

# TP 3 – Floyd-Warshall et largeur maximum

Gaëtan Staquet

2026

Ce TP est découpé en deux parties. La première porte sur la recherche de chemins optimaux via l'algorithme de Floyd-Warshall. La seconde se concentre sur un problème NP-complet de graphes pondérés : la recherche d'un chemin qui passe par tous les sommets d'un graphe en minimisant le coût total.

Avant de vous lancer dans le TP, voici un rappel des consignes pour les TP de GRAAL.

- La première partie est la plus importante pour la suite du cours. Essayez quand même d'avancer le plus possible dans la seconde partie.
- Le langage à utiliser pendant les TP est Python.
- Vous êtes libre d'utiliser les outils que vous voulez, y inclus des LLM et autres IA génératives.
- Veuillez limiter votre usage de modules externes. En particulier, les bibliothèques de graphes (comme `NetworkX` ou `graph-tool`) sont interdites, étant donné que le but est d'implémenter les algorithmes par vous-même.
- **Il vous est demandé d'utiliser le même projet Python pendant l'ensemble des TP. Ceci vous permettra de mettre en application des bonnes pratiques à plus long terme, dans un contexte dans lequel les besoins évoluent et vous sont inconnus. Notamment, pensez à bien structurer, documenter et tester votre code en amont afin de pouvoir facilement le modifier plus tard.**
- À cette fin, créez un ou des modules pour la première partie. Le code pour la seconde partie peut ensuite se reposer sur ces modules. En d'autres termes, séparez les algorithmes génériques des applications spécifiques.

# Partie I — Chemins optimaux

Au terme de cette partie, vous aurez une implémentation de l'**algorithme de Floyd-Warshall**. Cet algorithme trouve les poids des chemins optimaux dans un graphe orienté pondéré dont les poids sont dans  $\mathbb{R}$ . Autrement dit, contrairement à l'algorithme de Dijkstra, avoir des valeurs négatives n'est pas un problème.

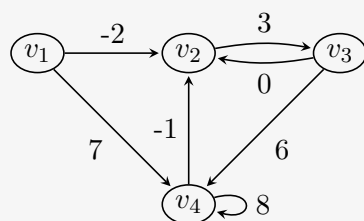
## I.1. Implémentation des graphes pondérés

Vu que nous allons manipuler des graphes pondérés orientés, il faut étendre votre implémentation du premier TP. Dans celui-ci, il vous était demandé de créer une classe pour les arcs. La raison devrait maintenant être claire : il suffit de changer cette classe (et les méthodes pour créer des arcs) pour supporter des poids.

**Exercice I.1.1.** Ajoutez des poids dans votre implémentation du TP 1. Pour minimiser le travail à réaliser, vous pouvez garder exactement la même structure de classes, rajouter un champ `weight` dans `Edge` et ajouter un argument (avec une valeur par défaut de 0) dans `add_edge`. La signature de cette fonction doit être `add_edge(self, source, target, weight = 0)`.

L'algorithme dont on va parler par la suite se base sur la **représentation matricielle** d'un graphe pondéré. L'idée est similaire à la représentation matricielle d'un graphe non pondéré : on a une matrice carrée et, pour chaque paire de sommets  $u$  et  $v$ , on stocke le poids de l'arc de  $u$  à  $v$ . S'il n'y a pas d'arc de  $u$  à  $v$ , on met  $+\infty$ .<sup>1</sup> Remarquez qu'avoir un zéro dans la matrice ne représente plus l'absence d'arc, contrairement à la matrice d'adjacence vu en cours.

*Exemple I.1.2.* Voici un graphe orienté pondéré et sa représentation matricielle.



$$\begin{array}{c} v_1 \quad v_2 \quad v_3 \quad v_4 \\ \begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{matrix} \begin{pmatrix} +\infty & -2 & +\infty & 7 \\ +\infty & +\infty & 3 & +\infty \\ +\infty & 0 & +\infty & 6 \\ +\infty & -1 & +\infty & 8 \end{pmatrix} \end{array}$$

**Exercice I.1.3.** Implémentez une méthode qui retourne la représentation matricielle d'un graphe orienté pondéré.

1. En Python, vous pouvez utiliser `float("inf")`.

## I.2. Algorithme de Floyd-Warshall

L'**algorithme de Floyd-Warshall** (parfois appelé simplement « algorithme de Floyd ») est un algorithme qui calcule le poids d'un chemin optimal entre **n'importe quelle paire de sommets**. Contrairement à l'algorithme de Dijkstra, on ne fixe donc pas un sommet de départ. Cet algorithme s'exprime facilement grâce à la représentation matricielle du graphe pondéré. L'idée est assez proche du calcul de la fermeture transitive via la matrice d'adjacence d'un graphe orienté.

Dans un premier temps, on modifie la matrice pour mettre des 0 sur la diagonale, *i. e.*, on encode explicitement le fait qu'un chemin optimal de  $v$  à  $v$  est  $(v)$  dont le poids est nul. Ensuite, on itère plusieurs fois sur la matrice en ne gardant que le plus petit poids vu jusqu'à présent.

---

### Algorithme 1. Algorithme de Floyd-Warshall

---

```

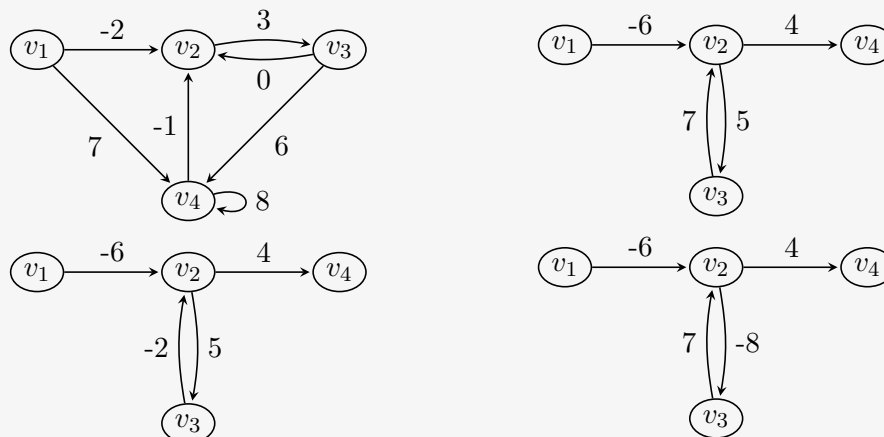
1 : procédure FLOYDWARSHALL( $\mathcal{G} = (V, E)$ )
2 :    $M \leftarrow$  la représentation matricielle de  $\mathcal{G}$ 
3 :   pour chaque  $i \in \{1, \dots, |V|\}$  faire
4 :      $M_{i,i} \leftarrow 0$ 
5 :   pour chaque  $k \in \{1, \dots, |V|\}$  faire
6 :     pour chaque  $i \in \{1, \dots, |V|\}$  faire
7 :       pour chaque  $j \in \{1, \dots, |V|\}$  faire
8 :          $M_{i,j} \leftarrow \min(M_{i,j}, M_{i,k} + M_{k,j})$ 
9 :   retourner  $M$ 

```

---

**Exercice I.2.1.** Quelle est la complexité de l'algorithme de Floyd-Warshall ? À votre avis, à quoi correspondent les différentes boucles de l'algorithme ?

**Exercice I.2.2.** Implémentez cet algorithme et testez-le sur les graphes suivants (au minimum).



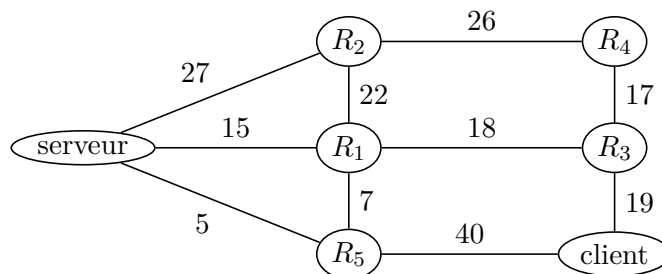
Est-ce que les résultats vous semblent corrects ? Sinon, quel graphe pose un problème et pourquoi ?

La suite de ce TP s'intéresse à une application d'un algorithme dérivé de Floyd-Warshall.

## Partie II — Un chemin le plus large

Jusqu'à présent, on a travaillé sur minimiser le coût total d'un chemin entre deux sommets. Parfois, on a plutôt besoin de chercher un chemin qui maximise le poids de l'arc de poids minimum. Autrement dit, plutôt que de minimiser la somme des poids, on va chercher un chemin le long duquel la plus petite valeur est la plus grande possible.

Par exemple, interprétons les sommets d'un graphe comme les routeurs d'un réseau et le poids des arcs comme la bande-passante disponible entre les deux routeurs. Dans ce cas, on veut maximiser la bande-passante du chemin entre le serveur et le client. La bande-passante du chemin complet est bornée par la plus petite bande-passante entre deux routeurs de ce chemin. Pour illustrer ceci, prenons le graphe non orienté suivant.



Un chemin qui maximise la bande-passante minimum est (serveur,  $R_2$ ,  $R_1$ ,  $R_3$ , client). En effet, n'importe quel autre chemin doit nécessairement passer par une arête de poids inférieur à 18. En pratique, on doit pouvoir transmettre des messages de manière efficace entre n'importe quelle paire de nœuds du réseau. Il faut donc qu'on ait connaissance de la bande-passante maximum entre chaque paire.

### II.1. Largeur maximum

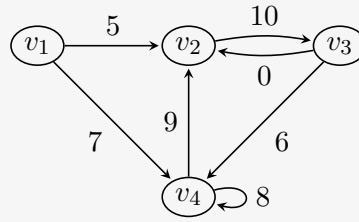
Commençons par formaliser l'objet qu'on va calculer.

**Définition II.1.1.** Soit  $\mathcal{G} = (V, E, p)$  un graphe orienté pondéré. On appelle **largeur** d'un chemin  $(v_1, \dots, v_n)$  le minimum des poids de ses arcs, *i. e.*, la valeur

$$\min_{i \in \{1, \dots, n-1\}} p((v_i, v_{i+1})).$$

On s'intéresse à la largeur **maximum** entre deux sommets.

*Exemple II.1.2.* Prenons le graphe suivant.



La largeur du chemin  $(v_1, v_2, v_3)$  est le minimum de  $\{5, 10\}$ , donc 5. Ce n'est pas la largeur maximum de  $v_1$  à  $v_3$  : le chemin  $(v_1, v_4, v_2, v_3)$  a une largeur de 7. On peut résumer les largeurs maximums entre chaque paire de sommets dans un tableau. On ignore la largeur d'un chemin d'un sommet vers lui-même (vu qu'il suffit de ne pas bouger). Une valeur de  $+\infty$  indique qu'il n'y a pas de chemin.

	$v_1$	$v_2$	$v_3$	$v_4$
$v_1$		7	7	7
$v_2$	$+\infty$		10	6
$v_3$	$+\infty$	6		6
$v_4$	$+\infty$	9	9	

Notre objectif est de calculer la largeur maximum entre n'importe quelle paire de sommets de manière efficace. On peut y arriver via une adaptation de l'algorithme de Floyd-Warshall. Au lieu de faire la somme des poids des chemins de  $v_i$  à  $v_k$  et de  $v_k$  à  $v_j$ , on récupère le maximum des deux poids.

---

**Algorithme 2.** Un algorithme pour la largeur maximum

---

```

1 : procédure LARGEURMAXIMUM( $\mathcal{G} = (V, E)$ )
2 :    $M \leftarrow$  la représentation matricielle de  $\mathcal{G}$ 
3 :   pour chaque  $k \in \{1, \dots, |V|\}$  faire
4 :     pour chaque  $i \in \{1, \dots, |V|\}$  faire
5 :       pour chaque  $j \in \{1, \dots, |V|\}$  faire
6 :         si  $i \neq j \wedge i \neq k \wedge j \neq k$  alors  $\triangleright$  On ignore les chemins de  $v$  vers  $v$ 
7 :           si  $M_{i,k} \neq +\infty \wedge M_{k,j} \neq +\infty$  alors  $\triangleright$  On évite de propager  $+\infty$ 
8 :             si  $M_{i,j} = +\infty$  alors  $\triangleright$  On écrase un  $+\infty$ 
9 :                $M_{i,j} \leftarrow \min(M_{i,k}, M_{k,j})$ 
10 :            sinon
11 :               $M_{i,j} \leftarrow \max(M_{i,j}, \min(M_{i,k}, M_{k,j}))$ 
12 :   retourner  $M$ 

```

---

**Exercice II.1.3.** Implémentez cet algorithme.

## II.2. Élections plurinominales

La recherche de la largeur maximum entre les sommets d'un graphe peut servir à identifier qui gagne des élections quand les votes sont classés : au lieu de choisir une personne parmi  $\{A, B, C, D\}$ , on les classe par ordre de préférence. Ainsi, un vote potentiel est  $BDCA$ , pour indiquer qu'on préfère  $B$  à  $D$ ,  $D$  à  $C$  et  $C$  à  $A$ . On va ici utiliser la **méthode de Schulze**. L'idée est que  $A$  bat  $B$  si une majorité de personnes classe  $A$  avant  $B$ . On étend ce principe par **transitivité** : si  $A$  bat  $B$  et  $B$  bat  $C$ , alors  $A$  bat  $C$ .

Prenons un exemple pour expliquer cette méthode, toujours avec  $\{A, B, C, D\}$  comme candidates et candidats. Supposons qu'il y ait 40 électeurices dont les votes sont donnés dans la table suivante.

Vote	Nombre de voix
ABDC	6
DCAB	10
CBAD	9
BADC	9
CDAB	6

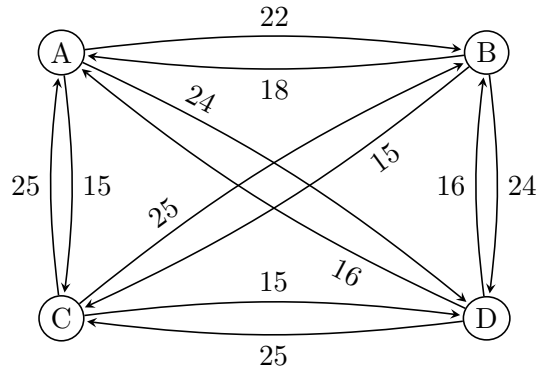
**Exercice II.2.1.** Implémentez une fonction qui génère un ensemble de votes de manière aléatoire. La fonction doit prendre en paramètre le nombre de candidates et candidats et le nombre d'électeurices. Chaque vote doit être un classement strict entre les candidates et candidats, *i.e.*, on interdit les égalités.

**Attention :** on interdit les votes blancs. Donc, la somme du nombre de voix exprimées doit être égale au nombre d'électeurices.

Dans un premier temps, on construit une nouvelle table qui encode les préférences entre les candidates et candidats. Par exemple, il y a 22 votes qui préfèrent  $A$  à  $B$ .

Préférences	A	B	C	D
A		22	15	24
B	18		15	24
C	25	25		15
D	16	16	25	

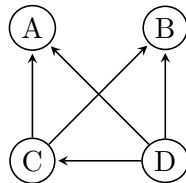
Il n'est pas évident de déterminer qui remporte les élections depuis ce tableau. On va donc appliquer le principe de transitivité pour renforcer les préférences. Pour ce faire, on construit (évidemment) un graphe pondéré orienté en utilisant les valeurs de la table. Ainsi, on aura un arc de  $A$  à  $B$  avec un poids de 22.



Maintenant, on cherche la largeur maximum entre chaque pair de sommets distincts. À nouveau, les résultats peuvent être donnés dans une table. Par exemple, la largeur de A à B est 24, via le chemin  $(A, D, C, B)$ . Dans l'autre sens, la largeur est également 24 par le chemin  $(B, D, C, A)$ .

Largeurs maximums	A	B	C	D
A		24	24	24
B	24		24	24
C	25	25		24
D	25	25	25	

À partir de cette table, on peut déterminer que C **bat** A, vu que la largeur maximum de C à A est **strictement plus grande** que la largeur maximum de A à C. Construisons un nouveau graphe (non pondéré) qui représente cette relation.



Il nous reste à récupérer le sommet de plus grand degré sortant, ici D. Donc, D remporte l'élection selon la méthode de Schulze.

**Exercice II.2.2.** Implémentez la méthode de Schulze.

**Attention :** en cas d'égalité à la fin, personne ne remporte l'élection.