LINFO1114
Mathématiques discrètes

RAPPORT DE PROJET
DISTANCE DU PLUS COURT CHEMIN : DIJKSTRA, BELLMAN-FORD ET FLOYD-WARSHALL

Carlier Louis - 19371800
Etlik Umit - 29302100
Schamroth Arthur - 27541800

Décembre 2022

Contents

1 Introduction

Dans le cadre de notre cours LINFO1114: Mathématiques Discrètes, il nous a été demandé d'implémenter et de tester trois algorithmes étudiés : L'algorithme de **Dijkstra**, de **Bellman-Ford** et de **Floyd-Warshall**. Ces trois algorithmes ont pour objectif de déterminer le chemin le plus court entre deux noeuds d'un graphe.
Pour commencer, nous avons dû réaliser le calcul théorique de la plus petite distance entre deux points d'un graphe qui nous avait été attitré, ce calcul se base sur l'algorithme de Dijkstra et nous avons dû utiliser la méthode étudiée lors des différentes séances d'exercices. Ce calcul théorique sera par la suite vérifié au moyen de notre intégration python de cet algorithme.

2 Rappels Théoriques

Comme expliqué lors de l'introduction les trois algorithmes intégrés dans ce projet ont pour objectif commun de déterminer le chemin le plus court entre deux noeuds d'un même graphe et ainsi de régler le problème algorithmique du plus court chemin. Néanmoins, ils se distinguent les uns des autres de par leur manière de procéder et de leurs propriétés. Revenons dans un premier sur ces différents algorithmes.

2.1 Rappel Algorithme de Dijkstra

L'algorithme de Dijkstra permet de résoudre le problème du plus court chemin dans des graphes ne comportant que des poids positifs entre les différents noeuds de celui-ci.
Son fonctionnement est simple, il va commencer par initialiser les distances d'un noeud source à l'infini avec les autres noeuds du graphe. Il va ensuite placer les autres sommets du graphe dans une file de priorité, l'ordre de cette file de priorité sera géré par la distance séparant les deux noeuds l'un de l'autre. Il va ensuite passer de noeud en noeud en vérifiant à chaque fois que la distance totale parcourue jusque là est la plus petite possible par rapport aux autres chemins envisageables, si c'est le cas, le chemin continue jusqu'à arriver au noeud final. On obtient alors le plus court chemin du graphe allant d'un point A à un point B.

Voici l'équation correspondant à l'algorithme de Dijkstra :

$$L_k(v) = \min\{L_{k-1}(v), L_{k-1}(u) + w(u, v)\}$$

Où :

- k représente l'itération acutelle,
- v représente un sommet du graphe,
- u représente un sommet adjacent au sommet v,
- w(u, v) représente la distance entre le noeud u et le noeud v.

2.2 Rappel Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford permet également de résoudre le problème du plus court chemin, à la différence de l'algorithme de Dijkstran, cet algorithme prend également en compte les graphes comportant des distances de poids négatif.
Son fonctionnement consiste à initialiser dans un premier temps tous les sommets du graphe à une distance infinie du sommet de départ qui est quand à lui initialisé à une distance de 0. Ensuite, il va répéter une opération qui consiste à vérifier si la distance d'un noeud de départ vers un autre noeud peut être optimisée en passant par un autre noeud ou non. Enfin, il va vérifier si le graphe contient des poids négatifs entre certains noeuds en répétant l'opération précédente une seconde fois.

Voici l'équation correspondant à l'algorithme de Bellman-Ford :

$$d_i(v) = \min_{0 \leq j < i} \{d_j(v), d_j(u) + w(u, v)\}$$

Où :

- $d_i(v)$ indique la distance du sommet de départ au sommet v après la i-ème itération de l'algorithme.,
- $d_j(v)$ indique la distance du sommet de départ au sommet v avant la j-ème itération de l'algorithme,

- $d_j(u)$ indique la distance du sommet de départ au sommet u avant la j-ème itération de l'algorithme
- w(u, v) représente la distance entre le noeud u et le noeud v.

2.3 Rappel Algorithme de Floyd-Warshall

Tout comme l'algorithme de Dijkstra, cet algorithme permet de déterminer le plus court chemin entre deux noeuds d'un graphe uniquement si celui-ci ne comporte que des distances de poids positifs.
Tout comme pour les deux algorithmes précédents, cette méthode va, dans un premier temps, initialiser une matrice de taille identique à celle de la matrice d'entrée en fixant pour chaque noeud une valeur infinie, excepté pour la distance entre noeud sur lui-même, la distance vaudra dans ce cas-ci 0. Ensuite, il va passer dans chaque ligne de la matrice et va vérifier si la distance entre deux points peut être minimisée en passant par un autre noeud, si c'est le cas, la distance est alors mise à jour et on obtient à la fin le plus court chemin entre deux noeuds.

Voici l'équation correspondant à l'algorithme de Floyd-Warshall :

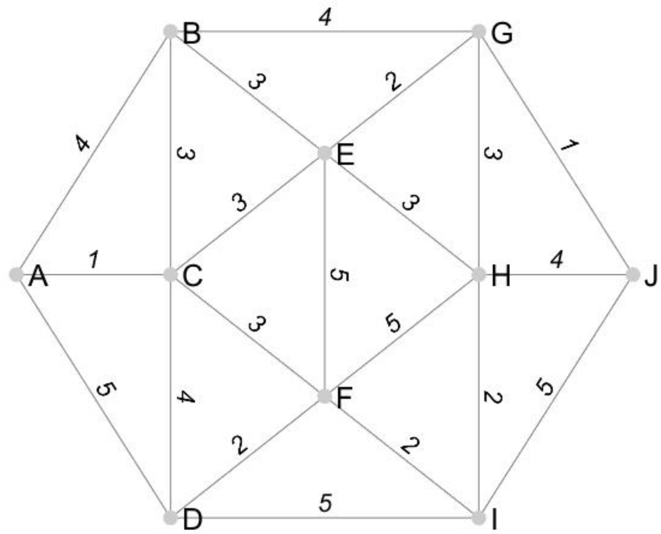
$$D_{i,j}^{k+1} = \min\left\{D_{i,j}^{k}, D_{i,k}^{k} + D_{k,j}^{k}\right\}$$

Où :

- $D_{i,j}^{k+1}$ indique la distance entre les sommets i et j après la (k+1)-ème itération de l'algorithme.
- $D_{i,j}^{k}$ indique la distance entre les sommets i et j avant la k-ème itération de l'algorithme.
- $D_{i,k}^{k}$ indique la distance entre les sommets i et k avant la k-ème itération de l'algorithme.
- $D_{k,j}^{k}$ indique la distance entre les sommets k et j avant la k-ème itération de l'algorithme.

3 Calcul Théorique

Pour rappel, il nous a été demander dans un premier temps de réaliser le calcul théorique, via l'algorithme de Dijkstra, du plus court chemin reliant le point A au point J d'un graphe.
Voici le graphe que nous nous sommes vus assigné pour ce projet :



3.1 Développement du calcul théorique

Pour effectuer ce calcul, nous sommes passés par différentes étapes décrites ci-dessous au moyen de différents tableaux :

Étape d'initialisation :

Noeuds/Étapes	L0
A	0
B	∞
C	∞
D	∞
E	∞
F	∞
G	∞
H	∞
I	∞
J	∞

Étape 1: Plus court chemin vers A = {A}

Noeuds/Étapes	L0	L1
A	0	0
B	∞	4
C	∞	1
D	∞	5
E	∞	∞
F	∞	∞
G	∞	∞
H	∞	∞
I	∞	∞
J	∞	∞

Étape 2 : Plus court chemin vers C = {A, C}

Noeuds/Étapes	L0	L1	L2
A	0	0	-
B	∞	4	4
C	∞	1	1
D	∞	5	5
E	∞	∞	4
F	∞	∞	4
G	∞	∞	∞
H	∞	∞	∞
I	∞	∞	∞
J	∞	∞	∞

Étape 3 : Plus court chemin vers B = {A, B}

Noeuds/Étapes	L0	L1	L2	L3
A	0	0	-	-
B	∞	4	4	4
C	∞	1	1	-
D	∞	5	5	∞
E	∞	∞	4	4
F	∞	∞	4	∞
G	∞	∞	∞	8
H	∞	∞	∞	∞
I	∞	∞	∞	∞
J	∞	∞	∞	∞

Étape 4 : Plus court chemin vers E = {A, C, E}

Noeuds/Étapes	L0	L1	L2	L3	L4
A	0	0	-	-	-
B	∞	4	4	4	-
C	∞	1	1	-	-
D	∞	5	5	∞	∞
E	∞	∞	4	4	4
F	∞	∞	4	∞	4
G	∞	∞	∞	8	6
H	∞	∞	∞	∞	7
I	∞	∞	∞	∞	∞
J	∞	∞	∞	∞	∞

Étape 5 : Plus court chemin vers F = {A, C, F}

Noeuds/Étapes	L0	L1	L2	L3	L4	L5
A	0	0	-	-	-	-
B	∞	4	4	4	-	-
C	∞	1	1	-	-	-
D	∞	5	5	∞	∞	5
E	∞	∞	4	4	4	-
F	∞	∞	4	∞	4	4
G	∞	∞	∞	8	6	∞
H	∞	∞	∞	∞	7	7
I	∞	∞	∞	∞	∞	6
J	∞	∞	∞	∞	∞	∞

Étape 6 : Plus court chemin vers D = {A, D}

Noeuds/Étapes	L0	L1	L2	L3	L4	L5	L6
A	0	0	-	-	-	-	-
B	∞	4	4	4	-	-	-
C	∞	1	1	-	-	-	-
D	∞	5	5	∞	∞	5	5
E	∞	∞	4	4	4	-	-
F	∞	∞	4	∞	4	4	-
G	∞	∞	∞	8	6	∞	∞
H	∞	∞	∞	∞	7	7	∞
I	∞	∞	∞	∞	∞	6	6
J	∞	∞	∞	∞	∞	∞	∞

Étape 7 : Plus court chemin vers G = {A, C, E, G}

Noeuds/Étapes	L0	L1	L2	L3	L4	L5	L6	L7
A	0	0	-	-	-	-	-	-
B	∞	4	4	4	-	-	-	-
C	∞	1	1	-	-	-	-	-
D	∞	5	5	∞	∞	5	5	-
E	∞	∞	4	4	4	-	-	-
F	∞	∞	4	∞	4	4	-	-
G	∞	∞	∞	8	6	∞	∞	6
H	∞	∞	∞	∞	7	7	∞	7
I	∞	∞	∞	∞	∞	6	6	∞
J	∞	∞	∞	∞	∞	∞	∞	7

Étape 8 : Plus court chemin vers I = {A, C, F, I}

Noeuds/Étapes	L0	L1	L2	L3	L4	L5	L6	L7	L8
A	0	0	-	-	-	-	-	-	-
B	∞	4	4	4	-	-	-	-	-
C	∞	1	1	-	-	-	-	-	-
D	∞	5	5	∞	∞	5	5	-	-
E	∞	∞	4	4	4	-	-	-	-
F	∞	∞	4	∞	4	4	-	-	-
G	∞	∞	∞	8	6	∞	∞	6	-
H	∞	∞	∞	∞	7	7	∞	7	7
I	∞	∞	∞	∞	∞	6	6	∞	6
J	∞	∞	∞	∞	∞	∞	∞	7	7

Étape 9 : Plus court chemin vers H = {A, C, E, H}

Noeuds/Étapes	L0	L1	L2	L3	L4	L5	L6	L7	L8	L9
A	0	0	-	-	-	-	-	-	-	-
B	∞	4	4	4	-	-	-	-	-	-
C	∞	1	1	-	-	-	-	-	-	-
D	∞	5	5	∞	∞	5	5	-	-	-
E	∞	∞	4	4	4	-	-	-	-	-
F	∞	∞	4	∞	4	4	-	-	-	-
G	∞	∞	∞	8	6	∞	∞	6	-	-
H	∞	∞	∞	∞	7	7	∞	7	7	7
I	∞	∞	∞	∞	∞	6	6	∞	6	-
J	∞	∞	∞	∞	∞	∞	∞	7	7	7

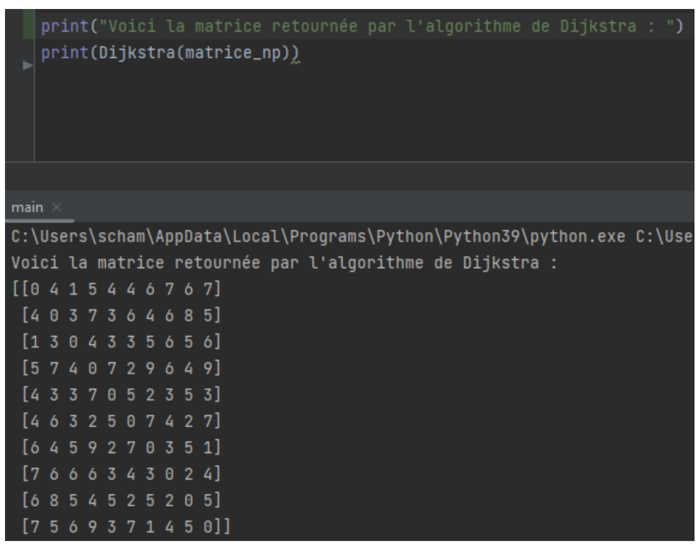
Étape 10 : Plus court chemin vers J = {A, C, E, G, J}

Noeuds/Étapes	L0	L1	L2	L3	L4	L5	L6	L7	L8	L9
A	0	0	-	-	-	-	-	-	-	-
B	∞	4	4	4	-	-	-	-	-	-
C	∞	1	1	-	-	-	-	-	-	-
D	∞	5	5	∞	∞	5	5	-	-	-
E	∞	∞	4	4	4	-	-	-	-	-
F	∞	∞	4	∞	4	4	-	-	-	-
G	∞	∞	∞	8	6	∞	∞	6	-	-
H	∞	∞	∞	∞	7	7	∞	7	7	-
I	∞	∞	∞	∞	∞	6	6	∞	6	-
J	∞	∞	∞	∞	∞	∞	∞	7	7	7

Ainsi, le chemin le plus court reliant les noeuds A et J = {A, C, E, G, J}. Ce chemin possède une distance totale de 7.

3.2 Vérification du calcul théorique avec l'implémentation Python de Dijkstra

Vérifions à présent notre résultat avec celui trouvé par notre implémentation de l'algorithme.



```
print("Voici la matrice retournée par l'algorithme de Dijkstra : ")
print(Dijkstra(matrice_np))
```

```
main
C:\Users\scham\AppData\Local\Programs\Python\Python39\python.exe C:\Use
Voici la matrice retournée par l'algorithme de Dijkstra :
[[0 4 1 5 4 4 6 7 6 7]
 [4 0 3 7 3 6 4 6 8 5]
 [1 3 0 4 3 3 5 6 5 6]
 [5 7 4 0 7 2 9 6 4 9]
 [4 3 3 7 0 5 2 3 5 3]
 [4 6 3 2 5 0 7 4 2 7]
 [6 4 5 9 2 7 0 3 5 1]
 [7 6 6 6 3 4 3 0 2 4]
 [6 8 5 4 5 2 5 2 0 5]
 [7 5 6 9 3 7 1 4 5 0]]
```

Figure 1: Vérification du plus court chemin avec l'algorithme en python

Comme nous pouvons le constater en regardant la dernière colonne (correspondant au noeud J) de la première ligne (correspondant à la ligne A), nous pouvons conclure que le chemin le plus court entre le noeud A et le noeud J possède une distance de 7.
Visuellement, il est également relativement simple de constater que le chemin le plus rapide entre ces deux points correspondant effectivement au trajet {A, C, E, G, J}, comme le démontre l'image ci-dessous.

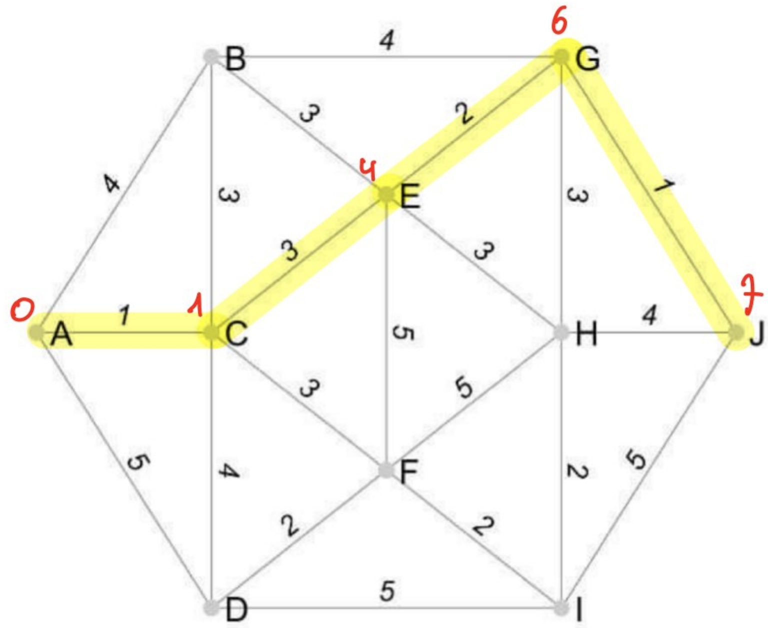
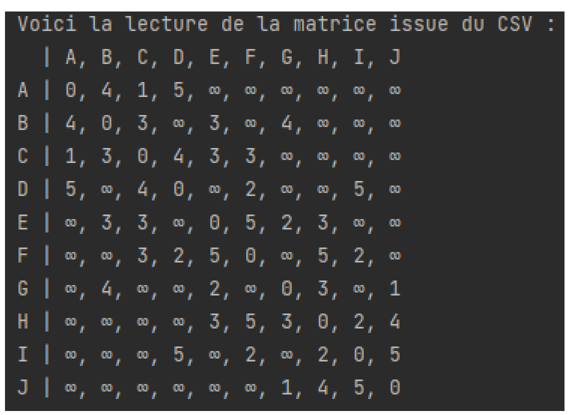


Figure 2: Vérification du plus court chemin sur le graphe

4 Procédure Main

Pour rappel, il nous était également demandé d'implémenter une procédure "main" permettant de lancer automatiquement une lecture de la matrice de coûts écrite dans un fichier CSV. Cette procédure devait également lancer les différents algorithmes et nous indiquer les matrices de sortie des trois fonctions implémentées.
Pour commencer, voici la sortie affichée correspondant à la lecture du graphe :



```
Voici la lecture de la matrice issue du CSV :
  | A, B, C, D, E, F, G, H, I, J
A | 0, 4, 1, 5, ∞, ∞, ∞, ∞, ∞, ∞
B | 4, 0, 3, ∞, 3, ∞, 4, ∞, ∞, ∞
C | 1, 3, 0, 4, 3, 3, ∞, ∞, ∞, ∞
D | 5, ∞, 4, 0, ∞, 2, ∞, ∞, 5, ∞
E | ∞, 3, 3, ∞, 0, 5, 2, 3, ∞, ∞
F | ∞, ∞, 3, 2, 5, 0, ∞, 5, 2, ∞
G | ∞, 4, ∞, ∞, 2, ∞, 0, 3, ∞, 1
H | ∞, ∞, ∞, ∞, 3, 5, 3, 0, 2, 4
I | ∞, ∞, ∞, 5, ∞, 2, ∞, 2, 0, 5
J | ∞, ∞, ∞, ∞, ∞, ∞, 1, 4, 5, 0
```

Figure 3: Lecture de la matrice de coûts

Comme nous pouvons le constater, chaque noeud est correctement représenté et les liens entre ces différents noeuds sont également correctement exprimés.

Ensuite, voici les différentes matrice obtenues par nos trois fonctions liées aux différents algorithmes :

```
Voici la matrice issue de Dijkstra :
[[0 4 1 5 4 4 6 7 6 7]
 [4 0 3 7 3 6 4 6 8 5]
 [1 3 0 4 3 3 5 6 5 6]
 [5 7 4 0 7 2 9 6 4 9]
 [4 3 3 7 0 5 2 3 5 3]
 [4 6 3 2 5 0 7 4 2 7]
 [6 4 5 9 2 7 0 3 5 1]
 [7 6 6 6 3 4 3 0 2 4]
 [6 8 5 4 5 2 5 2 0 5]
 [7 5 6 9 3 7 1 4 5 0]]
```

(a) Matrice de distances obtenus via Dijkstra

```
Voici la matrice issue de Bellman-Ford :
[[0 4 1 5 4 4 6 7 6 7]
 [4 0 3 7 3 6 4 6 8 5]
 [1 3 0 4 3 3 5 6 5 6]
 [5 7 4 0 7 2 9 6 4 9]
 [4 3 3 7 0 5 2 3 5 3]
 [4 6 3 2 5 0 7 4 2 7]
 [6 4 5 9 2 7 0 3 5 1]
 [7 6 6 6 3 4 3 0 2 4]
 [6 8 5 4 5 2 5 2 0 5]
 [7 5 6 9 3 7 1 4 5 0]]
```

(b) Matrice de distances obtenus via Bellman-Ford

```
Voici la matrice issue de Floyd-Warshall :
[[0 4 1 5 4 4 6 7 6 7]
 [4 0 3 7 3 6 4 6 8 5]
 [1 3 0 4 3 3 5 6 5 6]
 [5 7 4 0 7 2 9 6 4 9]
 [4 3 3 7 0 5 2 3 5 3]
 [4 6 3 2 5 0 7 4 2 7]
 [6 4 5 9 2 7 0 3 5 1]
 [7 6 6 6 3 4 3 0 2 4]
 [6 8 5 4 5 2 5 2 0 5]
 [7 5 6 9 3 7 1 4 5 0]]
```

(c) Matrice de distances obtenus via Floyd-Warshall

Nous pouvons constater qu'elles retournent chacune une matrice identique aux autres, ce qui indique que nos trois algorithmes sont correctement implémentés.

5 Conclusion

En conclusion, nous sommes satisfaits des résultats obtenus, ceux-ci nous semblent tout à fait corrects. De plus, ce projet nous a permis de pousser notre maitrise et notre compréhension des différents algorithmes tout en réalisant un exemple complexe de cas d'utilisation de ces différents algorithmes.

6 Annexe

6.1 Traitement du CSV

```python
import numpy as np

liens = []
relations_matrice = {}
relations_array = []
matrice = []
# Traitement de la matrice depuis le csv vers matrice numpy
with open("../data.csv") as file:
    lines = file.readlines()
    for i in range(len(lines)):
        if i == 0:
            for j in lines[i].strip("\n").split(", "):
                relations_matrice[j] = []
        else:
            relations = lines[i].strip("\n").split(", ")
            relations = np.array(relations)
            for j in range(len(relations)):
                if int(relations[j]) == 1000000000000:
                    relations[j] = 0
            relations = relations.astype(int)
            liens.append(relations)
    matrice_np = np.array(liens)

```

Figure 5: Traitement de la matrice depuis le CSV

6.2 Algorithme de Dijkstra

```python
def Dijkstra(C: np.matrix):
    # Initialisation de la matrice des distances de sortie.
    # Ces distances sont initialisées à l'infini.
    D = np.full((C.shape[0], C.shape[1]), float('inf'))

    # Passage dans chaque ligne de la matrice
    for source in range(C.shape[0]):
        # Initialisation des distances à zéro pour les noeuds envers eux-mêmes
        D[source, source] = 0
        # Initialisation des noeuds précédents chaque noeud du graphe.
        # Initialement à -1 car cela indique que ce noeud n'a pas encore de prédécesseur.
        noeuds_precedents = np.full(C.shape[0], -1)

        # Création de l'ensemble des noeuds non visités.
        # Liste des indices des noeuds non-visités.
        nodes_index = []
        for i in range(C.shape[0]):
            nodes_index.append(i)
        # Passage en set pour être sûr de l'immuabilité de la liste des index
        noeuds_non_visites = set(nodes_index)

        while noeuds_non_visites:
            # Recherche du noeud le plus proche.
            noeud_courant = min(noeuds_non_visites, key=lambda x: D[source, x])
            # Suppression du noeud courant des noeuds non visités.
            noeuds_non_visites.remove(noeud_courant)
```

Figure 6: Algorithme de Dijkstra 1ère Partie

```python
    while noeuds_non_visites:
        # Recherche du noeud le plus proche.
        noeud_courant = min(noeuds_non_visites, key=lambda x: D[source, x])
        # Suppression du noeud courant des noeuds non visités.
        noeuds_non_visites.remove(noeud_courant)

        # Mise à jour des distances des voisins
        # Passage dans chaque élément de chaque ligne de la matrice.
        for indice_voisin, poids_voisin in enumerate(C[noeud_courant]):
            # Si noeud pas connecté à voisin, ca passe
            if poids_voisin == 0:
                continue
            # Calcul du nouveau poids du voisin
            nouveau_poids = D[source, noeud_courant] + poids_voisin
            # Si nouveau poids inférieur au poids enregistré pour ce voisin :
            # Le noeud courant change de valeur et le noeud courant est mis à jour.
            if nouveau_poids < D[source, indice_voisin]:
                D[source, indice_voisin] = nouveau_poids
                noeuds_precedents[indice_voisin] = noeud_courant

    # Gestion des "." indésirables dans la matrice de sortie.
    D = np.around(D)
    D = D.astype(int)
    return D
```

Figure 7: Algorithme de Dijkstra 2ème Partie

6.3 Algorithme de Floyd-Warshall

```python
# Traitement de la matrice qui contient des distances entre chaque noeuds.
#
# Données : Un graphe orienté pondéré.
# Résultat : Le plus court chemin entre toute paire de sommets.
def Floyd_Warshall(C: np.matrix):
    # L'algorithme est constitué de N itération principales;
    # pour chaque itération k, on calcule les plus courts chemins entre toute paire de sommets
    # avec des sommets intermédiaires appartenant uniquement à l'ensemble {1,2,3,...k}
    for k in range(len(C)):
        for i in range(len(C)):
            for j in range(len(C)):
                # ici si le cout est égale à 0,ceci signifie que le cout est 10 exposant 12
                # bien évidemment, à l'exception des elements appartenant à la diagonale.
                if i != j and C[i][j] == 0:
                    C[i][j] = 1000000000

                # Le noeud courant change de la valeur s'il existe un sommet intermédiaire qui donc possède un
                # circuit de coût plus petit.
                C[i][j] = min(int(C[i][j]), int(C[i][k]) + int(C[k][j]))
    D = C
    return D
```

Figure 8: Algorithme de Floyd-Warshall

6.4 Algorithme de Bellman-Ford

```python
def Bellman_Ford(C: np.matrix):
    # N = Nombre de noeuds du graphe
    N = C.shape[0]
    nv_matrice = []
    # Mise à jour de la matrice pour qu'elle transforme les 0 en valeur infinie sauf pour les noeuds envers eux-même
    for i in range(len(C)):
        ligne = []
        for j in range(len(C[i])):
            if i != j:
                if C[i][j] == 0:
                    ligne.append(float("inf"))
                else:
                    ligne.append(C[i][j])
            else:
                ligne.append(0)
        nv_matrice.append(ligne)

    # Création de la matrice des distances minimales.
    # Les distances initiales sont celles de la matrice d'entrée.
    D = np.array(nv_matrice)

    # Répéter l'algorithme N-1 fois
    for i in range(N - 1):
        # Mettre à jour les distances minimales en prenant en compte tous les chemins possibles
        for j in range(N):
            for k in range(N):
                if D[j, i] != np.inf and D[i, k] != np.inf:
                    D[j, k] = min(D[j, k], D[j, i] + D[i, k])

    return D
```

Figure 9: Algorithme de Bellman-Ford

6.5 Procédure Main

```python
if __name__ == '__main__':
    matrice_lecture = []
    lecture = "  | A, B, C, D, E, F, G, H, I, J \n"
    with open("../data.csv") as file:
        lines = file.readlines()
        for i in range(len(lines)):
            ligne = []
            if i == 0:
                continue
            else:
                for j in lines[i].split(", "):
                    ligne.append(int(j))
            matrice_lecture.append(ligne)
    for i in range(len(matrice_lecture)):
        for j in range(len(matrice_lecture[i])):
            if i == 0 and j == 0 and matrice_lecture[i][j] == 1000000000000:
                lecture += "A | ∞, "
            elif i == 0 and j == 0 and matrice_lecture[i][j] != 1000000000000:
                lecture = lecture + "A | " + str(matrice_lecture[i][j]) + ", "
            elif i == 1 and j == 0 and matrice_lecture[i][j] == 1000000000000:
                lecture += "B | ∞, "
            elif i == 1 and j == 0 and matrice_lecture[i][j] != 1000000000000:
                lecture = lecture + "B | " + str(matrice_lecture[i][j]) + ", "
            elif i == 2 and j == 0 and matrice_lecture[i][j] == 1000000000000:
                lecture += "C | ∞, "
            elif i == 2 and j == 0 and matrice_lecture[i][j] != 1000000000000:
                lecture = lecture + "C | " + str(matrice_lecture[i][j]) + ", "
            elif i == 3 and j == 0 and matrice_lecture[i][j] == 1000000000000:
```

Figure 10: Procédure main 1ère partie

```python
            elif i == 7 and j == 0 and matrice_lecture[i][j] == 1000000000000:
                lecture += "H | ∞, "
            elif i == 7 and j == 0 and matrice_lecture[i][j] != 1000000000000:
                lecture = lecture + "H | " + str(matrice_lecture[i][j]) + ", "
            elif i == 8 and j == 0 and matrice_lecture[i][j] == 1000000000000:
                lecture += "I | ∞, "
            elif i == 8 and j == 0 and matrice_lecture[i][j] != 1000000000000:
                lecture = lecture + "I | " + str(matrice_lecture[i][j]) + ", "
            elif i == 9 and j == 0 and matrice_lecture[i][j] == 1000000000000:
                lecture += "J | ∞, "
            elif i == 9 and j == 0 and matrice_lecture[i][j] != 1000000000000:
                lecture = lecture + "J | " + str(matrice_lecture[i][j]) + ", "
            elif j == len(matrice_lecture[i]) - 1 and matrice_lecture[i][j] == 1000000000000:
                lecture += "∞ \n"
            elif j == len(matrice_lecture[i]) - 1 and matrice_lecture[i][j] != 1000000000000:
                lecture = lecture + str(matrice_lecture[i][j]) + " \n"
            elif j != len(matrice_lecture[i]) - 1 and matrice_lecture[i][j] == 1000000000000:
                lecture += "∞, "
            elif j != len(matrice_lecture[i]) - 1 and matrice_lecture[i][j] != 1000000000000:
                lecture = lecture + str(matrice_lecture[i][j]) + ", "

    print("Voici la lecture de la matrice issue du CSV : ")
    print(lecture)
    print("Voici la matrice issue de Dijkstra : ")
    print(Dijkstra(matrice_np))
    print("Voici la matrice issue de Floyd-Warshall : ")
    print(Floyd_Warshall(matrice_np))
    print("Voici la matrice issue de Bellman-Ford : ")
    print(Bellman_Ford(matrice_np))
```

Figure 11: Procédure main 2ème partie