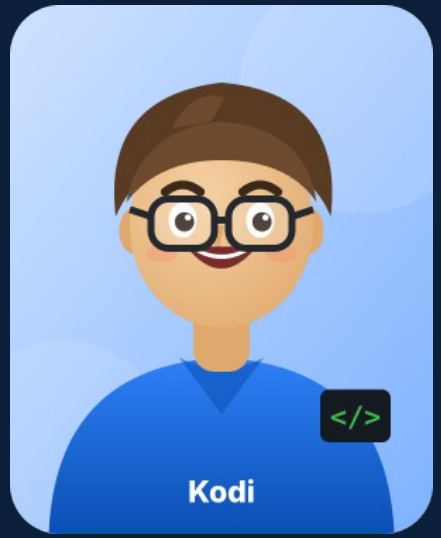
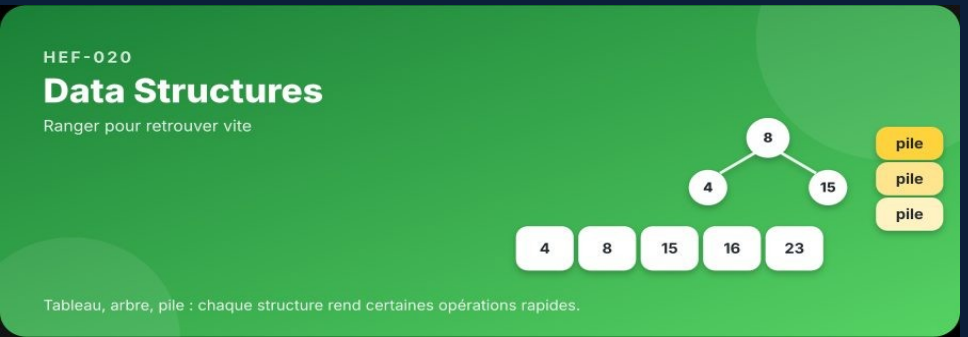
HEF-020 · MISSION 09 / 10

Explorer un graphe : DFS et BFS

Pour trouver un chemin, vérifier une connexion ou explorer un réseau, il faut un plan de parcours.



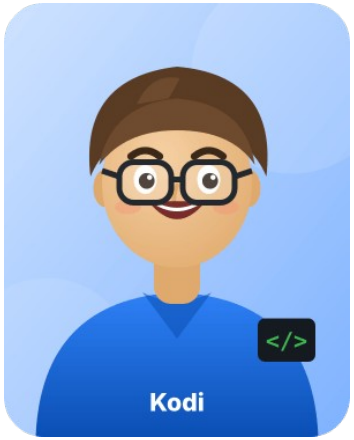


Badge : Explorateur·rice

80 min · débutant

L'histoire

Dans le réseau de mentorat, Maya demande : « **Combien de relations séparent Ada de Dan ?** » Et aussi : « **Tous les membres sont-ils connectés entre eux ?** »



> *« Il te faut une méthode pour visiter les sommets sans tourner en rond », dit Sam.*

Tu apprends les deux parcours fondamentaux, qui servent partout (cartes, jeux, internet, compilation).

Ta mission

À la fin, je serai capable de…

1. expliquer le parcours en largeur (BFS) et en profondeur (DFS)
2. implémenter BFS avec une file
3. implémenter DFS avec une pile ou la récursion
4. trouver le plus court chemin dans un graphe non pondéré
5. détecter les composantes connexes

Défi d'échauffement

Tu cherches ton chat dans un grand immeuble. Deux stratégies : (A) tu vérifies **tous les appartements de l'étage** avant de monter à l'étage suivant ; (B) tu **entres dans le premier appartement** et tu explores toutes ses pièces avant d'aller voir le suivant. Laquelle trouve plus vite un chat qui est proche de l'entrée ? Laquelle explore plus loin d'un coup ?

Le concept

Le problème du parcours

Visiter tous les sommets accessibles depuis un sommet de départ, **sans visiter deux fois le même** (sinon on boucle dans les cycles). On garde un ensemble des sommets **déjà vus**.

BFS — parcours en largeur

On visite d'abord tous les voisins, puis les voisins des voisins… par **cercles concentriques**. On utilise une **file**.

```
from collections import deque

def bfs(graphe, depart):
    vus = {depart}
    file = deque([depart])
    ordre = []
```

Le concept

BFS — parcours en largeur (suite)

```
    while file:
        s = file.popleft()
        ordre.append(s)
        for v in graphe[s]:
            if v not in vus:
                vus.add(v)
                file.append(v)
    return ordre
```

Propriété magique : dans un graphe non pondéré, BFS trouve les **plus courts chemins** (en nombre d'arêtes).

DFS — parcours en profondeur

On s'enfonce le plus loin possible, puis on revient. On utilise une **pile** (ou la récursion, qui utilise la pile d'appels).

Le concept

DFS — parcours en profondeur (suite)

```
def dfs(graphe, depart, vus=None):
    if vus is None:
        vus = set()
    vus.add(depart)
    ordre = [depart]
    for v in graphe[depart]:
        if v not in vus:
            ordre += dfs(graphe, v, vus)
    return ordre
```

Quand utiliser quoi ?

Besoin	Parcours
Plus court chemin (non pondéré)	BFS
Niveaux, distance	BFS

Le concept

Quand utiliser quoi ? (suite)

Besoin	Parcours
Détecter des cycles, composantes, ordre topologique	DFS
Explorer toutes les possibilités (labyrinthe, puzzle)	DFS

Composantes connexes

Une **composante connexe** est un groupe de sommets reliés entre eux. Pour les trouver : lancer un parcours depuis chaque sommet non encore vu.

Exemple pas à pas

Problème : distance (nombre de relations) entre deux membres.

```
from collections import deque

def distance(graphe, a, b):
    if a == b:
        return 0
    vus = {a}
    file = deque([(a, 0)])
    while file:
        s, d = file.popleft()
        for v in graphe[s]:
            if v == b:
                return d + 1
            if v not in vus:
                vus.add(v)
                file.append((v, d + 1))
    return None        # pas de chemin
```

Exemple pas à pas

Exemple pas à pas (suite)

```
g = {
    "Ada": ["Bob", "Cléo"], "Bob": ["Ada", "Eve"],
    "Cléo": ["Ada", "Dan"], "Dan": ["Cléo"],
    "Eve": ["Bob"], "Zoé": [],
}
print(distance(g, "Ada", "Dan"))    # 2
print(distance(g, "Ada", "Eve"))    # 2
print(distance(g, "Ada", "Zoé"))    # None
```

Pourquoi BFS fonctionne : les sommets sont découverts par ordre croissant de distance. La première fois qu'on atteint `b`, c'est par le chemin le plus court.

À toi, avec aide

À toi, avec aide

Écris `composantes(graphe)` qui renvoie la liste des groupes de sommets connectés. Teste sur :

```
{"a": ["b"], "b": ["a"], "c": [], "d": ["e"], "e": ["d"]}
```

Le résultat doit contenir 3 groupes : `{a, b}`, `{c}`, `{d, e}`.

Indices progressifs

Indice 1

Garde un ensemble vus global.

Indice 2

Pour chaque sommet non vu, lance un parcours (BFS ou DFS) qui collecte les sommets atteints.

Indice 3

Ajoute chaque groupe à la liste de résultats.

À toi, avec aide · réponse

Réponse

```
from collections import deque

def composantes(graphe):
    vus, groupes = set(), []
    for depart in graphe:
        if depart in vus:
            continue
        groupe = set()
        file = deque([depart])
        vus.add(depart)
        while file:
            s = file.popleft()
            groupe.add(s)
            for v in graphe[s]:
                if v not in vus:
                    vus.add(v)
                    file.append(v)
        groupes.append(groupe)
    return groupes
```

Réponse (suite)

```
g = {"a": ["b"], "b": ["a"], "c": [], "d": ["e"], "e": ["d"]}
res = composantes(g)
assert sorted(map(sorted, res)) == [["a", "b"], ["c"], ["d", "e"]]
print(res)
```

À toi, en solo

Labyrinthe. Une grille est une liste de textes (`#` = mur, `.` = libre, `S` = départ, `E` = arrivée). Écris `plus_court_chemin(grille)` qui renvoie le nombre de pas minimal de S à E (ou `None`).

```
["S.#",
 ".##",
 "..E"]
```

Résultat attendu : 4.

Indice : les sommets sont les cases libres ; les voisins sont haut, bas, gauche, droite.

À toi, en solo · réponse

Pistes de correction

```
from collections import deque

def plus_court_chemin(grille):
    h, l = len(grille), len(grille[0])
    for i in range(h):
        for j in range(l):
            if grille[i][j] == "S":
                depart = (i, j)
    file = deque([(depart, 0)])
    vus = {depart}
    while file:
        (i, j), d = file.popleft()
        if grille[i][j] == "E":
            return d
        for di, dj in ((1, 0), (-1, 0), (0, 1), (0, -1)):
            ni, nj = i + di, j + dj
            if 0 <= ni < h and 0 <= nj < l and grille[ni][nj] != "#" and (ni, nj)
   not in vus:
                vus.add((ni, nj))
```

Pistes de correction (suite)

```
                file.append(((ni, nj), d + 1))
    return None

print(plus_court_chemin(["S.#", ".##", "..E"]))   # 4
print(plus_court_chemin(["S#E"]))                 # None
```

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Oublier l'ensemble des sommets vus	Boucle infinie dans les cycles	Marquer chaque sommet visité
Marquer trop tard dans BFS	Un sommet est mis plusieurs fois dans la file	Marquer au moment de l'ajout
Récursion profonde (DFS)	`RecursionError` sur un très grand graphe	Utiliser une pile explicite
BFS sur un graphe pondéré	Le plus court en arêtes n'est pas le moins coûteux	Utiliser un algorithme dédié (Dijkstra)

Mini-boss

Mini-boss

Ordre des cours. À partir du graphe des prérequis, écris `ordre_topologique(graphe)` qui renvoie une liste de cours telle que chaque cours apparaît **après** ses prérequis, ou lève une erreur s'il y a un **cycle**.
Teste sur le graphe de la mission 08 (HEF-000 → HEF-010 → HEF-020 → HEF-030), puis ajoute une arête HEF-030 → HEF-000 : l'erreur doit apparaître.

Mini-boss · réponse

Pistes de correction

```
def ordre_topologique(graphe):
    etat = {}          # 1 = en cours, 2 = terminé
    resultat = []

    def visiter(s):
        if etat.get(s) == 2:
            return
        if etat.get(s) == 1:
            raise ValueError("cycle détecté")
        etat[s] = 1
        for v in graphe[s]:
            visiter(v)
        etat[s] = 2
        resultat.append(s)

    for s in graphe:
        visiter(s)
    return resultat[::-1]
```

Mini-boss · réponse

Pistes de correction (suite)

```
g = {"HEF-000": ["HEF-010"], "HEF-010": ["HEF-020", "HEF-030"], "HEF-020": ["HEF-0
   30"], "HEF-030": []}
print(ordre_topologique(g))   # ['HEF-000', 'HEF-010', 'HEF-020', 'HEF-030']
g["HEF-030"] = ["HEF-000"]
try:
    ordre_topologique(g)
except ValueError as e:
    print(e)                  # cycle détecté
```

Quiz éclair

Quiz éclair

1. BFS utilise :

A. une pile uniquement
B. une file
C. un tas

2. BFS trouve dans un graphe non pondéré :

A. le chemin le plus long
B. un chemin au hasard
C. le plus court chemin en nombre d'arêtes

3. Pour ne pas boucler dans un cycle, on garde :

A. un ensemble des sommets déjà vus

Quiz éclair (suite)

B. une copie du graphe

C. rien

4. 🔁 *Révision mission 08* : Pour un graphe avec peu d'arêtes, la meilleure représentation est :

A. la matrice

B. la liste d'adjacence

C. aucune

5. 🔁 *Révision mission 07* : Une file de priorité retire toujours :

A. le premier arrivé

B. le dernier arrivé

C. l'élément de plus haute priorité

Quiz éclair · réponse

Réponses

1. **B**.
2. **C**.
3. **A**.
4. **B**.
5. **C**.

Je réfléchis

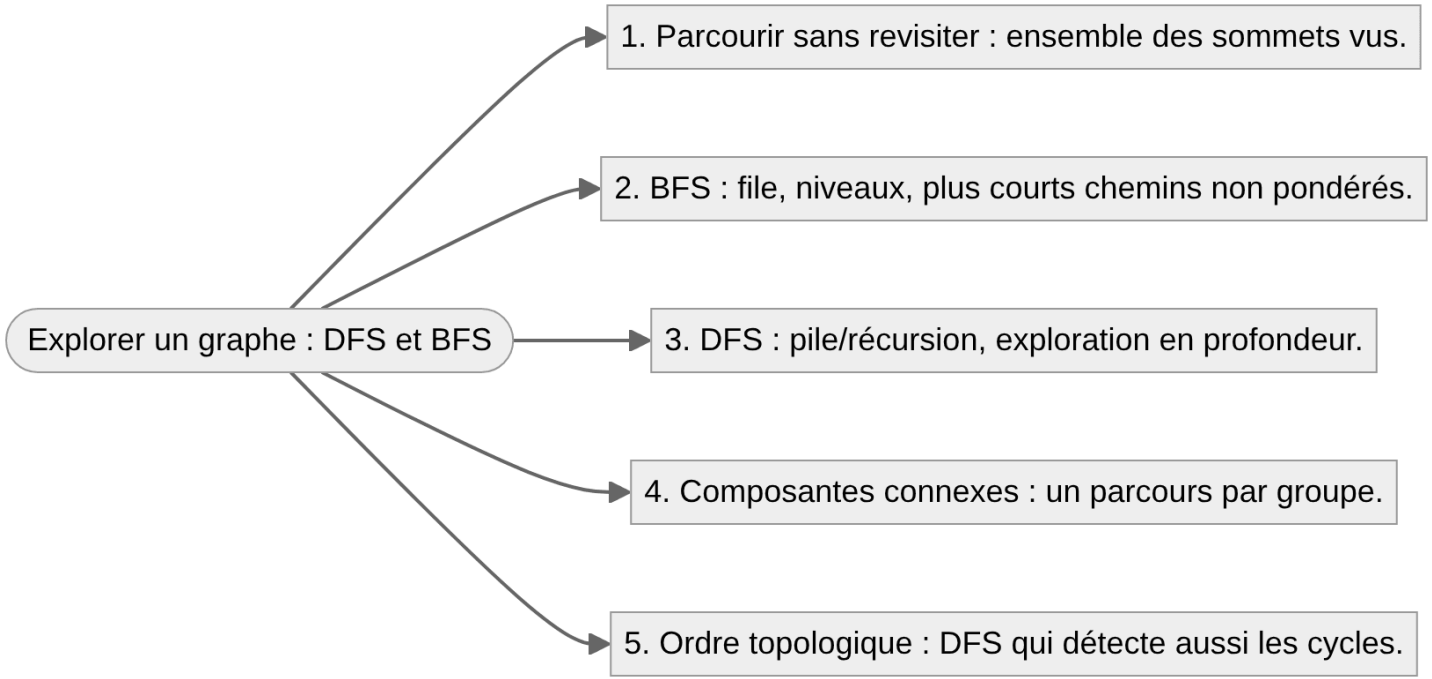
1. Dans ta recherche du chat, quelle stratégie as-tu choisie spontanément ?
2. Dans quels problèmes de ta vie la distance « en nombre d'étapes » compte-t-elle ?

À retenir

1. Parcourir sans revisiter : ensemble des sommets vus.
2. BFS : file, niveaux, plus courts chemins non pondérés.
3. DFS : pile/récursion, exploration en profondeur.
4. Composantes connexes : un parcours par groupe.
5. Ordre topologique : DFS qui détecte aussi les cycles.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Explorateur·rice

Prochaine mission : Choisir, prouver, comparer

