



# S2.02

- Exploration algorithmique d'un problème -

Basset Adrien  
Hisabu Nathan Tekeste  
Magadiyev Imam  
Groupe B (B7)

## Table des matières

|   |    |
|---|----|
| <b>I - Comparaison d'algorithmes de plus courts chemins.</b>  |    |
| <b>1) Connaissance des algorithmes de plus courts chemins :</b>                                       | 3  |
| 1.1 Présentation de l'algorithme de Dijkstra :  | 3  |
| 1.2 Présentation de l'algorithme de Bellman-Ford :  | 6  |
| <b>2) Dessin d'un graphe et d'un chemin à partir de sa matrice.</b>                                   | 7  |
| 2.1 Dessin d'un graphe.....   | 7  |
| 2.2 Dessin d'un chemin .....  | 8  |
| <b>3) Génération aléatoire de matrices de graphes pondéré</b>   |    |
| 3.1 Graphes avec 50% de flèches.....  | 9  |
| 3.2 Graphes avec une proportion variables p de flèches  | 9  |
| <b>4) Codage des algorithmes de plus court chemin</b> .....   | 10 |
| 4.1 Codage de l'algorithme de Dijkstra .....  | 10 |
| 4.2 Codage de l'algorithme de Bellman-Ford.....   | 10 |
| <b>5) Influence du choix de la liste ordonnée des flèches pour l'algorithme de Bellman-Ford</b> ..... | 10 |
| <b>6) Comparaison expérimentale des complexités</b>   |    |
| 6.1 Deux fonctions "temps de calcul" .....  | 11 |
| 6.2 Comparaison et identification des deux fonctions temps.....                                       | 12 |
| 6.3 Conclusion.....   | 13 |
| <b>II - Seuil de forte connexité d'un graphe orienté.</b> .....                                       | 13 |
| <b>7) Test de forte connexité</b> .....   | 13 |
| <b>8) Forte connexité pour un graphe avec p = 50% de flèches</b> .....                                | 14 |
| <b>9) Détermination du seuil de forte connexité</b> .....   | 14 |
| <b>10) Etude et identification de la fonction seuil</b>   |    |
| 10.1 Représentation graphique de seuil(n).....  | 14 |
| 10.2 Identification de la fonction seuil(n).....  | 15 |

## I - Comparaison d'algorithmes de plus courts chemins.

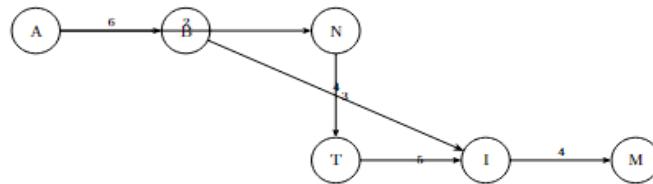
### 1) Connaissance des algorithmes de plus courts chemins :

#### 1.1 Présentation de l'algorithme de Dijkstra :

Soit la matrice M définie suivante

$$M = \begin{array}{cccccc} & A & B & N & T & I & M \\ A & \left( \begin{array}{cccccc} 0 & 6 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right) \\ B & \\ N & \\ T & \\ I & \\ M & \end{array}$$

La représentation graphique G de la matrice M :



Nous cherchons à déterminer le **chemin le plus court** entre le sommet **A** (point de départ) et le sommet **M** (point d'arrivée).

#### Etape d'initialisation

Nous créons une table dans laquelle chaque sommet est initialisé avec une distance infinie, sauf le sommet de départ **A**, qui est initialisé à 0. Nous ajoutons également une colonne « Choix » pour indiquer à chaque étape le sommet sélectionné.

| Etape          | A    | B  | N  | T  | I  | M  | Choix |
|----------------|------|----|----|----|----|----|-------|
| Initialisation | 0(A) | +∞ | +∞ | +∞ | +∞ | +∞ | A(0)  |

#### Étape 1 : depuis le sommet A

À partir de **A**, nous avons deux chemins accessibles :

Vers **B** avec un coût de  $6 \rightarrow 0 + 6 = 6$

Vers **N** avec un coût de  $2 \rightarrow 0 + 2 = 2$

Nous mettons à jour les distances de ces sommets :

| Etape          | A    | B    | N    | T  | I  | M  | Choix |
|----------------|------|------|------|----|----|----|-------|
| Initialisation | 0(A) | +∞   | +∞   | +∞ | +∞ | +∞ | A(0)  |
| 1              | X    | 6(A) | 2(A) | +∞ | +∞ | +∞ | N(2)  |

Le sommet **N** a la plus petite distance, il est sélectionné comme prochain sommet définitif.

**Étape 2 : depuis le sommet N**

Depuis **N**, nous pouvons aller vers :

- ° **T** avec un coût de  $4 \rightarrow 2(N) + 4 = 6$

Le sommet **T** est mis à jour avec la distance 6.

| Etape          | A    | B    | N    | T    | I  | M  | Choix |
|----------------|------|------|------|------|----|----|-------|
| Initialisation | 0(A) | +∞   | +∞   | +∞   | +∞ | +∞ | A(0)  |
| 1              | X    | 6(A) | 2(A) | +∞   | +∞ | +∞ | N(2)  |
| 2              | X    | 6(A) | X    | 6(N) | +∞ | +∞ | B(6)  |

**Étape 3 : depuis le sommet B**

Depuis **B**, nous pouvons aller vers :

**T** avec un coût de  $3 \rightarrow 6(B) + 3 = 9$

| Etape          | A    | B    | N    | T    | I  | M  | Choix |
|----------------|------|------|------|------|----|----|-------|
| Initialisation | 0(A) | +∞   | +∞   | +∞   | +∞ | +∞ | A(0)  |
| 1              | X    | 6(A) | 2(A) | +∞   | +∞ | +∞ | N(2)  |
| 2              | X    | 6(A) | X    | 6(N) | +∞ | +∞ | B(2)  |
| 3              | X    | X    | X    | 6(N) | +∞ | +∞ | T(6)  |

Cependant, le sommet **T** a déjà une distance de **6** (meilleure), donc on ne met pas à jour.

**Étape 4 : depuis le sommet T**

Depuis **T**, nous avons un accès à :

**I** avec un coût de  $5 \rightarrow 6 + 5 = 11$

Nous mettons à jour **I** :

| Etape          | A    | B    | N    | T    | I     | M  | Choix |
|----------------|------|------|------|------|-------|----|-------|
| Initialisation | 0(A) | +∞   | +∞   | +∞   | +∞    | +∞ | A(0)  |
| 1              | X    | 6(A) | 2(A) | +∞   | +∞    | +∞ | N(2)  |
| 2              | X    | 6(A) | X    | 6(N) | +∞    | +∞ | B(2)  |
| 3              | X    | X    | X    | 6(N) | +∞    | +∞ | T(6)  |
| 4              | X    | X    | X    | X    | 11(T) | +∞ | I(11) |

**Étape 5 : depuis le sommet I**

Depuis **I**, nous avons accès à :

M avec un coût de 4 →  $11 + 4 = 15$

Nous mettons à jour M :

| Etape          | A    | B    | N    | T    | I     | M     | Choix |
|----------------|------|------|------|------|-------|-------|-------|
| Initialisation | 0(A) | +∞   | +∞   | +∞   | +∞    | +∞    | A(0)  |
| 1              | X    | 6(A) | 2(A) | +∞   | +∞    | +∞    | N(2)  |
| 2              | X    | 6(A) | X    | 6(N) | +∞    | +∞    | B(2)  |
| 3              | X    | X    | X    | 6(N) | +∞    | +∞    | T(6)  |
| 4              | X    | X    | X    | X    | 11(T) | +∞    | I(11) |
| 5              | X    | X    | X    | X    | X     | 15(I) | M(15) |

### Conclusion

Le sommet M est atteint avec un coût total de 15.

Le chemin le plus court de A à M est donc :

**A → N → T → I → M**

Avec les coûts :

- A → N = 2
  - N → T = 4
  - T → I = 5
  - I → M = 4
- **Total = 2 + 4 + 5 + 4 = 15**

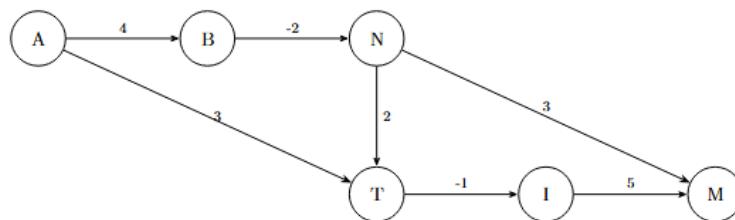
L'algorithme de Dijkstra est maintenant terminé, car le sommet d'arrivée **M** est devenu définitif.

## 1.2 Présentation de l'algorithme de Bellman-Ford :

Soit la Matrice M2 suivante :

$$M = \begin{array}{ccccccc} & A & B & N & T & I & M \\ \begin{matrix} A \\ B \\ N \\ T \\ I \\ M \end{matrix} & \left( \begin{array}{cccccc} 0 & 4 & 0 & 3 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 3 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right) \end{array}$$

Voici la représentation graphique G2 de notre matrice M2 associé :



On constate que la matrice M2 contient des valeurs négatives, contrairement à la matrice M. L'objectif est de déterminer le chemin le plus court entre le sommet A (point de départ) et le sommet M (point d'arrivée). Pour cela, nous allons utiliser l'algorithme de Bellman-Ford. Cet algorithme effectue au maximum (*nombre de sommets – 1*) itérations. Étant donné qu'il y a 6 sommets dans notre graphe, cela représente un maximum de 5 étapes.

On décide de prendre le chemin suivant : AB, AT, BN, NT, TI, IM , NM

### Initialisation

Commençons par créer un tableau, avec pour entête les sommets. Considérons qu'un sommet ayant des prédécesseurs possédant des valeurs définies (différentes de +∞) qui ne changent pas 2 étapes de suite seront définitifs.

| Etape | A | B | N | T | I | M |
|-------|---|---|---|---|---|---|
| -     | - | - | - | - | - | - |

Initialisons maintenant les valeurs :

On initialise tous les sommets avec une distance de  $+\infty$ , sauf le sommet de départ A, qui vaut 0. À chaque étape, on tente de relâcher les arêtes (c'est-à-dire améliorer les valeurs des sommets atteignables) en utilisant les distances actuelles. Un sommet est considéré comme définitif si sa valeur ne change pas pendant deux étapes consécutives.

| Etape          | A    | B         | N         | T         | I         | M         |
|----------------|------|-----------|-----------|-----------|-----------|-----------|
| Initialisation | 0(A) | $+\infty$ | $+\infty$ | $+\infty$ | $+\infty$ | $+\infty$ |

**Étape 1 :**

On suit les arêtes dans l'ordre :

- $A \rightarrow B$  (poids 4)  $\Rightarrow B = 0 + 4 = 4$
- $A \rightarrow T$  (poids 3)  $\Rightarrow T = 0 + 3 = 3$
- $B \rightarrow N$  (poids -2)  $\Rightarrow N = 4 - 2 = 2$
- $N \rightarrow T$  (poids 2)  $\Rightarrow T = \min(3 \text{ (valeur actuelle)}, 2 + 2 = 4 \text{ (nouvelle valeur)}) = 3 \Rightarrow$   
pas de changement
- $T \rightarrow I$  (poids -1)  $\Rightarrow I = 3 - 1 = 2$
- $I \rightarrow M$  (poids 5)  $\Rightarrow M = 2 + 5 = 7$
- $N \rightarrow M$  (poids 3)  $\Rightarrow M = \min(7 \text{ (valeur actuelle)}, 2 + 3 = 5 \text{ (nouvelle valeur)}) = 5 \Rightarrow$   
mise à jour

| Etape          | A    | B    | N    | T    | I    | M    |
|----------------|------|------|------|------|------|------|
| Initialisation | 0(A) | +∞   | +∞   | +∞   | +∞   | +∞   |
| Etape 1        | 0(A) | 4(A) | 2(B) | 3(A) | 2(T) | 5(N) |

**Étape 2 :**

On réapplique toutes les arêtes, mais aucun sommet ne voit sa valeur changer. Donc on s'arrête là. Ainsi le plus court chemin est  $A \rightarrow B \rightarrow N \rightarrow M$ .

| Etape          | A    | B    | N    | T    | I    | M    |
|----------------|------|------|------|------|------|------|
| Initialisation | 0(A) | +∞   | +∞   | +∞   | +∞   | +∞   |
| Etape 1        | 0(A) | 4(A) | 2(B) | 3(A) | 2(T) | 5(N) |
| Etape 2        | 0(A) | 4(A) | 2(B) | 3(A) | 2(T) | 5(N) |

## 2) Dessin d'un graphe et d'un chemin à partir de sa matrice.

### 2.1 Dessin d'un graphe

Un graphe pondéré est un graphe dans lequel chaque arête a un poids (ou un coût) associé. Ces poids peuvent représenter des distances, des temps, des coûts, ou toute autre valeur numérique utile pour modéliser une situation réelle.

#### II. Recherche et choix d'un outil adapté

Une exploration d'outils en ligne a été menée pour identifier des solutions permettant de dessiner un graphe à partir d'une matrice d'adjacence pondérée. Parmi les options disponibles, l'outil sélectionné est :

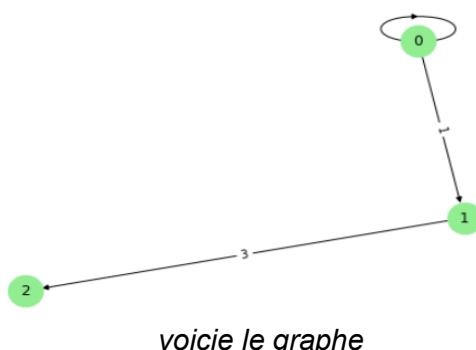
- **networkx (Python)** : une bibliothèque puissante et largement reconnue pour la modélisation, l'analyse et la visualisation de graphes. Elle propose une interface intuitive, de nombreux algorithmes intégrés et une documentation abondante.

Exemple 1 — Graphe simple orienté

```
adj_matrix = [
    [2, 1, 0],
    [0, 0, 3],
    [0, 0, 0]
]
```



Graphe à partir d'une matrice



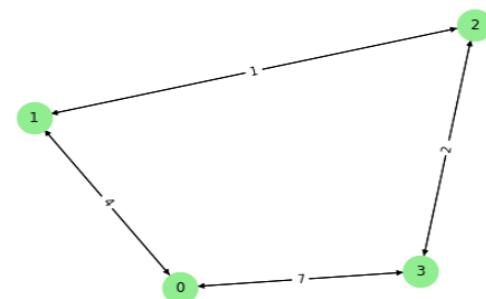
*voici le graphe*

Exemple 2 — Graphe non orienté plus complexe (4)

```
adj_matrix = [
    [0, 4, 0, 7],
    [4, 0, 1, 0],
    [0, 1, 0, 2],
    [7, 0, 2, 0]
]
```



Graphe à partir d'une matrice



*voici le graphe*

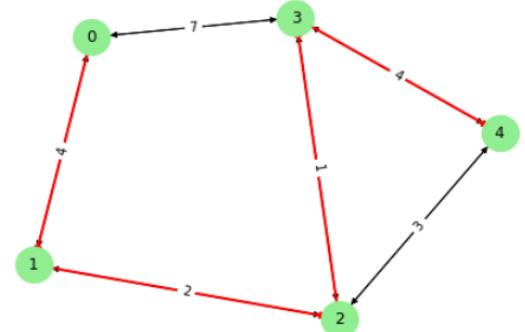
## 2.2 Dessin d'un chemin

Exemple 1 : matrice non orientée

```
adj_matrix = [
    [0, 4, 0, 7, 0],
    [4, 0, 2, 0, 0],
    [0, 2, 0, 1, 3],
    [7, 0, 1, 0, 4],
    [0, 0, 3, 4, 0]
]
```



Graphe avec chemin en rouge



*voici le graphe*

Voilà le chemin :

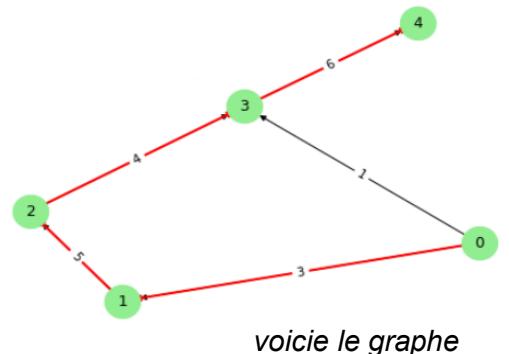
**path = [0, 1, 2, 3, 4]**

Exemple 2: graphe orienté

```
adj_matrix = [
    [0, 3, 0, 1, 0],
    [0, 0, 5, 0, 0],
    [0, 0, 0, 4, 0],
    [0, 0, 0, 0, 6],
    [0, 0, 0, 0, 0]
]
path = [0, 1, 2, 3, 4]
```



Graphe avec chemin en rouge



### 3) Génération aléatoire de matrices de graphes pondéré

#### 3.1 Graphes avec 50% de flèches

Nous allons compléter une matrice de 0 ou 1 aléatoirement.

Ensuite, on va remplacer les 0 par inf et les 1 par une valeur aléatoire entre a et b.

Voici un exemple et un affichage de la fonction graphe :

J'ai appelé la fonction `graphe(6, 1, 10)` — info : taille de 6, les 1 sont remplacés par une valeur aléatoire entre a et b.

```
Documents/UNIVERSITY 2 semestre
[[ 6. inf 2. inf 8. inf]
 [inf inf inf inf inf 1.]
 [ 5. inf inf inf 7. 8.]
 [ 3. 9. inf inf inf inf]
 [inf 7. inf 8. inf 2.]
 [ 1. 5. inf 8. inf 4.]]
```

#### 3.2 Graphes avec une proportion variables p de flèches

Ensuite, on a codé la fonction `graphe2(n, p, a, b)` en modifiant notre code précédent.

Le seul changement que l'on a fait est que, maintenant, on peut faire varier la proportion de flèches dans la matrice avec p. Ce n'est plus du 50/50.

Voici un test :

```
Documents/UNIVERSITY 2 semestre
[[ 4. 3. 3. 2. 8. 2.]
 [ 9. 8. 6. 4. 7. 1.]
 [ 8. 2. inf 5. inf 5.]
 [ 8. 4. 5. 5. 5. inf]
 [inf 3. 8. 7. 7. 7.]
 [ 2. 3. 2. 4. 7. inf]]
```

Avec `graphe2(6, 1, 10, 0.9)`, on a une majorité de flèches.

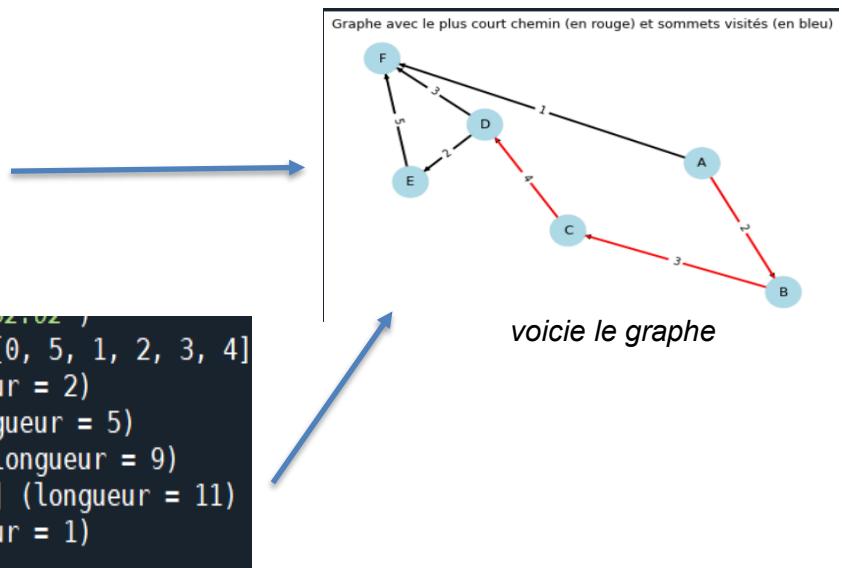
## 4) Codage des algorithmes de plus court chemin

### 4.1 Codage de l'algorithme de Dijkstra

Voici un exemple :

```
M = np.array([
    [0, 2, 0, 0, 0, 1], # A
    [0, 0, 3, 0, 0, 0], # B
    [0, 0, 0, 4, 0, 0], # C
    [0, 0, 0, 0, 2, 3], # D
    [0, 0, 0, 0, 0, 5], # E
    [0, 0, 0, 0, 0, 0] # F
])
```

```
documents/university_2/semaine/32/02
Sommets visités dans l'ordre : [0, 5, 1, 2, 3, 4]
Chemin de 0 à 1 : [0, 1] (longueur = 2)
Chemin de 0 à 2 : [0, 1, 2] (longueur = 5)
Chemin de 0 à 3 : [0, 1, 2, 3] (longueur = 9)
Chemin de 0 à 4 : [0, 1, 2, 3, 4] (longueur = 11)
Chemin de 0 à 5 : [0, 5] (longueur = 1)
```



### 4.2 Codage de l'algorithme de Bellman-Ford

Voici un exemple :

```
M = [
    [0, -1, 5, 0],
    [0, 5, -3, 0],
    [8, 0, 0, -3],
    [0, 6, 7, 0]
]
```

```
Chemin de 2 à 0 : longueur = 8, itinéraire = [2, 0]
Chemin de 2 à 1 : longueur = 3, itinéraire = [2, 3, 1]
Chemin de 2 à 3 : longueur = -3, itinéraire = [2, 3]
```

## 5) Influence du choix de la liste ordonnée des flèches pour l'algorithme de Bellman-Ford

Nous avons réalisé trois parcours : un aléatoire, un en profondeur et un en largeur.

- Pour le parcours aléatoire, on affiche directement la liste des flèches.
- Pour les fonctions pp (parcours en profondeur) et pl (parcours en largeur), on affiche le parcours sous forme de liste, par exemple [1, 2, 3, 4, 5, 6], puis on le convertit en liste de flèches à l'aide d'une fonction que nous avons créée : transforme\_parcours\_en\_fleche(M, parcours), qui prend en paramètre une matrice et un parcours.

Ensuite, au lieu de créer trois versions différentes de Bellman-Ford pour chaque type de parcours, nous avons ajouté deux paramètres à notre implémentation initiale : parcours et désignation.

Ainsi, nous passons simplement la liste des flèches en paramètre, que ce soit pour un parcours en largeur, en profondeur ou aléatoire. Le code de Bellman-Ford reste inchangé. Le paramètre désignation sert uniquement à l'affichage.

### Appel de la fonction :

```
Bellman_Ford(M, s0, fleche_liste, label="")
```

Voici un test de nos trois versions de Bellman-Ford avec un graphe de poids 50, généré aléatoirement avec 10 % de flèches.

```
== Bellman-Ford with ALEATOIRE ==
[Aleatoire] Nombre de tours effectués : 6

== Bellman-Ford with PROFONDEUR ==
[Profondeur] Nombre de tours effectués : 4

== Bellman-Ford with LARGEUR ==
[Largeur] Nombre de tours effectués : 2
```

## 6) Comparaison expérimentale des complexités

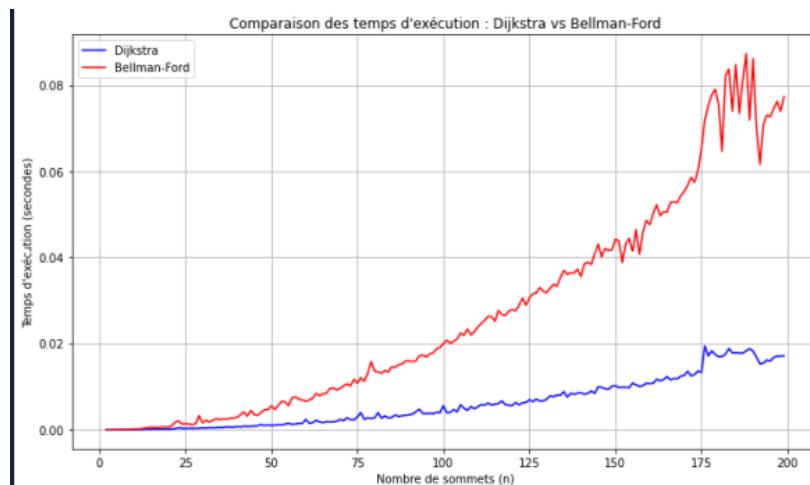
### 6.1 Deux fonctions "temps de calcul"

Pour générer aléatoirement une matrice, nous avons importé la fonction que nous avions codée à l'étape 3.2.

En fait, pour toutes nos importations et afin d'éviter d'importer chaque fonction une par une, nous avons regroupé toutes les fonctions nécessaires dans un fichier séparé.

## 6.2 Comparaison et identification des deux fonctions temps

Voici l'affichage des deux fonctions TempsDij(n) et TempsBf(n) pour n entre 2 et 200.



Nous avons comparé les performances de **Bellman-Ford** et **Dijkstra** :

- Sur de petites matrices, les deux algorithmes donnent des temps de calcul acceptables.
- Sur de grandes matrices, **Dijkstra** s'avère beaucoup plus rapide que **Bellman-Ford**.

Si une fonction  $t(n)$  est approximativement de la forme  $c * n^a$  pour  $n$  grand (croissance polynomiale d'ordre  $a$ ), alors sa représentation graphique en coordonnées log-log est approximativement une droite de pente  $a$ .

$$\log(t(n)) = \log(c * n^a)$$

$$\log(t(n)) = \log(c) + \log(n^a)$$

$$\log(t(n)) = a * \log(n) + \log(c)$$

On retrouve donc une fonction affine en  $\log(n)$ .

Notre droite a pour équation :

$$y = a * \log(n) + \log(c)$$

Ainsi, une représentation graphique en coordonnées log-log donne approximativement une droite de pente  $a$ .

Déterminons l'exposant  $a$  pour chacune des fonctions en utilisant *polyfit* de la bibliothèque *Numpy* :

```
DOCUMENTS/UNIVERSITY 2 SEMESTRE/S2.02 / 
Pente a pour Dijkstra : 1.8921180288123804
Pente a pour Bellman-Ford : 2.0257928866078783
```

Faisant ce même calcul pour  $p$  fixe avec  $p = 0.5$  et pour  $p = 1/n$  pour  $p = 0.5$  :

```
Reloaded modules: ImportationPourQuestion6
Pente a pour Dijkstra : 1.8464733956628872
Pente a pour Bellman-Ford : 1.9585205214253274
```

pour  $p = 1/n$

```
Documents/University 2 semestre/S2.02 ;
Reloaded modules: ImportationPourQuestion6
Pente a pour Dijkstra : 1.0674675851659916
Pente a pour Bellman-Ford : 1.9475418327920289
```

### 6.3 Conclusion

Pour conclure cette étude comparative des algorithmes de plus court chemin, plusieurs points importants peuvent être retenus.

- **Le choix du parcours** a un impact significatif sur l'efficacité de l'algorithme de Bellman-Ford. Nos tests montrent que les parcours structurés, comme ceux en profondeur (PP) ou en largeur (PL), donnent de meilleurs résultats que les parcours aléatoires. Cela s'explique par le fait qu'une bonne organisation des arêtes permet à Bellman-Ford de converger plus rapidement vers la solution.
- **Dijkstra** est largement plus rapide que Bellman-Ford lorsque les poids des arêtes sont tous positifs. Pour des graphes de grande taille, les différences de temps d'exécution deviennent très marquées en faveur de Dijkstra, ce qui en fait l'algorithme à privilégier dans ce cas.
- **Bellman-Ford**, en revanche, est plus général et plus robuste : il peut traiter les graphes contenant des poids négatifs, ce que Dijkstra ne peut pas faire. Dijkstra échoue en présence de poids négatifs, car il suppose que les chemins déjà explorés sont optimaux, ce qui n'est pas garanti lorsque des coûts négatifs peuvent survenir plus tard dans le parcours.

## II - Seuil de forte connexité d'un graphe orienté.

### 7) Test de forte connexité

Un graphe orienté est fortement connexe s'il existe un chemin orienté entre chaque paire de sommets  $(u,v)$ . Autrement dit, pour tout couple  $(u,v)$ , il existe un chemin de  $u$  vers  $v$ .

La fermeture transitive d'un graphe  $G$  est un graphe  $G'$  ayant le même ensemble de sommets que  $G$ , et dans lequel :

On ajoute une flèche  $(u,v)$  s'il existe un chemin de manière transitive.

Un graphe est fortement connexe si et seulement si sa matrice de fermeture transitive est remplie de 1,

car cela signifie qu'**un chemin existe entre chaque paire de sommets** dans le graphe orienté.

## 8) Forte connexité pour un graphe avec $p = 50\%$ de flèches

On a décidé de tester pour 500 graphe et pour ça on a utilisé notre fonction `graphe(n,a,b)` qu'on a coder à la question 3 et la fonction `fc(M)` à la question précédente.

Pour  $n = 11$

```
98.0
```

Pour  $n = 12$

```
99.0
```

Ainsi on peut dire que la condition est "toujours vraie" pour  $n \geq 12$ .

## 9) Détermination du seuil de forte connexité

Test de la fonction `test_stat_fc2(n,p)` pour  $n = 8$  et  $p = 1$ .

```
100.0
```

Test de la fonction `seuil(n)` pour  $n = 12$

```
0.5299999999999996
```

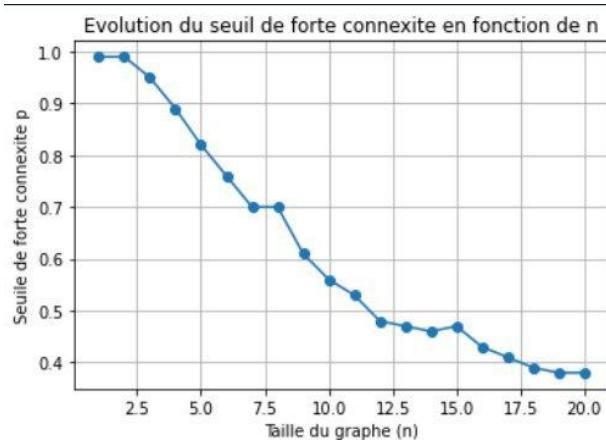
On retrouve environ 50% de flèche ainsi notre code marche.

## 10) Etude et identification de la fonction seuil

### 10.1 Représentation graphique de $\text{seuil}(n)$

L'étude proposée sur l'intervalle [10, 40] n'a pas été réalisée car la puissance de calcul de notre machine ne le permet pas (temps d'exécution trop long).

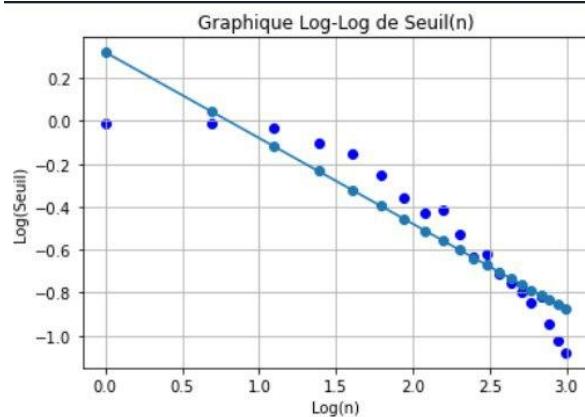
Nous avons donc limité l'intervalle à [1, 20], ce qui reste suffisant pour observer la tendance décroissante du seuil de forte connexité  $\text{seuil}(n)$ .



Oui, la courbe de seuil( $n$ ) est décroissante, comme on s'y attend. En effet, plus la taille du graphe augmente, plus il devient probable qu'il soit fortement connexe même avec une plus faible proportion de flèches. Cela s'explique par le fait qu'un grand graphe offre plus de chemins possibles entre les sommets, ce qui augmente les chances de connexité globale.

## 10.2 Identification de la fonction seuil( $n$ )

La fonction seuil( $n$ ) est asymptotiquement de type puissance car, si l'on prend le logarithme de chaque côté de l'égalité  $\text{seuil}(n) = c \times n^a$ , on obtient  $\log(\text{seuil}(n)) = \log(c) + a \times \log(n)$ , qui est l'équation d'une droite en coordonnées  $(\log(n), \log(\text{seuil}(n)))$ .



Pour vérifier cette propriété, on trace le graphique log-log de seuil( $n$ ) en fonction de  $n$  et on effectue une régression linéaire. La pente de la droite ainsi ajustée donne l'exposant  $a$  et l'ordonnée à l'origine  $\log(c)$