Devoir Surveillé ITC n°1

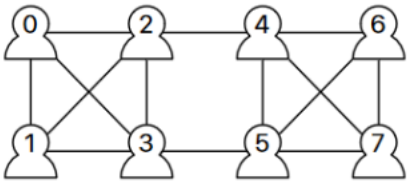
Étude de réseaux sociaux

Le but de cette étude est de regrouper des personnes par affinité dans un réseau social. Pour cela, on cherche à répartir les personnes en deux groupes de sorte à minimiser le nombre de liens d'amitié entre les deux groupes. Une telle partition s'appelle une coupe minimale du réseau. Notre étude se limitera à la préparation de la coupe minimale.

Structure de données Nous supposerons que les individus sont numérotés de 0 à $n-1$ où n est le nombre total d'individus. Nous représenterons chaque lien d'amitié entre deux individus i et j par une liste contenant leurs deux numéros dans un ordre quelconque, c'est-à-dire par la liste $[i, j]$ ou par la liste $[j, i]$ indifféremment. Un réseau social R entre n individus sera représenté par une liste `réseau` à deux éléments où :

- `reseau[0]` contient le nombre n d'individus appartenant au réseau ;
- `reseau[1]` est la liste non-ordonnée (et potentiellement vide) des liens d'amitié déclarés entre les individus.

La figure 1 donne l'exemple d'un réseau social et d'une représentation possible sous la forme de liste. Chaque lien d'amitié entre deux personnes est représenté par un trait entre elles. Les listes représentant les liens d'amitiés sont triées par ordre lexicographique croissant afin de n'oublier personne.




```
reseau_exemple = [8, [[0, 1],
[0, 3], [0, 2], [1, 2], [1, 3],
[2, 3], [2, 4], [3, 5], [4, 5],
[4, 6], [4, 7], [5, 6], [5, 7],
[6, 7]]]
```

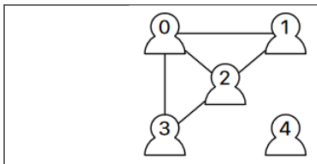
Figure 1 : réseau de 8 individus ayant déclarés 14 liens d'amitié sous forme de schéma et sous forme d'une liste formée comme suit :[nombre d'individus, liste des liens d'amitié]

Important : dans tout le sujet, les méthodes suivantes sont INTERDITES
`liste.insert` , `liste.sort` , `liste.remove` , `liste.index`

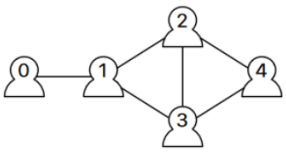
La méthode `liste.append` est autorisée.

I. Manipulation de représentation de réseaux sociaux

1 – Donner une représentation sous forme de listes pour chacun des deux réseaux sociaux ci-dessous :



reseau_a



reseau_b

```
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]]]
```

2 – Écrire une fonction `créer_réseau_vide(n: int) -> list` qui crée, initialise et renvoie la représentation sous forme de liste du réseau à n individus n'ayant aucun lien d'amitié déclaré.

```
def créer_réseau_vide(n: int) -> list:
    réseau = []
    réseau.append(n)
    réseau.append([])
    return réseau
```

3 – Écrire une fonction `est_un_lien_entre(paire: list, i: int, j: int) -> bool` où `paire` est une liste à deux éléments et i et j sont deux entiers, et qui renvoie `True` si les deux éléments contenus dans paire sont i et j dans un ordre quelconque, et renvoie `False` sinon.

```
def est_un_lien_entre(paire: list, i: int, j: int) -> bool:
    lien = False
    if (paire[0] == i and paire[1] == j) or (paire[1] == i and paire[1] == j):
        lien = True
    return lien

# autre proposition
def est_un_lien_entre(paire: list, i: int, j: int) -> bool:
    if  [i, j] == paire or [j, i] == paire:
        return True
    return False

# autre proposition
def est_un_lien_entre(paire: list, i: int, j: int) -> bool:
    return paire == [i , j] or paire == [j , i]
```

4 – Écrire une fonction `sont_amis(réseau: list, i: int, j: int) -> bool` qui renvoie `True` s'il existe un lien d'amitié entre les individus i et j dans le réseau `réseau` ; et renvoie `False` sinon. On pourra utiliser la fonction précédente `est_un_lien_entre` .

```
def sont_amis(réseau: list, i: int, j: int) -> bool:
    lien = False
    for lien in réseau[1]:
        if est_un_lien_entre(lien, i, j):
            lien = True
    return lien
```

5 –Écrire une procédure `déclare_amis(réseau: list, i: int, j: int)` qui modifie le réseau `réseau` pour y ajouter le lien d'amitié entre les individus i et j si ce lien n'y figure pas déjà.

```
def déclare_amis(réseau: list, i: int, j: int):
    if not sont_amis(réseau, i, j):
        réseau[1].append([i, j])
```

6 – Écrire une fonction `liste_des_amis_de(réseau: list, i: int) -> list` qui renvoie la liste des amis de i dans le réseau `réseau` .

```
def liste_des_amis_de(réseau: list, i: int) -> list:
    liste_amis = []  # Initialise la liste attendue
    for lien in réseau[1]:
        if lien[0] == i:
            liste_amis.append(lien[1])
        elif lien[1] == i:
            liste_amis.append(lien[0])
    return liste_amis

# autre proposition
def liste_des_amis_de(réseau: list, i: int) -> list:
```

```
    liste_amis = []  # Initialise la liste attendue
    n = réseau[0]
    for j in range(n):
        if sont_amis(réseau, i, j):
            liste_amis.append(j)
    return liste_amis
```

II. Partitions en groupes d'un ensemble

Une partition en k groupes d'un ensemble A à n éléments consiste à scinder l'ensemble A en k sous-ensembles disjoints non-vides $A_1, ..., A_k$ de A dont l'union est A, c'est-à-dire tels que

$$A_1 \cup ... \cup A_k = A \quad \text{et pour tout } i \neq j, \quad A_i \cap A_j = \varnothing$$

Exemple : $A_1 = \{1, 3\},\ A_2 = \{0, 4, 5\},\ A_3 = \{2\}$ est une partition en trois groupes de $A = [\![6]\!]$, où $[\![6]\!]$ représente l'ensemble des entiers de 0 à 5.

Dans cette partie, nous implémentons une structure de données très efficace pour coder des partitions de $[\![n]\!]$. Le principe de cette structure de données est que les éléments de chaque groupe sont structurés par une relation filiale :

- chaque élément a un (unique) parent choisi dans le groupe
- et l'unique élément du groupe qui est son propre parent est nommé représentant du groupe.

On s'assure par construction que chaque élément i du groupe a bien pour ancêtre le représentant du groupe, c'est-à-dire que le représentant du groupe est bien le parent du parent du parent etc...(autant de fois que nécessaire) du parent de l'élément i. La figure 2 ci-dessous présente un exemple de cette structure de données où la relation filiale est symbolisée par une flèche allant de l'enfant au parent.

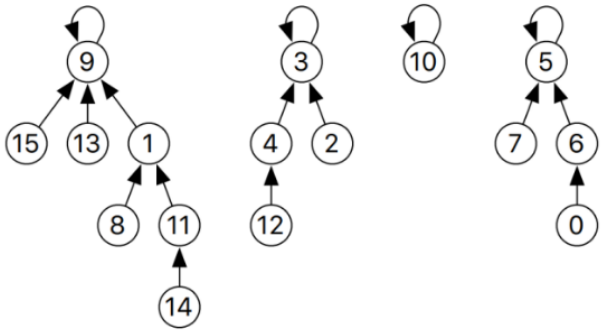


Figure 2 : représentation filiale de $[\![16]\!]$ en quatre groupes : $\{1, 8, 9, 11, 13, 14, 15\}$, $\{2, 3, 4, 12\}$, $\{10\}$ et $\{0, 5, 6, 7\}$ dont les représentants respectifs sont 9, 3, 10 et 5.

Dans l'exemple de cette figure, 14 a pour parent 11 qui a pour parent 1 qui a pour parent 9 qui est son propre parent. Ainsi, 9 est le représentant du groupe auquel appartiennent 14, 11, 1 et 9. Notons que ce groupe contient également 8, 13 et 15. A noter que la représentation n'est pas unique (si l'on choisit un autre représentant pour un groupe et une autre relation filiale, on aura une autre représentation du groupe).

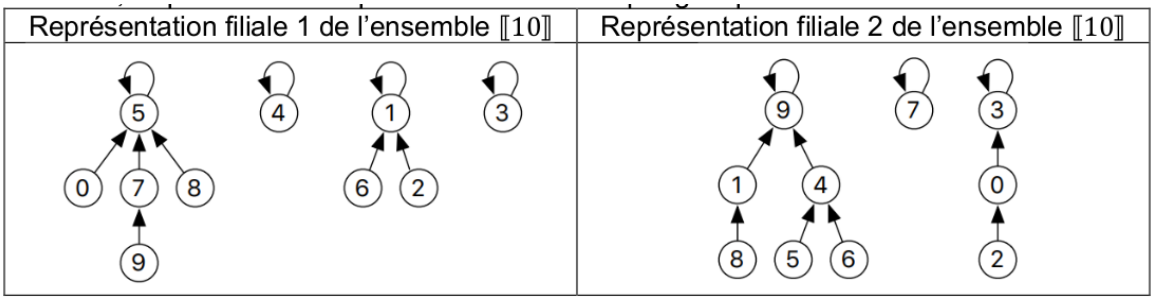
Pour coder cette structure, on utilise un tableau `parent` sous forme de liste à n éléments où la case `parent[i]` contient le numéro du parent de i. Par exemple, les valeurs du tableau parent encodant la représentation filiale donnée dans la figure 2 sont :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
parent[i]	6	9	3	3	3	5	5	5	1	9	10	1	4	9	11	9

Ainsi, dans le cas de la figure 2, on peut noter que

```
parent = [6, 9, 3, 3, 3, 5, 5, 5, 1, 9, 10, 1, 4, 9, 11, 9]
```

7 – Donner les valeurs des tableaux `parent1` et `parent2` encodant les deux représentations filiales des partitions de $[\![10]\!]$ modélisées sous forme de schémas ci-dessous, et préciser les représentants de chaque groupe.




```
parent_1 = [5, 1, 1, 3, 4, 5, 1, 5, 5, 7]
parent_2 = [3, 9, 0, 3, 9, 4, 4, 7, 1, 9]
```

Au départ de l'algorithme de partition, chaque élément de $[\![n]\!]$ est son propre représentant et la partition initiale d'un ensemble consiste à scinder l'ensemble en n groupes contenant chacun un seul élément, soit en n singletons. Ainsi, initialement `parent[i] = i` pour tout i appartenant à $[\![n]\!]$.

8 – Écrire une fonction `créer_partition_en_singletons(n: int) -> list` qui crée et renvoie un tableau parent sous forme de liste de n éléments dont les valeurs sont initialisées à `parent[i] = i` pour tout i appartenant à $[\![n]\!]$ comme indiqué ci-dessus.

```
def créer_partition_en_singletons(n: int) -> list:
    parent = []  #initialisation du tableau
    for i in range(n):
        parent.append([i])
    return parent

# autre proposition
def créer_partition_en_singletons(n: int) -> list:
    return [i for i in range(n)]
```

9 – Écrire une fonction `représentant(parent: list, i: int) -> int` qui utilise le tableau `parent` pour trouver et renvoyer l'indice du représentant du groupe auquel appartient i dans la partition encodée par le tableau parent.

```
def représentant(parent: list, i: int) -> int:
    rep = parent[i]  # on part du parent de i
    while rep != parent[rep]:
        rep = parent[rep]
    return rep
```

Opération de fusion Pour réaliser la fusion de deux groupes désignés par l'un de leurs éléments i et j respectivement, on applique l'algorithme suivant :

- calculer les représentants p et q des deux groupes contenant i et j respectivement ;
- faire `parent[p] = q` .

La figure 3 présente la structure filiale obtenue après la fusion des groupes contenant respectivement 6 et 14 de la figure 2.

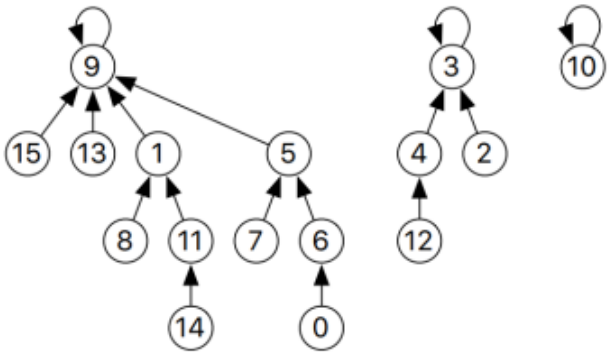


Figure 3 : représentation filiale de la fusion des groupes contenant respectivement les éléments 6 et 14 de la figure 2

10 – Écrire une procédure `fusion(parent: list, i: int, j: int)` qui modifie le tableau `parent` donné en paramètre pour fusionner les deux groupes contenant i et j respectivement.

```
def fusion(parent: list, i: int, j: int):
    p = représentant(parent, i)
    q = représentant(parent, j)
    parent[p] = q
```

11 – En suivant l'algorithme proposé, que se passe-t-il si i et j appartiennent au même groupe ?

Si i et j appartiennent au même groupe, $p = q$ avant l'appel de la procédure, donc celle-ci ne change rien puisqu'un représentant de groupe est, par définition, son propre parent.

12 – Quelles préconditions peut-on envisager pour cette procédure ?

- i et $j < n$ (indices valides)
- `parent` ne contient que des entiers $\in [\![0; n-1]\!]$
- i et j n'appartiennent pas au même groupe pour qu'une modification soit opérée (secondaire)

Le résultat des fusions peut conduire à des structures avec de longues relations parent-enfant ; par exemple la figure 3 montre qu'il y a beaucoup d'intermédiaires entre 14 et le représentant du groupe 9. Pour compacter la structure, on peut choisir un élément i du groupe et faire en sorte que chacun de ses ancêtres, i y compris, ait pour parent direct le représentant. Dans l'exeple de la figure 3, le résultat de la compression menée depuis 14 est présenté figure 4.

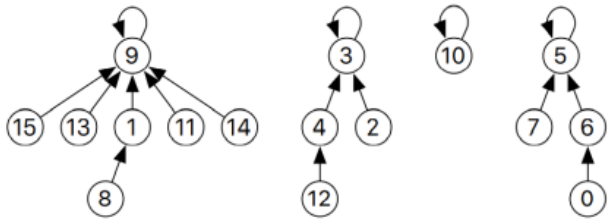


Figure 4 : résultat de la compression depuis 14 dans la représentation filiale de la figure 2

13 –Écrire une procédure `compression(parent: list, i: int) -> int` qui modifie le tableau `parent` fourni en paramètre pour faire pointer directement tous les ancêtres de i vers le représentant de i une fois qu'il a été trouvé.

```
def compression(parent: list, i: int) -> int:
    rep = représentant(parent, i)
    précédent = i
```

```
    while précédent != rep:
        suivant = parent[précédent]
        parent[précédent] = rep
        précédent = suivant
```

III. Algorithme randomisé pour la coupe minimum (hors barême)

Cette partie est **hors-barême** ; elle ne sera évaluée que si toutes les questions précédentes ont été traitées.

Revenons à présent à notre objectif principal : trouver une partition des individus d'un réseau social en deux groupes qui minimise le nombre de liens d'amitiés entre les deux groupes. Pour résoudre ce problème nous allons utiliser l'algorithme randomisé suivant :

- Créer une partition P en n singletons de $[\![n]\!]$.
- Initialement aucun lien d'amitié n'est marqué.
- Tant que la partition P contient au moins trois groupes et qu'il reste des liens d'amitié non-marqués dans le réseau faire :
  - ▷ Choisir un lien uniformément au hasard parmi les liens non-marqués du réseau ; notons-le $[i, j]$.
  - ▷ Si i et j n'appartiennent pas au même groupe dans la partition P, fusionner les deux groupes correspondants.
  - ▷ Marquer le lien $[i, j]$.
- Si P contient $k > 3$ groupes, faire $k - 1$ fusions pour obtenir deux groupes.
- Renvoyer la partition P.

La figure 5 présente une exécution possible de cet algorithme randomisé sur le réseau de la figure 1.

14 – Écrire une fonction `coupe_minimum_randomisée(réseau)` qui renvoie le tableau parent correspondant la partition calculée par l'algorithme ci-dessus. On considèrera que la fonction `randint(a: int, b: int) -> int` qui renvoie un entier entre a et b **inclus** est disponible. On pourra introduire toute fonction utile pour aider à la lisibilité de l'algorithme, par exemple pour compter le nombre de partitions ou lister les partitions.

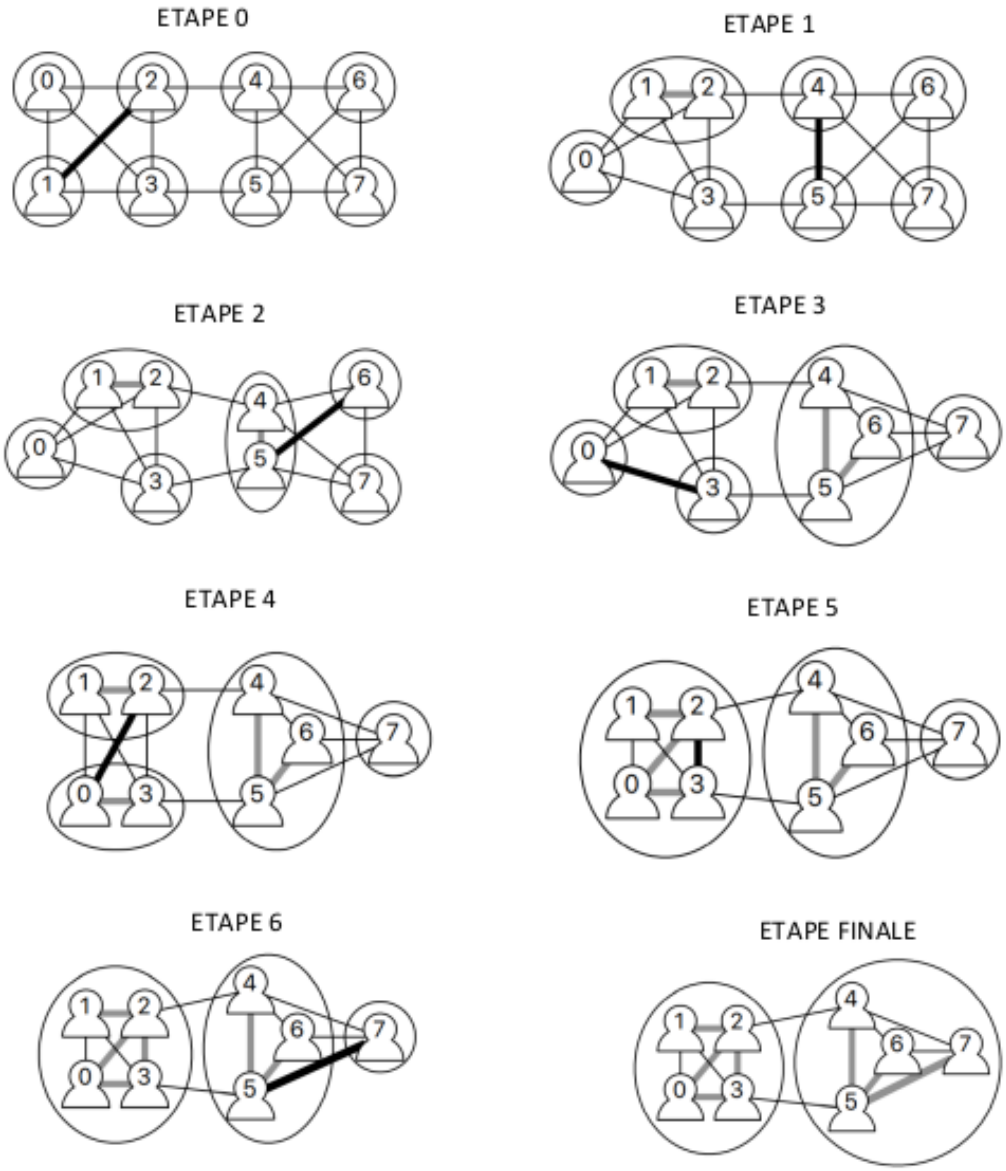


Figure 5 : Une exécution de l'algorithme randomisé sur le réseau de la figure 1 où les liens sélectionnés aléatoirement sont dans l'ordre : [2,1], [4,5], [6,5], [0,3], [2,0], [3,2] et [5,7]. Les liens représentes en noir épais sont les liens sélectionnés au hasard 0 l'étape courante ; les liens épais et gris sont les liens marqués par l'algorithme ; les ronds représentent la partition à l'étape courante.

```python
def coupe_minimum_randomisée(réseau):
    n, n_liens = réseau[0], len(réseau[1])
    liens_restants = [k for k in range(n_liens)]
    parent = créer_partition_en_singletons(n)
    n_groupes, n_restants = n, n_liens
    # Première phase
    while n_groupes > 2 and n_restants > 0:  # on peut arriver à cours de lien ou de groupes
        # on cherche un lien restant
        num_lien = randint(0, n_restants-1)
        lien = lients_restants[num_lien]
        # on extrait les individus du lien non marqué, on fusionne si possible
        i, j = réseau[1][lien]
        if représentant(parent, i) != représentant(parent, j):
            fusion(parent, i, j)
            n_groupes -= 1
        # on retire le lien en le plaçant en dernière position
        a, b = liens_restants[n_restants-1], liens_restants[num_lien]
        liens_restants[num_lien], liens_restants[n_restants-1] = a, b
        liens_restants.pop()  # on enlève le dernier
        n_restants -= 1       # len(liens_restants) = n_restants
    # Si il reste plus de deux groupes : on fusionne
    while n_groupes > 2:
        # on cherche deux personnes dans des groupes différents
        i = randint(0, n-1)
        rep_i = représentant[i]
        j = 0
        while rep_i != représentant[j]:
            j += 1
        fusion(parent, i, j)
        n_groupes -= 1
    return parent
```