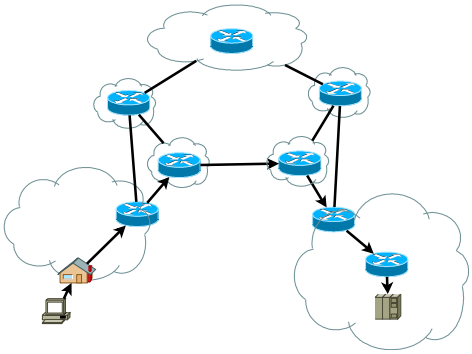
Les graphes

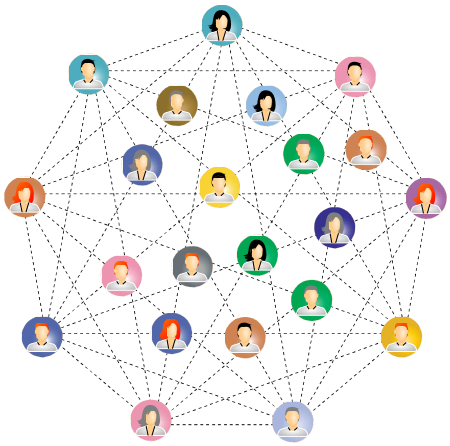
Les graphes sont une structure de données très riche permettant de modéliser des situations variées de relations entre un ensemble d'entités :

- entre les ordinateurs du réseau internet ;



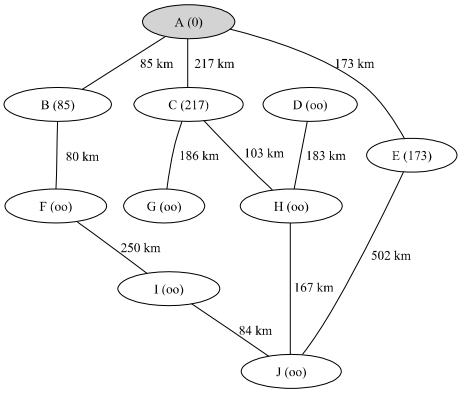
Source : Wikipedia, Mro, licence CC BY-SA 3.0

- entre des personnes sur un réseau social ;



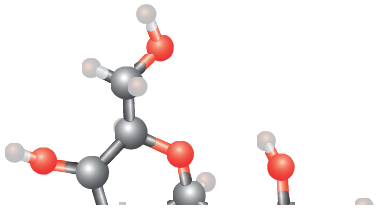
Source : Pixabay

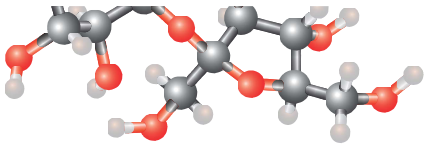
- entre les villes dans un réseau routier ou de distribution ;



Source : Wikipedia, HB, licence CC BY-SA 3.0

- entre les atomes d'une molécule ;





Source : Wikipedia, William Crochot, licence CC BY-SA 4.0

- entre les ressources du Web (les relations sont les liens hypertextes) ;
- entre deux situations dans un jeu ;
- etc.

Les relations peuvent être orientées ou non.

Définitions et vocabulaire

Un graphe est constitué d'un ensemble de **sommets** et dans le cas orienté d'un ensemble d'**arcs** reliant chacun un sommet à un autre, dans le cas non orienté d'un ensemble d'**arêtes** entre deux sommets.

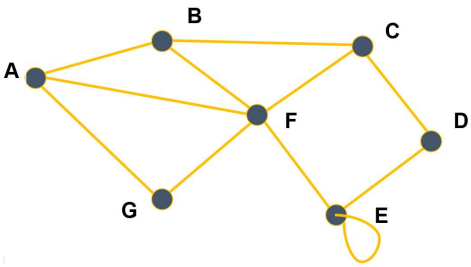
Mathématiquement, un graphe G est donc un couple formé de deux ensembles : $X = \{x_1, x_2, \dots, x_n\}$ dont les éléments sont appelés les sommets et un ensemble $A = \{a_1, a_2, \dots, a_m\}$ dont les éléments sont appelés les arêtes dans le cas non orienté ou les arcs dans le cas orienté. Une arête (ou un arc) a_i est un couple de deux sommets, par exemple : $a_i = (x_2, x_5)$ symbolise le lien (arête ou arc) entre les sommets x_2 et x_5. On peut noter $G = (X, A)$.

Vocabulaire des graphes *non orientés*

Dans le cas des graphes non orientés, les relations entre deux sommets se font dans les deux sens. On appelle ses relations des **arêtes** (*edges* en anglais), et on a les définitions suivantes.

- **Sommets adjacents** : deux sommets sont adjacents s'ils sont reliés entre eux par une arête. On dit que l'arête est incidente aux deux sommets.
- **Voisins d'un sommet** x : ce sont tous les sommets reliés à x par une arête.
- **Degré d'un sommet** x : nombre d'arêtes incidentes au sommet, on le note $d(x)$.
- **Chaîne** : séquence ordonnée d'arêtes telle que chaque arête a une extrémité en commun avec l'arête suivante.
- **Cycle** : dans un graphe non orienté, un cycle est une suite d'arêtes consécutives (chaîne) dont les deux sommets extrémités sont identiques.
- **Boucle** : il peut exister des arêtes entre un sommet x et lui-même. Elles sont appelés *boucles*.

Exemple



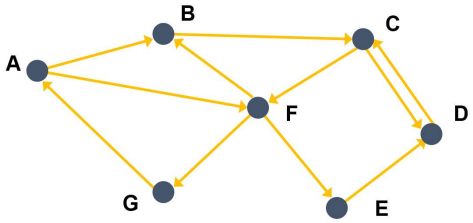
- Ce graphe *non orienté* est donné par :
  - un ensemble de sommets : $\{A, B, C, D, E, F, G\}$,
  - un ensemble d'arêtes : $\{(A, B), (A, F), (A, G), (B, C), (B, F), \dots\}$.
- Les sommets A et B sont adjacents mais B et D ne le sont pas.
- Les voisins du sommet A sont B, F et G.
- Le degré du sommet A est égal à 3 ($d(A) = 3$).
- A, B, C, D est une chaîne, A, B, F aussi.
- A, F, G, A est un cycle.
- Il y a une boucle (E, E). Le degré de E est égal à 4.

Vocabulaire des graphes *orientés*

Dans le cas des graphes orientés, les arêtes ont un sens et elles sont appelées **arcs**. Par exemple, l'arête $a = (x, y)$ indique qu'il y a un arc d'origine x et d'extrémité finale y. De plus, on a les définitions suivantes.

- **Successeurs et prédécesseurs d'un sommet** x : dans un graphe orienté on ne parle plus de *voisins* d'un sommet mais de ses successeurs et de ses prédécesseurs : le successeurs de x sont tous les sommets y tels qu'il existe un arc (x, y) (de x vers y) et les prédécesseurs de x sont tous les sommets w tels qu'il existe un arc (w, x) (de w vers x).
- **Chemin** : séquence ordonnée d'arcs consécutifs (on parlait de *chaîne* dans un graphe non orienté).
- **Circuit** : dans un graphe orienté, un circuit est une suite d'arcs consécutifs (chemin) dont les deux sommets extrémités sont identiques.
- **Degré d'un sommet** x : cette notion existe aussi dans le cas des graphes orientés. On distingue le degré entrant d'un sommet x (noté $d_-(x)$ = nombre de prédécesseurs de x) et le degré sortant d'un sommet x (noté $d_+(x)$ = nombre de sucesseurs de x). Le degré d'un sommet x vaut $d(x) = d_+(x) + d_-(x)$.
- **Boucle** : ce sont les arcs entre un sommet et lui-même.

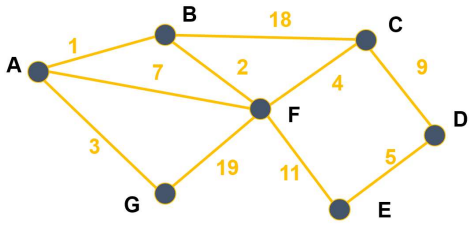
Exemple



- Ce graphe *orienté* est donné par :
  - un ensemble de sommets : $\{A, B, C, D, E, F, G\}$,
  - un ensemble d'arcs : $\{(A, B), (A, F), (B, C), (C, F), (C, D), \ldots\}$.
- Les successeurs de A sont les sommets B et F, le seul prédécesseur de A est G.
- Le degré du sommet A est égal à 3 ($d(A) = d_+(A) + d_-(A) = 2 + 1 = 3$).
- A, B, C, D est un chemin mais A, B, F n'en est pas un car il n'y a pas d'arc (B, F) (de B vers F).
- A, F, G, A est un circuit.
- Il n'y a pas de boucle ici.

Graphes valués

Certains graphes (orientés ou non) sont dits *valués* : on ajoute un coût (ou valuation, ou poids) à chaque arête/arc. Dans le cas d'un graphe représentant un réseau routier, le coût sur chaque arête pourrait, par exemple, être la distance entre deux villes.



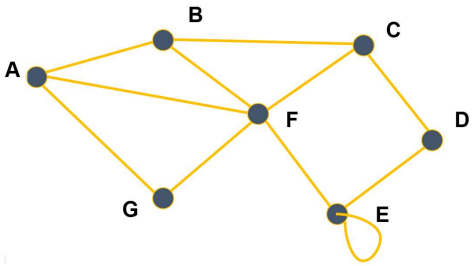
Représentations d'un graphe

Représentation par matrice d'adjacence

Une **matrice** M est un tableau de nombres, qui peut être représenté en machine par un tableau de tableaux (ou une liste de listes) noté `matrice`. Chaque nombre de cette matrice est repéré par son numéro de ligne i et son numéro de colonne j. On note ce nombre $M_{i,j}$ et on peut y accéder par l'instruction `matrice[i][j]`.

Un graphe à n sommets peut être représentée par une **matrice d'adjacence** de taille $n \times n$, où la valeur du coefficient d'indice i, j dépend de l'existence d'une arête ou d'un arc reliant les sommets i et j.

Exemple (graphe orienté) : Si les sommets $A, B, C, \ldots$ du graphe



sont respectivement numérotés $0, 1, 2$, etc. alors sa matrice d'adjacence est

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

Par exemple, le sommet C correspond à la troisième ligne. Celle-ci contient dans cet ordre les nombres 0, 1, 0, 1, 0, 1, 0 donc cela signifie qu'il y a des **arêtes** (C, B), (C, D) et (C, F) (les 1) mais pas entre C et les sommets A, C, E et G (les 0).

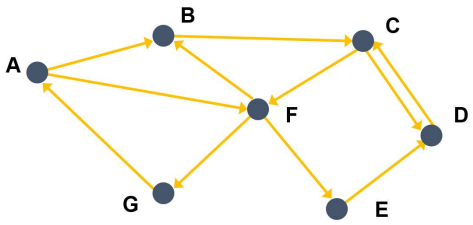
Cette matrice peut être mémorisée en machine par le tableau de tableaux suivant.

In [1]:
```
matrice = [
    [0, 1, 0, 0, 0, 1, 1],
    [1, 0, 1, 0, 0, 1, 0],
    [0, 1, 0, 1, 0, 1, 0],
    [0, 0, 1, 0, 1, 0, 0],
    [0, 0, 0, 1, 1, 1, 0],
    [1, 1, 1, 0, 1, 0, 1],
    [1, 0, 0, 0, 0, 1, 0]
]
```

Dans le cas d'un graphe non orienté, la matrice d'adjacence est nécessairement *symétrique* par rapport à sa diagonale : on a $M_{i,j} = M_{j,i}$.

Exemple (graphe non orienté) :

C'est le même principe en faisant attention au sens des arcs : $M_{i,j} = 1$ s'il y a un arc d'origine i et d'extrémité j et $M_{i,j} = 0$ sinon. Ainsi, le graphe



a pour matrice d'adjacence

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

Par exemple, le sommet C correspond à la troisième ligne. Celle-ci contient dans cet ordre les nombres 0, 0, 0, 1, 0, 1, 0 donc cela signifie qu'il y a des **arcs** (C, D) et (C, F) (les 1) mais pas entre C et les autres sommets (les 0).

> Comme les arcs ont un sens, la matrice d'adjacence d'un graphe orienté n'est généralement pas symétrique.

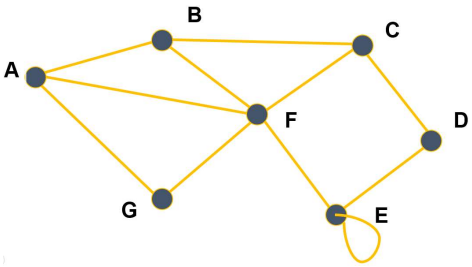
Représentation par listes des successeurs

Une autre façon de représenter un graphe est d'associer à chaque sommet la liste des sommets auxquels il est relié. Dans le cas d'un graphe orienté, on parle de **liste de successeurs**, alors que dans le cas d'un graphe non orienté on parle de **liste de voisins**.

Une façon simple et efficace est d'utiliser un *dictionnaire* où chaque sommet est associé à la liste de ses successeurs/voisins.

Exemples :

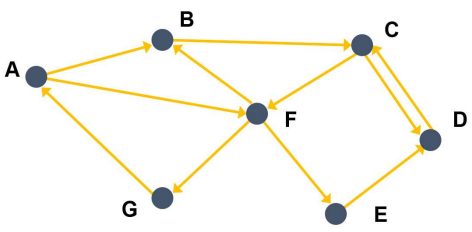
- Le graphe orienté



peut être représenté par le dictionnaire suivant, où les clés sont les sommets et les valeurs sont les listes de voisins.

In [2]:
```
dico1 = {
    "A": ["B", "F", "G"],
    "B": ["A", "C", "F"],
    "C": ["B", "D", "F"],
    "D": ["C", "E"],
    "E": ["D", "E", "F"],
    "F": ["A", "B", "C", "E", "G"],
    "G": ["A", "F"]
}
```

- Le graphe orienté



peut être représenté par le dictionnaire suivant, où les clés sont les sommets et les valeurs sont les listes de successeurs.

In [3]:
```
dico2 = {
    "A": ["B", "F"],
    "B": ["C"],
```

```
    "C": ["D", "F"],
    "D": ["C"],
    "E": ["D"],
    "F": ["B", "E", "G"],
    "G": ["A"]
}
```

Efficacité des représentations

La matrice d'adjacence est simple à mettre en oeuvre mais nécessite un espace mémoire proportionnel à $n \times n$ (où n est le nombre de sommets). Ainsi, un graphe de 1000 sommets nécessitent une matrice d'un million de nombres même si le graphe contient peu d'arêtes/arcs. Pour le même graphe contenant peu d'arêtes/arcs, le dictionnaire ne mémoriserait pour chaque sommet que les voisins/successeurs (les 1) sans avoir à mémoriser les autres (les 0). En revanche, pour un graphe contenant beaucoup d'arêtes/arcs, la dictionnaire occuperait plus d'espace mémoire que la matrice d'adjacence.

Cela implique en outre que l'accès aux voisins/sucesseurs d'un sommet est plus rapide avec le dictionnaire car il n'est pas nécessaire de parcourir toute la ligne de la matrice (n valeurs) alors même que celle-ci peut ne contenir que très peu de 1.

De plus, l'utilisation d'un dictionnaire permet de nommer les sommets sans ambiguité et ne les limite pas à des entiers comme c'est le cas pour la matrice d'adjacence (même si on peut associer chacun de ces entiers au sommet correspondant, ce que nous avons fait précédemment).

Enfin, au lieu d'utiliser le type liste (`list` de Python ici) pour mémoriser les voisins/successeurs, on peut avantageusement utiliser le type ensemble (type prédéfini `set` de Python) qui est une structure de données permettant un accès plus efficace aux éléments (l'implémentation se fait par des *tables de hachage*, hors programme de NSI).

> **A faire** : tous les exercices du notebook d'activités !

Implémentations

La fin de ce cours résume une partie de ce qui a été fait en exercices, notamment deux implémentations du type `GrapheNonOriente` défini par l'interface suivante :

- `faire_graphe(sommets)` pour construire un graphe (sans les arêtes) à partir de la liste `sommets` de ses sommets.
- `ajouter_arete(G, x, y)` pour ajouter une arête entre les sommets `x` et `y` du graphe `G`.
- `sommets(G)` pour accéder à la liste des sommets du graphe `G`.
- `voisins(G, x)` pour accéder à la liste des voisins du sommet `x` du graphe `G`.

La première implémentation se fait par une classe `GrapheNoMa` s'appuyant sur la représentation par une matrice d'adjacence et la seconde par une classe `GrapheNoLs` s'appuyant sur les listes de successeurs (qui sont les voisins dans le cas d'un graphe non orienté).

On termine en présentant comment passer d'une représentation à l'autre.

Par une matrice d'adjacence

Voici la classe `GrapheNoMa` s'appuyant sur une matrice d'adjacence.

In [4]:
```
class GrapheNoMa:
    def __init__ (self, sommets):
        self.som = sommets
        self.dimension = len(sommets)
        self.adjacence = [[0 for i in range(self.dimension)] for j in range(self.dimension)]

    def ajouter_arete(self, x, y):
        i = self.som.index(x)
        j = self.som.index(y)
        self.adjacence[i][j] = 1
        self.adjacence[j][i] = 1

    def sommets(self):
        return self.som

    def voisins(self, x):
        i = self.som.index(x)
        return [self.som[j] for j in range(self.dimension) if self.adjacence[i][j] == 1]
```

Par une liste de successeurs

Voici la classe `GrapheNoLs` s'appuyant sur un dictionnaire contenant les listes de successeurs de chaque sommet.

In [5]:
```python
class GrapheNoLs:
    def __init__ (self, sommets):
        self.som = sommets
        self.dic = {sommet: [] for sommet in self.som} # création par compréhension

    def ajouter_arete(self, x, y):
        if y not in self.dic[x]:
            self.dic[x].append(y)
        if x not in self.dic[y]:
            self.dic[y].append(x)

    def sommets(self):
        return self.som

    def voisins(self, x):
        return self.dic[x]
```

On peut alors créer des graphes comme objets de ces deux classes et leur ajouter des arrêtes.

In [6]:
```python
# graphe g1 représenté par une matrice d'adjacence
g1 = GrapheNoMa(["a", "b", "c", "d"])
g1.ajouter_arete("a", "b")
g1.ajouter_arete("a", "c")
g1.ajouter_arete("c", "d")

# graphe g2 représenté par liste de successeurs
g2 = GrapheNoLs(["a", "b", "c", "d"])
g2.ajouter_arete("a", "b")
g2.ajouter_arete("a", "c")
g2.ajouter_arete("c", "d")
```

On peut accéder aux graphes à travers les fonctions de l'interface du type abstrait de manière totalement identique.

In [7]:
```python
print(g1.sommets())
print(g1.voisins("c"))
```

```
['a', 'b', 'c', 'd']
['a', 'd']
```

In [8]:
```python
print(g2.sommets())
print(g2.voisins("c"))
```

```
['a', 'b', 'c', 'd']
['a', 'd']
```

En Python, un utilisateur malin pourra observer la façon dont sont mémorisées les graphes dans les deux cas :

In [9]:
```python
g1.adjacence
```

Out[9]:
```
[[0, 1, 1, 0], [1, 0, 0, 0], [1, 0, 0, 1], [0, 0, 1, 0]]
```

In [10]:
```python
g2.dic
```

Out[10]:
```
{'a': ['b', 'c'], 'b': ['a'], 'c': ['a', 'd'], 'd': ['c']}
```

Mais nous avons vu qu'il est possible de palier à ce problème en définissant une méthode de représentation identique dans chacune des deux classes pour *masquer* cette différence d'implémentation, qui importe peu à l'utilisateur de la classe.

Passage d'une représentation à l'autre

Les deux implémentations sont totalement équivalentes et on peut passer de l'une à l'autre simplement en énumérant les sommets et les voisins depuis une représentation tout en construisant l'autre représentation.

Par exemple, la fonction suivante permet de passer d'une matrice d'adjacence à une liste de successeurs (la fonction de traduction réciproque est similaire).

In [11]:
```python
def ma_to_ls(gma):
    gls = GrapheNoLs(gma.sommets())
    for x in gma.sommets():
        for y in gma.voisins(x):
            gls.ajouter_arete(x,y)
    return gls
```

In [12]:
```python
g3 = ma_to_ls(g1)
print("représentation de départ :", g1.adjacence)
print("traduction :", g3.dic)
```

```
représentation de départ : [[0, 1, 1, 0], [1, 0, 0, 0], [1, 0, 0, 1], [0, 0, 1, 0]]
traduction : {'a': ['b', 'c'], 'b': ['a'], 'c': ['a', 'd'], 'd': ['c']}
```

> On peut implémenter le type abstrait `GrapheOriente` de façon quasiment similaire (fait en exercices).

Bilan

- Le **graphe** est une structure de données permettant de modéliser de nombreuses situations de la vie courante.
- Il est définit par un ensemble de *sommets* et de *liaisons*. Ces liaisons peuvent être orientées ou non et on les appelle alors respectivement des **arcs** ou des **arêtes**.
- On peut représenter en machine un graphe par une *matrice d'adjacence* ou par un dictionnaire contenant les *listes de successeurs* (de chaque sommet).
- La programmation par des classes de deux implémentations d'un graphe non orienté (resp. orienté) a prouvé que celles-ci étaient indépendantes de l'interface du type abstrait `GrapheNonOriente` (resp. `GrapheOriente`) et qu'elles sont équivalentes, on peut notamment passer de l'une à l'autre facilement.
- Nous verrons dans un prochaine chapitre différents algorithmes sur les graphes (parcours en profondeur d'abord, en largeur d'abord, repérer des cycles, recherche de chemin dans un graphe).

Références :

- Equipe pédagogique DIU EIL, Université de Nantes.

Germain BECKER, Lycée Mounier, ANGERS