Devoir Surveillé d'ITC N°2
le 24/05/2024
MPSI & PCSI

Consignes

- Les codes doivent être présentés en mentionnant explicitement l'indentation, au moyen par exemple de barres verticales sur la gauche.
- Pour qu'un code soit compréhensible, il convient de choisir des noms de variables les plus explicites possibles.
- La numérotation des exercices (et des questions) doit être respectée et mise en évidence. Les résultats (hors questions purement informatiques) doivent être encadrés proprement.
- Il est important de numéroter correctement les pages des copies qui seront données à la correction. Chaque candidat est responsable de la vérification de son sujet d'épreuve : pagination et impression de chaque page.
- Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il convient de le signaler sur la copie et de poursuivre la composition en expliquant les raisons des initiatives qui ont été prises.
- Les candidats ne doivent avoir aucune communication entre eux ou avec l'extérieur durant l'épreuve. Aussi, l'utilisation des téléphones portables et, plus largement, de tout appareil permettant des échanges ou la consultation d'informations, est interdite.
- À l'issue de la durée prévue pour cette épreuve, les candidats doivent déposer le stylo et ne sont plus autorisés à écrire quoi que ce soit sur leur copie. Tout retard donne lieu à une pénalité sur la note finale.
- **L'usage de la calculatrice est interdit.**

Les signatures des fonctions devront toutes être écrites !

Problème — Coupe minimale d'un réseau social (X2016 largement remanié)

NOTATIONS. On désigne par $[\![0, n-1]\!]$ l'ensemble des entiers de 0 à $n-1$, c'est-à-dire : $[\![0\,,\, n-1]\!] = \{0, 1, \dots, n-1\}$.

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 groupes de façon à minimiser le nombre de liens d'amitié entre les groupes. Suivant les cas, on pourra chercher à former un nombre de groupes au moins égal à deux, ou exactement égal à deux. Dans le cas où on forme exactement deux groupes, on parle de *coupe minimale du réseau*.

COMPLEXITÉ. Lorsqu'il est demandé de donner une certaine complexité, le candidat devra donner la complexité asymptotique et justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

IMPLÉMENTATION. On rappelle qu'en python, on dispose des opérations suivantes, qui ont toutes une complexité constante (car en python, les listes sont en fait des tableaux de taille dynamique).

- `[]` crée une liste vide (c'est-à-dire ne contenant aucun élément).
- `[x]*n` crée une liste (ou un tableau) à `n` éléments contenant tous la valeur contenue dans `x`. Par exemple, `[1]*3` renvoie la liste (ou le tableau) `[1, 1, 1]` à 3 cases contenant toutes la même valeur 1.
- `len(lst)` renvoie la longueur de la liste `lst`.
- `lst[i]` désigne le `(i + 1)`-ème élément de la liste `lst` s'il existe et produit une erreur sinon. Noter que le premier élément de la liste est `lst[0]`.
- `lst.append(x)` ajoute le contenu de `x` à la fin de la liste `lst` qui s'allonge ainsi d'un élément.
- `lst.insert(i, x)` insère l'élément `x` en position i dans la liste `lst`,
- `lst.pop()` renvoie la valeur du dernier élément de la liste `lst` et l'élimine de la liste.
- `rd.randint(a, b)` (après avoir réalisé l'importation `import random as rd` en préambule) renvoie un entier tiré (pseudo)-aléatoirement et uniformément dans l'ensemble $\{a, a+1, ..., b-1, b\}$ (on supposera que le module `random` a été préalablement importé).
- `True` et `False` sont les deux valeurs booléennes *vrai* et *faux*.

IMPORTANT. L'usage de toute autre commande sur les listes, par exemple `lst.remove(x)`, `lst.index(x)`, `lst.sort(x)`, est **rigoureusement interdit**. Ces fonctions devront être programmées explicitement si nécessaire.

Dans la suite, on distingue *fonction* et *procédure*. Une *fonction* renvoie une valeur (un entier, une liste, *etc.*). Une *procédure* ne renvoie aucune valeur, mais peut modifier des variables par effet de bord.

Partie I — Réseaux sociaux et graphes

Nous supposerons que les individus sont numérotés de 0 à $n-1$ où n est le nombre total d'individus.

Structure de données. Un *réseau social* R entre n individus sera représenté par un graphe non orienté dont les sommets sont les individus, et où les arêtes représentent les liens d'amitiés entre individus. Ainsi les sommets i et j sont reliés par une arête si et seulement s'il existe un lien d'amitié entre les individus. On peut par ailleurs représenter le réseau R par un dictionnaire `d_r` (dictionnaire d'adjacence) dans lequel :

- les clés sont les n entiers désignant chacun un individu (il y a exactement n clés distinctes) ;
- `d_r[i]` est une liste d'entiers classés par ordre croissant, potentiellement vide, contenant l'ensemble des entiers j correspondants aux individus ayant un lien d'amitié avec l'individu numéro i.

La figure ci-dessous donne l'exemple d'un réseau social et d'une représentation possible sous la forme de dictionnaire. Chaque lien d'amitié entre deux personnes est représenté par un trait entre elles.

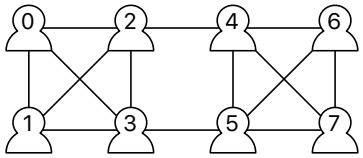
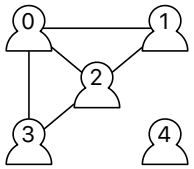


Figure 1. – Réseau à 8 individus ayant 14 liens d'amitié déclarés et une de ses représentations possibles en mémoire sous forme d'un dictionnaire

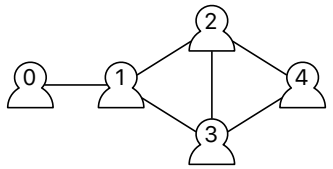
Le dictionnaire associé est dans ce cas :

```
d_r = {0:[1,2,3], 1:[0,2,3], 2:[0,1,3,4], 3:[0,1,2,5], \
↪ 4:[2,5,6,7], 5:[3,4,6,7], 6:[4,5,7], 7:[4,5,6]}
```

1. Donner une représentation sous forme de dictionnaires pour chacun des deux réseaux sociaux ci-dessous.



Réseau A



Réseau B

2. Écrire une fonction `creer_reseau_vide(n:int)->dict` qui crée, initialise et renvoie la représentation sous forme de dictionnaire du réseau à n individus n'ayant aucun lien d'amitié déclaré entre eux.
3. Écrire une fonction `liste_des_amis_de(d_r:dict,i:int)` qui renvoie la liste des amis de l'individu `i` dans le réseau.
4. Écrire une fonction `appartient(L:[int],i:int)->bool` où `L` est une liste d'entiers, `i` un entier, qui renvoie `True` si `i` appartient à la liste `L`, `False` sinon, et qui vérifie les contraintes suivantes:)
   - la fonction ne doit pas utiliser le test d'appartenance à une liste : `i in L`,
   - la fonction ne doit pas utiliser de boucle `for`,
   - la complexité dans le meilleur des cas (*resp.* le pire des cas) est en $O(1)$ (*resp.* $O(\text{len}(L))$).

   On justifiera que les complexités de la fonction proposée sont bien celles demandées.
5. Écrire une fonction `sont_amis(d_r:dict,i:int,j:int)->bool` qui renvoie `True` s'il existe un lien d'amitié entre les individus `i` et `j` dans le réseau `d_r` et renvoie `False` sinon. On pourra utiliser la fonction `appartient` précédente.
6. **6.1)** Écrire une procédure `insertion(L:list, x:float)` qui étant donnée une liste `L` triée dans l'ordre croissant, insère `x` dans `L` afin qu'elle reste triée dans l'ordre croissant. Par exemple, si `L = [1, 3, 4]`, `insertion(L, 2)` modifiera `L` en `[1, 2, 3, 4]`, alors que `insertion(L, 10)` modifiera `L` en `[1, 3, 4, 10]`.

   6.2) Écrire une procédure `declare_amis(d_r:dict,i:int,j:int)->None` qui modifie le dictionnaire `d_r` pour y ajouter le lien d'amitié entre les individus `i` et `j` *si ce lien n'y figure pas déjà. On veillera à maintenir chaque liste de valeurs triée dans l'ordre croissant.*
7. On souhaite ici décerner une médaille à la (ou les) personne(s) ayant le plus d'amis dans le réseau social. On souhaite donc renvoyer la liste des personnes ayant le plus d'amis. Cela revient donc :
   - à créer une liste `Popu = [0]`,
   - une variable `nb_amis` initialisée à `nb_amis = len(d_r[Popu[0]])`.
   - Puis, de parcourir les clefs du dictionnaire `d_r`, calculer le nombre d'amis `nb` de cette clef, puis effectuer les modifications nécessaires sur `Popu` et `nb_amis`

(en fonction de si `nb` est strictement supérieur à `nb_ami` ou égal).
Par exemple, pour le dictionnaire `d_r` précédent, la fonction devra renvoyer `[2, 3, 4, 5]`. Écrire une fonction `plus_populaire(d_r:dict)->list` qui renvoie cette liste.

Parcours du réseau. On peut appliquer les algorithmes de parcours à ce réseau social, et on désignera par DFS (*resp.* BFS) les algorithmes de parcours en profondeur (*resp.* en largeur). *Dans les deux questions qui suivent, on pourra présenter les parcours sous forme d'un tableau ayant la forme ci-après :*

Sommet en cours de visite	Liste des visités	Pile ou file (à préciser)
…	…	…

Quelques remarques de présentation :

- *Lorsque la liste des sommets visités est complète, on pourra s'autoriser à expliquer la fin des parcours à l'aide d'une phrase pour gagner du temps.*
- *On pourra représenter la file/pile par une simple liste en enfilant/empilant par la droite, en dépilant par la droite et en défilant par la gauche.*

8. Appliquer à la main le parcours DFS au graphe présenté en Figure 1, à partir du sommet $i = 4$, et déterminer la liste `L_DFS` des sommets visités dans l'ordre de leur visite. On utilisera la même convention de sélections de sommets à visiter que dans le cours pour le parcours DFS, *i.e.* qui dépend du dictionnaire `d_r`.

9. Appliquer à la main le parcours BFS au graphe présenté en Figure 1, à partir du sommet $i = 4$, et déterminer la liste `L_BFS` des sommets visités dans l'ordre de leur visite. On utilisera la même convention de sélections de sommets à visiter que dans le cours pour le parcours BFS, *i.e.* qui dépend du dictionnaire `d_r`.

On donne ci-dessous une fonction (incomplète) qui permet de réaliser le parcours `BFS` au graphe associé au réseau à partir d'un sommet `v`.

```
def bfs(d_r:dict, visited:dict, v:int):
    q = deque()
    lst_visited = []
    q.append(v)
    while len(q) > 0:
        w = q.popleft()
        if not visited[w]:
            ____________________
            ____________________
            for u in  _______ :
                ____________________
    return lst_visited
```

Cette fonction prend pour argument :

- `d_r`, dictionnaire d'adjacence du réseau,
- `visited`, dictionnaire dont chaque clé `s` est un sommet du graphe avec pour valeur associée `False` si `s` n'a pas encore été visité, et `True` sinon,
- `v`, sommet de départ pour le parcours.

Elle utilise en outre une structure de données de type `deque` utilisée en tant que file, on rappelle que l'on défile un sommet avec la commande `q.popleft()` et que l'on enfile avec la commande `q.append(..)`. Pour réaliser un premier parcours à partir d'un sommet `v1` et affecter la liste des sommets visités à une liste `l_v`, on pourra par exemple écrire les deux instructions suivantes :

```
visited = {s:False for s in d_r}
l_v = bfs(d_r, visited, v1)
```

10. Recopier et compléter la fonction proposée, afin qu'elle renvoie la liste des sommets visités selon le parcours BFS.

Séparation du réseau. Une première méthode pour séparer le réseau en groupes d'individus consiste à déterminer quelles sont les composantes connexes du graphe associé au réseau. Ainsi les individus appartenant à deux composantes connexes distinctes n'auront aucun lien d'amitié entre eux.

11. Expliquer précisément ce que renvoie la fonction ci-dessous.

```
def mystere(visited : dict)->int:
    L = [i for i in visited if visited[i] == True]
    n = len(L)
    M = [0]*(n+1)
    for e in L :
        if e <= n :
            M[e] = 1
    i = 0
    while M[i] == 1 :
        i += 1
    return i
```

12. En utilisant la fonction précédente, en déduire le script d'une fonction `comp_connexes(d_r:dict)->list` qui prend comme argument `d_r` le dictionnaire du graphe associé au réseau social, et qui renvoie une liste constituée des listes des sommets formant les différentes composantes connexes du réseau. Par

exemple, la fonction `comp_connexes` appliquée au réseau A du début du sujet devra renvoyer la liste `[[0,1,2,3],[4]]`.

En pratique, les individus présents sur un réseau social sont tellement interconnectés que les graphes associés sont connexes, et la méthode précédente ne peut être utilisée. On présente par la suite une autre méthode qui permet de réaliser une coupe minimale du réseau (*cf.* introduction), méthode qui nécessite d'utiliser une autre représentation des liens entre individus d'un réseau.

Dans cette nouvelle représentation, les individus sont toujours numérotés de 0 à $n-1$ où n est le nombre total d'individus, et chaque lien d'amitié entre deux individus i et j est représenté 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 (à chaque lien correspond une seule liste).

Plus précisément, le réseau social R entre n individus sera représenté par une liste `lst_r` à deux éléments où :

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

Pour le réseau présenté en Figure 1, on aura donc le représentation suivante :

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

On désire créer une fonction qui fournit la représentation du réseau sous la forme de la liste `lst_r`, à partir de la représentation sous la forme d'un dictionnaire `d_r`. Pour cela, il faut créer une liste `liens` qui correspond à `lst_r[1]`, dans laquelle chaque lien apparait une seule fois. On propose la méthode suivante :

- initialiser une liste `liens` à la liste vide,
- pour chaque sommet `s` du graphe, examiner chacun des voisins `v` de `s`, et, si `v > s`, ajouter le lien `[s, v]` à `liens`.

13. Écrire une fonction `dict_2_lst(d_r:dict)->list` qui renvoie la représentation du réseau sous la forme d'une liste `[n, liens]` à partir du dictionnaire `d_r` qui représente ce même réseau.

Les parties suivantes s'intéressent à cette deuxième méthode permettant d'obtenir une coupe minimale du réseau.

PARTIE II — PARTITIONS

Une *partition* en k groupes d'un ensemble A à n éléments consiste en k sous-ensembles disjoints non-vides $A_1, \dots, A_k$ de A dont l'union est A, c'est-à-dire 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 $A = [\![0\,,\,5]\!]$.

L'objet de cette partie est d'implémenter une structure de données efficace pour coder des partitions de $[\![0\,,\,n-1]\!]$.

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'est-à-dire que le représentant du groupe est bien le parent du parent du parent … (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 2 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. Ce groupe contient également 8, 13 et 15.

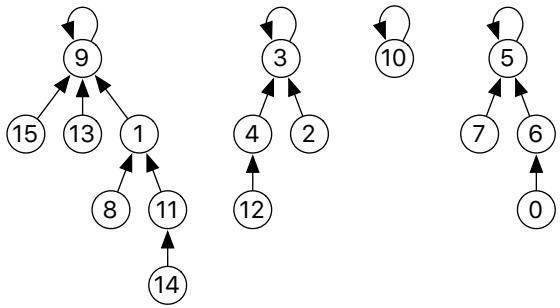


FIGURE 2. – Une représentation filiale de la partition suivante de $[\![0\,,\,15]\!]$ en quatre groupes $\{1,8,9,11,13,14,15\}$, $\{2,3,4,12\}$, $\{10\}$ et $\{0,5,6,7\}$ dont les représentants respectifs sont 9, 3, 10 et 5.

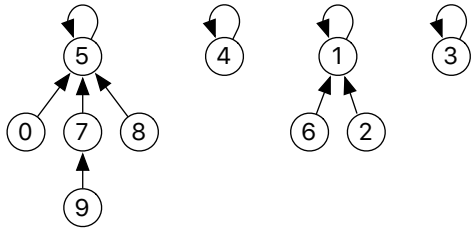
À noter que la représentation n'est pas unique. Si l'on choisit un autre représentant pour un groupe et une autre relation filiale, on aura une autre représentation du groupe.

Pour coder cette structure, on utilise un tableau `parent` à n éléments où la case `parent[i]` contient le numéro du parent de `i`. Par exemple, les valeurs du tableau

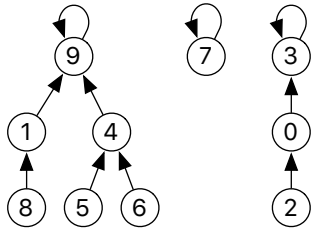
`parent` encodant la représentation filiale donnée dans la Figure 2 sont :

`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

14. Donner les valeurs du tableau `parent` encodant les représentations filiales des deux partitions de $[\![0\,,\,9]\!]$ ci-dessous et préciser les représentants de chaque groupe.



Représentation filiale A



Représentation filiale B

Initialisation. Initialement, chaque élément de $[\![0, n-1]\!]$ est son propre représentant et la partition initiale consiste en n groupes contenant chacun un individu. Ainsi, initialement, `parent[i] = i` pour tout $i \in [\![0\,,\,n-1]\!]$.

15. Écrire une fonction `creer_partition_en_singletons(n:int)->list` qui crée et renvoie la liste correspondant au tableau `parent` à `n` éléments dont les valeurs sont initialisées de sorte à encoder la partition de $[\![0\,,\,n-1]\!]$ en n groupes d'un seul élément.

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 $[\![0\,,\,5]\!]$ donnée en exemple au tout début de cette partie donnera la partition en deux groupes $A_2 = \{0,4,5\}$ et A_4 où $A_4 = A_1 \cup A_3 = \{1,2,3\}$.

16. Écrire une fonction `representant(parent:int,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`. *On pourra recopier et compléter le code ci-après :*

```
def representant(parent:list,i:int)->int:
    j = i
    while ___________ != ___________ :
        j = ___________
    return ___________
```

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.

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.

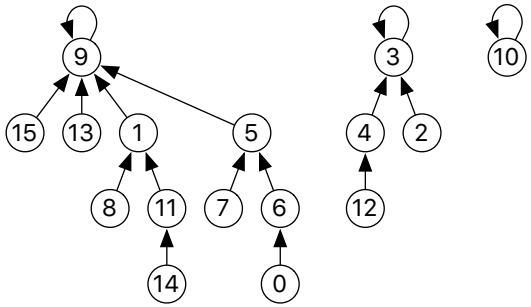


FIGURE 3. – Représentation filiale obtenue après la fusion des groupes contenant respectivement 6 et 14 de la Figure 2.

17. Écrire une procédure `fusion(parent:list,i:int,j:int)->None` qui modifie le tableau parent pour fusionner les deux groupes contenant `i` et `j` respectivement.

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

18. Proposer une suite de $(n-1)$ fusions à partir de la partition initiale en n singletons de $[\![0\,,\,n-1]\!]$, donnant en sortie une partition en un seul groupe, dont le nombre d'opérations est en $O(n^2)$.

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)`, on modifie le tableau `parent` de façon à ce que chaque ancêtre (c'est-à-dire parent de parent …de parent) de `i` (et `i` compris), ait pour parent direct `p`. Noter bien que même si un appel à `representant(parent, i)` renvoie le représentant de `i`, elle modifiera également le tableau `parent` avec cette nouvelle version.

Si l'on reprend l'exemple de la Figure 2, 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 4.

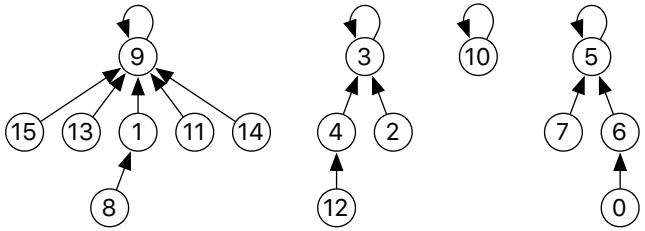


FIGURE 4. – Résultat de la compression depuis 14 dans la représentation filiale de la Figure 2.

19. Écrire une fonction **récursive** `representant_bis(parent, i)`, qui calcule le représentant de `i` et qui 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.

20. Afin d'afficher de manière lisible la partition codée par un tableau `parent`, on souhaite calculer un dictionnaire dont les clefs sont les représentants, et les valeurs les listes des éléments du groupe associé au représentant. Une sortie *possible* pour le tableau `parent` correspondant à la Figure 2 serait :

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

Écrire une fonction `dico_des_groupes(parent:list)->dict` qui renvoie un tel dictionnaire associé au tableau `parent`.

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.

MARQUAGE DES LIENS On reprend la représentation du réseau à l'aide de `lst_r`, introduite en première partie, dans laquelle la liste `lst_r[1]` est la liste des liens d'amitiés entre individus. On appellera à nouveau `liens` cette liste (`lst_r[1]` et `liens` représentent la même liste). Dans la liste `liens`, on distingue certains liens, appelés « *liens marqués* », des autres liens. L'intérêt du marquage est précisé ci-dessous. Pour marquer un lien, on pourra le positionner à la fin de la liste `liens`. Ainsi, pour une liste `liens` contenant m liens,

- si un seul lien a été marqué, la liste `liens[:m-1]` contient les $m-1$ liens non marqués restants,
- si deux liens ont été marqués, la liste `liens[:m-2]` contient les $m-2$ liens non marqués restants,
- si k liens ont été marqués (avec $k \leq m$), la liste `liens[:m-k]` contient les $m-k$ liens non marqués restants.

On peut donc travailler sur les liens non marqués en considérant ceux placés au début de la liste.

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 $[\![0\,,\,n-1]\!]$.
- 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, noté `[i,j]`, uniformément au hasard parmi les liens non-marqués du réseau ;
  - ◇ 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 $k \geqslant 3$ groupes, faire $k-1$ fusions pour obtenir deux groupes.

Sortie : renvoyer la partition P.
La Figure 5 (en fin de sujet) présente une exécution possible de cet algorithme randomisé sur le réseau de la Figure 1.

21. Écrire une fonction `coupe_minimum_randomisee(lst_r:list)->list` qui renvoie le tableau `parent` correspondant à la partition calculée par l'algorithme ci-dessus.

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`?

22. Écrire une fonction `taille_coupe(lst_r:list, parent:list)->int` qui calcule le nombre de liens entre les différents groupes de la partition représentée par `parent` dans le réseau `lst_r`.

On peut démontrer que cet algorithme renvoie une coupe de taille minimum avec une probabilité supérieure à $1/n$, ce qui fait que sur n exécutions indépendantes de cet algorithme on a une coupe optimale avec une probabilité supérieure à $1/\mathrm{e} \approx 0.36787\ldots$.

La structure de données filiale avec compression pour les partitions est particulièrement efficace aussi bien en pratique qu'en théorie. En effet, la complexité de k opérations est de $O(k\alpha(k))$ opérations élémentaires où $\alpha(k)$ est l'inverse de la fonction d'ACKERMANN, une fonction qui croit extrêmement lentement vers l'infini (par exemple $\alpha(1080) = 5$).

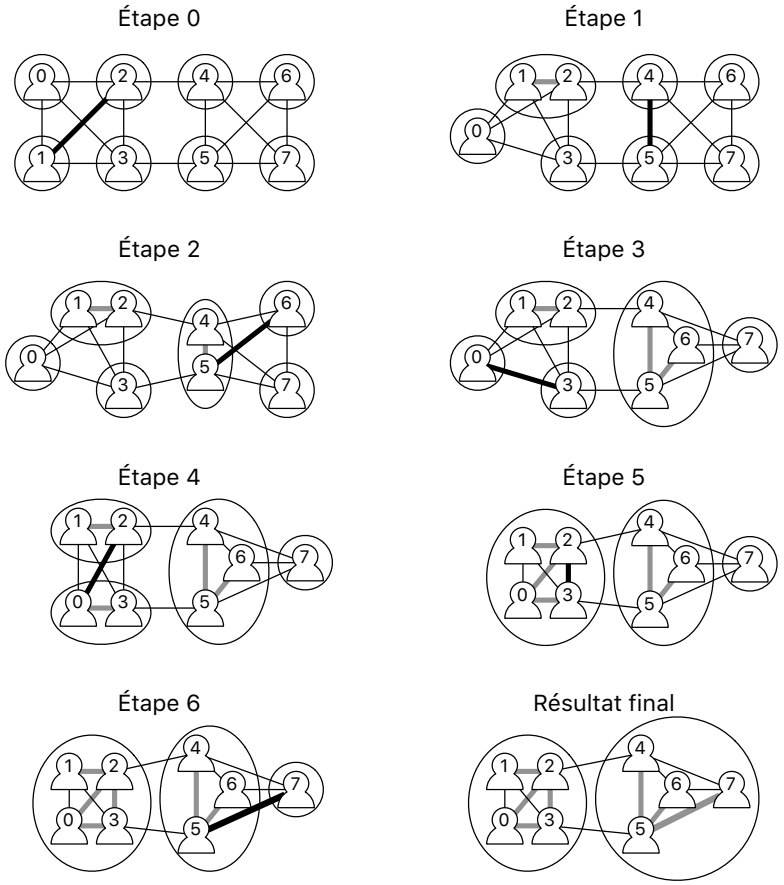


FIGURE 5. – Une exécution de l'algorithme randomisé sur le réseau de la Figure 1 où les liens sélectionnés aléatoirement sont dans l'ordre : [2,1], [4,5], [6,5], [0,3], [2,0], [3,2] et [5,7]. Les liens représentés en noir épais sont les liens sélectionnés au hasard à l'étape courante ; les liens épais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

Correction
du Devoir Surveillé d'ITC N°2
MPSI & PCSI

Solution

1.
```
reseau_A = {0 : [1, 2, 3], 1 : [0, 2], 2 : [0, 1, 3], 3 : [0, \
↪ 2], 4 : []}
reseau_B = {0 : [1], 1 : [0, 2, 3], 2 : [1, 3, 4], 3 : [1, 2, \
↪ 4], 4 : [2, 3]}
```

2.
```
def creer_reseau_vide(n:int)->dict:
    D = {}
    for i in range(n):
        D[i] = []
    return D
```
```
>>> creer_reseau_vide(10)
{0: [], 1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 8: \
↪ [], 9: []}
```
On peut aussi coder directement le dictionnaire par compréhension.
```
def creer_reseau_vide(n:int)->dict:
    return {i : [] for i in range(n)}
```
```
>>> creer_reseau_vide(10)
{0: [], 1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 8: \
↪ [], 9: []}
```

3.
```
def liste_des_amis_de(d_r:dict,i:int):
    return d_r[i]
```
```
>>> liste_des_amis_de(reseau_A, 1)
[0, 2]
```

4. C'est l'exemple typique d'utilisation d'un `while`.
```
1 def appartient(L:[int],i:int)->bool:
2     j = 0
3     n = len(L)
4     # recherche de i dans L
5     while j < n and L[j] != i:
6         j += 1
7     return j < n
```
```
>>> appartient([1, 2, 3], 2)
True
>>> appartient([1, 2, 3], 4)
False
```
Calculons à présent la complexité.
- Nous avons dans le meilleur des cas (l'élément i est situé en position 0) : 2 affectations (lignes 2/3), 2 tests en entrée du `while` et 1 test final (ligne 7). La complexité est $C_{\text{meilleure}}(n) = 5 = \boxed{O(1)}$.
- Nous avons dans le pire des cas (l'élément i n'est pas présent dans `L`) : 2 affectations (lignes 2/3), 2 tests en entrée du `while` et 2 opérations répétés n fois, puis 1 test de sortie de `while` et enfin 1 test final (ligne 7). La complexité est $C_{\text{pire}}(n) = 2+4n+1+1 = \boxed{O(n)}$.

5. Il s'agit simplement d'opérer à un test d'appartenance.
```
def sont_amis(d_r:dict,i:int,j:int)->bool:
    return appartient(d_r[i], j)
```
```
>>> sont_amis(reseau_A, 1, 2)
True
>>> sont_amis(reseau_A, 1, 3)
False
```

6. **6.1)**
```
def insertion(L:list, x:float)->None:
    i = 0
    while (i < len(L)) and (L[i] < x):
        i += 1
    # insertion a la bonne place
    L.insert(i, x)
```
```
>>> L = [1, 2, 4]
>>> insertion(L, 3)
>>> L
[1, 2, 3, 4]
>>> L = [1, 2, 4]
>>> insertion(L, 10)
>>> L
[1, 2, 4, 10]
```

6.2)
```
def declare_amis(d_r:dict,i:int,j:int)->None:
    if not sont_amis(d_r, i, j):
        insertion(d_r[i], j)
        insertion(d_r[j], i)
```
```
>>> declare_amis(reseau_B, 3, 4)
>>> reseau_B # pas de modif
{0: [1], 1: [0, 2, 3], 2: [1, 3, 4], 3: [1, 2, 4], 4: [2, \
↪ 3]}
>>> declare_amis(reseau_B, 3, 0)
```

```
>>> reseau_B # modif
{0: [1, 3], 1: [0, 2, 3], 2: [1, 3, 4], 3: [0, 1, 2, 4], \
↪ 4: [2, 3]}
```

```
d_r = {0:[1,2,3], 1:[0,2,3], 2:[0,1,3,4], 3:[0,1,2,5], \
↪ 4:[2,5,6,7], 5:[3,4,6,7], 6:[4,5,7], 7:[4,5,6]}

def plus_populaire(d_r:dict)->list:
    Popu = [0]
    nb_amis = len(d_r[Popu[0]])
    for i in d_r:
        nb = len(d_r[i])
        if nb > nb_amis:
            Popu = [i] # reset : on améliore str le précédent \
            ↪ max
            nb_amis = nb
        elif nb == nb_amis:
            Popu.append(i) # i a autant d'amis que le \
            ↪ précédent max
    return Popu
>>> plus_populaire(d_r)
[2, 3, 4, 5]
```

7.

8. Parcours DFS.

Sommet en cours de visite	Liste des visités	Pile
4	[4]	[2, 5, 6, 7]
7	[4, 7]	[2, 5, 6, 5, 6]
6	[4, 7, 6]	[2, 5, 6, 5, 5]
5	[4, 7, 6, 5]	[2, 5, 6, 5, 3]
3	[4, 7, 6, 5, 3]	[2, 5, 6, 5, 0, 1, 2]
2	[4, 7, 6, 5, 3, 2]	[2, 5, 6, 5, 0, 1, 0, 1]
1	[4, 7, 6, 5, 3, 2, 1]	[2, 5, 6, 5, 0, 1, 0, 0]
0	[4, 7, 6, 5, 3, 2, 1, 0]	[2, 5, 6, 5, 0, 1, 0]

La suite consiste simplement à dépiler les sommets successivement, mais comme tous les sommets ont été visités cela n'altère plus la liste des sommets visités. On a donc au final : `L_DFS = [4, 7, 6, 5, 3, 2, 1, 0]`.

9. Parcours BFS.

Sommet en cours de visite	Liste des visités	File
4	[4]	[2, 5, 6, 7]
2	[4, 2]	[5, 6, 7, 0, 1, 3]
5	[4, 2, 5]	[6, 7, 0, 1, 3, 3, 6, 7]
6	[4, 2, 5, 6]	[7, 0, 1, 3, 3, 6, 7, 7]
7	[4, 2, 5, 6, 7]	[0, 1, 3, 6, 7, 7]
0	[4, 2, 5, 6, 7, 0]	[0, 1, 3, 3, 6, 7, 7, 1, 3]
0	[4, 2, 5, 6, 7, 0]	[1, 3, 3, 6, 7, 7, 1, 3]
1	[4, 2, 5, 6, 7, 0, 1]	[3, 3, 6, 7, 7, 1, 3, 3]
3	[4, 2, 5, 6, 7, 0, 1, 3]	[3, 6, 7, 7, 1, 3, 3]

La suite consiste simplement à défiler les sommets successivement, mais comme tous les sommets ont été visités cela n'altère plus la liste des sommets visités. `L_BFS = [4, 2, 5, 6, 7, 0, 1, 3]`.

10.
```
def bfs(d_r:dict, visited:dict, v:int):
    q = deque()                          # création d'une file vide
    lst_visited = []                     # liste des sommets visités
    q.append(v)                          # enfilement du sommet de départ
    while len(q) > 0:                    # parcours des sommets non visités
        w = q.popleft()                  # défilement du sommet de q
        if not visited[w]:               # si w non déjà visité
            visited[w] = True            # w marqué comme visité
            lst_visited.append(w)        # ajout de w à la liste des sommets visités
            for u in d_r[w]:             # parcours des voisins u de w
                if not visited[u]:       # si u non déjà visité
                    q.append(u)          # enfilement de u
    return lst_visited
```

On peut tester sur le graphe des deux questions précédentes.

```
>>> bfs(d_r, {s:False for s in d_r}, 4) # on retrouve bien la \
↪ même liste que précédemment
```

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

11. La première ligne crée la liste des sommets qui ont déjà été visités (selon `visited`). Notons n = `len(L)`. La fonction `mystere` crée ensuite une liste de zéros de taille $n+1$, puis place un 1 en position $i \in [\![0, n]\!]$ lorsque i est présent dans `L`, la boucle `while` calcule ensuite le 1er entier $i \in [\![0, n]\!]$ non présent dans `L` (il existe forcément, puisqu'il y a un zéro en trop). En résumé, la fonction `mystere` renvoie le 1er (par ordre croissant) sommet non visité (ou 1 sommet fictif n s'ils l'ont tous été).

```
def mystere(visited : dict)->int:
    L = [i for i in visited if visited[i] == True]
    n = len(L)
    M = [0]*(n+1)
    for e in L :
        if e <= n :
            M[e] = 1
    i = 0
    while M[i] == 1 :
        i += 1
    return i
>>> mystere({1:True, 2:True})
0
>>> mystere({1:True, 2:False, 3:True})
0
```

12. On se sert de la fonction précédente afin d'éviter l'utilisation de `remove` comme en TP. On commence par calculer la composante connexe de 0, puis on recherche à l'aide `mystere` le 1er entier disponible afin de lancer le 2ème calcul. L'algorithme s'arrête lorsque `mystere` renvoie `len(L)`.

```
def comp_connexes(d_r:dict)->list:
    comp = []
    n = len(d_r)
    visited = {s:False for s in d_r}
    comp.append(bfs(d_r, visited, 0))
    j = mystere(visited)
    while j != n:
        comp.append(bfs(d_r, visited, j))
        j = mystere(visited)
    return comp
>>> comp_connexes(reseau_A)
[[0, 1, 2, 3], [4]]
```

13.

```
def dict_2_lst(d_r:dict)->list:
    n = len(d_r)
    liens = []
    for s in d_r:
        vois_s = d_r[s]
        for v in vois_s:
            if v > s:
                liens.append([s, v])
    return [n, liens]
>>> dict_2_lst(d_r)
[8, [[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3], [2, 4], \
↪ [3, 5], [4, 5], [4, 6], [4, 7], [5, 6], [5, 7], [6, 7]]]
```

14. La représentation filiale de A est le tableau suivant dans lequel les représentants des quatre groupes sont 1, 3, 4, 5.

i	0	1	2	3	4	5	6	7	8	9
parent[i]	5	1	1	3	4	5	1	5	5	7

La représentation filiale de B est le tableau suivant dans lequel les représentants des trois groupes sont 3, 7, 9.

i	0	1	2	3	4	5	6	7	8	9
parent[i]	3	9	0	3	9	4	4	7	1	9

15. Ici chaque sommet est son propre parent, on déduit la fonction ci-après.

```
def creer_partition_en_singletons(n:int)->int:
    return [i for i in range(n)]
```

16.

```
def representant(parent:list,i:int)->int:
    j = i
    while parent[j] != j:
        j = parent[j]
    return j
>>> parent_A = [5, 1, 1, 3, 4, 5, 1, 5, 5, 7]
>>> representant(parent_A, 9)
5
```

Dans le pire des cas, cette fonction parcourt entièrement le tableau `parent` avant de trouver le représentant du groupe auquel appartient `i`. Donc sa complexité est en $O(n)$. La complexité maximale est réalisée lorsque $[\![0, n-1]\!]$ est partitionné en un seul groupe et qu'on cherche le représentant de 0. Cet exemple correspond au tableau `[1, 2, ... , n-1, n-1]`.

17.

```
def fusion(parent:list, i:int, j:int)->None:
    p = representant(parent, i)
```

```
    q = representant(parent, j)
    parent[p] = q
```

18. Considérons la suite de fusions :

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

Pour tout i, l'appel `fusion(parent(0, i))` est en $O(i)$, donc la complexité de cette suite d'appels $C(n)$ est :

$$C(n) = \sum_{i=1}^{n-1} O(i) = O(n^2).$$

La complexité est quadratique.

19.

```
def representant_bis(parent:list, i:int)->int:
    if parent[i] != i:
        # on est pas arrivé à la racine
        j = representant_bis(parent, parent[i])
        # ici le représentant a été trouvé
        parent[i] = j
    return parent[i]
```

```
>>> parent_A = [5, 1, 1, 3, 4, 5, 1, 5, 5, 7]
>>> representant_bis(parent_A, 9)
5
>>> parent_A
[5, 1, 1, 3, 4, 5, 1, 5, 5, 5]
```

On voit bien ici que le tableau `parent_A` a été correctement modifié : tout le groupe a pour ancêtre direct 5.

20.

```
def dico_des_groupes(parent:list)->dict:
    n = len(parent)
    dico_gr = {}
    for i in range(n):
        rep_i = representant(parent, i)
        if rep_i == i and i not in dico_gr:
            dico_gr[i] = []
        elif rep_i != i and rep_i not in dico_gr:
            dico_gr[rep_i] = [i]
        elif rep_i != i and rep_i in dico_gr:
            dico_gr[rep_i].append(i)
    return dico_gr
```

```
>>> parent_B = [3, 9, 0, 3, 9, 4, 4, 7, 1, 9]
>>> dico_des_groupes(parent_B)
{3: [0, 2], 9: [1, 4, 5, 6, 8], 7: []}
```

21.

```
def coupe_minimum_randomisee(lst_r):
    n = lst_r[0]
    parent = creer_partition_en_singletons(n)
    nb_groupes = n
    nb_liens = len(lst_r[1]) # nb liens non marqués
    while nb_groupes > 2 and nb_liens > 0:
        nb_liens -= 1
        k = rd.randint(0, nb_liens)
        [i, j] = lst_r[1][k]
        ri = representant(parent, i)
        rj = representant(parent, j)
        if ri != rj:
            fusion(parent, ri, rj)
            nb_groupes -= 1
        lst_r[1][k], lst_r[1][nb_liens] = lst_r[1][nb_liens], \
↪ lst_r[1][k] # marquage
    if nb_groupes > 2:
        r0 = representant(parent, 0)
        i = 1
        while nb_groupes > 2:
            ri = representant(parent, i)
            if r0 != ri:
                fusion(parent, r0, ri)
                nb_groupes -= 1
            i += 1
    return parent

lst_r = [8, [[0, 1], [1, 3], [2, 3], [0, 2], [0, 3], [1, 2], \
↪ [4, 5], [5, 7], [6, 7], [4, 6], [4, 7], [5, 6], [2, 4], [3, \
↪ 5]]]
```

```
>>> parent = coupe_minimum_randomisee(lst_r)
>>> parent
[1, 5, 5, 5, 5, 5, 7, 7]
```

La variable `nb_liens` 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 `lst_r[1]` d'indices `k` et `nb_liens`. 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 $O(n\alpha(n))$. En définitive, la complexité totale de cette fonction est un $\boxed{O((m+n)\alpha(n))}$.

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

```
def taille_coupe(lst_r, parent):
    nb_liens = 0
    for [i, j] in lst_r[1]:
        if representant(parent, i) != representant(parent, j):
            nb_liens += 1
    return nb_liens
```

```
>>> taille_coupe(lst_r, parent)
4
```