S.A.E. 2.02 : Exploration algorithmique d'un problème.

2022-2023

AL Ramini Emad
Miled Wilem
El Kaddouri Amine
Weter Liam

Sommaire

1 Connaissance des algorithmes de plus courts chemins

1.2 Algorithme de Dijkstra :

Introduction :

Pour expliquer l'algorithme de Dijkstra, prenons l'exemple d'un graphe pondéré simple avec des poids positifs. Supposons que nous ayons le graphe suivant :

Voici la matrice suivante :

↱	a	b	c	d	e	f
a	0	4	∞	∞	∞	∞
b	4	0	5	∞	∞	∞
c	∞	5	0	∞	∞	6
d	∞	∞	∞	0	3	∞
e	∞	∞	∞	3	0	4
f	∞	∞	6	∞	∞	0

Initialisation :

Nous allons maintenant appliquer l'algorithme de Dijkstra à la matrice donnée pour trouver le plus court chemin entre le sommet de départ d et le sommet de fin s. Voici les différentes étapes:

1- Nous commençons par créer un tableau de distances initiales, appelé 'distances', qui va stocker les distances les plus courtes à partir du sommet de départ d.

	a	*b*	*c*	*d*	*e*	*f*
A={d}	(∞,/)	(∞,/)	(∞,/)	(0,/)	(3,d)	(∞,/)

Ensuite, nous allons chercher toutes les variables autres que notre sommet de départ étant susceptibles d'avoir ce sommet en tant que prédécesseur. Pour tout cela, on leur assigne comme prédécesseur de sur le tableau.

↱	a	b	c	d	e	F
a	0	4	∞	∞	∞	∞
b	4	0	5	∞	∞	∞
c	∞	5	0	∞	∞	6
d	**∞**	**∞**	**∞**	**0**	**3**	**∞**
e	∞	∞	∞	3	0	4
f	∞	∞	6	∞	∞	0

Dans ce cas-là, on va donc regarder dans ma matrice les valeurs pour la ligne de la variable d étant différentes de l'infini. On trouve seulement e.

Nous pouvons donc commencer la deuxième ligne du tableau de poids.

	a	*b*	*c*	*d*	*e*	*f*
A={d}	(∞,/)	(∞,/)	(∞,/)	(0,/)	(3,d)	(∞,/)
A={d e}						

Maintenant, l'initialisation est terminée. On peut donc entrer dans le vif du sujet.

Développement :

Puisque les variables d et e font déjà partie de l'ensemble A, on peut les enlever du tableau de distances. Pour cela, on entre une simple croix sur la ligne suivante comme ceci :

	a	*b*	*c*	*d*	*e*	*f*
A={d}	(∞,/)	(∞,/)	(∞,/)	(0,/)	(3,d)	(∞,/)
A={d e}						

On prend ensuite la dernière variable entrée dans l'ensemble A et regardons la matrice pour voir tous les entiers pouvant être des prédécesseurs de ce sommet (ceux étant différents de l'infini) :

↱	a	b	c	d	e	f
a	0	4	∞	∞	∞	∞
b	4	0	5	∞	∞	∞
c	∞	5	0	∞	∞	6
d	∞	∞	∞	0	3	∞
e	**∞**	**∞**	**∞**	**3**	**0**	**4**
f	∞	∞	6	∞	∞	0

Ici, d fait déjà partie de l'ensemble A et e est le sommet courant. On a alors seulement le sommet f.

Cette manipulation va être répétée jusqu'à ce que tous les sommets soient au minimum.

En premier temps, on va réécrire tous les sommets n'ayant pas été sélectionnés :

	a	*b*	*c*	*d*	*e*	*f*
A={d}	(∞,/)	(∞,/)	(∞,/)	(0,/)	(3,d)	(∞,/)
A={d e}	(∞,/)	(∞,/)	(∞,/)			

```
poids(d) + dist(d,e) < poids(e) # On remplace l'ancienne valeur par celle de la somme
poids(d) + dist(d,e) >= poids(e) # On ne change pas la valeur
```

Dans la formule, d est considéré comme le sommet courant et le sommet étudié.

Ici, on a donc : 3 + 4 ? ∞

7 étant plus petit que l'infini, on remplace donc l'ancien poids par 7 et on change le prédécesseur nul par e. Voici le résultat.

On applique donc la même chose pour la suite.

	a	*b*	*c*	*d*	*e*	*f*
A={d}	(∞,/)	(∞,/)	(∞,/)	(0,/)	(3,d)	(∞,/)
A={d e}	(∞,/)	(∞,/)	(∞,/)			(7,e)

Le sommet f a le plus petit poids.

	a	*b*	*c*	*d*	*e*	*f*
A={d}	(∞,/)	(∞,/)	(∞,/)	(0,/)	(3,d)	(∞,/)
A={d e}	(∞,/)	(∞,/)	(∞,/)			(7,e)
A={d e f}						

Pour le sommet c :

6 + 7 < ∞

Alors :

	a	*b*	*c*	*d*	*e*	*f*
A={d}	(∞,/)	(∞,/)	(∞,/)	(0,/)	(3,d)	(∞,/)
A={d e}	(∞,/)	(∞,/)	(∞,/)			(7,e)
A={d e f}	(∞,/)	(∞,/)	(13,f)			

Le sommet c a le plus petit poids.

	a	*b*	*c*	*d*	*e*	*f*
A={d}	(∞,/)	(∞,/)	(∞,/)	(0,/)	(3,d)	(∞,/)
A={d e}	(∞,/)	(∞,/)	(∞,/)			(7,e)

A={d e f}	(∞,/)	(∞,/)	(13,f)			
A={d e f c}						

Pour le sommet b :

5 + 13 < ∞

	a	*b*	*c*	*d*	*e*	*f*
A={d}	(∞,/)	(∞,/)	(∞,/)	(0,/)	(3,d)	(∞,/)
A={d e}	(∞,/)	(∞,/)	(∞,/)			(7,e)
A={d e f}	(∞,/)	(∞,/)	(13,f)			
A={d e f c}	(∞,/)	(18,c)				

Le sommet b a le plus petit poids.

	a	*b*	*c*	*d*	*e*	*f*
A={d}	(∞,/)	(∞,/)	(∞,/)	(0,/)	(3,d)	(∞,/)
A={d e}	(∞,/)	(∞,/)	(∞,/)			(7,e)
A={d e f}	(∞,/)	(∞,/)	(13,f)			
A={d e f c}	(∞,/)	(18,c)				
A={d e f c b}						

Pour le sommet a :

4 + 18 < ∞

	a	*b*	*c*	*d*	*e*	*f*
A={d}	(∞,/)	(∞,/)	(∞,/)	(0,/)	(3,d)	(∞,/)
A={d e}	(∞,/)	(∞,/)	(∞,/)			(7,e)
A={d e f}	(∞,/)	(∞,/)	(13,f)			
A={d e f c}	(∞,/)	(18,c)				
A={d e f c b}	(22,b)					

On peut donc entrer, sans soucis d'incohésion, la dernière ligne :

	a	*b*	*c*	*d*	*e*	*f*
A={d}	(∞,/)	(∞,/)	(∞,/)	(0,/)	(3,d)	(∞,/)
A={d e}	(∞,/)	(∞,/)	(∞,/)			(7,e)
A={d e f}	(∞,/)	(∞,/)	(13,f)			
A={d e f c}	(∞,/)	(18,c)				
A={d e f c b}	(22,b)					
A={d e f c b a}						

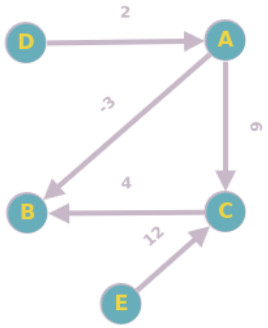
L'ensemble A est donc le suivant : {d e f c b a}

1.2 Présentation de l'algorithme de Bellman-Ford

Tout comme l'algorithme de Dijkstra, le but de l'algorithme de Bellman-Ford est d'obtenir le chemin le plus court entre deux points dans un graphe. Cependant la méthode pour obtenir ce chemin diffère de celle de Dijkstra. Ici nous allons utiliser les chemins au lieu de leurs sommets.

Pour expliquer cet algorithme, nous allons nous appuyer sur un exemple. Pour faire cet algorithme nous allons avoir besoin d'un graphe dans lequel les chemins peuvent être positifs et contrairement à l'algorithme de Dijkstra, les chemins négatif sont aussi autorisés.

	A	B	C	D	E
A	+∞	-3	6	+∞	+∞
B	+∞	+∞	+∞	+∞	+∞
C	+∞	4	+∞	+∞	+∞
D	2	+∞	+∞	+∞	+∞
E	+∞	+∞	12	+∞	+∞



A partir de ce graphe nous devons en tirer sa matrice M. Nous allons aussi avoir besoin de deux tableaux. Dans le premier tableau, on va devoir initialiser le point de départ puis nous y mettrons les chemins les plus courts jusqu'aux différents points du graphe. A la première position de chaque cases, nous aurons la distance totale parcourue jusqu'à ce sommet (initialisé à +∞) et à la deuxième position le prédécesseur de ce sommet (initialement vide). Le sommet de départ est ici D qu'on va initialiser à 0,D.

	A	B	C	D	E
Initialisation (dist,préd)	+∞, ∅	+∞, ∅	+∞, ∅	0,D	+∞, ∅

Dans le deuxième tableau nous vérifier toutes les flèches du graphe avec la relation suivante avec x et y deux sommets :

dist(x) + M(x,y) et dist(y)

Si ***dist(x) + M(x,y) < dist(y)*** alors on remplace au sommet y la distance par dist(x)+M(x,y) et le sommet prédécesseur par x.
Si ***dist(x) + M(x,y) ⩾ dist(y)*** alors on ne change rien.
Commençons par la flèche DA :

Flèche	Tour1
DA	0 + 2 < +∞

On met ensuite à jour le premier tableau :

A	B	C	D	E
+∞,Ø 2,D	+∞,Ø	+∞,Ø	0,D	+∞,Ø

Et on continue avec toutes les flèches :

Tableau 1 :

A	B	C	D	E
+∞,Ø	+∞,Ø	+∞,Ø	0,D	+∞,Ø
2,D	-1,A	8,A		

Tableau 2 :

Flèches	Tour 1	Tour 2
DA	0+2 < +∞	0+2 ≥ 2
AC	2+6 < +∞	2+6 ≥ 8
AB	2-3 < +∞	2-3 ≥ -1
CB	8+4 ≥ -1	8+4 ≥ -1
EC	+∞+12 ≥ +∞	+∞+12 ≥ +∞

Une fois le premier tour fait nous devons faire d'autre tours jusqu'à ce que nous ayons le signe ≥ sur toutes les lignes et si ce n'est pas possible on s'arrête au nombre de sommets – 1.

Maintenant il va falloir regarder le tableau 1. Pour trouver le chemin le plus court vers le sommet C, nous allons lister les prédécesseurs de C un à un. Le prédécesseur de C est A et le prédécesseur de A est D. La distance parcourue est de 8. Pour interpréter le chemin il va donc falloir inverser le chemin que l'on vient de trouver. On obtient alors D,A,C.

On peut remarquer que nous avons à la fin des distances positives, négatives et infinies. Ainsi on peut en conclure différentes issues.

- Lorsque le résultat est positif, il représente simplement le poids total du chemin emprunté. (D,C)
- Lorsque le résultat est négatif, il n'y a pas de chemin le plus court. (D,B)
- Lorsque le résultat est infini, il n'y a aucun chemin vers ce sommet. (D,E)

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

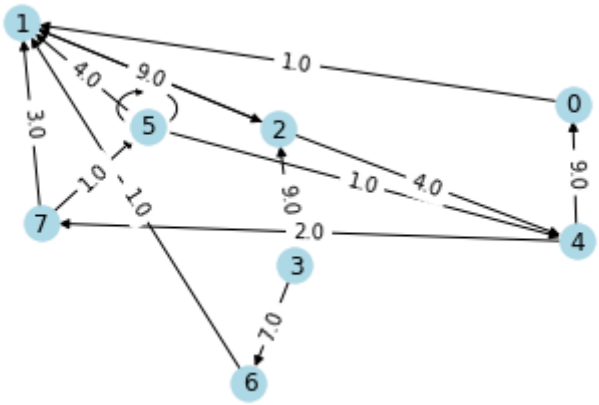
2.1 Dessin d'un graphe

(voire ligne 13 dans le code du fichier SAE2)

Pour dessiner un graphe nous avons utilisé l'outil : NetworkX :

Nous avons mis le code permettant d'afficher un graphe avec ses flèches et poids dans une fonction « dessiner_graphe » prenant en entrée une matrice et des chemins cette fonction nous permet aussi d'afficher les plus court chemins

Exemple un graphe avec 20% de flèches et sa matrice avec 8 sommets pour tester fonction « dessiner_graphe »: **(voire ligne 98 dans le code du fichier SAE2)**



Sa matrice:

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

2.2 Dessin d'un chemin

(voire ligne 46 dans le code du fichier SAE2)

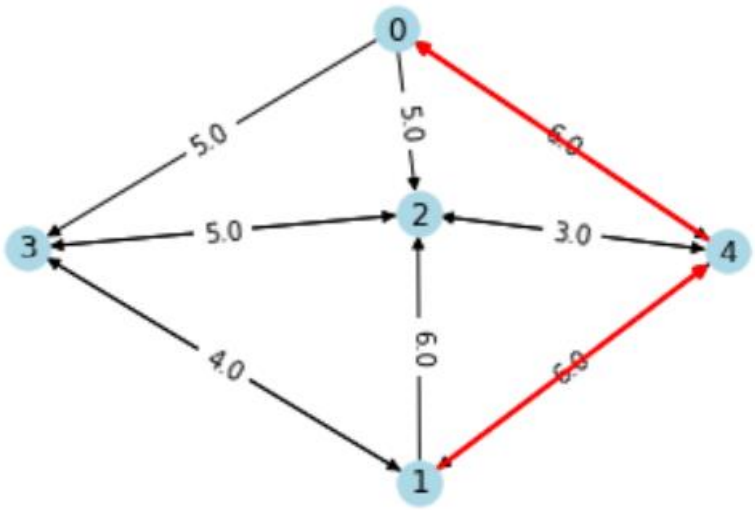
Pour que le chemin s'affiche en rouge on modifie la fonction a l'aid de la fonction get_chemin (Fonction auxiliaire pour récupérer le chemin du sommet de départ au sommet d'arrivée en utilisant le dictionnaire de prédécesseur.) :

```
# Mettre en évidence les plus courts chemins s'ils sont fournis
if chemins:
    for chemin in chemins.values():
        if isinstance(chemin[1], list):
            arêtes_chemin = [(chemin[1][i], chemin[1][i + 1]) for i in range(len(chemin[1]) - 1)]
            nx.draw_networkx_edges(G, positions, edgelist=arêtes_chemin, edge_color='red', width=2.0)
```

Explication du code :

On vérifie premièrement si le chemin est non vide, puis pour les chemins dans les dictionnaires de chemin, si le deuxième élément du chemin est une liste d'arêtes alors on crée une liste d'arêtes à partir du chemin donné, ensuite on dessine le chemin à l'aide de la bibliothèque NetworkX stocké dans la liste « arêtes_chemin » les arêtes seront dessiner en rouges.

Voici un exemple de graphe avec un chemin tracé :



3 Génération aléatoire de matrices de graphes pondérés

3.1 Graphe avec 50% de flèches

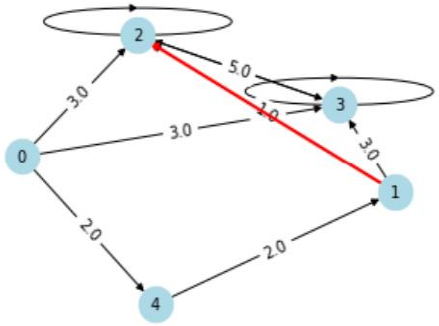
(voire ligne 71 dans le code du fichier SAE2)

Voila un exemple de matrice généré à moitié avec des poids entier compris entre [a,b[et des coefficient inf.

Voici un exemple de matrice :

```
[[ 6. inf inf  9.  3.]
 [ 8.  3.  2. inf  3.]
 [inf inf  7. inf inf]
 [inf inf  9. inf  8.]
 [inf inf  3.  2.  3.]]
```

Et voici un graphe avec 50% de flèches :



On remarque que les flèche vont seulement d'un nœud à l'autre et pas dans les deux directions.

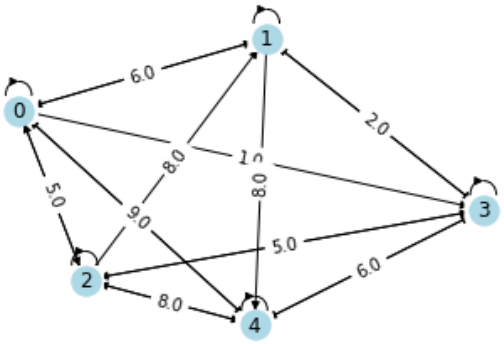
3.2 Graphes avec une proportion variable de flèche

(voire ligne 87 dans le code du fichier SAE2)

```
def graphe2(n, p, a, b):
    # Génération d'une matrice avec une proportion p de 1 et (1 - p) de 0
    matrice = np.random.binomial(2, p, size=(n, n))
```

Cette fonction permet d'augmenter la probabilité d'avoir un 1 et donc d'avoir de inf dans le tableau

Voici un exemple de matrice 90% de flèches et 5 sommets :




```
# Exemple d'utilisation
matrice = graphe2(5, 0.9, 1, 10)

print(matrice)
dessiner_graphe(matrice)
```

Sa matrice:

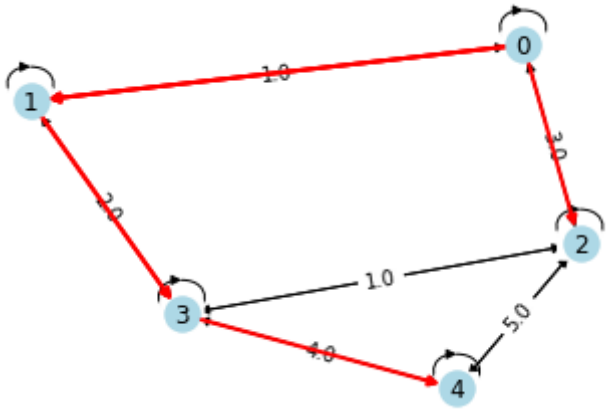
```
[[ 1.  7.  6.  1.  5.]
 [ 6.  3. inf  1.  8.]
 [ 5.  8.  8.  1.  6.]
 [inf  2.  5.  2.  9.]
 [ 9. inf  8.  6.  7.]]
```

4 Codage des algorithmes de plus court chemin

4.1 Codage de l'algorithme de Dijkstra

(voire ligne 103 dans le code du fichier SAE2)

Voici un exemple de graphe avec fonction Dijkstra pour montrer tous les plus courts chemins depart de sommet '0' vers tous les sommets : **(voire ligne 164 dans le code du fichier SAE2)**



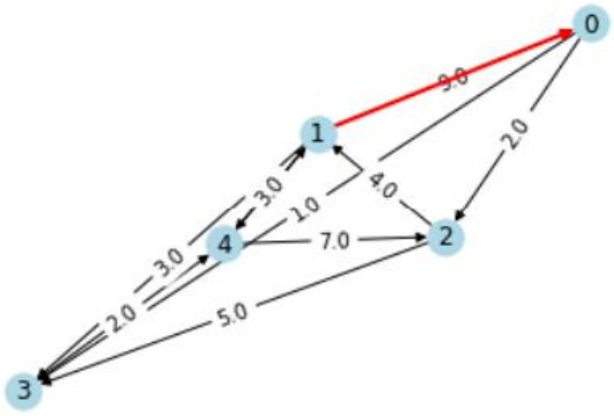
Plus court chemin :

```
Plus courts chemins:
De 0 à 1:
Distance: 1.0
Chemin: [0, 1]

De 0 à 2:
Distance: 3.0
Chemin: [0, 2]

De 0 à 3:
Distance: 3.0
Chemin: [0, 1, 3]

De 0 à 4:
Distance: 7.0
Chemin: [0, 1, 3, 4]
```

4.2 Codage de l'algorithme de Bellman-Ford

(voire ligne 185 dans le code du fichier SAE2)

Graphe avec 5 sommets, sommet de depart 1, sommet d'arrivée 0: **(voire ligne 227 dans le code du fichier SAE2)**



`(9.0, [1, 0])` : 9 correspond au poids, et [1,0] correspond au plus court chemin, avec comme départ 0 et comme arrivé 1.

L'algorithme de Bellman Ford permet de vérifier le chemin pour une arrête seulement

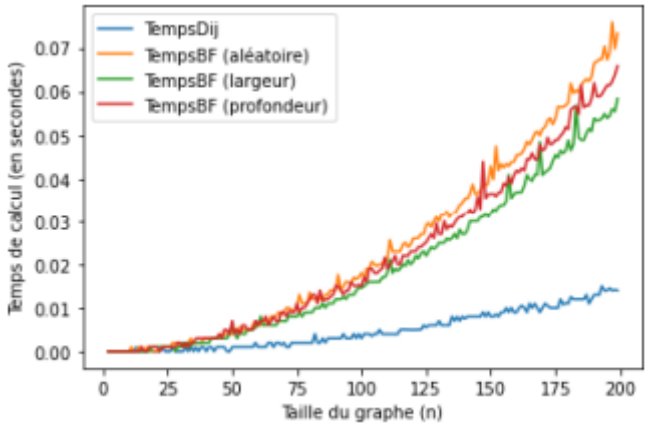
5 Influence du choix de la liste ordonnée des flèches pour l'algorithme de Bellman-Ford

(voire ligne 248 dans le code du fichier SAE2)

Après la génération d'un graphe pondéré à 50 sommets, nous vérifions le nombre de tours effectués pour différents types d'ordres. Pour le parcours en largeur, nous avons observé 1 tour, pour le parcours en profondeur et le parcours aléatoire, nous constatons que c'est fait 2 tours. Après avoir généré de nombreuses matrices de graphe pondéré, nous avons remarqué que le parcours en largeur était presque toujours celui qui nécessitait le moins de tours à effectuer. Nous en concluons donc que c'est le parcours le plus efficace.

```
Nombre de tours (ordre aléatoire) : 2

Nombre de tours (ordre largeur) : 1

Nombre de tours (ordre profondeur) : 2
```

On a comparé aussi le temps les résultats sur un même graphe pour les 3 variantes et on a trouvé que l'ordre en largeur est plus rapide que les autres:



6 Comparaison expérimentale des complexités

6.1 Deux fonctions "temps de calcul"

(voire ligne 384 dans le code du fichier SAE2)

TempsDij(n) : fonction pour calculer le temps pour faire algorithme de Dijkstra qui prend en entrée un entier positif n(Taille de matrice).

Exemple pour nombre de sommets = 1000:

```
Temps de calcul TempsDij : 0.723651647567749 secondes
```

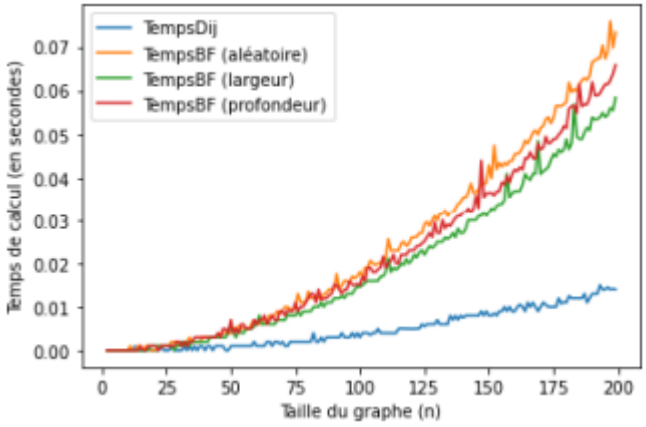
TempsBF **(voire ligne 402 dans le code du fichier SAE2)**

6.2 Comparaison et identification des deux fonctions temps

(voire ligne 424 dans le code du fichier SAE2)

Quel est l'algorithme le plus rapide?

Le plus rapide est l'algorithme de Dijkstra car on peut voir que la courbe bleue est largement inférieure aux autres algorithmes de Bellman-Ford.



Lorsque nous considérons une fonction t(n) qui a une croissance polynomiale d'ordre a pour n suffisamment grand, cela signifie que la fonction augmente rapidement à mesure que n augmente. La notation "cn^a" indique que la fonction peut être approximée par une autre fonction de la forme c * n^a, où c' est une constante.

```
l'exposant a est 2.316366878104195
```

Lorsque nous représentons graphiquement cette fonction en utilisant des coordonnées log-log, cela signifie que nous prenons le logarithme de la valeur de n sur l'axe des abscisses et le logarithme de la valeur correspondante de t(n) sur l'axe des ordonnées. Cela a pour effet de comprimer l'échelle des valeurs, ce qui peut rendre plus facile la visualisation des tendances de croissance à grande échelle.

Lorsque nous prenons le logarithme de t(n), nous obtenons log(t(n)) = log(c * n^a) = log(c) + a * log(n). Cette équation a la forme d'une droite avec une pente a et une constante de décalage log(c). Cela signifie que la représentation graphique de log(t(n)) en fonction de log(n) sera une ligne droite avec une pente a.

Ainsi, si une fonction t(n) a une croissance polynomiale d'ordre a, sa représentation graphique en coordonnées log-log sera approximativement une droite avec une pente a. Cela peut être utile pour analyser le comportement asymptotique d'une fonction et est souvent utilisé pour étudier la complexité des algorithmes tels que Bellman-Ford et Dijkstra, qui sont couramment utilisés pour résoudre des problèmes de plus courts chemins dans les graphes.

6.3 Conclusion

En résumé, l'algorithme de bellman-ford est bien plus lent que celui de Dijkstra dans les 3 différents.

- Graphes avec des arêtes pondérées négativement : Si le graphe contient des arêtes avec des poids négatifs, l'algorithme de Bellman-Ford est nécessaire car il peut détecter et gérer les cycles de poids négatifs. En revanche, l'algorithme de Dijkstra ne peut pas être utilisé dans ce cas car il suppose que tous les poids sont positifs ou nuls.

- Graphes sans cycles de poids négatif : Si le graphe est dépourvu de cycles de poids négatif, l'algorithme de Dijkstra est généralement préféré car il est plus efficace que l'algorithme de Bellman-Ford. L'algorithme de Dijkstra a une complexité en temps de O((|E| + |V|) log |V|), où |E| est le nombre d'arêtes et |V| est le nombre de sommets du graphe.

- Graphes avec cycles de poids négatif accessibles depuis la source : L'algorithme de Bellman-Ford doit être utilisé car il peut détecter ces cycles et indiquer qu'il n'y a pas de solution. L'algorithme de Dijkstra peut donner des résultats incorrects dans ce cas car il suppose que tous les poids sont positifs ou nuls.

- Graphes fortement connectés : L'algorithme de Bellman-Ford peut être utilisé pour trouver le plus court chemin à partir de n'importe quel sommet. En revanche, l'algorithme de Dijkstra nécessite un point de départ fixe.

- Graphes non orientés : L'algorithme de Dijkstra peut être utilisé car il ne considère que les arêtes sortant des sommets.

En résumé, si le graphe contient des arêtes pondérées négativement, des cycles de poids négatif accessibles depuis la source ou si le point de départ peut varier, l'algorithme de Bellman-Ford est plus approprié. Sinon, si le graphe est sans cycles de poids négatif et le point de départ est fixe, l'algorithme de Dijkstra est généralement préféré pour sa meilleure efficacité. Il est important d'analyser les caractéristiques du graphe et les contraintes du problème pour choisir l'algorithme le plus adapté.

7 Test de forte connexité

(voire ligne 447 dans le code du fichier SAE2)

Lorsque l'on parle de connexité nous parlons de l'intensité des liens entre les différents sommets d'un graphe. Ainsi lorsque l'on parle de forte connexité on veut dire que tous les sommets se rejoignent. On peut donc partir de n'importe quel sommet et accéder à n'importe quel autre sommet par au moins un trajet.

Mathématiquement cela veut dire que la matrice de la fermeture transitive doit être uniquement remplie de 1.

```
# Exemple d'utilisation
M = np.array([[0, 1, 0, 0, 0],
              [0, 0, 1, 0, 0],
              [0, 0, 0, 0, 1],
              [0, 0, 1, 0, 0],
              [0, 1, 0, 0, 0]])

is_connexe = fc(M)
print(is_connexe)
```

La matrice M n'est pas fortement connexe car fc(M) retourne False

8 Forte connexité pour un graphe avec p=50% de flèches

(voire ligne 482 dans le code du fichier SAE2)

On peut remarquer qu'en utilisant des graphes avec 50% de flèches, il faut une taille de plus de 12 sommets pour que nous obtenions une forte connexité de 99%.

```
Pourcentage de graphes avec p=50% de flèches fortement connexes pour
n=50 : 100.0%
```

```
pour matrice de taille  6 --> 65.0
pour matrice de taille  7 --> 75.0
pour matrice de taille  8 --> 91.0
pour matrice de taille  9 --> 88.0
pour matrice de taille  10 --> 98.0
pour matrice de taille  11 --> 98.0
pour matrice de taille  12 --> 99.0
pour matrice de taille  13 --> 99.0
pour matrice de taille  14 --> 100.0
pour matrice de taille  15 --> 100.0
```

9 Détermination du seuil de forte connexité

(voire ligne 503 dans le code du fichier SAE2)

La fonction test stat fc2(n,p) porte maintenant sur des matrices de taille n avec une proportion p.

Exemple n = 50 , p =50%

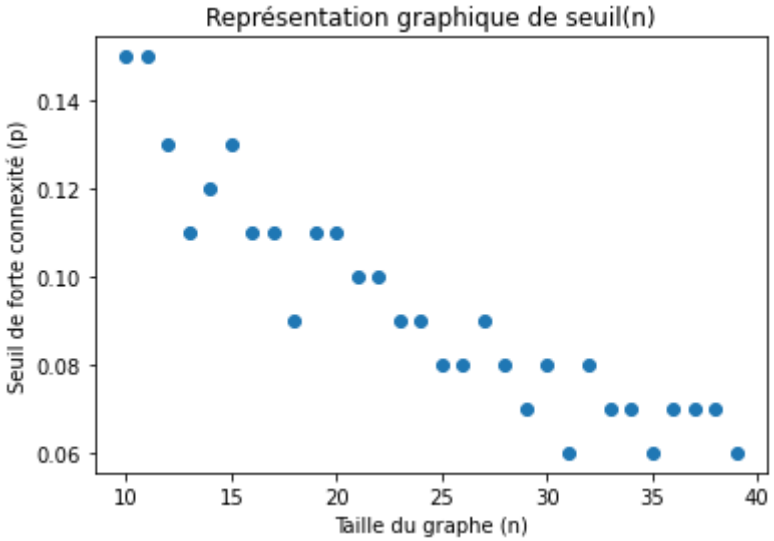
```
Pourcentage de graphes fortement connexes pour n=50, p=0.5 : 100.0%
```

10 Etude et identification de la fonction seuil

10.1 Représentation graphique de seuil(n)

(voire ligne 532 dans le code du fichier SAE2)

Contre toutes attentes, nous pouvons voir sur le nuage de points que le seuil de connexité régresse en fonction de la taille du graphe.



La suite seuil(n) représentée graphiquement devrait être décroissante, conformément aux attentes. Cela signifie que pour des valeurs croissantes de n, le seuil de forte connexité p diminue.

La décroissance de la suite seuil(n) est généralement attendue, car à mesure que la taille du graphe n augmente, il devient plus difficile d'obtenir une forte connexité. Par conséquent, la probabilité p nécessaire pour atteindre une forte connexité diminue progressivement.

En examinant la représentation graphique de seuil(n), vous pourrez confirmer si la suite est effectivement décroissante dans l'intervalle [10, 40] ou plus, selon la puissance de calcul disponible.

10.2 Identification de la fonction seuil(n)

Effectivement ce graphe possède une fonction asymptomatique que l'on peut voir sur le graphique ci-dessous en rouge. **(voire ligne 542 dans le code du fichier SAE2)**