Les graphes

Représenter une situation réelle par un graphe

Exercice 1

Les situations suivantes peuvent être décrits par des graphes :

- ✗ un réseau d'ordinateurs,
- ✗ des maisons avec des boîtes aux lettres,
- ✗ le métro parisien,
- ✗ une ville, dans laquelle il n'y a plus ni radio ni télé, et dans laquelle se propage par SMS l'information de l'arrivée d'un tsunami,
- ✗ une population au sein de laquelle se propage une épidémie.

1. Pour chaque situation, **indiquer** à quoi correspondent les sommets et les arrêtes.

Situation	Sommets	Arrêtes
réseau d'ordinateurs	Ordinateurs	Câbles réseaux
maisons avec des boîtes aux lettres	Maisons	Routes reliant chaque maison
métro parisien	Stations	Réseau ferré du métro
SMS d'information	Population	Envoi de SMS
une population	Personnes	Contamination entre personnes

2. **Décrire** à quoi correspond le plus court chemin entre deux sommets.

Situation	Plus court chemin
réseau d'ordinateurs	Route avec le minimum de lien entre deux ordinateurs
maisons avec des boîtes aux lettres	Chemin le plus court entre deux maisons
métro parisien	Chemin avec le moins de changement entre deux stations
SMS d'information	Le nombre d'envoi intermédiaires pour être averti
une population	Le nombre contaminations intermédiaires pour être infecté

Exercice 2

1. **Décrire** un réseau social comme un graphe en indiquant ce que représentent les sommets et les arêtes. **Justifier** que ce graphe n'est pas forcément orienté.

Dans un réseau social les sommets sont les personnes et les arrêtes sont les liens d'amitié. Dans un réseau social A peut connaître B sans que B connaisse A (cas des célébrités)

2. **Indiquer** les liens sociaux correspondant à deux sommets adjacents

Deux sommets adjacents se connaissent directement

Analyser un graphe

Exercice 3

Imaginer un réseau social où chaque personne a dix amis,

1. **Déterminer** combien de personnes au maximum un message diffusé aux amis des amis des amis, etc. peut-il atteindre en une heure, si le délai entre la réception et le rediffusion d'un message est de cinq minutes.

Le message peut être rediffusé 12 fois (toutes les 5 minutes pendant 1 heure).
Puisque chaque personne contacte 10 amis, après 2 envois il y aura 10 x 10 personnes averties donc après 12 envois il y aura 10^{12} personnes averties si il n'y a pas d'amis communs entre personnes

2. **Interpréter** ce résultat par rapport au risque de propagation d'une rumeur sur un réseau social.

On voit par ce chiffre la vitesse de propagation des fake News si celles ci sont renvoyées sans discernement entre chaque personnes

Exercice 4

Sur le plus grand réseau social du monde, il y a environ un milliard de personnes connectées chacune à cent autres personnes environ. Un chemin de deux arêtes – ami d'ami – connecte donc une personne à environ 100×100 = 10 000 autres personnes si on néglige les amis communs et, disons, à environ 100×50 = 5 000 si l'on tient compte des amis communs.
Un chemin de trois arêtes – ami d'ami d'ami – connecte une personne à environ 100 × 50×50 = 250 000 autres personnes.

1. En supposant grossièrement qu'on ne gagne ainsi que la moitié de nouveaux contacts à chaque arête, **déterminer** à combien de personnes environ est-on connecté avec un chemin de six arêtes.
   N = 100x50x50x50x50x50 = 3,1 x10^{10} personnes
2. Dans un tel réseau social, **déterminer** la distance maximale entre deux personnes.

Dans un réseau social de 1 milliard d'utilisateur seulement 5 personnes maxi séparent 2 utilisateurs. Entre moi et Barack Obama, il n'y a que 5 personnes

Il se trouve que ce calcul approximatif donne un résultat très proche de la réalité.

Représenter un graphe par une matrice d'adjacence

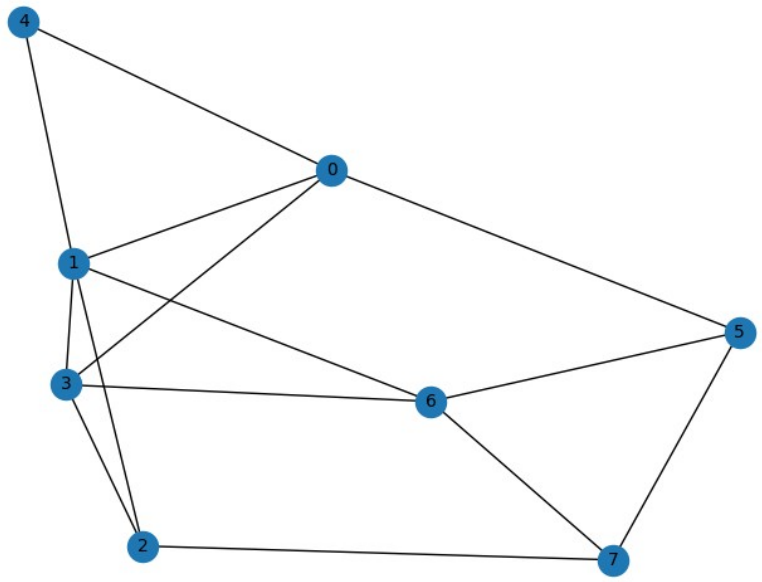
Considérons un réseau social dans lequel les relations entre individus sont des relations symétriques : si *A* est ami avec *B*, alors *B* est aussi ami avec *A*.
On peut alors construire un graphe dans lequel les nœuds sont les individus et les arêtes représentent les relations d'amitié.

On considère aussi que ce réseau est un mini réseau qui contient 8 personnes : Alban, Béatrice, Charles, Déborah, Éric, Fatima, Gérald et Hélène dont les relations qui lient d'amitié ces membre sont :

- Alban est ami avec Béatrice, Déborah, Éric et Fatima.

- Béatrice est amie avec ~~Alban~~, Charles, Déborah, Éric et Gérald.
- Charles est ami avec ~~Béatrice~~, Déborah et Hélène.
- Déborah est amie avec ~~Alban~~, ~~Béatrice~~, ~~Charles~~ et Gérald.
- Éric est ami avec ~~Alban~~ et ~~Béatrice~~.
- Fatima est amie avec ~~Alban~~, Gérald et Hélène.
- Gérald est ami avec ~~Béatrice~~, ~~Déborah~~, ~~Fatima~~ et Hélène.
- Hélène est amie avec ~~Charles~~, ~~Gérald~~ et ~~Fatima~~.

1. **Illustrer** par un graphe ce réseau social.



2. **Déterminer** la matrice d'adjacence de ce réseau

,	,	,	,	,	,	,	
,	,	,	,	,	,	,	
,	,	,	,	,	,	,	
,	,	,	,	,	,	,	
,	,	,	,	,	,	,	
,	,	,	,	,	,	,	
,	,	,	,	,	,	,	
,	,	,	,	,	,	,	

La classe Graphe suivante permet de définir un graphe grâce à sa table d'adjacence.

```
class Graphe :
    '''Graphe represente par sa matrice d'adjacence,
    où les sommets sont les entiers o, 1, ..., n-1'''

    def __init__(self, n) :
        self.__n = n
        self.__adj = [[False] * n for _ in range(n)]

    def ajouter_arete(self, s1, s2) :
        '''Ajoute une arete entre les sommets s1 et S2'''
        self.__adj[s1][s2] = True
        self.__adj[s1][s2] = True
```

```
    def arete(self, s1, s2) :
    '''Indique la presence d'un arc ou non
    entre s1 et s2'''
          return self.__adj[s1][s2]

    def voisins(self, s) :
          '''Renvoie la liste des voisins de s'''
          v =[]
          for i in range(self.__n) :
                if self.__adj[s][i] :
                      v.append(i)
          return v
```

3. **Implémenter** ce graphe avec la classe Graphe.

```
p={'Alban':0,'Béatrice':1, 'Charles':2, 'Déborah':3, 'Eric':4, 'Fatima':5,
'Gérald':6, 'Hélène':7}
g = Graphe(8)
g.ajouter_arete(p['Alban'],p['Béatrice'])
g.ajouter_arete(p['Alban'],p['Déborah'])
g.ajouter_arete(p['Alban'],p['Eric'])
g.ajouter_arete(p['Alban'],p['Fatima'])
g.ajouter_arete(p['Béatrice'],p['Charles'])
g.ajouter_arete(p['Béatrice'],p['Déborah'])
g.ajouter_arete(p['Béatrice'],p['Eric'])
g.ajouter_arete(p['Béatrice'],p['Gérald'])
g.ajouter_arete(p['Charles'],p['Déborah'])
g.ajouter_arete(p['Charles'],p['Hélène'])
g.ajouter_arete(p['Déborah'],p['Gérald'])
g.ajouter_arete(p['Fatima'],p['Gérald'])
g.ajouter_arete(p['Fatima'],p['Hélène'])
g.ajouter_arete(p['Gérald'],p['Hélène'])
```

4. **Modifier** la classe Graphe afin d'ajouter les méthode suivantes :
   - ➔ Méthode **`Afficher()`** pour afficher le graphe sous la forme suivante :
     0 → 1 3
     1 → 0 3 4
   - ➔ Méthode **`supprimer_arc(s1,s2)`** pour supprimer l'arc entre les sommets s1 et s2
   - ➔ Méthode **`degre(s)`** qui donne le nombre d'arêtes issus du sommet s
   - ➔ Méthode **`nb_aretes()`** qui donne le nombre d'arêtes du graphe

```
class Graphe :
'''Graphe represente par sa matrice d'adjacence,
où les sommets sont les entiers o, 1, ..., n-1'''


    def __init__(self, n) :
          self.__n = n
          self.__adj = [[False] * n for _ in range(n)]

    def ajouter_arete(self, s1, s2) :
          '''Ajoute une arete entre les sommets s1 et S2'''
          self.__adj[s1][s2] = True
          self.__adj[s2][s1] = True

    def supprimer_arete(self, s1, s2) :
          '''Ajoute une arete entre les sommets s1 et S2'''
```

```
            self.__adj[s1][s2] = False
            self.__adj[s2][s1] = False

      def arete(self, s1, s2) :
            '''Indique la presence d'une arete ou non entre s1 et s2'''
            return self.__adj[s1][s2]

      def voisins(self, s) :
            '''Renvoie la liste des voisins de s'''
            v =[]
            for i in range(self.__n) :
                  if self.__adj[s][i] :
                        v.append(i)
            return v

      def degre(self,s) :
            '''Nombre d'aretes sur le sommet s'''
            return self.__adj[s].count(True)

      def nb_aretes(self,) :
            ''' Nombre d'aretes dans le graphe'''
            n = 0
            for i in range(self.__n) :
                  n+=self.degre(i)
            return n/2

      def afficher(self) :
            for i in range(self.__n):
                  proches = self.voisins(i)
                  print (i, ' -> ', end='')
                  for p in proches :
                        print (p, end =' ')
                  print()

p={'Alban':0,'Béatrice':1, 'Charles':2, 'Déborah':3, 'Eric':4, 'Fatima':5,
'Gérald':6, 'Hélène':7}
g = Graphe(8)
g.ajouter_arete(p['Alban'],p['Béatrice'])
g.ajouter_arete(p['Alban'],p['Déborah'])
g.ajouter_arete(p['Alban'],p['Eric'])
g.ajouter_arete(p['Alban'],p['Fatima'])
g.ajouter_arete(p['Béatrice'],p['Charles'])
g.ajouter_arete(p['Béatrice'],p['Déborah'])
g.ajouter_arete(p['Béatrice'],p['Eric'])
g.ajouter_arete(p['Béatrice'],p['Gérald'])
g.ajouter_arete(p['Charles'],p['Déborah'])
g.ajouter_arete(p['Charles'],p['Hélène'])
g.ajouter_arete(p['Déborah'],p['Gérald'])
g.ajouter_arete(p['Fatima'],p['Gérald'])
g.ajouter_arete(p['Fatima'],p['Hélène'])
g.ajouter_arete(p['Gérald'],p['Hélène'])
```

➔ Méthode **`matrix_to_list()`** qui renvoie la liste des voisins à partir de la matrice d'adjacence définie en attribut.

Le graphe renvoyé sera codé sous la forme d'un dictionnaire, les clés étant les sommets, numérotés de 0 à n-1 et les valeurs associées sont les listes des successeurs du sommet.

```python
def matrix_to_list(self) :
    matrix = {}
    for i in range(len(self.__adj)) :
        matrix[i] = []
        for j in range(len(self.__adj[i])) :
            if self.__adj[i][j] : matrix[i].append(j)
    return matrix
```

Représentation d'un graphe par une liste des successeurs

La classe Graphe_oriente_ls suivante permet de définir un graphe grâce à une liste des successeurs.

```python
class Graphe_oriente_la :
'''Graphe decrit par un dictionnaire d'adjacence'''
    def __init__(self) :
        self.__adj = {}

    def ajouter_sommet(self,s) :
        if s not in self.__adj : self.__adj[s]=[]

    def ajouter_arc(self, s1, s2) :
        self.ajouter_sommet(s1)
        self.ajouter_sommet(s2)
        self.__adj[s1].append(s2)

    def arc_existe(self, s1, s2) :
        return s2 in self.__adj[s1]

    def sommets(self) :
        return list(self.__adj.keys())

    def voisins(self, s) :
        return self.__adj[s]
```

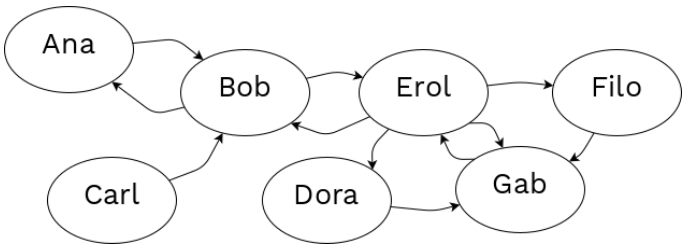
Programme disponible sur .\donnee\NSI\graphes\graphe_oriente_ls.py

1. **Ajouter** à la classe, une méthode **predecesseurs()** qui renvoie un dictionnaire dont les clés sont les sommets et les valeurs associées sont les listes des prédécesseurs du sommet.
2. **Ajouter** à la classe, une méthode **arc_existe(s1, s2)** qui renvoie vrai si un arc est défini entre s1 et s2

```python
class Graphe_oriente_la :
'''Graphe decrit par un dictionnaire d'adjacence'''
    def __init__(self) :
        self.__adj = {}

    def __repr__(self) :
        out = ''
        for cle in self.__adj.keys() :
            out += cle + ' -> ' + str(self.__adj[cle])[1:-1] + '\n'
        return out

    def ajouter_sommet(self,s) :
```

```
        if s not in self.__adj : self.__adj[s]=[]

    def ajouter_arc(self, s1, s2) :
        self.ajouter_sommet(s1)
        self.ajouter_sommet(s2)
        self.__adj[s1].append(s2)

    def arc_existe(self, s1, s2) :
        return s2 in self.__adj[s1]


    def sommets(self) :
        return list(self.__adj.keys())

    def voisins(self, s) :
        return self.__adj[s]

    def predecesseurs(self) :
        previous ={s :[] for s in self.sommets()}
            for s1 in self.__adj.keys() :
                for s2 in self.__adj[s1] :
                    previous[s2].append(s1)
        return previous
```

On considère, dans cet exercice, le graphe orienté figure 1, représentant les personnes suivies par d'autres personnes sur un réseau social.



3. **Définir** le dictionnaire correspondant au graphe proposé ci-dessus.

```
{'Ana': ['Bob'], 'Bob': ['Ana', 'Erol'], 'Carl': ['Bob'], 'Erol': ['Bob', 'Filo', 'Gab', 'Dora'], 'Filo': ['Gab'], 'Gab': ['Erol'], 'Dora': ['Gab']}
```

Les arêtes de ce graphe peuvent être représentées par un tuple de deux chaînes de caractères. La première chaîne de caractère représentant le sommet de départ et la deuxième le sommet d'arrivée de l'arc. Pour exemple le tuple suivant représente l'arc repéré sur la figure 1 liant Ana à Bob :

('Ana', 'Bob')

La liste de tous les arcs de ce graphe est la suivante :

```
arcs = [('Ana', 'Bob'), ('Bob', 'Ana'), ('Carl', 'Bob'), ('Erol', 'Bob'), ('Bob', 'Erol'), ('Erol', 'Filo'), ('Erol', 'Gab'), ('Filo', 'Gab'), ('Gab', 'Erol'), ('Erol', 'Dora'), ('Dora', 'Gab')]
```

4. **Créer** une fonction **charge_arcs(graphe, arcs)** qui prend pour argument une instance de la classe graphe_oriente_ls et la liste des arcs. Cette fonction doit insérer dans l'objet de la classe graphe_oriente_ls, les sommets et les arcs définis dans la liste arcs. **Tester** et **valider** cette fonction

```
def charger_arcs(g,arcs) :
    for l in arcs :
        g.ajouter_arc(l[0], l[1])
```

5. **Ecrire** une fonction **amis_d_amis(graphe, pers)** qui prend en argument une instance de la classe Graphe_oriente_la et une chaîne de caractère au nom de la personne et qui renvoie la liste des amis des amis de pers, à l'exclusion d'elle même et sans doublon.

```
def amis_d_amis(g, pers) :
    ami_de_ami = []
    for ami in g.voisins(pers) :
        for ami_de in g.voisins(ami) :
            if ami_de is not pers and ami_de not in ami_de_ami :
                ami_de_ami.append(ami_de)
    return ami_de_ami
```