# Les graphes


## RÉSUMÉ

Un réseau social (qui est ami avec qui ?), une carte routière (quelles villes sont reliées ?), le plan d'un réseau informatique : tous ces systèmes peuvent être modélisés par un **graphe**, c'est-à-dire un ensemble de points (les **sommets**) reliés par des liens (les **arêtes**). Les graphes permettent de répondre à des questions concrètes : quel est le chemin le plus court entre deux villes ? Deux personnes sont-elles connectées ? Peut-on visiter toutes les villes sans repasser deux fois par la même route ? C'est l'un des outils les plus puissants de l'informatique.

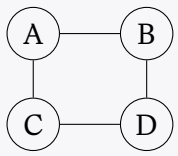

## ① MÉMO

### Vocabulaire

**Graphe** Ensemble de **sommets** (ou nœuds) reliés par des **arêtes** (non orienté) ou des **arcs** (orienté).
**Graphe orienté** Les liens ont un sens : un arc de $A$ vers $B$ ne signifie pas qu'il y a un arc de $B$ vers $A$.
**Graphe non orienté** Les liens sont symétriques : une arête entre $A$ et $B$ va dans les deux sens.
**Voisins (adjacents)** Sommets directement reliés à un sommet donné.
**Degré** Nombre de voisins d'un sommet.
**Chemin** Suite de sommets consécutivement reliés.
**Cycle** Chemin qui revient à son point de départ.
**Graphe connexe** Il existe un chemin entre toute paire de sommets.
**Graphe pondéré** Chaque arête porte un poids (distance, coût...).

### Représentations

**Matrice d'adjacence** Tableau $n \times n$ où $M[i][j] = 1$ s'il existe une arête de $i$ vers $j$, 0 sinon. Pour un graphe pondéré, on met le poids à la place de 1.
**Liste d'adjacence** Dictionnaire : chaque sommet est associé à la liste de ses voisins.




| | A | B | C | D |
|---|---|---|---|---|
| **A** | 0 | 1 | 1 | 0 |
| **B** | 1 | 0 | 0 | 1 |
| **C** | 1 | 0 | 0 | 1 |
| **D** | 0 | 1 | 1 | 0 |

```
graphe = {
  "A": ["B", "C"],
  "B": ["A", "D"],
  "C": ["A", "D"],
  "D": ["B", "C"]
}
```

### Parcours en profondeur (DFS)

On explore le plus loin possible avant de revenir en arrière. Utilise une **pile** (ou la récursion).

```python
def dfs(graphe, depart):
    visites = set()
    pile = [depart]
    while pile:
        sommet = pile.pop()
        if sommet not in visites:
            visites.add(sommet)
            for voisin in graphe[sommet]:
                pile.append(voisin)
    return visites
```

**Complexité :** $O(S + A)$ avec une liste d'adjacence ($S$ = nombre de sommets, $A$ = nombre d'arêtes), $O(S^2)$ avec une matrice d'adjacence.

### Parcours en largeur (BFS)

On explore tous les voisins d'abord, puis les voisins des voisins. Utilise une **file**.

```python
from collections import deque

def bfs(graphe, depart):
    visites = set()
    file = deque([depart])
    visites.add(depart)
    while file:
        sommet = file.popleft()
        for voisin in graphe[sommet]:
            if voisin not in visites:
                visites.add(voisin)
                file.append(voisin)
    return visites
```

Le BFS donne le **plus court chemin** (en nombre d'arêtes) dans un graphe non pondéré.
**Complexité :** $O(S + A)$ avec une liste d'adjacence, comme pour le DFS.

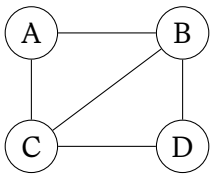
### Erreurs classiques

| Code erroné | Code correct | Explication |
|---|---|---|
| DFS ou BFS sans ensemble `visites` | `visites = set()` et vérifier avant chaque visite | Sans marquer les sommets visités, un cycle provoque une boucle infinie : le parcours repasse indéfiniment par les mêmes sommets. |
| BFS : ajouter le départ dans la file sans le marquer visité | `visites.add(depart)` dès l'initialisation | Si le sommet de départ n'est pas marqué visité immédiatement, il peut être enfilé plusieurs fois par ses propres voisins. |
| Graphe orienté traité comme non orienté | Vérifier si les arêtes sont symétriques | Dans un graphe orienté, un arc de $A$ vers $B$ n'implique pas un arc de $B$ vers $A$ : la liste d'adjacence de $B$ ne contient pas forcément $A$. |
| Utiliser le BFS pour le plus court chemin dans un graphe pondéré | Utiliser l'algorithme de Dijkstra | Le BFS donne le chemin avec le moins d'arêtes, pas celui de poids minimal. Pour un graphe pondéré, il faut un algorithme dédié. |

## ② EXEMPLES

### Programme 4 · Création d'un graphe (liste d'adjacence)

```
graphe = {
    "A": ["B", "C"],
    "B": ["A", "C", "D"],
    "C": ["A", "B", "D"],
    "D": ["B", "C"]
}
```




### Programme 5 · DFS récursif

```
def dfs_rec(graphe, sommet, visites=None):
    if visites is None:
        visites = set()
    visites.add(sommet)
    print(sommet, end=" ")
    for voisin in graphe[sommet]:
        if voisin not in visites:
            dfs_rec(graphe, voisin, visites)
    return visites

# dfs_rec(graphe, "A") donne A B C D (ou autre ordre selon les voisins)
```

### Programme 6 · BFS avec reconstruction du chemin

```
def bfs_chemin(graphe, depart, arrivee):
    file = deque([depart])
    visites = {depart}
    parent = {depart: None}
    while file:
        sommet = file.popleft()
        if sommet == arrivee:
            # Reconstruction du chemin
            chemin = []
            while sommet is not None:
                chemin.append(sommet)
                sommet = parent[sommet]
            return chemin[::-1]
        for voisin in graphe[sommet]:
            if voisin not in visites:
                visites.add(voisin)
                parent[voisin] = sommet
                file.append(voisin)
    return None   # pas de chemin
```

### Programme 7 · Détection de cycle (graphe non orienté)

```
def a_cycle(graphe):
    visites = set()
    for sommet in graphe:
        if sommet not in visites:
            if _dfs_cycle(graphe, sommet, None, visites):
                return True
    return False

def _dfs_cycle(graphe, sommet, parent, visites):
    visites.add(sommet)
    for voisin in graphe[sommet]:
        if voisin not in visites:
            if _dfs_cycle(graphe, voisin, sommet, visites):
                return True
        elif voisin != parent:
            return True
    return False
```

### Programme 8 · Composantes connexes

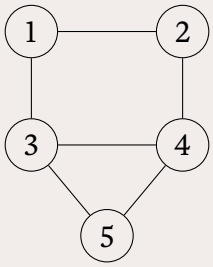
```
def composantes_connexes(graphe):
    visites = set()
    composantes = []
    for sommet in graphe:
        if sommet not in visites:
            composante = set()
            dfs_composante(graphe, sommet, visites, composante)
            composantes.append(composante)
    return composantes

def dfs_composante(graphe, sommet, visites, composante):
    visites.add(sommet)
```

```
    composante.add(sommet)
    for voisin in graphe[sommet]:
        if voisin not in visites:
            dfs_composante(graphe, voisin, visites, composante)
```

## ③ EXERCICES

**Exercice 1** ***Représentations*** Soit le graphe non orienté suivant :




1. Donner la matrice d'adjacence.
2. Donner la liste d'adjacence (dictionnaire).
3. Donner le degré de chaque sommet.
4. Ce graphe est-il connexe ? Comporte-t-il un cycle ?

**Exercice 2** ***Parcours*** En utilisant le graphe de l'exercice précédent :

1. Donner l'ordre de visite du DFS en partant du sommet 1 (en choisissant les voisins dans l'ordre croissant).
2. Donner l'ordre de visite du BFS en partant du sommet 1.
3. Donner le plus court chemin de 2 à 5 (en nombre d'arêtes). Justifier avec le BFS.

**Exercice 3** ***Algorithme de plus court chemin*** Écrire une fonction `distance(graphe, depart, arrivee)` qui renvoie la longueur du plus court chemin (en nombre d'arêtes) entre deux sommets d'un graphe non pondéré.

Renvoyer $-1$ si aucun chemin n'existe.

## SOLUTIONS DES EXERCICES

**Corrigé de l'exercice 1.** **1. Matrice d'adjacence :**

```
    1  2  3  4  5
1 [ 0, 1, 1, 0, 0 ]
2 [ 1, 0, 0, 1, 0 ]
3 [ 1, 0, 0, 1, 1 ]
4 [ 0, 1, 1, 0, 1 ]
5 [ 0, 0, 1, 1, 0 ]
```

Matrice symétrique (graphe non orienté).

**2. Liste d'adjacence :**

```
{1: [2, 3], 2: [1, 4], 3: [1, 4, 5], 4: [2, 3, 5], 5: [3, 4]}
```

**3. Degrés :** sommet 1 : 2 ; sommet 2 : 2 ; sommet 3 : 3 ; sommet 4 : 3 ; sommet 5 : 2.

On vérifie : la somme des degrés vaut $2 + 2 + 3 + 3 + 2 = 12 = 2 \times 6$ (deux fois le nombre d'arêtes). ✓

**4.** Le graphe est connexe (on peut atteindre tout sommet depuis tout autre sommet). Il comporte des cycles, par exemple $1 - 2 - 4 - 3 - 1$ ou $3 - 4 - 5 - 3$.

**Corrigé de l'exercice 2.** **1. DFS depuis 1 (voisins en ordre croissant) :**

- Visite 1, empile [2, 3]. Dépile 3.
- Visite 3, empile [1, 4, 5]. 1 déjà visité. Dépile 5.
- Visite 5, empile [3, 4]. 3 déjà visité. Dépile 4.
- Visite 4, empile [2, 3, 5]. 3 et 5 déjà visités. Dépile 2.
- Visite 2. Tous ses voisins sont visités.

Ordre : 1, 3, 5, 4, 2.

**Remarque :** avec la version pile (itérative), le dernier voisin ajouté est visité en premier, ce qui explique que 3 est visité avant 2.

Avec la version récursive (voisins en ordre croissant), on obtiendrait 1, 2, 4, 3, 5.

**2. BFS depuis 1 :**

- File : [1]. Visite 1, enfile [2, 3].
- File : [2, 3]. Visite 2, enfile [4]. File : [3, 4].
- Visite 3, enfile [5] (4 déjà visité par 2). File : [4, 5].
- Visite 4. Tous ses voisins sont visités. File : [5].
- Visite 5. File vide.

Ordre : 1, 2, 3, 4, 5.

**3. Plus court chemin de 2 à 5 :**

Le BFS depuis 2 visite les sommets par distance croissante :

- distance 0 : {2} ;
- distance 1 : {1, 4} ;
- distance 2 : {3, 5}.

Le plus court chemin est de longueur 2. Par exemple : $2 - 4 - 5$.

**Corrigé de l'exercice 3.**

```python
from collections import deque

def distance(graphe, depart, arrivee):
    if depart == arrivee:
        return 0
    visites = {depart}
    file = deque([(depart, 0)])
    while file:
        sommet, dist = file.popleft()
        for voisin in graphe[sommet]:
            if voisin == arrivee:
                return dist + 1
            if voisin not in visites:
                visites.add(voisin)
                file.append((voisin, dist + 1))
    return -1
```

**Principe :** c'est un BFS classique où l'on stocke la distance dans la file. Le premier chemin trouvé vers `arrivee` est garanti le plus court (propriété fondamentale du BFS).

**Complexité :** $O(S + A)$ où $S$ est le nombre de sommets et $A$ le nombre d'arêtes.