Fiche Méthode : Graphes & Algorithmique (NSI)

1. Vocabulaire et Définitions Fondamentales

- **Graphe (G) :** Un ensemble de **sommets** (V) reliés par des **arêtes** (E).
- **Graphe Orienté :** Les arêtes ont un sens (on parle d'**arcs**).
- **Adjacence :** Deux sommets sont dits **adjacents** s'ils sont reliés par une arête.
- **Degré :** Nombre d'arêtes incidentes à un sommet (le nombre de ses voisins).
- **Graphe Connexe :** S'il existe toujours un chemin entre n'importe quelle paire de sommets.
- **Cycle :** Une suite d'arêtes revenant au sommet de départ sans repasser deux fois par la même arête.
- **Graphe Acyclique :** Un graphe qui ne contient aucun cycle (ex: un arbre ou un DAG).

---

2. Représentations en Python

Il existe deux manières principales de stocker un graphe en mémoire :

A. Matrice d'adjacence (Liste de listes)

Un tableau 2D de taille n×n. M[i][j] = 1 (ou True) si une arête existe, 0 (ou False) sinon.

- **Avantage :** Accès instantané pour savoir si i et j sont reliés.
- **Inconvénient :** Très gourmand en mémoire si le graphe a peu d'arêtes.

B. Dictionnaire d'adjacence

Une structure où chaque clé est un sommet et chaque valeur est la liste de ses voisins.

- **Avantage :** Plus efficace pour parcourir les voisins d'un sommet.
- **Exemple :** G = {"A": ["B", "C"], "B": ["A", "D"], ...}

---

3. Méthode de Coloration (Algorithme de Welsh-Powell)

Objectif : Colorer les sommets de sorte que deux sommets adjacents n'aient pas la même couleur, en utilisant le minimum de couleurs (nombre chromatique $\chi(G)$).

Algorithme :

1. Classer les sommets par **ordre décroissant de degré**.
2. Prendre la première couleur et l'attribuer au premier sommet de la liste.
3. Parcourir la liste et attribuer cette même couleur à chaque sommet qui n'est **adjacent à aucun sommet déjà coloré** avec cette couleur.
4. Recommencer avec une nouvelle couleur pour les sommets restants jusqu'à épuisement.

Pour vous entrainer : https://gemini.google.com/share/1852033955a6

Astuces pour le Contrôle

- **Matrice d'adjacence :** Vérifiez toujours si le graphe est orienté ou non. S'il n'est pas orienté, la matrice doit être symétrique par rapport à la diagonale.
- **Coloration :** Si vous trouvez un triangle (3 sommets reliés entre eux), vous aurez besoin d'au moins 3 couleurs.
- **Piles/Files :** En Python, pour simuler une **Pile**, on utilise L.append() et L.pop(). Pour une **File**, on utilise L.append() et L.pop(0).

Bonus (à consulter après avoir fait le devoir blanc)

---

4. Stratégies de Parcours (Initiation)

Même sans avoir fait le cours formel, on peut explorer un graphe via deux structures de données :

Structure	Type de Parcours	Logique	Résultat visuel
File (FIFO)	**Largeur** (BFS)	On traite les voisins dans l'ordre d'arrivée.	On explore "en couches" (tous les voisins à distance 1, puis distance 2...).
Pile (LIFO)	**Profondeur** (DFS)	On traite le dernier voisin découvert immédiatement.	On explore "un tunnel" jusqu'au bout avant de revenir en arrière.

---

5. Lien avec la Programmation Dynamique

Lorsqu'on cherche à compter des chemins ou trouver un coût optimal dans un **Graphe Orienté Acyclique (DAG)** :

- **Le problème :** Un sommet peut être atteint par plusieurs chemins. Un calcul récursif "naïf" recalculerait plusieurs fois la même chose.
- **La solution :** On stocke le résultat du calcul pour chaque sommet dans un dictionnaire memo.
- **Principe :** Valeur(Sommet) = Optimisation(Valeur(Voisins)).

Sujet 1 : Devoir d'entraînement (90 min)

Objectif : Consolider les bases de représentation et l'usage des structures linéaires.

Exercice 1 : Le Réseau de Transport (60 min - 14 points)

Inspiré des sujets E3C et épreuves pratiques.

On modélise un réseau de bus par un graphe dont les sommets sont des arrêts et les arêtes les lignes directes. On utilise la classe Graphe suivante utilisant un dictionnaire d'adjacence.

```
class Graphe:
    def __init__(self, sommets):
        self.sommets = sommets
        self.adj = {s: [] for s in sommets}

    def ajouter_arete(self, u, v):
        if v not in self.adj[u]:
            self.adj[u].append(v)
        if u not in self.adj[v]:
            self.adj[v].append(u)

    def voisins(self, u):
        return self.adj[u]
```

Partie A : Manipulation de base

1. Écrire une méthode *degre(self, u)* qui renvoie le nombre de voisins de l'arrêt u.

2. On considère les arrêts "A", "B", "C", "D". Écrire les instructions Python pour créer le graphe et ajouter les arêtes (A,B), (B,C) et (B,D).

Partie B

On souhaite vérifier si on peut aller d'un arrêt *depart* à un arrêt *arrivee*. On utilise pour cela une **File**. On suppose qu'une classe File est déjà définie avec les méthodes *enfiler(x)*, *defiler()* et *est_vide()*.

3. **Compléter** la fonction *est_accessible(g, depart, arrivee)* ci-dessous :

```
def est_accessible(g, depart, arrivee):
    f = File()
    f.enfiler(depart)
    visites = [depart]
    while not f.est_vide():
        u = f.defiler()
        if u == arrivee:
            return True
        for v in g.voisins(u):
            if v not in visites:
                visites.append(v)
                f.enfiler(v) # <-- Ligne à compléter
    return False
```

4. **Analyse du comportement :**
   On considère un graphe où l'arrêt "A" est relié à "B" et "C". "B" est relié à "D". Si on lance *est_accessible(g, "A", "D")* :
   - a) Quels sont les sommets qui seront présents dans la File juste après avoir traité le sommet "A" ?
   - b) Lequel de ces sommets sera traité (défilé) en premier ?
   - c) Entre "B" et "D", lequel sera découvert en premier ? Est-ce que l'algorithme explore d'abord tous les voisins directs de "A" avant de s'éloigner vers "D" ?

Partie C : Optimisation (Dynamique) On suppose maintenant que le graphe est orienté et sans cycle (chaque arête a un coût en minutes). On veut stocker le temps minimum pour atteindre chaque station depuis le dépôt.

Expliquez pourquoi l'utilisation de la programmation dynamique est pertinente pour calculer un chemin dans un graphe sans cycle.

Exercice 2 : Planification et Coloration (30 min - 6 points)

Un centre d'examen doit organiser des épreuves pour 6 matières :
Mathématiques (M), NSI (N), Physique (P), Chimie (C), SVT (S) et Philosophie (Ph).
Deux matières ne peuvent pas avoir lieu en même temps si au moins un élève suit ces deux spécialités. Les incompatibilités sont les suivantes :

- M est incompatible avec N, P et Ph.
- N est incompatible avec M, P et C.
- P est incompatible avec M, N, C et S.
- C est incompatible avec N, P et S.
- S est incompatible avec P et C.
- Ph est incompatible avec M.

1. Représenter cette situation par un graphe où les sommets sont les matières.
2. Donner la matrice d'adjacence de ce graphe (en suivant l'ordre alphabétique : C, M, N, P, Ph, S). (à la place de True et False utiliser 1 et 0).
3. On souhaite minimiser le nombre de créneaux horaires. En utilisant un algorithme de coloration glouton (en traitant les sommets par ordre de degré décroissant), déterminez le nombre chromatique (nombre minimal de couleurs pour colorier, sans que deux sommets connectés aient la même couleur) de ce graphe.
4. Quel est le nombre minimal de créneaux nécessaires ? Donnez une répartition possible.

Corrigé du Sujet 1 : Devoir d'entraînement

Exercice 1 : Réseau de Transport

1. **Méthode degre :**

```
def degre(self, u):
   return len(self.adj[u])
```

2. **Instructions :**

```
g = Graphe(["A", "B", "C", "D"])
g.ajouter_arete("A", "B")
g.ajouter_arete("B", "C")
g.ajouter_arete("B", "D")
```

3. La ligne à ajouter est f.enfiler(v).

4. a) Après "A", la file contient **["B", "C"]**.
4. b) C'est **"B"** qui sort (premier entré, premier sorti).
4. c) Oui. Comme on utilise une File (FIFO), tous les voisins directs de "A" sont enfilés et sortiront **avant** les voisins de ces voisins (comme "D"). On explore par "distance" croissante.

5. Pourquoi la programmation dynamique est-elle pertinente ici ?

Dans un graphe orienté acyclique (DAG), pour calculer le temps minimum (ou le nombre de chemins), on se rend compte qu'on repasse souvent par les mêmes sommets.

- **Problème du calcul récursif simple :** Si une station S est accessible par plusieurs chemins, un algorithme récursif classique va recalculer le temps minimum depuis S à chaque fois qu'il la rencontre. Cela entraîne une explosion du temps de calcul (complexité exponentielle).
- **Solution Dynamique :**
  En utilisant la **mémoïstication** (stockage des résultats dans un dictionnaire ou un tableau), on ne calcule le temps minimum depuis la station S qu'**une seule fois**. Les appels suivants se contentent de lire la valeur en mémoire. On passe ainsi d'une complexité exponentielle à une complexité linéaire O(S+A) où S est le nombre de sommets et A le nombre d'arêtes.

Exercice 2 : Planification et Coloration

1. **Le Graphe :** Le graphe possède 6 sommets. Les arêtes relient les matières incompatibles.
2. **Matrice d'adjacence (Ordre : C, M, N, P, Ph, S) :**
$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$
3. **Coloration (Welsh-Powell) :**
   - Calcul des degrés : P(4),C(3),M(3),N(3),S(2),Ph(1).
   - Couleur 1 : **P**, puis **Ph** (non adjacent à P).
   - Couleur 2 : **C**, puis **M** (non adjacent à C).
   - Couleur 3 : **N**, puis **S** (non adjacent à N).
4. **Conclusion :** Le nombre chromatique est 3. Il faut 3 créneaux : {P, Ph}, {C, M} et {N, S}.