Algorithmes et structures de données avancées : TD 7(corrigé)

Graphes - Matrice d'Adjacence - algorithmes sur les graphes

- Dans ce TP, nous travaillons exclusivement sur des graphes non-orientés.
- Pour connaître la taille d'une liste ou d'une liste des listes (par exemple dans une fonction ...), vous pouvez vous en servir de la fonction python `len`.
- Pour intialiser des listes avec `n` élements, vous disposez de l'instruction suivante :

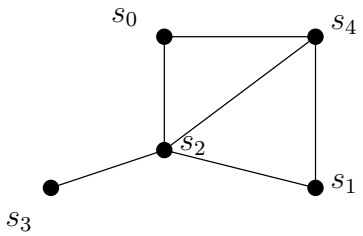
```
l = [ 0  for i in range(n) ]
```

- Pour intialiser une matrice qui est stockée dans une liste des listes A avec `lignes` lignes et `cols` colonnes vous disposez de l'instruction suivante :

```
A = [
    [ 0  for j in range(cols) ]
    for i in range(lignes)
]
```

Exercice 7.1 *Matrice d'adjacence pour un graphe non-orienté*

Dans cet exercice, vous allez élaborer pas-à-pas votre première structure de données pour les graphes ainsi que vos premières algorithmes sur les graphes.
1. Déclarer une variable `A` qui remplit la matrice d'adjacence avec le graphe **non-orienté** suivant :




```
A = [ [0,0,1,0,1],
      [0,0,1,0,1],
      [1,1,0,1,1],
      [0,0,1,0,0],
      [1,1,1,0,0] ]
```

2. Ecrire une fonction `def arete(A, s1, s2):` qui retourne `True` s'il y a une arête entre les sommets `s1` et `s2`, et `False` sinon.

```
def arete(A, s1, s2):
    if A[s1][s2] == 0:
        return True
    else:
        return False
```

3. Tester votre fonction `def arete(A, s1, s2):` en l'appelant pour quelques combinaisons de sommets.

```
print arete(A,0,2), arete(A,2,0)
print arete(A,1,3), arete(A,3,1)
```

4. Ecrire une fonction `def voisins(A, sommet):` qui affiche à l'écran les voisins du sommet numéro `sommet` du graphe représenté par la matrice d'adjacence `A` à l'écran (de la même manière que l'exemple suivant du sommet s_1) :

```
Sommmet s0 a comme voisin : s2 s4
```

```
def voisins(A, sommet):
    print "Sommet ",sommet," a comme voisin :",

    s=0
    while s<len(A):
        if A[s][sommet]==True: #ou bien
        #if arete(s, sommet)==True:
            print "s",s,
        s=s+1
    print
```

5. Tester votre fonction `def voisins(A, sommet):` en l'appelant pour chaque sommet du graphe.

```
s=0
while s<5:
    voisins(A, s)
    s=s+1
```

Exercice 7.2 *Algorithmes sur des graphes non-orientés*

1. Ecrire une fonction `def degre(A, sommet):` qui renvoie le degré du sommet numéro `sommet` du graphe représenté par la matrice d'adjacence `A`.

```
def degre(A, sommet):
    d=0
    s=0
    while s<len(A):
        if A[s][sommet]==True: #ou bien
        #if arete(s, sommet)==True:
            d=d+1
        s=s+1
    return d
```

2. Tester votre fonction en l'appelant pour chaque sommet du graphe.

```
s=0
while s<5:
    print "le degre du sommet ",s," est : ",degre(A, s)
    s=s+1
```

3. Ecrire une fonction `def degreMaximum(A):` qui renvoie le degré maximum de tous les sommets du graphe représenté par la matrice d'adjacence `A`. Utiliser la fonction `def degre(A, sommet):` que vous avez implémentée précedemment.

```
def degreMaximum(A):
    plusgrand = 0
    s=0
    while s<len(A):
        deg = degre(A, s)
        if deg>plusgrand:
            plusgrand=deg
        s=s+1
    return plusgrand
```

4. Tester votre fonction.

```
print "le plus grand degre d'un sommet du graphe est ", degreMaximum(A)
```

5. Quelle est la complexité asymptotique de la fonction `degre` ?
   complexité linéaire $O(n)$, n étant le nombre des sommets du graphe
6. Quelle est la complexité asymptotique de la fonction `degreMaximum` ? Quelle règle avez-avez vous appliqué ?
   complexité quadratique $O(n^2)$, n étant le nombre des sommets du graphe

Exercice 7.3 *Algorithmes sur des graphes non-orientés*

1. Ecrire une fonction `def nombreAretes(A):` qui renvoie le nombre d'arêtes du graphe représenté par la matrice d'adjacence `A`.
   1ère version

```
def nombreAretes(A):
    aretes = 0
    s=0
    while s<len(A):
        t=s+1
        while t<len(A):
            if A[s][t]==True: #ou bien
            #if arete(s, sommet)==True:
                aretes = aretes + 1
            t=t+1
        s=s+1
    return aretes
```

2ème version (il est facile de prouver par récurrence que le nombre d'arêtes d'un graphe est égal à $\frac{\sum_{i=1}^{n} degre(s_i)}{2}$ 1ère version

```
def nombreAretes(A):
    aretes = 0
```

```
    s=0
    while s<len(A):
        t=s+1
        while t<len(A):
            if A[s][t]==True: #ou bien
            #if arete(s, sommet)==True:
                aretes = aretes + 1
            t=t+1
        s=s+1
    return aretes
```

2. Quelle est la complexité asymptotique de votre fonction `nombreAretes` ?

La 1ère version a une complexité quadratique. La 2ème version a une complexité quadratique également, car l'appel de la fonction `degre` a une complexité linéaire en fonction du nombre des sommets du graphe : nous avons donc appliqué la régle du produit.

Exercice 7.4 *Algorithmes sur des graphes non-orientés*

1. Ecrire une fonction `rajouterArete` qui prends en paramètre un graphe représenté par une matrice d'adjacence, ainsi que deux sommets `s` et `t`, et qui permet de rajouter l'arête représenté par les deux sommets s et t au graphe.

```
def rajouterArete(A, s, t):
    A[s][t] = 1
    A[t][s] = 1
```

2. Appeler cette fonction et pour qu'elle rajoute une arête entre les sommets s_0 et s_3.

```
rajouterArete(A, 0, 3)
```

3. Vérifier que vos changement ont pris effet en appelant la fonction `voisin`.

```
s=0
while s<5:
    voisins(A, s)
    s=s+1
```

4. Quelle est la complexité asymptotique de votre fonction `rajouterArete` ?

Complexité constante $O(1)$

Exercice 7.5 *Algorithmes sur des graphes non-orientés*

1. Ecrire une fonction `def afficher(A):` qui permet d'afficher la matrice d'adjacence représenté par la matrice d'adjacence `A`.

```
def afficher(A):
    s=0
    while s<len(A):
        t=0
        while t<len(A):
            print A[s][t]," ",
            t=t+1
        print
        s=s+1
```