Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall est un algorithme permettant de déterminer un plus court chemin entre deux sommets d'un graphe orienté, pondéré mais dont les poids peuvent être négatifs (dans le cas où certains poids sont négatifs, l'algorithme de Dijkstra ne fonctionne pas).

On supposera que le graphe ne comporte pas de cycle de poids négatif : s'il existe un cycle de poids négatif dans le graphe, il n'y aura pas de plus court chemin entre certains sommets puisque en effectuant ce cycle on diminuera la longueur du chemin.

1. Sous-structure optimale

Un plus court chemin allant d'un sommet `s1` à un sommet `s2` qui passe par l'intermédiaire d'un sommet `s` est la concaténation d'un plus court chemin allant de `s1` à `s` et d'un chemin plus court chemin allant de `s` à `s2`.

L'idée de l'algorithme de Floyd-Warshall est, pour déterminer un plus court chemin de `s1` à `s2`, de commencer par limiter les étapes intermédiaires de ce chemin à certains des sommets du graphe : si on suppose que les sommets du graphes sont numérotés de 0 à $n-1$, on va commencer à déterminer un plus court chemin de `s1` à `s2` dont toutes les étapes intermédiaires sont dans l'ensemble $G_k = \{0, \dots, k-1\}$, même si les sommets `s1` et `s2` ne sont pas dans G_k :

— Pour $k = 0$, G_0 est vide. Il existera donc un chemin entre `s1` et `s2` qui passe par les sommets de G_0 si et seulement si `s1` est directement relié à `s2` par une arrête (dans ce sens car le graphe est orienté)

— Si on suppose avoir déterminer les plus courts chemins (s'ils existent) entre tous les couples de sommets de `G` qui passent uniquement par l'intermédiaire des sommets de G_k, un plus court chemin qui va de `s1` à `s2` passant par les sommets de G_{k+1} est

- soit un chemin qui ne passe que par l'intermédiaire de G_k (et donc qui ne passe pas par le sommet `k`)
- soit un chemin qui passe une seule fois par le sommets `k` (il ne peut pas y passer deux fois car sinon il contient un cycle dont le poids serait alors positif). Dans ce cas le chemin est constitué par une première partie allant de `s1` à `k` et d'une seconde partie allant de `k` à `s2`, chacune de ces deux parties ne passant que par l'intermédiaire des sommets de G_k.

On peut donc définir une suite de matrices $(D_k)_{0 \leqslant k \leqslant n} \in \mathcal{M}_n(\mathbb{R})$ telle que, si on note, pour $(i,j) \in [\![0, n-1]\!]^2$, $d_{i,j}^{(k)}$ les coefficients de D_k qui représentent la longueur minimale d'un chemin allant du sommet i au sommet j en ne passant que par G_k. On a alors

$$d_{i,j}^{(0)} = \begin{cases} p_{i,j} & \text{s'il existe une arrête de poids } p_{i,j} \text{ allant de } i \text{ à } j \\ +\infty & \text{sinon} \end{cases}$$

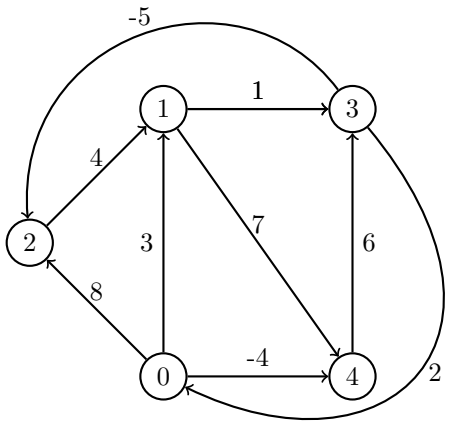
$$\text{pour } k \geqslant 1, \quad d_{i,j}^{(k)} = \min\left\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right\}$$

La longueur d'un plus court chemin allant du sommet i au sommet j, en empruntant éventuellement tous les autres sommets du graphe, est donc $d_{i,j}^{(n)}$.
On peut remarquer que D_0 est en fait la matrice d'adjacence du graphe G.

2. Implémentation en Python

On suppose que le graphe est implémenté par ses listes d'adjacence : `G` est un dictionnaire dont les clés sont les sommets et la valeur associé à un sommet `s` est une liste de couples `(s',p)` où `s'` est un sommet lié à `s` (dans ce sens) et `p` le poids de l'arrête allant de `s` à `s'`.
Par exemple, si on considère le graphe suivant, le dictionnaire `G` est

```
G = {0:[(1,3),(2,8),(4,-4)],1:[(3,1),(4,7)],2:[(1,4)],3:[(0,2),(2,-5)
    ],4:[(3,6)]}
```



On peut alors définir une fonction `tableauFW(G:dict)->list` qui détermine la matrice D_n, représentée par une liste de listes. Comme seule la matrice D_{k-1} est nécessaire pour déterminer D_k, on peut se contenter d'utiliser une seule variable `D` qui contient initialement D_0 et que l'on fait évoluer pour contenir les matrices D_k successives (plutôt que de stocker toutes les matrices D_k)

```
def tableauFW(G) :
    n = len(G)
    D = [[float('inf') for j in range(n)] for i in range(n)]
    # initialisation : calcul de D_0
    for i in G :
        D[i][i] = 0
        for (j,p) in G[i] :
            D[i][j] = p
    for k in range(n) :
        # calcul de D_{k+1} à partir de D_k
        for i in range(n) :
            for j in range(n) :
                if D[i][k]+D[k][j] < D[i][j] :
                    D[i][j] = D[i][k]+D[k][j]
                # dans le cas contraire d_{i,j}^{(k+1)} = d_{i,j}^{(k)} donc rien à faire
    return D
```

Dans l'exemple précédent, la fonction `tableauFW` renvoie la matrice

$$\begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

On peut remarquer que comme le graphe de cet exemple est connexe, il existe forcément un chemin entre tous les couples de sommets donc aucun des coefficients de la matrice finale ne vaut $+\infty$ (mais c'était le cas dans certaines des matrices précédentes).
La longueur d'un plus court chemin allant de `0` à `2` est alors $d_{0,2}^{(5)} = -3$.
Pour obtenir la distance du plus court chemin allant de `s1` à `s2` :

```
def FW(G,s1,s2) :
    return tableauFW(G)[s1][s2]
```

La complexité de cette fonction est $O\left(n^3\right)$.

3. Détermination d'un plus court chemin

On va maintenant déterminer les étapes intermédiaires d'un plus cours chemin allant de `s1` à `s2`.
Pour cela, on va introduire une autre suite de tableaux $A_k, 0 \leqslant k \leqslant n$ carrés de taille n pour laquelle le coefficient $a_{i,j}^{(k)}$ contient le numéro du sommet dont on provient juste avant d'arriver au sommet j, dans un chemin de longueur minimale allant de i à j et ne passant que par l'intermédiaire des sommets des de G_k : $a_{i,j}^{(k)}$ sera le numéro du dernier sommet avant j dans un chemin de longueur $d_{i,j}^{(k)}$ allant de i à j. Il suffit alors de faire évoluer ce tableau en même temps que le tableau des distances :

si $d_{i,j}^{(k)} = d_{i,j}^{(k-1)}$, on ne change rien

si $d_{i,j}^{(k)} = d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$, on vient de trouver un chemin plus court allant de i à j dont le dernier sommet avant j est $a_{k,j}^{(k-1)}$

À partir du tableau A_n, il suffit de reconstruire le chemin allant de `s1` à `s2` en partant du sommet `s2` et en remontant jusqu'au sommet `s1`
Dans notre exemple, les deux tableaux D_5 et A_5 sont

$$D_5 = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \text{et} \quad A_5 = \begin{pmatrix} & 2 & 3 & 4 & 0 \\ 3 & & 3 & 1 & 0 \\ 3 & 2 & & 1 & 0 \\ 3 & 2 & 3 & & 0 \\ 3 & 2 & 3 & 4 & \end{pmatrix}$$

Les coefficients diagonaux de A_5 restent vides car il n'y a pas d'intermédiaire dans un plus court chemin allant de i à lui même (toujours parce que le graphe ne contient pas de cycle de poids négatif).
Un plus court chemin allant de `0` à `2` est de longueur $d_{0,2}^{(5)} = -3$; on arrive en `2` de puis $a_{0,2}^{(5)} = 3$ auquel on arrive depuis $a_{0,3}^{(5)} = 4$ auquel on arrive depuis $a_{0,4}^{(5)} = 0$ donc un chemin de longueur -3 allant de `0` à `2` est `[0,4,3,2]`.

Reste à coder cette fonction :

```
def cheminFW(G,s1,s2) :
    n = len(G)
    Distances = [[float('inf') for j in range(n)] for i in range(n)]
    Origines = [[None for j in range(n)] for i in range(n)] # None quand il n'
        y a pas de chemin de i à j
    # initialisation : calcul de D_0 et A_0
    for i in G :
        Distances[i][i] = 0
        for (j,p) in G[i] :
            Distances[i][j] = p
            Origines[i][j] = i  # chemin direct depuis i
    for k in range(n) :
        # calcul de D_{k+1} à partir de D_k et de A_{k+1} à partir de A_k
        for i in range(n) :
            for j in range(n) :
                if Distances[i][k]+Distances[k][j] < Distances[i][j] :
                    # nouveau chemin
                    Distances[i][j] = Distances[i][k]+Distances[k][j]
                    Origines[i][j] = Origines[k][j]
                # toujours rien à faire sinon
    # on reconstruit le chemin
    L = [s2]
    while s2 != s1 :
        s2 = Origines[s1][s2]
        L.append(s2)
    return L[::-1]  # et on le remet à l'endroit
```

La complexité de cette fonction est à nouveau $O(n^3)$.