Plus courts-chemins dans un graphe : Algorithme Floyd-Warshall

On considère un graphe orienté pondéré $G = (S, A, \omega)$, où S est l'ensemble des sommets, A l'ensemble (x, y) des arêtes $x \to y$ et ω la fonction donnant le poids de chaque arête.
On a ainsi $A \subset S \times S$ et $\omega : A \to \mathbb{R}$. On note $n = \operatorname{card} S$ et $m = \operatorname{card} A$.

Un chemin Γ est une suite d'arêtes $s_0 \to s_1 \to ... \to s_p$
Son poids $\pi(\Gamma)$ est la somme des poids de ses arêtes, c'est-à-dire $\pi(\Gamma) = \sum_{k=1}^{p} \omega(s_{k-1}, s_k)$.

La distance entre deux sommets est définie par le poids minimal d'un chemin, s'il en existe au moins un, entre les deux sommets, et $+\infty$ sinon. L'algorithme de Floyd-Warshall calcule, pour **chaque** paire de sommets la distance entre les deux sommets.

On suppose dans tout le sujet que le graphe n'a aucun cycle de poids négatif :
Autrement dit, pour tout cycle $\Gamma : s_0 \to s_1 \to ... \to s_p = s_0$, on a $\pi(\Gamma) = \sum_{k=1}^{p} \omega(s_{k-1}, s_k) \geq 0$

1) Justifier (en donnant les arguments essentiels) que la distance est bien définie. *Indication* : Il s'agit de prouver que s'il existe un chemin de s_0 à s_p, il en existe un de poids minimal. Faire intervenir l'ensemble des **chemins réduits** de s à t, c'est-à-dire les chemins ne passant pas deux fois par un même sommet.

2) Justifier que la propriété de sous-optimalité est vérifiée : si $s_0 \to s_1 \to ... \to s_{p-1} \to s_p$ est un plus court-chemin Γ allant de s_0 à s_p, alors pour tous $0 \leq i \leq j \leq p$, le sous-chemin $\Gamma' : s_i \to s_{i+1} \to ... \to s_{j-1} \to s_j$ est optimal, c'est-à-dire est un plus court-chemin allant de s_i à s_j.

Dans la suite, on suppose les sommets du graphe numérotés de 0 **à** $n-1$, c'est-à-dire $S = \{0, 1, 2, ..., n-1\}$, et on suppose le graphe défini par la liste G des listes d'adjacences pondérées : autrement dit, pour tout $0 \leq i < n$, `G[i]` est la liste des couples $(j, \omega(i, j))$, où j décrit l'ensemble des successeurs de i.

Par exemple, `G = [ [(0,0),(1,0.5),(2,0.75)] , [(1,0)] , [(1,1.5),(2,0)] ]`
Remarque : On considère que pour tout i, l'arête $i \to i$ existe et est de poids $\omega(i, i) = 0$.

Pour parcourir la liste $G[i]$, on pourra utiliser la syntaxe : `for (j,w) in G[i] : ...`

3) Pour tout $0 \leq k \leq n$, on note $D_k[i][j]$ la longueur du plus court chemin entre les sommets i et j parmi les chemins **dont tous les sommets intermédiaires appartiennent à** $\{0, 1, ..., k-1\}$.

En particulier, on a $D_0[i][j] = \omega(i, j)$ si $(i, j) \in A$ et $+\infty$ sinon.

Pour $0 \leq k < n$ exprimer $D_{k+1}[i][j]$ en fonction de $D_k[i][j]$, $D_k[i][k]$ et $D_k[k][j]$.

Justifier brièvement votre réponse.

Remarque : En Python, `float("inf")` code $+\infty$.
Par exemple, `1 + float("inf")` renvoie `float("inf")` et `min(2,float("inf"))` renvoie `2`.

4) Écrire une fonction `distances(G)` qui prend un graphe représenté par la liste `G` des listes d'adjacences et renvoie la matrice D des distances entre les sommets.

On utilisera ici une méthode itérative avec une boucle `for` sur k.

Remarque : La matrice `D` est codée par une liste de listes : `D[i][j]` est la distance de i à j.

Elle peut être initialement construite par `[ [math.inf for i in range(n)] for j in range(n)]`

5) Préciser la complexité de cet algorithme.

6) Écrire une fonction récursive `chemin(D,G,i,j)` qui étant donnée la matrice des distances `D` obtenue au 4) renvoie en temps $O(n^2)$ un plus court-chemin (codé par la liste des sommets parcourus) reliant i à j.

Remarque : On suppose ici que `D[i][j]` n'est pas égal à $+\infty$, c'est-à-dire qu'un tel chemin existe.

Indication : Noter que $i \to k \to ... \to j$ est un plus court-chemin reliant i à j si et seulement si une certaine relation est vérifiée entre `D[i][j]`et `D[k][j]`.

7) [*Question supplémentaire*] Écrire, en utilisant une approche descendante ("du haut vers le bas"), une variante de la fonction `distances(G)`, **de même complexité** que celle de la question 4).

Remarque : Utiliser une fonction auxiliaire qui modifie `D`.

Corrigé

1) Comme il n'y a pas de cycle de poids négatif, on peut se restreindre aux chemins réduits (car supprimer des boucles diminue le poids, puisque tout cycle est de poids positif). Les chemins réduits de s à t sont en nombre *fini*, donc il en existe de poids minimal (s'il existe au moins un chemin). Donc la distance est bien définie.

2) Supposons par l'absurde qu'il existe un chemin $s_i \to t_1 \to ... \to t_{r-1} \to s_j$ de poids strictement inférieur au poids de $s_i \to s_{i+1} \to ... \to s_{j-1} \to s_j$.
Alors le poids de $s_0 \to s_1 \to ... \to s_i \to t_1 \to ... \to t_{r-1} \to s_j \to ... \to s_p$ est strictement inférieur à celui de $s_0 \to s_1 \to ... \to s_{p-1} \to s_p$, d'où une contradiction.

3) Notons $\Omega_{s,t}^{(k)}$ l'ensemble des chemins injectifs dont les sommets intermédiaires appartiennent à $\{0, 1, ..., k-1.\}$ reliant le sommet s au sommet t.
Un chemin de $\Omega_{s,t}^{(k+1)}$ appartient à $\Omega_{s,t}^{(k-1)}$ ou bien est le concaténé d'un chemin de $\Omega_{s,k}^{(k)}$ et d'un chemin de $\Omega_{k,t}^{(k)}$ (en considérant la position unique où le chemin passe par le sommet k). On a donc :

$$D_k[s,t] = \min(D_{k-1}[s,t], D_{k-1}[s,k] + D_{k-1}[k,t])$$

4) On obtient les distances en calculant $D_n[s,t]$

```
def distances(G) :
    n = len(G) ; D = [ [float("inf") for i in range(n)] for j in range(n)]
    for i in G :
        for (i,w) in G[i] :  D[i,j] = w
    for k in range(n) :
        Dex = [ L.copy() for L in D ]  # faire une copie (profonde), sinon problème de mise à jour
        for i in range(n) :
            for j in range(n) :
                D[i,j] = min(Dex[i,j],Dex[i,k]+Dex[k,j])
    return D
```

5) La complexité est en $O(n^3)$ compte tenu des trois boucles `for` imbriquées.

6) L'idée est de trouver, lorsque $i \neq k$, le successeur k de i dans le (un) chemin optimal, puis d'itérer le procédé. Le sommet k se caractérise par la propriété $D[i,j] = \omega(i,k) + D[k,j]$.

```
def chemin(D,G,i,j) :
    if i == j :  return [i]
    for (k,w) in G[i] :
        if D[i,j] == w + D[k,j] :  return [i] + chemin(D,G,k,j)
    assert False   # normalement, cette instruction n'est jamais effectuée.
```

7)
```
def distances(G) :
    def aux(k) :
        if k == 0 :
            for i in range(n) :
                for (j,w) in G[i] :  D[i,j] = w
        else :
            aux(k-1) ; Dex = [ L.copy() for L in D ]
            for i in range(n) :
                for j in range(n) :
                    D[i,j] = min(Dex[i,j],Dex[i,k]+Dex[k,j])
    n = len(G) ; D = [ [math.inf for i in range(n)] for j in range(n)]
    aux(n) ; return D
```

Remarque : `aux` est une procédure qui modifie le tableau `D`