SAE 102 : Communautés dans un réseau

Réaliser par :
Mohammed HACHIM
Christophe LIN HUANG

⚠️ Les questions comparatives 2 et 11 sont en rouge !

In [1]:
```
from comu import *
from test_comu import *
from time import *
```

Questions 1 :
Définir une fonction **cree_reseau** semblable à la fonction dico_reseau de la SAE 101 : elle prend en paramètre un tableau de couples d'amis qu'elle traverse une seule fois, et pour chaque couple (a,b) ajoute b à la liste des amis de a, et a à la liste des amis de b. Elle renvoie le réseau correspondant.

Voici le test unitaire de la fonction **cree_reseau** :

In [2]:
```
test_cree_reseau()
```

```
La fonction cree_reseau est correct.
```

Question 2 :
Comparer théoriquement et pratiquement les fonctions **cree_reseau** et **dico_reseau**. (Cette dernière construit le réseau en cherchant d'abord les amis d'Alice, puis de Bob, etc.).

In [3]:
```
# Nous allons mesurer la différence de temps d'exécution du code.
amis = ["Bob","Alice","Bob","Dan","Bob","Carl","Alice","Dan",]
nb_tests = 10000
total_time_dico = 0
i = 0
while i < nb_tests:
    tac = time()
    dico_reseau(amis)
    tic = time()
    total_time_dico += (tic - tac)
    i += 1

total_time_cree = 0
i = 0
while i < nb_tests:
    tac = time()
    cree_reseau(amis)
    tic = time()
    total_time_cree += (tic - tac)
    i += 1

moyenne_dico = (total_time_dico / nb_tests) * 1000
```

```
moyenne_cree = (total_time_cree / nb_tests) * 1000

print("le temps d'exécution de la fonction dico_reseau  : " ,round(1000 * moyenn
print("le temps d'exécution de la fonction cree_reseau  : " ,round(1000 * moyenn
```

```
le temps d'exécution de la fonction dico_reseau  :  8.69 µs
le temps d'exécution de la fonction cree_reseau  :  1.83 µs
```

On peut observer que la fonction **dico_reseau** commence par extraire la liste des personnes uniques, puis parcourt l'ensemble de la liste pour identifier les paires d'amis. En revanche, la fonction **cree_reseau** ajoute directement les paires d'amis en un seul passage. La complexité de **dico_reseau** est **quadratique** en raison de la présence d'une boucle **while** imbriquée dans une autre boucle **while**, tandis que la fonction **cree_reseau** n'a qu'une seule boucle **while**, ce qui lui confère une **complexité linéaire**. Sur le plan pratique, on remarque que la fonction **dico_reseau** repose sur trois fonctions, tandis que **cree_reseau** en utilise une seule, ce qui explique l'écart significatif dans les temps d'exécution : **dico_reseau** prend entre 4µs et 8µs, tandis que **cree_reseau** ne prend que 0,5 à 2 µs, soit au moins 2 fois plus rapide.

Question 3 :
Définir la fonction **liste_personnes** qui à partir d'un réseau renvoie la liste des membres du réseau dans un tableau. Avec le réseau donné en exemple, elle doit renvoyer ["Alice", "Bob", "Carl","Dan"] (l'ordre n'a pas d'importance).

Voici le test unitaire de la fonction **liste_personnes** :

In [4]:
```
test_liste_personnes()
```

```
La fonction liste_personnes est correct.
```

Question 4 :
Définir la fonction **sont_amis** qui teste si deux membres du réseau sont amis. Pour "Alice" et "Bob"de l'exemple, elle renvoie True. Pour "Alice" et "Carl" elle renvoie False.

Voici le test unitaire de la fonction **sont_amis** :

In [5]:
```
test_sont_amis()
```

```
La fonction sont_amis est correct.
```

Question 5 :
Définir la fonction **sont_amis_de** qui prend une personne et un groupe (et le réseau), et teste si tous les membres du groupe sont amis de cette personne. Pour "Alice" et le groupe ["Bob", "Dan"] elle renvoie True, et False pour "Alice" et le groupe ["Bob", "Carl"].

Voici le test unitaire de la fonction **sont_amis_de** :

In [6]:
```
test_sont_amis_de()
```

```
La fonction sont_amis_de est correct.
```

Question 6 :
Définir la fonction **est_comu** qui teste si un groupe du réseau est une communauté.

(Groupe et réseau doivent être en paramètre.) Par exemple, c'est le cas de ["Alice", "Bob", "Dan"] mais pas de ["Alice", "Bob", "Carl"].

Voici le test unitaire de la fonction **est_comu** :

In [7]:
```
test_est_comu()
```

```
La fonction est_comu est correct.
```

Question 7 :
Dans un groupe on peut trouver une communauté maximale, au sens où personne ne peut y être ajouté. Définir la fonction **comu** qui part d'une communauté vide et examine successivement les membres du groupe passé en paramètre. Chacun est ajouté à la communauté s'il est ami de tous les membres de la communauté déjà créée. Avec le réseau en exemple, elle renvoie :- ["Alice", "Bob", "Dan"] pour le groupe ["Alice", "Bob", "Carl", "Dan"]- ["Carl", "Bob"] pour le groupe ["Carl", "Alice", "Bob", "Dan"]- ["Carl"] pour le groupe ["Carl", "Alice", "Dan"].

Voici le test unitaire de la fonction **comu** :

In [8]:
```
test_comu()
```

```
La fonction comu est correct.
```

Question 8 :
Définir la fonction **tri_popu** qui trie un groupe par popularité (nombre d'amis) décroissante. (Le réseau doit évidemment être en paramètre.) Ainsi le tri du groupe ["Alice", "Bob", "Carl"] donne ["Bob", "Alice", "Carl"].

Voici le test unitaire de la fonction **tri_popu** :

In [9]:
```
test_tri_popu()
```

```
La fonction tri_popu est correct.
```

Question 9 :
Définir la fonction **comu_dans_reseau** qui trie les membres d'un réseau par popularité décroissante, applique l'algorithme de construction de la **question 7** et renvoie la communauté obtenue. Avec le réseau en exemple, elle renvoie ["Bob", "Alice", "Dan"] (l'ordre n'a pas d'importance).

Voici le test unitaire de la fonction **comu_dans_reseau** :

In [10]:
```
test_comu_dans_reseau()
```

```
La fonction comu_dans_reseau est correct.
```

Question 10 :
On construit maintenant une communauté maximale d'une autre façon. Définir la fonction **comu_dans_amis** qui part d'une communauté réduite à une personne et examine ses amis dans l'ordre de popularité décroissante. Comme avant, chacun est ajouté à la communauté s'il est ami de tous les membres de la communauté déjà créée.

Si on part d'une personne des plus populaires, l'algorithme est semblable au précédent (les communautés créées peuvent différer si des personnes ont le même nombre d'amis). En partant d' "Alice" on renvoie ["Alice", "Bob", "Dan"]; en partant de "Carl" on renvoie ["Carl", "Bob"].

Voici le test unitaire de la fonction **comu_dans_amis** :

In [11]:
```
test_comu_dans_amis()
```

```
La fonction comu_dans_amis est correct.
```

Question 11 :

Comparer théoriquement et pratiquement les fonctions **comu_dans_reseau** et **comu_dans_amis** appliquée à une personne des plus populaires (la recherche de la personne la plus populaire sera prise en compte dans la complexité).

In [12]:
```
# Nous allons mesurer la différence de temps d'exécution du code.
reseau =  {"Alice":["Bob", "Dan"],"Bob":["Alice","Carl","Dan"],"Carl":["Bob"],"D

nb_tests = 10000
total_time_reseau = 0
i = 0
while i < nb_tests:
    tac = time()
    comu_dans_reseau(reseau)
    tic = time()
    total_time_reseau += (tic - tac)
    i += 1


total_time_amis = 0
i = 0
while i < nb_tests:
    tac = time()
    comu_dans_amis(reseau, "Alice")
    tic = time()
    total_time_amis += (tic - tac)
    i += 1


moyenne_reseau = (total_time_reseau / nb_tests) * 1000
moyenne_amis = (total_time_amis / nb_tests) * 1000


print("le temps d'exécution de la fonction comu_dans_reseau : " ,round(1000 * mo
print("le temps d'exécution de la fonction comu_dans_amis  : " ,round(1000 * moy
```

```
le temps d'exécution de la fonction comu_dans_reseau :  4.35 µs
le temps d'exécution de la fonction comu_dans_amis  :  1.9 µs
```

Nous pouvons voir que les deux fonctions utilisent d'autres fonctions, comu_dans_reseau utilise 3 fonctions alors que comu_dans_amis utilise que 2, ce qui joue sur l'augmentation de la durée d'exécution (plus d'instructions à exécuter, plus le temps augmente). Ainsi, la fonction comu_dans_amis se focalise sur une personne et son groupe d'amis, donc elle travaille sur une petite partie du réseau, ce qui mène à conclure que sa complexité est faible (quadratique pour le tri des amis et également quadratique pour la vérification des

connexions). Puisqu'elle ne trie et vérifie que les amis d'une seule personne, elle est plus rapide et moins compliquée que comu_dans_reseau, qui parcourt tout le réseau et l'analyse, ce qui inclut un tri de toutes les personnes du réseau (quadratique pour le tri de tous le reseau) avant de construire une communauté. Les deux fonctions utilisent un tri, qui est assez lent car il compare les éléments deux par deux (complexité quadratique). Cependant, comu_dans_reseau applique ce tri à l'ensemble du réseau, ce qui prend beaucoup plus de temps, alors que comu_dans_amis se limite à trier uniquement un petit groupe d'amis, ainsi Les tests pratiques montrent que comu_dans_amis est beaucoup plus rapide entre 1 µs et 3 µs que comu_dans_reseau, qui varie entre 6 µs et 11 µs. Cette différence s'explique brièvement par le fait que comu_dans_amis traite un petit sous-ensemble du réseau, tandis que comu_dans_reseau effectue des calculs plus lourds (quadratiques pour les vérifications dans tout le réseau) en parcourant tout le réseau.

Question 12 :
Définir la fonction **comu_max** qui applique **comu_dans_amis** à tous les membres d'un réseau et renvoie la plus grande communauté trouvée. Avec le réseau en exemple elle renvoie ["Alice", "Bob", "Dan"] (peu importe l'ordre).

Voici le test unitaire de la fonction **comu_max** :

In [13]:
```
test_comu_max()
```

```
La fonction comu_max est correct.
```

Fichiers Python combinés

```
### comu.py ###

#Fonction de la question 1
def cree_reseau(amis):
    """
      Construit un réseau d'amis sous forme de dictionnaire à partir d'une liste de paires.

    Chaque paire dans la liste `amis` représente deux personnes qui sont amies.
    La fonction retourne un dictionnaire où chaque clé est une personne, et la valeur
    correspond à une liste de ses amis.

    Paramètre:
        amis (type: list): Liste contenant des paires d'amis sous forme [a1, b1, a2, b2, ...].

    Return:
         (type: dict): Un dictionnaire représentant le réseau d'amis, où chaque clé est une personne
                       et la valeur est une liste de ses amis.
    """
    reseau = {}
    i = 0
    while i < len(amis):
        a = amis[i]
        b = amis[i + 1]
          # Si la personne 'a' n'existe pas encore dans le réseau, l'ajouter avec une liste vide
        if a not in reseau:
            reseau[a] = []
          # Si la personne 'b' n'existe pas encore dans le réseau, l'ajouter avec une liste vide
        if b not in reseau:
            reseau[b] = []
        # Ajouter chaque ami dans la liste des amis de l'autre
        if b not in reseau[a] :
            reseau[a].append(b)
        if a not in reseau[b] :
            reseau[b].append(a)
        i += 2
    return reseau


#fonctions prises de la correction de la première SAE
def personnes(amis):
    """
    retourne le tableau des différentes personnes du tableau amis
    """
    personnes=[]
    i=0
    while i<len(amis):
        if amis[i] not in personnes:
```

```
            personnes.append(amis[i])
        i+=1
    return personnes

def ses_amis(amis, prenom):
    """
    Retourne le tableau des amis de prenom
    """
    ses_amis=[]
    i=0
    while i<len(amis)/2:
        if amis[2*i]== prenom :
            ses_amis.append(amis[2*i+1])
        elif amis[2*i+1]==prenom :
            ses_amis.append(amis[2*i])
        i+=1
    return ses_amis

#version non optimisée
def dico_reseau(amis):
    reseau={}
    # membres du réseau
    pers=personnes(amis)
    # construction du dictionnaire
    i=0
    while i<len(pers):
        reseau[pers[i]]=ses_amis(amis,pers[i])
        i+=1
    return reseau


#Fonction de la question 3
def liste_personnes(reseau):
    """
    Retourne la liste des personnes dans le réseau.

     Cette fonction prend un réseau d'amis sous forme de dictionnaire où chaque clé est
une personne,
    et retourne une liste contenant toutes les personnes du réseau.

    Paramètre:
         reseau (type: dict): Un dictionnaire représentant un réseau d'amis. Chaque clé
est une personne,
                              et la valeur correspond à une liste de ses amis.

    Return :
        (type: list): Une liste contenant toutes les personnes du réseau.
    """
    return list(reseau)


#Fonction de la question 4
```

```
def sont_amis(reseau, personne1, personne2):
    """
    Vérifie si deux personnes sont amies dans un réseau donné.

     Cette fonction prend un réseau d'amis sous forme de dictionnaire, et deux noms de personnes.
    Elle retourne un booléen indiquant si les deux personnes sont amies.

    Paramètre :
        - reseau (type: dict): Un dictionnaire représentant le réseau d'amis. Chaque clé est une personne,
                               et la valeur correspond à une liste de ses amis.
        - personne1 (type: str): Le nom de la première personne.
        - personne2 (type: str): Le nom de la seconde personne.

    Return :
        (type: bool): "True" si les deux personnes sont amies, sinon "False".
    """
    # Vérifie si les deux personnes existent dans le réseau
    if personne1 in reseau and personne2 in reseau:
        # Vérifie si personne2 est dans la liste des amis de personne1
        return personne2 in reseau[personne1]
    return False


#Fonction de la question 5
def sont_amis_de(reseau, personne, groupe):
    """
    Vérifie si une personne est amie avec tous les membres d'un groupe donné.

    Cette fonction prend un réseau d'amis sous forme de dictionnaire, une personne et un groupe (liste de personnes).
     Elle retourne un booléen indiquant si la personne est amie avec tous les membres du groupe.

    Paramètre :
         - reseau (type: dict): Un dictionnaire représentant le réseau d'amis. Chaque clé est une personne,
                               et la valeur correspond à une liste de ses amis.
        - personne (type: str): Le nom de la personne à vérifier.
            - groupe (type: list): Une liste contenant les noms des personnes avec lesquelles on veut vérifier l'amitié.

    Return :
         (type: bool) : "True" si la personne est amie avec tous les membres du groupe, sinon "False".
    """
    # Vérifie si la personne existe dans le réseau
    if personne not in reseau:
        return False
    i = 0
    # Parcourt les membres du groupe
```

```
    while i < len(groupe):
        # Vérifie que chaque membre du groupe est un ami de la personne
        if groupe[i] != personne and groupe[i] not in reseau[personne]:
            return False
        i += 1
    return True


#Fonction de la question 6
def est_comu(reseau, groupe):
    """
    Vérifie si un groupe de personnes forme une communauté dans un réseau.

     Une communauté est définie comme un groupe où chaque personne est amie avec toutes les autres
    membres du groupe.

    Paramètre :
         reseau (type: dict): Un dictionnaire représentant le réseau d'amis. Chaque clé est une personne,
                              et la valeur correspond à une liste de ses amis.
           groupe (type: list): Une liste contenant les noms des personnes formant le groupe.

    Return :
        (type: bool): "True" si le groupe est une communauté (chaque membre est ami avec tous les autres), sinon "False".
    """
    i = 0
    # Parcourt chaque membre du groupe
    while i < len(groupe):
        # Vérifie si chaque personne est amie avec tous les autres membres du groupe
        if not sont_amis_de(reseau, groupe[i], groupe):
            return False
        i += 1
    return True


#Fonction de la question 7
def comu(reseau, groupe):
    """
    Trouve une communauté maximale dans un groupe donné.

    Une communauté maximale est définie comme un sous-ensemble du groupe tel que :
        - Tous les membres de la communauté sont amis entre eux.
        - Aucun autre membre du groupe ne peut être ajouté à cette communauté.

    Paramètre :
         - reseau (type: dict): Un dictionnaire représentant le réseau d'amis. Chaque clé est une personne,
                                et la valeur correspond à une liste de ses amis.
            - groupe (type: list): Une liste contenant les noms des personnes parmi
```

```
lesquelles chercher une communauté.

    Return :
        (type: list): Une liste contenant les membres de la communauté maximale.
    """
    communaute=[]  # La communauté maximale, initialement vide.
    i=0
    # Parcourt chaque personne dans le groupe
    while i<len(groupe):
        personne = groupe[i]
          # Ajoute une personne à la communauté si elle est amie avec tous les membres
déjà présents
        if sont_amis_de(reseau, personne, communaute):
            communaute.append(personne)
        i += 1
    return communaute


#Fonction de la question 8
def tri_popu(reseau, groupe):
    """
    Trie un groupe par popularité (nombre d'amis) décroissante.

    Paramètre :
        reseau (type: dict) : Le réseau des amis.
        groupe (type: list) : Le groupe à trier.

    Return :
        (type: list) : Le groupe trié par popularité décroissante.
    """
    i = 0
    while i < len(groupe) - 1:
        # Comparer l'élément courant avec le suivant
        if len(reseau[groupe[i]]) < len(reseau[groupe[i + 1]]):
            # Échanger si l'élément suivant est plus populaire
            groupe[i],groupe[i+1]=groupe[i+1],groupe[i]
            # Revenir au début pour vérifier les échanges précédents
            i = 0
        else:
            # Passer au prochain élément
            i += 1
    return groupe


#Fonction de la question 9
def comu_dans_reseau(reseau):
    """
    Trie les membres d'un réseau par popularité décroissante et applique
    l'algorithme de la fonction comu pour construire une communauté maximale.

    Paramètre :
         reseau (type: dict): Un dictionnaire représentant le réseau d'amis. Chaque clé
```

```
est une personne,
                            et la valeur correspond à une liste de ses amis.

    Return :
        (type: list): Une liste contenant les membres de la communauté maximale obtenue.
    """
      # Appliquer la fonction de tri pour obtenir les membres triés par popularité
décroissante
    membres_triees = tri_popu(reseau, list(reseau))
    # Appliquer  la fonction comu pour obtenir la communauté maximale
    return comu(reseau, membres_triees)


#Fonction de la question 10
def comu_dans_amis(reseau, personne):
    """
    Crée une communauté maximale à partir d'une personne, en explorant ses amis
    par popularité décroissante (nombre d'amis). Chaque ami est ajouté à la communauté
    s'il est ami avec tous les membres de la communauté déjà présente.

    Paramètre :
        - reseau (type: dict): Le réseau des amis.
         - personne_initiale (type: str): La personne à partir de laquelle commencer la
communauté.

    Return :
        (type: list): La communauté maximale formée.
    """
    # Initialiser la communauté avec la personne de départ
    communaute = [personne]
    # Récupérer les amis de la personne de départ, triés par popularité (nombre d'amis)
    amis = tri_popu(reseau, reseau[personne])
    # Initialiser l'index pour parcourir les amis avec while
    i = 0
    # Utilisation de while pour explorer les amis
    while i < len(amis):
        ami = amis[i]
        # Vérifier si l'ami est ami avec tous les membres de la communauté
        if sont_amis_de(reseau, ami, communaute):
            communaute.append(ami)  # Ajouter l'ami à la communauté
        i += 1  # Passer à l'ami suivant
    return communaute


#Fonction de la question 12
def comu_max(reseau):
    """
        Trouve la plus grande communauté maximale dans un réseau en appliquant
comu_dans_amis à tous les membres.

    Pour chaque personne dans le réseau, la fonction construit une communauté maximale à
partir d'elle
```

```
    en utilisant la fonction comu_dans_amis. Elle compare ensuite toutes les communautés trouvées
    et retourne la plus grande.

    Paramètre :
         reseau (type: dict): Un dictionnaire représentant le réseau d'amis. Chaque clé est une personne,
                              et la valeur correspond à une liste de ses amis.

    Return :
        (type: list): La plus grande communauté maximale trouvée dans le réseau.
    """
    max_communaute = []  # Initialisation de la plus grande communauté
    # Appliquer comu_dans_amis à chaque membre du réseau
    membres = list(reseau)
    i = 0
    while i < len(membres):
        personne = membres[i]
        communaute = comu_dans_amis(reseau, personne)
        # Mettre à jour la plus grande communauté si celle-ci est plus grande
        if len(communaute) > len(max_communaute):
            max_communaute = communaute
        i += 1
    return max_communaute

### test_comu.py ###

from math import *
from comu import *

#test unitaire de la question 1
def test_cree_reseau():
    """
     Cette fonction est un test unitaire de la fonction cree_reseau qui vérifie si la fonction marche correctement.
    """
    amis1 = ["Alice", "Bob", "Alice", "Dan", "Bob", "Carl"]
           reponse1   =   {"Alice":   ["Bob",   "Dan"],"Bob":   ["Alice",   "Carl"],"Dan": ["Alice"],"Carl": ["Bob"]}
    amis2 = []
    reponse2 = {}
    amis3 = ["Alice", "Bob"]
    reponse3 = {"Alice": ["Bob"],"Bob": ["Alice"]}
    amis4 = ["Alice", "Bob", "Alice", "Bob"]



    assert cree_reseau(amis1) == reponse1
    assert cree_reseau(amis2) == reponse2
    assert cree_reseau(amis3) == reponse3


    print("La fonction cree_reseau est correct.")
```

```
#test unitaire de la question 3
def test_liste_personnes():
    """
    Test unitaire pour la fonction liste_personnes.
    Vérifie que la fonction retourne correctement la liste des personnes dans différents
cas.
    """
    reseau1 = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl"],"Dan": ["Alice"],"Carl":
["Bob"]}
    reponse1=["Alice", "Bob", "Dan", "Carl"]
    reseau2 = {}
    reponse2 = []
    reseau3 = {"Alice": []}
    reponse3 = ["Alice"]

    assert liste_personnes(reseau1) == reponse1
    assert liste_personnes(reseau2) == reponse2
    assert liste_personnes(reseau3) == reponse3

    print("La fonction liste_personnes est correct.")


#test unitaire de la question 4
def test_sont_amis():
    """
    Test unitaire pour la fonction sont_amis.
    Vérifie si la fonction identifie correctement les relations d'amitié.
    """
    reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl"],"Dan": ["Alice"],"Carl":
["Bob"]}

    assert sont_amis(reseau, "Alice", "Bob") == True
    assert sont_amis(reseau, "Alice", "Carl") == False
    assert sont_amis(reseau, "Alice", "Gopi") == False
    assert sont_amis(reseau, "Ezzat", "Théo") == False
    assert sont_amis(reseau, "Bob", "Carl") == True

    print("La fonction sont_amis est correct.")


#test unitaire de la question 5
def test_sont_amis_de():
    """
    Test unitaire pour la fonction sont_amis_de.
    Vérifie si la fonction identifie correctement les relations d'amitié dans un groupe.
    """
    reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl", "Dan"],"Dan": ["Alice",
"Bob"],"Carl": ["Bob"]}

    assert sont_amis_de(reseau, "Alice", ["Bob", "Dan"]) == True
```

```
    assert sont_amis_de(reseau, "Alice", ["Bob", "Carl"]) == False
    assert sont_amis_de(reseau, "Eve", ["Bob", "Dan"]) == False
    assert sont_amis_de(reseau, "Alice", []) == True
    assert sont_amis_de(reseau, "Alice", ["Alice", "Bob"]) == True

    print("La fonction sont_amis_de est correct.")


#test unitaire de la question 6
def test_est_comu():
    """
    Test unitaire pour la fonction est_comu.
    Vérifie si la fonction identifie correctement les communautés dans un réseau.
    """
        reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Dan"],"Dan": ["Alice",
"Bob"],"Carl": ["Eve"],"Eve": ["Carl"]}

    assert est_comu(reseau, ["Alice", "Bob", "Dan"]) == True
    assert est_comu(reseau, ["Alice", "Bob", "Carl"]) == False
    assert est_comu(reseau, ["Alice"]) == True
    assert est_comu(reseau, []) == True
    assert est_comu(reseau, ["Carl", "Eve", "Alice"]) == False

    print("La fonction est_comu est correct.")


#test unitaire de la question 7
def test_comu():
    """
    Test unitaire pour la fonction comu.
       Vérifie si la fonction identifie correctement les communautés maximales dans
différents groupes.
    """
     reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl", "Dan"],"Dan": ["Alice",
"Bob"],"Carl": ["Bob"]}
    groupe1 = ["Alice", "Bob", "Carl", "Dan"]
    reponse1 = ["Alice", "Bob", "Dan"]
    groupe2 = ["Carl", "Alice", "Bob", "Dan"]
    reponse2 = ["Carl", "Bob"]
    groupe3 = ["Carl", "Alice", "Dan"]
    reponse3 = ["Carl"]
    groupe4 = []
    reponse4 = []
    groupe5 = ["Alice"]
    reponse5 = ["Alice"]

    assert comu(reseau, groupe1) == reponse1
    assert comu(reseau, groupe2) == reponse2
    assert comu(reseau, groupe3) == reponse3
    assert comu(reseau, groupe4) == reponse4
    assert comu(reseau, groupe5) == reponse5
```

```
    print("La fonction comu est correct.")


#test unitaire de la question 8
def test_tri_popu():
    """
    Test unitaire pour la fonction tri_popu.
    Vérifie si la fonction trie correctement un groupe par popularité décroissante.
    """
     reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl", "Dan"],"Dan": ["Alice",
"Bob"],"Carl": ["Bob"]}
    groupe1 = ["Alice", "Bob", "Carl", "Dan"]
    reponse1 = ["Bob", "Alice", "Dan", "Carl"]
    groupe2 = ["Carl", "Dan"]
    reponse2 = ["Dan", "Carl"]
    groupe3 = []
    reponse3 = []
    groupe4 = ["Alice"]
    reponse4 = ["Alice"]

    assert tri_popu(reseau, groupe1) == reponse1
    assert tri_popu(reseau, groupe2) == reponse2
    assert tri_popu(reseau, groupe3) == reponse3
    assert tri_popu(reseau, groupe4) == reponse4

    print("La fonction tri_popu est correct.")


#test unitaire de la question 9
def test_comu_dans_reseau():
    """
    Test unitaire pour la fonction comu_dans_reseau.
     Vérifie si la fonction trouve correctement la communauté maximale dans un réseau
donné.
    """
     reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl", "Dan"],"Dan": ["Alice",
"Bob"],"Carl": ["Bob"]}
    reponse1 = ["Bob", "Alice", "Dan"]
    resultat1 = comu_dans_reseau(reseau)
    reseau_vide = {}
    reponse2 = []
    resultat2 = comu_dans_reseau(reseau_vide)
    reseau_simple = {"Alice": []}
    reponse3 = ["Alice"]
    resultat3 = comu_dans_reseau(reseau_simple)

    assert resultat1 == reponse1
    assert resultat2 == reponse2
    assert resultat3 == reponse3

    print("La fonction comu_dans_reseau est correct.")
```

```
#test unitaire de la question 10
def test_comu_dans_amis():
    """
    Test unitaire pour la fonction comu_dans_amis.
     Vérifie si la fonction trouve correctement la communauté maximale à partir d'une personne donnée.
    """
     reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl", "Dan"],"Dan": ["Alice", "Bob"],"Carl": ["Bob"]}
    prenom1 = "Alice"
    reponse1 = ["Alice", "Bob", "Dan"]
    resultat1 = comu_dans_amis(reseau, prenom1)
    prenom2 = "Carl"
    reponse2 = ["Carl", "Bob"]
    resultat2 = comu_dans_amis(reseau, prenom2)
    prenom3 = "Hachim"
    reseau["Hachim"] = []
    reponse3 = ["Hachim"]
    resultat3 = comu_dans_amis(reseau, prenom3)

    assert resultat1 == reponse1
    assert resultat2 == reponse2
    assert resultat3 == reponse3

    print("La fonction comu_dans_amis est correct.")


#test unitaire de la question 12
def test_comu_max():
    """
    Test unitaire pour la fonction comu_max.
     Vérifie si la fonction trouve correctement la plus grande communauté maximale dans un réseau.
    """
     reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl", "Dan"],"Dan": ["Alice", "Bob"],"Carl": ["Bob"]}
    reponse1 = ["Alice", "Bob", "Dan"]
    resultat1 = comu_max(reseau)
    reseau_vide = {}
    reponse2 = []
    resultat2 = comu_max(reseau_vide)
    reseau_simple = {"Alice": []}
    reponse3 = ["Alice"]
    resultat3 = comu_max(reseau_simple)

    assert resultat1 == reponse1
    assert resultat2 == reponse2
    assert resultat3 == reponse3
    print("La fonction comu_max est correct.")
```

LES FONCTIONS SOUS VISUAL STUDIO CODE :

```
#Fonction de la question 1
def cree_reseau(amis):
    """
    Construit un réseau d'amis sous forme de dictionnaire à partir
d'une liste de paires.

    Chaque paire dans la liste `amis` représente deux personnes qui
sont amies.
    La fonction retourne un dictionnaire où chaque clé est une
personne, et la valeur
    correspond à une liste de ses amis.

    Paramètre:
        amis (type: list): Liste contenant des paires d'amis sous forme
[a1, b1, a2, b2, ...].

    Return:
        (type: dict): Un dictionnaire représentant le réseau d'amis, où
chaque clé est une personne
                      et la valeur est une liste de ses amis.
    """
    reseau = {}
    i = 0
    while i < len(amis):
        a = amis[i]
        b = amis[i + 1]
        # Si la personne 'a' n'existe pas encore dans le réseau,
l'ajouter avec une liste vide
        if a not in reseau:
            reseau[a] = []
        # Si la personne 'b' n'existe pas encore dans le réseau,
l'ajouter avec une liste vide
        if b not in reseau:
            reseau[b] = []
        # Ajouter chaque ami dans la liste des amis de l'autre
        if b not in reseau[a] :
            reseau[a].append(b)
        if a not in reseau[b] :
            reseau[b].append(a)
```

```
        i += 2
    return reseau


#fonctions prises de la correction de la première SAE
def personnes(amis):
    """
    retourne le tableau des différentes personnes du tableau amis
    """
    personnes=[]
    i=0
    while i<len(amis):
        if amis[i] not in personnes:
            personnes.append(amis[i])
        i+=1
    return personnes

def ses_amis(amis, prenom):
    """
    Retourne le tableau des amis de prenom
    """
    ses_amis=[]
    i=0
    while i<len(amis)/2:
        if amis[2*i]== prenom :
            ses_amis.append(amis[2*i+1])
        elif amis[2*i+1]==prenom :
            ses_amis.append(amis[2*i])
        i+=1
    return ses_amis

#version non optimisée
def dico_reseau(amis):
    reseau={}
    # membres du réseau
    pers=personnes(amis)
    # construction du dictionnaire
    i=0
    while i<len(pers):
        reseau[pers[i]]=ses_amis(amis,pers[i])
        i+=1
    return reseau
```

```python
#Fonction de la question 3
def liste_personnes(reseau):
    """
    Retourne la liste des personnes dans le réseau.

    Cette fonction prend un réseau d'amis sous forme de dictionnaire où chaque clé est une personne,
    et retourne une liste contenant toutes les personnes du réseau.

    Paramètre:
        reseau (type: dict): Un dictionnaire représentant un réseau d'amis. Chaque clé est une personne,
                             et la valeur correspond à une liste de ses amis.

    Return :
        (type: list): Une liste contenant toutes les personnes du réseau.
    """
    return list(reseau)


#Fonction de la question 4
def sont_amis(reseau, personne1, personne2):
    """
    Vérifie si deux personnes sont amies dans un réseau donné.

    Cette fonction prend un réseau d'amis sous forme de dictionnaire, et deux noms de personnes.
    Elle retourne un booléen indiquant si les deux personnes sont amies.

    Paramètre :
        - reseau (type: dict): Un dictionnaire représentant le réseau d'amis. Chaque clé est une personne,
                               et la valeur correspond à une liste de ses amis.
        - personne1 (type: str): Le nom de la première personne.
        - personne2 (type: str): Le nom de la seconde personne.

    Return :
```

```
        (type: bool): "True" si les deux personnes sont amies, sinon
"False".
    """
    # Vérifie si les deux personnes existent dans le réseau
    if personne1 in reseau and personne2 in reseau:
        # Vérifie si personne2 est dans la liste des amis de personne1
        return personne2 in reseau[personne1]
    return False


#Fonction de la question 5
def sont_amis_de(reseau, personne, groupe):
    """
    Vérifie si une personne est amie avec tous les membres d'un groupe
donné.

    Cette fonction prend un réseau d'amis sous forme de dictionnaire,
une personne et un groupe (liste de personnes).
    Elle retourne un booléen indiquant si la personne est amie avec
tous les membres du groupe.

    Paramètre :
        - reseau (type: dict): Un dictionnaire représentant le réseau
d'amis. Chaque clé est une personne,
                               et la valeur correspond à une liste de ses
amis.
        - personne (type: str): Le nom de la personne à vérifier.
        - groupe (type: list): Une liste contenant les noms des
personnes avec lesquelles on veut vérifier l'amitié.

    Return :
        (type: bool) : "True" si la personne est amie avec tous les
membres du groupe, sinon "False".
    """
    # Vérifie si la personne existe dans le réseau
    if personne not in reseau:
        return False
    i = 0
    # Parcourt les membres du groupe
    while i < len(groupe):
        # Vérifie que chaque membre du groupe est un ami de la personne
        if groupe[i] != personne and groupe[i] not in reseau[personne]:
            return False
```

```
        i += 1
    return True


#Fonction de la question 6
def est_comu(reseau, groupe):
    """
    Vérifie si un groupe de personnes forme une communauté dans un
réseau.

    Une communauté est définie comme un groupe où chaque personne est
amie avec toutes les autres
    membres du groupe.

    Paramètre :
        reseau (type: dict): Un dictionnaire représentant le réseau
d'amis. Chaque clé est une personne,
                             et la valeur correspond à une liste de ses
amis.
        groupe (type: list): Une liste contenant les noms des personnes
formant le groupe.

    Return :
        (type: bool): "True" si le groupe est une communauté (chaque
membre est ami avec tous les autres), sinon "False".
    """
    i = 0
    # Parcourt chaque membre du groupe
    while i < len(groupe):
        # Vérifie si chaque personne est amie avec tous les autres
membres du groupe
        if not sont_amis_de(reseau, groupe[i], groupe):
            return False
        i += 1
    return True


#Fonction de la question 7
def comu(reseau, groupe):
    """
    Trouve une communauté maximale dans un groupe donné.
```

```
    Une communauté maximale est définie comme un sous-ensemble du
groupe tel que :
        - Tous les membres de la communauté sont amis entre eux.
        - Aucun autre membre du groupe ne peut être ajouté à cette
communauté.

    Paramètre :
        - reseau (type: dict): Un dictionnaire représentant le réseau
d'amis. Chaque clé est une personne,
                               et la valeur correspond à une liste de
ses amis.
        - groupe (type: list): Une liste contenant les noms des
personnes parmi lesquelles chercher une communauté.

    Return :
        (type: list): Une liste contenant les membres de la communauté
maximale.
    """
    communaute=[]  # La communauté maximale, initialement vide.
    i=0
    # Parcourt chaque personne dans le groupe
    while i<len(groupe):
        personne = groupe[i]
        # Ajoute une personne à la communauté si elle est amie avec
tous les membres déjà présents
        if sont_amis_de(reseau, personne, communaute):
            communaute.append(personne)
        i += 1
    return communaute


#Fonction de la question 8
def tri_popu(reseau, groupe):
    """
    Trie un groupe par popularité (nombre d'amis) décroissante.

    Paramètre :
        reseau (type: dict) : Le réseau des amis.
        groupe (type: list) : Le groupe à trier.

    Return :
        (type: list) : Le groupe trié par popularité décroissante.
    """
```

```
    i = 0
    while i < len(groupe) - 1:
        # Comparer l'élément courant avec le suivant
        if len(reseau[groupe[i]]) < len(reseau[groupe[i + 1]]):
            # Échanger si l'élément suivant est plus populaire
            groupe[i],groupe[i+1]=groupe[i+1],groupe[i]
            # Revenir au début pour vérifier les échanges précédents
            i = 0
        else:
            # Passer au prochain élément
            i += 1
    return groupe


#Fonction de la question 9
def comu_dans_reseau(reseau):
    """
    Trie les membres d'un réseau par popularité décroissante et
applique
    l'algorithme de la fonction comu pour construire une communauté
maximale.

    Paramètre :
        reseau (type: dict): Un dictionnaire représentant le réseau
d'amis. Chaque clé est une personne,
                             et la valeur correspond à une liste de ses
amis.

    Return :
        (type: list): Une liste contenant les membres de la communauté
maximale obtenue.
    """
    # Appliquer la fonction de tri pour obtenir les membres triés par
popularité décroissante
    membres_triees = tri_popu(reseau, list(reseau))
    # Appliquer  la fonction comu pour obtenir la communauté maximale
    return comu(reseau, membres_triees)


#Fonction de la question 10
def comu_dans_amis(reseau, personne):
    """
```

```
    Crée une communauté maximale à partir d'une personne, en explorant
ses amis
    par popularité décroissante (nombre d'amis). Chaque ami est ajouté
à la communauté
    s'il est ami avec tous les membres de la communauté déjà présente.

    Paramètre :
        - reseau (type: dict): Le réseau des amis.
        - personne_initiale (type: str): La personne à partir de
laquelle commencer la communauté.

    Return :
        (type: list): La communauté maximale formée.
    """
    # Initialiser la communauté avec la personne de départ
    communaute = [personne]
    # Récupérer les amis de la personne de départ, triés par popularité
(nombre d'amis)
    amis = tri_popu(reseau, reseau[personne])
    # Initialiser l'index pour parcourir les amis avec while
    i = 0
    # Utilisation de while pour explorer les amis
    while i < len(amis):
        ami = amis[i]
        # Vérifier si l'ami est ami avec tous les membres de la
communauté
        if sont_amis_de(reseau, ami, communaute):
            communaute.append(ami)  # Ajouter l'ami à la communauté
        i += 1  # Passer à l'ami suivant
    return communaute


#Fonction de la question 12
def comu_max(reseau):
    """
    Trouve la plus grande communauté maximale dans un réseau en
appliquant comu_dans_amis à tous les membres.

    Pour chaque personne dans le réseau, la fonction construit une
communauté maximale à partir d'elle
    en utilisant la fonction comu_dans_amis. Elle compare ensuite
toutes les communautés trouvées
    et retourne la plus grande.
```

```
    Paramètre :
        reseau (type: dict): Un dictionnaire représentant le réseau
d'amis. Chaque clé est une personne,
                             et la valeur correspond à une liste de
ses amis.

    Return :
        (type: list): La plus grande communauté maximale trouvée dans
le réseau.
    """
    max_communaute = []  # Initialisation de la plus grande communauté
    # Appliquer comu_dans_amis à chaque membre du réseau
    membres = list(reseau)
    i = 0
    while i < len(membres):
        personne = membres[i]
        communaute = comu_dans_amis(reseau, personne)
        # Mettre à jour la plus grande communauté si celle-ci est plus
grande
        if len(communaute) > len(max_communaute):
            max_communaute = communaute
        i += 1
    return max_communaute
```

LES TESTS UNITAIRES SOUS VISUAL STUDIO CODE:

```
from math import *
from comu import *

#test unitaire de la question 1
def test_cree_reseau():
    """
    Cette fonction est un test unitaire de la fonction cree_reseau qui
vérifie si la fonction marche correctement.
    """
    amis1 = ["Alice", "Bob", "Alice", "Dan", "Bob", "Carl"]
    reponse1 = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl"],"Dan":
["Alice"],"Carl": ["Bob"]}
```

```
    amis2 = []
    reponse2 = {}
    amis3 = ["Alice", "Bob"]
    reponse3 = {"Alice": ["Bob"],"Bob": ["Alice"]}
    amis4 = ["Alice", "Bob", "Alice", "Bob"]


    assert cree_reseau(amis1) == reponse1
    assert cree_reseau(amis2) == reponse2
    assert cree_reseau(amis3) == reponse3


    print("La fonction cree_reseau est correct.")


#test unitaire de la question 3
def test_liste_personnes():
    """
    Test unitaire pour la fonction liste_personnes.
    Vérifie que la fonction retourne correctement la liste des
personnes dans différents cas.
    """
    reseau1 = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl"],"Dan":
["Alice"],"Carl": ["Bob"]}
    reponse1=["Alice", "Bob", "Dan", "Carl"]
    reseau2 = {}
    reponse2 = []
    reseau3 = {"Alice": []}
    reponse3 = ["Alice"]

    assert liste_personnes(reseau1) == reponse1
    assert liste_personnes(reseau2) == reponse2
    assert liste_personnes(reseau3) == reponse3

    print("La fonction liste_personnes est correct.")


#test unitaire de la question 4
def test_sont_amis():
    """
    Test unitaire pour la fonction sont_amis.
    Vérifie si la fonction identifie correctement les relations
d'amitié.
```

```
    """
    reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl"],"Dan":
["Alice"],"Carl": ["Bob"]}

    assert sont_amis(reseau, "Alice", "Bob") == True
    assert sont_amis(reseau, "Alice", "Carl") == False
    assert sont_amis(reseau, "Alice", "Gopi") == False
    assert sont_amis(reseau, "Ezzat", "Théo") == False
    assert sont_amis(reseau, "Bob", "Carl") == True

    print("La fonction sont_amis est correct.")


#test unitaire de la question 5
def test_sont_amis_de():
    """
    Test unitaire pour la fonction sont_amis_de.
    Vérifie si la fonction identifie correctement les relations
d'amitié dans un groupe.
    """
    reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl",
"Dan"],"Dan": ["Alice", "Bob"],"Carl": ["Bob"]}

    assert sont_amis_de(reseau, "Alice", ["Bob", "Dan"]) == True
    assert sont_amis_de(reseau, "Alice", ["Bob", "Carl"]) == False
    assert sont_amis_de(reseau, "Eve", ["Bob", "Dan"]) == False
    assert sont_amis_de(reseau, "Alice", []) == True
    assert sont_amis_de(reseau, "Alice", ["Alice", "Bob"]) == True

    print("La fonction sont_amis_de est correct.")


#test unitaire de la question 6
def test_est_comu():
    """
    Test unitaire pour la fonction est_comu.
    Vérifie si la fonction identifie correctement les communautés dans
un réseau.
    """
    reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Dan"],"Dan":
["Alice", "Bob"],"Carl": ["Eve"],"Eve": ["Carl"]}

    assert est_comu(reseau, ["Alice", "Bob", "Dan"]) == True
```

```
    assert est_comu(reseau, ["Alice", "Bob", "Carl"]) == False
    assert est_comu(reseau, ["Alice"]) == True
    assert est_comu(reseau, []) == True
    assert est_comu(reseau, ["Carl", "Eve", "Alice"]) == False

    print("La fonction est_comu est correct.")


#test unitaire de la question 7
def test_comu():
    """
    Test unitaire pour la fonction comu.
    Vérifie si la fonction identifie correctement les communautés
maximales dans différents groupes.
    """
    reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl",
"Dan"],"Dan": ["Alice", "Bob"],"Carl": ["Bob"]}
    groupe1 = ["Alice", "Bob", "Carl", "Dan"]
    reponse1 = ["Alice", "Bob", "Dan"]
    groupe2 = ["Carl", "Alice", "Bob", "Dan"]
    reponse2 = ["Carl", "Bob"]
    groupe3 = ["Carl", "Alice", "Dan"]
    reponse3 = ["Carl"]
    groupe4 = []
    reponse4 = []
    groupe5 = ["Alice"]
    reponse5 = ["Alice"]

    assert comu(reseau, groupe1) == reponse1
    assert comu(reseau, groupe2) == reponse2
    assert comu(reseau, groupe3) == reponse3
    assert comu(reseau, groupe4) == reponse4
    assert comu(reseau, groupe5) == reponse5

    print("La fonction comu est correct.")


#test unitaire de la question 8
def test_tri_popu():
    """
    Test unitaire pour la fonction tri_popu.
    Vérifie si la fonction trie correctement un groupe par popularité
décroissante.
```

```
    """
    reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl",
"Dan"],"Dan": ["Alice", "Bob"],"Carl": ["Bob"]}
    groupe1 = ["Alice", "Bob", "Carl", "Dan"]
    reponse1 = ["Bob", "Alice", "Dan", "Carl"]
    groupe2 = ["Carl", "Dan"]
    reponse2 = ["Dan", "Carl"]
    groupe3 = []
    reponse3 = []
    groupe4 = ["Alice"]
    reponse4 = ["Alice"]

    assert tri_popu(reseau, groupe1) == reponse1
    assert tri_popu(reseau, groupe2) == reponse2
    assert tri_popu(reseau, groupe3) == reponse3
    assert tri_popu(reseau, groupe4) == reponse4

    print("La fonction tri_popu est correct.")


#test unitaire de la question 9
def test_comu_dans_reseau():
    """
    Test unitaire pour la fonction comu_dans_reseau.
    Vérifie si la fonction trouve correctement la communauté maximale
dans un réseau donné.
    """
    reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl",
"Dan"],"Dan": ["Alice", "Bob"],"Carl": ["Bob"]}
    reponse1 = ["Bob", "Alice", "Dan"]
    resultat1 = comu_dans_reseau(reseau)
    reseau_vide = {}
    reponse2 = []
    resultat2 = comu_dans_reseau(reseau_vide)
    reseau_simple = {"Alice": []}
    reponse3 = ["Alice"]
    resultat3 = comu_dans_reseau(reseau_simple)

    assert resultat1 == reponse1
    assert resultat2 == reponse2
    assert resultat3 == reponse3

    print("La fonction comu_dans_reseau est correct.")
```

```python
#test unitaire de la question 10
def test_comu_dans_amis():
    """
    Test unitaire pour la fonction comu_dans_amis.
    Vérifie si la fonction trouve correctement la communauté maximale à
partir d'une personne donnée.
    """
    reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl",
"Dan"],"Dan": ["Alice", "Bob"],"Carl": ["Bob"]}
    prenom1 = "Alice"
    reponse1 = ["Alice", "Bob", "Dan"]
    resultat1 = comu_dans_amis(reseau, prenom1)
    prenom2 = "Carl"
    reponse2 = ["Carl", "Bob"]
    resultat2 = comu_dans_amis(reseau, prenom2)
    prenom3 = "Hachim"
    reseau["Hachim"] = []
    reponse3 = ["Hachim"]
    resultat3 = comu_dans_amis(reseau, prenom3)

    assert resultat1 == reponse1
    assert resultat2 == reponse2
    assert resultat3 == reponse3

    print("La fonction comu_dans_amis est correct.")


#test unitaire de la question 12
def test_comu_max():
    """
    Test unitaire pour la fonction comu_max.
    Vérifie si la fonction trouve correctement la plus grande
communauté maximale dans un réseau.
    """
    reseau = {"Alice": ["Bob", "Dan"],"Bob": ["Alice", "Carl",
"Dan"],"Dan": ["Alice", "Bob"],"Carl": ["Bob"]}
    reponse1 = ["Alice", "Bob", "Dan"]
    resultat1 = comu_max(reseau)
    reseau_vide = {}
    reponse2 = []
    resultat2 = comu_max(reseau_vide)
    reseau_simple = {"Alice": []}
```

```
    reponse3 = ["Alice"]
    resultat3 = comu_max(reseau_simple)

    assert resultat1 == reponse1
    assert resultat2 == reponse2
    assert resultat3 == reponse3
    print("La fonction comu_max est correct.")
```