```

```python
    def arc(self, s1, s2) :
        return s2 in self.adj[s1]

    # Renvoie tous les sommets
    def sommets(self) :
        return self.adj.keys()

    # Renvoie les voisins d'un sommet
    def voisins(self, s) :
        return self.adj[s]
```

```python
# Création du graphe
graphe = Graphe(4)
graphe.ajouter_arc(0,1)
graphe.ajouter_arc(1,0)
graphe.ajouter_arc(1,2)
graphe.ajouter_arc(2,1)
graphe.ajouter_arc(3,0)
graphe.ajouter_arc(3,1)

# Jeu de tests
print(graphe.sommets())   # Attendu : 0,1,2,3
print(graphe.voisins(0))  # Attendu : 1
```

Le **dictionnaire d'adjacence** est une méthode efficace pour représenter des graphes. En effet, les insertions sont d'une complexité constante et celle de la détermination des voisins est égale à leur nombre (on n'est pas obligé de parcourir tous les nœuds. Il n'y a par ailleurs plus la limitation des nombres entiers pour caractériser les nœuds.
On préfèrera toutefois utiliser la matrice d'adjacence dans les cas où le graphe est pratiquement complet (place en mémoire).

II/ Parcours d'un graphe

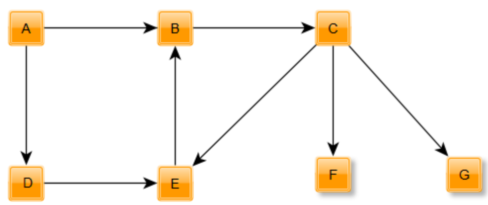
1/ Parcours en profondeur

Le **parcours en profondeur** permet de lister tous les chemins possibles à partir d'un sommet, ceux qui sont traversés sont « marqués », ce qui évite les cycles notamment.
Lorsqu'un chemin a été trouvé (ou déjà vu), on « remonte » au sommet précédent et ainsi de suite.

Voici un exemple, on partira du sommet A :

Sommets vus	*Actions menées*
A	A marqué, on va vers B (on aurait pu aller à D),
AB	B marqué, on va vers C,
ABC	C marqué, on va vers E,
ABCE	E marqué, on va vers B,
ABCEB	B est déjà marqué, chemin fini. On remonte à E,
ABCE	pas d'autres chemins, on remonte à C,
ABC	on va vers F,
ABCF	F marqué, chemin fini. On remonte à C,
ABC	On vers G,
ABCG	G marqué, chemin fini. On remonte à C,
ABC	pas d'autres chemins, on remonte à B,
AB	pas d'autres chemins, on remonte à A,
A	on va vers D,
AD	D marqué, on va vers E,
ADE	E déjà vu, chemin fini. On remonte à D,
AD	pas d'autres chemins, on remonte à A,
A	pas d'autres chemins, on remonte à A
.	pas d'autres chemin, parcours terminé.

Graphe à étudier



Le parcours en profondeur permet de déterminer **l'existence d'un chemin d'un sommet à un autre** : en effet, tous les sommets du graphe ne sont pas forcément atteignables à partir d'un sommet (certains graphes orientés par exemple).

Au niveau de la **programmation**, le principe est celui-ci : « Si un sommet n'est pas marqué, on le marque et on parcourt tous ses voisins de la même façon, sinon, on passe »

Voici un exemple :

```
# Parcours en profondeur du graphe
def parcours_profondeur(graphe, sommets_vus, sommet) :
    # Cas général : nouveau sommet trouvé
    if sommet not in sommets_vus :
        # Sommet vu désormais
        sommets_vus.append(sommet)
        print(sommets_vus) # Affiche les sommets vus en temps réel
        # Parcours des voisins
        for voisin in graphe.voisins(sommet) :
            parcours_profondeur(graphe, sommets_vus, voisin)
```

```
# Création du graphe
graphe = Graphe(4)
graphe.ajouter_arc(0,1)
graphe.ajouter_arc(1,0)
graphe.ajouter_arc(1,2)
graphe.ajouter_arc(2,1)
graphe.ajouter_arc(3,0)
graphe.ajouter_arc(3,1)

# Jeu de test
sommets_vus = []
# Attendu : 0,1,2
parcours_profondeur(graphe,sommets_vus,0)
```

<u>Remarques</u> :

- le **cas d'arrêt** est inclus dans l'instruction conditionnelle : le sommet ne doit pas être vu, sinon, l'appel récursif ne s'exécute pas.
- On est certain que l'algorithme s'arrête : au bout d'un moment, tous les sommets susceptibles d'être visités le seront.

Voici un autre exemple, à partir d'une **pile** : le fait de « remonter » au sommet précédent lorsque cela est nécessaire revient à dépiler une pile.

```
# Parcours en profondeur du graphe
def parcours_profondeur_pile(graphe, sommets_vus, sommet)
    # Initilisation de la pile
    pile = Pile()
    pile.empiler(sommet)

    # Boucle classique avec les piles
    while not pile.est_vide() :
        sommet = pile.depiler()
        # On passe au sommet précédent si déjà marqué
        if sommet in sommets_vus :
            continue

        # Sommet désormais marqué
        sommets_vus.append(sommet)

        # On empile les voisins
        for voisin in graphe.voisins(sommet) :
            pile.empiler(voisin)

    return sommets_vus
```

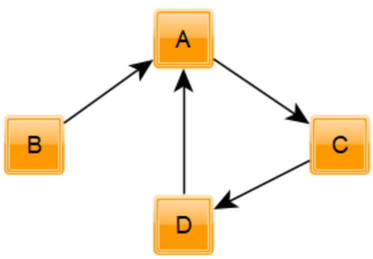
```
# Création du graphe
graphe = Graphe(4)
graphe.ajouter_arc(0,1)
graphe.ajouter_arc(1,0)
graphe.ajouter_arc(1,2)
graphe.ajouter_arc(2,1)
graphe.ajouter_arc(3,0)
graphe.ajouter_arc(3,1)

# Jeu de test
sommets_vus = []
# Attendu : 0,1,2
print(parcours_profondeur_pile(graphe,sommets_vus,0))
```

<u>2/ Détection de cycles</u>

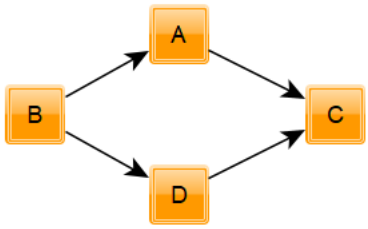
Le **parcours en profondeur** permet découvrir des **cycles** par le marquage des sommets visités. Cependant, cette condition n'est pas suffisante et un chemin peut être stoppé sans pour autant révéler la présence d'un cycle.

<u>Exemple 1</u> : Cas d'un cycle

En partant du sommet B (marqué), on va en A (marqué), puis en C (marqué) puis en D (marqué) et en A : comme ce sommet à déjà été marqué, le chemin s'arrête : il y a bien un cycle.



<u>Exemple 2</u> : Cas d'un non cycle



En partant du sommet B (marqué), on va en A (marqué) puis en C (marqué).
On revient ensuite en A (pas d'autres voisins) puis en B.
On part en D (marqué) puis en C mais il a déjà été marqué : il n'y a pourtant pas de cycles.

Pour y remédier, on va ajouter un **troisième type** au duo marqué / non marqué, à savoir s'il le sommet marqué indique **la fin d'un chemin ou pas**.
On aurait ainsi les trois états suivants :

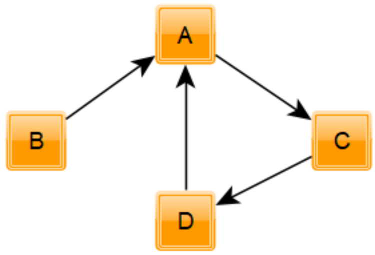
- <u>Etat 0</u> : sommet non marqué.
- <u>Etat 1</u> : sommet marqué mais chemin non terminé (cycle ?).
- <u>Etat 2</u> : sommet marqué en fin de chemin.

Au départ, tous les sommets sont à l'état 0.

Que se passe-t-il pour les deux exemples ci-dessus avec ce système ?

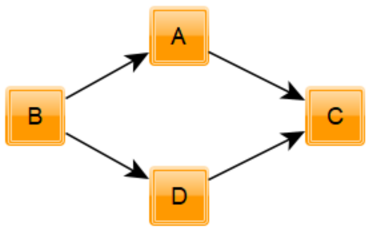
<u>Exemple 1</u> : Cas d'un cycle

Etat initial	A, B, C, D : Etat 0
Sommet de départ, B	B : Etat 1 ; A, C, D : Etat 0
Vers A	A, B : Etat 1 ; C, D : Etat 0
Vers C	A, B, C : Etat 1 ; D : Etat 0
Vers D	A, B, C, D : Etat 1
Vers A	A est à l'état 1 : **cycle**



<u>Exemple 2</u> : Cas d'un non cycle

Etat initial	A, B, C, D : Etat 0
Sommet de départ, B	B : Etat 1 ; A, C, D : Etat 0
Vers A	A, B : Etat 1 ; C, D : Etat 0
Vers C	A, B, C : Etat 1 ; D : Etat 0
Chemin terminé, retour en A	C : Etat 2 ; A, B : Etat 1 ; D : Etat 0
Chemin terminé, retour en B	A, C : Etat 2 ; B : Etat 1 ; D : Etat 0
Vers D	A, C : Etat 2 ; B, D : Etat 1
Vers C, déjà marqué d'un chemin fini	A, C : Etat 2 ; B, D : Etat ; 1 ; **pas de cycle**
Retour en D, pas d'autres chemins	A, C, D : Etat 2 ; B : Etat 1
Retour en B, pas d'autres chemins	A, B, C, D : Etat 2



Exemple de programme :

```python
# Etats des sommets
NON_MARQUE, MARQUE, MARQUE_FINI = 0, 1, 2


def detection_cycle(graphe, etats, sommet) :
    # Parcours en profondeur

    # Cas d'arrêt (sommet visité)
    if etats[sommet] == 1 : # Cycle
        return True
    elif etats[sommet] == 2 : # Chemin fini
        return False

    # Cas général (nouveau sommet)
    else :
        etats[sommet] = 1 # Sommet visité
        for voisin in graphe.voisins(sommet) :
            # S'il y a un cycle
            if detection_cycle(graphe, etats, voisin)
                return True

        etats[sommet] = 2 # Chemin fini
        return False      # Pas de cycle


def test_cycle(graphe) :
    etats = {}
    # Initialisation des sommets à `non marqué`
    for sommet in graphe.sommets() :
        etats[sommet] = 0
    # Recherche de cycle à partir de tous les sommets
    for sommet in graphe.sommets() :
        if detection_cycle(graphe, etats, sommet) :
            return True

    return False
```

```python
# Jeu de test
graphe_1 = { "B" : ["A"], "A" : ["C"], "C" : ["D"], "D" : ["A"]}
exemple_1 = Graphe(4)
exemple_1.ajouter_arc("B","A")
exemple_1.ajouter_arc("A","C")
exemple_1.ajouter_arc("C","D")
exemple_1.ajouter_arc("D","A")

print(test_cycle(exemple_1)) # Attendu : True


# Attendu : False
graphe_2 = { "B" : ["A", "D"], "A" : ["C"], "C" : [], "D" : ["C"]}
exemple_2 = Graphe(4)
exemple_2.ajouter_arc("B","A")
exemple_2.ajouter_arc("B","D")
exemple_2.ajouter_arc("A","C")
exemple_2.ajouter_arc("D","C")

print(test_cycle(exemple_2)) # Attendu : False
```

```
True
False
```