Chapitre 9
Graphes

Les graphes permettent de modéliser des situations où des objets sont en relation : un réseau social (les utilisateurs et leurs amitiés), un réseau routier (les villes et les routes), Internet (les machines et les liaisons), un labyrinthe (les cases et les passages). Ce chapitre montre comment **modéliser** de telles situations par des graphes, comment les **implémenter** en Python (matrice d'adjacence, listes de successeurs et de prédécesseurs), comment les **parcourir** (en profondeur, en largeur), comment **détecter un cycle** et comment **chercher un chemin**.

I. Modéliser des situations par des graphes

1. Des relations entre objets

Dans de nombreuses situations, on étudie des objets (des personnes, des villes, des ordinateurs, des cases d'un labyrinthe) et les relations qui les lient (une amitié, une route, une liaison, un passage). Le **graphe** est l'outil mathématique et informatique qui permet de modéliser de telles situations.

Définition 9.1.
Un **graphe** est constitué d'un ensemble de **sommets** (les objets étudiés) et de **liens** entre certains de ces sommets. Lorsque les liens sont non orientés (on peut les parcourir dans les deux sens), on parle d'**arêtes** ; lorsqu'ils sont orientés (on ne peut les parcourir que dans un sens), on parle d'**arcs**.

■ **Exemple 9.1.**
Dans un réseau social, chaque utilisateur est un sommet et chaque relation d'amitié est une arête : si A est ami avec B, alors B est ami avec A. Le graphe des amitiés est donc non orienté. ■

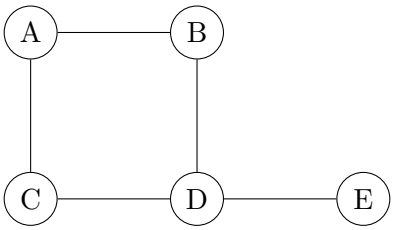
■ **Exemple 9.2.**
Dans un réseau routier, chaque ville est un sommet et chaque route entre deux villes est une arête : on peut circuler dans les deux sens. Le graphe est non orienté. ■

■ **Exemple 9.3.**
Si certaines rues sont à sens unique, la relation « être relié » n'est plus symétrique : on utilise alors des arcs, qui ont un sens. Le graphe est orienté. On peut le voir

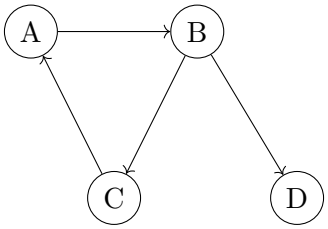
comme un réseau de flèches : l'arc qui va de A vers B ne permet pas de revenir de B vers A. ■

Définition 9.2.
Un graphe dont tous les liens sont des arêtes est dit **non orienté** ; un graphe dont tous les liens sont des arcs est dit **orienté**. On parle aussi de **graphe orienté** pour insister sur le sens des liens.

■ **Exemple 9.4.**
Internet est un graphe : les routeurs et les machines sont les sommets, et le routage des paquets distingue le sens des liaisons : on le modélise souvent par un graphe orienté. Le réseau électrique, lui, est plutôt modélisé par un graphe non orienté : les postes électriques sont les sommets, les lignes les arêtes. ■



Graphe non orienté G : les sommets A, B, C, D, E et les arêtes AB, AC, BD, CD, DE.



Graphe orienté H : les arcs AB, BC, CA et BD.

R Le graphe G ci-dessus (à cinq sommets) servira de fil rouge tout au long du chapitre : on écrira sa matrice d'adjacence, on le parcourra, on cherchera des chemins dedans. Le graphe H, orienté, illustrera la détection d'un cycle.

2. Vocabulaire des graphes

Pour décrire les algorithmes sur les graphes, il faut un vocabulaire précis.

Définition 9.3.
Deux sommets reliés par une arête (ou par un arc) sont dits **adjacents** ; chacun est un **voisin** de l'autre. Le **degré** d'un sommet d'un graphe non orienté est son nombre de voisins. Pour un graphe orienté, on distingue le **degré entrant** (nombre d'arcs qui arrivent sur le sommet) et le **degré sortant** (nombre d'arcs qui partent du sommet).

■ **Exemple 9.5.**
Dans le graphe G, les voisins de A sont B et C : le degré de A vaut 2. Les voisins de D sont B, C et E : le degré de D vaut 3. Dans le graphe orienté H, le degré entrant de C vaut 1 (l'arc BC) et son degré sortant vaut 1 (l'arc CA). ■

Définition 9.4.
Un **chemin** d'un sommet A vers un sommet B est une suite de sommets qui commence à A, finit à B, et dont chaque sommet est relié au suivant. Un **cycle** est un chemin dont le premier et le dernier sommet sont identiques, tous les autres sommets étant deux à deux distincts. Dans un graphe non orienté simple (sans boucle ni arête multiple), un cycle comporte au moins trois sommets.

■ **Exemple 9.6.**
Dans le graphe G, la suite A, B, D, E est un chemin de A vers E. Le graphe G contient le cycle A, B, D, C, A : en suivant les arêtes AB, BD, DC et CA, on revient au point de départ. Le graphe orienté H contient lui aussi un cycle, A, B, C, A : en suivant le sens des arcs, on revient au point de départ. ■

Définition 9.5.
Un graphe non orienté est **connexe** si, pour toute paire de sommets, il existe un chemin reliant ces deux sommets.

■ **Exemple 9.7.**
Le graphe G est connexe : il existe un chemin entre deux sommets quelconques. En revanche, un graphe formé de deux composantes séparées (aucune arête entre elles) n'est pas connexe. ■

R Pour un graphe orienté, la notion de connexité est plus subtile : on dit qu'un sommet B est **accessible** depuis un sommet A s'il existe un chemin de A vers B, en suivant le sens des arcs. Les parcours étudiés dans ce chapitre permettent précisément de déterminer l'ensemble des sommets accessibles depuis un sommet donné.

II. Implémenter un graphe

1. Le choix de la représentation

Un même graphe peut être représenté en machine de plusieurs façons. Le programme demande de savoir écrire les implémentations correspondantes d'un graphe — **matrice d'adjacence** et **listes de successeurs et de prédécesseurs** — et de passer d'une représentation à une autre. Le choix de la représentation dépend du traitement que l'on veut mettre en place.

R Deux grandes familles de représentations :
- la **matrice d'adjacence** : un tableau carré qui indique, pour chaque couple de sommets, s'il existe un lien ;
- les **listes d'adjacence** : pour chaque sommet, la liste de ses voisins. Dans un graphe orienté, on parle de **successeurs** pour les sommets vers lesquels partent des arcs, et de **prédécesseurs** pour les sommets d'où viennent des arcs.

Dans un graphe non orienté, successeurs et prédécesseurs sont confondus : ce sont simplement les voisins.

2. Une implémentation objet

L'exemple des graphes permet d'illustrer l'utilisation des classes en programmation. On modélise un graphe avec trois classes : `Noeud` (un sommet), `Arete` (un lien entre deux sommets) et `Graphe` (l'ensemble des sommets et des arêtes).

▪ Exemple 9.8.
On définit la classe `Noeud` : chaque sommet possède un nom. ▪

```
class Noeud:
    def __init__(self, nom):
        self.nom = nom
```

▪ Exemple 9.9.
On définit la classe `Arete` : chaque arête relie deux nœuds, que l'on note `noeud_1` et `noeud_2`. ▪

```
class Arete:
    def __init__(self, noeud_1, noeud_2):
        self.noeud_1 = noeud_1
        self.noeud_2 = noeud_2
```

▪ Exemple 9.10.
On définit la classe `Graphe` : un graphe possède une liste de nœuds et une liste d'arêtes. ▪

```
class Graphe:
    def __init__(self, liste_noeuds, liste_aretes):
        self.liste_noeuds = liste_noeuds
        self.liste_aretes = liste_aretes
```

R On utilise le mot `Noeud` pour désigner un sommet : c'est le vocabulaire de la modélisation choisie. Les sommets du graphe sont des **objets** de la classe `Noeud`, pas de simples chaînes de caractères.

■ **Exemple 9.11.**
On construit deux graphes : le premier avec les nœuds 1, 2, 3 et les arêtes (1, 2) et (1, 3) ; le second avec les nœuds A, B, C et les arêtes (A, B) et (B, C). La fonction `make_graphes` permet de recréer ces deux graphes à la demande, sans avoir à remonter dans le fichier pour relancer les cellules précédentes. ■

```
def make_graphes():
    noeud_1 = Noeud(1)
    noeud_2 = Noeud(2)
    noeud_3 = Noeud(3)
    noeud_a = Noeud("A")
    noeud_b = Noeud("B")
    noeud_c = Noeud("C")

    premier_graphe = Graphe(
        [noeud_1, noeud_2, noeud_3],
        [Arete(noeud_1, noeud_2), Arete(noeud_1, noeud_3)]
    )

    deuxieme_graphe = Graphe(
        [noeud_a, noeud_b, noeud_c],
        [Arete(noeud_a, noeud_b), Arete(noeud_b, noeud_c)]
    )
    return premier_graphe, deuxieme_graphe
```

3. La matrice d'adjacence

Définition 9.6.

La **matrice d'adjacence** d'un graphe à n sommets est un tableau carré à n lignes et n colonnes : la case située à la ligne i et à la colonne j vaut 1 si les sommets i et j sont reliés, et 0 sinon.

■ **Exemple 9.12.**

Matrice d'adjacence du graphe G (sommets A, B, C, D, E dans cet ordre). Le graphe est non orienté : chaque arête apparaît deux fois dans la matrice (case (i, j) et case (j, i)), la matrice est **symétrique** par rapport à sa diagonale. ■

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

■ **Exemple 9.13.**

On implémente la matrice d'adjacence par un **dictionnaire de dictionnaires** : à chaque nœud (clé du dictionnaire externe) est associé un dictionnaire qui donne, pour chaque nœud, la valeur 0 ou 1. La méthode `make_matrice_adjacence` construit cette matrice à partir de la liste des arêtes. ■

```
class Graphe:
    def __init__(self, liste_noeuds, liste_aretes):
        self.liste_noeuds = liste_noeuds
        self.liste_aretes = liste_aretes
        self.matrice_adjacence = {}

    def make_matrice_adjacence(self):
        self.matrice_adjacence = {
            noeud_ligne.nom: {noeud_colonne.nom: 0 for noeud_colonne in
                self.liste_noeuds}
            for noeud_ligne in self.liste_noeuds
        }
        liste_arete_temp = [[arete.noeud_1, arete.noeud_2] for arete in
            self.liste_aretes]
        for noeud_ligne in self.liste_noeuds:
            for noeud_colonne in self.liste_noeuds:
                if noeud_ligne == noeud_colonne:
                    self.matrice_adjacence[noeud_ligne.nom][noeud_colonne.
                        nom] = 0
                elif ([noeud_ligne, noeud_colonne] in liste_arete_temp
                      or [noeud_colonne, noeud_ligne] in liste_arete_temp):
                    self.matrice_adjacence[noeud_ligne.nom][noeud_colonne.
                        nom] = 1
                else:
                    self.matrice_adjacence[noeud_ligne.nom][noeud_colonne.
                        nom] = 0
```

R Pour un graphe **non orienté**, l'arête entre deux nœuds est notée 1 dans les deux cases correspondantes : on teste donc les deux ordres possibles, `[noeud_ligne, noeud_colonne]` et `[noeud_colonne, noeud_ligne]`, dans la liste des arêtes. La diagonale reste nulle tant que le graphe n'a pas de boucle (arête reliant un sommet à lui-même).

4. Graphe orienté et attribut `oriente`

■ **Exemple 9.14.**

Pour un graphe **orienté**, seul l'arc qui va de l'origine vers la destination est noté 1 : la matrice n'est plus symétrique. On ajoute à la classe `Graphe` un attribut booléen `oriente` : la même méthode `make_matrice_adjacence` s'adapte aux deux cas. ■

```
class Graphe:
    def __init__(self, liste_noeuds, liste_aretes):
        self.liste_noeuds = liste_noeuds
        self.liste_aretes = liste_aretes
        self.matrice_adjacence = {}
        self.oriente = True

    def make_matrice_adjacence(self):
        self.matrice_adjacence = {
            noeud_origine.nom: {noeud_destination.nom: 0
                                for noeud_destination in self.liste_noeuds}
            for noeud_origine in self.liste_noeuds
        }
        liste_arete_temp = [[arete.noeud_1, arete.noeud_2] for arete in
            self.liste_aretes]
        for noeud_origine in self.liste_noeuds:
            for noeud_destination in self.liste_noeuds:
                if noeud_origine == noeud_destination:
                    self.matrice_adjacence[noeud_origine.nom][
                        noeud_destination.nom] = 0
                elif [noeud_origine, noeud_destination] in liste_arete_temp
                    :
                    self.matrice_adjacence[noeud_origine.nom][
                        noeud_destination.nom] = 1
                elif not self.oriente and [noeud_destination, noeud_origine
                    ] in liste_arete_temp:
                    self.matrice_adjacence[noeud_origine.nom][
                        noeud_destination.nom] = 1
                else:
                    self.matrice_adjacence[noeud_origine.nom][
                        noeud_destination.nom] = 0
```

R L'attribut `oriente` vaut `True` par défaut. La case (destination, origine) n'est remplie que si le graphe est non orienté : c'est le rôle du test `not self.oriente`. La même méthode sert donc pour les deux types de graphes.

5. La liste des voisins

■ **Exemple 9.15.**

À partir de la matrice d'adjacence, on peut calculer pour chaque nœud la liste de ses **voisins** : la méthode `make_voisins` parcourt la ligne de la matrice correspondant à chaque nœud et retient les nœuds pour lesquels la case vaut 1. Les voisins sont stockés dans l'attribut `voisins` du nœud. ■

```
def make_voisins(self):
    for noeud in self.liste_noeuds:
        ligne = self.matrice_adjacence[noeud.nom]
```

```
        voisins = [nom for nom in ligne if ligne[nom] == 1]
        noeud.voisins = voisins
```

Définition 9.7.
La **liste des voisins** d'un sommet (ou **liste de successeurs** dans un graphe orienté) est la liste des sommets directement reliés à lui. La liste des **prédécesseurs** d'un sommet est la liste des sommets d'où part un arc vers lui. Pour un graphe non orienté, les deux notions coïncident : ce sont les voisins.

R Pour préparer les parcours, on modifie la construction de la matrice : les dictionnaires sont désormais indexés par les **objets** `Noeud` eux-mêmes (et non plus par leurs noms). Ainsi, la méthode `make_voisins` peut stocker dans `noeud.voisins` des **objets** `Noeud`, qui pourront porter l'attribut `visite` utilisé par les parcours.

```
def make_voisins(self):
    for noeud in self.liste_noeuds:
        ligne = self.matrice_adjacence[noeud]
        voisins = [autre for autre in ligne if ligne[autre] == 1]
        noeud.voisins = voisins
```

6. Passer d'une représentation à une autre

■ **Exemple 9.16.**
La méthode `make_voisins` construit les listes d'adjacence à partir de la matrice d'adjacence : elle **passe** d'une représentation à l'autre. Réciproquement, on peut construire la matrice d'adjacence à partir des listes de voisins : c'est l'objet d'un exercice. ■

R La représentation la plus adaptée dépend du traitement :
- la **matrice d'adjacence** permet de savoir en temps constant si deux sommets sont reliés (une simple lecture de case), mais elle occupe toujours n^2 cases, même pour un graphe avec très peu d'arêtes ;
- les **listes d'adjacence** ne stockent que les arêtes existantes : elles sont économes en mémoire pour les graphes « creux » (beaucoup de sommets, peu de liens), comme un réseau social où chaque utilisateur n'a que quelques centaines d'amis parmi des millions de comptes.

7. Générer et afficher un graphe aléatoire

■ **Exemple 9.17.**
On génère un graphe aléatoire : on crée les nœuds, puis on ajoute chaque arête possible avec une probabilité p. Avec $n = 20$ sommets et $p = 0{,}9$, presque tous

les couples de sommets sont reliés : le graphe obtenu est très dense, et il est très probablement connexe. ■

```
import random as rd

n = 20
p = 0.9

liste_noeuds = [Noeud(str(i)) for i in range(n)]

liste_aretes = []
for i in range(n):
    for j in range(i + 1, n):
        if rd.random() < p:
            liste_aretes.append(Arete(liste_noeuds[i], liste_noeuds[j]))

graphe_aleatoire = Graphe(liste_noeuds, liste_aretes)
```

■ **Exemple 9.18.**
Pour visualiser le graphe, on peut utiliser la bibliothèque externe `networkx`, spécialisée dans la manipulation des graphes. On ajoute une méthode `afficher_graphe` à la classe `Graphe`. ■

```
import networkx as nx

class Graphe:
    def afficher_graphe(self):
        g = nx.Graph()
        g.add_nodes_from(self.liste_noeuds)
        g.add_edges_from([[a.noeud_1, a.noeud_2] for a in self.liste_aretes
            ])
        nx.draw(g)
```

R La bibliothèque `networkx` n'est utilisée ici que pour l'**affichage** : elle ne fait pas partie des notions exigibles. En revanche, l'ordre des opérations est essentiel : avant de lancer un parcours (ou un affichage qui en a besoin), il faut avoir calculé la matrice d'adjacence puis les listes de voisins.

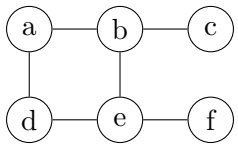
III. Parcourir un graphe

1. Le principe du parcours

Définition 9.8.
Parcourir un graphe consiste à visiter tous les sommets accessibles depuis un sommet de départ, en passant par les arêtes (ou les arcs). Pour ne pas visiter plusieurs fois le même sommet, on le marque comme **visité** : on ajoute pour cela un attribut `visite` aux nœuds.

```
class Noeud:
    def __init__(self, nom):
        self.nom = nom
        self.voisins = []
        self.visite = False # pour les parcours
```

■ **Exemple 9.19.**
Le parcours d'un labyrinthe est un exemple classique d'algorithme sur les graphes : chaque case est un sommet, chaque passage une arête. Partir d'une case et explorer tout le labyrinthe revient à parcourir le graphe des cases. ■



Un labyrinthe à six cases : le graphe des cases et des passages.

R Les deux parcours étudiés dans ce chapitre sont le **parcours en profondeur** (on s'enfonce dans une branche jusqu'au bout avant de revenir en arrière) et le **parcours en largeur** (on explore les sommets par distance croissante au départ, niveau par niveau). Depuis un même sommet, ils visitent exactement les mêmes sommets accessibles, mais dans un ordre différent.

2. Parcours en profondeur (DFS)

■ **Exemple 9.20.**
Le **parcours en profondeur** (depth-first search, DFS) s'écrit naturellement de manière **récursive** : on marque le sommet courant comme visité, on l'affiche, puis on lance le parcours sur chacun de ses voisins non encore visités. ■

```
def parcours_profondeur(sommet):
    sommet.visite = True
    print(sommet.nom)
    for voisin in sommet.voisins:
        if not voisin.visite:
            parcours_profondeur(voisin)
```

R Lorsque tous les voisins d'un sommet ont été visités, l'appel récursif se termine et l'on revient au sommet précédent : c'est le **retour en arrière** (backtracking). La récursivité joue ici le rôle d'une pile : on explore toujours en priorité la branche la plus récemment ouverte.

3. Parcours en largeur (BFS)

■ **Exemple 9.21.**

Le **parcours en largeur** (breadth-first search, BFS) s'écrit de manière **itérative** avec une **file** : on traite les sommets dans l'ordre où ils ont été découverts. Chaque sommet découvert est marqué visité et placé dans la file ; on retire ensuite le premier sommet de la file, on l'affiche, et l'on ajoute ses voisins non visités. ■

```
def parcours_en_largeur(sommet):
    file = [sommet]
    sommet.visite = True
    while len(file) > 0:
        sommet = file.pop(0)
        print(sommet.nom)
        for voisin in sommet.voisins:
            if not voisin.visite:
                file.append(voisin)
                voisin.visite = True
```

R La liste `file` est utilisée comme une file FIFO : on retire le sommet en tête avec `pop(0)` et on ajoute en fin avec `append`. Dans un programme plus performant, on utiliserait la classe `deque` du module `collections`, dont le retrait en tête est plus efficace.

4. Comparer les deux parcours

Le tableau suivant résume les différences entre les deux parcours.

	Parcours en profondeur (DFS)	**Parcours en largeur (BFS)**
Ordre de visite	On s'enfonce branche par branche	On explore niveau par niveau
Structure utilisée	Une pile (ici, la récursivité)	Une file (FIFO)
Écriture	Naturellement récursive	Itérative
Propriété	Explore toute une branche avant de revenir	Découvre les sommets par distance croissante au départ : le premier chemin trouvé est un plus court chemin

■ **Exemple 9.22.**
Sur le graphe G, un parcours en largeur depuis A visite les sommets par distance croissante : d'abord A (distance 0), puis ses voisins B et C (distance 1), puis D (distance 2), puis E (distance 3). Un parcours en profondeur depuis A s'enfonce : A, puis B, puis D, puis C (voisin de D non encore visité), puis E. ■

R Dans un graphe connexe, les deux parcours visitent tous les sommets. Dans un graphe non connexe, ils ne visitent que la **composante** du sommet de départ : l'ensemble des sommets accessibles depuis ce sommet. Pour visiter tout le graphe, il faudrait relancer un parcours depuis chaque sommet non encore visité.

IV. Cycles et chemins

1. Détecter un cycle

■ **Exemple 9.23.**

Dans un réseau routier, un cycle est un itinéraire qui revient à son point de départ sans repasser par le même carrefour. Détecter la présence d'un cycle dans un graphe sert par exemple à repérer une boucle dans un circuit électrique ou une redondance dans un réseau. ■

Définition 9.9.

Un **cycle** est un chemin dont le premier et le dernier sommet sont identiques, tous les autres sommets étant deux à deux distincts. Dans un graphe non orienté simple, un cycle comporte au moins trois sommets.

■ **Exemple 9.24.**

Dans le graphe orienté H, la suite A, B, C, A forme un cycle : en suivant les arcs AB, BC et CA, on revient en A. Le graphe G, non orienté, contient lui aussi un cycle : A, B, D, C, A, en suivant les arêtes AB, BD, DC et CA. ■

```
def contient_cycle(sommet, parent=None):
    sommet.visite = True
    for voisin in sommet.voisins:
        if not voisin.visite:
            if contient_cycle(voisin, sommet):
                return True
        elif voisin != parent:
            return True
    return False
```

R Attention : lorsqu'on arrive sur un sommet v depuis son voisin u, le sommet u est déjà visité, mais l'arête (u, v) n'est pas un cycle : c'est simplement l'arête par laquelle on est arrivé. Il faut donc **exclure le sommet parent** de la détection (paramètre `parent`). Une version naïve qui se contenterait de vérifier si un voisin est déjà visité détecterait un cycle dans un graphe à deux sommets reliés par une seule arête, ce qui est faux : ce retour vers le parent n'est pas un cycle.

2. Chercher un chemin

■ **Exemple 9.25.**

Dans Internet, le **routage** consiste à acheminer un paquet d'une machine à une autre : chaque paquet doit suivre un chemin dans le graphe des routeurs. Savoir s'il existe un chemin entre deux sommets, et le trouver, est donc une question centrale.

■

Définition 9.10.

Chercher un chemin entre deux sommets consiste à déterminer s'il existe une suite de sommets reliant le départ à l'arrivée, chaque sommet étant relié au suivant (en suivant le sens des arcs pour un graphe orienté).

```
def recherche_chemin(depart, arrivee):
    file = [depart]
    depart.visite = True
    while len(file) > 0:
        sommet = file.pop(0)
        for voisin in sommet.voisins:
            if voisin == arrivee:
                return True
            if not voisin.visite:
                file.append(voisin)
                voisin.visite = True
    return False
```

R La recherche de chemin repose sur un parcours en largeur : si l'on atteint le sommet d'arrivée, un chemin existe. Le parcours en largeur découvrant les sommets par distance croissante au départ, le premier chemin trouvé est un **plus court chemin** en nombre d'arêtes. La fonction ci-dessus répond seulement « existe-t-il un chemin ? » ; retrouver la liste des sommets du chemin est l'objet d'un exercice de réflexion.

3. Applications : labyrinthe et routage

Le programme cite deux applications des algorithmes sur les graphes : le parcours d'un labyrinthe et le routage dans Internet.

■ Exemple 9.26.
Dans un labyrinthe, le parcours en profondeur correspond à la méthode « suivre le mur » : on explore un passage jusqu'au bout, et si l'on arrive dans une impasse, on revient sur ses pas pour essayer un autre passage. ■

■ Exemple 9.27.
Le routage se modélise par une recherche de chemin sur le graphe des routeurs : le protocole RIP choisit la route avec le plus petit nombre de sauts (distance en arêtes), le protocole OSPF choisit la route de plus faible coût. Ces deux protocoles reposent sur des algorithmes de recherche de chemin dans un graphe. ■

V. Exercices

Les exercices suivants reprennent et complètent les notions du chapitre. Pensez à bien lire les énoncés, à écrire des programmes propres et à tester vos fonctions sur les petits graphes du cours.

Exercices d'automatismes

Les exercices d'automatismes exercent les réflexes fondamentaux : vocabulaire des graphes, lecture de code court, prédiction d'affichage. Chaque question doit pouvoir être traitée en moins de trente secondes.

Exercice 9.1.
Dire si les affirmations suivantes sont vraies ou fausses.

1. Un graphe orienté possède des arêtes.
2. Dans un graphe non orienté, si A est voisin de B, alors B est voisin de A.
3. Le degré d'un sommet d'un graphe non orienté est son nombre de voisins.
4. Un chemin peut passer deux fois par le même sommet.
5. Dans un graphe non orienté simple, un cycle comporte au moins trois sommets.

■

Exercice 9.2.
On exécute le programme suivant :

```
matrice = {"A": {"A": 0, "B": 1, "C": 0},
           "B": {"A": 1, "B": 0, "C": 1},
           "C": {"A": 0, "B": 1, "C": 0}}
print(matrice["A"]["B"])
print(matrice["C"]["A"])
```

Que va afficher ce programme ? Combien de voisins possède le sommet B ? ■

Exercice 9.3.
On considère la matrice d'adjacence suivante, avec les sommets A, B, C, D dans cet ordre :

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

1. Ce graphe est-il orienté ? Justifier.
2. Combien d'arêtes contient ce graphe ?
3. Quel est le degré du sommet C ?

■

■ **Exercice 9.4.**
On considère le graphe G du cours (sommets A, B, C, D, E ; arêtes AB, AC, BD, CD, DE).

1. Quel est l'ordre de visite d'un parcours en largeur depuis A, en visitant les voisins par ordre alphabétique ?
2. Quel est l'ordre de visite d'un parcours en profondeur depuis A, en visitant les voisins par ordre alphabétique ?

■

■ **Exercice 9.5.**
On rappelle quelques instructions vues en Première. Que valent les variables `liste` et `dictionnaire` après l'exécution du programme suivant ?

```
liste = [i * 2 for i in range(4)]
dictionnaire = {nom: 0 for nom in ["A", "B", "C"]}
print(liste)
print(dictionnaire)
```

■

Exercices d'application

Les exercices d'application demandent d'écrire les classes et les fonctions du chapitre, puis de les tester sur les graphes du cours.

■ **Exercice 9.6.**
On reprend les classes `Noeud`, `Arete` et `Graphe` du cours.

1. Créer un nœud `noeud_1` de nom 1 et un nœud `noeud_2` de nom 2.

2. Créer l'arête reliant ces deux nœuds.
3. Créer un graphe contenant ces deux nœuds et cette arête.
4. Écrire une fonction `make_graphes` qui renvoie deux graphes : le premier avec les nœuds 1, 2, 3 et les arêtes (1, 2) et (1, 3) ; le second avec les nœuds A, B, C et les arêtes (A, B) et (B, C).

■

■ **Exercice 9.7.**
On reprend la classe `Graphe` du cours (version non orientée). Écrire la méthode `make_matrice_adjacence` qui construit la matrice d'adjacence d'un graphe non orienté, puis tester sur le graphe G du cours : pour chaque nœud, la ligne de la matrice doit contenir un 1 exactement pour chacun de ses voisins.

```
def make_matrice_adjacence(self):
    ...
```

■

■ **Exercice 9.8.**
On reprend la classe `Graphe` du cours (version orientée, attribut `oriente`).

1. Écrire la méthode `make_matrice_adjacence` qui construit la matrice d'adjacence d'un graphe orienté.
2. Tester sur le graphe orienté H du cours : la case (A, B) doit valoir 1 mais la case (B, A) doit valoir 0.
3. Écrire une méthode `make_predecesseurs` qui calcule, pour chaque nœud, la liste de ses **prédécesseurs** (les sommets d'où part un arc vers lui), en s'aidant de la matrice d'adjacence.

■

■ **Exercice 9.9.**
On reprend la classe `Graphe` du cours (version où la matrice est indexée par les objets).

1. Écrire la méthode `make_voisins` qui calcule pour chaque nœud la liste de ses voisins (objets `Noeud`).
2. Tester sur le graphe G du cours : vérifier que `liste_noeuds[0].voisins` contient les voisins de A.
3. Réciproquement, écrire une fonction qui construit la matrice d'adjacence d'un graphe non orienté à partir des listes de voisins : passer d'une représentation à l'autre.

■

■ Exercice 9.10.
Générer un graphe aléatoire avec $n = 20$ nœuds et une probabilité $p = 0{,}9$ de conserver chaque arête, comme dans le cours.

1. Construire la liste des nœuds `['0', '1', ..., '19']`.
2. Construire la liste des arêtes en parcourant toutes les paires de nœuds distincts et en conservant chaque paire avec la probabilité p.
3. Créer le graphe, calculer sa matrice d'adjacence, ses listes de voisins, puis l'afficher avec la méthode `afficher_graphe` (bibliothèque `networkx`).

```
import random as rd

n = 20
p = 0.9
liste_noeuds = [Noeud(str(i)) for i in range(n)]
liste_aretes = []
...
```

■

■ Exercice 9.11.
On reprend le graphe aléatoire de l'exercice précédent (ou le graphe G du cours). Écrire la fonction `parcours_profondeur` qui parcourt le graphe en profondeur depuis un sommet donné, en affichant les sommets visités.

```
def parcours_profondeur(sommet):
    ...
```

Avant de lancer le parcours, penser à calculer la matrice d'adjacence et les listes de voisins. Tester depuis le sommet 0 : tous les sommets accessibles doivent être visités. ■

■ Exercice 9.12.
On reprend le même graphe. Écrire la fonction `parcours_en_largeur` qui parcourt le graphe en largeur depuis un sommet donné, en affichant les sommets visités. Comparer l'ordre d'affichage avec celui du parcours en profondeur.

```
def parcours_en_largeur(sommet):
    ...
```

■

■ **Exercice 9.13.**
On reprend le même graphe. Écrire la fonction `contient_cycle` qui détermine si le graphe contient un cycle, en utilisant un parcours récursif avec exclusion du sommet parent.

```
def contient_cycle(sommet, parent=None):
    ...
```

Tester sur le graphe G du cours (qui contient le cycle A-B-D-C-A), sur le graphe orienté H (qui en contient un aussi) et sur le premier graphe construit par `make_graphes` (arêtes (1, 2) et (1, 3)), qui n'en contient pas. ■

■ **Exercice 9.14.**
On reprend le même graphe. Écrire la fonction `recherche_chemin` qui détermine s'il existe un chemin entre deux sommets donnés, en utilisant un parcours en largeur.

```
def recherche_chemin(depart, arrivee):
    ...
```

Tester en cherchant un chemin entre deux sommets tirés au hasard avec `rd.sample(graphe.liste_n 2)`. ■

Exercices de réflexion

Les exercices de réflexion demandent d'analyser les algorithmes du chapitre et d'aller un peu plus loin.

■ **Exercice 9.15.**
La fonction `recherche_chemin` du cours répond seulement si un chemin existe. La modifier pour qu'elle renvoie la **liste des sommets** d'un plus court chemin entre le départ et l'arrivée. On pourra mémoriser, pour chaque sommet découvert, le sommet qui l'a découvert (son « prédécesseur » dans le parcours), puis remonter de l'arrivée vers le départ.

```
def chemin(depart, arrivee):
    file = [depart]
    depart.visite = True
    predecesseur = {depart: None}
    ...
```

■

■ **Exercice 9.16.**
On lance successivement deux parcours sur le même graphe, sans recréer le graphe entre les deux.

1. Expliquer pourquoi le second parcours ne visite alors aucun sommet.
2. Écrire une méthode `reinitialiser_visites` de la classe `Graphe` qui remet l'attribut `visite` de tous les nœuds à `False`.
3. Que se passe-t-il si l'on oublie de marquer un sommet comme visité pendant un parcours en profondeur ? Justifier.

■

■ **Exercice 9.17.**
Un réseau social compte un million d'utilisateurs, chacun ayant au plus quelques centaines d'amis.

1. Combien de cases contiendrait la matrice d'adjacence de ce graphe ? Commenter.
2. Quelle représentation est la plus adaptée pour ce graphe : matrice d'adjacence ou listes d'adjacence ? Justifier.
3. Dans quel cas la matrice d'adjacence est-elle préférable ?

■

■ **Exercice 9.18.**
On veut compter le nombre de **composantes connexes** d'un graphe non orienté : le nombre de « morceaux » de sommets reliés entre eux.

1. Expliquer comment utiliser un parcours (en profondeur ou en largeur) pour compter les composantes connexes : on relancera un parcours depuis chaque sommet non encore visité.
2. Écrire la fonction `nombre_composantes` correspondante.
3. Tester sur un graphe formé de plusieurs morceaux séparés.

■