Partie 1

Algorithmes sur les Graphes

1.1 Exercice 1 : Échauffement - Maximum dans un dictionnaire

Objectif : Se familiariser avec la manipulation des dictionnaires avant de travailler sur les graphes.

Soit un dictionnaire dont les clés sont des entiers et les valeurs sont des chaînes de caractères.

```
notes = {15: "Alice", 12: "Bob", 18: "Charlie", 14: "Diana"}
```

Question : Compléter la fonction `max_cle(dico)` qui retourne la clé maximale du dictionnaire.

```
def max_cle(dico):
    """Retourne la clé maximale du dictionnaire"""
    maximum = None
    for cle in dico:
        if maximum is None or ... :  # À compléter
            maximum = ...  # À compléter
    return maximum
```

Exemple d'utilisation :

```
>>> max_cle(notes)
18
```

1.2 Exercice 2 : Représentation d'un graphe

On représente un graphe orienté par un dictionnaire d'adjacence où : - Chaque clé est un sommet (chaîne de caractères) - Chaque valeur est la liste des sommets adjacents (successeurs)

Exemple :

```
graphe1 = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['D'],
    'D': []
}
```

Questions :

2.1 Dessiner le graphe correspondant à `graphe1`.

2.2 Compléter la fonction `nombre_sommets(graphe)` qui retourne le nombre de sommets du graphe.

```
def nombre_sommets(graphe):
    """Retourne le nombre de sommets du graphe"""
    return len(...)  # À compléter
```

2.3 Écrire une fonction `nombre_aretes(graphe)` qui retourne le nombre d'arêtes (arcs) du graphe.

1.3 Exercice 3 : Degré des sommets

Pour un graphe orienté : - Le **degré sortant** d'un sommet est le nombre d'arcs qui partent de ce sommet - Le **degré entrant** d'un sommet est le nombre d'arcs qui arrivent à ce sommet

Questions :

3.1 Compléter la fonction `degre_sortant(graphe, sommet)` qui retourne le degré sortant d'un sommet.

```
def degre_sortant(graphe, sommet):
    """Retourne le degré sortant d'un sommet"""
    return len(graphe[...])  # À compléter
```

3.2 Écrire une fonction `degre_entrant(graphe, sommet)` qui retourne le degré entrant d'un sommet.

3.3 Écrire une fonction `sommet_isole(graphe)` qui retourne la liste des sommets isolés (degré entrant et sortant égaux à 0).

Exemple avec `graphe1` :

```
>>> degre_sortant(graphe1, 'A')
2
>>> degre_entrant(graphe1, 'D')
2
```

1.4 Exercice 4 : Parcours en largeur (BFS)

Le parcours en largeur explore le graphe niveau par niveau à partir d'un sommet de départ.

Question : Compléter la fonction `parcours_largeur(graphe, depart)` qui retourne la liste des sommets visités dans l'ordre du parcours en largeur.

```
def parcours_largeur(graphe, depart):
    """Parcours en largeur à partir du sommet depart"""
    visites = []
    a_visiter = [depart]  # File d'attente

    while len(a_visiter) > 0:
        sommet = a_visiter.pop(0)  # Retirer le premier élément (file FIFO)

        if sommet not in visites:
            visites.append(...)  # À compléter

            # Ajouter les voisins non visités à la file
            for voisin in graphe[sommet]:
                if voisin not in visites and voisin not in a_visiter:
                    a_visiter.append(...)  # À compléter

    return visites
```

Exemple :

```
graphe2 = {
```

```
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

>>> parcours_largeur(graphe2, 'A')
['A', 'B', 'C', 'D', 'E', 'F']
```

Question supplémentaire : Que retourne `parcours_largeur(graphe2, 'B')` ? Déterminez le résultat à la main avant de tester.

1.5 Exercice 5 : Parcours en profondeur (DFS)

Le parcours en profondeur explore le graphe en allant le plus loin possible dans chaque branche avant de revenir en arrière.

Question 5.1 : Écrire une fonction `parcours_profondeur(graphe, depart)` qui retourne la liste des sommets visités dans l'ordre du parcours en profondeur.

Algorithme : 1. Utiliser une pile (avec une liste Python) 2. Marquer les sommets visités 3. Pour chaque sommet, explorer en profondeur avant de passer au suivant

Remarque : On peut aussi implémenter une version récursive.

Question 5.2 : Que retourne `parcours_profondeur(graphe2, 'A')` ? Déterminez le résultat à la main avant de tester votre fonction.

Rappel : `graphe2` est le graphe défini dans l'exercice 4.

1.6 Exercice 6 : Détection de chemin

Question : Compléter la fonction `existe_chemin(graphe, depart, arrivee)` qui retourne `True` s'il existe un chemin entre le sommet `depart` et le sommet `arrivee`, et `False` sinon.

```
def existe_chemin(graphe, depart, arrivee):
    """Retourne True s'il existe un chemin de depart vers arrivee"""
    visites = parcours_largeur(graphe, ...)  # À compléter
    return ... in visites  # À compléter
```

Exemple :

```
>>> existe_chemin(graphe2, 'A', 'F')
True
>>> existe_chemin(graphe2, 'D', 'A')
False
```

1.7 Exercice 7 : Plus court chemin (BFS)

Dans un graphe non pondéré, le parcours en largeur permet de trouver le plus court chemin (en nombre d'arêtes).

Question : Compléter la fonction `plus_court_chemin(graphe, depart, arrivee)` qui retourne la liste des sommets formant le plus court chemin entre `depart` et `arrivee`. Si aucun chemin n'existe, retourner `None`.

```python
def plus_court_chemin(graphe, depart, arrivee):
    """Retourne le plus court chemin entre depart et arrivee"""
    if depart == arrivee:
        return [depart]

    predecesseurs = {depart: None}  # Dictionnaire des prédécesseurs
    a_visiter = [depart]

    while len(a_visiter) > 0:
        sommet = a_visiter.pop(0)

        for voisin in graphe[sommet]:
            if voisin not in predecesseurs:
                predecesseurs[voisin] = ...  # À compléter
                a_visiter.append(voisin)

                # Si on a atteint l'arrivée, reconstruire le chemin
                if voisin == arrivee:
                    chemin = []
                    s = arrivee
                    while s is not None:
                        chemin.insert(0, s)  # Insérer au début
                        s = predecesseurs[...]  # À compléter
                    return chemin

    return None  # Aucun chemin trouvé
```

Exemple :

```
>>> plus_court_chemin(graphe2, 'A', 'F')
['A', 'C', 'F']
```

1.8 Exercice 8 : Détection de cycle

Un graphe orienté contient un cycle s'il existe un chemin d'un sommet vers lui-même.

Question : Compléter la fonction `contient_cycle(graphe)` qui retourne `True` si le graphe contient au moins un cycle, `False` sinon.

```python
def contient_cycle(graphe):
    """Retourne True si le graphe contient un cycle"""
    NON_VISITE = 0
    EN_COURS = 1
```

```
    TERMINE = 2

    etat = {sommet: NON_VISITE for sommet in graphe}

    def visite_profondeur(sommet):
        etat[sommet] = ...  # À compléter (marquer comme en cours)

        for voisin in graphe[sommet]:
            if etat[voisin] == EN_COURS:
                return True  # Cycle détecté !
            elif etat[voisin] == NON_VISITE:
                if visite_profondeur(voisin):
                    return ...  # À compléter

        etat[sommet] = TERMINE
        return False

    # Tester tous les sommets (pour graphes non connexes)
    for sommet in graphe:
        if etat[sommet] == NON_VISITE:
            if visite_profondeur(sommet):
                return True

    return False
```

Testez votre fonction sur :

```
graphe_cycle = {
    'A': ['B'],
    'B': ['C'],
    'C': ['A']  # Cycle A -> B -> C -> A
}

graphe_sans_cycle = {
    'A': ['B'],
    'B': ['C'],
    'C': []
}
```

1.9 Exercice 9 : Graphe fortement connexe

Un graphe orienté est **fortement connexe** si pour toute paire de sommets (u, v), il existe un chemin de u vers v ET un chemin de v vers u.

Question : Écrire une fonction `est_fortement_connexe(graphe)` qui retourne `True` si le graphe est fortement connexe.

Indication : Une approche simple consiste à vérifier, pour chaque paire de sommets, qu'il existe un chemin dans les deux sens (mais cela peut être optimisé).

1.10 Exercice 10 : Tri topologique

Un tri topologique d'un graphe orienté acyclique (DAG) est un ordre linéaire des sommets tel que pour chaque arc (u, v), u apparaît avant v dans l'ordre.

Questions :

10.1 Écrire une fonction `tri_topologique(graphe)` qui retourne une liste représentant un tri topologique du graphe. Si le graphe contient un cycle, retourner `None`.

10.2 Tester votre fonction sur le graphe suivant représentant des dépendances entre tâches :

```
taches = {
    'A': ['C'],
    'B': ['C', 'D'],
    'C': ['E'],
    'D': ['E'],
    'E': []
}
```

Algorithme de Kahn :

1. Calculer le degré entrant de chaque sommet
2. Mettre dans une file tous les sommets de degré entrant 0
3. Tant que la file n'est pas vide :
    - Retirer un sommet et l'ajouter au résultat
    - Diminuer le degré entrant de ses voisins
    - Ajouter à la file les voisins dont le degré entrant devient 0
4. Si tous les sommets ont été traités, retourner le résultat, sinon il y a un cycle