Structures relationnelles : graphes

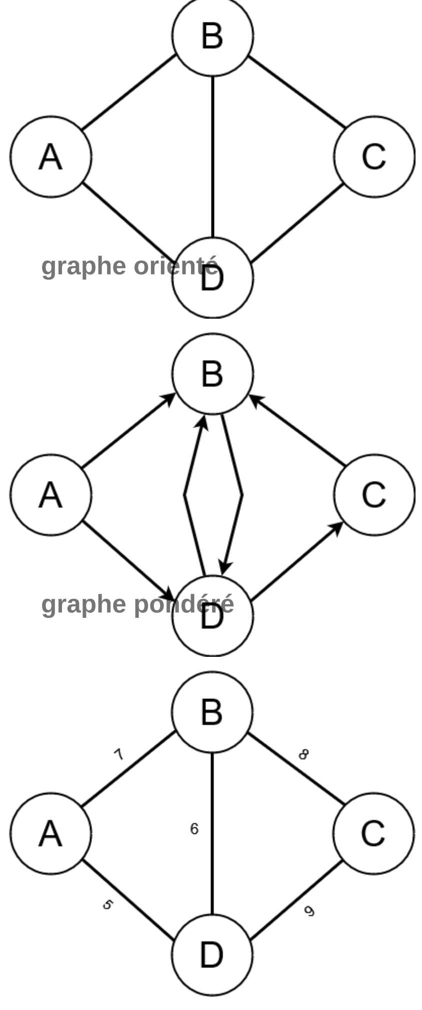
Cours

Un graphe est un type abstrait de données constitué de **sommets** reliés entre eux par des **arêtes** ou **arcs** (selon le type de graphe).

Un graphe peut être :

- **non-orienté**, chaque **arête** peut-être parcourue dans les deux sens.
- **orienté**, chaque **arc** ne peut-être parcouru que dans un seul sens indiqué par une flèche.
- **pondéré**, chaque arête porte une valeur (aussi appelée poids ou coût).

graphe non orienté



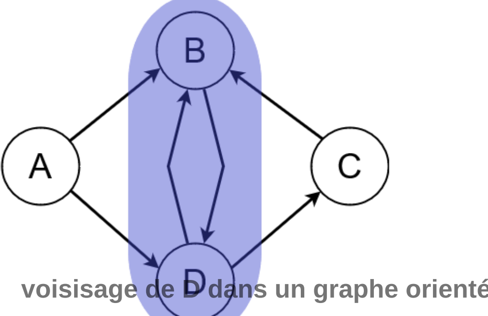
Les graphes trouvent de nombreuses applications en informatique, par exemple dans une modélisation de réseau de routeurs, de réseau social, de réseau routier, de labyrinthe, etc.

Cours

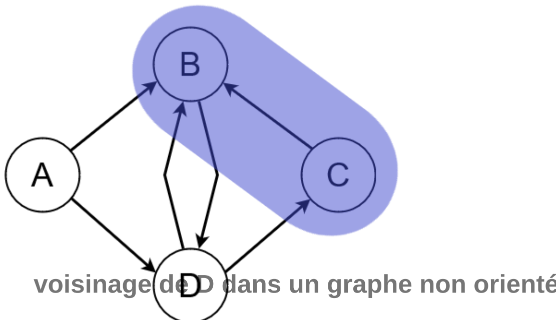
Le nombre de sommets d'un graphe est l'**ordre** du graphe, le nombre d'arêtes est la **taille** du graphe.

Deux sommets reliés entre eux par une arête sont dits **adjacents** ou **voisins**.

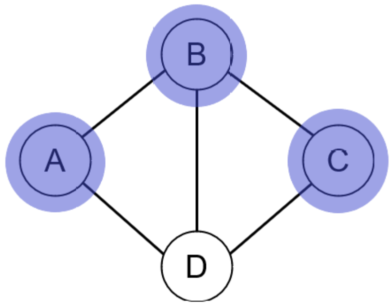
voisinage de A dans un graphe orienté



voisisage de D dans un graphe orienté

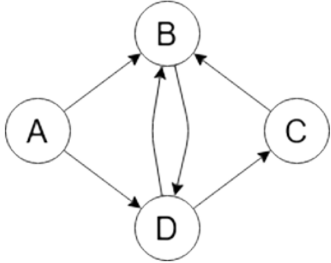


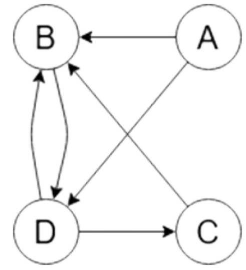
voisinage de D dans un graphe non orienté

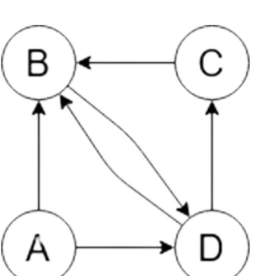


Le **degré** d'un sommet est le nombre d'arêtes issues de ce sommet. La somme des degrés des sommets est le double du nombre d'arêtes du graphe[1].

Un même graphe peut avoir de nombreuses représentations graphiques. Voici trois représentation d'un même graphe :







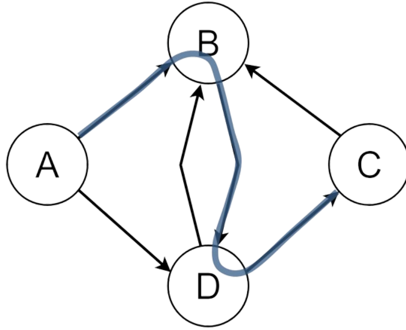
Cours

Un **chemin** (ou **chaine** pour les graphes non orientés) est une suite de sommets reliés par des arêtes.

Un **cycle** (ou **circuit** pour les graphes non orientés) est un chemin fermé.

Remarque : dans un graphe orienté, il peut exister un chemin menant du sommet x au sommet y, alors que l'inverse n'est pas possible.

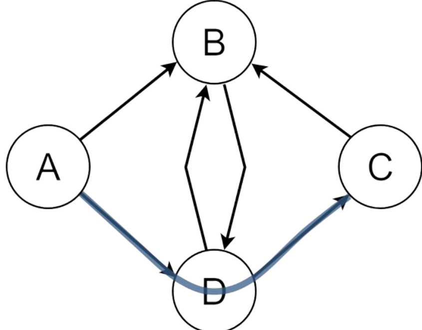
Exemple : chemin menant de A à C, que l'on peut noter A→B→D→C



Cours

La **distance** entre deux sommets d'un arbre est la longueur (nombre d'arêtes) du **chemin le plus court** (s'il y en a un) reliant ces deux sommets.

Exemple : la distance entre A et C est la distance du plus court chemin de A à C (A→D→C), soit 2 (arêtes).

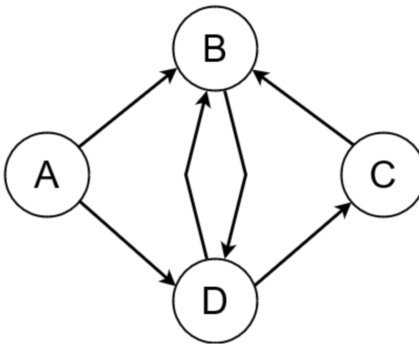


Cours

Un graphe non orienté est **connexe** si pour toute paire (x, y) de sommets, il existe un chemin de x à y.

Un graphe orienté est connexe si le graphe non orienté obtenu en ne tenant pas compte du sens des arêtes est connexe. Un graphe orienté est **fortement connexe** si pour toute paire (x, y) de sommets, il existe un chemin de x à y et un chemin de y à x.

Exemple : Un graphe orienté connexe, mais pas fortement connexe (il n'existe pas de chemin menant à A).



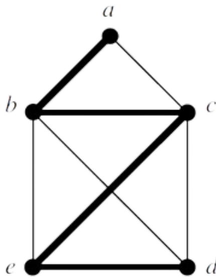
Cours

Un chemin **eulérien** est un chemin dans le graphe qui passe par toutes les arêtes juste une seule fois. Si ce chemin est fermé, on parlera de cycle eulérien. Un graphe est dit eulérien s'il possède un cycle eulérien.

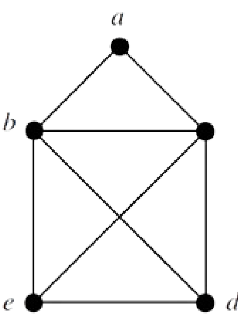
Un chemin **hamiltonien** est un chemin dans le graphe qui passe par tous les sommets une et une seule fois. Si ce chemin est fermé, on parlera de cycle hamiltonien. Un graphe est dit hamiltonien s'il possède un cycle hamiltonien.

Exemples :

- Un graphe avec un chemin hamiltonien *d-e-c-b-a* et un cycle hamiltonien *d-e-b-a-c*.



- Un graphe avec un chemin eulérien *e-b-d-e-c-a-b-c-d* mais pas de cycle eulérien.

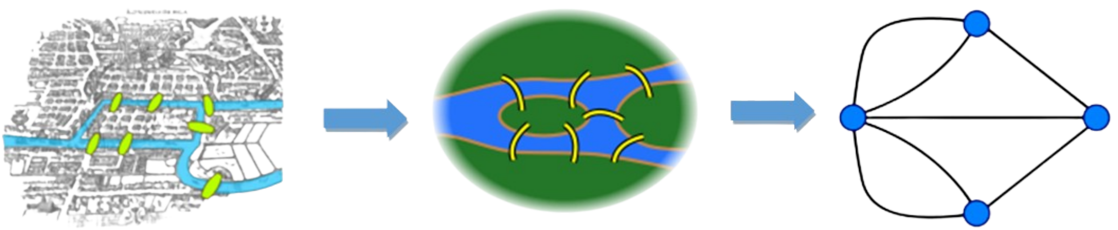


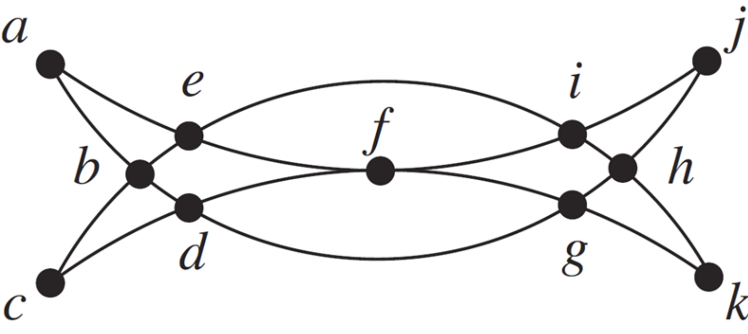
Cours

Théorème d'Euler:

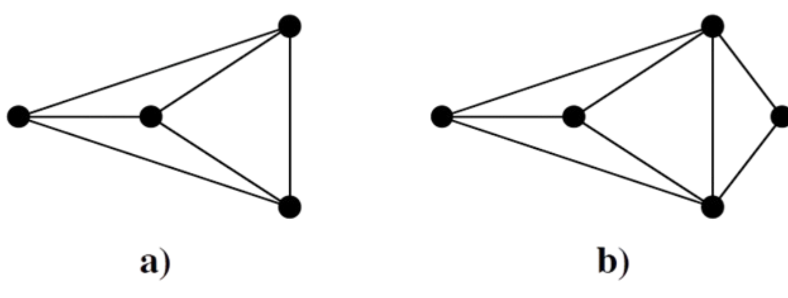
- Un graphe connexe admet un **chemin eulérien** si et seulement si ses sommets sont tous de degré pair sauf au plus deux.
- Un graphe connexe admet un **circuit eulérien** si et seulement si tous ses sommets sont de degré pair.

Le problème des sept ponts de Königsberg[2] est le problème de savoir si on peut traverser chaque pont de la ville de Königsberg en une promenade, une fois sur chaque pont. Comme le montre la figure ci-dessous, le problème se modélise à l'aide d'un graphe comme suit : les ponts constituent les arêtes et les îles et les berges les sommets. Comme ce graphe n'admet pas de chemin eulérien, le problème n'a pas de solutions.



> **Exercice corrigé**
>
> Montrer que le dessin ci-dessous peut être tracé entièrement sans lever le crayon. Proposer un cycle eulérien.
>
> 
>

> **Réponse**

> **Exercice corrigé**
>
> Un facteur désire faire sa tournée sans passer deux fois dans la même rue. Est-ce possible si sa tournée a les profils suivants (où chaque rue est représentée par une arête) :
>
> 
>

> **Réponse**

Interface

Les principales primitives constituant l'interface d'un graphe sont :

- `creer() → graphe` : construire un graphe vide.
- `est_vide() → bool` : vérifier si un graphe est vide ou non.
- `taille() → int` : renvoyer la taille d'un graphe (nombre d'arêtes).
- `ordre() → int` : renvoyer l'ordre d'un graphe (nombre de noeuds).
- `voisins(S) → [sommets]` : renvoyer la liste des sommets voisins de S.
- `chemin(S1, S2) → [sommets]` : renvoyer un chemin allant de S1 à S2 (s'il existe).

Implémentation

Un graphe est entièrement défini par l'ensemble de ses sommets et l'ensemble de ses arêtes. Les sommets d'un graphe peuvent représenter n'importe quel type de donnée. Le choix d'une représentation plutôt qu'une autre dépend des usages que l'on souhaite faire du graphe (construction, parcours, …)

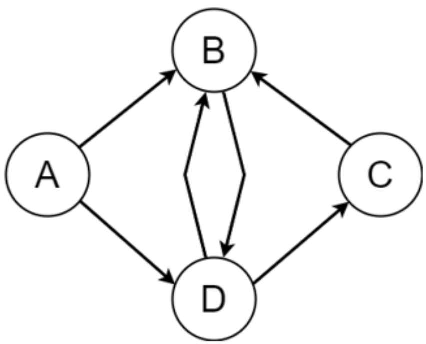
Matrice d'adjacence

Cours

Soit un graphe de sommets, d'indices , , ..., .

La matrice d'adjacence de ce graphe est un tableau à deux dimensions, de taille , contenant des booléens indiquant s'il y a adjacence entre les sommets d'indices et .

Exemple : matrice d'adjacence du graphe orienté ci-dessous :



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

Implémentons un graphe avec une matrice d'adjacence dans une classe `Graphe` :

```python
class Graphe:
    def __init__(self, oriente = False):
        self.matrice = []    # matrice d'adjacence
        self.oriente = oriente
        self.sommets = []    # etiquettes des sommets

    def ordre(self):
        """ renvoie le nombre de sommet"""
        return len(self.matrice)

    def est_vide(self):
        return self.ordre() == 0
```


Pour ajouter un sommet, il faut étendre le tableau d'une ligne et d'une colonne :

```
        def ajouter_sommet(self, s):
        """ ajoute un sommet avec l'étiquette s """
        assert s not in self.sommets, 'sommet déjà existant'
        n = self.ordre()
        for i in range(n):
            self.matrice[i].append(0)   # rajoute 0 à chaque ligne
        self.matrice.append([0] * (n+1)) # rajoute une ligne de 0s
        self.sommets.append(s)
```

Pour ajouter un arc (ou une arête pour un graphe non orienté) entre sommets, il faut trouver la position des sommets dans la matrice d'adjacence :

```
    def ajouter_arc(self, s1, s2):
        """ ajoute une arête (arc) entre les sommet s1 et s2"""
        assert s1 in self.sommets and s2 in self.sommets, 'sommets inexistants'
        i1 = self.sommets.index(s1)
        i2 = self.sommets.index(s2)
        """ ajoute un arc allant de s1 à s2 """
        self.matrice[i1][i2] = 1
        if not self.oriente:
            self.matrice[i2][i1] = 1
```

Créons maintenant le graphe orienté :

```
G = Graphe(oriente=True)
G.ajouter_sommet("A")   # sommet A
G.ajouter_sommet("B")   # sommet B
G.ajouter_sommet("C")   # sommet C
G.ajouter_sommet("D")   # sommet D
G.ajouter_arc("A", "B")  # arc de A vers B
G.ajouter_arc("A", "D")  # arc de A vers D
G.ajouter_arc("B", "D")  # arc de B vers D
G.ajouter_arc("C", "B")  # arc de C vers B
G.ajouter_arc("D", "B")  # arc de D vers B
G.ajouter_arc("D", "C")  # arc de D vers C
```

Et ajoutons des méthodes qui renvoient les voisins et le degré d'un nœud :

```
    def voisins(self, s):
        """ renvoie les voisins de s"""
        n = self.sommets.index(s)
        v = []
        for i in range(self.ordre()):
            if self.matrice[n][i] == 1:
                v.append(self.sommets[i])
        return v

    def degre(self, s):
        """ renvoie le degre de s"""
        n = self.sommets.index(s)
        d = 0
        for i in range(self.ordre()):
            if self.matrice[n][i] == 1:    # les arêtes partant de A
                d += 1
            if self.matrice[i][n] == 1:    # les arêtes arrivant en A
                d += 1
        if self.oriente:
```

```
            return d
        # si le graphe n'est pas orienté, on a compté les arêtes 2 fois
        else:
            return d//2
```

Affichons enfin la matrice avec un peu de formatage :

```
    def __str__(self):
        # ligne avec les noms d'étiquettes
        affiche = '\t' + '\t'.join([str(s) for s in self.sommets]) + '\n'
        # pour chaque ligne
        for ligne in range(len(self.matrice)):
            # on affiche le nom du sommet
            affiche = affiche + str(self.sommets[ligne]) + '\t'
            # les valeurs de chaque colonne
            affiche = affiche + '\t'.join([str(val) for val in self.matrice[ligne]])
            # retour à la ligne
            affiche = affiche + '\n'
        return affiche

>>> print(G)
    A   B   C   D
A   0   1   0   1
B   0   0   0   1
C   0   1   0   0
D   0   1   1   0
```

Avantages et Inconvénients de cette structure :

- La dimension de la matrice est égale au carré du nombre de sommets ($n \times n$), ce qui peut représenter un important espace en mémoire.
- Le fonction pour obtenir les voisins d'un sommet a un coût d'ordre $O(n)$.

Liste ou dictionnaire d'adjacence

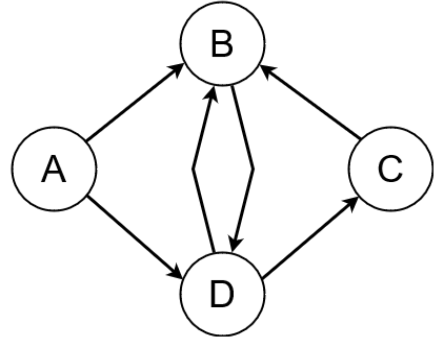
Cours

Dans un **graphe non-orienté**, la **liste d'adjacence** associe chaque sommet à la liste de ses voisins.

Dans un **graphe orienté**, la **liste des successeurs** associe chaque sommet à la liste des sommets que l'on peut atteindre directement par un arc à partir de ce sommet ; et la **liste des prédécesseurs** donne la liste des sommets menant à ce sommet.

Exemple : le dictionnaire d'adjacence du graphe ci-contre :

```
{A: [B, D],
 B: [D],
 C: [B],
 D: [B, C]}
```



L'ordre des voisins d'un sommet n'a pas d'importance, et on ne veut pas dupliquer les voisins, on peut préférer utiliser des ensembles[3] :

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

Implémentons un graphe avec un dictionnaire d'adjacence dans une classe `Graphe` :

```
class Graphe:
    """ """
    def __init__(self, oriente = True):
        self.A = {}                 # Dictionnaire d'adjacence
        self.oriente = oriente      # Graphe orienté ou pas

    def est_vide(self):
        return len(self.A) == 0

    def ordre(self):
        return len(self.A)

    def __repr__(self):
        return str(self.A)

    def ajouter_sommet(self, x):
        if not x in self.A:
            self.A[x] = set()

    def ajouter_arete(self, x, y):
        """ Ajoute une arête entre les sommets x et y """
        self.ajouter_sommet(x)
        self.ajouter_sommet(y)
        self.A[x].add(y)
        if not self.oriente:
            self.A[y].add(x)
```

Ajoutons quelques méthodes :

```
    def voisins(self, x):
        """ Renvoie l'ensemble des voisins du sommet x """
        return self.A[x]

    def arete(self, x, y):
        """ Renvoie True s'il existe une arête entre les sommets x et y """
        return x in self.A[y]
```

Et créons maitenant le graphe orienté :

```
g = Graphe()
g.ajouter_arete('A', 'B')
g.ajouter_arete('A', 'D')
g.ajouter_arete('B', 'D')
g.ajouter_arete('C', 'B')
g.ajouter_arete('D', 'B')
g.ajouter_arete('D', 'C')
print(g. A)
```

On peut ajouter des méthodes qui renvoient les degrés d'un nœud et ses voisins :

```
    def degre(self, x):
        return len(self.A[x])

    def voisin(self, x):
        return self.A[x]
```

Avantages et Inconvénients de cette structure :

- les données représentées par les sommets peuvent être directement les clés du dictionnaire, sous réserve d'être hashable (dans le cas contraire, on utilisera, comme pour la matrice d'adjacence, des indices).
- le dictionnaire d'un graphe contenant beaucoup d'arêtes occupe plus de mémoire que la matrice d'adjacence.
- l'obtention des voisins d'un sommet est cette fois-ci une opération en temps constant.

---

1. En ajoutant les degrés de chaque sommet (c'est à dire le nombre d'arêtes issues de ce sommet), on comptabilise deux fois chaque arête (une fois avec le sommet d'une extrémité et une seconde fois avec le sommet de l'autre extrémité de l'arête). Il en découle que la somme des degrés des sommets est nécessairement paire et donc que le nombre de sommets de degré impair est pair. ↩
2. voir "Briller en Société #27: Les 7 ponts de Königsberg" sur https://www.youtube.com/watch?v=F1G4srEXq2s ↩
3. Les ensembles Python, de type `set`, ne sont pas au programme de NSI. Un ensemble est une collection d'éléments non ordonnés, non indexés, qui n'accepte pas de contenir plusieurs fois le même élément. Les éléments d'un ensemble sont écrits entre accolades mais un ensemble est défini par la fonction `set()` (et pas par `{}` qui permet de définir un dictionnaire vide). On peut ajouter un élément avec la méthode `.add()` et supprimer un élément avec `.remove()`. On peut utiliser la fonction `len()` et le mot clé `in` comme pour les autres types construits. ↩