

TD 23 – Partie 1 – Découverte des graphes

Dans ce TD, il est imposé d'utiliser la bibliothèque **Numpy** du langage python pour la gestion des matrices.

Rappels : une matrice carrée de taille N s'initialise remplie de zéros :

```
import numpy as np
M = np.zeros( [N,N] )
```

puis on parcourt la matrice et on remplace le « zéro » en position $M_{i,j}$ ($i^{\text{ème}}$ ligne, $j^{\text{ème}}$ colonne) par l'expression désirée :

```
for i in range(N) : # i num de ligne, i va de 0 à N-1
    for j in range(N) : # j num de colonne
        M[i,j] = ... # M[i,j] désigne l'élément ième ligne jème colonne
                    # à écraser le 0 actuel par la valeur voulue.
                    # Vous pouvez bien sûr mettre ici un « if » et
                    # n'écraser M[i,j] qui sous certaines conditions
```

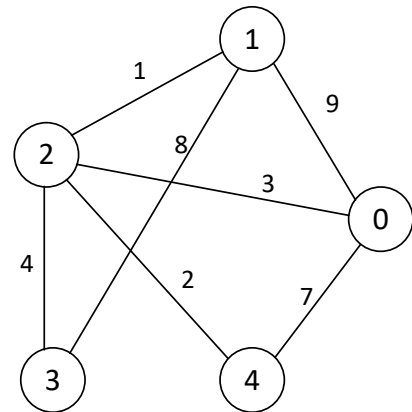
Coût d'un chemin sur un graphe pondéré non orienté

On considère le graphe $G = (S, A)$ ci-contre, où le nombre situé sur l'arête joignant deux sommets est leur distance, supposée entière et positive.

Pour *Python*, nous noterons :

```
S = [0,1,2,3,4] # l'ensemble des sommets
A = { 0:{1:9,2:3,4:7}, 1:{0:9,2:1,3:8}, 2:{0:3,1:1,3:4,4:2},
      3:{1:8,2:4}, 4 :{0:7,2:2} }
```

A est une liste d'adjacence pondérée, ici représentée sous forme de dictionnaire de dictionnaires. Le poids entre deux sommets i et j , s'il existe, s'appelle $A[i][j]$.



On prendra pour convention que la distance entre un sommet et lui-même vaut 0. Attention, vous noterez que dans le dictionnaire A , aucune distance n'a été renseignée entre un élément et lui-même. Vous n'avez pas le droit de modifier A .

Q1 – 📁 Écrire une fonction `Mat_dist (S_adj, Dico_adj)` prenant en argument une liste des sommets et un dictionnaire des adjacences (même format que S et A présentés ci-dessus, mais de taille quelconque, pas forcément 5 sommets). Et qui renvoie la matrice des distance, définie et calculée comme suit :

- Chaque élément $M_{i,j}$ de la matrice renvoyée correspond à la distance directe, si elle existe, entre les deux sommets i et j .
 - On convient que, si les sommets ne sont pas reliés par une arête, cette distance vaut -1 .
 - La distance du sommet i à lui-même est prise, par convention ici, égale à 0.
 - Vous complèterez d'abord le triangle supérieur ($j > i$) (2 boucles *for* imbriquées)
 - Puis dans un second temps, vous complèterez le triangle inférieur ($i < j$) par symétrie, car $M_{i,j} = M_{j,i}$
- Aide : `np.transpose(Mat)` est une méthode qui renvoie la transposée d'une matrice `Mat`.

Q2 – 📁 À l'aide de votre fonction, générer la matrice M , **variable globale**, liée au graphe donné dans l'énoncé. Afficher cette matrice et vérifier à la main que le résultat obtenu est cohérent.

Q3 –  Écrire une fonction `voisins(Mat, i)` prenant en argument une matrice des distances `Mat` et un numéro de sommet `i` et qui renvoie la liste des sommets voisins du sommet `i`.
Tester votre fonction pour afficher les voisins de 2 (d'une part) et de 4 (d'autre part).

Q4 –  Écrire une fonction `degre(Mat, i)` prenant en argument une matrice des distances `Mat` et un numéro de sommet `i` et qui renvoie le degré du sommet `i`, c'est-à-dire le nombre d'arêtes issues de ce sommet.
Tester votre fonction pour afficher le degré de 2 (d'une part) et de 4 (d'autre part).

Nous prendrons la convention qu'un chemin est représenté par une liste de valeurs de `S` (les sommets). Par exemple, le chemin $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 4$ sera traduit par la liste `[0, 1, 3, 2, 4]`.

Q5 –  Écrire une fonction `chem_valide(Mat, chem)` prenant en argument une matrice des distances `Mat` et un chemin décrit tel que ci-dessus, et qui renvoie un booléen : *True* s'il est possible de faire le chemin dans le graphe, *False* sinon.

Exemple : pour tester votre fonction

```
chem_valide(M, [0,1,3,2,4]) # doit renvoyer True
chem_valide(M, [0,4,3]) # doit renvoyer False car 4
                        # et 3 ne sont pas voisins.
```

Q6 –  Écrire une fonction `longueur(Mat, chem)` prenant en argument une matrice des distances `Mat` et un chemin, et qui :

- Teste si le chemin est invalide. Si c'est le cas elle lève une assertion :

```
assert ..., "erreur" # à vous de remplir la condition
# en complétant les ... (déclenche la détection d'erreur et la sortie de la fonction)
```

- Dans le cas contraire (chemin valide), elle renvoie la longueur du trajet décrit par cette liste `chem`, c'est à dire la somme des longueurs des arêtes empruntées.

Pour tester votre fonction,

```
longueur(M, [0,1,3,2,4]) # doit renvoyer 23
longueur(M, [0,4,3]) # doit générer une erreur
```