Corrigé : étude de réseaux sociaux (X MP-PC 2016)

Partie I. Réseaux sociaux

Question 1. Une représentation possible pour chacun des deux réseaux est :

```
reseau_A = [ 5,
             [ [0, 1], [0, 2], [0, 3], [1, 2], [2, 3] ]
           ]

reseau_B = [ 5,
             [ [0, 1], [1, 2], [1, 3], [2, 3], [2, 4], [3, 4] ]
           ]
```

Question 2.

```python
def creerReseauVide(n):
    return [n, []]
```

Question 3.

```python
def estUnLienEntre(paire, i, j):
    return paire == [i, j] or paire == [j, i]
```

Question 4.

```python
def sontAmis(reseau, i, j):
    for paire in reseau[1]:
        if estUnLienEntre(paire, i, j):
            return True
    return False
```

La fonction `estUnLienEntre` est de coût constant ; dans le pire des cas (par exemple lorsque i et j ne sont pas amis), la liste des liens d'amitié est parcourue dans son entier, donc la complexité de la fonction `sontAmis` est en $O(m)$, où m désigne le nombre de liens d'amitiés déclarés dans le réseau.

Question 5.

```python
def declareAmis(reseau, i, j):
    if not sontAmis(reseau, i, j):
        reseau[1].append([i, j])
```

la méthode append est de coût constant donc la complexité de la fonction `declareAmis` est celle de la fonction `sontAmis`, à savoir en $O(m)$.

Question 6.

```python
def listeDesAmisDe(reseau, i):
    amis = []
    for paire in reseau[1]:
        if paire[0] == i:
            amis.append(paire[1])
        elif paire[1] == i:
            amis.append(paire[0])
    return amis
```

La méthode append étant de coût constant, la complexité de la fonction `listeDesAmisDe` est proportionnelle au nombre de liens d'amitié déclarés dans le réseau, à savoir en $\Theta(m)$.

Partie II. Partitions

Question 7. La représentation filiale A correspond au tableau :

```
parent = [5, 1, 1, 3, 4, 5, 1, 5, 5, 7]
```

les représentants des quatre groupes sont 5, 4, 1 et 3.

La représentation filiale B correspond au tableau :

```
parent = [3, 9, 0, 3, 9, 4, 4, 7, 1, 9]
```

les représentants des trois groupes sont 9, 7 et 3.

Question 8.

```python
def creerPartitionEnSingletons(n):
    return [i for i in range(n)]
```

Question 9.

```python
def representant(parent, i):
    while parent[i] != i:
        i = parent[i]
    return i
```

Dans le pire des cas, cette fonction parcourt le tableau `parent` dans son entier avant de trouver le représentant du groupe auquel appartient `i`, donc sa complexité est en $O(n)$. La complexité maximale est réalisée lorsque $[\![n]\!]$ est partitionné en un seul groupe représenté par le dessin ci-dessous :

et qu'on cherche le représentant de 0. Cet exemple correspond au tableau suivant :

```
parent = [1, 2, ... , n-1, n-1]
```

Question 10.

```python
def fusion(parent, i, j):
    p = representant(parent, i)
    q = representant(parent, j)
    parent[p] = q
```

Question 11. Considérons la suite de fusions suivante :

```
fusion(parent, 0, 1)
fusion(parent, 0, 2)
fusion(parent, 0, 3)
...
fusion(parent, 0, n-1)
```

Partant de la partition en n singletons, cette succession de fusions aboutit à la partition en un seul groupe représenté question 9. D'après cette même question, la complexité $C(n)$ de cette suite de fusions vérifie la relation de récurrence $C(n) = C(n-1) + \Theta(n)$ donc $C(n) = \Theta(n^2)$; la complexité est quadratique.

Question 12. Pour effectuer la modification demandée, il est intéressant de choisir une version récursive de la fonction `representant` :

```python
def representant(parent, i):
    if parent[i] != i:
        j = representant(parent, parent[i])
        parent[i] = j
    return parent[i]
```

Notons $C(k)$ la complexité de cette fonction, où k désigne la distance qui sépare i de son représentant. Le calcul de l'entier j défini dans la fonction se réalise avec un coût égal à $C(k-1)$ donc la fonction C vérifie la relation $C(k) = C(k-1) + O(1)$, ce qui montre que $C(k) = \Theta(k)$. Cette complexité est identique à celle de la fonction `représentant` écrite à la question 10 ; on peut donc considérer que cette optimisation de la structure filiale est « gratuite ».

Question 13. La fonction qui suit utilise deux listes de même taille : `groupes` qui contient les groupes en voie de formation et `rep` qui contient les représentants des groupes déjà rencontrés.
Pour chaque élément i de $[\![n]\!]$, on calcule le représentant r de son groupe. S'il n'est pas déjà présent dans la liste `rep`, il y est ajouté et un nouveau groupe vide est créé dans la liste `groupes`. On ajoute ensuite i au groupe associé à r.

```
def listeDesGroupes(parent):
    groupes = []
    rep = []
    for i in range(len(parent)):
        r = representant(parent, i)
        k = 0
        while k < len(rep) and rep[k] != r:
            k += 1
        if k == len(rep):
            rep.append(r)
            groupes.append([])
        groupes[k].append(i)
    return groupes
```

Partie III. Algorithme randomisé pour la coupe minimum

Question 14. La fonction se contente de suivre pas-à-pas l'algorithme décrit dans l'énoncé :

```
def coupeMinimumRandomisee(reseau):
    n = reseau[0]
    parent = creerPartitionEnSingletons(n)
    nbGroupes = n
    nbLiens = len(reseau[1])
    while nbGroupes > 2 and nbLiens > 0:
        nbLiens -= 1
        k = random.randint(0, nbLiens)
        [i, j] = reseau[1][k]
        ri = representant(parent, i)
        rj = representant(parent, j)
        if ri != rj:
            fusion(parent, ri, rj)
            nbGroupes -= 1
        reseau[1][k], reseau[1][nbLiens] = reseau[1][nbLiens], reseau[1][k]
    if nbGroupes > 2:
        r0 = representant(parent, 0)
        i = 1
        while nbGroupes > 2:
            ri = representant(parent, i)
            if r0 != ri:
                fusion(parent, r0, ri)
                nbGroupes -= 1
            i += 1
    return parent
```

La variable `nbLiens` tient à jour le nombre de liens non marqués. Ainsi, pour marquer le lien de rang `k` il suffit de permuter les cases de la liste `reseau[1]` d'indices `k` et `nbLiens`.
Pour l'étape 4 de l'algorithme, on fusionne autant de groupes que nécessaires avec le groupe contenant 0.

Pour passer d'une partition en n groupes à une partition en 2 groupes il faut réaliser $n-2$ fusions ; la complexité d'une fusion est en $O(\alpha(n))$ donc le coût total de ces fusion est en $O(n\alpha(n))$.
Dans le pire des cas, pour chaque lien présent on détermine le représentant de la classe de chacun des deux protagonistes, ce qui occasionne un coût en $O(m\alpha(n))$.

Enfin, l'étape 4 peut dans le pire des cas occasionner la recherche de $n-1$ représentants dans le tableau `parent`, avec là encore un coût total en $\mathrm{O}(n\alpha(n))$.
En définitive, la complexité totale de cette fonction est un $\mathrm{O}((m+n)\alpha(n))$.

Question 15. Pour chaque lien d'amitié déclaré dans le réseau on détermine si les deux protagonistes sont dans le même groupe ou pas.

```
def tailleCoupe(reseau, parent):
    nbLiens = 0
    for [i, j] in reseau[1]:
        if representant(parent, i) != representant(parent, j):
            nbLiens += 1
    return nbLiens
```