KOH Virgil Shaun
ULLAH Maimuna-Ramisa
LFARH Mouataz

SAE 2.02 :

Exploration algorithmique d'un problème

Dirigé par Mr.GANNAM Lahcen

I-Comparaison d'algorithmes de plus courts chemins

1. Connaissances des Algorithmes de plus courts chemins

Deux algorithmes majeurs permettent de déterminer le plus court chemin dans un graphe : **Dijkstra** et **Bellman-Ford**. Le choix entre ces algorithmes repose sur la nature du graphe et les contraintes temporelles liées au calcul du chemin optimal. Des exemples concrets illustreront le fonctionnement de chaque algorithme.

1.1 L'algorithme de Dijkstra

L'algorithme de Dijkstra est un algorithme de recherche de chemin le plus court dans un **graphe pondéré**, c'est-à-dire un réseau de nœuds connectés par des arêtes ayant des poids ou des coûts associés. L'algorithme détermine le chemin le plus court entre un nœud de départ et tous les autres nœuds du graphe.

Cet algorithme utilise une approche gloutonne pour trouver le chemin optimal. Il attribue initialement une distance infinie à tous les nœuds, sauf au nœud de départ qui a une distance de 0.

Ensuite, à chaque itération, l'algorithme sélectionne le nœud non visité avec la distance la plus faible, appelé le « nœud courant ». Il met à jour les distances des nœuds voisins en les comparant à la distance actuelle du nœud courant, plus le poids de l'arête les reliant.
En d'autres termes il effectue une relaxation des distances pour les nœuds adjacents au nœud courant. La relaxation consiste à comparer la distance actuelle d'un nœud adjacent à la somme de la distance du nœud courant et du poids de l'arête qui les relie. Si cette somme est inférieure à la distance actuelle, la distance du nœud adjacent est mise à jour. Si la nouvelle distance est plus petite, elle est mise à jour.

L'algorithme répète ce processus jusqu'à ce que tous les nœuds aient été visités ou que la distance minimale vers le nœud d'arrivée soit trouvée.

Voici l'algorithme complet sous forme d'un pseudo code :

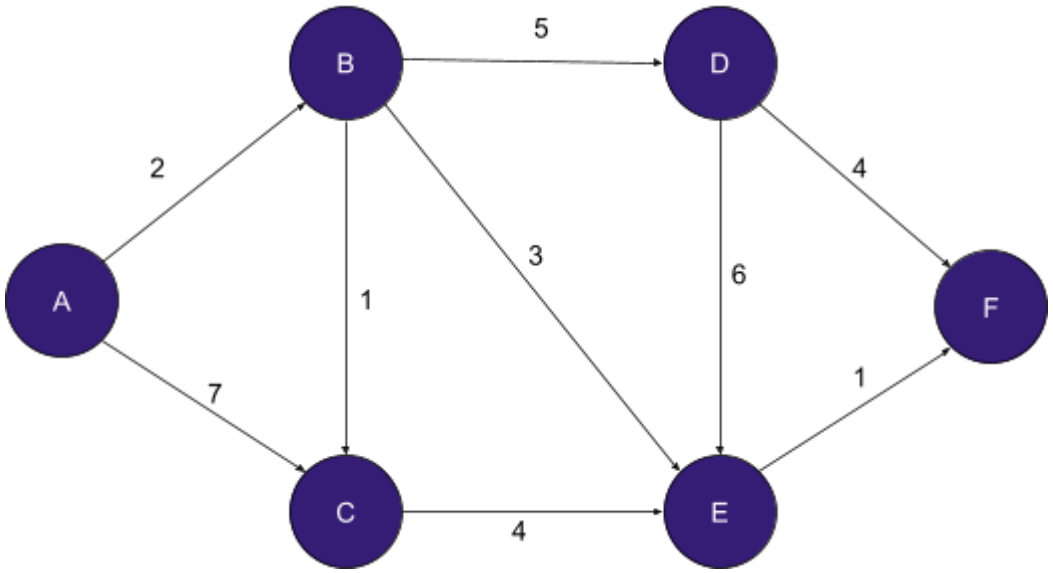
```
Procédure Dijkstra(M, d):
    Initialiser dist comme un dictionnaire de taille de M ou les clés sont les sommets avec toutes les valeurs à l'infini
   Mettre dist[d] à 0
    Initialiser pred comme un dictionnaire de taille de M ou les clés sont les sommets avec toutes les valeurs à -1
   Initialiser sommetsVisites comme une liste vide

   Tant que la longueur de sommetsVisites est inférieure à taille de M:
      Mettre u à -1
      Mettre sMin à l'infini
      Pour i allant de 0 à la taille de M:
         Si i n'est pas dans sommetsVisites et dist[i] est inférieur à sMin:
            Mettre sMin à dist[i]
            Mettre u à i
      Si u est -1, sortir de la boucle
      Ajouter u à sommetsVisites

      Pour v allant de 0 à la taille de M:
         Si v n'est pas dans sommetsVisites et M[u][v] > 0:
            Si dist[u] + M[u][v] est inférieur à dist[v]:
               Mettre dist[v] à dist[u] + M[u][v]
               Mettre pred[v] à u

   Initialiser cheminTrouve comme un dictionnaire vide
   Pour s allant de 0 à len(M):
      Si s est différent de d:
         Si dist[s] est inférieur à l'infini:
            Initialiser itineraire comme une liste contenant s
            Mettre u à pred[s]
            Tant que u est différent de -1:
               Insérer u au début de itineraire
               Mettre u à pred[u]
                     Mettre cheminTrouve[s] à un tuple contenant dist[s] et itinéraire
         Sinon:
```

Mettre cheminTrouve[s] à "sommet non joignable à d par un chemin dans le graphe G"

Retourner cheminTrouve

Démonstration :

Représentation du graphe



Nous avons pour but de trouver le chemin le plus court du sommet A au sommet F.

La calcul principal pour trouver la distance la plus courte pour chaque sommet se fait comme ci-dessous :

Si courant a un successeur
    Si distance(courant) + poids(courant, successeur) < distance(successeur)
        Remplacer distance(successeur) par distance(courant) + poids(courant, successeur)
    Sinon
        Garder distance(successeur)

Ensuite, on initialise un tableau avec les sommets a l'infini SAUF le sommet de départ. On ajoute une colonne pour le choix des sommets.

Sommet	A	B	C	D	E	F	Choix
Initialisation	0, ∅	inf, ∅	inf, ∅	inf, ∅	inf, ∅	inf, ∅	A(0)

Le symbole ∅ signifie qu'il n'y a pas de prédécesseur pour le sommet.

En appliquant le calcul principal :
On cherche les successeurs du sommet A : B et C
Distance(A) + poids(AB) = 0 + 2 = 2 < Distance(B). On met 2(A) au sommet B.
Distance(A) + poids(AC) = 0 + 7 = 7 < Distance(C). On met 7(A) au sommet C.
Il n'y a plus de successeurs à vérifier, donc on garde les autres distances.
Pour le choix, on prend la distance la plus courte entre les sommets sauf le sommet courant. Dans ce cas, c'est le sommet B.

Sommet	A	B	C	D	E	F	Choix
Initialisation	0, ∅	inf, ∅	inf, ∅	inf, ∅	inf, ∅	inf, ∅	A
Iteration 1	0, ∅	2, A	7, C	inf, ∅	inf, ∅	inf, ∅	B(2)

Nous avons bien le prochain sommet (B) pour répéter les étapes précédentes. Comme nous avons parcouru le sommet A, nous n'allons plus le regarder.
Alors, on continue avec le sommet B :

Sommet	A	B	C	D	E	F	Choix

Initialisation	0, ∅	inf, ∅	inf, ∅	inf, ∅	inf, ∅	inf, ∅	A
Iteration 1	0, ∅	2, A	7, C	inf, ∅	inf, ∅	inf, ∅	B(2)
Iteration 2		2, A	3, B	7, B	5, B	inf, ∅	C(3)

Les successeurs du sommet B : C, D, E

Distance(B) + poids(BC) = 2 + 1 = 3 < Distance(C). On met 3(B) au sommet C.
Distance(B) + poids(BD) = 2 + 5 = 7 < Distance(D). On met 7(B) au sommet D.
Distance(B) + poids(BE) = 2 + 3 = 5 < Distance(E). On met 5(B) au sommet B.
Il n'y a plus de successeurs du sommet B, on garde les autres distances.
On choisit la distance la plus courte parmi les sommets sauf le sommet courant.

Nous continuons jusqu'à ce que tous les sommets soient visités.

Sommet	A	B	C	D	E	F	Choix
Initialisation	0, ∅	inf, ∅	inf, ∅	inf, ∅	inf, ∅	inf, ∅	A
Iteration 1	0, ∅	2, A	7, C	inf, ∅	inf, ∅	inf, ∅	B(2)
Iteration 2		2, A	3, B	7, B	5, B	inf, ∅	C(3)
Iteration 3			3, B	7, B	5, B	inf, ∅	E(5)
Iteration 4				7, B	5, B	6, E	F(6)
Iteration 5				7, B		6, E	D(7)

Après toutes les itérations, nous avons bien le chemin le plus court d'un sommet vers un autre sommet. Reprenons la problématique avant : Trouver le chemin le plus court de A à F. Pour le trouver, il suffit de prendre tous les sommets précédents à partir du sommet d'arrivée.

Donc, le chemin le plus court de A à F est : [A, B, E, F] avec une distance de 6.

Nous pouvons conclure que l'algorithme de Dijkstra est très efficace pour trouver le chemin le plus court dans un graphe. Cependant il y a une limite. L'algorithme ne retourne pas le bon résultat pour les graphes ayant des flèches du poids négatif. Pour résoudre ce problème, l'algorithme suivant nous intéresse.

1.2 L'algorithme de Bellman-Ford

L'algorithme de Dijkstra s'impose comme un outil redoutable pour identifier rapidement les chemins les plus courts dans un graphe, à condition que les poids des arêtes soient positifs. Sa simplicité et sa rapidité d'exécution en font un choix privilégié pour de nombreux cas d'usage.

Imaginons un réseau routier où chaque ville est représentée par un nœud et chaque route par une arête. Le poids de chaque arête correspond à la distance entre les deux villes. Dans ce contexte, Dijkstra peut être utilisé pour déterminer le chemin le plus court entre deux villes quelconques.

Cependant, **Dijkstra** a ses limites. Il ne peut fonctionner que sur des graphes dont les arêtes ont des poids positifs. Si des poids négatifs ou des cycles négatifs sont présents, Dijkstra risque de s'égarer et de fournir des résultats erronés.

C'est là qu'intervient **Bellman-Ford**, un algorithme plus robuste et flexible. Contrairement à Dijkstra, **Bellman-Ford** peut gérer les graphes avec **des poids négatifs ou des cycles négatifs**. Il explore

progressivement toutes les possibilités pour trouver le chemin le plus court, même dans les environnements les plus complexes.

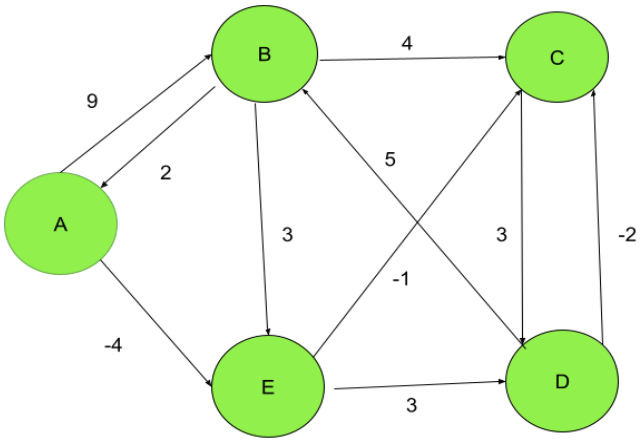
voir l'algorithme ci-dessous :

```
fonction bellman ford(graphe, départ):
    initialiser distances(graphe, départ)
    initialiser prédécesseurs(graphe, départ)
    n = nombre de sommets(graphe)

    // Itérations pour mettre à jour les distances
    Pour i de 1 à n-1:
        Si mettre à jour distances(graphe) est faux:
            Sortir de la boucle // Pas de modification au dernier tour

    // Vérification des cycles de poids négatif
    Si cycle poids négatif(graphe):
        Retourner "Cycle de poids négatif détecté"

    // Construction des résultats
    résultats = {}
    Pour chaque nœud s dans graphe:
        Si dist(s) == infini:
            résultats[s] = "Pas de chemin de d vers s"
        Sinon:
             chemin = reconstruire chemin(départ, s)
            résultats[s] = (dist(s), chemin)

    Retourner résultats

fonction mettre à jour distances(graphe):
    modification = faux
    Pour chaque arête (u, v) dans graphe:
        Si dist[u] != infini et dist[u] + poids(u, v) < dist[v]:
            dist[v] = dist[u] + poids(u, v)
            pred[v] = u
            modification = vrai
    Retourner modification

fonction cycle poids négatif(graphe):
    Pour chaque arête (u, v) dans graphe:
        Si dist[u] != infini et dist[u] + poids(u, v) < dist[v]:
            Retourner vrai
```

```
    Retourner faux

fonction initialiser distances(graphe, départ):
    Pour chaque sommet v dans graphe:
        dist[v] = infini
    dist[départ] = 0

fonction initialiser prédécesseurs(graphe, départ):
    Pour chaque sommet v dans graphe:
        pred[v] = null


fonction reconstruire chemin(départ, final):
    chemin = []
    noeud_actuel = final
    Tant que noeud_actuel != départ:
        chemin.insérer(0, noeud_actuel)


            noeud_actuel = pred[noeud_actuel]
    chemin.insérer(0, départ)
    Retourner chemin

fonction nombre_de_sommets(graphe):
    retourner nombre de sommets dans graphe
```

Démonstration :

La représentation du graphe



Le but est de trouver le chemin le plus court de A à D.

Tout d'abord, on crée une liste des flèches dans l'ordre lexicographique :
[AB, AE, BA, BC, BE, CD, DB, DC, EC, ED]

Il faut maintenir l'ordre de la liste pendant l'exécution. Le calcul principal se fait comme suit :

Si distance(s) + poids(ss') < distance(s')
remplacer distance(s') par distance(s) + poids(ss')
Sinon
On garde distance(s')

Ensuite, Nous établissons un tableau de sommets avec leurs distances à l'infini, SAUF le sommet de départ, de valeur 0.

	A	B	C	D	E
Initialisation	0, ⌀	inf, ⌀	inf, ⌀	inf, ⌀	inf, ⌀

Le symbole ⌀ signifie qu'il n'y a pas de sommet prédécesseur pour un sommet s.

Après, Nous utilisons le calcul principal pour chaque flèche dans la liste créée avant.

	A	B	C	D	E
Initialisation	0, ⌀	inf, ⌀	inf, ⌀	inf, ⌀	inf, ⌀
Iteration 1	0, ⌀	9, A	-5, E	-1, E	-4, A

En appliquant le calcul :

Distance(A) + poids(AB) = 0 + 9 = 9 < distance(B). On met 9(A) au sommet B.
Distance(A) + poids(AE) = 0 + (-4) = -4 < distance(E). On met -4(A) au sommet E.
Distance(B) + poids(BA) = 9 + 2 = 11 > Distance(A). On garde 0 au sommet A.
Distance(B) + poids(BC) = 9 + 4 = 13 < Distance(C). On met 13(B) au sommet C.
Distance(B) + poids(BE) = 9 + 3 = 12 > Distance(E). On garde -4(A) au sommet E.

Distance(C) + poids(CD) = 13 + 3 = 16 < Distance(D). On met 16(C) au sommet D.
Distance(D) + poids(DB) = 16 + 5 = 21 > Distance(B). On garde 9(A) au sommet B.
Distance(D) + poids(DC) = 16 + (-2) = 14 > Distance(C). On garde 13(B) au sommet C.
Distance(E) + poids(EC) = (-4)+ (-1) = -5 < Distance(C). On remplace 13(B) par -5(E)
Distance(E) + poids(ED) = (-4)+ 3 = -1 < Distance(D). On remplace 16(C) par -1(E)

Après avoir parcouru toutes les flèches, on constate que la première itération est finie.

Le nombre d'itérations dépend du nombre de sommets dans le graphe. C'est-à-dire :

Nombre d'itérations = Nombre de sommets

On continue le parcours des flèches dans la liste jusqu'à ce qu'on atteigne la dernière itération.

	A	B	C	D	E
Initialisation	0, ⌀	inf, ⌀	inf, ⌀	inf, ⌀	inf, ⌀
Iteration 1	0, ⌀	9, A	-5, E	-1, E	-4, A
Iteration 2	0, ⌀	3, D	-5, E	-2, C	-4, A
Iteration 3	0, ⌀	3, D	-5, E	-2, C	-4, A
Iteration 4	0, ⌀	3, D	-5, E	-2, C	-4, A
Iteration 5	0, ⌀	3, D	-5, E	-2, C	-4, A

	A	B	C	D	E
Initialisation	0, ⌀	inf, ⌀	inf, ⌀	inf, ⌀	inf, ⌀
Iteration 1	0, ⌀	9, A	-5, E	-1, E	-4, A
Iteration 2	0, ⌀	3, D	-5, E	-2, C	-4, A

Iteration 3	0, ∅	3, D	-5, E	-2, C	-4, A
Iteration 4	0, ∅	3, D	-5, E	-2, C	-4, A
Iteration 5	0, ∅	3, D	-5, E	-2, C	-4, A

Nous voyons qu'à partir de la troisième itération, nous avons un résultat constant. Nous pouvons conclure que nous avons bien un chemin le plus court d'un sommet à un autre sommet. Ensuite, nous reprenons la problématique précédente, trouver le chemin le plus court du sommet A au sommet D. Pour trouver ce chemin, il suffit de prendre les sommets précédents à partir du sommet d'arrivée.

Donc, le chemin le plus court est [A, E, C, D] avec une distance de -2.

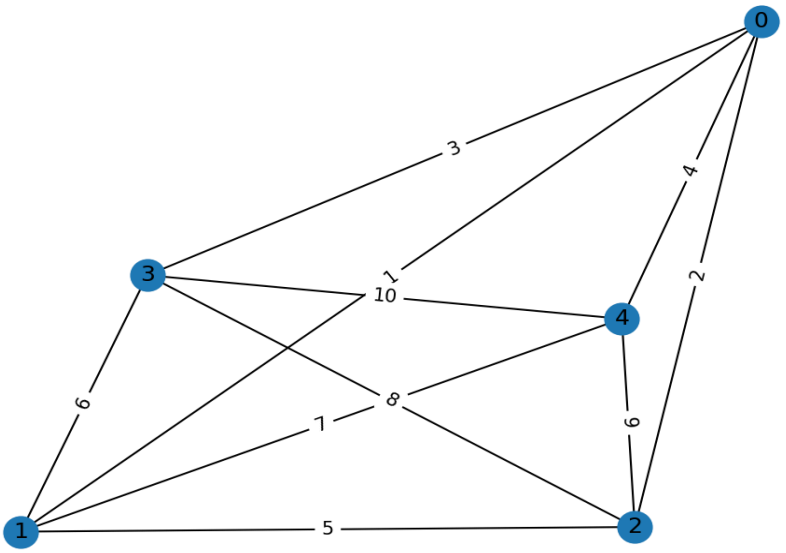
L'algorithme de Bellman-Ford peut régler les problèmes rencontrés par l'algorithme de Dijkstra. Cependant le calcul du chemin le plus court

peut prendre beaucoup de temps quand le nombre de sommets tend vers l'infini. Il y aura aussi des cas où les distances de chaque sommet ne restent pas constantes. Dans ce cas là, l'algorithme ne pourrait pas trouver une solution puisque nous atteignons le nombre d'itérations demandé.

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

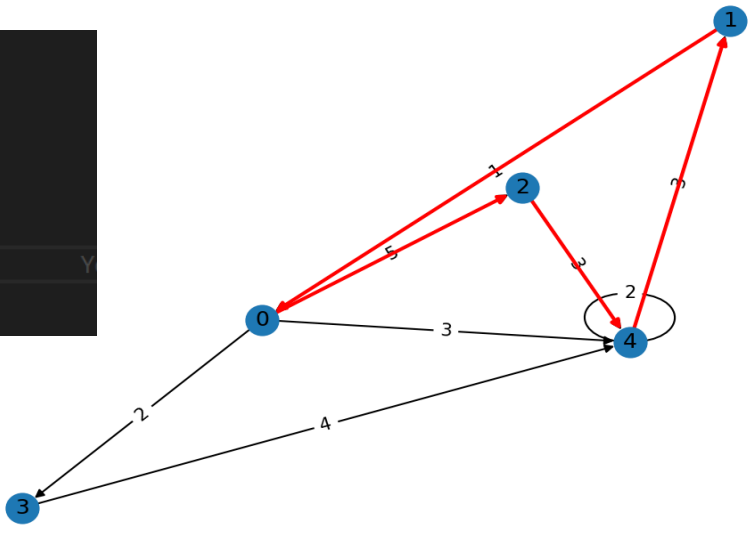
2.1 Dessin d'un graphe

```
# Test de la fonction construireGraphe
M = gm.graphe(5, 1, 20)
M1 = [[0, 1, 2, 3, 4],
      [1, 0, 5, 6, 7],
      [2, 5, 0, 8, 9],
      [3, 6, 8, 0, 10],
      [4, 7, 9, 10, 0]]
construireGraphe(M1)        You, 1 second
```



2.2 Dessin d'un chemin

```
M = [[0, 0, 5, 2, 3],
     [1, 0, 0, 0, 0],
     [0, 0, 0, 0, 3],
     [0, 0, 0, 0, 4],
     [0, 3, 0, 0, 2]]
construireChemin(M, [0, 2, 4, 1, 0])
```



3.Génération aléatoire de matrices de graphes pondérés

3.1 Graphes avec 50% de flèches

Cette fonction génère une matrice de taille n . n remplie de valeurs aléatoires. Les coefficients de la matrice sont initialisés avec une probabilité de 50% d'être 0 et 50% d'être 1.

Ensuite, les coefficients 1 sont remplacés par des valeurs aléatoires comprises entre a et b à l'aide de la fonction np.random.randint(a, b). Les coefficients 0 de la matrice sont ensuite remplacés par +∞ (infini) en convertissant la matrice en un type de données 'float64'. Enfin, la matrice modifiée est renvoyée en sortie.

```
print("Graphe 1")
print(graphe(6, 1, 20))
```

```
Graphe 1
[[inf inf 13. inf inf inf]
 [10. 18. inf 14. inf inf]
 [inf 14. 19.  3.  7. inf]
 [11. 18. inf inf inf  8.]
 [inf inf inf 18.  8. 12.]
 [11.  5.  5. inf inf inf]]
```

3.2 Graphes avec une proportion variables p de flèches

La fonction graphe2 prend en entrée les paramètres n (le nombre de sommets du graphe), p (la proportion de flèches), a et b (les bornes pour les valeurs des flèches). En utilisant la fonction np.random.binomial, nous générons une matrice de taille (n, n) remplie de valeurs binomiales. La valeur 1 est tirée avec une probabilité p, ce qui permet d'obtenir une proportion variable de flèches dans la matrice. ensuite nous parcourons chaque élément de la matrice M. Si un élément vaut 1, cela signifie qu'il s'agit d'une flèche et nous remplaçons sa valeur par un entier aléatoire compris entre a et b à l'aide de la fonction np.random.randint. Enfin,

nous convertissons la matrice en utilisant M.astype('float64') pour que les coefficients soient de type flottant.

Nous remplaçons ensuite les coefficients 0 par float('inf'), qui représente une valeur supérieure à tous les réels.

```
print("Graphe 2")
print(graphe2(6, 0.4, 1, 20))
```

```
Graphe 2
[[inf inf inf inf inf  4.]
 [inf inf inf inf inf inf]
 [inf inf inf 18. inf inf]
 [inf inf inf  5. inf 17.]
 [inf inf 14. 13. inf inf]
 [10. 18. inf  7. inf inf]]
```

4.Codage des algorithmes de plus court chemin

4.1 Codage de l'algorithme de Dijkstra

L'implémentation de l'algorithme de Dijkstra :

```
def Dijkstra(M, d):
      # création d'un dictionnaire pour stocker les sommets et leurs distances
    dist = {v : np.inf for v in range(len(M))}
    dist[d] = 0
      # création d'un dictionnaire pour stocker les sommets et leurs prédécesseurs
    pred = {v : -1 for v in range(len(M))}
    # création d'une liste pour stocker les sommets visités
    sommetsVisites = []

    # la boucle principale
    while len(sommetsVisites) < len(M):
        # trouver le sommet u non visité avec la distance minimale
        u = -1
        sMin = np.inf
        for i in range(len(M)):
            if i not in sommetsVisites and dist[i] < sMin:
                sMin = dist[i]
                u = i
        # sauter la boucle si u est -1 (aucun sommet trouvé)
        if u == -1:
            break

        # marquer u comme visité
        sommetsVisites.append(u)

        # mettre à jour les distances des sommets non visités
        for v in range(len(M)):
            if v not in sommetsVisites and M[u][v] > 0:
                if dist[u] + M[u][v] < dist[v]:
                    dist[v] = dist[u] + M[u][v]
                    pred[v] = u
```

```
    # construire les résultats
    cheminTrouve = {}
    for s in range(len(M)):
        if s != d:
            if dist[s] < np.inf:
                # Chemin trouvé, construire l'itinéraire
                itinéraire = [s]
                u = pred[s]
                while u != -1:
                    itinéraire.insert(0, u)
                    u = pred[u]

                cheminTrouve[s] = (dist[s], itineraire)
            else:
                    cheminTrouve[s] = "sommet non joignable à d par un
chemin dans le graphe."

    return cheminTrouve
```

4.2 Codage de l'algorithme de Bellman-Ford

L'implémentation de l'algorithme de Bellman-Ford :

```
def BellmanFord(M, d):
    # Initialisation des distances et des prédécesseurs
    distances = {v : np.inf for v in range(len(M))}
    distances[d] = 0
    prédécesseurs = {v : -1 for v in range(len(M))}

    # Mettre à jour les distances des sommets
    for i in range(len(M)):
        if not mettre_à_jour_distances(M, distances, prédécesseurs):
            break # No modification in the last round

    # Vérifier si un cycle de poids négatif existe
    if cycle_poids_négatif(M, distances):
        print("Negative weight cycle detected")

    # Construire les résultats
    résultats = {}
    for s in range(len(M)):
        if distances[s] == np.inf:
            résultats[s] = "No path from d to s"
        else:
            chemin = reconstruire_chemin(prédécesseurs, d, s)
            résultats[s] = (distances[s], chemin)

    return résultats

# retourne le poids d'une arête (u, v) dans le graphe M
def poids(u, v, M):
    return M[u][v]

# Mettre à jour les distances des sommets
def mettre_à_jour_distances(M, distances, predecesseurs):
    modification = False
```

```
    for u in range(len(M)):
        for v in range(len(M)):
            if distances[u] != np.inf and distances[u] + poids(u, v, M) < distances[v]:
                distances[v] = distances[u] + poids(u, v, M)
                prédécesseurs[v] = u
                modification = True
    return modification

# Vérifier si un cycle de poids négatif existe
def cycle_poids_négatif(M, distances):
    for u in range(len(M)):
        for v in range(len(M)):
            if distances[u] != np.inf and distances[u] + poids(u, v, M) < distances[v]:
                return True
    return False

# Reconstruire le chemin le plus court
def reconstruire_chemin(prédécesseurs, départ, final):
    chemin = []
    noeud_actuel = final
    while noeud_actuel != départ:
        chemin.insert(0, noeud_actuel)
        noeud_actuel = prédécesseurs[noeud_actuel]
    chemin.insert(0, départ)
    return chemin
```

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

Comme l'algorithme de Bellman-Ford est utilisé pour rechercher le plus court chemin dans un graphe avec des poids négatifs, le choix de la liste ordonnée des flèches peut grandement influencer ses performances de plusieurs façons :

Réduction du nombre d'itérations

En favorisant la relaxation des arcs avec des poids négatifs, l'algorithme peut converger plus rapidement, réduisant ainsi le nombre d'itérations nécessaires.

Détection de cycles négatifs

L'algorithme peut également détecter la présence de cycles négatifs dans un graphe. Si un cycle négatif est présent, l'algorithme ne convergera pas, indiquant l'existence de tels cycles.

Optimisation par différents ordres

Pour optimiser l'algorithme, nous pouvons utiliser différents types d'ordres pour organiser les flèches :

- Ordre aléatoire : On choisit la liste des flèches de manière aléatoire.

- Parcours en largeur : On utilise une file pour trouver la liste des flèches.

- Parcours en profondeur : On utilise une pile pour trouver la liste des flèches.

5.1 Parcours aléatoire

```python
def hasard(M):
    fleche = []
    for i in range(0,len(M)-1):
        for y in range(0,len(M)-1):
            if M[i][y] != np.inf :
                fleche.append((i, y))  # Ajouter la fleche (i, j) a la liste des resultats
    return random.sample(fleche, len(fleche))
```

Pour chaque indice non-nulle de la matrice, on ajoute l'arête dans la liste des flèches à parcourir.

5.2 Parcours Largeur

```python
# parcours en largeur
def parcour_largeur(M, d):
    mat = conversion(M)
    n = len(M)
    couleur = {}
    for i in range(n):      Virgil Shaun Koh, 2 weeks ago via PR #1 • Programmer les Variantes
        couleur[i] = 'blanc'
    couleur[d] = 'vert'
    file = [d]
    Resultat = []
    while file != []:
        i = file[0]
        for j in range(n):
            if (mat[file[0]][j] == 1 and couleur[j] == 'blanc'):
                file.append(j)
                couleur[j] = 'vert'
                Resultat.append((i, j))  # Ajouter la fleche (i, j) a la liste des resultats
        file.pop(0)
    return Resultat
```

On initialise une file avec le sommet de départ, et crée un dictionnaire pour suivre la couleur (état de visite) de chaque sommet. Ensuite, on parcourt le graphe, ajoutant chaque sommet non visité adjacent au sommet actuel à la file et à la liste des résultats, et marquant le sommet comme visité. On continue jusqu'à ce que tous les sommets soient visités. Enfin, elle renvoie la liste des arêtes du parcours en largeur.

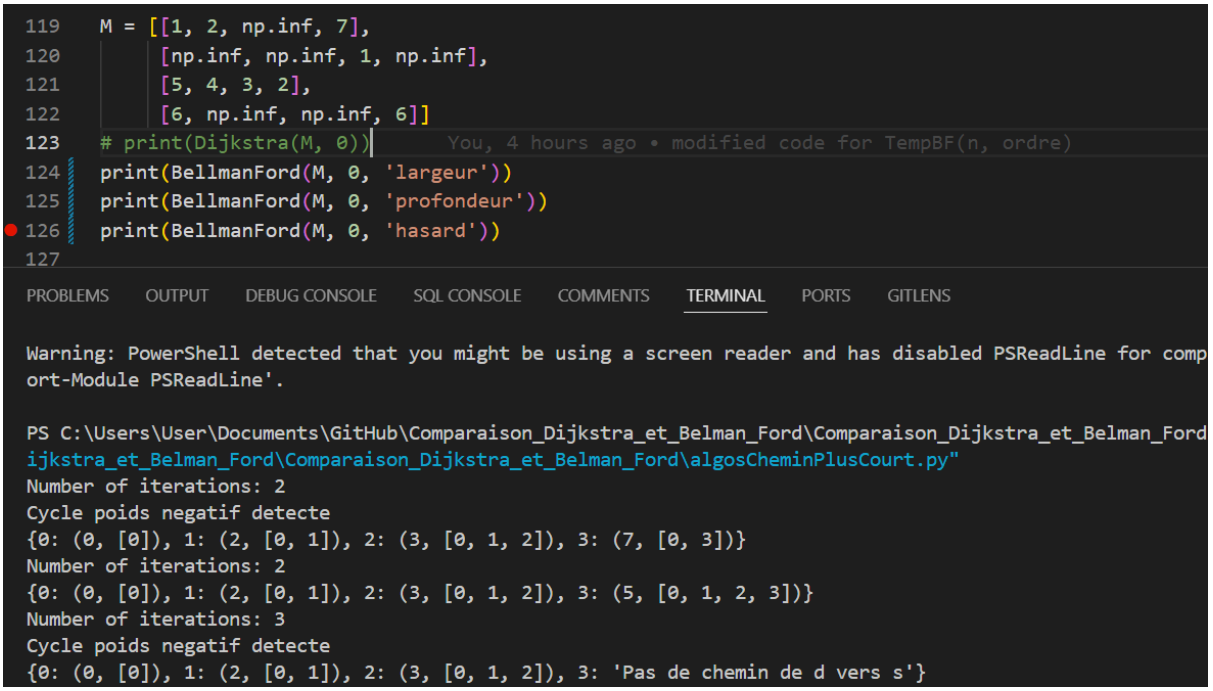
5.3 Parcours Profondeur

```python
# parcours en profondeur
def parcour_profondeur(M, d):
    mat = conversion(M)
    n = len(M)
    couleur = {}      Virgil Shaun Koh, 2 weeks ago via PR #1 • Programmer les Variantes
    for i in range(n):
        couleur[i] = 'blanc'
    couleur[d] = 'vert'
    pile=[d]
    Resultat=[]
    while pile !=[]:
        i=pile[-1]
        Succ_blanc=[]
        for j in range(n):
            if (mat[i,j]==1 and couleur[j]=='blanc'):
                Succ_blanc.append(j)
        if Succ_blanc!=[]:
            v= Succ_blanc[0]
            couleur[v]='vert'
            pile.append(v)
            Resultat.append((i, v))  # Ajouter la fleche (i, j) a la liste des resultats
        else:
            pile.pop()
    return Resultat
```

On initialise une pile avec le sommet de départ, et crée un dictionnaire pour suivre la couleur (état de visite) de chaque sommet. Ensuite, on parcourt le graphe, ajoutant chaque sommet non visité adjacent au sommet actuel à la pile et à la liste des résultats, et marquant le sommet comme visité. Si un sommet n'a pas de voisins non visités, il est retiré de la pile. On continue jusqu'à ce que tous les sommets soient visités. Enfin, la fonction renvoie la liste des arêtes du parcours en profondeur.

Choix de la liste des flèches à parcourir :

En faisant ces 3 types de parcours, on peut trouver le parcours le plus efficace. Pour cela, on compte le nombre d'itérations dès que l'algorithme finit une itération.



On voit que le parcours largeur et le parcours longueur ont moins d'itérations que le parcours en hasard. On peut conclure que le type de parcours influence bien l'efficacité de l'algorithme de Bellman-Ford.

6.Comparaison expérimentale des complexités

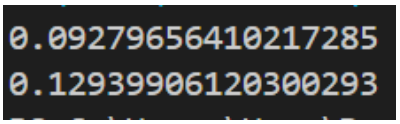
6.1 Deux fonctions ”temps de calcul” :

Pour mesurer les performances des algorithmes de plus courts chemins, nous allons créer deux fonctions en Python : TempsDij(n) et TempsBF(n). La première génère une matrice aléatoire d'un graphe pondéré de taille n , utilise l'algorithme de Dijkstra pour calculer les plus courts chemins depuis le premier sommet, et retourne le temps de calcul. La seconde fait de même, mais en utilisant l'algorithme de Bellman-Ford. Ces fonctions nous permettront de comparer l'efficacité des deux algorithmes en termes de temps de calcul.

Voici l'implémentation pour les deux fonctions :

```
# retourne le temps de calcul de l'algorithme de Dijkstra
def TempsDij(n):
    M = gm.graphe(n, 1, 200)
    debut = time.time()
    for i in range(n):
        acpc.Dijkstra(M, i)      You, 2 minutes ago • Uncommit
    fin = time.time()
    return fin - debut

# retourne le temps de calcul de l'algorithme de Bellman-Ford
def TempsBF(n, ordre='largeur'):
    M = gm.graphe(n, 1, 200)
    debut = time.time()
    for i in range(n):
        acpc.BellmanFord(M, i, ordre)
    fin = time.time()
    return fin - debut
```

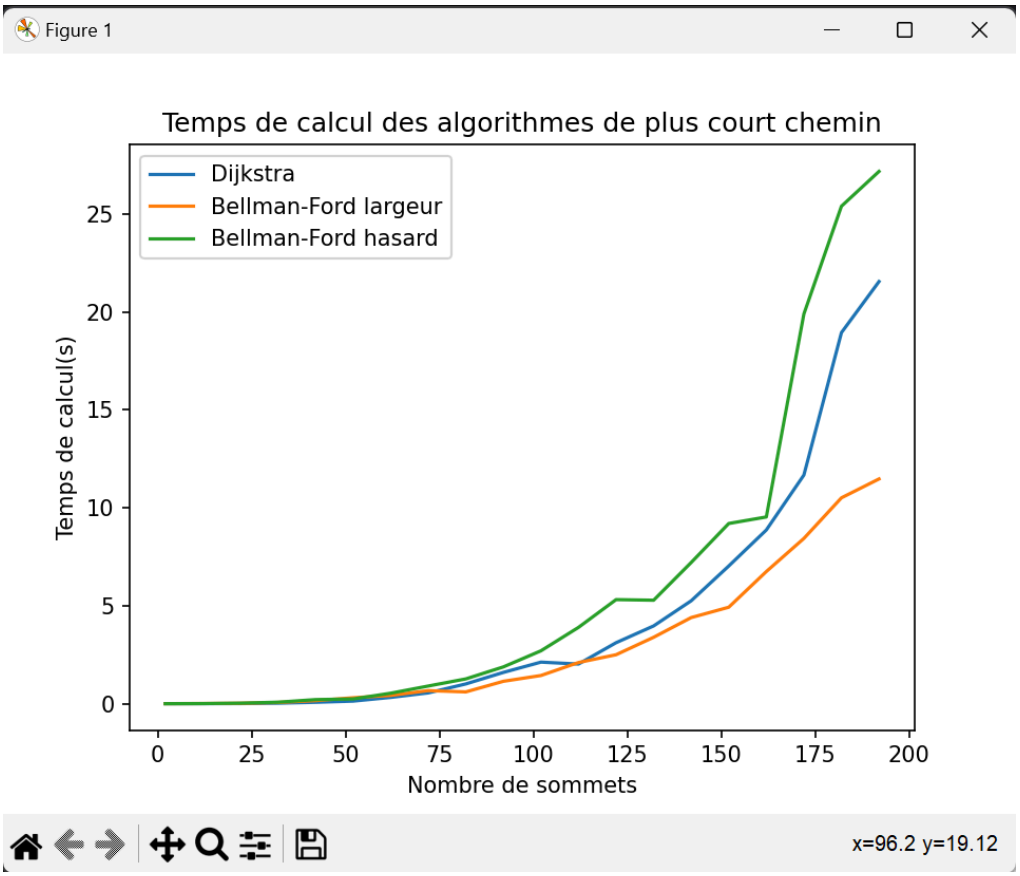
On prend n = 100, voici les temps de calcul retournés :



Le premier est pour l'algorithme de Dijkstra et le second pour l'algorithme de Bellman-Ford avec un parcours en largeur.

6.2 Comparaison et identification des deux fonctions temps

Pour analyser et comparer les performances des algorithmes de Dijkstra et de Bellman-Ford, nous allons représenter graphiquement les temps de calcul des fonctions TempsDij(n) et TempsBF(n) pour n allant de 2 à 200.



Cette figure compare les temps de calcul des algorithmes de plus court chemin : Dijkstra, Bellman-Ford en largeur, et Bellman-Ford en hasard, en fonction du nombre de sommets dans le graphe. Pour des graphes de petite à moyenne taille, les trois algorithmes montrent des performances similaires avec des temps de calcul relativement faibles. Cependant, à mesure que le nombre de sommets augmente, des différences notables apparaissent. Dijkstra et Bellman-Ford en largeur affichent une croissance plus régulière du temps de calcul, bien que Dijkstra commence à montrer une augmentation plus rapide

au-delà de 100 sommets. En revanche, Bellman-Ford en hasard présente des fluctuations et des pics de temps de calcul significatifs, surtout pour des graphes de grande taille, dépassant souvent les deux autres algorithmes. Ainsi, bien que Bellman-Ford en largeur soit plus stable et performant pour des graphes plus grands, Dijkstra devient moins efficace, et Bellman-Ford en hasard est le moins prévisible, avec le temps de calcul le plus élevé pour les très grands graphes.

Donc, selon cette représentation, l'algorithme de Bellman-Ford en parcours largeur est le plus rapide.

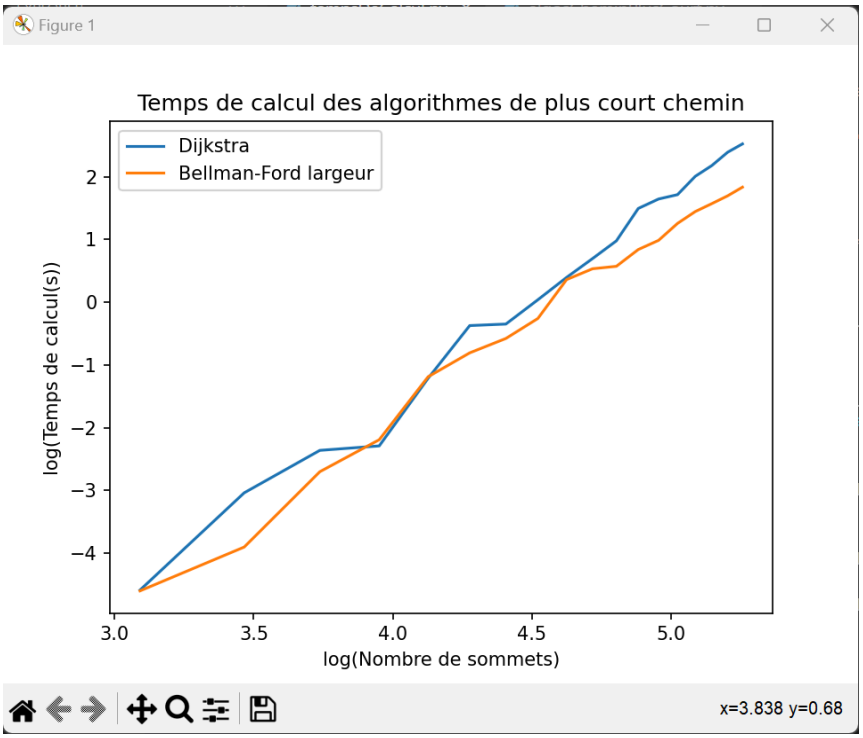
Si une fonction t(n) = cn^a pour n grand, alors c'est une fonction polynomiale d'ordre a. Si on prend les valeurs log-log pour chaque côté, alors :

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

Il s'agit d'une fonction affine sous forme de Y = mX + c, avec m = a, impliquant que a est bien la pente de la fonction.

On a vu dans la représentation précédente que le temps de calcul semble un parabole, concluant bien que les fonctions sont polynomiales avec un exposant a.

Pour trouver la pente a pour les deux algorithmes, j'ai utilisé np.log pour tracer les courbes :



Pour calculer à partir des courbes :

Dijkstra(bleu) :

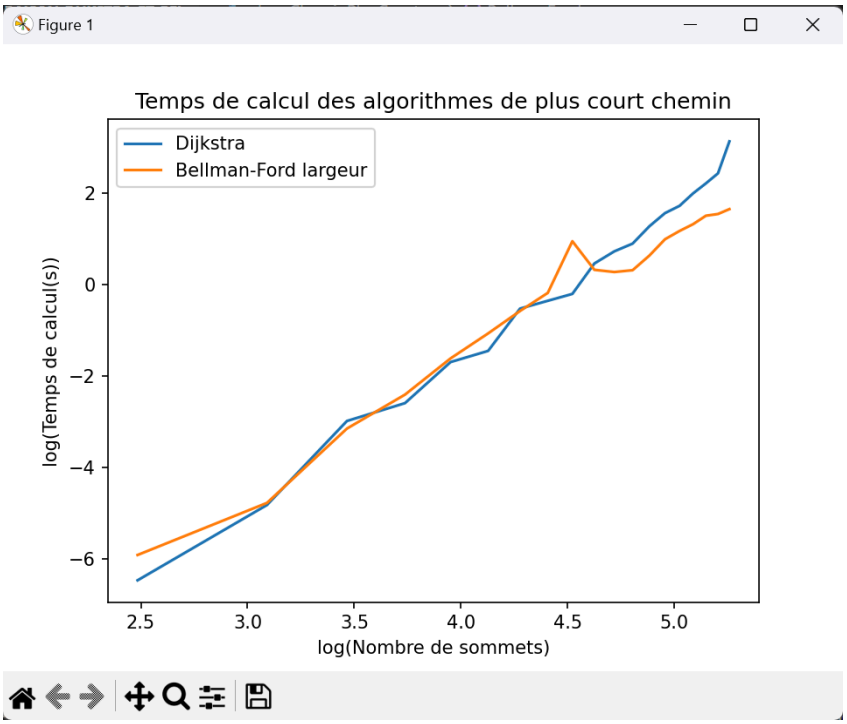
$$a = \frac{y2 - y1}{x2 - x1} = \frac{1{,}5 + 4}{5 - 3{,}25} = 3{,}1428 = \frac{22}{7}$$

Bellman-Ford en parcours largeur :
(x1, y1) = (4,0; -2), (x2, y2) = (5,0; 1)

$$a = \frac{y2 - y1}{x2 - x1} = \frac{1+2}{5 - 4} = 3$$

On voit que la pente de l'algorithme de Bellman-Ford en parcours largeur est plus petite que celle de l'algorithme de Dijkstra. Donc, l'algorithme de Bellman-Ford en parcours largeur est plus efficace que l'algorithme de Dijkstra.

Une autre représentation avec une proportion de flèches p = 0,4 :



Nous voyons que Temps Dij(n) est plus régulier que BellmanFord(n), disant que l'algorithme de Dijkstra est plus cohérent quand le nombre de sommets tend vers l'infini. Même si l'algorithme de Bellman-Ford semble plus efficace que celui-ci, il y a des complications dépendant du graphe donné.

6.3 Conclusion

Pour conclure, l'algorithme de Dijkstra est plus efficace que l'algorithme de Bellman-Ford en parcours hasard au niveau du temps de calcul. Cependant, il y a aussi des facteurs à considérer comme la taille du graphe, le nombre de flèches dans le graphe. Quand le nombre de sommets augmente, l'algorithme de Bellman-Ford en parcours largeur semble plus constant que l'autre, mais il va y avoir des cycles poids négatif. Par contre, pour l'algorithme de Dijkstra, il peut calculer très

rapidement pour environ 100 flèches, mais il ne peut pas gérer les arêtes négatives.

Pour les graphes non-orientés (sans les arêtes négatives), il est préférable d'utiliser l'algorithme de Dijkstra. Contrairement aux graphes orientés de grandes tailles, il est préférable d'utiliser l'algorithme de Bellman-Ford en parcours largeur.

II-Seuil de forte connexité d'un graphe orienté

7. Test de forte connexité

Définition : Un graphe orienté est dit fortement connexe si, pour chaque paire de sommets (u,v), il existe un chemin orienté de u à v et un chemin orienté de v à u. En d'autres termes, cela signifie que chaque sommet est accessible depuis n'importe quel autre sommet du graphe.

La matrice de fermeture transitive d'un graphe orienté est une matrice T où l'élément T[i][j] est 1 s'il existe un chemin orienté du sommet i au sommet j, et 0 sinon.

Si la matrice de fermeture transitive ne contient que des 1, cela signifie que pour chaque paire de sommets (u,v), il existe un chemin de u à v. Puisque chaque sommet est accessible depuis tous les autres sommets, le graphe est fortement connexe.

Pour vérifier si une matrice est fortement connexe, nous avons créé une fonction `fc(M)` qui prend en entrée une matrice de n'importe quelle taille. Nous avons importé un fichier pour utiliser un algorithme de parcours en profondeur afin de vérifier cette propriété.

(Voir code forte.Connexité.py)

8.Forte connexité pour un graphe avec 50 % de flèches

L'objectif de cette partie est de vérifier si un graphe orienté avec 50% de flèches est presque toujours fortement connexe pour des matrices de grande taille. "Presque toujours" signifie ici plus de 99% des cas testés. Pour ce faire, nous allons créer une fonction test_stat_fc(n) qui évalue le pourcentage de graphes fortement connexes parmi plusieurs centaines de graphes aléatoires de taille n.

Pour ce faire, nous avons besoin de générer des graphes aléatoires avec 50% de flèches. Nous avons déjà écrit cette fonction dans la partie 3, nous allons donc l'importer. De même, pour vérifier si le graphe généré est bien fortement connexe, nous avons déjà écrit cette fonction dans la partie 7, donc nous importons également.

(Voir code forte.Connexité.py)

Voici un test de la fonction `trouver_n()` qui retourne la plus petite valeur de \(n \) pour laquelle les matrices générées sont fortement connexes. Elle commence par n = 1 et elle exécute la fonction test_stat_fc(n) tant que la fonction elle-même donne un résultat < 0.99.

On conclut qu'à partir de \(n = 6 \), les matrices générées sont presque toujours fortement connexes.

9.Détermination du seuil de forte connexité

Pour un (n) fixé, nous allons faire varier la proportion (p) de flèches (ou de 1 dans la matrice) en la diminuant jusqu'à obtenir un seuil (seuil(n)). Ce seuil correspond à la proportion minimale pour laquelle le graphe est presque toujours fortement connexe.

La fonction `seuil` détermine le seuil de proportion de flèches pour lequel plus de 99% des graphes de taille 12 sont fortement connexes, taille trouvée dans la partie 8. Pour y parvenir, elle utilise la fonction `test_stat_fc2` en fixant la taille du graphe à 12 et en faisant varier la proportion \(p \) de flèches, en commençant à 1 et en diminuant de 0,1 à chaque itération.

Résultat

À partir de 27% de flèches dans un graphe de taille 12, on obtient presque toujours (>99%) un graphe fortement connexe.

(Voir code forte.Connexité.py)

10.Études et identifications de la fonction seuil

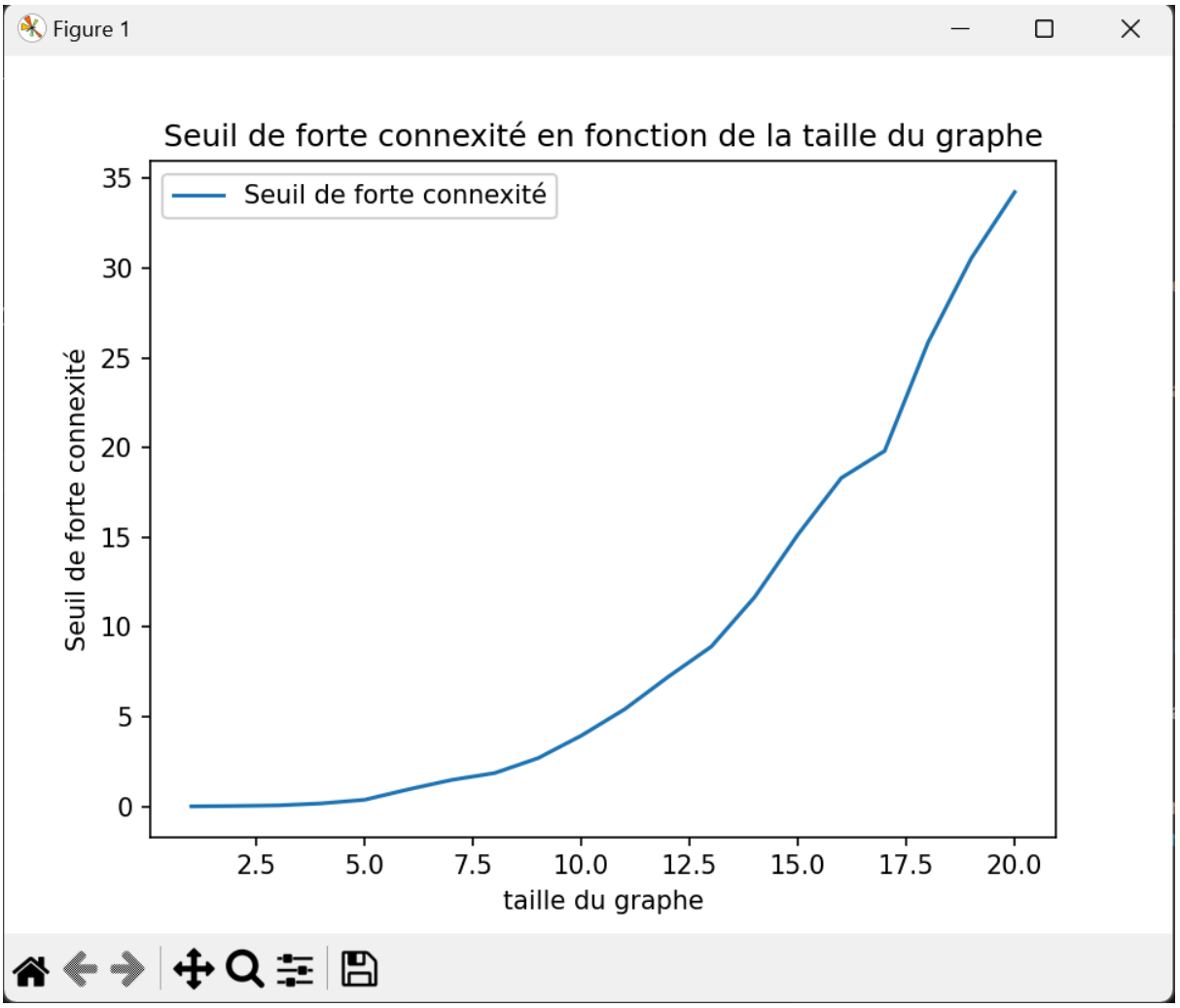
10.1 Représentation graphique de seuil(n)

Nous avons créé une fonction graphSeuil() qui trace une courbe de la seuil de forte connexité en fonction de la taille du graphe pour l'intervalle [10, 40].
(voir etudeDeSeuil.py)

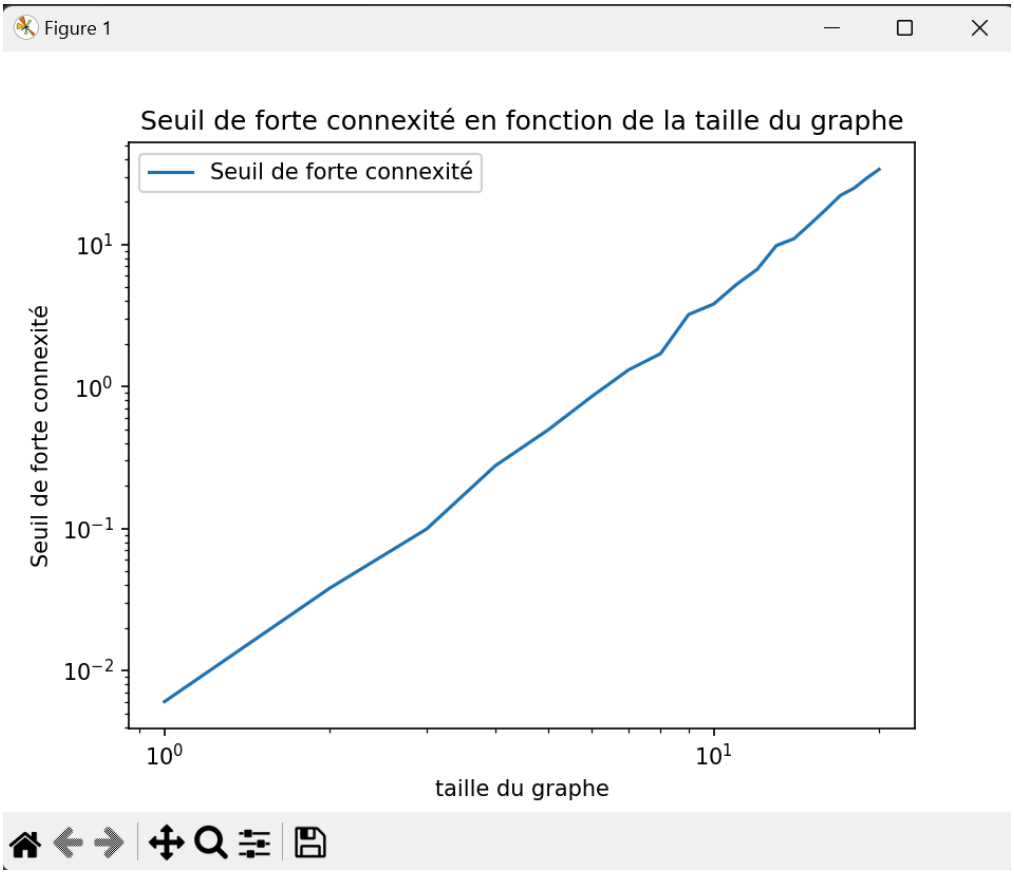
Selon la représentation graphique ci-dessus, on peut bien voir que la courbe est décroissante.

10.2 Identification de la fonction seuil(n)

Nous avons créé une fonction TempsSeuil(n) qui prend en entrée la taille du graphe n. Elle calcule le temps d'exécution de la fonction seuil(n). Cette fonction sert à tracer une courbe représentant la complexité en fonction du temps de calcul lorsque la taille de la matrice augmente.



Ici, nous avons créé un intervalle [1, 20]. Selon la représentation graphique ci-dessus, on conclut que la fonction seuil(n) est bien une fonction puissance. On peut l'exprimer sous la forme cn^a. Pour trouver les valeurs de c et a, nous utiliserons plt.loglog().



Nous voyons que c'est quasiment une droite.

Pour calculer la pente a est la valeur c :

$$a = \frac{y2 - y1}{x2 - x1} = \frac{5 - 0{,}0055}{10 - 1} = 0,5549$$

On prend (10, 5)

$$log(y) = a\, log(x) + log(c)$$
$$5 = 0,5549(10) + log(c)$$
$$log(c) = -0,549$$
$$c = e^{-0,549} = 0,5775$$

Donc, nous pouvons exprimer la fonction sous forme :

$$0,5775n^{0,5549}$$