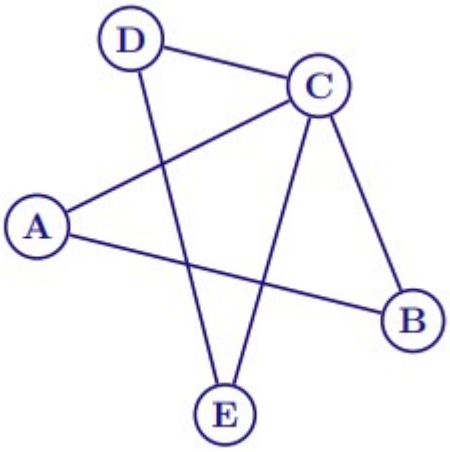
graphe

April 15, 2021

1 Graphes

1.1 Introduction

Un graphe est un ensemble de sommets reliés par des arcs (arêtes).



On donne ci-dessous un graphe G:

Le graphe G possède 5 **sommets** A, B, C, D, E et 6 **arcs** ou arêtes.

On dit que 2 sommets sont adjacents s'ils sont reliés par un arc. On peut dresser la liste des sommets adjacents:

- Les sommets adjacents à A sont : B et C;
- Les sommets adjacents à B sont : A et C;
- Les sommets adjacents à C sont : A, B, C et D;
- Les sommets adjacents à D sont : C et E;
- Les sommets adjacents à E sont : C et D;

Maintenant, on dresse un tableau qui contient autant de lignes et colonnes que le nombre de sommets du graphe. Chaque colonne et chaque ligne représente un sommet du graphe (dans l'ordre alphabétique).
Lorsque les sommets sont adjacents on inscrit le nombre 1.
S'ils ne sont pas adjacents, on inscrit 0.

	A	B	C	D	E
A	0	1	1	0	0
B	1	0	1	0	0
C	1	1	0	1	1
D	0	0	1	0	1
E	0	0	1	1	0

On note ce tableau sous forme de matrice écrite uniquement avec les nombres 0 et 1 entre deux parenthèses englobantes. Soit M cette **matrice d'adjacence**.

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \tag{1}$$

- Le nombre de ligne (et colonnes) donne la dimension de la matrice: la dimension de la matrice M est $n = 5$
- On repère chaque valeur de la matrice M par $M[i, j]$ où i est un indice de ligne et j un indice de colonne. Les indices i et j sont compris entre 0 et $n - 1$.
  On $M[0, 1] = 1$ et $M[4, 2] = 1$.

1.1.1 Conclusion

Nous avons une approche mathématique du graphe. En informatique, la liste des sommets adjacents et la matrice d'adjacence sont deux approches pour définir un même graphe.
Nous définirons la liste des sommets adjacents aves un **dictionnaire** et la matrice d'adjacence sera construite avec une **liste** de valeurs booléennes.

1.2 Matrice d'adjacence d'un graphe

Dans cette partie, on représente en Python la **matrice d'adjacence** par une liste. L'indice de la liste représente un sommet du graphe : le sommet A sera l'indice 0, le sommet B d'indice 1, etc.
On rappelle que le graphe G n'est pas **orienté**.

La matrice d'adjacence M de notre graphe G peut se définir en Python par la liste:

```
M=[[0,1,1,0,0],[1,0,1,0,0],[1,1,0,1,1],[0,0,1,0,1],[0,0,1,1,0]]
```

Pour savoir si deux sommets i et j du graphe sont adjacents, un test sur la valeur du coefficient $M[i][j]$ suffit.
La fonction **est_adjacent** prend en paramètre la matrice M et les indices i et j de deux sommets. La fonction renvoie un booléen de valeur **True** si les sommets sont adjacents et **False** dans le cas contraire.

```
def est_adjacent(M,i,j):
    assert (0<=i<len(M) and 0<=j<len(M)),"erreur indices sommets"
    return M[i][j]==1
```

La création d'une matrice d'adjacence d'un graphe donné se fait en deux temps. D'abord on crée une matrice initialisée avec des zéros, ensuite on ajoute les arcs en modifiant les valeurs de la matrice à 1.

La fonction **init** prend en paramètre la matrice la dimension n de la matrice et renvoie une matrice d'adjacence où toutes les valeurs sont nulles. Cette matrice est associée à un graphe où les sommets sont tous **isolés**.

```
def init(n):
    return [[0]*n for _ in range(n)]
```

Remarque:

> La construction par **compréhension** crée une liste remplie de n listes toutes initialisées avec n zéros ($[0] * n$).

L'ajout d'un arc entre deux sommets dépend de la nature du graphe. Le graphe n'étant pas orienté, les sommets sont mutuellement adjacents.

La fonction **ajouter_arc** prend en paramètre la matrice M et les sommets d'indices i et j. Elle modifie la valeur $M[i][j]$ et $M[j][i]$ à 1.

```
def ajouter_arc(M,i,j):
    assert (0<=i<len(M) and 0<=j<len(M)),"erreur indices sommets"
    M[i][j]=1
    M[j][i]=1
```

Remarque:

> Pour un graphe orienté, un arc du sommet i vers le sommet j modifie uniquement la valeur $M[i][j]$; on a $M[i][j] = 1$ alors que $M[j][i] = 0$.

La suppression d'un arc entre deux sommets est semblable à la fonction précédente, la valeur du coefficient de la matrice valant 0.

```
def supprimer_arc(M,i,j):
    assert (0<=i<len(M) and 0<=j<len(M)),"erreur indices sommets"
    M[i][j]=0
    M[j][i]=0
```

1.2.1 Classe d'objet : Graphe_mat

Finalement, on peut créer une classe avec pour attribut la matrice d'adjacence et les méthodes d'ajout, suppression et test d'adjacence. On obtient le code:

```
class Graphe_mat:
    def __init__(self,n):
        self.mat=[[0]*n for _ in range(n)]

    def est_adjacent(self,i,j):
        assert (0<=i<len(self.mat) and 0<=j<len(self.mat)),"erreur indices sommets"
        return self.mat[i][j]==1
```

```
    def ajouter_arc(self,i,j):
        assert (0<=i<len(self.mat) and 0<=j<len(self.mat)),"erreur indices sommets"
        self.mat[i][j]=1
        self.mat[j][i]=1

    def supprimer_arc(self,i,j):
        assert (0<=i<len(self.mat) and 0<=j<len(self.mat)),"erreur indices sommets"
        self.mat[i][j]=0
        self.mat[j][i]=0
```

[42]:
```
M1=[[0,1,1,0,0],[1,0,1,0,0],[1,1,0,1,1],[0,0,1,0,1],[0,0,1,1,0]]
```

[30]:
```
def est_adjacent(M,i,j):
    assert (0<=i<len(M) and 0<=j<len(M)),"erreur indices sommets"
    return M[i][j]==1

def init(n):
    return [[0]*n for _ in range(n)]

def ajouter_arc(M,i,j):
    assert (0<=i<len(M) and 0<=j<len(M)),"erreur indices sommets"
    M[i][j]=1
    M[j][i]=1
    return M

def supprimer_arc(M,i,j):
    assert (0<=i<len(M) and 0<=j<len(M)),"erreur indices sommets"
    M[i][j]=0
    M[j][i]=0
    return M
```

[37]:
```
class Graphe_mat:
    def __init__(self,n):
        self.mat=[[0]*n for _ in range(n)]

    def est_adjacent(self,i,j):
        assert (0<=i<len(self.mat) and 0<=j<len(self.mat)),"erreur indices␣
 ↪sommets"
        return self.mat[i][j]==1

    def ajouter_arc(self,i,j):
        assert (0<=i<len(self.mat) and 0<=j<len(self.mat)),"erreur indices␣
 ↪sommets"
        self.mat[i][j]=1
        self.mat[j][i]=1

    def supprimer_arc(self,i,j):
```

```
        assert (0<=i<len(self.mat) and 0<=j<len(self.mat)),"erreur indices␣
↪sommets"
        self.mat[i][j]=0
        self.mat[j][i]=0
```

[40]:
```
M=Graphe_mat(5)
M.ajouter_arc(0,1)
M.ajouter_arc(0,2)
M.ajouter_arc(1,0)
M.ajouter_arc(1,2)
M.ajouter_arc(2,0)
M.ajouter_arc(2,1)
M.ajouter_arc(2,3)
M.ajouter_arc(2,4)
M.ajouter_arc(3,2)
M.ajouter_arc(3,4)
M.ajouter_arc(4,2)
M.ajouter_arc(4,3)
```

[43]:
```
M.mat==M1
```

[43]:
```
True
```

1.3 Dictionnaire d'adjacence

L'implémentation précédente ne permet pas de repérer facilement les sommets du graphe. On peut implémenter le graphe G avec un dictionnaire. Chaque sommet du graphe sera une **clef** du dictionnaire et les **valeurs** associées à chacune des clefs sera l'ensemble de ces sommets adjacents.

La structure de donnée **ensemble** de Python s'appuie sur la définition mathématique (les événements en probabilités, solutions d'équations) et permet de réunir différentes valeurs et aussi réaliser quelques opérations comme l'intersection ou la réunion.

Remarque:

> on peut utiliser des listes pour les valeurs de sommets, mais celles-ci ne gèrent pas la duplication ce qui signifie que l'on peut avoir plusieurs fois le même sommet dans la liste.

On peut déclarer le dictionnaire d'adjacence du graphe G directement:

```
G= {'A': {'B', 'C'}, 'B': {'A', 'C'}, 'C': {'A', 'B', 'D', 'E'}, 'D': {'C', 'E'}, 'E': {'C', '
```

Il est possible de créer (initialiser) un dictionnaire d'adjacence vide et le remplir ensuite avec les sommets du graphe et les sommets adjacents.

La fonction **init** renvoie un dictionnaire vide.

```
def init():
    return dict()
```

L'ajout de sommets dans le graphe nécessite un test de présence du sommet. Si celui-ci n'est pas présent, on crée une **clef** avec le nom du sommet et on associe un ensemble vide comme valeur. La création d'un ensemble vide se fait soit par deux accolades vides ou la fonction **set()**.

La fonction **ajouter_sommet** prend en paramètre le graphe (dictionnaire) et le sommet à ajouter et renvoie le dictionaire avec une nouvelle clef et une valeur (ensemble) vide.

```
def ajouter_sommet(g,s):
    if s not in g:
        g[s]=set()
```

Ensuite, la création des arcs du graphe se fait en ajoutant les sommets adjacents à chaque sommet sous forme d'ensembles. L'ajout d'une valeur dans un ensemble se fait avec la méthode **add** qui prend en argument la valeur à ajouter.

La fonction **ajouter_arc** prend en paramètre un graphe (dictionnaire) et deux sommets à relier par un arc et renvoie le graphe. On commence par ajouter les sommets en appelant la fonction ajouter_sommet. Ensuite on complète chaque **clef/sommet** du graphe en ajoutant les sommets commes valeurs.

```
def ajouter_arc(g,s1,s2):
    ajouter_sommet(g,s1)
    ajouter_sommet(g,s2)
    g[s1].add(s2)
    g[s2].add(s1)
```

Remarque:

> Pour un graphe orienté, il faut prendre en compte le sens de l'arc.
> Par exemple, si le sommet s1 est relié vers le sommet s2, alors seule la valeur de la clef s1 est modifiée $s1 : \{s2\}$.

Pour vérifier l'existence d'un arc entre deux sommets, ce qui revient à vérifier que 2 sommets sont adjacents, on crée la fonction **arc** qui prend en paramètre le graphe et les deux sommets et renvoie un booléen qui sera **True** si les sommets sont adjacents et **False** sinon.

```
def arc(g,s1,s2):
    if s1 in g[s2] or s2 in g[s1]:
        return True
    else:
        return False
```

La liste des sommets du graphe s'obtient en renvoyant les clefs du dictionnaire définissant le graphe et la **liste** de tous les sommets adjacents à un sommet donné s'obtient en donnant la valeur associée à la clef sommet sous forme de liste.

La fonction **sommets** prend en argument un graphe et renvoie les sommets voulus. La méthode **keys()** convient parfaitement mais le type **dict_keys** est peut maniable et doit être transformé en liste avec la fonction **list** de Python:

```
def sommets(g):
    return list(g.keys())
```

La fonction **voisins** prend en argument un graphe et un sommet et renvoie la liste des sommets adjacents au sommet passé en argument. La valeur renvoyée est un ensemble et peut nécessiter une conversion en liste avec la fonction **list** de Python:

```
def voisins(g,s):
    return g[s]
```

1.3.1 Classe d'objet : Graphe_dict

On peut transformer les fonctions et le graphe précédent en **classe** permettant d'utiliser la programmation objet. Un seul attribut suffit pour créer notre graphe constitué d'un dictionnaire.

On obtient le code suivant:

```
class Graphe_dict:
    """un graphe comme un dictionnaire d'adjacence"""

    def __init__(self,oriente=True):
        self.adj={}

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

    def ajouter_arc(self,s1,s2):
        self.ajouter_sommet(s1)
        self.ajouter_sommet(s2)
        self.adj[s1].add(s2)
        self.adj[s2].add(s1)

    def arc(self,s1,s2):
        return s2 in self.adj[s1] or s1 in self.adj[s2]

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

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

1.4 De l'un à l'autre

On va créer deux fonctions qui crée un objet à partir de l'autre objet.

Par exemple, si on a la matrice d'adjacence M, la fonction renvoie le dictionnaire d'adjacence associé au graphe G. Et inversement, si on a le dictionnaire d'adjacence, on doit créer la matrice d'adjacence M du graphe G.

Soit **graphe_dict_to_mat** la fonction qui crée la matrice d'adjacence à partie du dictionnaire. Elle prend en paramètre un objet de type **Graphe_dict** et renvoie un objet de type **Graphe_mat**.

L'algorithme est le suivant: - on liste les sommets du graphe; - chaque sommet (unique) est associé

à un indice de liste qui correspond à l'indice de ligne et de colonne de la matrice;
Pour chaque sommet de la liste d'indice i: - on liste ses sommets voisins et pour chaque indice j du voisin, on met la valeur $M[i][j]$ de la matrice à 1. - On renvoie l'objet et donc sa matrice.

```
def graphe_dict_to_mat(g):
    # On liste les sommets du graphe
    sommets=g.sommets()
    # Le nombre de sommets du graphe donne la dimension de la matrice d'adjacence
    n=len(sommets)
    # Création de l'abjet matrice d'adjacence
    M=Graphe_mat(n)
    # On parcourt les sommets du graphe
    for s in sommets:
        # indice de ligne du sommet S
        i=sommets.index(s)
        # On parcourt les voisins du sommet s
        for v in g.voisins(s):
            # indice de colonne du sommet adjacent
            j=sommets.index(v)
            # coefficient (i,j) ey (j,i) mis à 1
            M.ajouter_arc(i,j)
    return M
```

Soit **graphe_mat_to_dict** la fonction qui crée le dictionnaire d'adjacence à partir de la matrice d'adjacence du graphe. La fonction prend en argument un objet de type **Graphe_mat** et renvoie un objet de type **Graphe_dict** contenant les sommets du graphhe associé à la matrice d'adjacence.

L'algorithme est le suivant: - on crée un objet Graphe_dict qui est un dictionnaire vide; - On note les sommets dans l'ordre alphabétique en majuscule en partant de **A**; - on ajoute les sommets du graphe; - on parcourt chaque coefficient de la matrice et on crée pour chaque sommet sa liste de sommets adjacents - on renvoie l'abjet graphe_dict

Remarque:

> En python, le code ASCII d'un caratère est donné par la fonction **ord(caractère)** et le caractère est donné par la fonction **chr(code_ascii)**

"'python

def graphe_mat_to_dict(m): # On crée l'objet g de type Graphe_dict g=Graphe_dict() # On récupère le code ascii (en décimal) de la lettre A rg=ord('A') # On parcourt chaque ligne de la matrice for i in range(len(m.mat)): # On parcourt chaque colonne d'une ligne de matrice for j in range(len(m.mat)): # si le coeffient vaut 1, les sommets sont adjacents, if m.mat[i][j]==1: # on ajoute un arc entre les sommets i et j à l'objet Graphe_dict g.ajouter_arc(chr(rg+i),chr(rg+j)) return g"'

[26]:
```
G= {'A': {'B', 'C'}, \
    'B': {'A', 'C'}, \
    'C': {'A', 'B', 'D', 'E'}, \
    'D': {'C', 'E'}, \
```

```
    'E': {'C', 'D'}}
print(G)
```

```
{'A': {'C', 'B'}, 'B': {'C', 'A'}, 'C': {'B', 'E', 'D', 'A'}, 'D': {'C', 'E'},
'E': {'C', 'D'}}
```

[27]:
```
def init():
    return dict()

def ajouter_sommet(g,s):
    if s not in g:
        g[s]=set()

def ajouter_arc(g,s1,s2):
    ajouter_sommet(g,s1)
    ajouter_sommet(g,s2)
    g[s1].add(s2)
    g[s2].add(s1)

def arc(g,s1,s2):
    if s1 in g[s2] or s2 in g[s1]:
        return True
    else:
        return False

def sommets(g):
    return list(g.keys())

def voisins(g,s):
    return list(g[s])

H=init()
ajouter_sommet(H,'A')
ajouter_sommet(H,'B')
ajouter_sommet(H,'A')
ajouter_arc(H,'A','B')
ajouter_arc(H,'A','C')
if arc(H,'B','C'):
    print("les sommets sont adjacents")
else:
    print("les sommets ne sont pas adjacents")
print(H)
print(sommets(H))
print(voisins(H,'A'))
```

```
les sommets ne sont pas adjacents
{'A': {'C', 'B'}, 'B': {'A'}, 'C': {'A'}}
['A', 'B', 'C']
```

```
{'C', 'B'}
```

[2]:
```python
class Graphe_dict:
    """un graphe comme un dictionnaire d'adjacence"""

    def __init__(self):
        self.adj={}

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

    def ajouter_arc(self,s1,s2):
        self.ajouter_sommet(s1)
        self.ajouter_sommet(s2)
        self.adj[s1].add(s2)
        self.adj[s2].add(s1)

    def arc(self,s1,s2):
        return s2 in self.adj[s1] or s1 in self.adj[s2]

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

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

[3]:
```python
G=Graphe_dict()
G.ajouter_sommet('A')
G.ajouter_arc('A','B')
G.ajouter_arc('A','D')
G.ajouter_arc('B','D')
G.ajouter_arc('B','C')
G.ajouter_arc('D','B')
print(G.adj)
```

```
{'A': {'B', 'D'}, 'B': {'D', 'A', 'C'}, 'D': {'B', 'A'}, 'C': {'B'}}
```

[4]:
```python
G.ajouter_arc('E','A')
```

[5]:
```python
G.ajouter_arc('B','E')
```

[9]:
```python
print(G.sommets())
```

```
['A', 'B', 'D', 'C', 'E']
```

[58]:
```
def graphe_dict_to_mat(g):
    # On liste les sommets du graphe
    sommets=g.sommets()
    # Le nombre de sommets du graphe donne la dimension de la matrice␣
 ↪d'adjacence
    n=len(sommets)
    # Création de l'abjet matrice d'adjacence
    M=Graphe_mat(n)
    # On parcourt les sommets du graphe
    for s in sommets:
        # indice de ligne du sommet S
        i=sommets.index(s)
        # On parcourt les voisins du sommet s
        for v in g.voisins(s):
            # indice de colonne du sommet adjacent
            j=sommets.index(v)
            # coefficient (i,j) ey (j,i) mis à 1
            M.ajouter_arc(i,j)
    return M


# On crée le dictionnaire d'adjacence du graphe d'introduction
G=Graphe_dict()
G.ajouter_arc('A','B')
G.ajouter_arc('A','C')
G.ajouter_arc('B','A')
G.ajouter_arc('B','C')
G.ajouter_arc('C','A')
G.ajouter_arc('C','B')
G.ajouter_arc('C','D')
G.ajouter_arc('C','E')
G.ajouter_arc('D','C')
G.ajouter_arc('D','E')
G.ajouter_arc('E','C')
print(G.adj)
M=graphe_dict_to_mat(G)
print(M.mat)
```

```
{'A': {'C', 'B'}, 'B': {'C', 'A'}, 'C': {'B', 'E', 'D', 'A'}, 'D': {'C', 'E'},
'E': {'C', 'D'}}
[[0, 1, 1, 0, 0], [1, 0, 1, 0, 0], [1, 1, 0, 1, 1], [0, 0, 1, 0, 1], [0, 0, 1,
1, 0]]
```

[72]:
```
def graphe_mat_to_dict(m):
    # On crée l'objet g de type Graphe_dict
    g=Graphe_dict()
    # On récupère le code ascii (en décimal) de la lettre A
```

```
    rg=ord('A')
    # On parcourt chaque ligne de la matrice
    for i in range(len(m.mat)):
        # On parcourt chaque colonne d'une ligne de matrice
        for j in range(len(m.mat)):
            # si le coeffient vaut 1, les sommets sont adjacents,
            if m.mat[i][j]==1:
                # on ajoute un arc entre les sommets i et j à l'objet␣
↪Graphe_dict
                g.ajouter_arc(chr(rg+i),chr(rg+j))
    return g

G1=graphe_mat_to_dict(M)
print(G1.adj)
```

```
{'A': {'C', 'B'}, 'B': {'C', 'A'}, 'C': {'B', 'E', 'D', 'A'}, 'D': {'C', 'E'},
'E': {'C', 'D'}}
```

[71]:
```
G1.adj==G.adj
```

[71]:
```
True
```

[ ]:
```

```