Amélioration du temps d'execution de l'algorithme Bellman-Ford pour des graphes orientés pondérés

Robin Soares
Paul-Tiberiu Iordache

8 Décembre 2023

Introduction

Dans le cadre de l'unité d'enseignement MOGPL (Modélisation, optimisation, graphes, programmation linéaire) à *Sorbonne Université*, nous sommes engagés dans l'étude approfondie et l'amélioration des performances de l'algorithme de Bellman-Ford.

Notre objectif principal consiste à introduire une phase de prétraitement au sein de l'algorithme Bellman-Ford afin de sélectionner judicieusement l'ordre dans lequel les sommets d'un graphe orienté pondéré seront examinés. Cette démarche vise à réduire le nombre d'itérations nécessaires à l'algorithme pour converger vers la solution optimale dans le cas où les exemples ne sont pas les pires. Ce faisant, nous autorisons un coût de prétraitement plus important (mais toujours polynomial) afin de réduire le temps d'exécution par instance et donc de réduire le nombre d'itérations.

Pour mener à bien ce projet, nous avons opté pour le langage de programmation Python. Notre choix s'est basé sur l'utilité de la bibliothèque NumPy, ainsi que sur les avantages offerts par les structures de données telles que les dictionnaires, que nous avons sélectionnés comme moyen de représenter nos graphes orientés dans le cadre de ce projet.

Contents

1 Contextualisation

1.1 Définition d'un graphe orienté pondéré G = (V,E)

Soit $G = (V, E)$ un graphe orienté pondéré.
V est l'ensemble des n sommets du graphe et E l'ensemble de ses m arcs, chaque arc ayant un poids.

2 Graphes

2.1 Choix d'implémentation de la structure du graphe G = (V,E) orienté pondéré

Pour l'implémentation de la structure d'un graphe non orienté, nous avons choisi de créer une classe Graph où nous stockons les sommets dans un ensemble **V** et l'ensemble d'arcs dans un dictionnaire **E** dont la clé est l'identifiant du sommet, et les valeurs sont des ensembles de tuples $(sommet, poids)$ représentant les sommets adjacents ainsi que les poids associés aux arcs. Par exemple, si l'arc (i, j) existe avec un poids w, alors dans notre dictionnaire, cela ressemblerait à :

{ i : {(j,w), ... }, ... }

3 Méthodes approchées

3.1 Algorithme Bellman-Ford

Soit $G = (V, E)$ un graphe orienté pondéré avec $|V| = n$ sommets et $|E| = m$ arcs. Pour un arc $e \in E$, nous notons w_e le poids de e. Certains poids peuvent être négatifs, mais le graphe ne contient aucun circuit de poids négatif. Étant donné un sommet particulier $s \in V$, l'algorithme de Bellman-Ford calcule le plus court chemin de s à chaque sommet $v \in V$. L'algorithme maintient une borne supérieure, d_v, sur la plus courte distance de s à v. Il commence par fixer $d_v = \infty$ (sauf $d_s = 0$) et procède en itérant sur tous les sommets et en considérant tous les arcs entrants, c'est-à-dire, pour un sommet v, en fixant

$$d_v = \min_{(y,v)\in E}(d_y + w(y, v))$$

Il est connu que si G n'a aucun circuit de poids négatif, l'algorithme Bellman-Ford converge après au plus $(n - 1)$ itérations à travers l'ordre complet des sommets.

Comme nous avons dit dans l'introduction, notre objectif est de diminuer le nombre d'itérations de l'algorithme. Pour déterminer le nombre d'itérations, nous utilisons une variable *converged*. Au début de notre boucle principale, qui itère sur tous les sommets, nous posons $converged = True$. Puis chaque fois quand la valeur de d_v se modifie, nous changeons la valeur de la variable: $converged = False$. Donc, nous allons sortir de la boucle lorsque la valeur de d_v ne change plus, ce qui signifie que notre algorithme a convergé.

Nous voulons préciser que notre algorithme sert également à determiner s'il y a un cycle négatif dans le graphe en entrée. Dans ce cas, nous arretons l'éxecution de l'algorithme car Bellman-Ford ne trouve pas une bonne solution si le graphe contient un cycle négatif. De plus, notre algortithme renvoie la distance du sommet s vers chaque sommet v, le chemin suivi jusqu'au sommet v, ainsi que le nombre d'itérations que l'algorithme à pris pour trouver les plus courts chemins.

Nous voulons parler également de la difference de nos deux algorithmes de Bellman-Ford. Le premier, `bellmanFord(self, start)`, sera utilisé dans la plupart des cas pour détérminer l'existance d'un cycle négatif dans le graphe mais également pour trouver un plus court chemin tandis que le déuxième, `bellmanFord_gloutonFas(self, start, ordre)`, nous permet de trouver le plus court chemin à partir de l'ordre de sommets passé en parametre.

3.2 Etape de prétraitement - choix d'un ordre de sommets $<_{tot}$ et MVP

3.2.1 Ordre $<_{tot}$

Pour pouvoir être plus précis sur le nombre d'itérations il faut choisir un ordre spécifique de sommets. Étant donné un graphe G et deux sommets s et v, soit $P_v = (s, x_1, x_2, \ldots, v)$ le plus court chemin de s à v. Soit $<_{tot}$ un ordre sur V où, pour deux sommets u et v, nous avons $u <_{tot} v$ si u apparaît avant v dans cet ordre. Pour un chemin P, notons $\text{dist}(<_{\text{tot}}, P)$ le nombre d'arcs dans P inversés par $<_{tot}$, c'est-à-dire:

$$\text{dist}(<_{\text{tot}}, P) = |\{(u, v) \in P \mid v <_{\text{tot}} u\}|$$

Étant donné un graphe G, le temps d'exécution de l'algorithme de Bellman-Ford en utilisant l'ordre $<_{tot}$ pour traiter les sommets est en

$$O(m \cdot \max_{v \in V} \text{dist}(<_{\text{tot}}, P_v))$$

Ce résultat met formellement en évidence l'avantage d'avoir des sommets dans un bon ordre.

3.2.2 MVP (Minimum Violation Permutation)

Étant donné un ensemble de sommets V et un ensemble de r chemins $P_1, P_2, \ldots, P_r$, où chaque $P_i \subseteq V$, trouver un ordre total $<_{\text{tot}}$ sur V qui minimise l'objectif

$$\max_{i \in [r]} \text{dist}(<_{\text{tot}}, P_i).$$

Relation de MVP avec le problème du plus court chemin. Considérons une solution $<_{\text{tot}}$ de MVP pour l'ensemble des chemins $<_{\text{tot}}$ de MVP pour l'ensemble des chemins

$$\bigcup_{i \in [k]} \bigcup_{v \in V} P_i^v.$$

Cette solution réduit le temps d'exécution dans le pire cas de l'algorithme de Bellman-Ford dans tous les graphes G. Le nombre maximum d'itérations sur n'importe quelle instance parmi ces k instances est au plus $\max_{i \in [k], v \in V} \text{dist}(<_{\text{tot}}, P_{i,v})$, qui est égal à la valeur de la solution $<_{\text{tot}}$ de l'instance de MVP.

Pour résoudre MVP, nous allons considérer une méthode gloutonne que l'on appellera `GloutonFas` dont le code peut être vu dans le fichier Graph.py. Elle prend en entrée un graphe correspondant à l'union des plus courts chemins et sélectionne de manière gloutonne les sommets pour construire un ordre linéaire des sommets.

4 Efficacité de l'ordre donné par la méthode GloutonFas (MVP)

4.1 Efficacité sur des instances de 5 sommets avec un nombre fixe de graphes pondérés G_i

Nous allons maintenant parler de l'efficacité de l'ordre donné par la méthode GloutonFas. Pour l'instant, nous testerons l'efficacité d'un ordre de 5 sommets, pour 3 graphes G_i et 1 graphe de test H.

4.1.1 Génération de graphes pondérés

Pour évaluer l'impact de l'ordre retourné par `GloutonFas` sur la réduction du nombre d'itérations, nous avons mis au point la fonction `random_graph_unary_weight(n, p)`. Cette fonction nous permet de générer un graphe orienté de n sommets avec une probabilité p d'avoir chaque arc. Puis, à l'aide de la fonction `weighed_graph(self, w)`, à partir du graphe G, nous allons contruire 3 graphes pondérés G1, G2, G3 chacun donné par une fonction de poids différente. De plus, nous allons générer encore un graphe H, qui est celui de test, qui sera donné par une autre fonction de poids. La seule difference entre les 4 graphes et le graphe de base est que les arcs seront pondérés par des fonctions des poids différentes. Pour ces graphes nous testons avec la fonction `bellmanFord(self, start)` si les graphes contiennent de cycle négatif. Nous avons choisi les graphes suivants avec 5 sommets:

```
Graphe random w1:
Sommets: {0, 1, 2, 3, 4}
Arcs: {0: set(), 1: {(0, -2), (4, -3), (2, 2)}, 2: {(0, 2)}, 3: set(), 4: {(1, 3), (0, -2), (2, -1)}}

Graphe random w2:
Sommets: {0, 1, 2, 3, 4}
Arcs: {0: set(), 1: {(2, -1), (0, -2), (4, 3)}, 2: {(0, 1)}, 3: set(), 4: {(0, 1), (1, 1), (2, 6)}}

Graphe random w3:
Sommets: {0, 1, 2, 3, 4}
Arcs: {0: set(), 1: {(2, -3), (0, -2), (4, -3)}, 2: {(0, -3)}, 3: set(), 4: {(1, 4), (2, -4), (0, 5)}}

Graphe random H (tests):
Sommets: {0, 1, 2, 3, 4}
Arcs: {0: set(), 1: {(2, -2), (0, 4), (4, -10)}, 2: {(0, 12)}, 3: set(), 4: {(0, -5), (1, 11), (2, 2)}}
```

(a) Génération des graphes G1, G2, G3 et H

Pour cette partie nous voulons préciser que nous avons codé encore une fonction qui sera utile pour nos tests. Elle sert a choisir un sommet de départ, celui pour lequel nous calculons le plus court chemin vers les autres sommets. Cette fonction s'appelle `can_reach_half(self)` et elle nous sert à choisir un sommet de départ qui peut atteindre au moins $|V|/2$ sommets.

4.1.2 Union des arborescences des plus courts chemins

Cette étape nous servira pour déterminer un ordre de sommets $<_{tot}$ qui nous aidera pour diminuer le nombre d'itérations. Tout d'abord, nous allons appliquer la fonction `bellmanFord(self, start)` pour pouvoir récuperer les plus courts chemin à partir d'un sommet qui peut atteindre au moins $|V|/2$ sommets. La fonction qui nous permet d'obtenir l'union des arborescences de plus courts chemins s'appelle `union_path(list_path)` et elle prend en paramètre la liste de plus courts chemins des graphes pondérés et renvoie l'union de ces plus courts chemins. Nous avons choisi de représenter

l'union comme un dictionnaire de listes de listes. Pour les graphes que nous venons de créer au point 4.1.1, l'union est la suivante:

T = 0: [[1, 4, 0], [1, 0], [1, 4, 2, 0]], 1: [[1]], 2: [[1, 4, 2], [1, 2]], 3: [[None, 3]], 4: [[1, 4]]

4.1.3 Détermination de l'ordre $<_{tot}$ à partir de l'union des arborescences des plus courts chemins T

Pour pouvoir déterminer l'ordre $<_{tot}$, nous allons appliquer l'algorithme GloutonFas sur l'union des arborescences des plus courts chemins T. Pour pouvoir le faire, comme GloutonFas prend en entrée un graphe et pas l'union T, nous avons développé une fonction, `from_tree_to_graph(T)`, qui transforme T dans un graphe. Ainsi, en appliquant GloutonFas sur le graphe obtenu, nous allons pouvoir déterminer l'ordre $<_{tot}$. Le graphe obtenu à partir de l'union et l'ordre $<_{tot}$ sont les suivants:

Arcs: 0: set(), 1: (0, 1), (4, 1), (2, 1), 2: (0, 1), 3: set(), 4: (0, 1), (2, 1)

$<_{tot}$: [1, 3, 4, 2, 0]

4.1.4 Bellman-Ford appliqué sur l'ordre $<_{tot}$ déterminé au point 4.1.3

Maintenant nous allons appliquer l'algorithme bellmanFord_gloutonFas sur l'ordre $<_{tot}$ et sur le graphe de test H. Le sommet source pour le graphe H est le sommet 1, car il est l'un de sommets qui peut atteindre au moins |V|/2 sommets. Nous avons obtenu:

Bellman-Ford en partant du sommet 1: 0: -15, 1: 0, 2: -8, 3: inf, 4: -10
Arbre des plus courts chemins: 0: [1, 4, 0], 1: [1], 2: [1, 4, 2], 3: [None, 3], 4: [1, 4]
Nombre d'iterations: 2

4.1.5 Bellman-Ford appliqué sur un ordre aléatoire

Pour pouvoir déduire si l'ordre de sommets a une importance sur le nombre d'itérations, nous allons générer un ordre aléatoire à l'aide de la fonction `random_order(self)` qui prend en entrée un graphe, dans ce cas le graphe de test H. Nous avons obtenu l'ordre suivant pour une taille de 5 sommets: [4, 3, 0, 1, 2]. Maintenant nous pouvons appliquer l'algorithme bellmanFord_gloutonFas sur l'ordre aléatoire et sur le graphe de test H, en partant également du sommet 1. Nous avons obtenu:

Bellman-Ford en partant de 1: 0: -15, 1: 0, 2: -8, 3: inf, 4: -10
Arbre des plus courts chemins: 0: [1, 4, 0], 1: [1], 2: [1, 4, 2], 3: [None, 3], 4: [1, 4]
Nombre d'iterations: 3

4.1.6 Comparaison de résultats par rapport au nombre d'itérations

Comme nous avons pu voir aux points 4.1.4 et 4.1.5, il est certain que l'ordre de sommets dans lequel nous exécutons l'algorithme Bellman-Ford est importante. Avec nos résultats nous pouvons donc confirmer ce qui a été dit au point 3.2.1. Mais nous allons continuer d'avancer dans nos tests, en incrémentant le nombre de sommets ainsi que les graphes pondérés qui servent pour l'apprentissage.

4.2 Efficacité en faisant varier la taille de l'instance ainsi que le nombre de graphes pondérés G_i

Pour pouvoir généraliser nos tests, nous avons codé une fonction qui s'appelle:

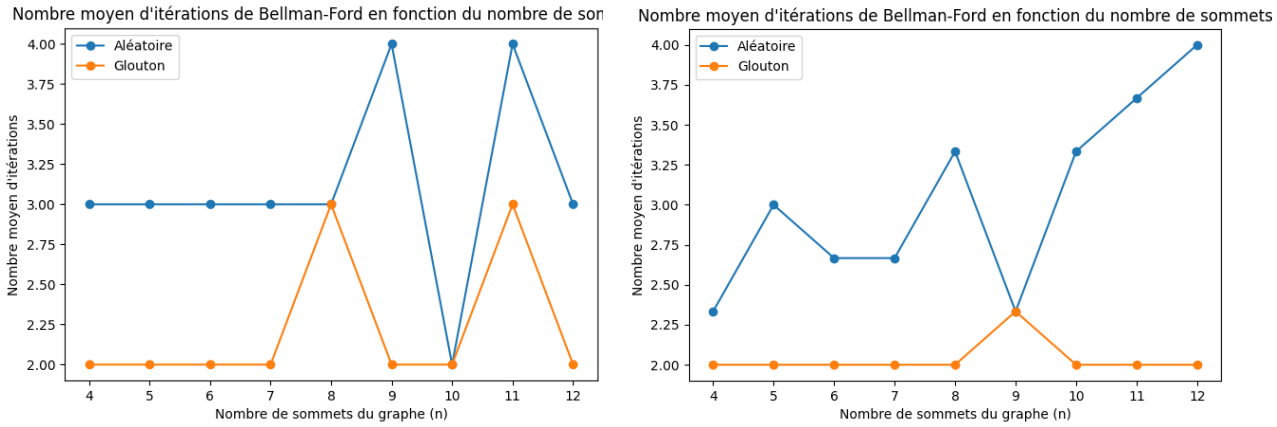
`analyze_vertex_iteration_nb(num_graphs_per_size, Nmax, p, nb_g, weight_interval)`

et qui nous servira pour avancer dans nos tests. Elle prend en entrée le nombre de graphes pour lesquels nous calculons la moyenne de nombre d'itérations pour le nombre de sommets donné Nmax, la probabilité p d'avoir chaque arc, le nombre des graphes pondérés nb_g, ainsi que l'intervalle de fonctions des poids [-w, w] donné par weight_interval. Cette fonction sert à tracer une figure avec en abscisse le nombre de sommets et en ordonné le nombre d'itérations.

Nous voulons préciser que nous avons choisi d'étudier le nombre d'itérations à partir de graphes de 4 sommets avec une probabilité relativement faible d'avoir des arcs. Cette décision vise à éviter la présence de cycles négatifs, car la vérification de tels cycles entraîne une perte de temps de calcul. De plus, nos tests ont été limités à un nombre d'environ 10 sommets. Cette limitation découle du fait que, avec les configurations d'entrée que nous avons sélectionnées, les vérifications nécessaires pour pouvoir appliquer notre fonction (génération de graphes aléatoires, test de cycle négatif, passage d'une arborescence à un graphe etc.) devient longue à partir de 10 sommets.

4.2.1 Efficacité en faisant varier la taille de l'instance

Dans cette section, nous allons faire varier la taille de l'instance, donc le nombre de sommets de nos graphes. Nous garderons 3 comme nombre de graphes pondérés G_i. Nous allons donc exécuter notre fonction.



(a) analyze_vertex_iteration_nb(1, 12, 0.3, 3, 10) (b) analyze_vertex_iteration_nb(3, 12, 0.3, 3, 10)

Nous voulons préciser que nous posons, dans la figure (a) num_graphs_per_size = 1, car dans ce test nous ne voulons pas faire une moyenne de nombre d'itérations par taille de l'instance. Sans faire de moyenne, nous pouvons donc observer dans la figure (a) que l'aléatoire peut avoir de la chance et il peut trouver plusieurs nombres d'itérations égaux à ceux de l'ordre $<_{tot}$ (pour le sommets 8 et 10), pourtant l'ordre $<_{tot}$ reste globalement meilleur. De même, nous pouvons observer que dans cette figure, le nombre d'itérations varie entre 2 et 3 pour un ensemble de 3 graphes d'apprentissage G_i. Nous allons également étudier cela dans la section 4.2.2.

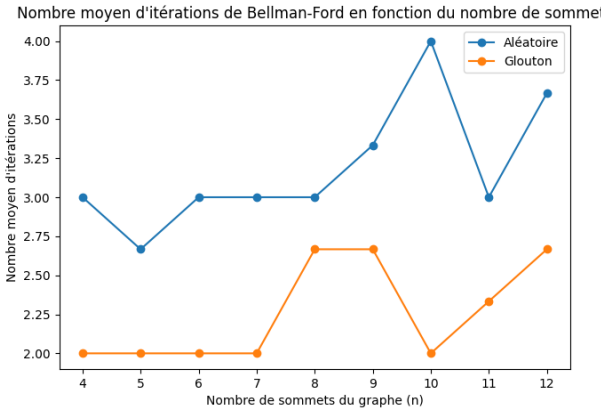
Dans la figure (b), nous avons choisi de calculer le nombre d'itérations moyen par taille de l'instance en posant num_graphs_per_size = 3. Comme nous pouvons voir dans la figure (b), l'ordre aléatoire

trouve ici un seul nombre d'itérations égal au nombre d'itérations trouvé par $<_{tot}$. Nous tenons à préciser que cette variation peut également être attribuée à la nature aléatoire de la génération des graphes, les rendant différents de ceux de la figure (a). En calculant des moyennes, la probabilité de trouver un nombre égal d'itérations dans les deux méthodes devient plus faible. Par ailleurs, en effectuant des moyennes, on observe que le nombre d'itérations semble plus stable, tendant vers 2 avec ce jeu de tests.

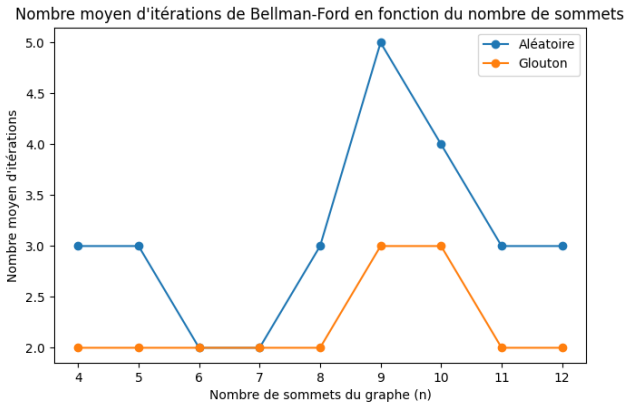
Nous pouvons donc de nouveau confirmer ce qui a été dit au point 3.2.1, donc que l'ordre de sommets influence le nombre d'itérations.

4.2.2 Influence de nombres des graphes pondérés G_i sur le nombre d'itérations

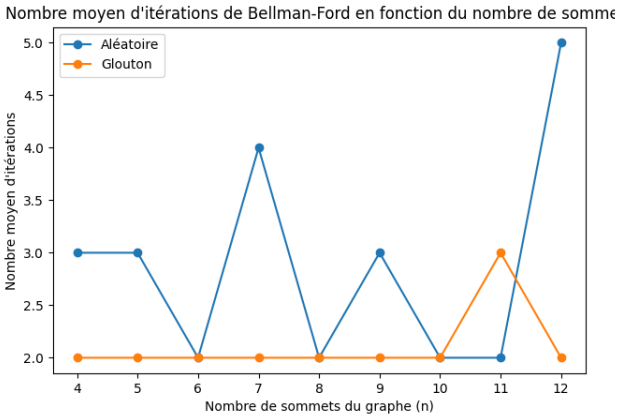
Dans cette section, nous allons faire varier le nombre de graphes pondérés utilisés pour l'apprentissage afin d'observer son impact sur le nombre d'itérations. Pour effectuer ce test, nous allons exécuter la même fonction sur plusieurs ensembles de graphes d'apprentissage G_i. Étant donné que nous générons des graphes de manière aléatoire, nous avons décidé de prendre une moyenne (sur 3 graphes différents) pour chaque taille d'instance afin d'obtenir des résultats les plus représentatifs de la réalité. De plus, dans la figure (a), nous avons effectué le test avec un seul graphe d'apprentissage, pour la figure (b) avec 3, pour la figure (c) avec 5, et pour la figure (d) avec 7.



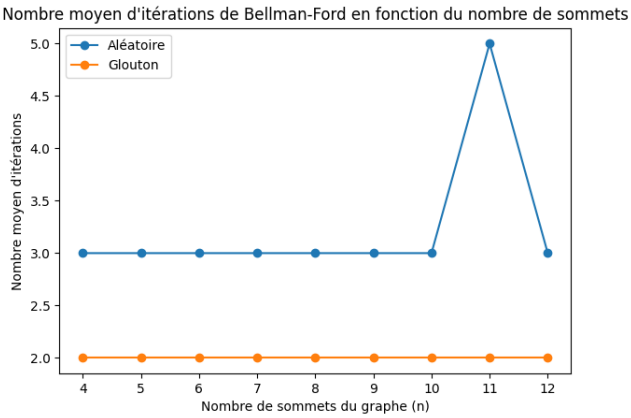
(a) analyze_vertex_iteration_nb(3, 12, 0.3, 1, 10)



(b) analyze_vertex_iteration_nb(3, 12, 0.3, 3, 10)



(c) analyze_vertex_iteration_nb(3, 12, 0.3, 5, 10)



(d) analyze_vertex_iteration_nb(3, 12, 0.3, 7, 10)

Tout d'abord, nous procéderons à une analyse de l'ordre choisi par l'algorithme glouton. Sur la figure (a), nous constatons une variation du nombre d'itérations pour un graphe d'apprentissage donné (les exécutions sur les sommets 8, 9, 12 ont, en moyenne, 2.75 itérations tandis que les autres n'en ont que 2. Cependant, à mesure que le nombre de graphes d'apprentissage augmente, nous observons que ces variations semblent diminuer. Sur la figure (b), cette variation diminue par rapport à la figure (a), passant de 3 sommets avec 3 itérations à 2 sommets avec 3 itérations. De (b) à (c), elle diminue de 2 à 1, et enfin, à (d), elle atteint une stabilité qui se trouve à 2 itérations. Cette observation suggère que le nombre de graphes de test influe sur le nombre d'itérations.

En ce qui concerne l'ordre aléatoire, une analyse similaire ne peut pas être effectuée, car les graphes d'apprentissage ne sont utilisés que pour améliorer l'ordre fourni par la fonction GloutonFas. Nous les avons représentés car nous avons utilisé la même fonction pour garantir une comparaison cohérente.

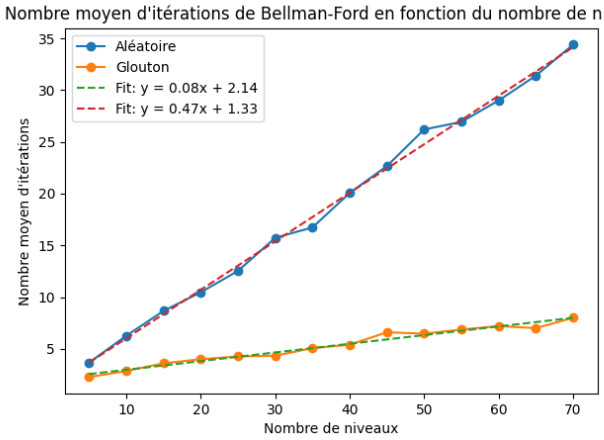
4.2.3 Efficacité sur une famille d'instances particulière

Pour cette section, nous allons évaluer l'intérêt de la méthode avec prétraitement en générant une famille d'instances selon le schéma suivant: un graphe par niveau avec 4 sommets pour chaque niveau et 2500 niveaux, où les sommets du niveau j précèdent tous les sommets du niveau j+1, et où pour chaque arc w_n, le poids est tiré de manière uniforme et aléatoire.

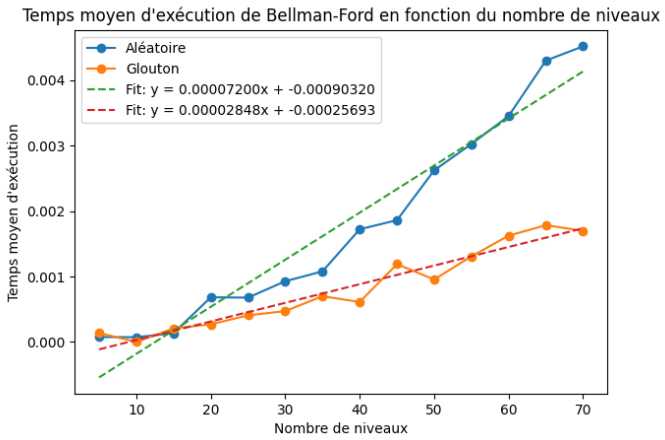
Tout d'abord, nous avons généré le graphe avec la fonction `create_graph_by_level`. Ensuite, à l'aide de la fonction `pretraitement_methode`, nous allons effectuer le même traitement que dans la partie 4.2.2, mais sans faire de moyenne car nous disposons d'un seul graphe.

Lors de l'exécution de la fonction `pretraitement_methode(self, nb_g, weight_interval)` avec `weight_interval` $\in [-10, 10]$ et avec 3 graphes d'apprentissage donc nb_g = 3, nous avons ajouté plusieurs instructions `print` pour surveiller son état, car nous nous attendions à un temps d'exécution prolongé. Nous avons constaté que, avec la manière dont nous avons implémenté notre code, la fonction `from_tree_to_graph`, qui nous permet de transformer l'union d'arborescences des plus courts chemins fournie par la fonction `union_path`, prend beaucoup de temps d'exécution pour cette famille d'instances. Donc, elle n'arrive pas à transformer l'union d'arborescences dans un graphe en temps raisonnable. Cette situation semble logique étant donné que notre fonction contient trois boucles `for` imbriquées qui implique une complexité en O(n*m*l) avec n = nombre de sommets dans le graphe, m = nombre de chemins associés à chaque sommet (au pire cas) et l = longueur maximale d'un chemin. Le passage à un graphe est trivial, car la fonction `GloutonFas` prend en entrée un graphe.

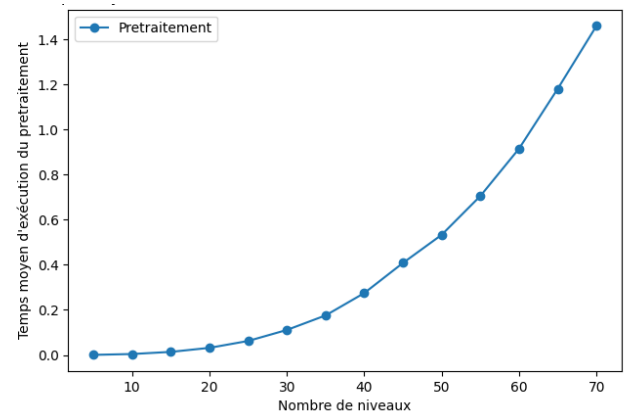
Pour réduire le temps d'exécution, nous avons opté pour des tests de la méthode de prétraitement sur des instances avec moins de niveaux. Ensuite, nous avons tenté d'estimer le nombre d'itérations pour nos 2500 niveaux. Pour ce faire, nous avons implémenté une nouvelle fonction :
`pretraitement_methode_graph`. Cette fonction a principalement pour objectif de tracer une figure où l'axe des abscisses représente le nombre d'itérations et l'axe des ordonnées représente le nombre de niveaux. De plus, elle permet également de trouver l'équation de la droite tracée (pour estimer le nombre d'itérations pour 2500 niveaux) et de tracer le temps d'exécution de la méthode de prétraitement. Nous avons choisi d'exécuter notre fonction avec 3 graphes d'entrainement, une fonction de poids entre [-10, 10], 70 niveaux, 4 sommets par niveau et 15 graphes crées par niveau (qui nous servent pour le calcul de moyennes).



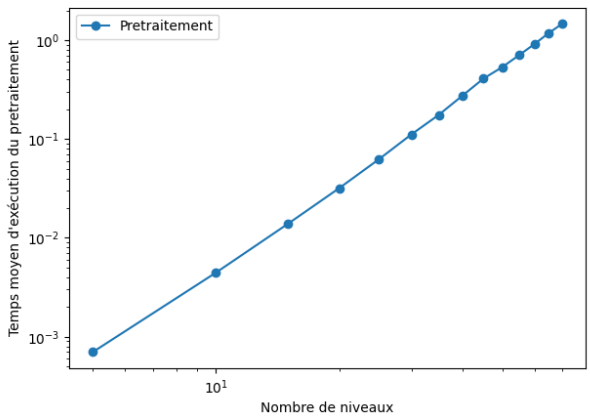
(a) pretraitement_method_graph(3, 10, 70, 4, 15)



(b) pretraitement_method_graph(3, 10, 70, 4, 15)



(c) pretraitement_method_graph(3, 10, 70, 4, 15)



(d) pretraitement_method_graph(3, 10, 70, 4, 15)

Dans la figure (a), nous pouvons observer que l'ordre donné par l'algorithme glouton a une pente plus faible que celui donné par l'ordre aléatoire. Cela suggère que le prétraitement fonctionne également dans ce cas. Nous avons également calculé l'équation de la droite pour les deux ordres, mais nous nous concentrerons sur celle de l'ordre donné par l'algorithme GloutonFas. Pour 70 niveaux, l'équation est y = 0.08x + 2.14. Par conséquent, nous prévoyons que, pour 2500 niveaux, nous aurons y = 0.08*2500 + 2.14 ≈ 202 itérations.

Dans la figure (b), nous avons représenté le temps moyen d'exécution de l'algorithme Bellman-Ford. Comme nous pouvons le constater, il s'exécute rapidement, et ce n'est pas lui qui contribue à l'augmentation du temps d'exécution.

Dans la figure (c), nous avons tracé le temps moyen d'exécution de la méthode de prétraitement en fonction du nombre de niveaux. Dans ce temps d'exécution, nous avons inclus trois fonctions, à savoir `union_path, from_tree_to_graph, glouton_fas`, car ces trois fonctions sont utilisées dans le prétraitement. Comme nous pouvons le constater, cette méthode n'augmente pas de manière linéaire. Nous allons donc étudier la nature de cette fonction.

La figure (d) vise à déterminer si la fonction est polynomiale. Pour ce faire, nous avons tracé les données de la figure (c) en échelle logarithmique, et comme on peut le constater, il s'agit bien d'une droite. Cela signifie que la fonction est polynomiale. Pour déterminer le degré du polynôme, nous avons calculé sa pente et trouvé 2.94, ce qui suggère un polynôme de degré approximatif 3.

Cette conclusion semble logique étant donné la complexité de la fonction `from_tree_to_graph`, qui contient trois boucles for. Ainsi, le prétraitement est la partie qui occupe la majeure partie du temps d'exécution.

Cependant, nous avons observé lors des exécutions que dans la plupart des cas pour cette instance spécifique, l'ordre des sommets renvoyé par GloutonFas suit une séquence particulière : les quatre premiers sommets correspondent au premier niveau, les quatre suivants au deuxième niveau, et ainsi de suite. Pour réaliser cela, nous aurions pu simplement construire une liste avec les sommets du graphe dans l'ordre de leur apparition, ce qui aurait été possible en O(n).

Ainsi, nous pouvons conclure que la méthode de prétraitement n'est pas adaptée à toutes les familles d'instances, en particulier pour les grandes instances, un cas qui s'est révélé être une contrainte en termes de temps d'exécution.

En addition, pour éviter le temps que la fonction `from_tree_to_graph` prend, nous avons réfléchi à une autre solution. Au lieu d'utiliser un graphe en entrée pour la fonction `GloutonFas`, nous avons envisagé de passer l'union des arborescences des plus courts chemins comme entrée. En faisant cela, il serait également possible de calculer les sources et le puits dans ce graphe, car cela est nécessaire dans la fonction `GloutonFas`. Cependant, si nous modifions l'entrée de la fonction `GloutonFas`, il faut également ajuster les fonctions `sources` et `sinks`, car les deux prennent en entrée un graphe. Nous devrions donc fournir l'union des arborescences des plus courts chemins comme entrée à la place. Avec cette approche, nous évitons d'utiliser la fonction `from_tree_to_graph`, mais nous augmentons également la complexité des autres fonctions telles que `sinks` et `sources`, car au lieu de prendre un graphe en entrée, elles prendront l'union des arborescences des plus courts chemins, représentée comme un dictionnaire de listes de listes, ce qui pourrait également devenir complexe pour la famille d'instances choisie. Cependant, en effectuant cette implémentation, une diminution du temps de calcul pour la fonction `GloutonFas` pourrait être réalisée.

De plus, nous avons essayé de trouver un autre moyen de trouver un bon ordre. L'idée aurait été de faire un parcours en hauteur du graphe spécifique à notre famille d'instances, ce qui aurait considérablement réduit le temps d'exécution, car un parcours en hauteur s'effectue en O(m+n). Cependant, avec un parcours en hauteur, notre ordre de sommets ne constituerait pas une permutation complète de tous les sommets. Nous aurions uniquement les sommets renvoyés par le parcours, mais pas nécessairement l'ensemble complet de sommets.

5 HowTo

5.1 Structure

Notre code est composé de deux fichiers, *Graph.py* qui comporte l'ensemble de nos algorithmes utiles au projet et *BellmanFord-Optimisation.py* qui nous sert a exécuter les différentes questions. Le fichier *BellmanFord-Optimisation.py* est divisé en différentes fonctions, chacune correspondant à une question spécifique.

5.2 Exécution

Pour exécuter une question spécifique, passez le numéro de la question comme argument en ligne de commande. Par exemple, pour exécuter la question 1, utilisez la commande suivante :

```
python3 BellmanFord-Optimisation.py 1
```

5.3 Dépendances

Assurez-vous que les dépendances telles que `numpy` et `matplotlib` sont correctement importées.