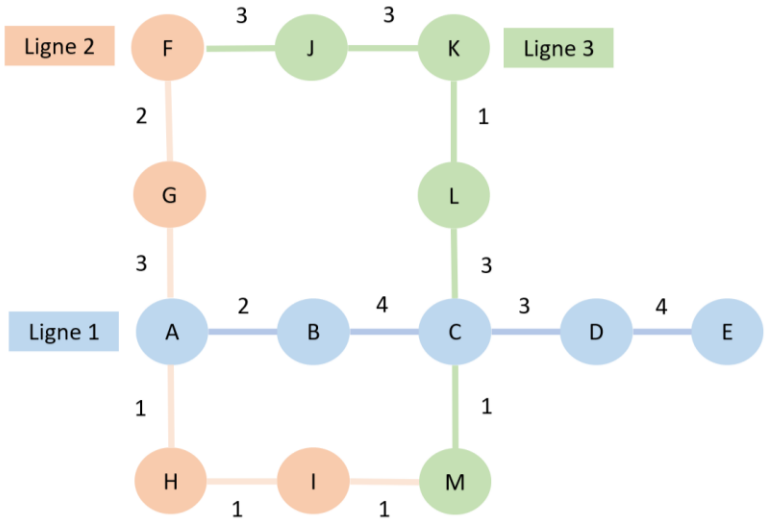
Informatique

11
Dictionnaires et programmation dynamique

TD 2-8
Distances dans un graphe - Floyd-Warshall

Contexte

Voici un plan de métro :



Le réseau est composé de trois lignes. Les stations sont nommées par des lettres afin de faciliter l'étude. Les temps de parcours en minutes sont indiqués sur les branches de chaque ligne entre chaque station. Nous allons programmer l'algorithme de Dijkstra afin de déterminer le plus court chemin sur ce réseau d'une gare de départ à une gare d'arrivée, par exemple dans notre cas, de F à C.

Floyd-Warshall

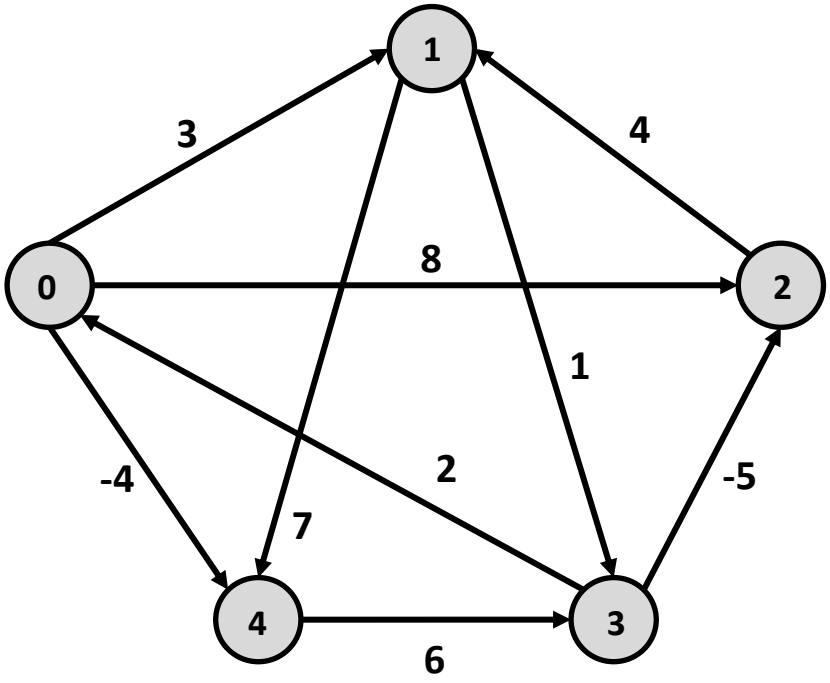
Nous avons en première année traité cet exercice à l'aide de l'algorithme de Dijkstra, permettant de trouver toutes les distances à partir d'une seule station « A » vers toutes les autres, nous cherchions le chemin vers « C ». Il était alors possible de faire tourner l'algorithme depuis chaque station du réseau pour obtenir l'ensemble des distances entre deux stations quelconques.

C'est alors que l'algorithme de Floyd-Warshall apparaît comme une solution qui va donner toutes ces distances. Cet algorithme permet par ailleurs de traiter des graphes dont les arcs ont des poids négatifs, mais le graphe ne doit pas présenter de cycles de poids total négatif. C'est-à-dire qu'en parcourant un cycle, la somme des poids ne doit pas être inférieure à 0.

Nous décrirons l'algorithme au cours de ce sujet.

Graphe support

Pour nous aider à comprendre l'algorithme mis en place, nous prendrons comme support le graphe orienté suivant :



Format array

Nous pourrions travailler avec des listes de listes, mais les array sont parfaitement utilisable pour représenter des matrices, en connaissant les quelques subtilités associés :

- Contrairement aux listes de listes, les array ne sont pas modifiables en dimensions
- Accès à une valeur : `M[i,j]`
- Initialisation : Créer une liste de listes puis la transformer en array avec `np.array(G)`
- Récupération des nombres de lignes et colonnes : `nl,nc = W.shape[0:2]`
- Copie : `W.copy()`
- Créer un array rempli de 0 de dimension n*n : `np.zeros([n,n])`

Par ailleurs, on utilisera `inf` dans le module `numpy` afin de représenter le nombre infini.

Exercice 1: Algorithme basé sur des matrices

Initialisation

La matrice d'adjacence G est définie comme suit : Du sommet i vers le sommet j :

- $G[i,j] = d_{ij}$ si l'arc orienté de i vers j existe avec un poids d_{ij}
- $G[i,j] = 0$ si $i = j$
- $G[i,j] = \infty$ si l'arc orienté de i vers j n'existe pas

Question 1: Ecrire les lignes de codes créant la matrice d'adjacence du graphe support

On initialise une matrice des intermédiaires I^0 ainsi :

- $I[i,j] = i$ si un arc existe de i vers j
- $I[i,j] = \infty$ si $G[i,j] = 0$ ou $G[i,j] = \infty$

Pour le moment, $I[i,j]$ indique s'il existe un chemin du sommet i vers le sommet j, et si oui ($I[i,j] \neq \infty$), donne le sommet de départ i.

Question 2: Créer la fonction inter(G) renvoyant la matrice I

```
>>> inter(G)
array([[inf,  0.,  0., inf,  0.],
       [inf, inf, inf,  1.,  1.],
       [inf,  2., inf, inf, inf],
       [ 3., inf,  3., inf, inf],
       [inf, inf, inf,  4., inf]])
```

Floyd-Warshall

Soit l'ensemble $E = \{0,1,\dots,n-1\} = \{e_1, e_2, \dots, e_n\}$ des sommets du graphe représenté par la matrice d'adjacence G et W^k une matrice telle que $W^0 = G$. L'algorithme de Floyd-Warshall est un exemple de programmation dynamique qui répond au sous problème suivant :

Pour $k > 0$, quels sont les poids minimaux W_{ij}^k des chemins, s'ils existent (∞ sinon), de tous les sommets i à tous les sommets j du graphe n'empruntant que des sommets intermédiaires dans $\{e_1, e_2, \dots, e_k\}$

Vous pourrez vérifier qu'au départ ($k = 0$), W_{ij}^0 représente le poids minimal des chemins du sommet i au sommet j en empruntant aucun sommet intermédiaire s'il existe, ∞ sinon. Puis, à l'itération $k = n$, W_{ij}^n indique le poids minimal des chemins du sommet i au sommet j en empruntant des sommets intermédiaires dans l'ensemble des sommets du graphe, ce qui répond à l'objectif.

A chaque itération $k \in [1, n]$, l'algorithme va mettre à jour W^{k-1} vers W^k en déterminant, pour chaque chemin de i vers j, si le chemin $W_{i,k-1}^{k-1} + W_{k-1,j}^{k-1}$ passant par l'intermédiaire $e_k = k - 1$ est plus court que le chemin W_{ij}^{k-1} actuellement trouvé de i vers j.
Si oui, il met à jour :

- La matrice W^k indiquant le poids minimal du chemin de i à j à l'itération k :
$$W_{ij}^k = min\left(W_{ij}^{k-1}, W_{i,k-1}^{k-1} + W_{k-1,j}^{k-1}\right)$$
- La matrice I^k indiquant **un** intermédiaire dans le chemin de poids minimal de i vers j à l'itération k
$$I_{ij}^k = k - 1$$

L'algorithme s'arrêtera si un cycle de poids négatif est trouvé.
On donne l'extrait suivant dans le code élèves (LIEN):

```
def FW(G,I):
    W = G.copy()
    n = W.shape[0]
    for k in range(1,n+1):
        for i in range(n):
            for j in range(n):
                dp = d1 = #####
                d2 = #####
                d = min(d1,d2)
                if #####: # Si distance modifiée
                    W[i,j] = #####
                    I[i,j] = #####
                    if #####:
                        assert ##### >= 0,"Cycle de poids négatif"
    return W
```

Question 3: Compléter la fonction FW(G,I) réalisant les itérations de l'algorithme de Floyd-Warshall, modifiant I en place et renvoyant la matrice W

Vérifier :

```
>>> FW(G,I)
array([[ 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.]])
```

```
>>> I
array([[inf,  4.,  4.,  4.,  0.],
       [ 3., inf,  3.,  1.,  3.],
       [ 3.,  2., inf,  1.,  3.],
       [ 3.,  2.,  3., inf,  0.],
       [ 3.,  3.,  3.,  4., inf]])
```

Question 4: Estimer la complexité de cet algorithme

Remarque : A partir du moment où $k - 1$ est un intermédiaire du chemin **le plus court** de i vers j, I_{ij}^k ne changera plus jamais $\left(I_{ij}^k = I_{ij}^n\right)$. On a par ailleurs la distance du chemin le plus court $W_{ij}^n = W_{ij}^k = W_{i,k-1}^{k-1} + W_{k-1,j}^{k-1}$ et les distances $W_{i,k-1}^{k-1}$ et $W_{k-1,j}^{k-1}$ ne changeront plus non plus. Donc les intermédiaires des chemins de i à $k - 1$ et de $k - 1$ à j non plus. La remontée des chemins sera réalisée à l'aide de la succession des intermédiaires stockés dans cette matrice.

Remontée des chemins

Observez bien la matrice I^n :

```
>>> I
array([[inf,  4.,  4.,  4.,  0.],
       [ 3., inf,  3.,  1.,  3.],
       [ 3.,  2., inf,  1.,  3.],
       [ 3.,  2.,  3., inf,  0.],
       [ 3.,  3.,  3.,  4., inf]])
```

Traitons d'abord deux exemples :

- $I^n_{14} = 3$: Dans le chemin allant de 1 vers 4 (1304), il faudra passer par 3
- $I^n_{20} = 3$: Dans le chemin allant de 2 vers 0 (2130), il faudra passer par 3

La remontée de la matrice I ne donne pas l'ordre des sommets, mais donne l'un des sommets du chemin à remonter, voisin direct de i ou j, ou non !

Voici les étapes successives de la remontée du chemin :

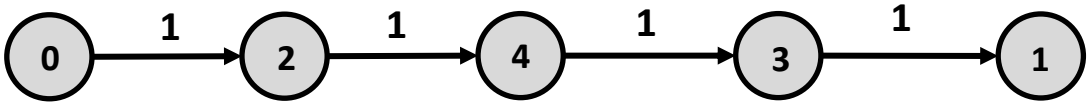
- 1304 de 1 vers 4 : 1→4=3, 3→4=0, 0→4=0

```
>>> I
array([[inf,  4.,  4.,  4.,  0.],
       [ 3., inf,  3.,  1.,  3.],
       [ 3.,  2., inf,  1.,  3.],
       [ 3.,  2.,  3., inf,  0.],
       [ 3.,  3.,  3.,  4., inf]])
```

- 2130 de 2 vers 0 : 2→0=3, 2→3=1, 2→1=2

```
>>> I
array([[inf,  4.,  4.,  4.,  0.],
       [ 3., inf,  3.,  1.,  3.],
       [ 3.,  2., inf,  1.,  3.],
       [ 3.,  2.,  3., inf,  0.],
       [ 3.,  3.,  3.,  4., inf]])
```

Dans ces exemples, on remarque que s est toujours un voisin direct de i ou de j. Mais ce n'est pas toujours vrai. Prenons l'exemple ci-dessous :



Question 5: En utilisant le travail précédent, créer GG, II et WW sur ce graphe et établir à la main les étapes de remontée du chemin de 0 vers 1

Question 6: Proposer une fonction chemin(i,j,G,I,W) renvoyant la liste des indices des sommets du chemin de i vers j

Remarques :

- On pourra créer une fonction récursive 😉
- Vérifiez votre fonction sur les deux graphes G et GG. Elle peut être fausse et donner de bons résultats sur G

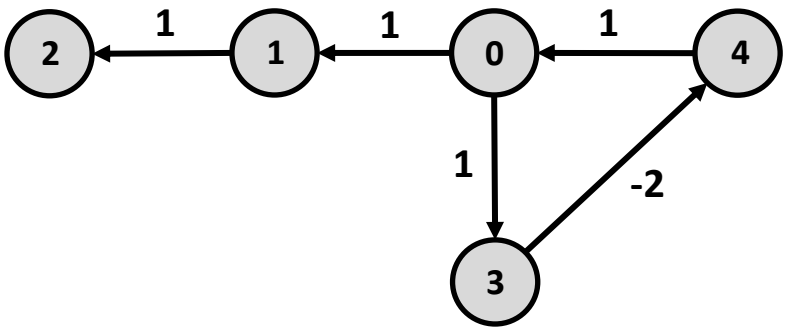
Question 7: Créer la fonction dc(i,j,G, I,W) affichant dans la console la distance et le chemin entre i et j

Vérifier :

```
>>> dc(2,0,G,I,W)
Distance 2->0: 7.0
[2, 1, 3, 0]

>>> dc(0,2,G,I,W)
Distance 0->2: -3.0
[0, 4, 3, 2]

>>> dc(1,4,G,I,W)
Distance 1->4: -1.0
[1, 3, 0, 4]

>>> dc(4,1,G,I,W)
Distance 4->1: 5.0
[4, 3, 2, 1]
```

Autour des cycles de poids nuls

Soit le graphe suivant contenant le cycle de poids nul 034:



Lorsque l'on remplace la condition « d != dp » de la fonction FW par « if d == d2 », on obtient :

```
>>> Wc                                  >>> Ic
array([[ 0.,  1.,  2.,  1., -1.], array([[4., 4., 4., 4., 4.],
       [inf,  0.,  1., inf, inf],        [4., 1., 2., 4., 4.],
       [inf, inf,  0., inf, inf],        [4., 4., 2., 4., 4.],
       [-1.,  0.,  1.,  0., -2.],        [4., 4., 4., 4., 4.],
       [ 1.,  2.,  3.,  2.,  0.]])       [4., 4., 4., 4., 4.]])
```

On remarque plusieurs choses dans cet exemple :

- La diagonale de Ic fait apparaître des valeurs non infinies. En effet :
  - De 1 à 1, la distance d1 dans Wc est nulle et la distance d2 passant par 1 l'est aussi. Donc le minimum de 0 et 0 est 0 (d=d2) et on stocke 1 comme intermédiaire. De même pour 2
  - A la dernière itération, l'algorithme trouve qu'il est possible d'aller de 0 à 0, 3 à 3 et 4 à 4 en passant par 4 sans que la distance (nulle) ne soit modifiée, du fait de la présence d'un cycle de poids nul
- Les chemins impossibles apparaissent puisque l'infini n'est pas modifié. Exemple de 2 à 1 à l'itération k=4 reste infini
- On voit apparaître 4 partout ailleurs (tous les chemins possibles), signe là aussi de la présence d'un cycle de poids nul. En effet, à la dernière itération (k=4), l'algorithme a trouvé que l'on pouvait aller à tous les sommets en passant par 4 avec la même distance que les résultats précédents, le test d=d2 étant vérifié à chaque fois, l'intermédiaire 4 est mis partout

Il n'est alors plus possible de remonter les chemins.

On retiendra donc qu'il faut veiller à ne changer la matrice I que si la distance dans W est modifiée 😉

Exercice 2: Algorithme avec dictionnaires et programmation dynamique

Programmation de l'algorithme itératif

Même si cette méthode n'apporte pas d'avantage en temps et consomme en mémoire avec les dictionnaires, voyons comment programmer Floyd-Warshall avec des dictionnaires. Avec les notations précédentes, on peut écrire :

$$\forall (i,j) \in [0,n-1]^2, \forall k \in [0,n], W_{ij}^k = \begin{cases} G_{ij} \; si \; k = 0 \\ min\begin{pmatrix} W_{ij}^{k-1} \\ W_{i,k-1}^{k-1} + W_{k-1,j}^{k-1} \end{pmatrix} \end{cases}$$

On souhaite donc réaliser l'algorithme de Floyd-Warshall à l'aide d'un dictionnaire en utilisant l'équation de Bellman suivante :

$$\forall (i,j) \in [0,n-1]^2, \forall k \in [0,n], D(i,j,k) = \begin{cases} G_{ij} \; si \; k = 0 \\ min\begin{pmatrix} D(i,j,k-1) \\ D(i,k-1,k-1) + D(k-1,j,k-1) \end{pmatrix} \end{cases}$$

Dans les prochaines questions, on travaille sur le dictionnaire D que l'on nommera dico_W.

Question 1: Proposer une fonction FW_dico(G) renvoyant le dictionnaire dico_W à l'aide d'un algorithme itératif ascendant

Question 2: Proposer une fonction dico2mat(dico,n,k) recréant la matrice W^k à l'aide du dictionnaire dico

Vérifier :

```
>>> dico2mat(dico_W,5,5)
array([[ 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.]])
```

Programmation de l'algorithme récursif

On souhaite maintenant réaliser l'algorithme de manière descendante par récursivité.

Question 3: Proposer une fonction FW_rec(G,i,j,k) renvoyant l'une des valeurs W_{ij}^k en programmation descendante récursive

Vérifier par exemple :

```
>>> FW_rec(G,0,3,5)
2.0
```

Question 4: Mettre en place la fonction FW_dico_rec(G) renvoyant le dictionnaire dico_W de manière récursive avec mémoïsation

Vérifier que vous obtenez le même dictionnaire, ou la même matrices W que précédemment.

Etapes intermédiaires

On souhaite maintenant remonter les chemins les plus courts entre deux sommets i et j quelconques du graphe. Il aurait été possible de créer un dictionnaire dico_I représentant la matrice I dans les fonctions FW_dico et FW_dico_rec (j'ai proposé ces fonctions dans le corrigé), mais nous allons voir comment reconstruire les solutions optimales avec le seul dictionnaire dico_W.
Nous allons dans un premier temps proposer une fonction recréant l'intégralité d'un chemin optimal de i vers j.
On donne l'extrait suivant dans le code élèves (LIEN):

```
def chemin_dico(D,i,j,k):
    if k==0:
        if D[(i,j,k)]==inf:
            return #####
        else:
            return #####
    else:
        if D[(i,j,k)] == D[(i,j,k-1)]:
            return chemin_dico(#####)
        else:
            L_i_km1 = chemin_dico(#####)
            L_km1_j = chemin_dico(#####)
            L_i_km1.pop()
            return L_i_km1 + L_km1_j
```

Précision : La distance la plus courte d'un sommet quelconques i et un sommet quelconque j est la distance à la fin des itérations $W_{i,j}^n$. Il faut donc remonter les chemins depuis n. Toutefois, si $k-1$ est un intermédiaire dans le chemin le plus court de i vers j, alors $W_{i,j}^k = W_{i,k-1}^{k-1} + W_{k-1,j}^{k-1}$ (itération k) est égal à $W_{i,j}^n$ (itération n). Les distances $W_{i,k-1}^{k-1}$ et $W_{k-1,j}^{k-1}$ ne peuvent être plus faibles et sont les distances minimales $W_{i,k-1}^n$ et $W_{k-1,j}^n$ quand $k = n$. Pour trouver les chemins les plus courts de i à $k-1$, et de $k-1$ à j, on peut donc repartir de l'itération $k-1$.

Question 5: Compléter la fonction récursive chemin_dico(D,i,j,k) prenant en argument le dictionnaire D (dico_W), les sommets de départ i et d'arrivée j et un indice k, et renvoyant le chemin optimal entre i et j

Vérifier les chemins du graphe support et sur le graphe de l'exemple avec cycle nulle Gc :

```
>>> chemin_dico(dico_Wc,2,1,5)
[]
```

```
>>> chemin_dico(dico_W,0,0,5)
[0, 0]

>>> chemin_dico(dico_W,2,0,5)
[2, 1, 3, 0]

>>> chemin_dico(dico_W,0,2,5)
[0, 4, 3, 2]

>>> chemin_dico(dico_W,1,4,5)
[1, 3, 0, 4]

>>> chemin_dico(dico_W,4,1,5)
[4, 3, 2, 1]
```

La procédure proposée ne remontant les chemins que si un changement des distances a eu lieu, elle ne tient pas compte des cycles de poids nuls.

La remontée de tous les chemins par la fonction chemin_dico est comparable à la remontée chemin avec utilisation de la matrice I^n abordée dans la première partie de ce sujet, à la petite différence qu'elle réalise $O(n)$ itérations pour chaque chemin au lieu d'être en $O(n_s)$ où n_s est le nombre de sommets du chemin.

Quelle que soit la procédure choisie, et après l'étude du graphe en $O(n^3)$:

- Si on veut trouver un chemin au cas par cas, le coût sera en $O(n)$ dans le pire des cas
- Si l'on veut stocker les n^2 chemins possibles d'un coup, la procédure coutera $O(n^3)$ avec une petite amélioration possible par mémoïsation (que nous ne ferons pas) sur les intermédiaires trouvés

Voyons donc finalement comment reconstruire la matrice I^n afin de permettre une remontée au cas par cas la plus optimale possible.

Question 6: Proposer une fonction récursive case_I(D,i,j,k) renvoyant la valeur de la case I_{ij}^n à l'aide du dictionnaire D=dico_W

Vérifier par exemple :

```
>>> case_I(dico_W,0,0,5)
inf

>>> case_I(dico_W,1,2,5)
3
```

Question 7: Proposer une fonction inter_dico(DW,n) prenant en argument le dictionnaire DW=dico_W et le nombre de sommets n, et créant le dictionnaire DI associé à la matrice I en utilisant une fonction récursive rec(D,i,j,k) basée sur la fonction case_I

Remarque : on pourra mettre un indice n en troisième argument des tuples (i,j,n) clés du dictionnaire DI afin de pouvoir réutiliser la fiction dico2mat.

Vérifier :

```
>>> dico2mat(dico_I,5,5)
array([[inf,  4.,  4.,  4.,  0.],
       [ 3., inf,  3.,  1.,  3.],
       [ 3.,  2., inf,  1.,  3.],
       [ 3.,  2.,  3., inf,  0.],
       [ 3.,  3.,  3.,  4., inf]])
```

Exercice 3: Application métro

On définit le réseau ainsi :

```
Ligne_1 = ['A','B','C','D','E']
Ligne_2 = ['F','G','A','H','I','M']
Ligne_3 = ['F','J','K','L','C','M']
Lignes = [Ligne_1,Ligne_2,Ligne_3]

Segments_1 = [[2,2],[4,4],[3,3],[4,4]]
Segments_2 = [[2,2],[3,3],[1,1],[1,1],[1,1]]
Segments_3 = [[3,3],[3,3],[1,1],[3,3],[1,1]]
Segments = [Segments_1,Segments_2,Segments_3]
```

On notera que les segments définis ci-dessus permettent de donner le temps de parcours dans les deux sens. Ainsi, bloquer le segment de B à C et de B à A revient à écrire :

```
Segments_1 = [[2,0],[0,4],[3,3],[4,4]]
```

A partir de cela, on propose les fonctions permettant de créer automatiquement

- La liste **Stations** des différentes stations du réseau
- Le nombre de stations **Nb**
- La matrice **Graphe** (array de numpy) des distances du réseau

Téléchargez le fichier en lien ici afin d'avoir le graphe à disposition. Vous le compléterez à la suite dans la partie réservée à l'application sur ce réseau.

Question 1: En utilisant le graphe « graphe » créé dans le code fourni, appliquer l'algorithme précédent puis trouver la distance minimum entre les stations A et C ainsi que le chemin le plus court réalisé