École polytechnique de Louvain

LINFO1114
-
Mathématiques discrètes
-
Projet

Auteurs :
Lai Loïc,
NOMA : 69712200
Felis Maxence,
NOMA : 30132301
Vanlippevelde Louis,
NOMA : 56862300
Groupe : 23

22 décembre 2024

1 Introduction

Dans le cadre du projet de mathématiques discrètes du cours LINFO1114, nous nous intéressons à l'étude des algorithmes de calcul des plus courts chemins dans un graphe. Les algorithmes étudiés sont ceux de Dijkstra, Bellman-Ford et Floyd-Warshall. Ils sont des outils fondamentaux en informatique et en recherche opérationnelle.

Le but de ce travail est de mettre en œuvre ces trois algorithmes afin de calculer la matrice des distances des plus courts chemins entre toutes les paires de nœuds d'un graphe pondéré, connecté, non-dirigé et non-négatifs. L'implémentation sera réalisée en Python, et nous utiliserons un graphe assigné sur Moodle pour tester et valider les résultats.

Ce projet se décompose en plusieurs étapes. Tout d'abord, une compréhension des algorithmes étudiés. Ensuite, leur implémentation rigoureuse et leur application sur des données à partir d'un graphe donné. En plus, nous calculerons la distance du chemin qui est le plus court entre une paire de noeud manuellement, sans les algorithmes python afin de vérifier la cohérence des résultats obtenus via les algorithmes.

Dans ce rapport, nous retrouverons un rappel théorique décrivant les trois algorithmes que l'on a codé (Dijkstra, Bellman-Ford et FloydWarshall) et leur équation. Mais également le calcul théorique et numérique (fait à la main) du plus court chemin entre le noeud A et le noeud J du graphe qui nous a été assigné sur Moodle via l'algorithme de Dijkstra afin de vérifier que nous obtenions les mêmes valeurs que notre algorithme.
Enfin, nous aurons le résultat de notre procédure main avec son impression de la matrice de coûts que nous avons définie d'après le graphe dont notre groupe a hérité et les trois matrices de distances (selon les algorithmes de Dijkstra, Bellman-Ford et FloydWarshall) avec quelques explications générales du code. Ainsi que le code complet et commenté en annexe.

2 Rappel théorique des algorithmes

2.1 Algorithme de Dijkstra

L'algorithme de Dijkstra est utilisé pour trouver les chemins les plus courts d'un nœud de base à tous les autres nœuds dans un graphe pondéré, à condition que les pondérations soient non négatives. L'algorithme change au fur et à mesure les distances estimées entre les nœuds. Pour chaque chemin, il regarde si c'est plus court que ce qu'on a trouvé jusqu'à présent dans quel cas la distance est mise à jour.

Principe :

1. Initialiser la distance du nœud source à 0 et celles des autres nœuds à l'infini (∞).
2. Maintenir un ensemble de nœuds visités.
3. À chaque itération, sélectionner le nœud ayant la plus petite distance actuelle parmi les nœuds non visités.
4. Mettre à jour les distances des noeuds voisins en appliquant l'équation :

$$d[v] = \min(d[v], d[u] + c[u, v])$$

où $d[u]$ est la distance du nœud u, $d[v]$ celle du nœud voisin v, et $c[u,v]$ le coût du lien entre u et v.

5. Répéter jusqu'à ce que tous les nœuds soient visités.

2.2 Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford est une méthode itérative pour trouver les plus courts chemins depuis un nœud source dans un graphe pondéré. Contrairement à Dijkstra, il permet de gérer des poids négatifs.

Principe :

1. Initialiser les distances : $d[source] = 0$ et $d[v] = \infty$ pour tous les autres nœuds v.
2. Répéter $n-1$ fois (où n est le nombre de nœuds du graphe) :
   — Pour chaque lien (u,v), appliquer l'équation :

$$d[v] = \min(d[v], d[u] + c[u,v])$$

3. Effectuer une dernière vérification. Si une distance est encore réduite, il existe une boucle à poids négatif.

2.3 Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall est une méthode qui permet de calculer directement les plus courts chemins entre toutes les paires de nœuds dans un graphe pondéré. L'algorithme s'applique donc qu'une seule fois pour tout le graphe contrairement aux deux prédécesseurs où l'algorithme devra être appliqué pour chaque noeud du graphe.

Principe :

1. Initialiser une matrice $D^{(0)}$, où $d_{ij}^{(0)} = c_{ij}$ (coût entre les nœuds i et j) ou $+\infty$ s'il n'y a pas de lien entre i et j.
2. Mettre $d_{ii}^{(0)} = 0$ pour tous les nœuds i.
3. Effectuer des mises à jour successives pour chaque nœud k :

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

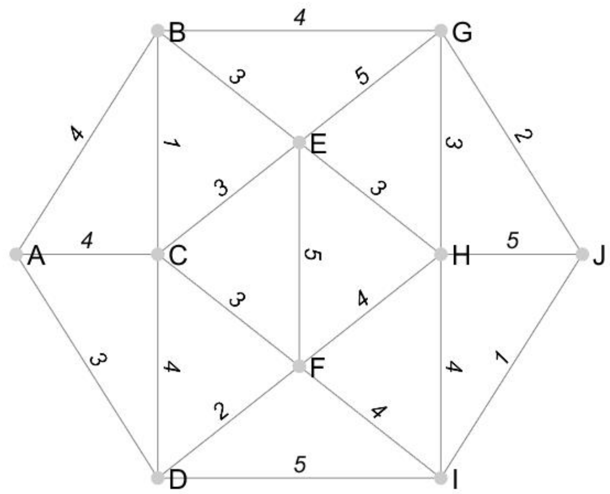
   où $d_{ij}^{(k)}$ est la plus courte distance entre i et j en utilisant seulement les nœuds 1 à k comme intermédiaires.
4. Répéter jusqu'à $k = n$, où n est le nombre de nœuds.

2.4 Comparaison des algorithmes

Algorithme	**Application principale**	**Complexité**	**Gestion des poids négatifs**
Dijkstra	Chemins à partir d'un seul nœud, coûts ≥ 0	$O((\lvert V\rvert + \lvert E\rvert) \cdot \log(\lvert V\rvert))$	Non
Bellman-Ford	Chemins à partir d'un seul nœud, coûts ≥ 0 ou < 0	$O(\lvert V\rvert \cdot \lvert E\rvert)$	Oui
Floyd-Warshall	Tous les plus courts chemins (paires de nœuds)	$O(\lvert V\rvert^3)$	Oui

3 Calcule numérique des algorithmes

3.1 Graphique étudié



3.2 Résolution de l'exercice via Dijkstra

Nous allons utiliser manuellement l'algorithme de Dijkstra pour trouver les plus courts chemins depuis le nœud a.
Pour ce faire, nous allons dessiner un tableau et mettre la valeur 0 à A et infini pour le reste.

Étape	**Nœud courant**	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[f]$	$d[g]$	$d[h]$	$d[i]$	$d[j]$
Initialisation	–	0.0	INF	INF	INF	INF	INF	INF	INF	INF	INF

Vu qu'il n'y a que le noeud A de disponible, on choisi lui comme prochain noeud courant. Pour chaque noeud par lequel nous ne sommes pas encore passé et connecté à ce dernier, on fait :

$$\text{Valeur du noeud courant actuel} + \text{coût pour aller aux noeuds voisin}$$

Si cette valeur est inférieur à la valeur déjà présente, nous le gardons. Ensuite nous prenons la plus petite valeur qui n'a pas encore été un noeud courant et il deviendra le prochain noeud courant.

Étape	**Nœud courant**	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[f]$	$d[g]$	$d[h]$	$d[i]$	$d[j]$
Initialisation	–	0.0	INF	INF	INF	INF	INF	INF	INF	INF	INF
1	A	**0.0**	4.0	4.0	3.0	INF	INF	INF	INF	INF	INF

Ici la plus petite valeur est 3 et il est associé au noeud D. Nous refaisons la même chose mais avec D comme nouveau point courant.

Étape	**Nœud courant**	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[f]$	$d[g]$	$d[h]$	$d[i]$	$d[j]$
Initialisation	–	0.0	INF	INF	INF	INF	INF	INF	INF	INF	INF
1	A	**0.0**	4.0	4.0	3.0	INF	INF	INF	INF	INF	INF
2	D	0.0	4.0	4.0	**3.0**	INF	5.0	INF	INF	8.0	INF

Nous répétons le même processus jusqu'à ce qu'il n'y ai plus de valeur infini sur la ligne.

Le tableau ci-dessous présente le résultat final, avec la mise à jour progressive des distances depuis A vers tous les autres nœuds.

Étape	**Nœud courant**	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[f]$	$d[g]$	$d[h]$	$d[i]$	$d[j]$
Initialisation	–	0.0	INF	INF	INF	INF	INF	INF	INF	INF	INF
1	A	**0.0**	4.0	4.0	3.0	INF	INF	INF	INF	INF	INF
2	D	0.0	4.0	4.0	**3.0**	INF	5.0	INF	INF	8.0	INF
3	B	0.0	**4.0**	4.0	3.0	7.0	5.0	8.0	INF	8.0	INF
4	C	0.0	4.0	**4.0**	3.0	7.0	5.0	8.0	INF	8.0	INF
5	F	0.0	4.0	4.0	3.0	7.0	**5.0**	8.0	9.0	8.0	INF
6	I	0.0	4.0	4.0	3.0	7.0	5.0	8.0	9.0	**8.0**	9.0
7	E	0.0	4.0	4.0	3.0	**7.0**	5.0	8.0	9.0	8.0	9.0

Le plus court chemin de A à J est trouvé avec une distance de 9. Ce chemin est :

$$A \to D \to I \to J$$

Pour trouver le chemin, nous partons tout simplement de J et on remonte le tableau en passant par les derniers noeuds courant jusqu'à arriver au noeud A.

Distances finales : $d[a] = 0,\ d[b] = 4,\ d[c] = 4,\ d[d] = 3, d[e] = 7,\ d[f] = 5,\ d[g] = 8,\ d[h] = 9,$ $d[i] = 8,\ d[j] = 9$

La distance obtenue par le calcul manuel est cohérente avec la valeur dans la matrice calculée par l'implémentation Python(voir 4.2).

4 Implémentation des algorithmes en python

4.1 Impression de la matrice coût

La matrice coût du graphe est :

$$\begin{bmatrix}
0 & 4 & 4 & 3 & \infty & \infty & \infty & \infty & \infty & \infty \\
4 & 0 & 1 & \infty & 3 & \infty & 4 & \infty & \infty & \infty \\
4 & 1 & 0 & 4 & 3 & 3 & \infty & \infty & \infty & \infty \\
3 & \infty & 4 & 0 & \infty & 2 & \infty & \infty & 5 & \infty \\
\infty & 3 & 3 & \infty & 0 & 5 & 5 & 3 & \infty & \infty \\
\infty & \infty & 3 & 2 & 5 & 0 & \infty & 4 & 4 & \infty \\
\infty & 4 & \infty & \infty & 5 & \infty & 0 & 3 & \infty & 2 \\
\infty & \infty & \infty & \infty & 3 & 4 & 3 & 0 & 4 & 5 \\
\infty & \infty & \infty & 5 & \infty & 4 & \infty & 4 & 0 & 1 \\
\infty & \infty & \infty & \infty & \infty & \infty & 2 & 5 & 1 & 0
\end{bmatrix}$$

4.2 Impressions des matrices des plus courts distances

4.2.1 Dijkstra

La matrice des distances des plus courts chemins en utilisant l'algo de Dijkstra :

$$\begin{bmatrix}
0 & 4 & 4 & 3 & 7 & 5 & 8 & 9 & 8 & 9 \\
4 & 0 & 1 & 5 & 3 & 4 & 4 & 6 & 7 & 6 \\
4 & 1 & 0 & 4 & 3 & 3 & 5 & 6 & 7 & 7 \\
3 & 5 & 4 & 0 & 7 & 2 & 8 & 6 & 5 & 6 \\
7 & 3 & 3 & 7 & 0 & 5 & 5 & 3 & 7 & 7 \\
5 & 4 & 3 & 2 & 5 & 0 & 7 & 4 & 4 & 5 \\
8 & 4 & 5 & 8 & 5 & 7 & 0 & 3 & 3 & 2 \\
9 & 6 & 6 & 6 & 3 & 4 & 3 & 0 & 4 & 5 \\
8 & 7 & 7 & 5 & 7 & 4 & 3 & 4 & 0 & 1 \\
9 & 6 & 7 & 6 & 7 & 5 & 2 & 5 & 1 & 0
\end{bmatrix}$$

4.2.2 Bellman-Ford

La matrice des distances des plus courts chemins en utilisant l'algo de Bellman-Ford :

$$\begin{bmatrix} 0 & 4 & 4 & 3 & 7 & 5 & 8 & 9 & 8 & 9 \\ 4 & 0 & 1 & 5 & 3 & 4 & 4 & 6 & 7 & 6 \\ 4 & 1 & 0 & 4 & 3 & 3 & 5 & 6 & 7 & 7 \\ 3 & 5 & 4 & 0 & 7 & 2 & 8 & 6 & 5 & 6 \\ 7 & 3 & 3 & 7 & 0 & 5 & 5 & 3 & 7 & 7 \\ 5 & 4 & 3 & 2 & 5 & 0 & 7 & 4 & 4 & 5 \\ 8 & 4 & 5 & 8 & 5 & 7 & 0 & 3 & 3 & 2 \\ 9 & 6 & 6 & 6 & 3 & 4 & 3 & 0 & 4 & 5 \\ 8 & 7 & 7 & 5 & 7 & 4 & 3 & 4 & 0 & 1 \\ 9 & 6 & 7 & 6 & 7 & 5 & 2 & 5 & 1 & 0 \end{bmatrix}$$

4.2.3 Floyd Warshall

La matrice des distances des plus courts chemins en utilisant l'algo de Floyd-Warshall :

$$\begin{bmatrix} 0 & 4 & 4 & 3 & 7 & 5 & 8 & 9 & 8 & 9 \\ 4 & 0 & 1 & 5 & 3 & 4 & 4 & 6 & 7 & 6 \\ 4 & 1 & 0 & 4 & 3 & 3 & 5 & 6 & 7 & 7 \\ 3 & 5 & 4 & 0 & 7 & 2 & 8 & 6 & 5 & 6 \\ 7 & 3 & 3 & 7 & 0 & 5 & 5 & 3 & 7 & 7 \\ 5 & 4 & 3 & 2 & 5 & 0 & 7 & 4 & 4 & 5 \\ 8 & 4 & 5 & 8 & 5 & 7 & 0 & 3 & 3 & 2 \\ 9 & 6 & 6 & 6 & 3 & 4 & 3 & 0 & 4 & 5 \\ 8 & 7 & 7 & 5 & 7 & 4 & 3 & 4 & 0 & 1 \\ 9 & 6 & 7 & 6 & 7 & 5 & 2 & 5 & 1 & 0 \end{bmatrix}$$

4.3 Explication

Pour l'implémentation de nos algorithmes, nous avons utilisé le module numpy de python. Ce dernier nous à permis de manipuler facilement nos données sous forme de matrices à l'aide de la structure "numpy.array".

Dans notre code, pour les liens inaccessible, nous avons utiliser la variable 'INF' = float('inf'). En python, float('inf') représente l'infini positif (∞) en tant que valeur flottante.

La matrice coût (4.1) est obtenu grâce au fichier main.py. En effet, il va lire un fichier .csv et ensuite organisé les données à l'intérieur sous forme de matrice numpy, stockée dans la variable nommée D.
Avec cela, nous pouvons l'utiliser comme paramètre de base pour nos fonctions d'algorithmes du plus court chemin par la suite.

Les trois matrices de distances (4.2) sont les résultats des fonctions dans shortest_path_algorithm.py. Il contient l'implémentation en python des trois algorithmes.
Ils prennent tous une matrice coût sous forme de matrice numpy en paramètre et retournent une matrice distance correspondant à la matrice coût. Étant donnée que dans ce projet nous étudions qu'un seul graphe, les trois fonctions prennent tous la même matrice en paramètre (D). Les trois matrices distances sont donc identiques.

L'algorithme de Dijkstra et Bellman-Ford peuvent seulement calculer les plus courts distances d'un noeud par rapport aux autres noeuds avec lui-même comme source. Or nous voulons toutes les distances entre toutes les pairs.
Nous avons donc créer une sous-fonctions pour chancun d'entre eux. Une boucle va appeler ces sous-fonctions qui permettent d'appliquer Dijkstra et Bellman-Ford pour chaque noeud comme source.

Pour plus de détail sur le code, consultez le code source en annexe.

5 Annexe

5.1 main.py

```
from shortest_path_algorithm import *

# Initiation des variables
INF = float('inf')
fichier_csv = 'graph.csv'

# Ouvrir et lire le fichier csv
with open(fichier_csv, 'r') as csv:
    # Lire les lignes
    lignes = csv.readlines()
    n = len(lignes)
    # Cr er une matrice vide de taille n x n
    D = np.empty((n, n), dtype=float)

    # Diviser chaque ligne en une liste avec ses valeurs
    for i in range(n):
        ligne = lignes[i].split(',')
        # Remplacer dans la matrice vide les valeurs correspondantes aux index
        for j in range(n):
            D[i][j] = ligne[j]

    # Fermer le csv
    csv.close()

mat_dk = Dijkstra(D)
mat_bf = Bellman_Ford(D)
mat_fw = Floyd_Warshall(D)

# V rifier si les 3 matrices sont  gaux
assert np.array_equal(mat_dk, mat_bf) and np.array_equal(mat_dk, mat_fw)
```

```
# Imprimer la matrice co t
print("La matrice co t du graphe est:\n" + str(D) + "\n")
# Imprimer les 3 matrices distances des plus courts chemins avec chaque algo
print("La matrice des distances des plus courts chemins en utilisant l'algo de Dijkstra:\n" + str(mat_dk) + "\n")
print("La matrice des distances des plus courts chemins en utilisant l'algo de Bellman-Ford:\n" + str(mat_bf) + "\n")
print("La matrice des distances des plus courts chemins en utilisant l'algo de Floyd Warshall:\n" + str(mat_fw) + "\n")
```

5.2 shortest_path_algorithm.py

```
import numpy as np

# Initiation de variable
INF = float('inf')

# Complexit  de O(n^3), peut  tre  am lior
def Dijkstra(C):
    """
    Calcule la matrice de distance des plus courts chemins entre toutes les paires de noeuds  d un  graphe en utilisant
    l'algorithme de Dijkstra.

    input: C --> une matrice numpy n x n de co t C
    output: Une matrice n   n contenant les distances des plus courts chemins
            entre toutes les paires de noeuds d'un graphe.
    """
    # Nombre de noeud
    n = len(C)
    # Cr ation d'une matrice de taille n x n vide
    D = np.empty((n, n), dtype=float)

    def chemins_plus_court(noeud):
        """
        Calcule les plus chemins le plus court d'un noeud

        Input: noeud --> un int qui repr sente l'index un noeud du graphe
        Output: Une liste de int avec la valeur des plus courtes distances du noeud
        """
        # Initialisation des distances   l'infini
        dist = [INF] * n
        # Distance au noeud lui-m me = 0
        dist[noeud] = 0
        # Garde une trace des noeuds visit s
        visited = [False] * n

        # It rer sur tous les noeuds
        for _ in range(n):
            # Trouver le noeud non visit  avec la plus petite distance
            min_distance = INF
            min_node = -1
            for i in range(n):
                if not visited[i] and dist[i] < min_distance:
                    min_distance = dist[i]
                    min_node = i

            # Si aucun noeud n'est accessible, arr ter
            if min_node == -1:
                break

            # Marquer ce noeud comme visit
            visited[min_node] = True
```

```
            # Mettre   jour les distances pour les voisins
            for voisin in range(n):
                if C[min_node][voisin] != INF and not visited[voisin]:
                    new_dist = dist[min_node] + C[min_node][voisin]
                    if new_dist < dist[voisin]:
                        dist[voisin] = new_dist

        return dist

    # Appliquer Dijkstra pour chaque noeud comme source
    for i in range(n):
        D[i] = chemins_plus_court(i)

    return D

def Bellman_Ford(C):
    """
    Calcule la matrice de distance des plus courts chemins entre toutes les paires de noeuds  d un  graphe en utilisant
    l'algorithme de Bellman-Ford.

    input: C --> une matrice numpy n x n de co t C
    output: Une matrice n   n contenant les distances des plus courts chemins
            entre toutes les paires de noeuds d'un graphe.
    """

    # Nombre de noeuds
    n = len(C)
    # Cr ation d'une matrice de taille n x n vide
    D = np.empty((n, n), dtype=float)

    def chemin_plus_court(noeud):
        """
        Calcule les plus chemins le plus court d'un noeud

        Input: noeud --> un int qui repr sente l'index un noeud du graphe
        Output: Une liste de int avec la valeur des plus courtes distances du noeud
        """
        # Initialisation des distances   l'infini
        dist = [INF] * n
        # Distance au noeud lui-m me = 0
        dist[noeud] = 0

        # Appliquer |V| - 1 relaxations pour toutes les ar tes
        for _ in range(n - 1):
            for i in range(n):
                for j in range(n):
                    if C[i, j] != INF and dist[i] + C[i, j] < dist[j]:
                        dist[j] = dist[i] + C[i, j]
        return dist

    # Appliquer Bellman-Ford pour chaque noeud comme source
    for i in range(n):
        D[i] = chemin_plus_court(i)

    return D

def Floyd_Warshall(C):
    """
        Calcule la matrice de distance des plus courts chemins entre toutes les paires de noeuds  d un  graphe en utilisant
        l'algorithme de Floyd-Warshall.

        input: C --> une matrice numpy n x n de co t C
        output: Une matrice n   n contenant les distances des plus courts chemins
```

```
                 entre toutes les paires de noeuds d'un graphe.
    """
    # Nombre de noeuds
    n = len(C)
    # Copier la matrice co t sur D
    D = C.copy()

    for k in range(n):  # Sommet interm diaire
        for i in range(n):  # Sommet source
            for j in range(n):  # Sommet destination
                # Mise   jour de la distance minimale
                if D[i][k] != INF and D[k][j] != INF:  #  viter  les additions avec des INF
                    D[i][j] = min(D[i][j], D[i][k] + D[k][j])

    return D
```