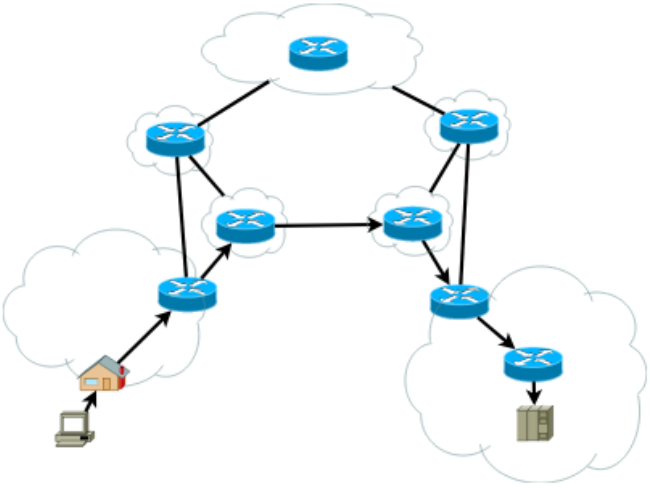
# Chapitre 13 - Graphes et algorithmes

## PARTIE 1 : Les graphes

## I - Introduction

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 ;

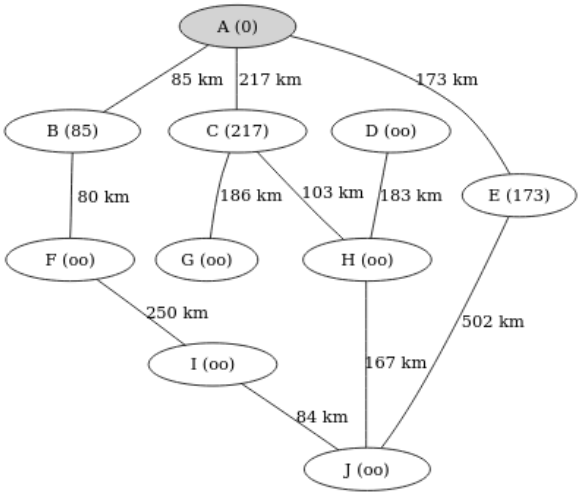


Source : Wikipedia, Mro, licence CC BY-SA 3.0

- entre des personnes sur un réseau social ;

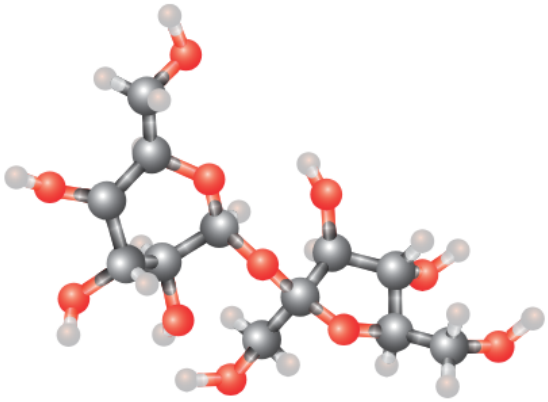
Source : Pixabay

- 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.

# II - Définitions et vocabulaire

**Définition 1**

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 :

- $X = \{x_1, x_2, \ldots, x_n\}$ dont les éléments sont appelés les sommets.
- $A = \{a_1, a_2, \ldots, a_m\}$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 $2$ et $5$ . On peut noter $G = (X, A)$.

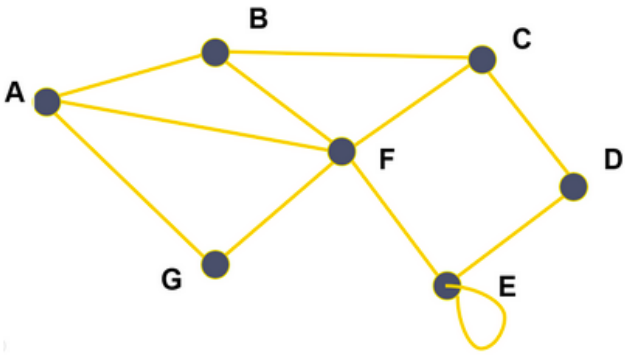
## Vocabulaire des graphes non orientés

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 : $\{(A, B), (A, F), (A, G), (B, F), (B, C)...\}$
- 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.



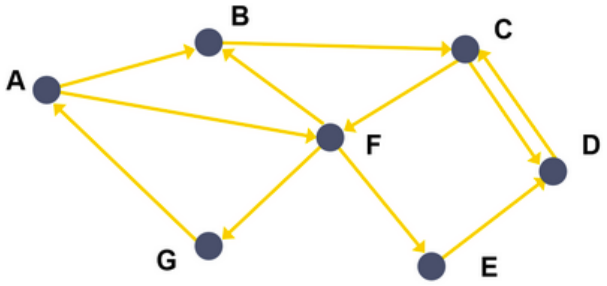
## Vocabulaire des graphes orientés

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 : dans un graphe orienté on ne parle plus de voisins d'un sommet mais de ses successeurs et de ses prédécesseurs :
  - ⋆ les successeurs de $x$ de $x$ sont tous les sommets $y$ tels qu'il existe un arc $(x, y)$ de $x$ vers $y$.
  - ⋆ 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 (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.

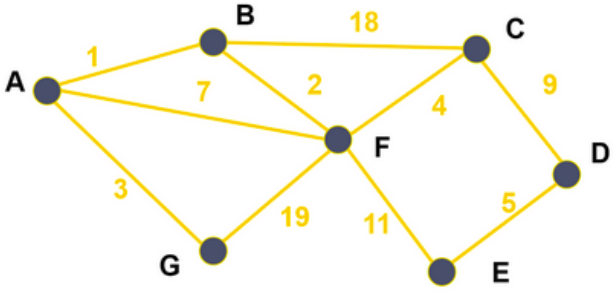
**Exemple**

- Ce graphe non orienté est donné par :
  - ⋆ un ensemble de sommets : $\{A, B, C, D, E, F, G\}$
  - ⋆ un ensemble d'arêtes : $\{(A, B), (A, F), (B, C), (F, B), (C, D)...\}$
- 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 : $d(A) = d_-(A) + d_+(A) = 1 + 2 = 3$.
- $A, B, C, D$ 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.



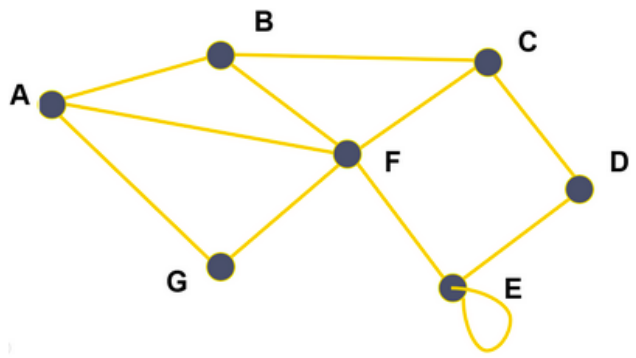
# III - Représentations d'un graphe

## 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 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 non orienté)** :




Si les sommets $A, B, C, D, E, F, G$ du graphe sont respectivement numérotés $1, 2, 3, 4, 5, 6, 7$ alors sa matrice d'adjacence est :

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

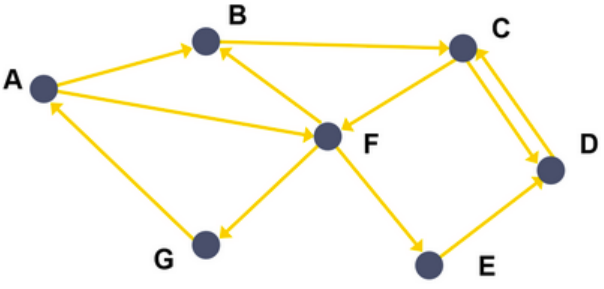
Par exemple, le sommet $C$ correspond à la troisième ligne. Celle-ci contient dans cet ordre les nombres 0, 1, 0, 1, 0, 1, 0 donc cela signifie qu'il y a des **arêtes** $C, B)$, $(C, D)$ et $C, F)$ (les 1) mais pas entre $C$ et les sommets $A$, $C$, $E$ et $G$ (les 0).

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

```
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]
]
```

**Remarque** : 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 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).

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

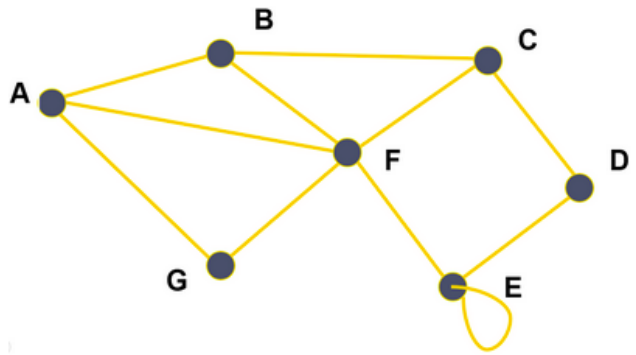
## Représentation par listes des successeurs

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.

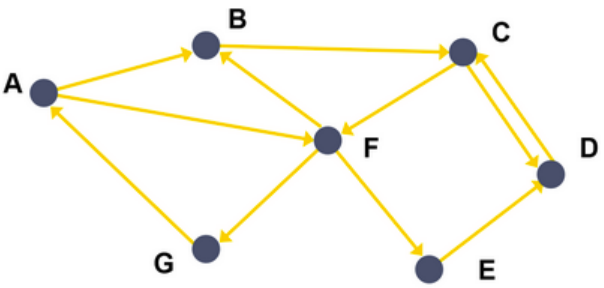
**Exemples :**

Le **graphe non orienté** peut être représenté par le dictionnaire suivant, où les clés sont les sommets et les valeurs sont les listes de voisins.




```
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.




```
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 nécessite 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).

## IV - 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`.

$\Rightarrow$ La première implémentation se fait par une classe `GrapheNoMa` s'appuyant sur la représentation par une matrice d'adjacence

$\Rightarrow$ 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).

### Par une matrice d'adjacence

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

```
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.

```
class GrapheNoLs:
    def __init__ (self, sommets):
        self.som = sommets
        self.dic = {sommet: [] for sommet in self.som} # creation par comprehension

    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.

```
>>> g1 = GrapheNoMa(["a", "b", "c", "d"])  # implementation par matrice d'adjacence
>>> g1.ajouter_arete("a", "b")
>>> g1.ajouter_arete("a", "c")
>>> g1.ajouter_arete("c", "d")
```

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

```
>>> g1.sommets()
['a', 'b', 'c', 'd']
>>> g1.voisins("c")
['a', 'd']
```

```
>>> g2.sommets()
['a', 'b', 'c', 'd']
>>> g2.voisins("c")
['a', 'd']
```

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

```
>>> g1.adjacence
[[0, 1, 1, 0], [1, 0, 0, 0], [1, 0, 0, 1], [0, 0, 1, 0]]
>>> g2.dic
{'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).

```
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
```

```
>>> g3 = ma_to_ls(g1)
>>> g1.adjacence  # representation de depart
[[0, 1, 1, 0], [1, 0, 0, 0], [1, 0, 0, 1], [0, 0, 1, 0]]
>>> g3.dic  # traduction
{'a': ['b', 'c'], 'b': ['a'], 'c': ['a', 'd'], 'd': ['c']}
```

# PARTIE 2 : Algorithmes sur les graphes

Un des premiers algorithmes qu'on doit savoir utiliser sur un graphe est celui de son parcours. **Parcourir un graphe, c'est visiter ses différents sommets, afin de pouvoir opérer une action tour à tour sur eux.**

Les deux algorithmes fondamentaux permettant de parcourir un graphe s'appellent :

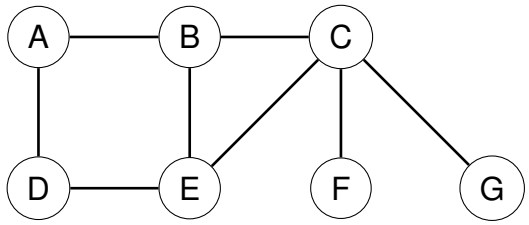
- **le parcours en profondeur d'abord** ;
- **le parcours en largeur d'abord**.

Selon les actions opérées au cours d'un parcours, on peut détecter des cycles dans le graphe, trouver le chemin le plus court entre deux sommets, calculer la distance entre deux sommets, etc.

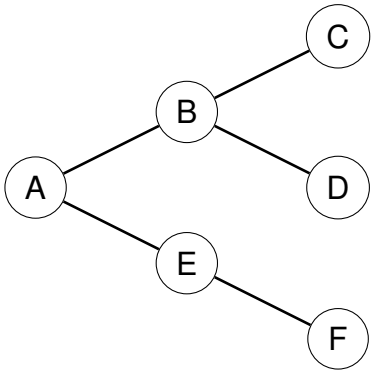
Les algorithmes sur les graphes sont très utilisés dans la vie courante, ils permettent par exemple :

- le routage des paquets de données dans un réseau ;
- de trouver le chemin le plus court entre deux villes (utilisé par les GPS) ;
- de sortir d'un labyrinthe ;
- etc.

Dans la suite, on considérera les deux graphes `g1` et `g2` suivants.




Graphe `g1`




Graphe `g2`

Ils seront représentés par listes de successeurs et implémentés par des dictionnaires.

```
g1 = {
    "A": ["B", "D"],
    "B": ["A", "C", "E"],
    "C": ["B", "E", "F", "G"],
    "D": ["A", "E"],
    "E": ["B", "C", "D"],
    "F": ["C"],
    "G": ["C"]
}

g2 = {
    "A": ["B", "E"],
    "B": ["A", "C", "D"],
    "C": ["B"],
    "D": ["B"],
    "E": ["A", "F"],
    "F": ["E"]
}
```

Les sommets d'un graphe sont les clés du dictionnaire. En particulier, on peut accéder aux voisins/successeurs d'un sommet en utilisant sa clé.

```
>>> g1["A"]  # voisins du sommet A dans g1
['B', 'D']
```

# V - Parcours en profondeur et en largeur

## Comparaison des deux algorithmes

Ces deux algorithmes ont le même but : explorer tous les sommets atteignables d'un graphe à partir d'un sommet de départ. L'idée est d'explorer les voisins (ou successeurs) rencontrés au fur et à mesure en marquant les sommets visités pour ne pas tourner en rond.

### Définition 2

**Parcours en profondeur d'abord** : à partir d'un sommet, on explore un de ses voisins (ou successeurs), et ainsi de suite. S'il n'y a plus de voisins, on revient au sommet précédent et on passe à un autre de ses voisins. Cette façon de faire implique que chaque "branche" est explorée jusqu'au bout, avant de revenir sur nos pas, d'où le nom de parcours en profondeur.

### Définition 3

**Parcours en largeur d'abord** : à partir d'un sommet, on explore tous ses voisins (ou successeurs), puis on explore tous les voisins de ces voisins, et ainsi de suite. Le parcours balaie ainsi chaque "branche" au même rythme, d'où le nom de parcours en largeur.

La seule différence entre ces deux algorithmes est donc l'ordre dans lequel les voisins sont traités. Cela permet d'écrire le même algorithme pour les deux parcours, en changeant juste la collection qui stocke les sommets à visiter :

- une **pile** pour le parcours en profondeur
- une **file** pour le parcours en largeur.

#### Principe de l'algorithme de parcours en profondeur

**Algorithme de parcours en profondeur**

- On choisit un sommet de départ
- On l'empile
- Tant que la pile n'est pas vide :
  - On dépile son sommet
  - S'il n'a pas encore été visité on le marque et on empile tous ses voisins non encore visités
  - Sinon, on ne fait rien (on passe donc directement à l'itération suivante)

En stockant les sommets encore à visiter dans une pile, on s'assure que ce sont les derniers sommets découverts qui vont être visités en premier (LIFO, Last In First Out), cela correspond au parcours en profondeur.
Contrairement au parcours sur les arbres, il est important ici de "marquer" les sommets déjà visités (plusieurs chemins peuvent mener à un même sommet).
Dans le code python suivant, on utilise ici un dictionnaire pour marquer les sommets (en les ajoutant dans le dictionnaire). La valeur d'un sommet visité (ici True) n'a pas d'importance. On aurait donc pu utiliser ici une structure de données abstraite plus simple : l'ensemble.

```
def parcours_prof(graphe, debut):
    visites = {}
    pile = [debut]
    while len(pile) > 0:
        s = pile.pop()
        if s in visites:   # si s a deja ete visite
            continue       # on passe a l'iteration suivante
        visites[s] = True  # sinon l'iteration en cours se poursuit
        for voisin in graphe[s]:
            if voisin not in visites:
                pile.append(voisin)
    return visites
```

### Principe de l'algorithme de parcours en largeur

C'est simple, il suffit de remplace la pile par une file !

**Algorithme de parcours en largeur**

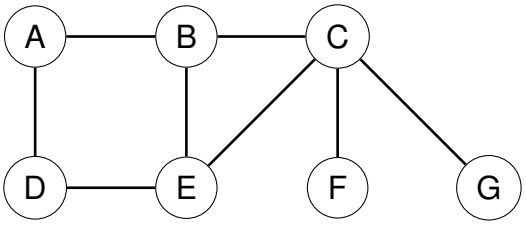
- On choisit un sommet de départ
- On l'enfile
- Tant que la file n'est pas vide :
  - On defile son premier élément
  - S'il n'a pas encore été visité on le marque et on enfile tous ses voisins non encore visités
  - Sinon, on ne fait rien (on passe donc directement à l'itération suivante)

En stockant les sommets encore à visiter dans une file, on s'assure que ce sont les premiers sommets découverts qui vont être visités en premier (FIFO, First In First Out), cela correspond au parcours en largeur :

```
def parcours_larg(graphe, debut):
    visites = {}
    file = [debut]
    while len(file) > 0:
        s = file.pop(0)
        if s in visites:   # si s a deje ete visite
            continue       # on passe a l'iteration suivante
        visites[s] = True  # sinon l'iteration en cours se poursuit
        for voisin in graphe[s]:
            if voisin not in visites:
                file.append(voisin)
    return visites
```

**Remarque :** Ici, les structures de pile et de file sont implémentées par des listes Python mais on peut bien sûr utiliser n'importe quelle autre implémentation.

**Exemple :**
En observant l'ordre d'ajout des clés dans le dictionnaire, on peut voir l'ordre des sommets visités par chaque parcours.



Graphe `g1`

```
>>> parcours_prof(g1, "A")
{'A': True, 'D': True, 'E': True, 'C': True, 'G': True, 'F': True, 'B': True}
>>> parcours_larg(g1, "A")
{'A': True, 'B': True, 'D': True, 'C': True, 'E': True, 'F': True, 'G': True}
```

- Le parcours en profondeur donne l'ordre de parcours $A \to D \to E \to C \to G \to F \to B$.
- Le parcours en largeur donne l'ordre de parcours $A \to B \to D \to C \to E \to F \to G$.

**Remarque :** L'odre de parcours dépend de l'ordre dans lequel sont stockés les voisins dans les listes de voisins/successeurs car celui-ci détermine l'ordre d'ajout dans la pile ou la file. Il y a donc plusieurs réponses possibles pour un même parcours.

## Version récursive du parcours en profondeur

En réalité, le parcours en profondeur est naturellement récursif. En effet, un part du sommet de départ et on explore l'un de ses voisins, puis on explore l'un des voisins du voisin, ainsi de suite jusqu'à ce qu'on ne trouve plus de voisins (on arrive à un "cul-de-sac"), auquel cas on revient au sommet précédent.

On peut traduire l'algorithme de parcours en profondeur de la façon très simple suivante : si un sommet n'est pas visité, on le marque et on parcourt récursivement tous ses voisins.

Comme souvent, l'énoncé d'un programme récursif est plus concis (et plus "logique"). Voici une implémentation récursive de cet algorithme :

```
def parcours(graphe, visites, s):
    """parcours en profondeur depuis le sommet s"""
    if s not in visites:
        visites[s] = True
        for voisin in graphe[s]:
            parcours(graphe, visites, voisin)
    return visites
```

Il suffit alors de lancer le premier appel avec un dictionnaire visites vide, ce que fait la fonction d'interface `parcours_prof_rec` suivante.

```
def parcours_prof_rec(graphe, debut):
    return parcours(graphe, {}, debut)
```

On peut vérifier que cela fonctionne tout autant.

```
>>> parcours_prof_rec(g1, "A")
{'A': True, 'B': True, 'C': True, 'E': True, 'D': True, 'F': True, 'G': True}
```

**Reamarque :** L'ordre des sommets visités n'est pas le même car ici c'est le premier voisin écrit dans la liste des sucesseurs (et pas encore visité) qui est exploré en premier. C'était le contraire avec la pile car les sommets étant empilés l'un après l'autre, celui en haut de la pile était le dernier écrit dans la liste de successeurs. On pourrait obtenir le même résultat si on empilait les voisins/successeurs dans l'ordre inverse.

Les algorithmes de parcours en profondeur ou en largeur permettent, selon les actions opérées sur les sommets découverts, à écrire de nouveaux algorithmes essentiels comme ceux permettant de :

- repérer la présence d'un cycle dans un graphe ;
- chercher un chemin dans un graphe

# VI - Repérer la présence d'un cycle dans un graphe

**Note :** Le terme cycle n'a de sens que dans un graphe non orienté, c'est pourquoi on ne considère dans ce paragraphe que des **graphes non orientés**.

## Principe de l'algorithme de détection de cycle

Il suffit d'adapter légèrement, au choix, l'un des deux algorithmes de parcours du graphe. Si lors du parcours on rencontre (en dépilant ou en défilant) un sommet déjà visité (marqué grâce au dictionnaire `visites`), on a trouvé un cycle ! En effet, cela signifie que ce sommet a été ajouté au moins deux fois dans la pile ou dans la file, ce qui veut dire que l'on peut l'atteindre par au moins deux sommets différents. Ces deux chemins ayant pour origine le sommet de départ du parcours, on a nécessairement un cycle.

L'algorithme est identique à celui d'un parcours en stoppant le parcours si un cycle est trouvé :

**Algorithme de détection de cycle**

- On choisit un sommet de départ
- On l'empile
- Tant que la pile n'est pas vide :
  - On depile son sommet
  - S'il n'a pas encore été visité on le marque et on empile tous ses voisins non encore visités
  - Sinon, **on a trouvé un cycle et on renvoie Vrai**

Si le parcours se termine sans trouver de cycle, on renvoie Faux.

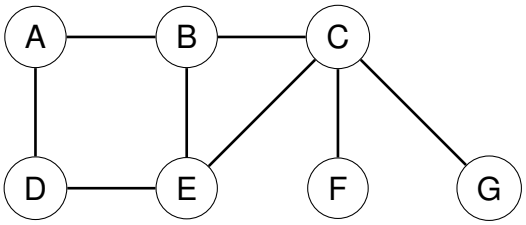
L'écriture d'une fonction repérant un cycle est donc quasiment identique à celle d'un parcours en profondeur (ou en largeur) : si on rencontre un sommet déjà visité, au lieu de passer au suivant (avec le mot-clé `continue`), il faut renvoyer `True`. Si à l'issue du parcours on n'a pas trouvé de cycle, on renvoie `False`.

La fonction `parcours_prof_cycle` qui suit renvoie `True` si et seulement si on trouve un cycle en partant du sommet `debut`.

```
def parcours_prof_cycle(graphe, debut):
    """Renvoie True ssi un cycle est detecte
    dans le parcours e partir du sommet debut."""
    visites = {}
    pile = [debut]
    while len(pile) > 0:
        s = pile.pop()
        if s in visites:
            return True  # on a remplace continue
        visites[s] = True
        for voisin in graphe[s]:
            if voisin not in visites:
                pile.append(voisin)
    return False  # on renvoie False et non plus le dictionnaire visites
```

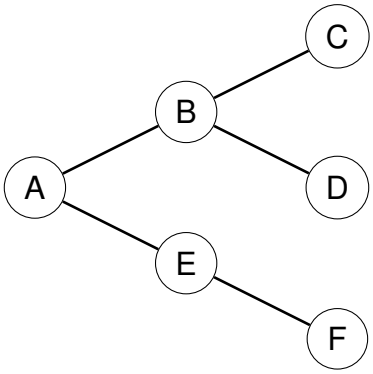
**Remarque :** Le principe serait le même en utilisant un parcours en largeur.

**Exemple :**
On peut vérifier qu'un cycle est bien détecté dans le graphe `g1` mais pas dans le graphe `g2`.




Graphe `g1`

```
>>> parcours_prof_cycle(g1, "A")
True
```




Graphe `g2`

```
>>> parcours_prof_cycle(g2, "A")
False
```

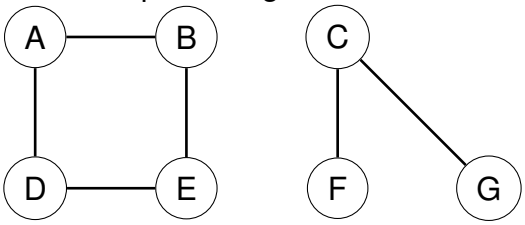
## Cas d'un graphe non connexe

### Définition 4

Un graphe $G$ est **connexe** si pour tout couple de sommets différents $x$ et $y$, il existe une chaîne entre $x$ et $y$.

**Exemple :**
Les graphes `g1` et `g2` sont connexes.
Le graphe `g3` suivant n'est pas connexe, il n'existe pas de cghin du sommet $A$ au sommet $F$

Graphe `g3`



Si le graphe (non orienté) est connexe, on peut tester la présence d'un cycle à partir de n'importe quel sommet de départ. En revanche, pour un graphe non connexe (et toujours non orienté), il faut s'assurer de parcourir tous ses sommets. On peut par exemple lancer la détection à partir de chaque sommet. La fonction suivante permet de faire ce travail.

```
def possede_cycle(graphe):
    # on lance le parcours a partir de chaque sommet x du graphe
    for x in graphe:
        # si on trouve un cycle a partir d'un sommet x la reponse est vrai
        if parcours_prof_cycle(graphe, x):
            return True
    return False  # sinon il n'y a pas de cycle
```

# VII - Recherche d'un chemin (ou d'une chaîne) dans un graphe

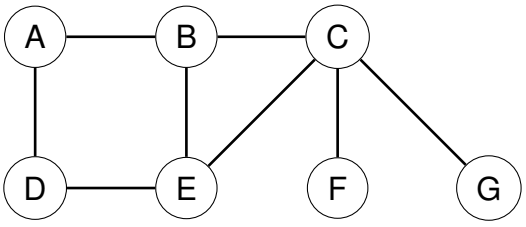
Rappelons quelques définitions :

- Dans un graphe non orienté, une **chaîne** est une séquence ordonnée d'arêtes telle que chaque arête a une extrémité en commun avec l'arête suivante.
- Dans un graphe orienté, un **chemin** désigne une séquence ordonnée d'arcs consécutifs.

**Note :** Dans la suite, on ne parlera que de chaînes ou de chemins simples, c'est-à-dire n'empruntant pas deux fois la même arête (ou le même arc).

En utilisant un parcours en profondeur ou en largeur, on peut trouver tous les sommets accessibles à partir d'un sommet de départ. Cela permet d'écrire très facilement un algorithme qui renvoie Vrai s'il existe un chemin pour aller d'un point A à un point B. Il suffit de lancer l'un des deux parcours à partir du sommet A et de regarder à la fin du parcours si le sommet B a été atteint (s'il est dans le dictionnaire `visites`).

Nous utiliserons dans cette section l'exemple du graphe `g1`.




Graphe `g1`

## Principe de l'algorithme de recherche d'un chemin

Une idée est d'utiliser différemment le dictionnaire `visites`. Celui-ci ne servira plus à marquer (à `True`) les sommets visités, mais associera à chaque sommet, le sommet qui permet de l'atteindre pour la première fois (le premier sommet duquel il est voisin dans le parcours).

Autrement dit, dès qu'on visite un sommet, il faut l'associer à tous ses voisins (non encore visités) dans le dictionnaire `visites`. On intialisera à `None` le sommet initial dans le dictionnaire `visites`. A la fin du parcours, il suffira de "remonter" le dictionnaire du sommet de fin au sommet de début.

Voici le principe plus en détail (en utilisant un parcours en profondeur) :

**Algorithme de recherche d'un chemin**

- On choisit le sommet de départ que l'on associe à `None`
- On l'empile
- Tant que la pile n'est pas vide :
  - On dépile son sommet `s`
  - (On ne le marque plus)
  - On empile tous ses voisins non encore visités et on les associe à la valeur `s` dans le dictionnaire `visites`.

La fonction `parcours_prof_ch` suivante permet de construire ce dictionnaire `visites`. Elle est basée sur un parcours en profondeur mais s'écrirait de la même manière avec un parcours en largeur (en remplaçant la pile par une file bien sûr).

```
def parcours_prof_ch(graphe, debut):
    visites = {debut: None}  # on associe le sommet de depart a None
    pile = [debut]
    while len(pile) > 0:
        s = pile.pop()
        # (on ne marque plus les sommets non visites)
        for voisin in graphe[s]:
            if voisin not in visites:
                pile.append(voisin)
                visites[voisin] = s  # on associe s a tous les voisins de s pas
                    encore visites
    return visites
```

On peut lancer le parcours sur le graphe `g1` à partir du sommet $A$.

```
>>> parcours_prof_ch(g1, "A")
{'A': None, 'B': 'A', 'D': 'A', 'E': 'D', 'C': 'E', 'F': 'C', 'G': 'C'}
```

Pour trouver le chemin entre le sommet $A$ et le sommet $G$, il faut "remonter" les sommets à partir de $G$ :

- On cherche $G$ : il est associé à la valeur $C$ donc on a pu atteindre $G$ à partir de $C$ ;
- On cherche $C$ : atteint à partir de $E$ ;
- On cherche $E$ : atteint à partir de $D$ ;
- On cherche $D$ : atteint à partir de $A$.

On a terminé puisqu'on a fini par tomber sur $A$. Un chemin possible entre $A$ et $G$ est donc :
$A \to D \to E \to C \to G$.

**Remarque :** On était sûr de remonter jusqu'à $A$ puisque $G$, se trouvant dans le dictionnaire visites, était nécessairement atteignable en partant de $A$. Si un sommet ne se trouve pas dans visites, on sait alors qu'il n'existe pas de chemin vers ce sommet en partant de $A$.

La fonction `chemin_prof(graphe, debut, fin)` permet d'effectuer ce travail en renvoyant une liste `ch` contenant les sommets du chemin trouvé entre les sommets `debut` et `fin`. Elle ajoute les sommets à `ch` au fur et à mesure de la remontée jusqu'à tomber sur celui de départ et renvoie ensuite cette liste qui a été préalablement renversée pour obtenir les sommets dans le bon ordre.

```
def chemin_prof(graphe, debut, fin):
    visites = parcours_prof_ch(graphe, debut)
    if fin not in visites:
        return None
    s = fin
    ch = [s]  # on ajoute le sommet de fin a partir duquel commence la "remontae"
    while s != debut:  # tant qu'on ne trouve pas le sommet de depart
        s = visites[s]  # on remonte en passant au sommet associe
        ch.append(s)  # qu'on ajoute au chemin
    ch.reverse()  # renverser la liste pour renvoyer le chemin dans le bon ordre
    return ch
```

On peut vérifier qu'elle renvoie le chemin trouvé à la main entre $A$ et $G$ pour le graphe `g1`.

```
d>>> chemin_prof(g1, "A", "G")
['A', 'D', 'E', 'C', 'G']
```

**Remarque :** On constate que le chemin n'est pas le plus court (en nombre d'arêtes) car on peut faire mieux : $A \to B \to C \to G$. Peut-on trouver le chemin le plus court ? La réponse est oui !

## Recherche d'un plus court chemin

En faisant la même recherche à partir d'un parcours en largeur, on obtiendrait un plus court chemin (en nombre d'arêtes/arcs).
En effet, l'algorithme de recherche en largeur explore d'abord les sommets à une distance 1 du sommet de départ, puis ceux à distance 2 du sommet de départ, etc. Ainsi, chacun des autres sommets est atteint en passant par un nombre minimal d'arêtes (ou arcs), ce qui assure de trouver un plus court chemin (en nombre d'arêtes/arcs) vers chacun des autres sommets.

```
def parcours_larg_ch(graphe, debut):    # on remplace la pile par une file
    visites = {debut: None} # on associe le sommet de depart a None
    file = [debut]
    while len(file) > 0:
        s = file.pop(0)
        # (on ne marque plus les sommets non visites)
        for voisin in graphe[s]:
            if voisin not in visites:
                file.append(voisin)
                visites[voisin] = s  # on associe s a tous les voisins de s pas
                   encore visites
    return visites

# exactement la meme fonction que chemin_prof (en remplacant juste l'appel a la
   premiere ligne)
def chemin_larg(graphe, debut, fin):
    visites = parcours_larg_ch(graphe, debut)
    if fin not in visites:
        return None
    s = fin
    ch = [s]
    while s != debut:
        s = visites[s]
        ch.append(s)
    ch.reverse()
    return ch
```

```
>>> chemin_larg(g1, "A", "G")
['A', 'B', 'C', 'G']
```

Nous venons de voir comment utiliser un parcours en largeur pour trouver le plus court chemin, en nombre d'arêtes/arcs, entre deux sommets d'un graphe.

## Distances entre les sommets

En modifiant le rôle du dictionnaire `visites` dans la recherche de chemin du parcours en largeur, on peut très facilement trouver la distance du sommet de départ à tous les autres. On va utiliser `visites` pour associer à chaque sommet la distance qui le sépare du sommet d'origine. La distance d'un sommet découvert est celle du sommet d'où on vient, plus 1 !

En initialisant une distance égale à 0 pour le sommet de départ on obtient, en changeant uniquement une ligne, les distances entre chaque sommet et le sommet de départ.

```
def parcours_larg_distance(graphe, debut):
    visites = {debut: 0} # debut est a distance 0 de lui-meme
    file = [debut]
    while len(file) > 0:
        s = file.pop(0)
        # (on ne marque plus les sommets non visites)
        for voisin in graphe[s]:
            if voisin not in visites:
                file.append(voisin)
                visites[voisin] = visites[s] + 1  # la distance est celle de s (d'
                   ou l'on vient) + 1
    return visites
```

On peut vérifier les distances entre le sommet A et les autres dans le graphe `g1`.

```
>> parcours_larg_distance(g1, "A")
{'A': 0, 'B': 1, 'D': 1, 'C': 2, 'E': 2, 'F': 3, 'G': 3}
```

Les notions de plus court chemin ou de distance que l'on vient de voir, correspondent au nombre d'arêtes/arcs séparant les sommets. On suppose donc que chaque arête à le même coût dans ce calcul. Autrement dit, nos algorithmes s'appliquent sur des graphes non pondérés (ou des graphes dans lesquels toutes les arêtes ont le même poids).

Avec des graphes pondérés, les arêtes n'ont pas toutes le même coût, ce qui redéfinit cette notion de distance. C'est le cas de la plupart des graphes rencontrés dans la vie courante. On ne peut donc plus appliquer l'algorithme de plus court chemin étudié. Il en existe heureusement d'autres : le plus connu d'entre eux est l'algorithme de Dijskstra.

## Algorithme de Dijskstra : pour trouver le plus court chemin pondéré

Voir cours sur les protocoles de routage.


Sources :

- Lycée Mounier - Angers