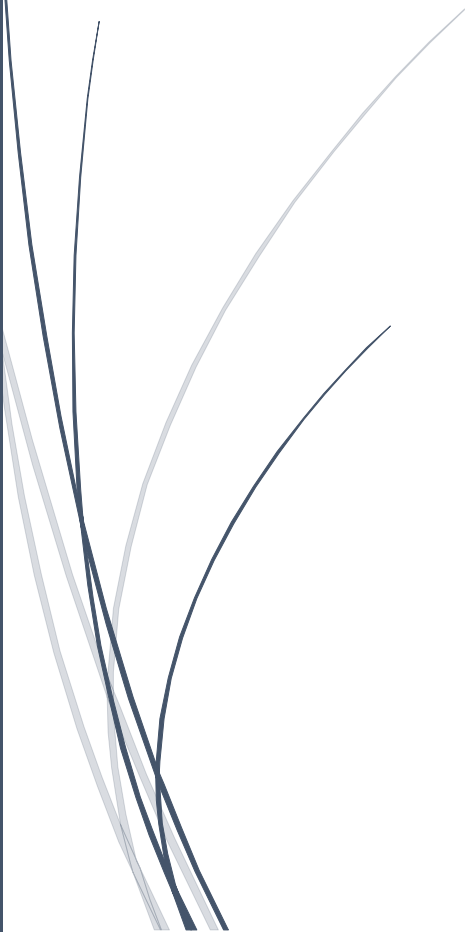


A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the date.

07/06/2023

SAE S202

Exploration algorithmique d'un problème

Several thin, curved lines in shades of blue and grey originate from the bottom left and sweep upwards and to the right.

Étudiants du groupe D2 :

- RIPICEANU Denisa
- DELAGNES Antoine
- TAMBOURA Kerima

Enseignant :

- PAUL Emmanuel

Table des matières

PARTIE I

1 Connaissance des algorithmes de plus courts chemins	2
1.1 Présentation de l'algorithme de Dijkstra	2
1.2 Présentation de l'algorithme de Bellman-Ford	3
2 Dessin d'un graphe et d'un chemin à partir de sa matrice.	4
2.1 Dessin d'un graphe.....	4
2.2 Dessin d'un chemin	5
3 Génération aléatoire de matrices de graphes pondérés.....	6
3.1 Graphes avec 50% de flèches.....	6
3.2 Graphes avec une proportion variables p de flèches	7
4 Codage des algorithmes de plus court chemin	8
4.1 Codage de l'algorithme de Dijkstra.....	8
4.2 Codage de l'algorithme de Belman-Ford	10
5 Influence du choix de la liste ordonnée des flèches pour l'algorithme de Bellman-Ford ...	13
6. Comparaison expérimentale des complexités	14
6.1 Deux fonctions "temps de calcul"	14
6.2 Comparaison et identification des deux fonctions temps	14
6.3 Conclusion	17

PARTIE II

7. Test de forte connexité	18
8. Forte connexité pour un graphe avec p=50% de fléchés.....	18
9. Détermination du seuil de forte connexité	19
10. Étude et identification de la fonction seuil	19
10.1 Représentation graphique de seuil(n)	19
10.2 Identification de la fonction seuil(n).....	20

Première partie - D2

Comparaison d'algorithmes de plus courts chemins

1 Connaissance des algorithmes de plus courts chemins

1.1 Présentation de l'algorithme de Dijkstra

Afin de se familiariser avec l'algorithme de Dijkstra, on demande ici de l'expliquer en détail sur un exemple de graphe à poids positifs, choisi par vous. L'exemple devra être suffisamment petit pour pouvoir être traité à la main, et soigneusement rédigé pour suggérer le code à effectuer.

On choisira la matrice M et le sommet de départ d de sorte que, selon le sommet final s utilisé on obtienne les deux types de réponses : "existence et affichage d'un plus court chemin", ou "pas de chemin de d vers s".

Voici la matrice M :

	a	b	c	d	e	f	g
a	$+\infty$	$+\infty$	$+\infty$	$+\infty$	2	4	$+\infty$
b	$+\infty$	0	8	$+\infty$	$+\infty$	$+\infty$	10
c	6	$+\infty$	$+\infty$	12	$+\infty$	$+\infty$	$+\infty$
d	$+\infty$	1	$+\infty$	$+\infty$	10	3	$+\infty$
e	$+\infty$	$+\infty$	4	$+\infty$	$+\infty$	$+\infty$	$+\infty$
f	$+\infty$	$+\infty$	5	$+\infty$	1	$+\infty$	19
g	$+\infty$	$+\infty$	$+\infty$	3	$+\infty$	2	$+\infty$

	a	b	c	d	e	f	g
(dist,pred) Initialisation : a	(0, a)	($+\infty$, \emptyset)	($+\infty$, \emptyset)	($+\infty$, \emptyset)	(2, a)	(4, a)	($+\infty$, \emptyset)
	x	(19, d)	(6, e)	(18, c)	x	x	(23, f)
		x	x	x			x
Résultat	(0, a)	(19, d)	(6, e)	(18, c)	(2, a)	(4, a)	(23, f)

On choisit comme point de départ le sommet 'a', donc on l'ajoute à la liste de sommets visités.

A = {a}

Parmi les sommets hors A, je sélectionne le sommet 'e' car dist(e) est minimale et je l'ajoute dans A.

A = {a, e} Successeurs de a hors de A : c

$2 + 4 = \text{dist}(e) + \text{poids}(e, c) < \text{dist}(c) = +\infty$

Parmi les sommets hors A, je sélectionne le sommet 'f' car dist(f) est minimale et je l'ajoute dans A.

A = {a, e, f} Successeurs de f hors de A : c, g

$4 + 5 = \text{dist}(f) + \text{poids}(f, c) \geq \text{dist}(c) = 6$

$4 + 19 = \text{dist}(f) + \text{poids}(f, g) < \text{dist}(g) = +\infty$

Parmi les sommets hors A, je sélectionne le sommet 'c' car dist(c) est minimale et je l'ajoute dans A.

A = {a, e, f, c} Successeurs de c hors de A : d

$6 + 12 = \text{dist}(c) + \text{poids}(c, d) < \text{dist}(d) = +\infty$

Parmi les sommets hors A, je sélectionne le sommet 'd' car dist(d) est minimale et je l'ajoute dans A.

A = {a, e, f, c, d} Successeurs de d hors de A : b

$18 + 1 = \text{dist}(d) + \text{poids}(d, b) < \text{dist}(b) = +\infty$

Parmi les sommets hors A, je sélectionne le sommet 'b' car $\text{dist}(b)$ est minimale et je l'ajoute dans A.

$A = \{a, e, f, c, d, b\}$ Successeurs de b hors de A : g

$$19 + 10 = \text{dist}(b) + \text{poids}(b, g) \geq \text{dist}(g) = 23$$

$A = \{a, e, f, c, d, b, g\}$ Successeurs de g hors de A : aucun

Le plus court chemin de a à g :

Longueur : 23

Chemin : $g \leftarrow f \leftarrow a : [a, f, g]$

Le plus court chemin de a à b :

Longueur : 19

Chemin : $b \leftarrow d \leftarrow c \leftarrow e \leftarrow a : [a, e, c, d, b]$

1.2 Présentation de l'algorithme de Bellman-Ford

Idem. On choisira ici un exemple à poids de signes variables, de sorte que, selon le choix du sommet final s l'algorithme retourne les trois possibilités : affichage du plus court chemin de d vers s, ou la mentions "pas de chemin de d vers s" ou la mention "existence de chemins de d vers s mais pas de plus court chemin (présence d'un cycle de poids négatif)".

Voici la matrice M :

	a	b	c	d	e	f
a	$+\infty$	9	$+\infty$	$+\infty$	$+\infty$	$+\infty$
b	$+\infty$	$+\infty$	4	$+\infty$	-2	$+\infty$
c	$+\infty$	-1	$+\infty$	$+\infty$	$+\infty$	-1
d	1	$+\infty$	$+\infty$	$+\infty$	8	$+\infty$
e	$+\infty$	$+\infty$	$+\infty$	-2	$+\infty$	$+\infty$
f	-1	$+\infty$	$+\infty$	0	$+\infty$	$+\infty$

Flèches	Tour 1	Tour 2
da	$0 + 2 = \text{dist}(d) + \text{poids}(da) < \text{dist}(a) = +\infty$	$0 + 2 \geq 2$
de	$0 + 8 < +\infty$	$0 + 8 \geq 8$
ab	$2 + 9 < +\infty$	$2 + 9 \geq 11$
bc	$11 + 4 < +\infty$	$11 + 4 \geq 15$
be	$11 + (-2) \geq 8$	$11 + (-2) \geq 8$
cb	$15 + (-1) \geq 11$	$15 + (-1) \geq 11$
cf	$15 + (-1) < +\infty$	$15 + (-1) \geq 14$
ed	$8 + (-3) < +\infty$	$8 + (-3) \geq 0$
fa	$14 + (-1) \geq 2$	$14 + (-1) \geq 2$
fd	$14 + 0 \geq 0$	$14 + 0 \geq 0$

On choisit comme point de départ le sommet 'd'. Donc, on initialise les variables (dist, pred) du sommet 'd' à (0,d) et pour les autres sommets à $(+\infty, \emptyset)$.

	a	b	c	d	e	f
(dist, pred) Initialisation : d	$(+\infty, \emptyset)$	$(+\infty, \emptyset)$	$(+\infty, \emptyset)$	(0, d)	$(+\infty, \emptyset)$	$(+\infty, \emptyset)$
Tour 1	(2, d)	(11, a)	(15, b)		(8, d)	(14, c)
Tour 2						

On constate que lors du deuxième tour, il n'y a aucune modification. Donc, on arrête l'algorithme.

Le plus court chemin de d à b :

Longueur : 11

Chemin : $b \leftarrow a \leftarrow d$: [d, a, b]

Le plus court chemin de d à c :

Longueur : 15

Chemin : $c \leftarrow b \leftarrow a \leftarrow d$: [d, a, b, c]

Le plus court chemin de d à f :

Longueur : 14

Chemin : $f \leftarrow c \leftarrow b \leftarrow a \leftarrow d$: [d, a, b, c, f]

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

2.1 Dessin d'un graphe

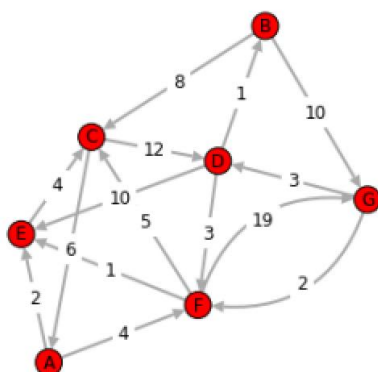
À partir d'une recherche sur internet, trouver un outil, de préférence en Python qui, à partir de la matrice d'un graphe pondéré donne un dessin de ce graphe. Présenter des exemples.

On a choisi d'installer igraph parce c'est un outil qui propose des fonctionnalités de visualisation puissantes pour représenter graphiquement des graphes. Il offre de nombreuses options de personnalisation pour ajuster l'apparence des sommets, des arêtes, des étiquettes, etc. Il permet également de changer les couleurs des graphes et d'afficher les points et les flèches en rouge pour le plus court chemin. Les flèches sont arrondies pour plus de visibilité. En plus, il peut utiliser différents formats de graphe, tels que les listes d'adjacence, les matrices d'adjacence, ce qui facilite l'importation et l'exportation de données graphiques. L'installation était difficile mais les fonctionnalités sont nombreuses.

La fonction se trouve dans le fichier **DessinGraphe.py**

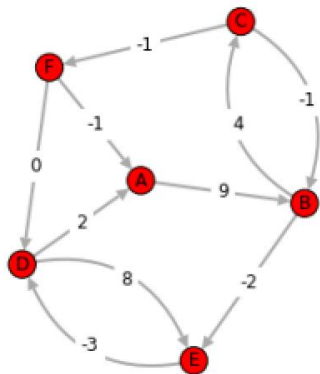
Exemples de graphes :

Le graphe de la partie 1.1



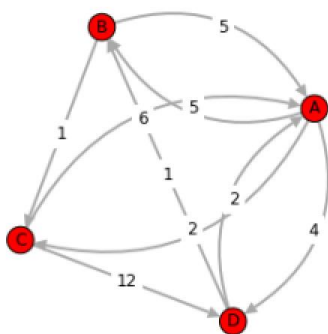
∞	∞	∞	∞	2	4	∞
∞	∞	8	∞	∞	∞	10
6	∞	∞	12	∞	∞	∞
∞	1	∞	∞	10	3	∞
∞	∞	4	∞	∞	∞	∞
∞	∞	5	∞	1	∞	19
∞	∞	∞	3	∞	2	∞

Le graphe de la partie 1.2

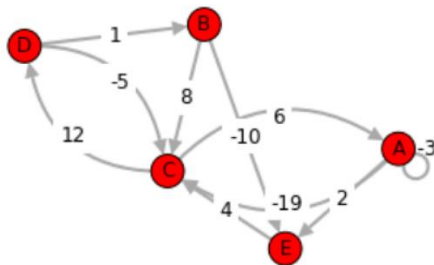


∞	9	∞	∞	∞	∞
∞	∞	4	∞	-2	∞
∞	-1	∞	∞	∞	-1
2	∞	∞	∞	8	∞
∞	∞	∞	-3	∞	∞
-1	∞	∞	0	∞	∞

Autres exemples :



∞	5	2	4
5	∞	1	∞
6	∞	∞	12
2	1	∞	∞



-3	∞	-19	∞	2
∞	∞	8	∞	-10
6	∞	∞	12	∞
∞	1	-5	∞	∞
∞	∞	4	∞	∞

2.2 Dessin d'un chemin

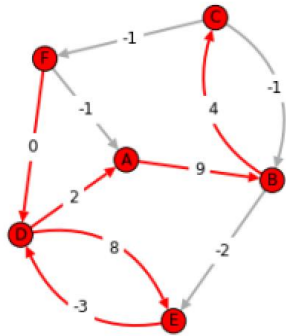
Améliorer cet outil afin qu'étant donné un chemin donné par sa suite de sommets visités, celui-ci s'affiche en rouge sur le graphe.

Présenter des exemples.

La fonction se trouve dans le fichier `affichageRouge.py`.

Exemples de graphes :

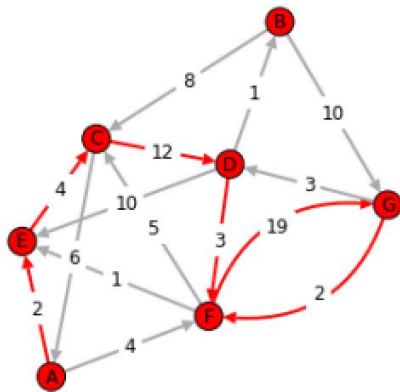
Le graphe de la partie 1.2



∞	9	∞	∞	∞	∞
∞	∞	4	∞	-2	∞
∞	-1	∞	∞	∞	-1
2	∞	∞	∞	8	∞
∞	∞	∞	-3	∞	∞
-1	∞	∞	0	∞	∞

chemin = ['A', 'F', 'D', 'E', 'D', 'A', 'B', 'C']

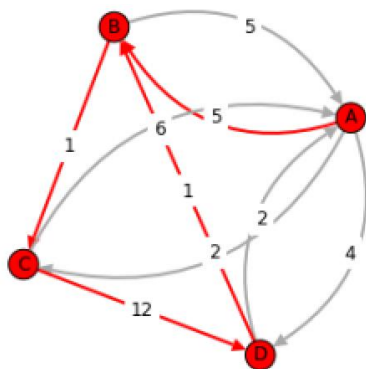
Le graphe de la partie 1.1



∞	∞	∞	∞	2	4	∞
∞	∞	8	∞	∞	∞	10
6	∞	∞	12	∞	∞	∞
∞	1	∞	∞	10	3	∞
∞	∞	4	∞	∞	∞	∞
∞	∞	5	∞	1	∞	19
∞	∞	∞	3	∞	2	∞

chemin = ['A', 'E', 'C', 'D', 'F', 'G', 'F']

Autre exemple :



∞	5	2	4
5	∞	1	∞
6	∞	∞	12
2	1	∞	∞

chemin = ['A', 'B', 'C', 'D', 'B']

3 Génération aléatoire de matrices de graphes pondérés

3.1 Graphes avec 50% de flèches

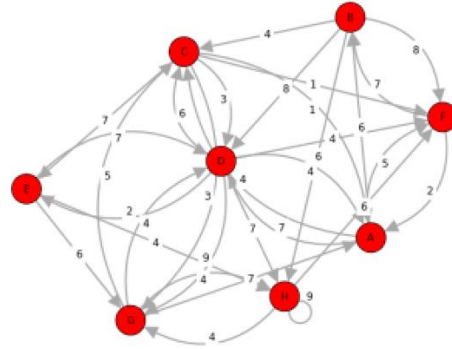
Construire une fonction Python `graphe(n,a,b)` qui prend en entrée un entier strictement positif n , deux entiers a et b , et qui génère aléatoirement une matrice de taille $n \times n$ présentant environ 50% de coefficients ∞ et 50% de coefficients avec des poids entiers dans l'intervalle $[a, b]$. Présenter des exemples et leurs représentations graphiques.

La fonction se trouve dans le fichier `graphe.py`.

Exemples :

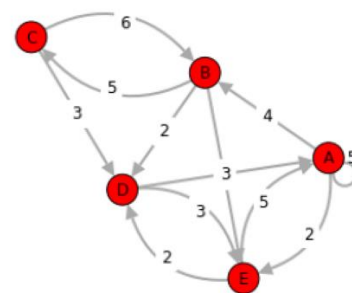
Matrice 8x8 présentant environ 50% de coefficients ∞ et 50% de coefficients avec des poids entiers dans l'intervalle $[0, 10[$.

```
[[inf 6. 4. 7. inf 5. inf inf]
 [inf inf 4. 8. inf 8. inf 6.]
 [1. inf inf 3. 7. 1. 3. inf]
 [4. inf 6. inf 2. 4. 9. 7.]
 [inf inf inf 7. inf inf 6. 4.]
 [2. 7. inf inf inf inf inf inf]
 [7. inf 5. 4. inf inf inf 4.]
 [inf inf inf inf inf 6. 4. 9.]]
```



Matrice 5x5 présentant environ 50% de coefficients ∞ et 50% de coefficients avec des poids entiers dans l'intervalle $[2, 7[$.

```
[[ 5. 4. inf inf 2.]
 [inf inf 5. 2. 2.]
 [inf 6. inf 3. inf]
 [ 3. inf inf inf 3.]
 [ 5. inf inf 2. inf]]
```



3.2 Graphes avec une proportion variables p de flèches

On souhaite pouvoir faire varier la proportion p de flèches (c'est-à-dire de 1 dans la matrice). Modifier la fonction précédente en une fonction `graphe2(n,p,a,b)` qui génère aléatoirement la matrice d'un graphe à n sommets avec une proportion p (pouvant varier de 0 à 1) de flèches (coefficients différents de l'infini). Présenter des exemples.

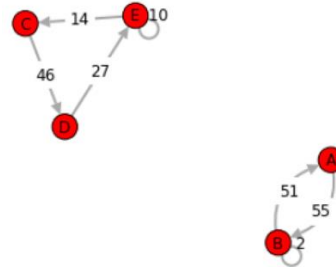
Suggestion. Utiliser la fonction binomiale a la place de `randint`.

La fonction se trouve dans le fichier `graphe.py`.

Exemples :

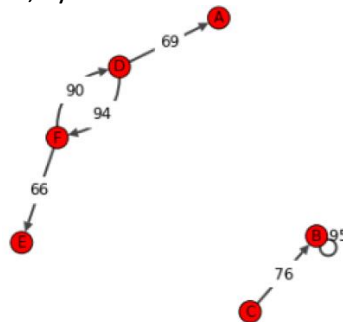
Matrice de taille 5 avec une proportion de flèches de 30%, ayant des valeurs entre 0 et 60

```
[[inf 55. inf inf inf]
 [51. 2. inf inf inf]
 [inf inf inf 46. inf]
 [inf inf inf inf 27.]
 [inf inf 14. inf 10.]]
```



Matrice de taille 6 avec une proportion de flèches de 20%, ayant des valeurs entre 1 et 100

```
[[inf inf inf inf inf inf]
 [inf 95. inf inf inf inf]
 [inf 76. inf inf inf inf]
 [69. inf inf inf inf 94.]
 [inf inf inf inf inf inf]
 [inf inf inf 90. 66. inf]]
```



4 Codage des algorithmes de plus court chemin

4.1 Codage de l'algorithme de Dijkstra

Créer une fonction Python `Dijkstra(M,d)` qui prend en entrée la matrice d'un graphe pondéré à poids positifs, un sommet `d` de ce graphe donné par son indice dans la liste des sommets, et qui, en exécutant l'algorithme de Dijkstra, retourne pour chacun des autres sommets `s` :

- soit la longueur et l'itinéraire du plus court chemin de `d` à `s` ;
- soit la mention "sommet non joignable à `d` par un chemin dans le graphe `G`". On codera à partir du pseudocode présenté dans le cours.

On pourra retourner le résultat sous forme de la liste des sommets du chemin obtenu, mais aussi en l'affichant graphiquement à l'aide de l'outil de §2.

Tester sur vos exemples du §1, et sur des exemples générés aléatoirement.

La fonction se trouve dans le fichier `Dijkstra.py` .

On a implémenté une fonction `Dijkstra(M,d)` qui prend en entrée la matrice d'un graphe pondéré à poids positifs, un sommet `d` de ce graphe donné par son indice dans la liste des sommets, et qui, en exécutant l'algorithme de Dijkstra, retourne pour chacun des autres sommets `s` :

- soit la longueur et l'itinéraire du plus court chemin de `d` à `s` ;
- soit la mention "sommet de départ" pour le chemin de `d` à `d`
- soit la mention "sommet non joignable à `d` par un chemin dans le graphe `G`". On codera à partir du pseudo code présenté dans le cours.

Mais, on a implémenté aussi une fonction `Dijkstra1(M,d,a)` qui prend en entrée la matrice d'un graphe pondéré à poids positifs, un sommet de départ de ce graphe donné par son indice dans la liste des sommets, un sommet d'arrivée de ce graphe donné par son indice dans la liste des sommets et qui, en exécutant l'algorithme de Dijkstra, retourne le plus court chemin entre le sommet de départ et celui d'arrivée et l'affiche en rouge s'il existe.

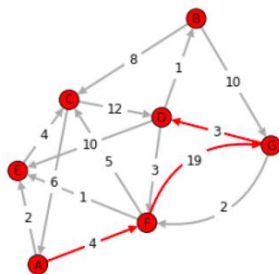
Pour ces deux programmes, on a suivi l'algorithme en utilisant une bibliothèque pour les distances entre les points, une bibliothèque pour les liens entre les points qui sont liés dans le chemin (principe de prédécesseurs) et une variable pour déterminer le prochain point du chemin.

Exemples :

Le graphe de la partie 1.1

∞	∞	∞	∞	2	4	∞
∞	∞	8	∞	∞	∞	10
6	∞	∞	12	∞	∞	∞
∞	1	∞	∞	10	3	∞
∞	∞	4	∞	∞	∞	∞
∞	∞	5	∞	1	∞	19
∞	∞	∞	3	∞	2	∞

Le plus court chemin de A vers D :

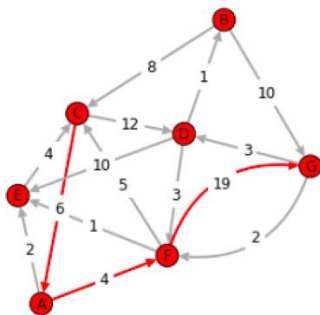


La longueur : 26

Le chemin : [A, F, G, D]

(26.0, 'A -> F -> G -> D')

Le plus court chemin de C vers G :



La longueur : 29

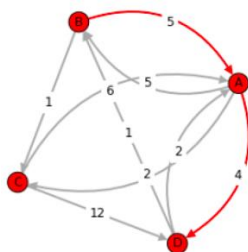
Le chemin : [C, A, F, G]

(29.0, 'C -> A -> F -> G')

Autre exemple

∞	5	2	4
5	∞	1	∞
6	∞	∞	12
2	1	∞	∞

Le plus court chemin de B vers D :



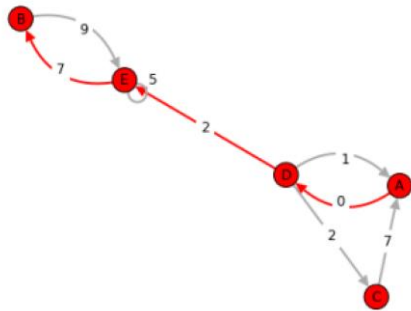
La longueur : 9

Le chemin : [B, A, D]

(9.0, 'B -> A -> D')

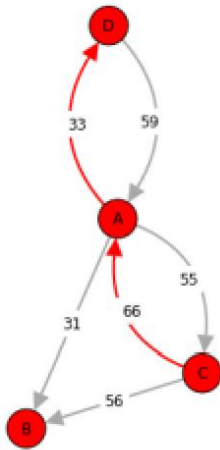
Exemples de graphes générés aléatoirement :

Matrice de taille 5 avec une proportion de flèches de 40%, ayant des valeurs entre 0 et 10
Le plus court chemin de A à B :



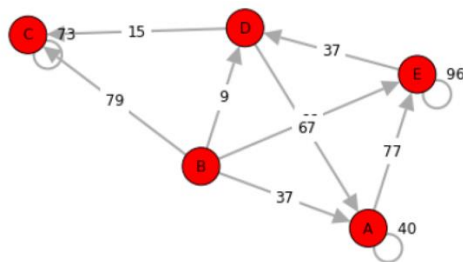
```
[[inf inf inf 0. inf]
 [inf inf inf inf 9.]
 [ 7. inf inf inf inf]
 [ 1. inf 2. inf 2.]
 [inf 7. inf inf 5.]]
(9.0, 'A -> D -> E -> B')
```

Matrice de taille 4 avec une proportion de flèches de 30%, ayant des valeurs entre 0 et 100
Le plus court chemin de C à D :



```
[[inf 31. 55. 33.]
 [inf inf inf inf]
 [66. 56. inf inf]
 [59. inf inf inf]]
(99.0, 'C -> A -> D')
```

Matrice de taille 5 avec une proportion de flèches de 40%, ayant des valeurs entre 0 et 100
Le plus court chemin de C à D :



```
[[40. inf inf inf 77.]
 [37. inf 79. 9. 66.]
 [inf inf 73. inf inf]
 [67. inf 15. inf inf]
 [inf inf inf 37. 96.]]
```

Le plus court chemin :
('Pas de chemin entre', 'C', 'et', 'D')

4.2 Codage de l'algorithme de Belman-Ford

Créer une fonction Python Bellman-Ford(M,d) qui prend en entrée la matrice d'un graphe pondéré à poids de signe quelconque, un sommet d de ce graphe donné par son indice dans la liste des sommets, et qui, en exécutant l'algorithme de Bellman-Ford, retourne pour chacun des autres sommets s :

- soit la longueur et l'itinéraire du plus court chemin de d à s ;
- soit la mention "sommet non joignable depuis d par un chemin dans le graphe G".
- soit la mention "sommet joignable depuis d par un chemin dans le graphe G, mais pas de plus court chemin (présence d'un cycle n négatif)".

On codera à partir du pseudocode présenté dans le cours.

On pourra retourner le résultat sous forme de la liste des sommets du chemin obtenu, mais aussi en l'affichant graphiquement à l'aide de l'outil de §2.

Tester sur vos exemples du §1, et sur des exemples généraux aléatoirement avec des poids de signes variables.

La fonction se trouve dans le fichier [BellmanFord.py](#).

On a implémenté une fonction `BellmanFord(M,d)` qui prend en entrée la matrice d'un graphe pondéré à poids positifs, un sommet `d` de ce graphe donné par son indice dans la liste des sommets, et qui, en exécutant l'algorithme de Bellman-Ford, retourne pour chacun des autres sommets `s` :

- soit la longueur et l'itinéraire du plus court chemin de `d` à `s` ;
- soit la mention "sommet non joignable depuis `d` par un chemin dans le graphe `G`".
- soit la mention "sommet joignable depuis `d` par un chemin dans le graphe `G`, mais pas de plus court chemin (présence d'un cycle `n` négatif)".

Mais, on a implémenté aussi une fonction `BellmanFord1(M,d,a)` qui prend en entrée la matrice d'un graphe pondéré à poids positifs, un sommet de départ de ce graphe donné par son indice dans la liste des sommets, un sommet d'arrivée de ce graphe donné par son indice dans la liste des sommets et qui, en exécutant l'algorithme de Bellman-Ford, retourne le plus court chemin entre le sommet de départ et celui d'arrivée et l'affiche en rouge s'il existe.

Pour ces deux programmes, on a suivi l'algorithme vu en cours en utilisant des bibliothèque pour les distances entre les points, les liens entre les points qui sont liés dans le chemin (principe de prédécesseurs), pour prendre en mémoire les flèches et leur poids, pour le résultat et pour les cycles à poids négatifs qui changent en permanence.

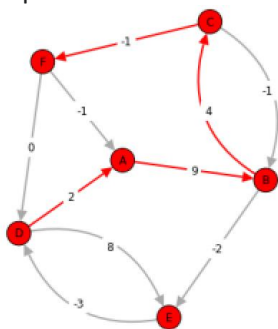
Le graphe de la partie 1.2

∞	9	∞	∞	∞	∞
∞	∞	4	∞	-2	∞
∞	-1	∞	∞	∞	-1
2	∞	∞	∞	8	∞
∞	∞	∞	-3	∞	∞
-1	∞	∞	0	∞	∞

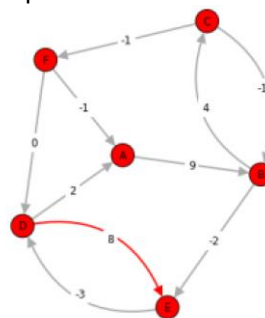
Les plus courts chemins de D vers les autres sommets :

```
(2.0, 'D -> A')
(11.0, 'D -> A -> B')
(15.0, 'D -> A -> B -> C')
('Sommet de départ', 'D')
(8.0, 'D -> E')
(14.0, 'D -> A -> B -> C -> F')
```

Le plus court chemin de D vers F :



Le plus court chemin de D à E :



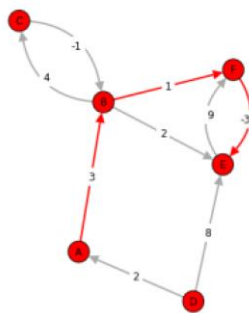
Autre exemple

∞	3	∞	∞	∞	∞
∞	∞	4	∞	2	1
∞	-1	∞	∞	∞	∞
2	∞	∞	∞	8	∞
∞	∞	∞	∞	∞	9
∞	∞	∞	∞	-3	∞

Les plus courts chemins de B vers les autres sommets :

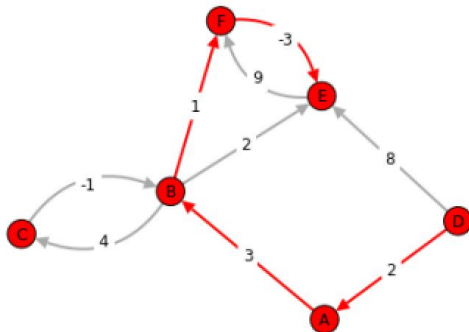
('Sommets non joignables depuis', 'B', 'par un chemin dans le graphe G')
('Sommets de départ', 'B')
(4.0, 'B → C')
('Sommets non joignables depuis', 'B', 'par un chemin dans le graphe G')
(-2.0, 'B → F → E')
(1.0, 'B → F')

Le plus court chemin de A vers E :



Plus court chemin trouvé !
Longueur : 1.0
Itinéraire : A → B → F → E

Le plus court chemin de D à E :

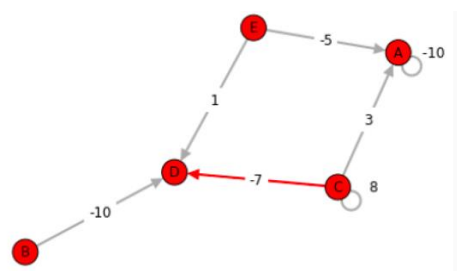


Plus court chemin trouvé !
Longueur : 3.0
Itinéraire : D → A → B → F → E

Exemples de graphes générés aléatoirement :

Matrice de taille 5 avec une proportion de flèches de 40%, ayant des valeurs entre -10 et 10

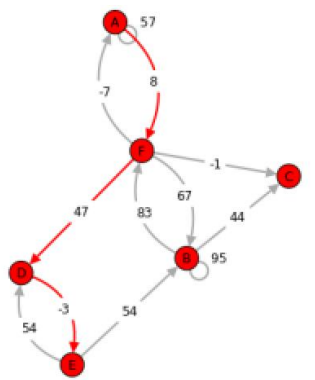
Le plus court chemin de C à D :



```
[[ -10.  inf  inf  inf  inf]
[  inf  inf  inf -10.  inf]
[   3.  inf   8.  -7.  inf]
[  inf  inf  inf  inf  inf]
[ -5.  inf  inf   1.  inf]]
```

Plus court chemin trouvé !
Longueur : -7.0
Itinéraire : C → D

Matrice de taille 6 avec une proportion de flèches de 30%, ayant des valeurs entre -15 et 100
 Le plus court chemin de A à E :



```
[[57. inf inf inf inf 8.]
 [inf 95. 44. inf inf 83.]
 [inf inf inf inf inf inf]
 [inf inf inf inf -3. inf]
 [inf 54. inf 54. inf inf]
 [-7. 67. -1. 47. inf inf]]
```

Plus court chemin trouvé !
 Longueur : 52.0
 Itinéraire : A -> F -> D -> E

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

On peut choisir cette liste ordonnée des flèches de plusieurs manières :

- De manière arbitraire, par choix aléatoire ;
- Par un ordre choisi "en largeur" : un parcours en largeur donne une liste ordonnée de sommets depuis le départ d. On prend toutes les flèches issues de ces sommets.
- Par un ordre choisi "en profondeur" : un parcours en profondeur donne une liste ordonnée de sommets depuis d. On prend toutes les flèches issues de ces sommets.

On voudrait vérifier ici si cela influence fortement ou non le temps de calcul de l'algorithme de Bellman-Ford.

- Modifier le code de Bellman-Ford en 3 variantes selon le mode de construction de la liste des flèches.
- Rajouter dans le code un compteur afin d'afficher le nombre de tours effectués.
- Comparer les résultats sur un même graphe "grand" (par exemple 50 sommets) généré aléatoirement, et conclure.

Les trois variantes de l'algorithme de Bellman-Ford selon le mode de construction de la liste des flèches se trouvent dans le fichier **ChoixBellmanFord**.

On voit que, pour un même graphe "grand" (par exemple 50 sommets) généré aléatoirement, l'algorithme le plus efficace est celui de parcours en **largeur**.

Matrice de taille 50x50 avec une proportion de flèches de 20% avec des valeurs entre -1 et 100.

Nombre de tours par choix arbitraire: 7

Nombre de tours parcours en profondeur : 5

Nombre de tours parcours en largeur: 4

Matrice de taille 50x50 avec une proportion de flèches de 40% avec des valeurs entre -3 et 20.

Nombre de tours par choix arbitraire: 50

Nombre de tours parcours en profondeur : 50

Nombre de tours parcours en largeur: 50

Matrice de taille 50x50 avec une proportion de flèches de 20% avec des valeurs entre -1 et 20.

Nombre de tours par choix arbitraire: 8

Nombre de tours parcours en profondeur : 8

Nombre de tours parcours en largeur: 4

Matrice de taille 25x25 avec une proportion de flèches de 20% avec des valeurs entre -1 et 20.

Nombre de tours par choix arbitraire: 5

Nombre de tours parcours en profondeur : 4

Nombre de tours parcours en largeur: 3

6. Comparaison expérimentale des complexités

On veut connaître expérimentalement la croissance du temps de calcul de chacun des deux algorithmes en fonction du nombre n de sommets. On reprend la même démarche que pour la comparaison des deux algorithmes de calcul de fermeture transitive, étudiée au R207 (exercices 12 et 18).

6.1 Deux fonctions "temps de calcul"

Créer une fonction Python TempsDij(n) qui prend en entrée un entier positif n et qui :

- Génère aléatoirement une matrice d'un graphe pondéré à poids positifs de taille n ;
- Calcule tous les plus courts chemins depuis le premier sommet vers tous les autres sommets par l'algorithme de Dijkstra (sans affichage du résultat) ;
- Retourne le temps de calcul utilisé.

La fonction se trouve dans le fichier **ComparaisonComplexités**.

Créer une fonction Python similaire TempsBF (n) qui utilise l'algorithme de Bellman-Ford, avec le choix de la liste de flèches le plus efficace, déterminé à la question précédente.

La fonction se trouve dans le fichier **ComparaisonComplexités**.

6.2 Comparaison et identification des deux fonctions temps

- Afficher sur un même graphique les représentations de ces deux fonctions TempsDij(n) et TempsBF (n) pour $n = 2, \dots, 100$. Quel algorithme est le plus rapide ?

La fonction se trouve dans le fichier **ComparaisonComplexités**.

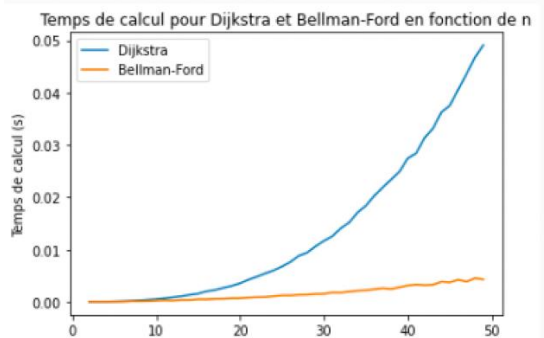
On voit sur le graphique que l'algorithme de Bellman-Ford, celui qui choisit la liste ordonnée des flèches par un ordre choisi "en largeur", est plus rapide que Dijkstra.

Bellman-Ford est une droite croissante avec un faible coefficient directeur, ce qui signifie que le temps d'exécution de Bellman-Ford augmente peu en fonction de la taille de la matrice.

Dijkstra est une parabole qui augmente vite.

C'est contrairement à ce qu'on s'attendait. On avait l'impression, faisant à la main, que l'algorithme de Dijkstra est plus rapide que celui de Bellman-Ford. Mais, on a obtenu le contraire avec deux fonctions. Après avoir vu les résultats sur le graphique, on a revu la fonction de Dijkstra qu'on a codée et on s'est rendu compte que ce résultat inattendu était dû à une approche de codage qui n'était pas optimale. Mais, on n'a pas eu le temps de la changer.

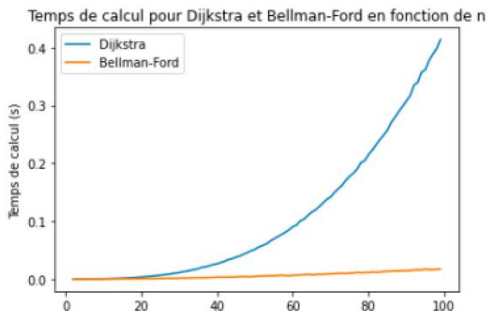
n = 50



Temps de calcul algorithmeⁿ de Dijkstra: 0.05384754199985764

Temps de calcul algorithme de Bellman-Ford: 0.004557749998639338

n = 100



Temps de calcul algorithme de Dijkstra: 0.42777337500228896

Temps de calcul algorithme de Bellman-Ford: 0.02030654199916171

- Expliquer pourquoi si une fonction $t(n)$ est approximativement de type cn^a pour n grand (croissance polynomiale d'ordre a) alors sa représentation graphique en coordonnées log-log est approximativement une droite de pente a .

Lorsqu'une fonction $t(n)$ est approximativement de type $c \times n^a$ où c est une constante, n est un grand nombre et a est l'ordre de croissance polynomiale, cela veut dire que l'on a un temps d'exécution pour cette fonction qui est proportionnel à la puissance du nombre n . Donc, si n est plus grand, le temps d'exécution de $t(n)$ est plus grand également avec une fonction polynomiale de n puissance a .

En représentant graphiquement la fonction en coordonnées logarithmiques, on prend le logarithme du temps d'exécution de $t(n)$ et le logarithme de n . On peut donc obtenir une équation linéaire à partir de la fonction polynomiale ce qui nous permet d'observer le résultat plus clairement.

En utilisant le logarithme dans l'équation $t(n) = c \times n^a$, on obtient :
 $\log(t(n)) = \log(c \times n^a)$

On peut donc transformer le résultat en : $\log(t(n)) = \log(c) + a \times \log(n)$

L'équation est : $y = mx + c$, où y est $\log(t(n))$,
 x est $\log(n)$,

m est l'ordre de croissance polynomiale a ,
 c est $\log(c)$.

La représentation graphique de la fonction est une droite de pente $c = a$. L'ordre de croissance polynomiale de t est donc proportionnel à la pente en logarithme.

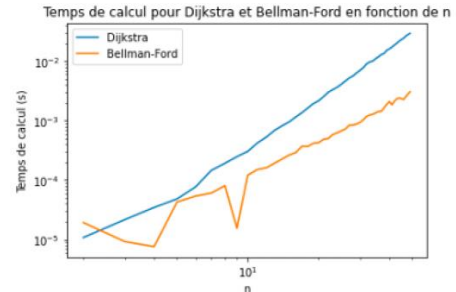
Ainsi, si une fonction $t(n)$ est approximativement de type $c \times n^a$ pour n grand, sa représentation graphique en coordonnées log-log sera approximativement une droite avec une pente. Cela permet d'identifier facilement l'ordre de croissance polynomiale de la fonction en observant la pente de la ligne sur le graphique log-log.

- Vérifier que les deux fonctions temps sont polynomiales et déterminer pour chacune d'elle l'exposant a .

Pour déterminer si les fonctions temps sont polynomiales et trouver l'exposant a , on doit regarder la forme de la courbe sur le graphique. Si la courbe est une ligne droite sur une échelle log-log, cela indique une relation de puissance et signifie que la fonction est polynomiale. L'exposant a correspond à la pente de la droite sur l'échelle log-log.

Dans ces graphiques, on observe que l'algorithme de Bellman-Ford est toujours plus rapide que l'algorithme de Dijkstra, on observe également que les droites représentant les temps d'exécution des deux algorithmes ne sont pas constantes même dans le graphe sous forme logarithmique, ce sont des graphes de forme $ax+b$, ce qui signifie que l'évolution du temps d'exécution en fonction de la taille des matrices est de forme polynomiale pour les deux programmes.

$n = 50$

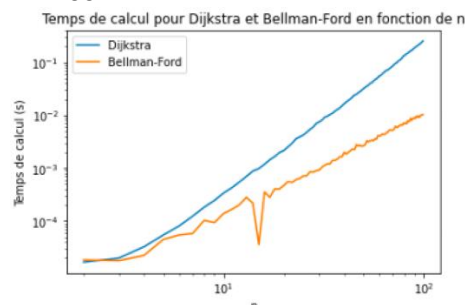


Temps de calcul algorithme de Dijkstra : 0.037847916999453446
 Temps de calcul algorithme de Bellman-Ford : 0.003851834000670351

Algorithme de Dijkstra :
 Coefficient directeur de la courbe pour x^2 : 1.8829235888783708e-05
 Coefficient directeur de la courbe pour x : 0.0006012616297720023

Algorithme de Bellman-Ford :
 Coefficient directeur de la courbe pour x^2 : 1.1482596763603104e-06
 Coefficient directeur de la courbe pour x : 5.813778441687031e-05

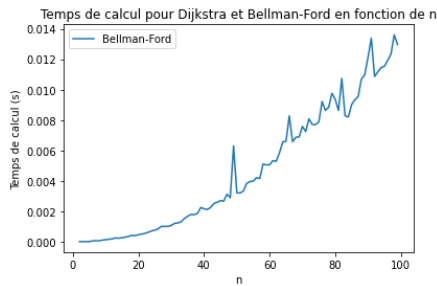
$n = 100$



Temps de calcul algorithme de Dijkstra : 0.3005726669998694
 Temps de calcul algorithme de Bellman-Ford : 0.012111042000469752

Algorithme de Dijkstra :
 Coefficient directeur de la courbe pour x^2 : 3.981967325432414e-05
 Coefficient directeur de la courbe pour x : 0.002372856700424393

Algorithme de Bellman-Ford :
 Coefficient directeur de la courbe pour x^2 : 1.3527638314099756e-06
 Coefficient directeur de la courbe pour x : 0.00012045497555097786



On a décidé d'afficher uniquement Bellman-Ford sur un graphe à part sans logarithme pour vérifier que la courbe était bien polynomiale et pas linéaire comme le 1er graphe laissait penser (puisque la courbe de Dijkstra écrasait celle de Bellman-Ford).

6.3 Conclusion

— Résumer les résultats de comparaison obtenus.

On a comparé l'algorithme de Dijkstra avec le plus efficace algorithme de Bellman-Ford, celui qui choisit la liste ordonnée des flèches par un ordre choisi "en largeur".

En affichant sur un même graphique les représentations des temps de calculs de ces deux algorithmes, on a vu que le plus rapide est celui de Bellman-Ford, contrairement à nos attentes. Mais, cela peut être expliqué par la manière différente de codage de ces deux algorithmes.

— Expliquer, selon le type de graphe considéré, le choix de l'algorithme que l'on doit effectuer.

Le choix entre les algorithmes de Dijkstra et de Bellman-Ford dépend du type de graphe considéré et des contraintes spécifiques du problème.

Graphe pondéré et non orienté sans cycles négatifs :

Si le graphe est non orienté, sans cycles négatifs et que toutes les arêtes ont des poids positifs, l'algorithme de Dijkstra est généralement préféré. Dijkstra est efficace dans ce cas, car il sélectionne progressivement les chemins les plus courts à partir d'un nœud source vers tous les autres nœuds ce qui signifie que l'on calcule tous les chemins de chaque point en même temps. L'algorithme prend plus de temps et ne marche pas pour les cycles négatifs mais il calcule tous les chemins pour chaque point de manière simple et compacte.

Graphe pondéré et orienté ou avec cycles négatifs :

Si le graphe est orienté ou s'il peut contenir des cycles négatifs, l'algorithme de Bellman-Ford est souvent utilisé. Bellman-Ford peut traiter ces situations, contrairement à Dijkstra. Il permet de détecter la présence de cycles négatifs et peut gérer les graphes où les poids des arêtes peuvent être négatifs. Bellman Ford a besoin de plus de données pour traiter les calculs mais de la manière dont on l'a implémenté il est plus rapide.

Deuxième partie

Seuil de forte connexité d'un graphe orienté.

7. Test de forte connexité

- En rappelant les définitions, expliquer pourquoi un graphe est fortement connexe si et seulement si la matrice de sa fermeture transitive n'a que des 1.

Un graphe est considéré fortement connexe quand il existe un chemin orienté entre chaque paire de sommets. La fermeture transitive d'un graphe est une matrice qui indique, pour chaque paire de sommets, s'il existe un chemin orienté entre eux.

On expliquera par la suite pourquoi un graphe n'est fortement connexe que si la matrice de la fermeture transitive du graphe ne contient que des uns.

- Si le graphe est fortement connexe, cela signifie qu'il existe un chemin orienté entre chaque paire de sommets. Par conséquent, dans la fermeture transitive, chaque entrée de la matrice correspondant à une paire de sommets a la valeur 1 car il existe un chemin entre les sommets.
- D'autre part, si la matrice de la fermeture transitive du graphe ne contient que des uns, cela signifie qu'il existe un chemin orienté entre chaque paire de sommets. Cela garantit que le graphe est fortement connecté, car chaque sommet peut être atteint à partir de n'importe quel autre sommet du graphe.

Par conséquent, la condition que la matrice de fermeture transitive d'un graphe ne contienne que des uns est une caractéristique importante des graphes fortement connexes.

- Écrire une fonction Python `fc(M)` qui prend en entrée la matrice d'un graphe orienté (non pondéré) et qui retourne `True` ou `False` à la question : "le graphe `G` donné par la matrice `M` (après numérotation de ses sommets) est-il fortement connexe ?" On pourra utiliser une des fonctions de fermeture transitive déjà créées en TD.

La fonction se trouve dans le fichier `TestForteConnexite.py`

8. Forte connexité pour un graphe avec $p=50\%$ de fléchés

On veut vérifier l'affirmation : "Lorsqu'on teste cette fonction `fc(M)` sur des matrices de taille n avec n grand, avec une proportion $p = 50\%$ de 1 (et 50% de 0), on obtient presque toujours un graphe fortement connexe."

On donnera un sens statistique à "presque toujours" : par exemple, "presque toujours" = "dans plus de 99% des cas testés". On demande donc ici une étude statistique en testant l'affirmation sur un grand nombre de cas (plusieurs centaines).

- Pour cela, on créera une fonction `test_stat_fc(n)` retournant le pourcentage de graphes de taille n fortement connexes pour un test portant sur quelques centaines de graphes.
- A partir de quel n l'affirmation ci-dessus est-elle vraie ?

La fonction se trouve dans le fichier `ForteConnexiteGraphe50.py`.

On a donc créé une fonction `test_stat_fc(n)` pour effectuer une étude statistique sur l'affirmation donnée. Cette fonction exécute un certain nombre de tests pour déterminer la proportion de graphes de taille n qui sont fortement connexes. Cette fonction renvoie ce pourcentage calculé en fonction des tests effectués.

La fonction `test_stat_fc(n)` exécute 200 tests sur des matrices générées à l'aide de la fonction `graphe2(n, 0,5, 0, 2)` qui renvoie une matrice de taille n avec un rapport de 50% de 1 (et 50% de 0). `test_stat_fc(n)` utilise ensuite la fonction `fc(M)` pour voir si chaque matrice générée a une relation de forte connexité. Un compteur est incrémenté à chaque fois qu'un graphe fortement connexe est trouvé. Le

pourcentage de graphiques fortement connexes est calculé en divisant le compteur par le nombre total de tests (num_tests) et en multipliant par 100. Ce pourcentage est retourné par la fonction. Au début de la fonction on peut ajuster la valeur de tests à effectuer. On pourra exécuter plus ou moins de tests pour une meilleure précision statistique.

Pour une matrice de taille 20x20, pour un test portant sur 200 graphes

La taille minimale pour laquelle l'affirmation est vraie est : 32

Pour une matrice de taille 5x5, pour un test portant sur 300 graphes

La taille minimale pour laquelle l'affirmation est vraie est : 35

9. Détermination du seuil de forte connexité

Pour n fixé, on va donc faire varier cette proportion p de fléchés (ou de 1 dans la matrice) en la diminuant jusqu'à obtenir un seuil $\text{seuil}(n)$ (il dépend de la taille n de la matrice) tel que pour $p \geq \text{seuil}(n)$ le graphe est presque toujours fortement connexe et en dessous duquel il ne l'est plus.

- Améliorer la fonction `test_stat_fc(n)` en une fonction `test_stat_fc2(n,p)` où le test porte maintenant sur des matrices de taille n avec une proportion p de 1.
- Écrire une fonction `seuil(n)` qui détermine ce seuil de forte connexité. (Pour une taille n donnée, on descendra p jusqu'à déterminer ce seuil).

La fonction se trouve dans le fichier `SeuilForteConnexite.py`.

Pour une matrice de taille 33x33, pour un test portant sur 300 graphes

Le seuil de forte connexité : 0.71

Pour une matrice de taille 50x50, pour un test portant sur 200 graphes

10. Étude et identification de la fonction seuil

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

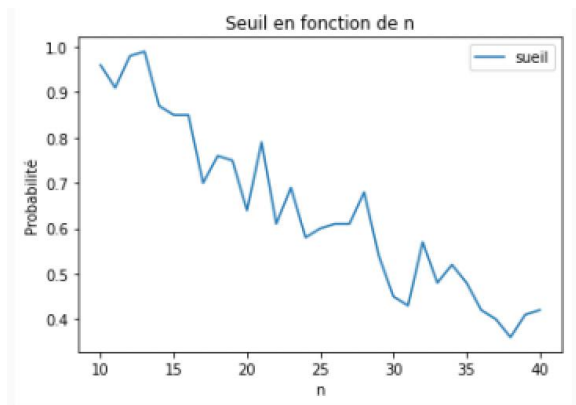
Représenter cette suite $\text{seuil}(n)$ sur l'intervalle $[10,40]$ (ou plus si la puissance de calcul le permet). Est-elle décroissante (comme on s'y attend) ?

La fonction se trouve dans le fichier `EtudeFonctionSeuil.py`.

La fonction `seuil(n)` détermine le seuil de forte connexité pour une taille donnée n en échelonnant la proportion p de 1 dans la matrice jusqu'à ce que le seuil soit déterminé. Le graphique de cette fonction dépend de la façon dont le pourcentage de tests réussis (p) varie avec la valeur de i à l'intérieur de la boucle `while`. La fonction `seuil(n)` utilise la fonction `test_stat_fc2(n, p)` pour effectuer un test de connectivité forte à divers rapports de p . À chaque itération de la boucle `while`, la fonction `test_stat_fc2(n, (100 - i) / 100)` est appelée pour trouver le pourcentage de tests réussis avec un certain pourcentage p .

Le but est de trouver la valeur maximale de p qui réussit le test. La représentation graphique dépend de la distribution des valeurs p en fonction de i . Si la valeur de p décroît linéairement avec l'augmentation de i , le graphique est une droite décroissante. Cependant, lorsque les valeurs de p varient de manière irrégulière ou non linéaire, comme dans notre cas, le graphique devient plus complexe.

On a représenté en log-log notre fonction pour $n=50$. On voit qu'il y a une forte décroissance, mais ce n'est pas régulière.



10.2 Identification de la fonction seuil(n)

— La fonction $\text{seuil}(n)$ est-elle asymptotiquement une fonction puissance ?

Dans notre cas, comme peut montrer le graphique, la fonction $\text{seuil}(n)$ n'est pas asymptotiquement une fonction puissance.

En effet, pour savoir si la fonction de $\text{seuil}(n)$ est asymptotiquement une fonction puissance, on doit analyser son comportement lorsque n tend vers l'infini. Mais, on doit tenir compte du fait que la fonction $\text{seuil}(n)$ ne dépend pas directement de n , mais de la variable i qui est incrémentée à l'intérieur de la boucle while.

Cette fonction utilise une boucle while qui s'exécute jusqu'à ce que i atteigne une valeur maximale de 100 ou jusqu'à ce que la valeur de p change. Le but de cette boucle est de trouver la valeur maximale de p qui passe le test de forte connexité. Le comportement de la fonction dépend donc de la répartition des valeurs de p en fonction de i . Si la valeur de p diminue linéairement et régulièrement avec l'augmentation de i , la fonction de $\text{seuil}(n)$ peut être approchée par une fonction puissance de la forme $f(x) = cx^a$, où a et c sont des constantes. Dans ce cas, la fonction de seuil est asymptotiquement une fonction puissance.

Cependant, si la valeur de p varie de manière irrégulière ou non linéaire lorsque i augmente, la fonction de seuil ne peut pas être approchée par une fonction de puissance, comme dans notre cas.

— Si oui, identifier cette fonction puissance cx^a (on pourra déterminer a et c à partir de l'équation de la droite de régression de la partie du nuage de points qui semble approximativement alignés sur le graphe log-log).

Même si notre fonction n'était pas asymptotiquement une fonction puissance, on a voulu quand même avoir une représentation 'de la droite de régression' de la partie du nuage de points. Dans notre cas, elle n'est pas du tout approximativement alignée sur le graphe log-log.

