Rapport de SAÉ S2.02

Exploration algorithmique d'un problème

Ellisa EE, Tristan GALLARDO,
Gérald LEBAN,
Kylian PAWILOWSKI, Clément VOISIN
Groupe B - Equipe B3

- Version du 10 juin 2024 -

Table des matières

1 Connaissance des algorithmes de plus courts chemins

1.1 Présentation de l'algorithme de Dijkstra

L'algorithme de Dijkstra est un algorithme qui permet de trouver le plus court chemin entre deux sommets d'un graphe à poids positifs. Cet algorithme ne permet pas de détecter les cycles à poids négatifs dans un graphe.

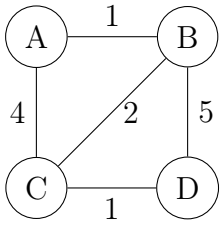

FIGURE 1 – Graphe non orienté à 4 sommets avec une pondération positive

Voici comment fonctionne l'algorithme pour trouver le chemin entre le sommet A et D :

1. On initialise toutes les distances à ∞ sauf pour le sommet de départ qui a une distance de 0.
2. On visite les sommets adjacents de A, ici nous avons les sommets B et C qui se trouvent respectivement à une distance de 1 et 4 du sommet A.
3. On choisit alors le plus court chemin jusqu'au sommet B : $B(1)$ que l'on met dans la colonne *choix*.
4. On part maintenant du sommet B où les sommets C et D se trouvent à une distance respectivement de $1 + 2$ et 4. On prend alors la plus petite distance entre *B(3)* et C(4) qui est *B(3)*. On ajoute dans la colonne *choix* le sommet $C(3)$.
5. En partant du sommet C, on ne peut aller qu'au sommet D. On a alors le choix entre $B(6)$ et $C(4)$. La plus petite distance est donc $C(4)$, que l'on ajoute dans la colonne *choix*.
6. Il suffit donc de remonter la liste des prédécesseurs pour trouver le plus court chemin. $D \to C, C \to B, B \to A$. Le plus court chemin entre A et D est donc A, B, C, D avec une distance de 4.

Étape	A	B	C	D	Choix
Initialisation	0	∞	∞	∞	A(0)
1	Déjà visité	A(1)	A(4)	∞	B(1)
2	Déjà visité	Déjà visité	B(3)	B(6)	C(3)
3	Déjà visité	Déjà visité	Déjà visité	C(4)	D(4)

TABLE 1 – Résultats de l'exécution de l'algorithme de Dijkstra réalisée à la main.

1.2 Présentation de l'algorithme de Bellman-Ford

L'algorithme de Bellman-Ford est quant à lui fonctionne même avec des arrêtes à poids négatif. L'algorithme est capable de détecter les cycles à poids négatifs.

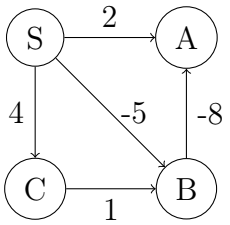

FIGURE 2 – Graphe orienté à 4 sommets avec des pondérations positives et négatives

Étape	S	A	B	C
Initialisation	0, S	∞, Ø	∞ ,Ø	∞, Ø
Tour 1	0, S	2,S	-5, S	4, S
Tour 2	0, S	-13, B	-5, S	4, S
Tour 3	0, S	-13, B	-5, S	4, S

TABLE 2 – Résultats de l'exécution de l'algorithme de Bellman-Ford réalisée à la main.

Voici comment fonctionne l'algorithme pour trouver le chemin entre le sommet S et A :

Initialisation On initialise toutes les distances à ∞, sauf la distance au point de départ S, qui sera à 0. Les sommets non visités sont représentés par $\emptyset$.

Tour 1

1. Pour aller en A :
   (a) S $\rightarrow$ A : Le poids est 2
   (b) B $\rightarrow$ A : On ne l'a pas encore visité donc ∞.

   Choix : On choisit l'arête avec le poids le plus faible : 2. $\implies$ $2, S$ dans la colonne A
2. Pour aller en B :
   (a) S $\rightarrow$ B : Le poids est de -5.
   (b) C $\rightarrow$ B : On ne l'a pas encore visité donc ∞.

   Choix : On choisit l'arête avec le poids le plus faible : -5. $\implies$ $-5, S$ dans la colonne B
3. Pour aller en C :
   (a) S $\rightarrow$ C : Le poids est de 4.

   Choix : On choisit l'arête avec le poids le plus faible : 4. $\implies$ $4, S$ dans la colonne C

Tour 2

1. Pour aller en A :
   (a) S $\rightarrow$ A : Le poids est 2

(b) B $\rightarrow$ A : Cette fois-ci, B a été visité au 1er tour : -5(S) + (-8) = -13

Choix : On choisit l'arête avec le poids le plus faible : -13. $\Longrightarrow$ $-13, S$ dans la colonne A

2. Pour aller en B :
   (a) S $\rightarrow$ B : Le poids est de -5.
   (b) C $\rightarrow$ B : Cette fois-ci, C a été visité au 1er tour : 4(S) + 1 = 5

Choix : On choisit l'arête avec le poids le plus faible : -5. $\Longrightarrow$ $-5, S$ dans la colonne B

3. Pour aller en C :
   (a) S $\rightarrow$ C : Le poids est de 4.

Choix : On choisit l'arête avec le poids le plus faible : 4. $\Longrightarrow$ $4, S$ dans la colonne C

Tour 3

1. Pour aller en A :
   (a) S $\rightarrow$ A : Le poids est 2
   (b) B $\rightarrow$ A : Cette fois-ci, B a été visité au 1er tour : -5(S) + (-8) = -13

Choix : On choisit l'arête avec le poids le plus faible : -13. $\Longrightarrow$ $-13, S$ dans la colonne A

2. Pour aller en B :
   (a) S $\rightarrow$ B : Le poids est de -5.
   (b) C $\rightarrow$ B : Cette fois-ci, C a été visité au 1er tour : 4(S) + 1 = 5

Choix : On choisit l'arête avec le poids le plus faible : -5. $\Longrightarrow$ $-5, S$ dans la colonne B

3. Pour aller en C :
   (a) S $\rightarrow$ C : Le poids est de 4.

Choix : On choisit l'arête avec le poids le plus faible : 4. $\Longrightarrow$ $4, S$ dans la colonne C

Conclusion Le plus court chemin pour aller en A depuis S est $S \rightarrow B \rightarrow C$. La distance du chemin vaut -13. Pour retrouver le chemin, il faut à la colonne **A** (point d'arrivée) se rendre au dernier tour **T3** où l'on peut lire **-13, B**. On se rend à la colonne **B** à la ligne **T2** où l'on lit **-5, S**. En se rendant à la colonne **S** du de la ligne **T1**, on lit **0, S**. Le chemin le plus court entre S et a est bien $S \rightarrow B \rightarrow C$ avec une distance de -13.

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

2.1 Dessin d'un graphe

Nous avons recherché un moyen de dessiner des graphes pondérés à partir de leur matrice en Python, et l'un des outils qui a retenu notre attention est la bibliothèque NetworkX. Elle permet de créer, analyser et visualiser des graphes. Voici quelques fonctions de la bibliothèque NetworkX que nous avons utilisé pour dessiner le graphe :

1. **nx.circular_layout()** : Positionner les nœuds sur un cercle
2. **nx.draw()** : Dessiner le graphe à l'aide de Matplotlib
3. **nx.draw_networkx** : Dessiner le graphe à l'aide de Matplotlib
4. **nx.draw_networkx_edges()** : Dessiner les arêtes du graphe
5. **nx.draw_networkx_edge_labels()** : Dessiner les libellés des arêtes sur le graphe

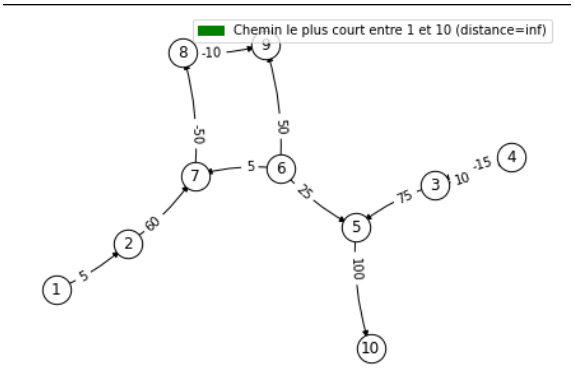

FIGURE 3 – Exemple de graphe dessiné avec NetworkX

2.2 Dessin d'un chemin

Nous avons ensuite amélioré la solution précédente en créant une fonction qui prend les paramètres suivants :

— **M** : la matrice d'origine représentant le graphe

— **resultat** : un dictionnaire de résultats généré par l'exécution des algorithmes de Dijkstra et de Bellman-Ford

— **d** : le sommet de destination vers lequel nous voulons trouver le chemin le plus court

À partir du dictionnaire de résultats, nous avons identifié le chemin indiqué, puis nous l'affichons avec la matrice originale. Cette fonction permet de visualiser clairement le chemin le plus court dans le graphe, en utilisant les résultats obtenus.

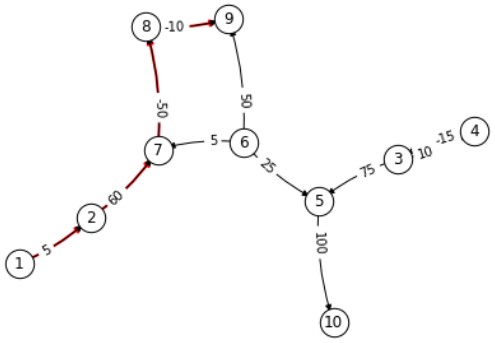

FIGURE 4 – Affichage d'un chemin indiqué par l'utilisateur

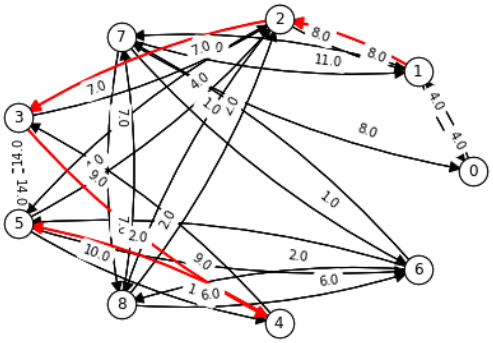

FIGURE 5 – Affichage d'un chemin indiqué par l'utilisateur

3 Génération aléatoire de matrices de graphes pondérés

3.1 Graphes avec 50% de flèches

Pour générer aléatoirement une matrice de taille donnée, nous avons créé la fonction suivante. Celle-ci nous permet de générer une matrice avec environ 50% de coefficients de valeur ∞ (en suivant la loi binomiale) et 50% de coefficients avec des poids entiers dans l'intervalle $p = [a, b[$.

```
def graphe(n, a, b):
    g = np.random.binomial(1, 0.5, size=(n, n))
    g = g.astype('float64')
    for i in range(n):
        for j in range(n):
            if g[i][j] == 0:
                g[i][j] = float('inf')
            else:
                g[i][j] = np.random.randint(a, b)
    return g
```

Le fonctionnement du programme

Les paramètres :

- **n** : la taille de la matrice
- **a** : la borne inférieure des poids des arrêtes
- **b** : la borne supérieure des poids des arrêtes

La fonction commence par la génération d'une matrice M de taille n en suivant la loi binomiale avec une probabilité de 0,5. Ensuite, la fonction assigne des poids aux coefficients de la matrice. Si $M[i, j]$ vaut 0, il est remplacé par ∞. Si l'élément vaut 1, il est remplacé par un entier aléatoire entre a et b. La fonction retourne la matrice générée.

Résultat d'exécution

```
array([[inf,  8., inf,  5.,  5., inf, inf, inf],
       [ 9., inf, inf,  6., inf,  6., inf,  8.],
       [ 5.,  2., inf,  2.,  8.,  4.,  7., inf],
       [inf,  9., inf,  4.,  9., inf,  3.,  8.],
       [inf,  5.,  4., inf,  1.,  5., inf, inf],
       [ 9.,  4., inf, inf, inf,  4.,  3.,  4.],
       [inf,  4., inf,  4., inf,  2.,  4.,  7.],
       [inf,  3.,  2., inf,  3., inf, inf,  5.]])
```

FIGURE 6 – Exemple de génération de matrice avec graphe(8,1,10)

```
array([[ 2., inf, inf, inf, inf,  2.,  2.,  1.],
       [inf, inf, inf,  3.,  4.,  1., inf, inf],
       [ 1., inf, inf, inf, inf,  1., inf,  1.],
       [ 8.,  6., inf, inf, inf,  1., inf,  8.],
       [ 7., inf,  9., inf,  4.,  2., inf, inf],
       [inf, inf,  7., inf,  7.,  6.,  1., inf],
       [ 2.,  9.,  2.,  9., inf,  9.,  4., inf],
       [ 3., inf,  3., inf,  3., inf,  6.,  6.]])
```

FIGURE 7 – Exemple de génération de matrice avec graphe(8,1,10)

3.2 Graphes avec une proportion p de flèches

Pour générer aléatoirement une matrice de taille donnée avec une proportion p de flèches, nous avons créé la fonction suivante.

```
def graphe2(n,p,a,b):
    g = np.random.binomial(1, p , size=(n, n))
    g = g.astype('float64')
    for i in range(n):
        for j in range(n):
            if g[i][j] == 0:
                g[i][j] = float('inf')
            else:
                g[i][j] = np.random.randint(a, b)
    return g
```

Le fonctionnement du programme

Les paramètres :

- **n** : la taille de la matrice
- **p** : la proportion de flèches
- **a** : la borne inférieure des poids des arrêtes
- **b** : la borne supérieure des poids des arrêtes

La fonction commence par la génération d'une matrice M de taille n en suivant la loi binomiale avec une proportion donnée. Ensuite, la fonction assigne des poids aux coefficients. Si $M[i, j]$ vaut 0, il est remplacé par ∞. Si $M[i, j]$ vaut 1, il est remplacé par un entier aléatoire entre a et b. La fonction retourne la matrice générée.

Résultat d'exécution

```
array([[inf, inf, inf, inf, inf, inf, inf, inf],
       [inf, inf, inf, inf,  5., inf, inf, inf],
       [inf,  5., inf, inf, inf, inf, inf, inf],
       [inf, inf, inf, inf, inf, inf,  6., inf],
       [inf, inf, inf, inf, inf, inf, inf,  5.],
       [inf, inf, inf, inf, inf, inf, inf, inf],
       [ 6.,  2., inf, inf, inf, inf, inf,  3.],
       [inf, inf, inf, inf, inf, inf, inf, inf]])
```

FIGURE 8 – Exemple de génération de matrice avec une proportion de 0.1

```
array([[ 6.,  1.,  4.,  8.,  4.,  6.,  2.,  5.],
       [ 4.,  1.,  2.,  2.,  5.,  2.,  5.,  7.],
       [ 3.,  2.,  9., inf,  2.,  6.,  4.,  9.],
       [ 8., inf,  4.,  4.,  6.,  4.,  7.,  7.],
       [ 7., inf,  6.,  1.,  9.,  3.,  2.,  3.],
       [inf,  3.,  8.,  3.,  8.,  5.,  9.,  2.],
       [ 7.,  4.,  7., inf,  7.,  9.,  4.,  8.],
       [ 1.,  7.,  5.,  9.,  4.,  3.,  5.,  2.]])
```

FIGURE 9 – Exemple de génération de matrice avec une proportion de 0.9

4 Codage des algorithmes de plus court chemin

4.1 Codage de l'algorithme de Dijkstra

4.1.1 Fonctionnement du programme de l'algorithme de Dijkstra

1. **Initialisation des structures de données**
   (a) Créer des dictionnaires **distance** et **sommetPrecedent** pour stocker les distances depuis le sommet de départ et les sommets précédent.
   (b) Initialiser les listes **visted** et **unvisited** pour les sommets visités et non visités
   (c) Pour chaque sommet dans le graphe, initialiser la distance à l'infini et le sommet précédent à None
   (d) Ajouter tous les sommets à la liste **unvisited** en utilisant le parcours séquentiel, le parcours en largeur et le parcours en profondeur
   (e) Définir la distance du somme de départ à 0
2. **Itération jusqu'à ce que tous les sommets soient visités**
   (a) Tant qu'il y a des sommets non visités
      i. Trouver le sommet non visité avec la plus petite distance dans le dictionnaire **distance**
      ii. Mettre à jour les distances des voisins du sommet actuel si une distance plus courte est trouvée
      iii. Retirer le sommet de la liste **unvisited** et l'ajouter à la liste **visited**
3. **Calcul des chemins les plus courts**
   (a) Pour chaque sommet dans le graphe, reconstruire le chemin le plus court en utilisant le dictionnaire **sommetPrecedent**
   (b) Stocker la distance ainsi que le chemin dans le dictionnaire **resultats**
4. **Retourner les résultats**
   (a) Retourner le dictionnaire **resultats** contenant les distances et les chemins les plus courts depuis le sommet de départ jusqu'à tous les autres sommets

4.1.2 Résultat d'exécution du programme de l'algorithme de Dijkstra

```
Matrice pondérations positives :
Matrice originale
array([[inf,  4., inf, inf, inf, inf, inf, inf, inf],
       [ 4., inf,  8., inf, inf, inf, inf, 11., inf],
       [inf,  8., inf,  7., inf,  4., inf, inf,  2.],
       [inf, inf,  7., inf,  9., 14., inf, inf, inf],
       [inf, inf, inf,  9., inf, 10., inf, inf, inf],
       [inf, inf,  4., 14., 10., inf,  2., inf, inf],
       [inf, inf, inf, inf, inf,  2., inf,  1.,  6.],
       [ 8., 11., inf, inf, inf, inf,  1., inf,  7.],
       [inf, inf,  2., inf, inf, inf,  6.,  7., inf]])
Solution trouvé en utilisant Dijkstra
{0: [0, [0]],
 1: [4.0, [0, 1]],
 2: [12.0, [0, 1, 2]],
 3: [19.0, [0, 1, 2, 3]],
 4: [26.0, [0, 1, 2, 5, 4]],
 5: [16.0, [0, 1, 2, 5]],
 6: [16.0, [0, 1, 7, 6]],
 7: [15.0, [0, 1, 7]],
 8: [14.0, [0, 1, 2, 8]]}
```

FIGURE 10 – Parcours produit pour une matrice avec des pondérations positives

4.2 Codage de l'algorithme de Bellman-Ford

4.2.1 Fonctionnement du programme de l'algorithme de Bellman-Ford

1. **Création de la matrice d'incidence**
   (a) Convertir la matrice originale en une matrice d'incidence en utilisant une fonction **matriceIncidence(m)**.
2. **Choix du parcours**
   (a) En fonction du choix spécifié, la fonction détermine une liste ordonnée des sommets du graphe à visiter en utilisant le parcours séquentiel, en largeur ou en profondeur.
3. **Initialisation des poids**
   (a) Initialiser les poids des sommets à l'infini, sauf le poids du sommet de départ, qui est fixé à 0.
4. **Calcul des chemins les plus courts**
   (a) Itérer sur chaque flèche du graphe et mettre à jour les poids de sommets si un chemin plus court est trouvé.
   (b) Continuer l'algorithme si un changement est effectué.
5. **Détection de cycles négatifs**
   (a) Vérifier la présence de cycles négatifs. Si un chemin plus court est encore trouvé, cela signifie qu'il y a un cycle négatif.
   (b) Définir les poids des sommets sur -∞ s'il y a un cycle négatif.
6. **Construction des résultats**
   (a) Stocker les résultats dans un dictionnaire où chaque sommet est associé à son poids le plus court et au chemin correspondant.
   (b) Enregistrer un message approprié si un sommet est inaccessible ou il existe un cycle négatif.
7. **Retour des résultats**
   (a) Le dictionnaire des résultats et le nombre d'itérations effectués sont renvoyés.

4.2.2 Résultat d'exécution du programme de l'algorithme de Bellman-Ford

```
Matrice pondérations positives :
Matrice originale
array([[inf,  4., inf, inf, inf, inf, inf, inf, inf],
       [ 4., inf,  8., inf, inf, inf, inf, 11., inf],
       [inf,  8., inf,  7., inf,  4., inf, inf,  2.],
       [inf, inf,  7., inf,  9., 14., inf, inf, inf],
       [inf, inf, inf,  9., inf, 10., inf, inf, inf],
       [inf, inf,  4., 14., 10., inf,  2., inf, inf],
       [inf, inf, inf, inf, inf,  2., inf,  1.,  6.],
       [ 8., 11., inf, inf, inf, inf,  1., inf,  7.],
       [inf, inf,  2., inf, inf, inf,  6.,  7., inf]])
Solution trouvé en utilisant Bellman-Ford
{1: [4.0, [0, 1]],
 2: [12.0, [0, 1, 2]],
 3: [19.0, [0, 1, 2, 3]],
 4: [26.0, [0, 1, 2, 5, 4]],
 5: [16.0, [0, 1, 2, 5]],
 6: [16.0, [0, 1, 7, 6]],
 7: [15.0, [0, 1, 7]],
 8: [14.0, [0, 1, 2, 8]]}
```

FIGURE 11 – Parcours produit pour une matrice avec des pondérations positives

```
Matrice pondérations négatives :
Matrice originale
array([[ inf,   5.,  inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf],
       [ inf,  inf,  20.,  inf,  inf,  30.,  60.,  inf,  inf,  inf],
       [ inf,  inf,  inf,  10.,  75.,  inf,  inf,  inf,  inf,  inf],
       [ inf,  inf, -15.,  inf,  inf,  inf,  inf,  inf,  inf,  inf],
       [ inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf, 100.],
       [ inf,  inf,  inf,  inf,  25.,  inf,   5.,  inf,  50.,  inf],
       [ inf,  inf,  inf,  inf,  inf,  inf,  inf, -50.,  inf,  inf],
       [ inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf, -10.,  inf],
       [ inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf],
       [ inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf,  inf]])
Solution trouvé en utilisant Bellman-Ford
{1: [5.0, [0, 1]],
 2: 'Sommet joignable depuis 0 par un chemin dans le graphe G, mais pas de '
    "plus court chemin (présence d'un cycle négatif)",
 3: 'Sommet joignable depuis 0 par un chemin dans le graphe G, mais pas de '
    "plus court chemin (présence d'un cycle négatif)",
 4: 'Sommet joignable depuis 0 par un chemin dans le graphe G, mais pas de '
    "plus court chemin (présence d'un cycle négatif)",
 5: [35.0, [0, 1, 5]],
 6: [40.0, [0, 1, 5, 6]],
 7: [-10.0, [0, 1, 5, 6, 7]],
 8: [-20.0, [0, 1, 5, 6, 7, 8]],
 9: 'Sommet joignable depuis 0 par un chemin dans le graphe G, mais pas de '
    "plus court chemin (présence d'un cycle négatif)"}
```

FIGURE 12 – Parcours produit pour une matrice avec des pondérations négatives

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

La liste ordonnée des flèches dans l'algorithme de Bellman-Ford peut être déterminée en utilisant différentes méthodes, à savoir le parcours séquentiel, le parcours en largeur ou le parcours en profondeur. Nous souhaitons donc examiner si le choix du parcours affecte le temps de calcul de l'algorithme de Bellman-Ford.

Pour ce faire, nous avons ajouté un paramètre dans l'algorithme de Bellman-Ford qui indique le choix de parcours : 1 pour le parcours séquentiel, 2 pour le parcours en largeur et 3 pour le parcours en profondeur. Si un autre nombre est passé en paramètre, le programme affiche "type invalide". Ensuite, la fonction ajoute des sommets non-identifiés en utilisant le parcours choisi dans la liste de sommets.

Nous avons également ajouté un compteur dans le programme qui est renvoyé à la fin pour calculer le nombre de tours effectués. Afin de comparer les résultats, nous avons utilisé un graphe de grande taille avec $n = 500$ et une proportion de flèches de $p = 0,01$.

Nous avons également ajouté un programme qui génère un grand graphe, puis exécute l'algorithme de Bellman-Ford 50 fois pour chaque type de parcours. Il enregistre le nombre de tours effectués à chaque exécution. Ensuite, il calcule la moyenne des tours pour chaque type de parcours et affiche ces moyennes dans un diagramme en barres. Voici le programme ajouté :

```
def TestBellmanFordParcours():
    M = graphe2(500, 0.01, 1, 100)
    listeSeq = []
    listeLargeur = []
    listeProfondeur = []
    for i in range(50):
        resultatSeq, nbToursSeq = BellmanFord(M, 0, 1)
        resultatLargeur, nbToursLargeur = BellmanFord(M, 0, 2)
        resultatProfondeur, nbToursProfondeur = BellmanFord(M, 0, 3)
        listeSeq.append(nbToursSeq)
        listeLargeur.append(nbToursLargeur)
        listeProfondeur.append(nbToursProfondeur)
    typeParcours = ['Sequentiel', 'En Largeur', 'En Profondeur']
    avgNbToursSeq = st.mean(listeSeq)
    avgNbToursLargeur = st.mean(listeLargeur)
    avgNbToursProfondeur = st.mean(listeProfondeur)

    nbTours = [avgNbToursSeq, avgNbToursLargeur, avgNbToursProfondeur]
    plt.title('Nombre de tours effectues par type de parcours')
    plt.xlabel('Type de parcours')
    plt.ylabel('Nombre de tours effectues')
    plt.bar(typeParcours, nbTours, width=0.5, color=["red", "blue",
        "green"])
    plt.show()
```

Avec le graphique obtenu, nous observons que le nombre de tours effectués en uti-

lisant le parcours séquentiel est de 8, en parcours en profondeur est de 7, et en parcours en largeur est de 6.

Nous pouvons constater que le parcours en largeur effectue généralement moins de tours que le parcours séquentiel et le parcours en profondeur. On peut donc conclure que le parcours en largeur est plus performant que les autres.

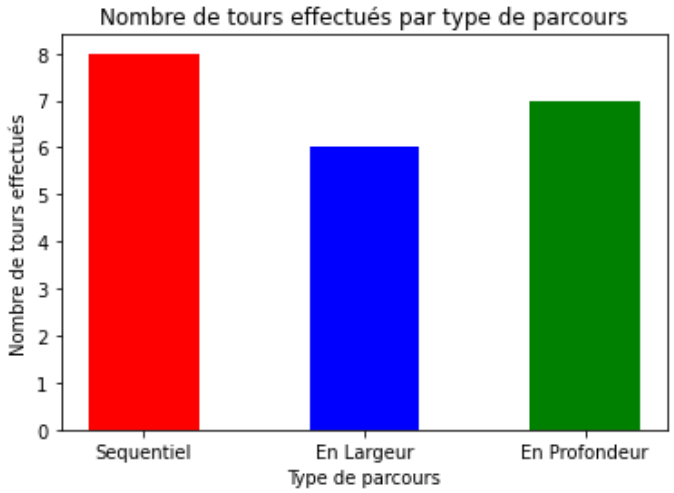

FIGURE 13 – Nombre de tours effectués pour chaque type de parcours pour l'algorithme de Bellman-Ford

6 Comparaison expérimentale des complexités

6.1 Deux fonctions "temps de calcul"

Pour calculer le temps d'exécution de chaque algorithme, nous avons créé deux fonctions $TempsDij(n)$ et $TempsBF(n)$.

```
def TempsDij(n, p):
    M = graphe2(n, p, 1, 100)
    debut = time.perf_counter()
    Dijkstra(M, 0)
    fin = time.perf_counter()
    return fin - debut

def TempsBellmanFord(n, p):
    M = graphe2(n, p, 1, 100)
    debut = time.perf_counter()
    BellmanFord(M, 0, 2)
    fin = time.perf_counter()
    return fin - debut
```

Le fonctionnement du programme
Les paramètres :

- **n** : Taille de la matrice
- **p** : Proportion d'arêtes dans le graphe

La fonction commence par la génération d'un graphe aléatoire de taille n avec une pondération p. En utilisant la fonctions $perf\_counter()$ du module **time**, le temps d'exécution de l'algorithme de Dijkstra est calculé à partir des enregistrements du début et de la fin de son exécution. La fonction retourne le temps d'exécution obtenu.

Résultat d'exécution
Pour calculer le temps de calcul pour les deux algorithmes, nous avons utilisé une matrice générée aléatoirement de taille $n = 1000$ avec une pondération de 0.5. Nous avons utilisé le parcours en largeur pour l'algorithme de Bellman-Ford car c'est le plus efficace parmi les 3 types de parcours.

```
Temps de calcul de l'algorithme de Dijkstra :  3.68 s
Temps de calcul de l'algorithme de Bellman-Ford :  56.73 s
```

FIGURE 14 – Temps d'exécution de l'algorithme Dijkstra et Bellman-Ford

D'après les résultats de l'exécution des deux fonctions, nous pouvons constater que le temps de calcul avec l'algorithme de Dijkstra est bien plus rapide que celui avec l'algorithme de Bellman-Ford.

6.2 Comparaison et identification des deux fonctions temps

6.2.1 Comparaison des temps de calculs avec une proportion p de flèches fixes

Afin de comparer le temps d'exécution et la complexité des deux algorithmes, nous avons besoin d'une fonction qui récupère une liste des valeurs du temps d'exécution pour des matrices de taille 2 à 200. Pour ce faire, nous avons écrit les fonctions suivantes, $ComplexiteDij(p)$ et $ComplexiteBF(p)$, qui prennent comme paramètre la pondération des matrices. Ces deux fonctions renverront une liste du temps d'exécution ainsi qu'une liste de la taille des matrices utilisées afin de construire un graphique par la suite.

```
def ComplexiteDij(p):
    n = []
    temps = []
    for i in range(2, 200):
        M = graphe2(i,p, 1, 100)
        debut = time.perf_counter()
        Dijkstra(M, 0)
        fin = time.perf_counter()
        n.append(i)
        temps.append(fin - debut)
    return n, temps

def ComplexiteBF(p):
    n = []
    temps = []
    for i in range(2, 200):
        M = graphe2(i,p, 1, 100)
        debut = time.perf_counter()
        BellmanFord(M, 0, 1)
        fin = time.perf_counter()
        n.append(i)
        temps.append(fin - debut)
    return n, temps
```

Nous avons également fait une autre fonction, $dessinerGrapheComplexite(p)$, qui prend une pondération p comme paramètre. Pour cette partie, nous avons utilisé la pondération de 0.5 pour toutes les générations de matrices.

Cette fonction génère d'abord des données en appelant $ComplexiteDij(p)$ et $ComplexiteBF(p)$ pour obtenir les nombres de sommets et les temps d'exécution des deux algorithmes. Il trace ensuite les graphes de complexité en utilisant $plt.plot()$, puis affiche le graphe avec $plt.show()$.

Ensuite, il transforme les données en échelle logarithmique avec $np.log$, ajuste des lignes droites aux données transformées avec $np.polyfit$. Le code trace un graphe en échelle log-log avec $plt.loglog$ pour les deux algorithmes et y ajoute les lignes ajustées. Comme le graphe précédent, le graphe en échelle log-log est affiché avec $plt.show()$.

Enfin, les pentes ajustées pour chaque algorithme sont affichées, montrant l'ordre de la croissance polynomiale.

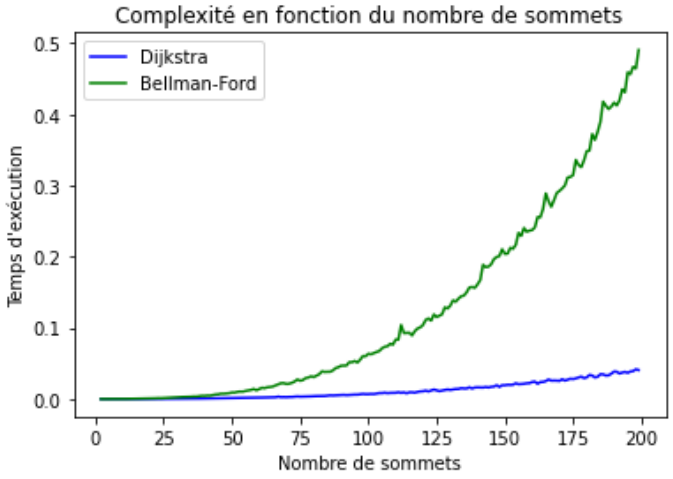

FIGURE 15 – Complexité en fonction de nombre de sommets

Selon la représentation graphique, nous observons que l'algorithme de Dijkstra est plus performant que celui de Bellman-Ford, car il présente un temps d'exécution plus faible que Bellman-Ford lorsque le nombre de sommets augmente. Bien que Bellman-Ford soit plus lent, il présente l'avantage de pouvoir fonctionner sur des graphes comportant des arêtes avec un poids négatif.

Nous observons également que les courbes du graphe sont approximativement de type cn^a, cela signifie que le temps d'exécution est proportionnel à une puissance de n, où a est l'ordre de croissance polynomiale de la fonction. Pour le prouver, si nous représentons cette fonction sur un graphique en coordonnées log-log, nous prenons le logarithme des deux côtés de l'équation :

$$log(t(n)) = log(c.n^a)$$

Que nous pouvons réécrire ainsi :

$$log(t(n)) = a.log(n) + log(c)$$

En comparant l'équation avec une droite $y = mx + b$, nous pouvons constater que a est la pente de la droite sur un graphe en coordonnées log-log et $log(c)$ est l'ordonnée à l'origine. Donc, puisque les courbes que nous avons observées ressemblent à cn^a, il serait intéressant de transformer le graphe en échelle logarithmique afin de calculer la croissance polynomiale de chaque courbe. C'est ce que la fonction $dessinerGrapheComplexite(p)$ fait ensuite pour générer le graphe suivant.

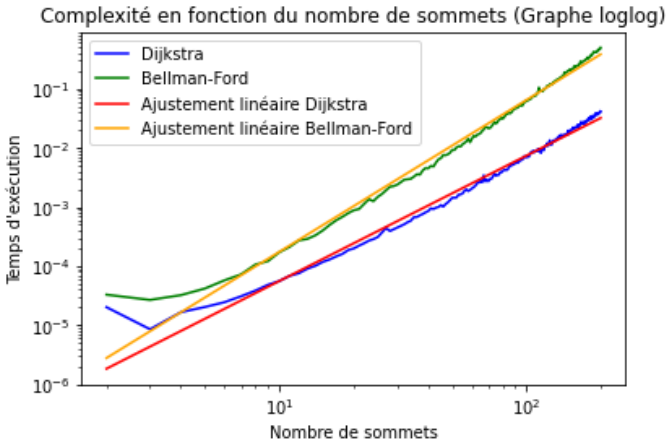

FIGURE 16 – Complexité en fonction de nombre de sommets en échelles log-log

Pour trouver les ajustements linéaires dans le graphique et la croissance polynomiale, nous avons utilisé le code suivant :

```
# Transformation en log et ajustement linaire
log_nD=np.log(nDij)
log_tempsD=np.log(tempsDij)
# Ajuste une ligne droite aux donnes
coefficientsDij = np.polyfit(log_nD,log_tempsD,1)
slopeD=coefficientsDij[0] # La pente de la ligne est l'exposant
```

La fonction *np.polyfit* ajuste un polynôme aux données fournies en utilisant la méthode des moindres carrés. Elle prend un tableau des coordonnées x, un tableau des coordonnées y et un degré du polynôme à ajuster comme paramètres. En sortie, nous avons un tableau de coefficients du polynôme, commençant par le coefficient de plus haut degré. Donc, pour trouver la pente (la croissance polynomiale), il suffit de prendre le dernier argument du tableau en sortie la ligne $slopeD = coefficientsDij[0]$.

Nous avons trouvé un ordre de croissance polynomiale de 2,12 pour l'algorithme de Dijkstra et un ordre de croissance polynomiale de 2,57 pour l'algorithme de Bellman-Ford.

Pour vérifier les chiffres obtenus, nous avons construit la fonction Python suivante permettant de calculer la valeur de a en utilisant la méthode des moindres carrés, que nous avons apprise dans les méthodes statistiques en particulier dans la ressource R2.08.

```
def croissancePolynomiale(x,y):
    covariance = np.cov(x,y,bias=True)[0][1]
    variance = np.var(x)
    return covariance/variance
```

Dans l'ajustement linéaire, pour trouver la pente a, nous avons besoin du rapport entre la covariance et la variance des données. Cette fonction Python effectue précisément cette opération. La covariance mesure comment deux variables changent

ensemble, tandis que la variance mesure la dispersion des données par rapport à leur moyenne. En divisant la covariance par la variance, nous obtenons un indicateur de la pente de la relation linéaire entre les variables.

En utilisant la fonction, nous avons réussi à trouver les mêmes résultats avec une différence de 1×10^{-15} pour l'algorithme de Dijkstra et 1.78×10^{-15} pour Bellman-Ford. Ces écarts infirmes sont souvent dus à des erreurs d'arrondi ou à des approximations numériques dans les calculs. Les résultats peuvent donc être considérés comme fiables.

```
Valeur de a (Dijkstra) en utilisant la fonction
croissancePolynomiale :  2.122278503381224
Valeur de a (Bellman-Ford) en utilisant la fonction
croissancePolynomiale :  2.5726840951398677
a (Dijkstra) :  2.122278503381225
a (Bellman-Ford) : 2.57268409513987
```

FIGURE 17 – Les ordres de croissance polynomiale obtenus

6.2.2 Comparaison des temps de calculs avec une proportion p de flèches diminuant

Ensuite, nous avons effectué les mêmes tests de calculs avec une proportion de flèches diminuant avec n, Pour cette partie nous avons utilisé $p = 1/n$. Voici le code que nous avons utilisé :

```python
def ComplexiteDijPDiminuant():
    n = []
    temps = []
    for i in range(2, 200):
        M = graphe2(i,1/i, 1, 100)
        debut = time.perf_counter()
        Dijkstra(M, 0)
        fin = time.perf_counter()
        n.append(i)
        temps.append(fin - debut)
    return n, temps

def ComplexiteBFPDiminuant():
    n = []
    temps = []
    for i in range(2, 200):
        M = graphe2(i,1/i, 1, 100)
        debut = time.perf_counter()
        BellmanFord(M, 0, 1)
        fin = time.perf_counter()
        n.append(i)
        temps.append(fin - debut)
    return n, temps
```

Avec la même méthode, nous avons constaté que la performance de Dijkstra est toujours plus efficace que celle de Bellman-Ford.

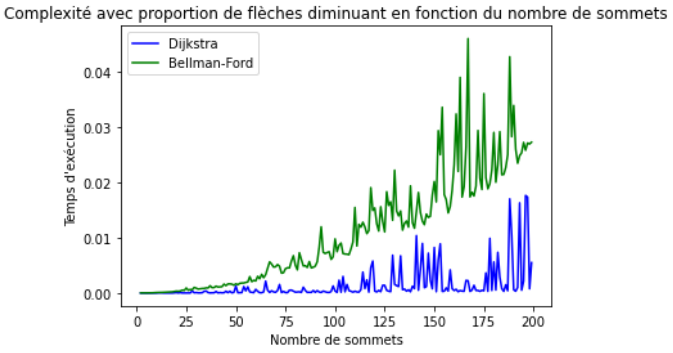

FIGURE 18 – Complexité avec une proportion de flèches diminuant en fonction de nombre de sommets

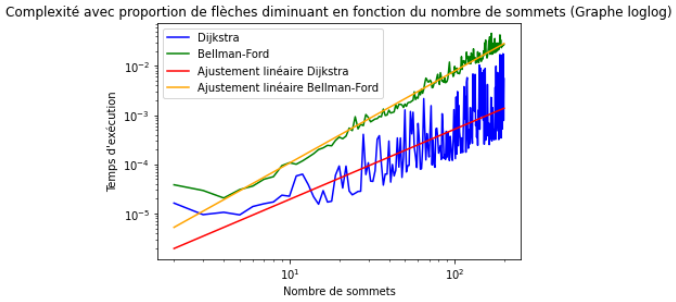

FIGURE 19 – Complexité avec une proportion de flèches diminuant en fonction de nombre de sommets en coordonnées log-log

Pour une proportion de flèches proportionnelle à la taille de la matrice, nous avons trouvé un ordre de croissance polynomiale de 1,21 pour l'algorithme de Dijkstra et un ordre de croissance polynomiale de 1,77 pour l'algorithme de Bellman-Ford.

```
Valeur de a (Dijkstra) en utilisant la fonction
croissancePolynomiale :  1.2058132936523882
Valeur de a (Bellman-Ford) en utilisant la fonction
croissancePolynomiale :  1.7743945544974429
a (Dijkstra) :  1.205813293652389
a (Bellman-Ford) : 1.774394554497443
```

FIGURE 20 – Les ordres de croissance polynomiale obtenus

Nous avons effectué le même processus que dans la section précédente et nous avons vérifié nos résultats en utilisant la fonction $croissancePolynomiale(x, y)$. Nous avons obtenu le même résultat avec une différence très minime.

6.3 Conclusion

La comparaison des temps de calculs avec une proportion fixe de flèches pour les algorithmes de Dijkstra et Bellman-Ford révèle une performance supérieure de Dijkstra. En effet, pour des graphes dont la proportion de flèches est proportionnelle à la taille de la matrice, nous avons trouvé un ordre de croissance polynomiale de 2,12 pour Dijkstra et de 2,57 pour Bellman-Ford.

Cela signifie que l'efficacité de Dijkstra est maintenue même lorsque la taille de la matrice augmente. En revanche, pour une proportion de flèches diminuant, Dijkstra demeure plus performant que Bellman-Ford. Dans ce cas, nous avons obtenu un ordre de croissance polynomiale de 1,21 pour Dijkstra et de 1,77 pour Bellman-Ford.

Il est important de noter que même si l'algorithme de Bellman-Ford est plus lent, il conserve une utilité significative dans le cas de graphes comportant des arêtes de pondération négative.

Ces situations exigent une attention particulière, car Dijkstra ne peut pas traiter les arêtes avec un poids négatif, tandis que Bellman-Ford peut. Ainsi, selon le type de graphe considéré, le choix de l'algorithme à utiliser dépendra de plusieurs facteurs, notamment la présence ou non d'arêtes de pondération négative et la complexité du graphe.

Dans les cas où le graphe est non pondéré ou contient uniquement des arêtes de pondération positive, Dijkstra est généralement le choix à privilégier en raison de sa meilleure performance. En revanche, pour les graphes comportant des arêtes de pondération négative, Bellman-Ford est souvent privilégié malgré sa lenteur, car il peut gérer ces situations spécifiques.

7 Test de forte connexité

Un graphe orienté est fortement connexe s'il ne possède qu'une seule composante connexe, et une composante connexe à la propriété suivante : *"deux sommets distincts s et s' sont dans une même composante fortement connexe s'il existe un chemin de s vers s' et un chemin de s' vers s dans G"*.
Donc si sa matrice transitive n'a que des 1, cela signifie que chaque sommet s a un chemin vers s' et inversement.

Voici la fonction Python $fc(M)$:

```
def fc(M):
    k=np.shape(M)[0]
    N=M
    P = M
    P=reduction2(P)
    for i in range(k-1):
        P=reduction(np.dot(M,P))
        N = reduction(N + P)
    O=np.ones((len(N),len(N)))
    return np.array_equal(O,N)
```

Le fonctionnement du programme
Les paramètres :

— **M** : la matrice à tester

La fonction commence par utiliser la fonction $reduction2(P)$ qui change les éléments dans la matrice qui sont ∞ vers des zéros. La fonction $reduction(P)$ sert à réduire des coefficients non nuls dans la matrice à 1. Ensuite, la fermeture transitive de la matrice est calculée. Si la fermeture transitive est égale à une matrice identité, la fonction retourne vrai, sinon faux.

Résultat d'exécution
En affichant des traces dans la fonction, nous avons trouvé le résultat suivant :

```
Matrice originale :
array([[0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]])
Fermeture transitive de la matrice :
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
La matrice est fortement connexe
```

FIGURE 21 – Exemple avec une matrice fortement connexe

```
Matrice originale :
array([[0., 0., 0., 0., 0.],
       [6., 0., 5., 2., 2.],
       [0., 5., 0., 0., 5.],
       [1., 2., 0., 0., 1.],
       [0., 2., 5., 1., 0.]])
Fermeture transitive de la matrice :
array([[0., 0., 0., 0., 0.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
La matrice n'est pas fortement connexe
```

FIGURE 22 – Exemple avec une matrice pas fortement connexe

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

La fonction $testStatFc(n)$ est utilisée pour évaluer le pourcentage de graphes de taille n qui sont fortement connexes. Elle fonctionne en générant des graphes aléatoires de taille n, avec une proportion de 50% de flèches. Pour chaque graphe, la fonction vérifie s'il est fortement connexe. Si c'est le cas, elle incrémente le compteur. À la fin, elle retourne le pourcentage de graphes fortement connexes parmi les 200 générés.

```
def testStatFc(n)
    connexe = 0
    for i in range(200):
        M = graphe(n, 1, 10)
        if fc(M):
            connexe += 1
    return connexe / (200) * 100
```

L'affirmation des matrices de taille n, avec une proportion p = 50% de 1 (et 50% de 0), donne presque toujours un graphe fortement connexe est vraie si n est égal à 13.

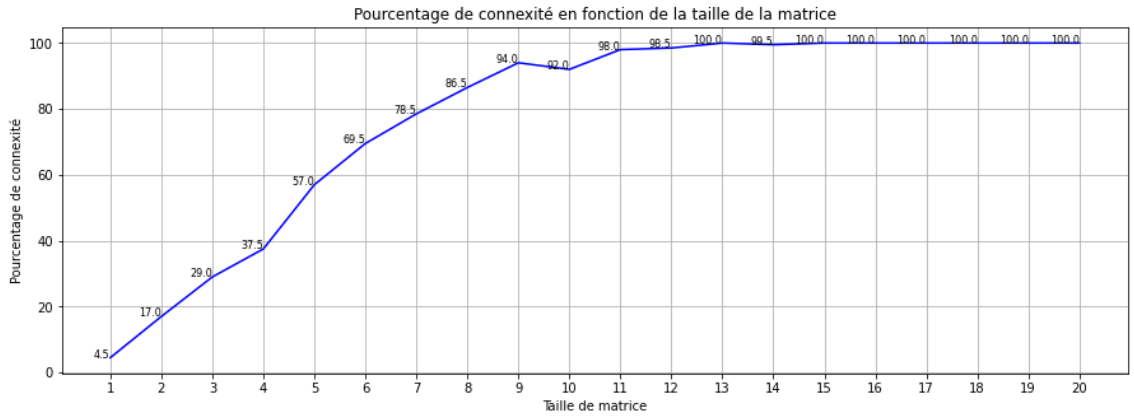

FIGURE 23 – Pourcentage de connexité en fonction de la taille de la matrice

9 Détermination du seuil de forte connexité

La fonction $testStatFc2(n, p)$ est une version améliorée de $testStatFc(n)$, permettant de spécifier la proportion p de 1 dans la matrice. Elle génère des graphes de taille n avec cette proportion p et vérifie s'ils sont fortement connexes.

```
def testStatFc2(n,p)
    connexe = 0
    for i in range(200):
        M = graphe2(n,p, 1, 2)
        if fc(M):
            connexe += 1
    return (connexe / 200) * 100
```

La fonction $seuil(n)$ détermine le seuil de forte connexion en ajustant progressivement la proportion p jusqu'à ce que le pourcentage de graphes fortement connexes descende en dessous de 99%. Une fois atteint, elle retourne et imprime la proportion correspondante.

```
def seuil(n)
    p=0.5
    while testStatFc2(n, p)>=99:
        p -= 0.01
    print (p + 0.01)
    return (p + 0.01)
```

10 Étude et identification de la fonction seuil

10.1 Représentation graphique de seuil(n)

Dans ce graphique, nous avons représenté la variation du seuil en fonction de la taille de matrice, spécifiquement sur l'intervalle [10,40]. En observant le graphique, nous remarquons que la courbe est globalement décroissante. Cela signifie que, de manière générale, plus la taille de la matrice augmente, plus le seuil diminue, même si il y a des fluctuations.

Nous pouvons nous attendre à ce résultat, car plus la taille de la matrice augmente, le nombre de sommets et la probabilité de création d'arêtes augmentent. Bien que plus le flèches soient nécessaires pour une matrice plus grande, le nombre potentiel de connexions croît beaucoup plus rapidement que le nombre d'arêtes nécessaires pour assurer une forte connectivité.

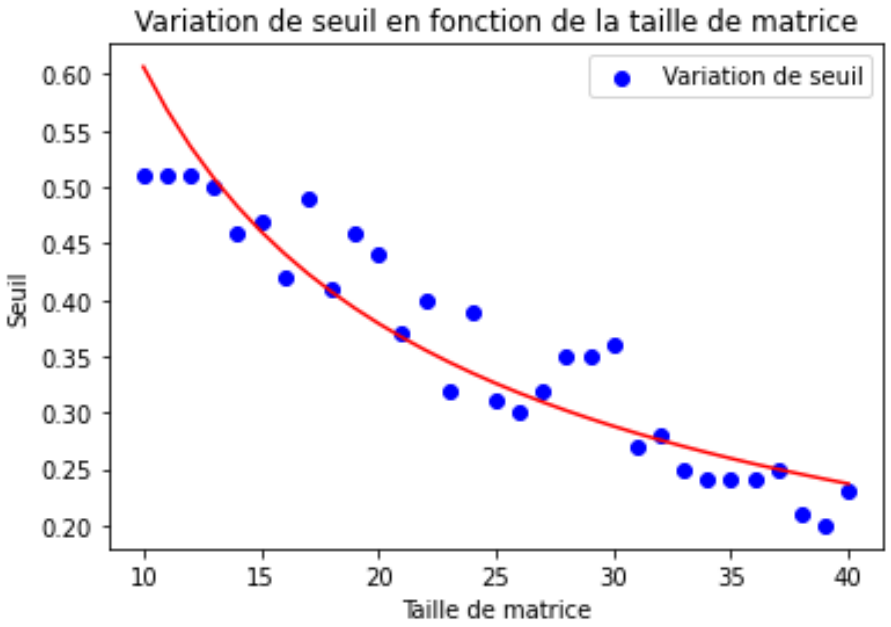

FIGURE 24 – Variation de seuil en fonction de la taille de matrice

10.2 Identification de la fonction seuil(n)

Nous observons que le graphique semble être asymptotiquement une fonction puissance. Nous avons formulé l'hypothèse qu'il s'agit d'une fonction puissance de la forme cn^a. Pour identifier la valeur de a, nous avons utilisé la fonction mentionnée dans la partie 6.1. Pour la valeur de c, nous avons utilisé la fonction suivante :

```
def valeurC(x,y,a):
    return np.mean(y)-(a*np.mean(x))
```

Cette fonction Python calcule la valeur de c est déterminée en soustrayant le produit de la moyenne de x et de a de la moyenne de y.

Résultat d'exécution

```
Valeur de a :  -0.6780019144424826
Valeur de c :  1.061283272554135
```

FIGURE 25 – Valeurs de a et c obtenues

Nous obtenons une valeur d'environ -0.68 pour a et une valeur d'environ 1.06 pour c en utilisant les fonctionnes mentionnées. Avec les valeurs obtenues, nous avons établi l'équation de la droite de régression.

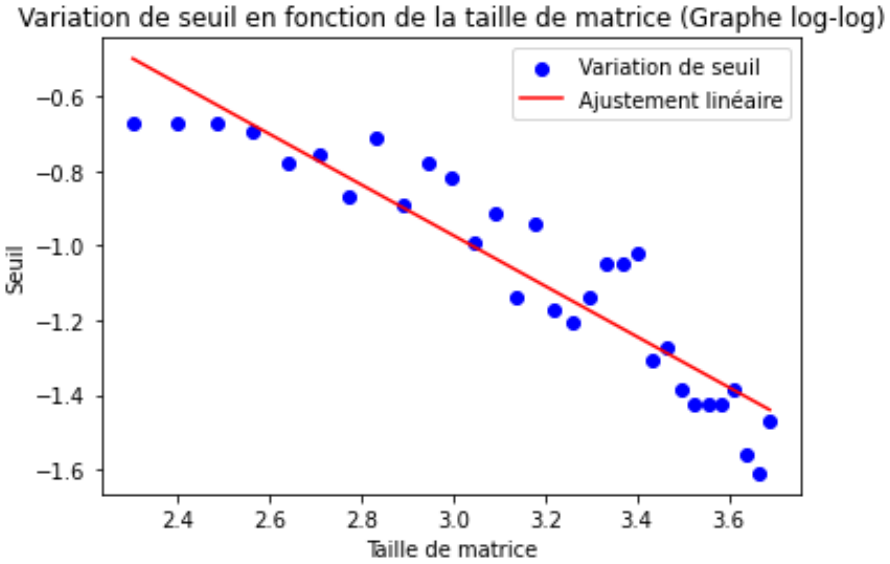

FIGURE 26 – Graphe log-log de la variation de seuil en fonction de la taille de matrice

Ainsi, nous pouvons conclure que nous avons obtenu une fonction de la forme $f(x) = -0,68x + 1,06$ avec les méthodes utilisées ci dessus. Ces résultats montrent que la variation du seuil peut être modélisée par une fonction linéaire asymptotique.

IUT Informatique Rangueil

133b, Avenue de Rangueil
31400 Toulouse Cedex 4 - France
www.iut.univ-tlse3.fr