TP10 (partie 1) : corrigé

Implémenter un graphe non orienté sous la forme d'un dictionnaire

```
import numpy as np
import random as rd

G1 = {}
G1[0] = [1,2,4]
G1[1] = [0,4]
G1[2] = [0,3,4]
G1[3] = [2,4,5]
G1[4] = [0,1,2,3,5]
G1[5] = [3,4]

def retraitSommet(G,s) :
    '''Renvoie un graphe (dictionnaire)
    correspondant au graphe G auquel on a
    retire le sommet s'''
    D = dict(G)
    del d[s]
    for sommet in d :
        voisins = d[sommet]
        if s in voisins :
            voisins.remove(s)
    return d
```

```
# >>> G2 = retraitSommet(G1,0)
# >>> G2
# {1: [4], 2: [3, 4], 3: [2, 4, 5], 4: [1,
    2, 3, 5], 5: [3, 4]}

# >>> len(G1[0])
# 3

def SommetMaxDeg(G) :
    '''Renvoie la liste des sommets de plus
    haut degre du graphe G'''
    val = 0
    for s in G :
        if len(G[s])>val :
            val = len(G[s])
    Lmax = []
    for s in G :
        if len(G[s])==val :
            Lmax.append(s)
    return Lmax

# >>> SommetMaxDeg(G1)
# [4]
```

Matrice d'adjacence d'un graphe

```
def dicToMat(G) :
    '''Renvoie la matrice d adjacence
    associee au graphe G'''
    liste = list(G.keys())
    liste.sort()
    n = len(liste)
    A = np.zeros((n,n))
    for i in range(n) :
        for j in range(n) :
            if liste[j] in G[liste[i]] :
                A[i,j]=1
    return A

# >>> MatG3 = np.array
    ([[0,1,1,0],[1,0,1,1],[1,1,0,1],[0,1,1,0]])

# >>> G3puissance3 = np.dot( np.dot(MatG3,
    MatG3) , MatG3 )
# >>> G3puissance3[0,3]
# 2
```

```
def MatToDic(A) :
    '''Renvoie un graphe dont la matrice d
    adjacence est A'''
    G = {}
    n = np.shape(A)[0]
    for i in range(n) :
        voisins = []
        for j in range(n) :
            if A[i,j]==1 :
                voisins.append(j)
        G[i] = voisins
    return G

# >>> G3 = MatToDic(MatG3)
# >>> G3
# {0: [1, 2], 1: [0, 2, 3], 2: [0, 1, 3], 3:
    [1, 2]}
```

```
Fourmi={}
Fourmi['A'] = ['B','D','F']
Fourmi['B'] = ['A','C','D','E']
Fourmi['C'] = ['B','D','E','F']
Fourmi['D'] = ['A','B','C','E']
Fourmi['E'] = ['B','C','D']
Fourmi['F'] = ['A','C']

def arrivee(n) :
    '''Renvoie le sommet ou se trouve la fourmi, partie de A, apres n deplacements'''
    position = 'A'
    for k in range(n) :
        voisins = Fourmi[position]
        position = rd.choice(voisins)
    return position

def frequence(n,nbRep,s) :
    '''Renvoie la frequence, lors de nbRep repetitions de l experience, a laquelle la
   fourmi arrive au sommet s apres n deplacements'''
    compteur = 0
    for i in range(nbRep) :
        position = arrivee(n)
        if position == s :
            compteur = compteur+1
    return compteur/nbRep

# >>> frequence(10,10000,'C')
# 0.2108

# >>> Mfourmi = dicToMat(Fourmi)

# >>> M10fourmi = np.array(Mfourmi)
# >>> for k in range(9) :
#           M10fourmi = np.dot(M10fourmi,Mfourmi)

# >>> chemin10AC = M10fourmi[0,2]
# >>> chemin10AC
# 43079.0

# >>> chemin10 = sum(M10fourmi[0,:])
# >>> chemin10
# 222337.0

# >>> chemin10AC/chemin10
# >>> 0.19375542532282075
```

La différence observée ($0,21$ versus $0,19$) s'explique par le fait que tous les chemins n'ont pas la même probabilité d'être empruntés !

Par exemple, lorsque la fourmi est au sommet B elle va sur chacun des sommets A, D, E, C avec une probabilité de $\frac{1}{4}$ alors que lorsque la fourmi est sur le sommet F elle va sur chacun des sommets A, C avec une probabilité de $\frac{1}{2}$.
Or lorsque que l'on calcule la probabilité recherchée en posant

$$p = \frac{\text{nombre de chemins de longueur 10 reliant } A \text{ à } C}{\text{nombre de chemins de longueur 10 partant de } A}$$

on fait l'hypothèse sous-jacente que tous les chemins sont équiprobables, ce qui n'est pas le cas.