TP N°04 : REPRESENTATIONS DE GRAPHES

OBJECTIFS DU TP

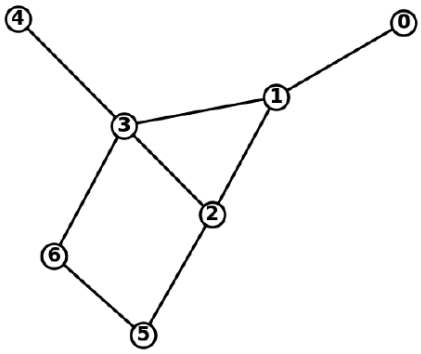
- Représenter un graphe à l'aide d'une liste et d'une matrice d'adjacence.
- Déterminer de degré, la taille d'un graphe ainsi que l'ordre entrant et sortant des sommets.

```
# Importation des bibliothèques
import numpy as np
import random as rd
import copy
```

I. REPRESENTATION DE GRAPHES EN UTILISANT LES LISTES D'ADJACENCE

I.1. Quelques manipulations classiques autour des graphes

On considère le graphe G(S,A) ci-dessous :



Q1. Ecrire la liste d'adjacence L du graphe G(S,A).

```
L = [[1],[0,2,3],[1,3,5],[1,2,4,6],[3],[2,6],[3,5]]

# pour visualiser le graphe :
Lg=Graphe(liste=L,oriente=False)
Lg.afficher()
```

Q2. Ecrire un fonction `ajout_sommet` qui prend en argument une liste L correspondant à la liste d'adjacence d'un graphe G et ajoutant un sommet à ce graphe. La fonction retourne la nouvelle liste d'adjacence du graphe.

```
def ajout_sommet (L:list)-> list:
    """
    ajoute un sommet au graphe G de liste d'adjacence L
    renvoie la nouvelle liste d'adjacence
    """
    L_copie = copy.deepcopy(L)
    L_copie=L_copie+[[]]
    return(L_copie)
```

Q3. Ecrire une fonction `ajout_arete` qui prend en argument une liste L correspondant à la liste d'adjacence d'un graphe G et deux entiers i et j représentant deux sommets du graphe G, permettant de créer, si besoin, une arête entre deux sommets. La fonction renvoie la nouvelle liste d'adjacence.

```
def ajout_arete (L:[list],i:int,j:int)->[list]:
    """
    ajoute une arete entre le sommet i et le sommet j du graphe G de liste d'adjacence L
    renvoie la liste d'adjacence associée
    """
    L[i].append(j)
    L[j].append(i)
    #la liste L est modifiée par effet de bords
```

Q4. Ecrire une fonction `ordre` prenant en argument une liste L correspondant à la matrice d'adjacence d'un graphe G et retournant un entier correspondant à l'ordre du graphe G.

```
def ordre (L:[list])-> int:
    """
    renvoie l'ordre du graphe décrit par la liste d'adjacence L, c'est à dire le nombre de sommets du graphes
    """
    return len(L)
```

Q5. Ecrire une fonction `degré` prenant en argument une liste L correspondant à la liste d'adjacence d'un graphe G et un entier i correspondant à un sommet du graphe et retournant un entier d correspondant au degré du sommet i dans le graphe G.

```
def degre (L:[list],i:int)-> int :
    """
    Renvoie le degré du sommet i du graphe décrit par la liste d'adjacence L
    """
    return len(L[i])
```

Q6. Ecrire une fonction `sont_voisins` prenant en argument une liste L correspondant à la liste d'adjacence d'un graphe G et deux entiers i et j représentant deux sommets du graphe non-orienté G et retournant le booléen `True` si les deux sommets sont adjacents, et `False` sinon.

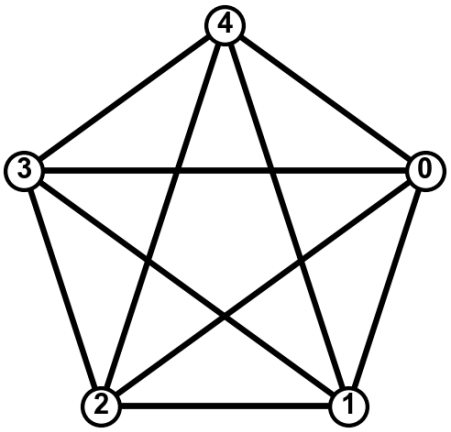
```
def sont_voisins(L:[list],i:int,j:int)-> bool :
    """
    renvoie True si les sommets i et j sont adjacents dans le graphe non-orienté décrit par la liste d'adjacence L
    """
    test=False
    if j in L[i]:
        test=True
    return(test)
```

Autre possibilité : `return( j in L[i] )`

Q7. Ecrire la matrice d'adjacence M du graphe G(S,A). Comparer la taille de l'espace mémoire pour stocker un graphe sous forme d'une liste d'adjacence ou sous forme d'une matrice d'adjacence.

```
M=[[0 1 0 0 0 0 0]
[1 0 1 1 0 0 0]
[0 1 0 1 0 1 0]
[0 1 1 0 1 0 1]
[0 0 0 1 0 0 0]
[0 0 1 0 0 0 1]
[0 0 0 1 0 1 0]]
```

Stocker sous forme d'une matrice d'adjacence ici nécessite une mémoire de 7^2 soit une taille n^2 quel que soit le graphe. Pour stocker le graphe G sous forme d'une liste d'adjacence, la mémoire est de 16 soit forcément inférieure ou égale à n^2.



Q8. Représenter la matrice et la liste d'adjacence du graphe H(S',A') ci-dessus. Conclure sur l'avantage de la représentation à partir d'une matrice d'adjacence. Dans quels cas cette utilisation est-elle pertinente ?

```
# Matrice d'adjacence
Hm  = [[0, 1, 1, 1, 1],
       [1, 0, 1, 1, 1],
       [1, 1, 0, 1, 1],
       [1, 1, 1, 0, 1],
       [1, 1, 1, 1, 0]]

# Liste d'adjacence
Hl = = [[1,2,3,4],[0,2,3,4],[0,1,3,4],[0,1,2,4],[0,1,2,3]]
```

La complexité spatiale pour le stocker Hm est $O(n^2)$, la complexité pour stocker Hl est n*(n-1) soit également $O(n^2)$, ici les deux représentations sont équivalentes, les graphes complets sont des graphes denses.

En fonction du problème rencontré, il peut donc être plus pertinent de manipuler une liste d'adjacence plutôt qu'une matrice et inversement.

Q9. Ecrire une fonction `matrice_vers_liste` qui prend en argument une matrice M correspondant à la matrice d'adjacence d'un graphe G et renvoyant la liste d'adjacence du graphe G correspondant.

```
def matrice_vers_liste (M:[list])-> [list]:
    """
    Convertit la matrice d'adjacence d'un graphe en liste d'adjacence
    """
    n=len(M)
    L=[]
    for i in range (n):
        ligne = []
        for j in range (n):
            if M[i][j]==1:
                ligne.append(j)
        L.append(ligne)
    return (L)
```

Q10. Ecrire une fonction `liste_vers_matrice` qui prend en argument une liste L correspondant à la liste d'adjacence d'un graphe G et renvoyant la matrice d'adjacence du graphe G correspondant.

```
def liste_vers_matrice(L:[list])->[list]:
    """
    Convertit la liste d'adjacence d'un graphe en matrice d'adjacence
    """
    n=len(L)
    M=[[0 for _ in range(n)]for j in range(n)]
    for i in range (n):
        for j in L[i]:
            M[i][j]=1
    return(M)
```

II. UTILISATION DE GRAPHES TOURNOIS

Dans cette partie, on utilisera la représentation des graphes sous forme de matrices d'adjacence.

II.1. Jeu d'enfants

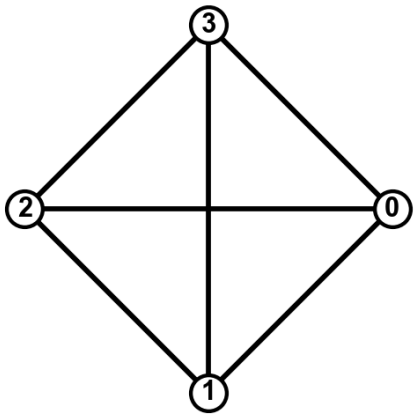
On s'intéresse à un jeu nommé J pour lequel les parties se jouent entre deux joueurs ; pour chaque partie du jeu J, il y a un gagnant et un perdant, il n'y a pas de match nul.

On considère une compétition du jeu J effectuée par n joueurs. Chaque joueur joue une seule fois au jeu J contre chaque autre joueur. Chaque joueur est représenté par un sommet d'un graphe non-orienté T, une arête de ce graphe entre les sommets s_i et s_j correspond à une partie du jeu J entre les joueurs i et j.
Par convention, avant le tournois, T_{ij} =1 $\forall(i,j) \in [\![0, n-1]\!]$ *tel que* $i \neq j$. Un joueur ne pouvant pas jouer contre lui-même, T_{ii} vaut 0.

Q11. Représenter le graphe T d'un tournoi à 4 joueurs. Ecrire la matrice d'adjacence T d'un tournois à 4 joueurs.

On note T la matrice du Graphe T

```
T = [ [0,1,1,1],
      [1,0,1,1],
      [1,1,0,1],
      [1,1,1,0]]
```



Q12. Ecrire une fonction `est_complet` prenant comme arguments la matrice d'adjacence G associée au graphe (non-orienté) G(S,A), et retournant le booléen `True` si le graphe G est complet, `False` sinon.

```
def est_complet(T:list)->bool :
    """
    renvoie True si le graphe est complet et False sinon
    """
    for i in range (len(T)):
        for j in range (i+1,len(T)):
            if T[i][j]==0:
                    return(False)
    return(True)
```

Q13. Ecrire une fonction `construction` prenant en argument un entier n correspondant à un nombre de joueurs et renvoyant la matrice d'adjacence T du graphe T(S,A) décrivant le tournoi.

```
def construction (n):
    """
    renvoie une matrice d'adjacence d'un graphe complet d'ordre n
    """
    G=[[1 for i in range (n)] for _ in range (n)]
    for i in range (n):
        G[i][i]=0
    return(G)
```

Q14. Proposer une suite d'instruction pour construire la matrice d'adjacence T du graphe d'un tournoi à 4 joueurs.

```
T = construction(4)
>>> [ [0,1,1,1],[1,0,1,1],[1,1,0,1],[1,1,1,0]]
```

Lors de l'organisation du tournois, il peut être nécessaire d'ajouter des joueurs au jeu au dernier moment. Cela revient à ajouter un sommet au graphe T.

Q15. Ecrire une fonction `ajout_joueur` prenant en argument une matrice T représentant la matrice d'adjacence d'un graphe T(S,A) et ajoutant un sommet au graphe T et en ajoutant les parties entre les autres joueurs et celui-ci.

```
def ajout_joueur (T:[list])->list :
    """
    renvoie la matrice d'ajacence d'un graphe complet possédant un sommet de
plus que le graphe T en entrée
    """
    M=copy.deepcopy(T)
    n = len(M)
    M=M+[[]]
    for i in range (n):
        M[i].append(1)
        M[n].append(1)
    M[n].append(0)
    return(M)
```

Q16. Proposer une suite d'instruction pour ajouter un 5ème joueur au tournoi précédent.

```
ajout_joueur(T)
>>> [[0, 1, 1, 1, 1], [1, 0, 1, 1, 1],[1, 1, 0, 1, 1],[1, 1, 1, 0, 1],
[1, 1, 1, 1, 0]])
```

Q17. Ecrire une fonction `partie` qui prend en argument deux entiers i et j correspondant à deux joueurs du tournois et renvoyant aléatoirement l'entier i ou j correspondant au vainqueur de la partie. On pourra utiliser la fonction `choice` de la bibliothèque `random`.

```
def partie (i,j):
    """ défini aléatoirement un vainqueur entre les joueurs i et j """
    return(rd.choice([i,j]))
```

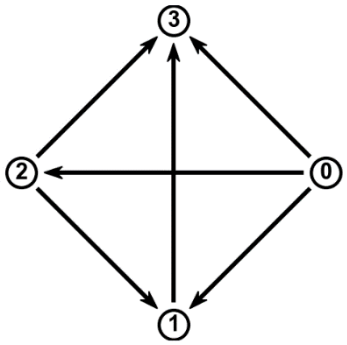
Q18. Ecrire une fonction `tournoi` qui prend comme argument la matrice d'adjacence T associées au graphe (non-orienté) T(S,A) et qui retourne la matrice d'adjacence R du graphe orienté R(S,A) représentant les résultats du tournois avec la convention suivante : l'arc est orienté du gagnant vers le perdant.

```
def tournoi (T:[list])->[list]:
    """ renvoie une matrice d'adjacence correspondant au graphe orienté donnant
les résultats du tournoi, l'arc est orienté du gagnant vers le perdant
    """
    res = copy.deepcopy(T)
    for i in range (len(res)):
        for j in range (i+1,len(res)):
            V=partie(i,j)
            if V==i :
                res[j][i]=0
            else :
                res[i][j]=0
    return (res)
```

Q19. Proposer une suite d'instruction pour afficher la matrice d'adjacence R des résultats du tournoi de matrice d'adjacence initiale T.

```
R =tournoi(T)
>>> [[0 1 1 1]
[0 0 0 1]
[0 1 0 1]
[0 0 0 0]]

#visualisation d'un exemple de graphe obtenu :
```



Le vainqueur du tournoi est le joueur qui a remporté le plus de parties.

Q20. Ecrire une fonction `vainqueur` prenant en argument un graphe T représentant les parties d'un tournoi et renvoyant :

- un entier correspondant au numéro du gagnant ou
- une liste avec le numéro des gagnants en cas d'égalité

```
def vainqueur(R:[list])->int or [int] :
    """
    Renvoie un entier correspondant au numéro du vainqueur du tournoi à partir
de la matrice d'adjacence des résultats du tournoi
    Renvoie la liste des numéros des gagnants en cas d'égalité

    """
    s_max = 0
    egalite=False
    winner = 0
    for i in range (len(R)):
        score=0
        for j in R[i]:
            score = score + j
```

```
        if score == s_max and egalite==False :
            egalite=True
            Lwin=[winner,i]

        elif score == s_max and egalite==True :
            Lwin.append(i)

        elif score > s_max:
            egalite =False
            winner = i
            s_max=score

    if egalite == False :
        return(winner,s_max)

    return(Lwin,s_max)
```

Q21. Déterminer le vainqueur du tournoi.

```
vainqueur(R)
```

II.2. Marquer des points

La plupart du temps, lors d'un match ou d'un jeu c'est le nombre de points marqués qui permet de déterminer le gagnant, il est donc nécessaire de **pondérer** les graphes avec des scores, le cas de match nul ou d'égalité devient alors possible.

Q22. Ecrire une fonction `score_parties` prenant en argument une matrice T correspondant à la matrice d'adjacence d'un graphe tournoi et renvoyant une matrice V telle que le terme V[i][j] correspondent au nombre de points marqués par le joueur i lors du match contre j.

```
def score_parties (T:[list])->[list]:
    """
    renvoie la matrice d'adjacence du graphe pondéré par les scores marqués par
chaque joueur à chaque partie du tournoi
    """
    n=len(T)
    V=copy.deepcopy(T)
    for i in range(n):
        for j in range(n):
            if i!=j :
                V[i][j]= rd.randint(0,50)
    return(V)
```

On utilisera `randint` de la bibliothèque `random`. Les scores pourront être fixés entre 0 et 50.

Afin de classer les différents joueurs (sommets), on attribue un score a(s_i) (selon la définition de Copeland) à chaque sommet $s_i \in S$, défini de la manière suivante :

$$a(s_i) = \sum_{s_j \in S} r(s_i, s_j)$$

Avec

$$r(s_i, s_j) = \begin{cases} +1 \text{ si le score du joueur } i \text{ est supérieur à celui de } j \\ -1 \text{ si le score du joueur } i \text{ est inférieur à celui du joueur } j \\ 0 \text{ sinon} \end{cases}$$

Le vainqueur du tournoi (au sens de Copeland) est le sommet s_v tel que $a(s_v) = \max\{s_i\}_{s_i \in S}$.

Q23. Ecrire une fonction `vainqueur_Cop` prenant en argument une matrice V correspondant à la matrice d'adjacence du graphe orienté pondéré V(S,A) contenant les scores des parties et renvoyant :

- un entier correspondant au numéro du gagnant au sens de Copeland ou
- une liste avec le numéro des gagnants en cas d'égalité.

```
def vainqueur_cop(V:[list])->([int],int) or (int,int):
    """
    Renvoie un entier correspondant au numéro du vainqueur du tournoi à partir
de la matrice d'adjacence des résultats du tournoi
    Renvoie la liste des numéros des gagnants en cas d'égalité

    """
    n = len(V)
    # constitution de la liste des scores selon Copeland
    r=[0 for _ in range (n)]
    for i in range (n):
        a=0
        for j in range (i+1,n):
            if V[i][j]>V[j][i]: # si i gagne le match
                r[i]=r[i]+1
                r[j]=r[j]-1
            elif V[i][j]<V[j][i]: # si j gagne le match
                r[i]=r[i]-1
                r[j]=r[j]+1
            # cas d'égalité ou de match nul, on ne fait rien

    # détermination du score maximal et du ou des gagnants

    s_max = r[0]
    winner=0
    egalite=False
    L
    for i in range (1,n):
        if r[i]==s_max and egalite==False :
            Lwin=[winner,i]
            egalite = True
        elif r[i]==s_max and egalite ==True :
            Lwin.append(i)

        elif r[i]>s_max :
            egalite=False
            winner = i
            s_max = r[i]
```

```
    if egalite == True :
        return(Lwin,s_max)
    return(winner,s_max)
```