Code de partage avec Capytale : 98b4-10266000

On utilisera les bibliothèques suivantes :

```
import numpy as np
import numpy.random as rd
import numpy.linalg as al
```

1 Exemple de constructions de graphes aléatoires

Dans ce TP, on considère des graphes d'ordre $n \in \mathbb{N}^*$ (à n sommets) simples, non orientés et on suppose que l'arête $\{i, j\}$ existe avec une probabilité p_n, pour toute paire de sommets $\{i, j\} \in \{1, ...n\}^2$, $i \neq j$
On s'intéressera surtout aux liens entre les sommets (liens sociaux dans un réseau), à savoir si on peut aller d'un sommet à l'autre, s'il y a des points isolés, ou si le graphe est connexe.
On rappelle que la matrice d'adjacence M d'un tel graphe est symétrique avec une diagonale nulle, donc construire un tel graphe revient à identifier les éléments non nuls au dessus de la diagonale de M.
On considère donc une matrice M carrée d'ordre n symétrique, de diagonale nulle, dont les éléments au dessus de la diagonale $m_{i,j}$ avec $j > i$ sont des variables aléatoires indépendantes suivant la loi de Bernoulli de paramètre p_n, et on construira le graphe de matrice d'adjacence M

1. Compléter la fonction suivante renvoyant une réalisation de M, n et p_n étant donnés en entrée :

   Les gammes de valeurs des boucles `for` sont choisies afin de ne remplir que le « triangle supérieur » de la matrice, c'est-à-dire les coefficients $m_{i,j}$ avec $i < j$
   On peut alors utiliser la loi de Bernoulli pour renseigner ces coefficients.
   Puis, en dehors de la diagonale (qui restera à la valeur 0 fixée par défaut), le reste de la matrice doit ensuite nécessairement être rempli par symétrie (on ne peut réutiliser une fonction aléatoire, car alors le résultat serait potentiellement différent). De fait, on ne remplit pas la dernière ligne (elle le sera par symétrie de la dernière colonne).

   ```
   def MatAlea(n, p):
       M=np.zeros((n, n)) # valeur zéro par défaut
       for i in range(0,n-1) : # toutes les lignes sauf la dernière
           for j in range(i+1, n) : # j>i pour n'avoir que le triangle
      supérieur
                M[i, j] = rd.binomial(1, p) #  loi de Bernoulli de paramètre p
                M[j, i] = M[i, j] # symétrie
       return M
   ```

2. Compléter la fonction suivante renvoyant la liste d'adjacence d'un graphe à partir de sa matrice d'adjacence M

   Vu en début d'année, il faut parcourir toute la matrice. Plus précisément, pour chaque ligne i (i.e. chaque sommet), on renseigne la liste du sommet i en faisant figurer le sommet j si le coefficient $m_{i,j}$ vaut 1

   ```
   def ListeAdj(M):
       n,m = np.shape(M) # pour connaitre la taille de la matrice
       L=[] # on commence avec une liste vide
       for i in range(n): # pour chaque ligne=sommet
           voisins=[] # on crée une liste vide
           for j in range(0,n): # pour chaque colonne
   ```

```
            if M[i,j]==1 : # si les sommets i et j sont reliés
                voisins.append(j) # on ajoute j à la liste des sommets
    adjacents au sommet i
        L.append(voisins) # on ajoute la liste des sommets adjacents au
    sommet i à la liste totale
     return L
```

3. Compléter la fonction suivante renvoyant `True` si le graphe de matrice d'adjacence M est connexe et `False` sinon.

   On utilise le critère de connexité, à savoir qu'un graphe est connexe si et seulement si tous les coefficients de la matrice $\sum_{k=0}^{n} M^k$ sont strictement positifs (où n est l'ordre du graphe).

```
def EstConnexe(M):
    n=np.len(M[0,:]) # pour connaitre l'ordre du graphe
  A=np.eye(n) # matrice identité
    for k in range(1,n): # on crée la matrice de "connexité"
        A=A+al.matrix_power(A,k)
    for i in range(n): # puis on parcourt cette matrice
        for j in range(n):
            if A[i, j] ==0: # si un coefficient nul est rencontré
                return False # la fonction renverra "False" (c'est le
     premier return qui s'appliquera
    return True # si pas de return auparavant, la fonction renverra "True"
```

4. Compléter la fonction suivante renvoyant le nombre de points isolés d'un graphe (= nombre de sommets sans voisins) de liste d'adjacence L

   A partir d'une liste d'adjacence, un sommet isolé se manifeste par une liste vide. On parcourt donc chaque liste de la liste d'adjacence et on augmente le compteur à chaque fois que l'on rencontre une liste de longueur nulle.

```
def NbIsole(L):
    nombre=0 # compteur, initialement à zéro
    n=len(L) # ordre du graphe = nombre de listes dans la liste d'adjacence
    for i in range(n): # on parcourt la liste d'adjacence
        if len(L[i])==0: # si une liste est vide = sommet isolé
            nombre +=1 # on incrémente le compteur
    return nombre
```

5. Créer 4 graphes aléatoires d'ordre 5 avec une probabilité de connexion $p_5 = \frac{2}{5}$
   Pour chacun d'eux, compter le nombre de points isolés et dire s'ils sont connexes ou non.

   On utilise les programmes précédents et comme on souhaite réaliser 5 fois le test, on peut le faire avec une boucle `for`

```
for i in range(4):
    M=MatAlea(5, 2/5)
    L=ListeAdj(M)
    print(M, NbIsole(L),EstConnexe(M))
```

Pour aller plus loin : composantes connexes

6. On cherche à savoir quels sont les sommets qui sont reliés à un sommet donné x_0, c'est ce qu'on appelle une composante connexe.
   L'objet de cette question est de lister toutes les composantes connexes. On remarquera que deux composantes connexes ne peuvent pas avoir de sommet commun.

Principe de l'algorithme pour obtenir une composante connexe contenant un sommet x_0 :

- initialiser trois listes, l'une `S` ne contenant qu'un seul éléments `x_0` une liste autre `C` initialement vide, et enfin une liste `Visite` de longueur `n` = ordre du graphe qui ne contient que des `False` et qui indiquera si un sommet a été visité ou non.
- à chaque étape, et jusqu'à ce que la liste `S` soit vide :
  - considérer `s` le dernier élément de `S` et le supprimer de `S`
  - si `Visite[s]==False` :
    - ▷ changer la valeur de `Visite[s]` en `True`
    - ▷ ajouter `s` élément à `C`,
    - ▷ augmenter `S` de tous les voisins `v` de `s` qui n'ont pas été visités.

Les éléments dans la liste `C` seront alors la composante connexe contenant x_0.

(a) Ecrire une fonction `Connexe(x, L)` qui renvoie la composante connexe contenant `x` d'un graphe de liste d'adjacence `L`
Tester la fonction sur les 4 graphes de la question **5** pour $x = 0$

(b) Comment tester qu'un graphe est connexe à partir de la fonction `Connexe` ?

(c) Ecrire une fonction renvoyant toutes les composantes connexes d'un graphe de liste d'adjacence `L` . Tester cette fonction sur les graphes obtenus dans la question **5**.