TP Chapitre 9 : Graphes

Objectifs du TP

Objectifs du TP. À l'issue de cette séance, tu dois être capable de :

— modéliser une situation par un graphe : sommets, arêtes, arcs, graphes orientés ou non orientés ;

— écrire les implémentations d'un graphe : matrice d'adjacence, listes de voisins, successeurs et prédécesseurs ;

— passer d'une représentation à une autre (matrice d'adjacence, listes d'adjacence) ;

— parcourir un graphe en largeur d'abord (BFS) et en profondeur d'abord (DFS) ;

— repérer la présence d'un cycle dans un graphe ;

— chercher un chemin entre deux sommets, et retrouver un plus court chemin en nombre d'arêtes.

Consignes générales. Tous les exercices se traitent en Python (console ou fichier). Enregistre régulièrement ton travail. Avant de lancer un parcours ou une recherche de chemin, pense à calculer la matrice d'adjacence puis les listes de voisins : les algorithmes de ce chapitre en ont besoin. Quand un programme ne fonctionne pas, lis le message d'erreur : il indique souvent la ligne et le type d'erreur. La partie B se fait *en autonomie* : tu écris des programmes entiers à partir d'une spécification, aucune ligne de code n'est fournie.

Partie A — Application guidée

Les exercices suivants se programment en Python. Ils sont classés du plus simple au plus difficile. Chaque exercice se termine par un *point de validation* (encadré vert) : le résultat exact que doit produire ton programme pour être juste. Remplace chaque ligne `# A COMPLETER` par les instructions Python correspondantes, en respectant l'indentation.

Exercice A1 — Le réseau de magasins : lire une matrice d'adjacence (vocabulaire des graphes)

Une chaîne de cinq magasins A, B, C, D, E est reliée par des liaisons de livraison : chaque liaison peut être parcourue dans les deux sens, le graphe est donc *non orienté*. Les arêtes du graphe sont AB, AC, BD, CD et DE. La matrice d'adjacence est donnée ci-dessous sous la forme d'un dictionnaire de dictionnaires : la case (X, Y) vaut 1 si les magasins X et Y sont reliés, 0 sinon. Deux sommets reliés par une arête sont *adjacents* ; le *degré* d'un sommet est son nombre de voisins.

Compléter le programme qui affiche : (1) les voisins de chaque magasin ; (2) le degré de chaque sommet ; (3) la liste des arêtes, chaque arête n'apparaissant qu'une seule fois.

```
matrice = {
    "A": {"A": 0, "B": 1, "C": 1, "D": 0, "E": 0},
    "B": {"A": 1, "B": 0, "C": 0, "D": 1, "E": 0},
    "C": {"A": 1, "B": 0, "C": 0, "D": 1, "E": 0},
    "D": {"A": 0, "B": 1, "C": 1, "D": 0, "E": 1},
    "E": {"A": 0, "B": 0, "C": 0, "D": 1, "E": 0},
}
sommets = ["A", "B", "C", "D", "E"]

# 1. Voisins de chaque sommet
for sommet in sommets:
    voisins = []
    for autre in sommets:
```

```
        # A COMPLETER : ajouter autre a voisins si la case vaut 1
    print("Voisins de", sommet, ":", voisins)

# 2. Degre de chaque sommet
for sommet in sommets:
    degre = 0
    for autre in sommets:
        # A COMPLETER : compter un voisin de plus si la case vaut 1
    print("Degre de", sommet, ":", degre)

# 3. Liste des aretes, chaque arete une seule fois
aretes = []
for i in range(len(sommets)):
    for j in range(i + 1, len(sommets)):
        # A COMPLETER : ajouter le couple (sommets[i], sommets[j]) si la
            case vaut 1
print("Aretes :", aretes)
```

Point de validation

Le programme doit afficher `Voisins de A : ['B', 'C']`, `Degre de A : 2`, `Degre de D : 3`, et `Aretes : [('A', 'B'), ('A', 'C'), ('B', 'D'), ('C', 'D'), ('D', 'E')]` (cinq arêtes).

Exercice A2 — Construire la matrice d'adjacence d'un réseau de magasins (graphe non orienté)

On reprend les classes `Noeud`, `Arete` et `Graphe` du cours. Compléter la méthode `make_matrice_adjacence` : pour chaque couple de sommets, la case doit valoir 1 si une arête relie les deux sommets (dans un sens *ou* dans l'autre), 0 sinon ; la diagonale vaut 0 (pas de boucle). Le graphe est construit par la fonction `make_graphe` : cinq magasins A, B, C, D, E et les arêtes AB, AC, BD, CD, DE.

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

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

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 # A COMPLETER : l'arete relie les deux sommets dans un
                    sens ou dans l'autre
                    self.matrice_adjacence[noeud_ligne.nom][noeud_colonne.
                        nom] = 1
                else:
                    self.matrice_adjacence[noeud_ligne.nom][noeud_colonne.
                        nom] = 0

def make_graphe():
    noeud_a = Noeud("A")
    noeud_b = Noeud("B")
    noeud_c = Noeud("C")
    noeud_d = Noeud("D")
    noeud_e = Noeud("E")
    return Graphe(
        [noeud_a, noeud_b, noeud_c, noeud_d, noeud_e],
        [Arete(noeud_a, noeud_b), Arete(noeud_a, noeud_c),
         Arete(noeud_b, noeud_d), Arete(noeud_c, noeud_d),
         Arete(noeud_d, noeud_e)]
    )

graphe = make_graphe()
graphe.make_matrice_adjacence()
print(graphe.matrice_adjacence)
```

Point de validation

La matrice affichée doit être identique à celle de l'exercice A1 : la case `("A", "B")` vaut 1, la case `("B", "A")` vaut aussi 1 (graphe non orienté : la matrice est symétrique), et la diagonale est nulle.

Exercice A3 — La liste des voisins : passer de la matrice aux listes d'adjacence

On reprend la classe `Graphe` de l'exercice A2 et on ajoute la méthode `make_voisins`. Pour chaque nœud, elle calcule la liste de ses *voisins* : les noms des sommets pour lesquels la ligne de la matrice vaut 1. On passe ainsi de la représentation « matrice d'adjacence » à la représentation « listes d'adjacence » : c'est la même information, rangée autrement.

```
class Graphe:
    # ... (classe de l'exercice A2)

    def make_voisins(self):
        for noeud in self.liste_noeuds:
            ligne = self.matrice_adjacence[noeud.nom]
            voisins = []
            for autre in self.liste_noeuds:
                # A COMPLETER : ajouter autre.nom a voisins si la case vaut
                    1
            noeud.voisins = voisins

graphe = make_graphe()
graphe.make_matrice_adjacence()
graphe.make_voisins()
for noeud in graphe.liste_noeuds:
    print(noeud.nom, ":", noeud.voisins)
```

Point de validation

Le programme doit afficher `A : ['B', 'C']`, `B : ['A', 'D']`, `C : ['A', 'D']`, `D : ['B', 'C', 'E']` et `E : ['D']`.

Exercice A4 — Parcourir un réseau de paiement en largeur (BFS)

Une fintech modélise son réseau de paiement par un graphe non orienté : les comptes sont les sommets, et une arête relie deux comptes qui peuvent s'échanger de l'argent (dans les deux sens). Depuis un compte de départ, on veut visiter *tous les comptes accessibles*, niveau par niveau : d'abord le compte de départ, puis ses voisins directs, puis leurs voisins, etc. C'est le *parcours en largeur* (BFS). Il s'écrit de manière itérative avec une *file* (FIFO) : on retire le premier sommet de la file, on l'affiche, et l'on ajoute ses voisins non encore visités.

On fournit la classe `Noeud` enrichie (attributs `voisins` et `visite`) et la classe `Graphe` « version objets » : la matrice est désormais indexée par les *objets* `Noeud`, ce qui permet de stocker dans `noeud.voisins` des objets portant l'attribut `visite`. Compléter la fonction `parcours_en_largeur`.

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

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

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

    def make_matrice_adjacence(self):
        self.matrice_adjacence = {
            noeud_ligne: {noeud_colonne: 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][noeud_colonne] = 0
                elif ([noeud_ligne, noeud_colonne] in liste_arete_temp
                      or [noeud_colonne, noeud_ligne] in liste_arete_temp):
                    self.matrice_adjacence[noeud_ligne][noeud_colonne] = 1
                else:
                    self.matrice_adjacence[noeud_ligne][noeud_colonne] = 0

    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
```

```
40
41 def parcours_en_largeur(sommet):
42     file = [sommet]
43     sommet.visite = True
44     while len(file) > 0:
45         # A COMPLETER : retirer le premier sommet de la file
46         print(sommet.nom)
47         for voisin in sommet.voisins:
48             if not voisin.visite:
49                 # A COMPLETER : ajouter le voisin a la file et le marquer
                       visite
50
51 def make_reseau():
52     a = Noeud("A")
53     b = Noeud("B")
54     c = Noeud("C")
55     d = Noeud("D")
56     e = Noeud("E")
57     return Graphe(
58         [a, b, c, d, e],
59         [Arete(a, b), Arete(a, c), Arete(b, d), Arete(c, d), Arete(d, e)]
60     )
61
62 reseau = make_reseau()
63 reseau.make_matrice_adjacence()
64 reseau.make_voisins()
65 parcours_en_largeur(reseau.liste_noeuds[0])
```

Point de validation

Depuis le compte A, le programme doit afficher `A`, `B`, `C`, `D`, `E` dans cet ordre : les sommets sont visités par distance croissante au départ (A à la distance 0, B et C à la distance 1, D à la distance 2, E à la distance 3).

Partie B — En autonomie

Tu écris maintenant des programmes *entiers*, sans squelette. Pour chaque exercice : lis la spécification (entrées, traitement, sorties, contraintes), réponds d'abord aux questions de conception de l'encadré orange (sur papier ou à voix haute), puis écris ton programme dans un fichier et teste-le avec les valeurs de l'encadré vert. L'objectif est de concevoir toi-même la solution : ne demande pas de lignes de code toutes faites.

Exercice B1 — Le graphe des transactions entre comptes (graphe orienté, successeurs et prédécesseurs)

Une banque modélise les transactions entre comptes par un *graphe orienté* : les comptes sont les sommets, et chaque arc $A \to B$ représente une transaction du compte A vers le compte B. On considère quatre comptes A, B, C, D et les transactions $A \to B$, $B \to C$, $C \to A$, $B \to D$. Dans un graphe orienté, on parle de *successeurs* (les sommets vers lesquels part un arc) et de *prédécesseurs* (les sommets d'où vient un arc).

Énoncé. Écrire un programme qui :

1. définit les classes `Noeud`, `Arete` et `Graphe` (avec l'attribut `oriente` à `True`) ainsi que la méthode `make_matrice_adjacence` adaptée à un graphe orienté : seul l'arc qui va de l'origine vers la destination vaut 1 ;

2. construit le graphe des transactions ci-dessus ;
3. affiche la matrice d'adjacence, ligne par ligne ;
4. calcule et affiche, pour chaque compte, la liste de ses successeurs (méthode `make_successeurs`) et la liste de ses prédécesseurs (méthode `make_predecesseurs`).

Spécification.

— *Entrées* : les données du graphe (comptes A, B, C, D ; transactions $A \to B$, $B \to C$, $C \to A$, $B \to D$), écrites dans le programme.

— *Traitement* : construire la matrice orientée ; en déduire les successeurs (case à 1 sur la *ligne* du compte) et les prédécesseurs (case à 1 sur la *colonne* du compte).

— *Sorties* : la matrice d'adjacence (une ligne par compte), puis pour chaque compte ses successeurs et ses prédécesseurs.

— *Contraintes* : attribut `oriente = True` ; dans la matrice, seule la case (origine, destination) est remplie ; pas de saisie clavier.

Pour t'aider — questions de conception (sans donner le code)

— Pourquoi la matrice d'un graphe orienté n'est-elle pas symétrique ?

— Dans la matrice, comment reconnaître les successeurs d'un compte ? les prédécesseurs ?

— Quel est le degré entrant du compte C ? le degré sortant du compte B ? (revoir le vocabulaire du cours)

Point de validation

La case `("A", "B")` vaut 1 mais la case `("B", "A")` vaut 0. Successeurs : A : `['B']`, B : `['C', 'D']`, C : `['A']`, D : `[]`. Prédécesseurs : A : `['C']`, B : `['A']`, C : `['B']`, D : `['B']`.

Exercice B2 — Explorer le réseau de livraison en profondeur (DFS et réinitialisation)

Une chaîne de magasins veut vérifier que tous ses magasins sont joignables depuis son entrepôt central A, en suivant les liaisons de livraison (graphe non orienté, arêtes AB, AC, BD, CD, DE). Le *parcours en profondeur* (DFS) explore une branche jusqu'au bout avant de revenir en arrière ; il s'écrit naturellement de manière *récursive*.

Énoncé. Écrire un programme qui :

1. définit la classe `Noeud` (avec les attributs `voisins` et `visite`) et la classe `Graphe` « version objets » (attribut `oriente = False`) avec les méthodes `make_matrice_adjacence` et `make_voisins` de l'exercice A4 ;
2. définit la fonction `parcours_profondeur(sommet)` qui marque le sommet comme visité, l'affiche, puis se relance sur chacun de ses voisins non encore visités ;
3. définit la fonction `reinitialiser_visites(graphe)` qui remet l'attribut `visite` de tous les sommets à `False` ;
4. construit le graphe des cinq magasins, lance le parcours depuis A, puis relance un second parcours depuis A *sans* réinitialiser, puis un troisième parcours après réinitialisation.

Spécification.

— *Entrées* : un graphe non orienté (sommets A, B, C, D, E ; arêtes AB, AC, BD, CD, DE) et le sommet de départ A.

— *Traitement* : parcours récursif en profondeur ; marquage des sommets visités ; réinitialisation des visites entre deux parcours.

— *Sorties* : l'ordre de visite des sommets (un nom par ligne).

— *Contraintes* : récursivité ; avant tout parcours, calculer la matrice d'adjacence puis les listes de voisins ; ne jamais visiter deux fois le même sommet.

Pour t'aider — questions de conception (sans donner le code)

— Pourquoi marquer un sommet comme visité *avant* de l'afficher (et pas après) ?

— Pourquoi la récursivité joue-t-elle le rôle d'une pile dans ce parcours ?

— Que se passe-t-il si l'on oublie de marquer un sommet visité ? (tester mentalement sur deux sommets reliés par une arête)

— Pourquoi faut-il réinitialiser les visites avant un second parcours ?

Point de validation

Premier parcours depuis A (voisins visités par ordre alphabétique) : `A`, `B`, `D`, `C`, `E`. Le second parcours, lancé sans réinitialisation, n'affiche que `A`. Après `reinitialiser_visites`, le troisième parcours affiche de nouveau `A`, `B`, `D`, `C`, `E`.

Exercice B3 — La livraison est-elle possible ? (recherche de chemin)

Le service logistique veut savoir si un magasin peut être livré depuis l'entrepôt, en suivant les liaisons existantes. Le réseau compte les cinq magasins A, B, C, D, E (arêtes AB, AC, BD, CD, DE) et un sixième magasin F, qui n'est relié à aucun autre pour l'instant.

Énoncé. Écrire une fonction `recherche_chemin(depart, arrivee)` qui renvoie `True` s'il existe un chemin reliant `depart` à `arrivee`, `False` sinon, en s'appuyant sur un parcours en largeur (file FIFO). Écrire ensuite un programme qui teste la fonction : de A vers E ; de A vers F ; de E vers A.

Spécification.

— *Entrées* : un graphe non orienté (sommets A à F ; arêtes AB, AC, BD, CD, DE ; F isolé) et deux sommets `depart` et `arrivee`.

— *Traitement* : parcours en largeur depuis `depart` ; dès que le sommet `arrivee` est découvert, renvoyer `True` ; si le parcours se termine sans l'atteindre, renvoyer `False`.

— *Sorties* : un booléen (`True` ou `False`).

— *Contraintes* : file FIFO (liste avec `append` et `pop(0)`) ; marquer les sommets visités ; réinitialiser les visites entre deux appels (ou recréer le graphe).

Pour t'aider — questions de conception (sans donner le code)

— Pourquoi un parcours (en largeur ou en profondeur) suffit-il pour répondre « existe-t-il un chemin ? » ?

— Pourquoi faut-il marquer les sommets visités ? (que se passerait-il sinon ?)

— Peut-on répondre `True` sans avoir visité tout le graphe ? Quand ?

— Quel est le rôle de la file dans le parcours en largeur ?

Point de validation

La fonction doit renvoyer `True` pour `recherche_chemin(A, E)`, `False` pour `recherche_chemin(A, F)` et `True` pour `recherche_chemin(E, A)`.

Exercice B4 — Combien de zones de livraison indépendantes ? (composantes connexes)

Le réseau de livraison complet compte les magasins A, B, C, D, E (arêtes AB, AC, BD, CD, DE) et le magasin F, isolé pour l'instant. Un camion ne peut passer d'une « zone » à une autre que s'il existe une suite de liaisons : le réseau est formé de *composantes connexes*. On veut les compter : c'est le nombre de « morceaux » de sommets reliés entre eux.

Énoncé. Écrire une fonction `nombre_composantes(graphe)` qui renvoie le nombre de composantes connexes d'un graphe non orienté, en relançant un parcours (en profondeur ou en largeur) depuis chaque sommet non encore visité. Tester sur le réseau A, B, C, D, E, F décrit ci-dessus, puis sur le seul réseau A, B, C, D, E.

Spécification.

— *Entrées* : un graphe non orienté (sommets, arêtes).

— *Traitement* : pour chaque sommet, si ce sommet n'est pas visité : lancer un parcours qui marque toute sa composante, et augmenter le compteur de 1.

— *Sorties* : le nombre de composantes connexes (un entier).

— *Contraintes* : réutiliser la fonction `parcours_profondeur` (ou un parcours en largeur) ; remettre toutes les visites à `False` avant de compter.

Pour t'aider — questions de conception (sans donner le code)

— Pourquoi un seul parcours ne suffit-il pas à tout visiter quand le graphe est non connexe ?

— Comment être sûr de ne pas compter deux fois la même composante ?

— Combien de parcours la fonction lance-t-elle au total sur le réseau A, B, C, D, E, F ? (comparer avec le nombre de composantes trouvé)

Point de validation

Avec le réseau A, B, C, D, E, F (F isolé) : `2` composantes. Avec le seul réseau A, B, C, D, E : `1` composante.

Partie C — Exercice de réflexion (à finir à la maison)

À finir à la maison

La livraison au plus court et le cycle suspect.

Contexte. Une chaîne de magasins en ligne livre ses magasins depuis un entrepôt central. Le réseau de livraison est modélisé par un graphe non orienté : les magasins sont les sommets, les liaisons les arêtes. Chaque liaison correspond à un trajet de camion facturé au même prix (par exemple 120 €) : le chemin qui passe par le *moins* de liaisons est donc aussi le moins cher. En parallèle, le service comptable surveille les transactions entre les comptes internes de la chaîne : dans un graphe orienté de transactions, un *cycle* (l'argent fait un tour complet et revient à son point de départ) est un signal de fraude classique. Le principe de détection d'un cycle est le même que celui d'une *redondance* dans un réseau de livraison : un itinéraire qui revient à son point de départ sans repasser par le même sommet, c'est-à-dire des kilomètres parcourus pour rien.

Objectif. Sur le réseau des cinq magasins des parties A et B (arêtes AB, AC, BD, CD, DE), écrire un programme qui :

1. renvoie la liste des sommets d'un *plus court chemin* entre deux sommets donnés (fonction `plus_court_chemin(depart, arrivee)`), et une liste vide s'il n'existe aucun chemin ;

2. détermine si un graphe contient un *cycle* (fonction `contient_cycle(sommet, parent=None)`), en s'inspirant du parcours en profondeur.

Pour la détection de cycle, on testera sur un plan simplifié du réseau, formé des liaisons AB, AC, BD et CE : ce plan est un *arbre*, il ne contient aucun cycle. On testera ensuite le même plan augmenté de la liaison BE : il doit alors contenir un cycle.

Questions à se poser avant de coder.

— Comment mémoriser, pour chaque sommet découvert pendant le parcours en largeur, le sommet qui l'a découvert ? (penser à un dictionnaire : clé = sommet découvert, valeur = sommet découvreur)

— Comment reconstruire le chemin en remontant de l'arrivée vers le départ ? Quand s'arrêter ?

— Pourquoi le premier chemin trouvé par un parcours en largeur est-il un plus court chemin en nombre d'arêtes ?

— Pour la détection de cycle : pourquoi faut-il exclure le sommet *parent* (celui par lequel on est arrivé) ? (reprendre la remarque du cours)

— Comment garantir que les parcours se terminent et ne repassent pas éternellement sur les mêmes sommets ?

Étapes suggérées.

1. Version 0 : reprendre la fonction `recherche_chemin` de l'exercice B3 (réponse booléenne) et la faire tourner sur le réseau des cinq magasins.
2. Version 1 : ajouter le dictionnaire des prédécesseurs et renvoyer la liste des sommets du chemin ; vérifier sur un petit exemple : de A vers E, un plus court chemin est `['A', 'B', 'D', 'E']` (il y en a un autre : `['A', 'C', 'D', 'E']`).
3. Version 2 : gérer le cas « pas de chemin » en renvoyant une liste vide (par exemple vers un magasin F isolé, comme dans l'exercice B3).
4. Version 3 : détection de cycle avec exclusion du parent ; tester sur le plan simplifié AB, AC, BD, CE (pas de cycle), puis sur le même plan augmenté de BE (cycle).
5. Bonus : réfléchir à la détection de cycle dans un *graphe orienté* de transactions : la notion de « parent » a-t-elle le même sens ? Un sommet déjà visité rencontré pendant le parcours indique-t-il toujours un cycle ?

Contraintes. Programmes testés sur les graphes du TP (réseau des cinq magasins et plan simplifié) ; programmes commentés ; réutiliser les classes `Noeud`, `Arete` et `Graphe` des exercices précédents ; penser à réinitialiser les visites entre deux appels ; les messages peuvent être écrits sans accents (ex. `chemin`, `cycle`) si ton éditeur pose problème.

Fin du TP — la partie C est à finir à la maison.