Devoir surveillé #3 — Coupe minimale randomisée (avec solutions)

Mardi 2026-03-17 ; durée : deux heures

Ce sujet est intégralement repris de l'épreuve de *composition d'informatique B* (filière MP hors spécialité info, filière PC) XELCR 2016.
Les différences principales sont la suppression d'une aide à la programmation Python et la correction d'une remarque culturelle en fin de sujet.

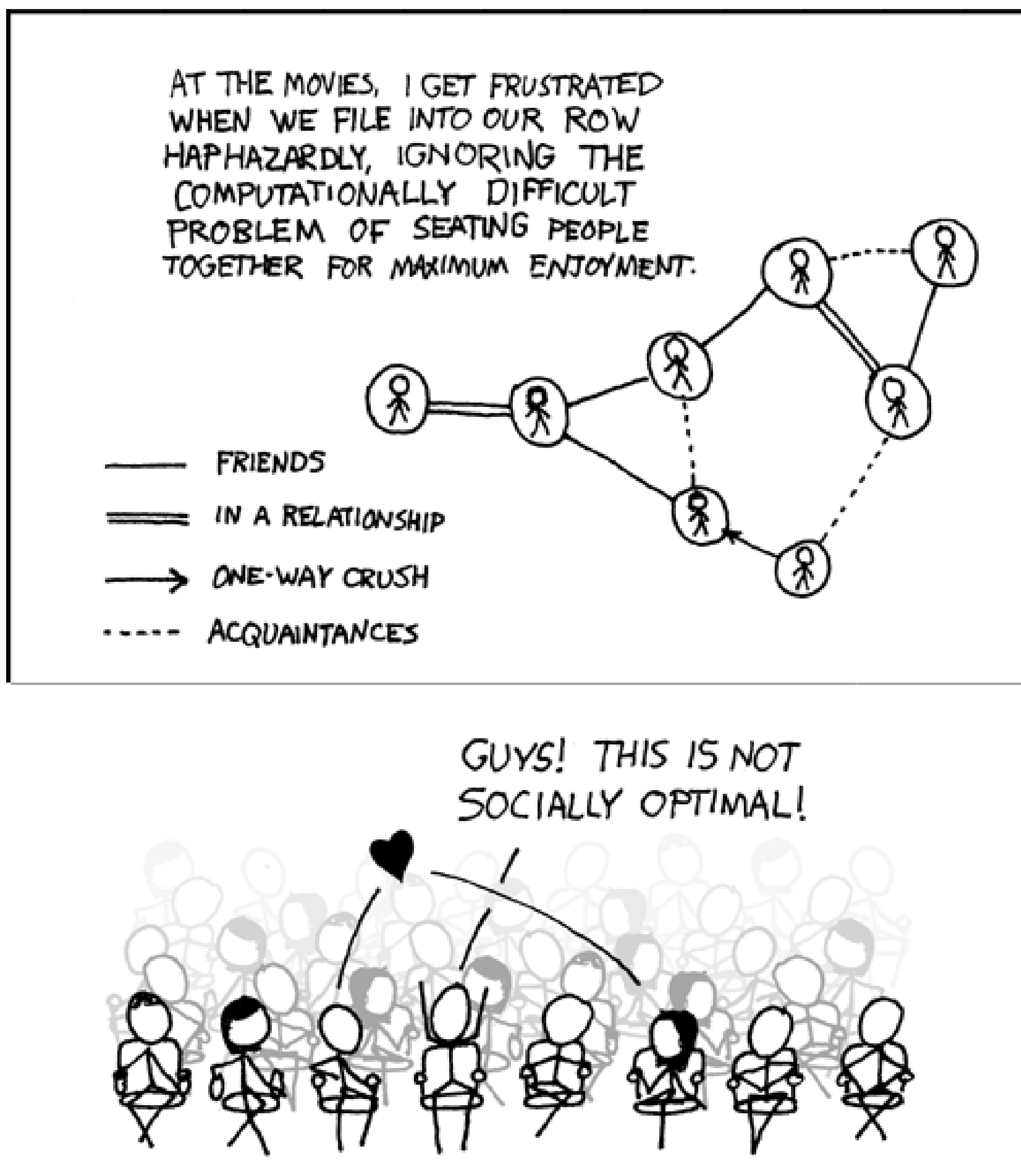


FIGURE 1 – xkcd #173 — *It's like the traveling salesman problem, but the endpoints are different and you can't ask your friends for help because they're sitting three seats down.*

Ce sujet est découpé en deux parties totalement indépendantes toutes deux portant sur l'étude de réseaux sociaux : la première porte sur de la programmation en Python ; la seconde porte sur l'interrogation de base de données à l'aide de requêtes en SQL.

A. Programmation en Python

Notations. On désignera par $[\![n]\!]$ l'ensemble des entiers de 0 à $n-1$: $[\![n]\!] = \{0,\dots,n-1\}$.

Objectif. Le but de cette partie 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*.

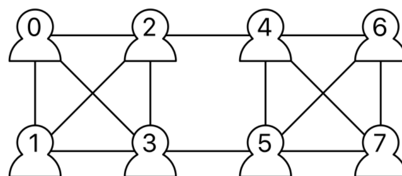
Complexité. La complexité, ou le temps d'exécution, d'un programme P est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc.) nécessaires à l'exécution de P. Lorsque cette complexité dépend de plusieurs paramètres n et m, on dira que P a une complexité en $O(\varphi(n,m))$, lorsqu'il existe trois constantes A, n_0 et m_0 telles que la complexité de P soit inférieure ou égale à $A \times \varphi(n,m)$, pour tout $n \geq n_0$ et $m \geq m_0$.
Lorsqu'il est demandé de donner une certaine complexité, le ou la candidate devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Partie I. Réseaux sociaux

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 `list` contenant leurs deux numéros dans un ordre quelconque, c.-à-d. par la `list` `[i,j]` ou par la `list` `[j,i]` indifféremment.
Un réseau social R entre n individus sera représenté par une `list` `reseau` à deux éléments où :

- `reseau[0]` $= n$ contient le nombre d'individus appartenant au réseau
- `reseau[1]` $=$ la `list` **non-ordonnée** *(et potentiellement vide)* des liens d'amitié déclarés entre les individus

La Figure 2 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.




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

FIGURE 2 – Un réseau à 8 individus ayant 14 liens d'amitié déclarés et l'une de ses représentations possibles en mémoire sous forme d'une liste [Nombre d'individus, liste des liens d'amitié].

1. Donner une représentation sous forme de `list` pour chacun des deux réseaux sociaux de la Figure 3.

Par exemple :

```python
reseauA = [ 5,
            [ [0, 1], [0, 2], [0, 3],
              [2, 1], [2, 3] ]
          ]

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

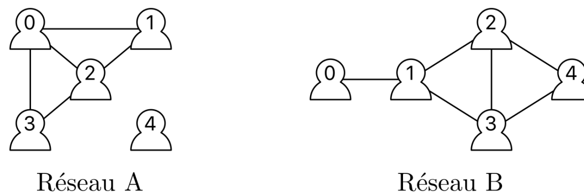


FIGURE 3 – Réseaux de la Question 1.

2. Écrire une fonction `creerReseauVide(n)` qui crée, initialise et renvoie la représentation sous forme de `list` du réseau à `n` individus n'ayant aucun lien d'amitié déclaré entre eux.

On propose :

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

3. Écrire une fonction `estUnLienEntre(paire,i,j)` où `paire` est une `list` 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.

On propose :

```python
def estUnLienEntre(paire, i, j):
    return (paire[0] == i and paire[1] == j) or (paire[1] == i and paire[0] == j)
```

4. Écrire une fonction `sontAmis(reseau,i,j)` qui renvoie `True` s'il existe un lien d'amitié entre les individus `i` et `j` dans le réseau `reseau` ; et renvoie `False` sinon.
Quelle est la complexité en temps de votre fonction dans le pire cas en fonction du nombre m de liens d'amitié déclarés dans le réseau ?

On propose :

```python
def sontAmis(reseau, i, j):
    for p in reseau[1]: # m itérations
        if estUnLienEntre(p, i, j): # coût O(1)
            return True
    return False
```

Le coût est un $O(m)$.

5. Écrire une fonction `declareAmis(reseau,i,j)` qui modifie le réseau `reseau` pour y ajouter le lien d'amitié entre les individus `i` et `j` *si ce lien n'y figure pas déjà*.
Quelle est la complexité en temps de votre fonction dans le pire cas en fonction du nombre m de liens d'amitié déclarés dans le réseau ?

On propose :

```python
def declareAmis(reseau, i, j):
    if sontAmis(reseau, i, j): # coût O(m) (question précédente)
        return
    reseau[1].append([i, j]) # coût O(1)
```

Le coût est un $O(m)$.

6. Écrire une fonction `listeDesAmisDe(reseau,i)` qui renvoie la `list` des amis de `i` dans le réseau `reseau`.
Quelle est la complexité en temps de votre fonction dans le pire cas en fonction du nombre m de liens d'amitié déclarés dans le réseau ?

On propose :

```
def listeDesAmisDe(reseau, i):
    amis = []
    for p in reseau[1]: # m itérations
        if p[0] == i: # coût O(1) pour ce cas
            amis.append(p[1])
        elif p[1] == i: # pareil
            amis.append(p[0])
    return amis
```

Le coût est un $O(m)$.

Partie II. Partitions

Une *partition* en k groupes d'un ensemble A à n éléments consiste en k sous-ensembles **disjoints non-vides** $A_1, \ldots, A_k$ de A dont l'union est A, c.-à-d. tels que $A_1 \cup \cdots \cup A_k = A$ et pour tout $i \neq j$, $A_i \cap A_j = \emptyset$. Par exemple $A_1 = \{1, 3\}$, $A_2 = \{0, 4, 5\}$, $A_3 = \{2\}$ est une partition en trois groupes de $A = [\![6]\!]$. 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 appelé le *représentant* du groupe. On s'assure par construction que chaque élément `i` du groupe a bien pour ancêtre le représentant du groupe, c.-à-d. 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 relation filiale est symbolisée par une flèche allant de l'enfant au parent dans la Figure 4 qui présente un exemple de cette structure de données. 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.

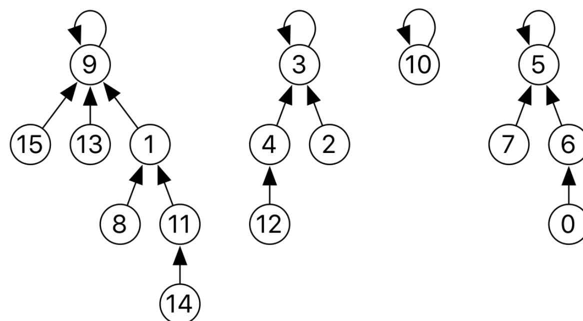


FIGURE 4 – Une représentation filiale de la partition suivante de $[\![16]\!]$ en quatre groupes : $\{1, 8, 9, 11, 13, 14, 15\}$, $\{2, 3, 4, 12\}$, $\{10\}$ et $\{0, 5, 6, 7\}$ dont les représentants respectifs sont 9, 3, 10 et 5.

Pour coder cette structure, on utilise un tableau `parent` à n éléments (représenté par une `list` Python) 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 4 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

7. Donner les valeurs du tableau `parent` encodant les représentations filiales des deux partitions de $[\![10]\!]$ de la Figure 5, et préciser les représentants de chaque groupe.

On a :

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

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

Les représentants des groupes (dans l'ordre gauche-droite) sont respectivement 5, 4, 1, 3 et 9, 7, 3.

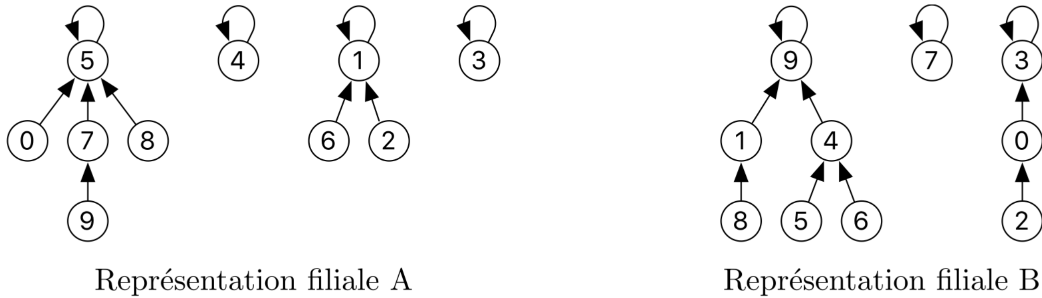


FIGURE 5 – Représentations filiales de la Question 7.

Initialement, chaque élément de $[\![n]\!]$ est son propre représentant et la partition initiale consiste en n groupes contenant chacun un individu. Ainsi, initialement, `parent[i]` $=$ `i` pour tout `i` $\in [\![n]\!]$.

8. Écrire une fonction `creerPartitionEnSingletons(n)` qui crée et renvoie un tableau `parent` (représenté comme une `list`) à `n` éléments dont les valeurs sont initialisées de sorte à encoder la partition de $[\![n]\!]$ en `n` groupes d'un seul élément.

On propose :

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

Nous sommes intéressés par deux opérations sur les partitions :

- Déterminer si deux éléments appartiennent au même groupe dans la partition.
- Fusionner deux groupes pour n'en faire plus qu'un. Par exemple, la fusion des groupes $A_1 = \{1,3\}$ et $A_3 = \{2\}$ dans la partition de $[\![6]\!]$ donnée en exemple au tout début de cette partie donnera la partition en deux groupes $A_2 = \{0,4,5\}$ et A_4 où $A_4 = A_1 \cup A_3 = \{1,2,3\}$.

9. Écrire une fonction `representant(parent,i)` 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`. Quelle est la complexité dans le pire cas de votre fonction en fonction du nombre total n d'éléments ? Donnez un exemple de tableau `parent` à n éléments qui atteigne cette complexité dans le pire cas.

On propose une version récursive ainsi qu'une version itérative :

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

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

Le coût pire cas est un $O(n)$, par exemple atteint pour un appel `representant(parent, 0)` avec `parent = [1, 2, 3, ..., n - 1, n - 1]`

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 6 présente la structure filiale obtenue après la fusion des groupes contenant respectivement 6 et 14 de la Figure 4.

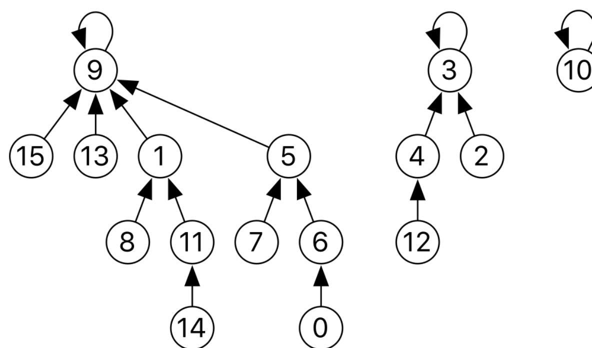


FIGURE 6 – Représentation filiale obtenue après la fusion des groupes contenant respectivement 6 et 14 de la Figure 4.

10. Écrire une fonction `fusion(parent,i,j)` qui modifie le tableau parent pour fusionner les deux groupes contenant `i` et `j` respectivement.

Il suffit de recopier l'algorithme donné dans le sujet :

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

Pour l'instant, la structure de données n'est pas très efficace comme le montre la question suivante.

11. Proposer une suite de $(n-1)$ fusions dont l'exécution à partir de la partition en n singletons de $[\![n]\!]$, nécessite de l'ordre de n^2 opérations élémentaires.

C'est par exemple le cas pour :

```
fusion(parent, 0, 1)       # [1, 1, 2, ...]
fusion(parent, 0, 2)       # [1, 2, 2, ...]
fusion(parent, 0, 3)       # [1, 2, 3, 3, ...]
# ...
fusion(parent, 0, n - 1)   # [1, 2, 3, 4, ..., (n-2), (n-1), (n-1)]
```

Pour remédier à cette mauvaise performance, une astuce consiste à *compresser la relation filiale* après chaque appel à la fonction `representant(parent,i)`. L'opération de compression consiste à faire la chose suivante : si p est le résultat de l'appel à la fonction `representant(parent,i)`, modifier le tableau parent de façon à ce que chaque ancêtre (c.-à-d. parent de parent . . . de parent) de `i`, `i` y compris, ait pour parent direct p. Noter bien que même si un appel à `representant(parent,i)` renvoie le représentant de `i` elle modifie également le tableau `parent`. Si l'on reprend l'exemple de la Figure 4, le résultat de l'appel `representant(parent,14)` est 9, que l'on a calculé en remontant les ancêtres successifs de 14 : 11, 1 puis 9. L'opération de compression consiste alors à donner la valeur 9 aux cases d'indices 14, 11, et 1 du tableau parent. La structure filiale obtenue après l'opération de compression menée depuis 14 est illustrée dans la Figure 7

12. Modifier votre fonction `representant(parent,i)` pour qu'elle modifie le tableau `parent` pour faire pointer directement tous les ancêtres de `i` vers le représentant de `i` une fois qu'il a été trouvé.
En quoi cette optimisation de la structure filiale peut-elle être considérée comme « gratuite » du point de vue de la complexité ?

La version récursive est celle qui se modifie le plus aisément :

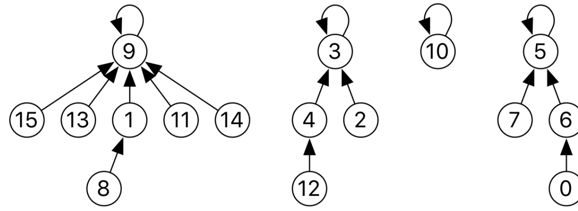


FIGURE 7 – Résultat de la compression depuis 14 dans la représentation filiale de la Figure 4.

```
def representant(parent, i):
    if parent[i] == i:
        return i
    p = representant(parent, parent[i])
    parent[i] = p # O(1)
    return p
```

Pour la version itérative, on peut introduire explicitement un objet stockant les valeurs des indices rencontrés lors de la recherche du représentant :

```
def representantIter(parent, i):
    path = []
    while parent[i] != i: # k itérations
        path.append(i)
        i = parent[i]
    for j in path: # k itérations
        parent[j] = i # O(1)
    return i
```

Dans les deux cas, le nombre d'opérations élémentaires supplémentaires effectué est du même ordre que le coût initial (on en rajoute un nombre constant par indice rencontré jusqu'au représentant), ce qui ne change donc pas le coût temporel et justifie que l'on puisse considérer cette optimisation comme « gratuite ». On peut cependant remarquer que le coût en *mémoire* n'est pas le même, puisque l'on passe d'un coût constant (à supposer que la *récursion terminale* de la version récursive soit correctement optimisée, ce qui n'est pas le cas en Python...) à un coût en $O(k)$ avec k la longueur du chemin de `i` à son représentant. On peut éviter ce surcoût en mémoire avec une variante de l'approche itérative, où l'on « recommence » la recherche depuis le début. Il faut alors prendre garde à bien mémoriser les informations nécessaires lors du second parcours :

```
def representantIter2(parent, i):
    x = i
    while parent[x] != x:
        x = parent[x]
    while parent[i] != x:
        p = parent[i]
        parent[i] = x
        i = p
    return x
```

Afin d'afficher de manière lisible la partition codée par un tableau `parent`, on souhaite calculer à partir du tableau `parent` la `list` des `list` des éléments des différents groupes. Une sortie possible pour le tableau `parent` correspondant à la Figure 4 serait :

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

13. Écrire une fonction `listeDesGroupes(parent)` qui renvoie la liste des différents groupes codés par le tableau `parent` sous la forme d'une `list` des `list` des éléments des différents groupes.

On peut procéder en deux temps : on commence par construire une `list` de `list` où l'on ajoute à la ième case les éléments de représentant `i`, puis l'on renvoie une `list` de ces `list` qui ne sont pas vides :

```python
def listeDesGroupes(parent):
    LG = [[] for _ in range(len(parent))]
    for i in range(len(parent)):
        LG[representant(parent, i)].append(i)
    LGbis = []
    for x in LG:
        if len(x) > 0:
            LGbis.append(x)
    return LGbis
```

Partie III. Algorithme randomisé pour la coupe minimum

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 :

Entrée : un réseau social à n individus

- 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 $k \geq 3$ groupes, faire $k-1$ fusions pour obtenir deux groupes.
- Renvoyer la partition P.

La Figure 8 présente une exécution possible de cet algorithme randomisé sur le réseau de la Figure 2.

14. Écrire une fonction `coupeMinimumRandomisee(reseau)` qui renvoie le tableau parent correspondant à la partition calculée par l'algorithme ci-dessus.
On utilisera la fonction `randint(a,b)` pour tirer uniformément au hasard un entier de l'ensemble $\{a, a+1, \ldots, b\}$.
Indication. Au lieu de marquer explicitement les liens déjà vus, on pourra avantageusement les positionner à la fin de la `list` *non-ordonnée* des liens du réseau et ainsi pouvoir tirer simplement les liens au hasard parmi ceux placés au début de la liste.
Quelle est la complexité de votre fonction en fonction de n, m et $\alpha(n)$, où m est le nombre de liens d'amitié déclarés dans le réseau et où $\alpha(n)$ désigne la complexité d'un appel à la fonction `representant` ?

On se contente de suivre l'algorithme décrit par le sujet On propose une version qui profite de `listeDesGroupes` pour implémenter la seconde boucle, mais ce n'est pas nécessaire :

```
def coupeMinimumRandomisee(reseau):
    k = reseau[0]
    partition = creerPartitionEnSingletons(k) # O(n)

    nl = len(reseau[1])
    while (k > 2 and nl > 0): # O(m * alpha(n))
        ridx = randint(0, nl - 1)
        i = reseau[1][ridx][0]
        j = reseau[1][ridx][1]
        p = representant(partition, i)
        q = representant(partition, j)
        if (p != q):
            fusion(partition, i, j) # alpha(n)
            k = k - 1
        nl = nl - 1
        reseau[1][nl], reseau[1][ridx] = reseau[1][ridx], reseau[1][nl]
    if (k == 2):
        return partition
    G = listeDesGroupes(partition) # O(n * alpha(n))
    for i in range(2, len(G)): # O(n * alpha(n))
        fusion(partition, G[0][0], G[i][0])
    return partition
```

Le coût total est un $O(\max(n,m) \times \alpha(n))$, qui ne se simplifie pas plus puisque l'on peut à la fois avoir $m = o(n)$ et $m = \Omega(n^2)$, et des réseaux qui atteignent la borne dans les deux cas (c'est évident pour le premier, et pour le second on peut penser à un réseau formé de trois composantes connexes complètes de tailles $\approx n/3$).

15. Écrire une fonction `tailleCoupe(reseau, parent)` qui calcule le nombre de liens entre les différents groupes de la partition représentée par `parent` dans le réseau `reseau`.

On propose :

```
def tailleCoupe(reseau, parent):
    taille = 0
```

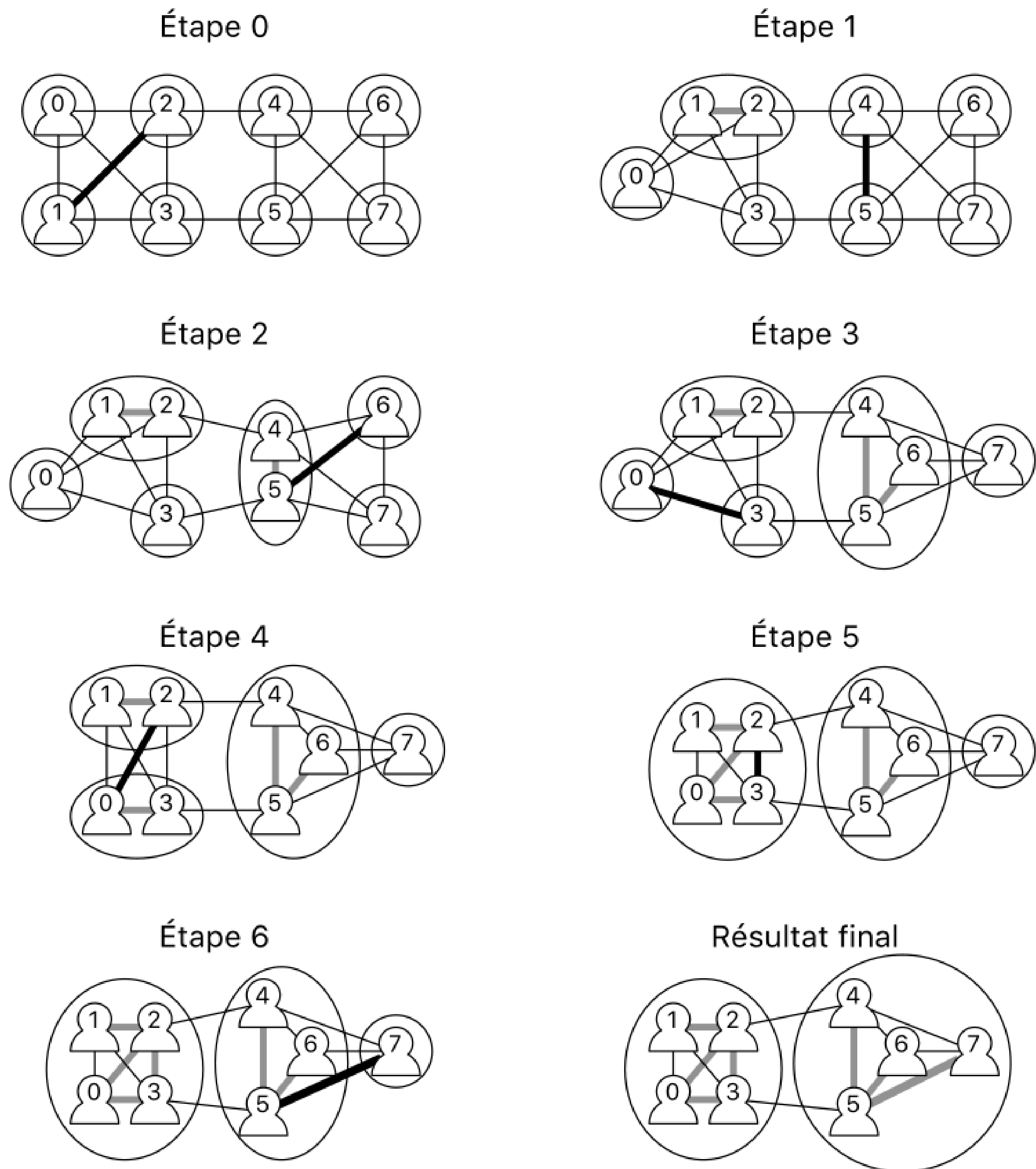


FIGURE 8 – Une exécution de l'algorithme randomisé sur le réseau de la Figure 2 où les liens sélectionnés aléatoirement sont dans l'ordre : `[2,1]`, `[4,5]`, `[6,5]`, `[0,3]`, `[2,0]`, `[3,2]` et `[5,7]`. Les liens représentés en noir épais sont les liens sélectionnés au hasard à l'étape courante ; les liens épais et grisés sont les liens marqués par l'algorithme ; les ronds représentent la partition à l'étape courante.

```
    for p in reseau[1]:
        if representant(parent, p[0]) != representant(parent, p[1]):
            taille = taille + 1
    return taille
```

On peut démontrer que cet algorithme renvoie une coupe de taille minimum avec une probabilité supérieure

à $1/n$, ce qui fait que la meilleure parmi n exécutions indépendantes de cet algorithme est effectivement minimum avec probabilité supérieure à $1/e \approx 0.36787$.

La structure de données filiale avec compression pour les partitions est connue sous le nom d'*union-find*. Elle est particulièrement efficace aussi bien en pratique qu'en théorie : la complexité de la création initiale d'un ensemble de n singletons suivie de (au plus) $n-1$ appels à `fusion` et k appels à `representant` est un $O(n + k \times (1 + \log_{2+k/n} n))$ opérations élémentaires.
Lorsque la compression est combinée avec une autre technique dite d'*union par rangs*, la complexité de m opérations quelconques sur une partition de n éléments est un $O(m\alpha(n))$ opérations élémentaires, où α est la *fonction Ackermann inverse*, une fonction qui croît *extrêmement* lentement vers l'infini (notamment, $\alpha(n) \leq 4$ pour tout n concevable).

B. Programmation SQL

Une représentation simplifiée, réduite à deux tables, de la base de données d'un réseau social est donnée par le schéma :

— **individus** : (<u>id</u> : entier ; nom : texte ; prenom : texte)
— **liens** : (<u>id1</u> : entier ; <u>id2</u> : entier)

La table **individus** répertorie les individus, et la table **liens** répertorie les liens d'amitiés entre individus.
On supposera par ailleurs que pour tout couple `(x,y)` dans la table **liens**, le couple `(y,x)` est également présent dans la table (contrairement à la partie précédente de cet énoncé).

16. Écrire une requête SQL qui renvoie les identifiants des amis de l'individu d'identifiant `x`.

On propose :
```
SELECT id1 FROM LIENS WHERE id2 = x;
```

17. Écrire une requête SQL qui renvoie les (noms,prénoms) des amis de l'individu d'identifiant `x`.

On propose :
```
SELECT nom, prenoms FROM INDIVIDUS
JOIN LIENS ON id = id1
WHERE id2 = x;
```

18. Écrire une requête SQL qui renvoie les identifiants des individus qui sont amis avec au moins un ami de l'individu d'identifiant `x`.

On propose :
```
SELECT DISTINCT id1 FROM LIENS l1
JOIN LIENS l2 ON l1.id2 = l2.id1 WHERE l2.id2 = x;
```
On peut visualiser l'idée derrière cette requête en représentant la table qu'elle construit :

l1.id1	l1.id2	l2.id1	l2.id2
a	b	b	x

Fin du sujet