Groupe 4
Master Informatique

Projet de MOGPL
Amélioration de l'algorithm de Bellman-Ford

Étudiants
Halimi Abdelkrim
Basci Onur

December 9, 2023

Contents

I Problématique

Soit $G = (V, E)$ un graphe orienté pondéré sans circuits de poids négatif. L'algorithme de Bellman-Ford calcule les plus courts chemins de s à chaque $v \in V$.
Si G n'a pas de circuits de poids négatif, l'algorithme converge au plus $n - 1$ itérations.

L'étude cherche à affiner cette estimation en considérant un ordre spécifique des sommets $<_{\text{tot}}$. On explore comment cet ordre impacte le nombre d'itérations, en introduisant $dist(<_{\text{tot}}, P)$ pour mesurer l'inversion d'arcs dans un chemin P.

II Proposition

Pour l'algorithme de Bellman-Ford sur le graphe G, le temps d'exécution avec l'ordre $<_{\text{tot}}$ est $O(m \max_{v \in V} \text{dist}(<_{\text{tot}}, P_v))$, soulignant l'importance d'un bon ordre des sommets.

En fixant $G = (V, E)$, avec des poids d'arcs $w_1, w_2, \ldots, w_k$, les chemins les plus courts P_i^v pour chaque i sont définis. La question clé est de savoir si la similarité entre G_i et H impacte l'amélioration des itérations de Bellman-Ford.

Le choix de l'ordre pour Bellman-Ford implique la résolution du problème MVP (Minimum Violation Permutation). Il vise à trouver un ordre $<_{\text{tot}}$ sur V minimisant $\max_{i \in [r]} \text{dist}(<_{\text{tot}}, P_i)$.

La résolution de MVP utilise une méthode gloutonne, GloutonFas, construisant un ordre linéaire en déplaçant les sommets avec un faible degré entrant au début et ceux avec un faible degré sortant à la fin.

III Analyse de l'Amélioration Potentielle des Itérations de Bellman-Ford en Fonction de la Similarité entre Graphes et Résolution du Problème MVP

1) Mise en place de la structure du Graphe et des fonctions essentielles pour l'analyse approfondie

Tout d'abord, nous avons décidé de représenter notre graphe avec une approche orientée objet, où le graphe est représenté par une liste d'adjacences. Nous lui avons donc associé les différentes méthodes demandées lors de ce projet.

a) **Classe Graph**
Cette classe représente la structure de nombre graphe orienté.

```
Classe Graph:
    Procédure Initialiser_Graph(liste_sommets):
        self.liste_sommets <- liste_sommets
        self.liste_arcs <- []
        self.ordre_sommets <- None
        self.graph <- {x: [] pour x dans self.liste_sommets}
        self.distances <- tableau_rempli(len(self.liste_sommets), inf)
        self.chemins <- None
        self.nb_iterations <- 0
    Fin Procédure
```

b) **Bellamn-Ford**
Ainsi, l'algorithme de Bellman-Ford a été implémenté en tant que méthode de cette

classe. Nous avons décidé de construire simultanément les arborescences des plus courts chemins depuis la source et d'arrêter l'algorithme dès que nous détectons une convergence.

Algorithm 1 Algorithme de Bellman Ford

Entrée : Vertex Source, Ordre
Sortie : chemins, distances, nombre itérations

```
1: nombre iteration ← 0
2: distances ← [0, inf, inf ... inf]
3: chemins ← List de lists
4: for i allant de 0 à |V| do
5:    MiseàJour ← False
6:    for sommet dans Ordre do
7:       for voisin, poid dans voisins(sommet) do
8:          if distances[sommet] ≠ inf et distances[sommet] + poid < distances[voisin] then
9:             distances[voisin] ← distance[sommet] + poids
10:            chemins[voisin] ← chemins[sommet] + [voisin]
11:            Miseàjour ← True
12:         end if
13:      end for
14:   end for
15:   incrémenter nombre iteration
16:   if not(MiseàJour) then
         On sort de la boucle
17:   end if
18: end for
```

2) Implémentation de l'Algorithme GloutonFas pour la Résolution du Problème MVP

L'implémentation de l'algorithme GloutonFas est fournie dans le présent rapport. Toutefois, il est important de noter que nous l'avons également intégrée en tant que méthode de notre classe "Graph".

a) Obtention efficace des sources

Considérons un graphe $G = (V, E)$ où nous avons décidé d'inverser le graphe afin d'obtenir les sources. L'inversion du graphe s'effectue en $O(E)$, ce qui revêt une importance cruciale pour les questions suivantes. L'algorithme d'inversion du graphe est décrit ci-dessous.

Algorithm 2 Algorithme du Graphe Inverse

Entrée : Graphe
Sortie : Graphe inversé

```
Graphe inverse ← nouveau Graph
for (Sommet, voisin, poid) dans Graph do
    Ajouter (voisin, sommet, poid) dans Graphe Inverse
end for
```

3) Génération de Graphes d'Entrainement et de Test

La division entre les ensembles d'entraînement et de test permet d'évaluer les performances de l'algorithme GloutonFas de manière indépendante. Cela donne une idée de son efficacité à généraliser sur de nouveaux graphes.

Nous avons décidé d'implémenter les méthodes ci-dessous en tant que méthodes statiques, ce qui nous permettra de les utiliser sur d'autres graphes.

a) **Génération d'un graphe aléatoire**
Pour créer un graphe avec un sommet à partir duquel on peut atteindre au moins $|V|//2$ sommets, nous fournissons à notre fonction le nombre de sommets ainsi que le nombre d'arêtes à générer, de manière à ce que $|E|$ soit nettement supérieur à $|V|$. Ensuite, pour vérifier que la source peut accéder à au moins $|V|//2$ sommets, nous exécutons BFS à partir du sommet source et comptons le nombre de sommets visités. Si ce dernier est supérieur à (—V—//2), nous acceptons le graphe.

Algorithm 3 Générer Graphe Aléatoire

Entrée: Nombre Sommet, Nombre arrêtes
Sortie: Graph aléatoire

```
pas_assez_sommets ← True
while pas_assez_sommets do
    Graphe_aléatoire ← nouveau Graph
    for i allant de 0 à nombre arrêtes do
        (u, v) ← deux sommets différents aléatoirement choisis entre [0, NombreSommets]
        ajouter (u, v, 1) dans Graphe_aléatoire
    end for
    Graphe_aléatoire ← Générer_poids_aléatoires(Graphe aléatoire)
    n ← Nombre sommet accesible dans Graphe aléatoire, de la source
    if n > longueur(NombreSommet)//2 then
        pas_assez_sommets ← False
    end if
end while
```

b) **Elimination des cycles négatifs**
Durant la génération de poids aléatoires, il était envisageable d'obtenir des cycles négatifs. Afin de remédier à cette situation, nous avons choisi de les détecter et de les éliminer à chaque génération. Pour cela, nous mettons les poids des arêtes composant le cycle à des valeurs positives. La méthode que nous décrivons ici pour repérer ces cycles négatifs

nous permet de créer rapidement des graphes de grande envergure sans présence de cycles négatifs, facilitant ainsi la réalisation de tests plus efficaces.

Algorithm 4 Trouver cycle Negatif

Entrée: chemin (une liste)
Sortie: cycle négatif

```
deja_vu ← dictionnaire
index_cycle ← -1
for i allant de 0 à longueur de chemin do
    if chemin[i] dans déjà_vu then
        index_cyle ← i
        Sortir de Pour
    end if
    déja_vu[valeur] ← i
    cycle ← sous séquence de chemin entre i_debut_cylce et index_cyle
    i_debut_cycle ← déja_vu[chemin[index_cycle]]
end for
```

c) **Exemple de graphes obtenus**
Voici un exemple de graphe d'entraînement, illustré dans la Figure 2, généré à partir du graphe de base de la Figure 1.

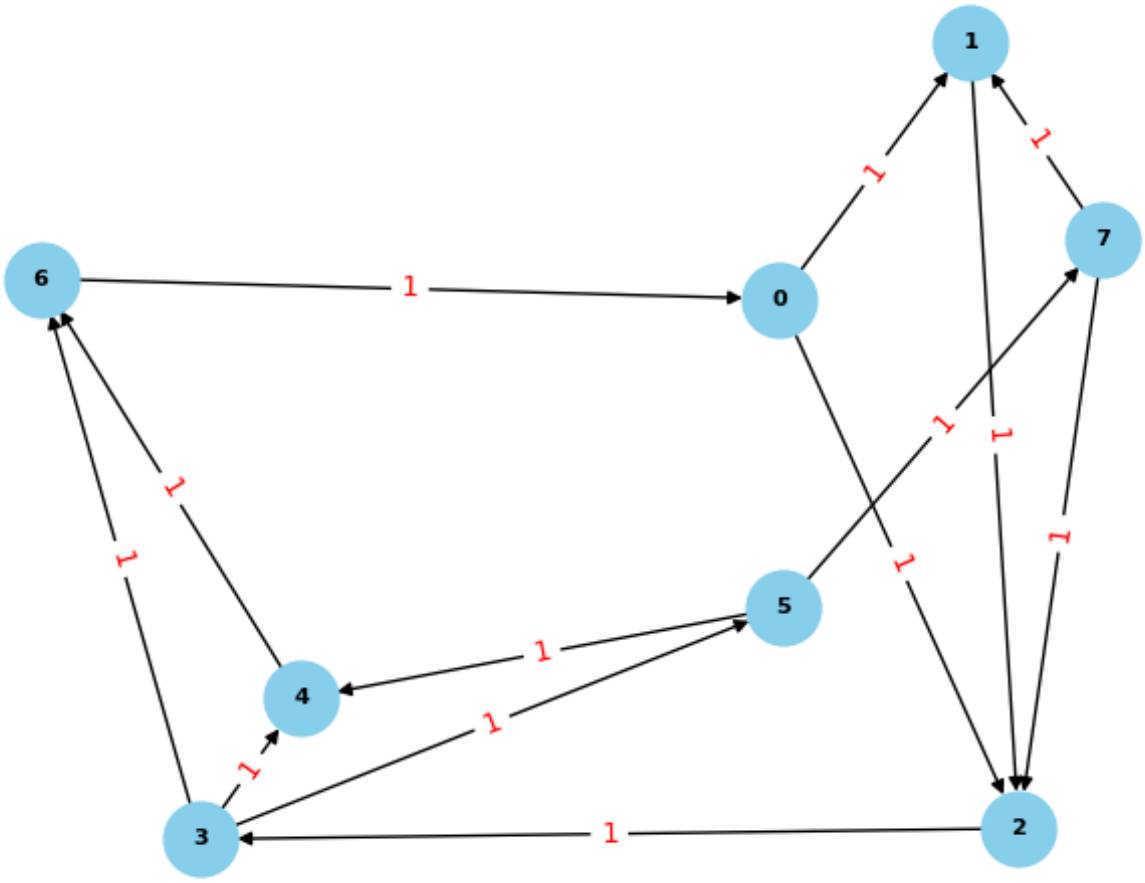

Figure 1: Graphe de base

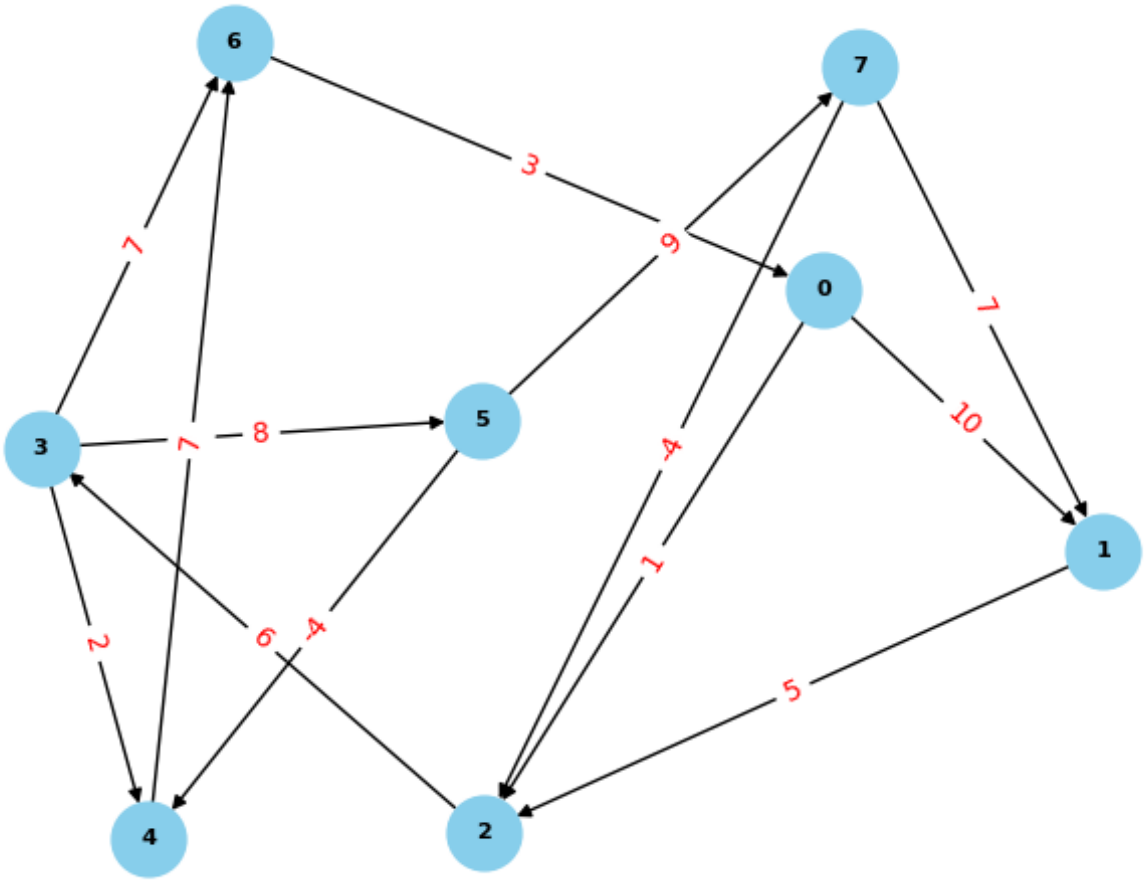

Figure 2: Graphe d'entraînement

4) Extraction d'un Ordre optimale à partir des Graphes d'Entrainements

L'objectif des graphes d'entraînement est d'utiliser l'union de leurs arborescences des plus courts chemins afin d'expérimenter avec GloutonFas l'extraction d'un ordre à utiliser comme heuristique pour l'algorithme de Bellman-Ford. Cet ordre pourra ensuite être utilisé dans d'autres graphes de même taille afin d'aboutir à une amélioration.

Il est important de noter que l'union de leurs arborescences déterminera un nouveau graphe. La fonction facilitant l'unification des arborescences est décrite ci-dessous.

a) Algorithme qui permet l'unification des arborescences

Algorithm 5 Unifier Chemins

Entréé : Chemins, List_sommets
Sortie : Graph Unifie

```
list_aretes ← liste vide
aretes_ajoutees ← Dictionnaire vide
for tout_les_chemins dans chemins do
    for chemin dans tout_les_chemins do
        ajouter_arete_depuis_chemin(chemin)
    end for
end for
Graphe_Unifie ← Nouveau Graph avec les composé de List_Sommets
ajouter les arêtes de list_arêtes dans Graph_Unifie
```

b) **Déterminer un ordre optimale**
L'application de GloutonFas sur l'union de l'arborescences des plus courts chemins des graphes d'entrainement donne le résultat suivant :

```
[INFO] Ordre obtenu : [0, 2, 3, 4, 5, 6, 7, 1]
```

5) Comparaison des Performances de Bellman-Ford en utilisant des Ordres générés avec glouton FAS, et Aléatoires dans les Graphes Sans Circuits de Poids Négatif

Après l'entraînement, nous obtenons un ordre que nous passerons en paramètre à l'algorithme de Bellman-Ford appliqué à notre graphe de test. Nous allons également effectuer un test avec un ordre aléatoire afin d'évaluer la pertinence de l'ordre obtenu à partir des graphes d'entraînement.

Génération d'un ordre aléatoire

Algorithm 6 Générer Ordre Aléatoire

Entréé : Liste_Sommets
Sortie : Ordre_aléatoire

```
n ← longuer de la Liste_Sommets
Ordre_aléatoire ← Tirage de n element sans remise
Return Ordre_aléatoire
```

Résultat des deux exécutions

Il est observable que l'ordre déterminé à partir des graphes d'entraînement produit des résultats nettement supérieurs à un ordre aléatoire simple, avec seulement 2 itérations comparées à 5. Ainsi, le temps investi dans l'entraînement de nos graphes est récupéré lors de l'exécution sur de nouveaux graphes

```
Comparaison Bellman avec prétraitement et Bellman avec un ordre aléatoire
Nb itération, Bellman avec prétraitement: 2
```

```
Nb itération, Bellman avec un ordre aléatoire: 5
[INFO] Question 8 terminée
```

6) Test de notre approche sur différents Graphes

Nous avons exécuté l'algorithme de Bellman-Ford sur plus de 100 graphes aléatoires, en utilisant un ordre aléatoire ainsi qu'un ordre estimé avec GloutonFas. L'objectif est de déterminer s'il existe des cas où l'ordre fourni par Glouton est moins efficace qu'un ordre aléatoire.

Paramètres utilisés

- 100 sommets
- 200 arrêtes
- 10 graphes d'entrainements

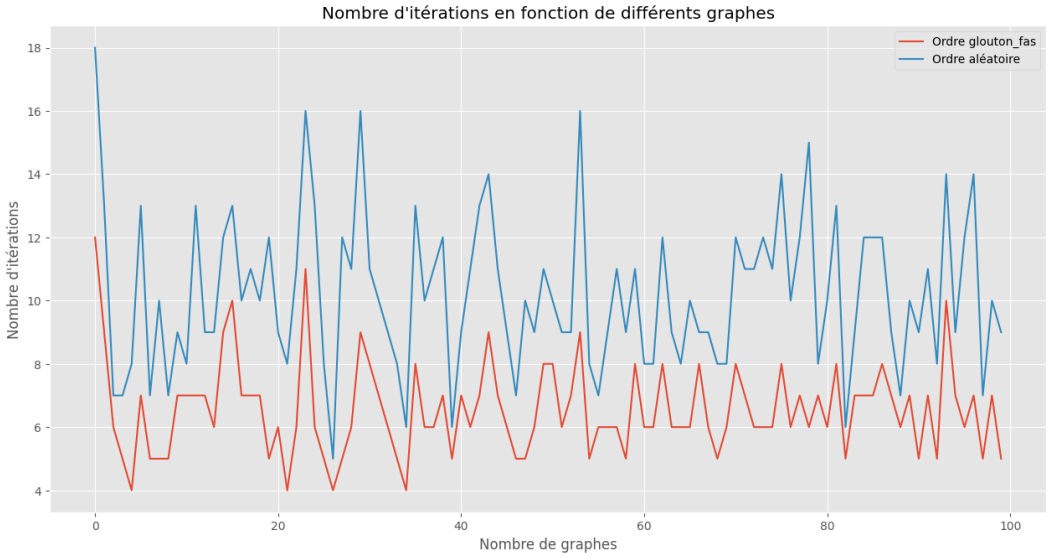

Figure 3: Comparaison des 2 approches sur plus de 100 grahes

Résultats : Comme il est observé sur la figure ci-dessus, l'ordre aléatoire n'a jamais surpassé l'ordre optimal, au mieux, il a produit des résultats ratiquement équivalents. On remarque également de nombreuses occurrences où l'ordre optimal a conduit à une amélioration significative du nombre d'itérations, avec une moyenne presque deux fois plus élevée. Il serait intéressant d'explorer si le nombre de graphes d'entraînement a une incidence sur les résultats de l'ordre optimal obtenu avec l'algorithme $glouton\_fas()$

7) L'influence du nombre de Graphes d'Entrainements sur le nombre d'Itérations.

Nous avons réalisé des tests pour évaluer les performances de l'ordre obtenu à partir de l'algorithme GloutonFas en fonction du nombre de graphes d'entraînement. Pour cela, nous

avons calculé la moyenne du nombre d'itérations obtenu pour différentes valeurs du nombre de graphes d'entraînement. Les paramètres utilisés dans les tests sont définis comme suit :

- NG = Nombre de graphes d'entraînement.
- NS = Nombre de sommets dans le graphe.
- NA = Nombre d'arêtes dans le graphe.
- NR = Nombre de répétitions pour calculer la moyenne.

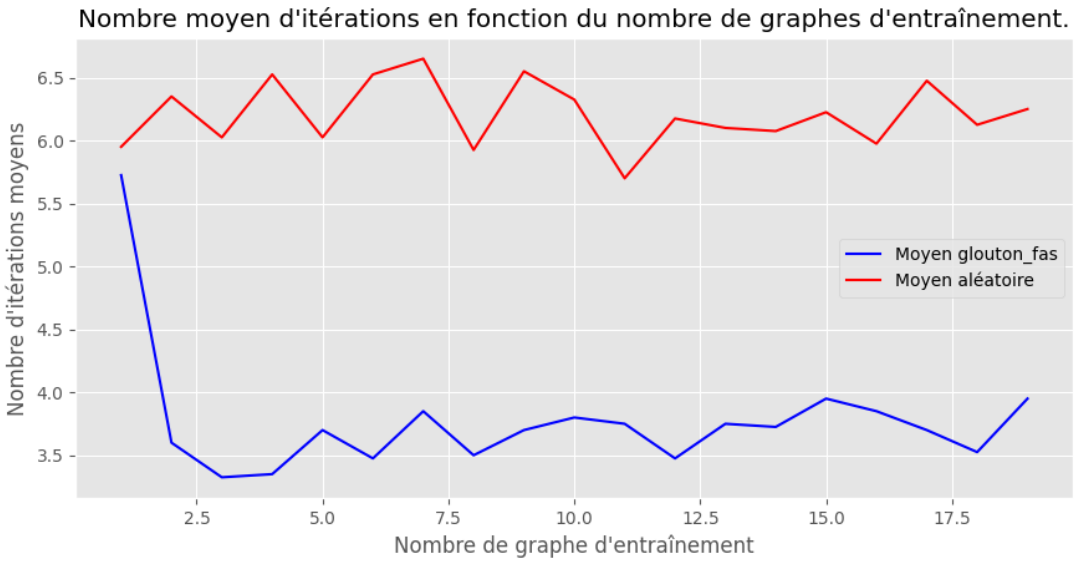

Figure 4: Comparaison des moyennes du nombre d'itérations en fonction du nombre de graphes d'entraînement. $NS = 30$, $NA = 75$, $NR = 40$

Résultats : On observe sur la figure ci-dessus que pour $NG = 1$, les moyennes du nombre d'itérations restent similaires. Cependant, une nette diminution est constatée dès que NG est augmenté à 2. Il semble que l'augmentation de NG au-delà de 3 n'ait aucune incidence significative sur les résultats obtenus. Ainsi, $NG = 3$ apparaît comme un paramètre optimal, nécessitant moins de puissance computationnelle tout en fournissant des résultats similaires à ceux obtenus avec des valeurs plus élevées de NG. C'est pourquoi dans les prochaines tests, nous avons utilisé $NG = 3$ comme paramètre.

Quelques tests supplémentaires

Nous avons également effectué des tests supplémentaires en fonction du nombre de sommets et du nombre d'arêtes.

a) **Tests en fonctions de nombres de sommets**

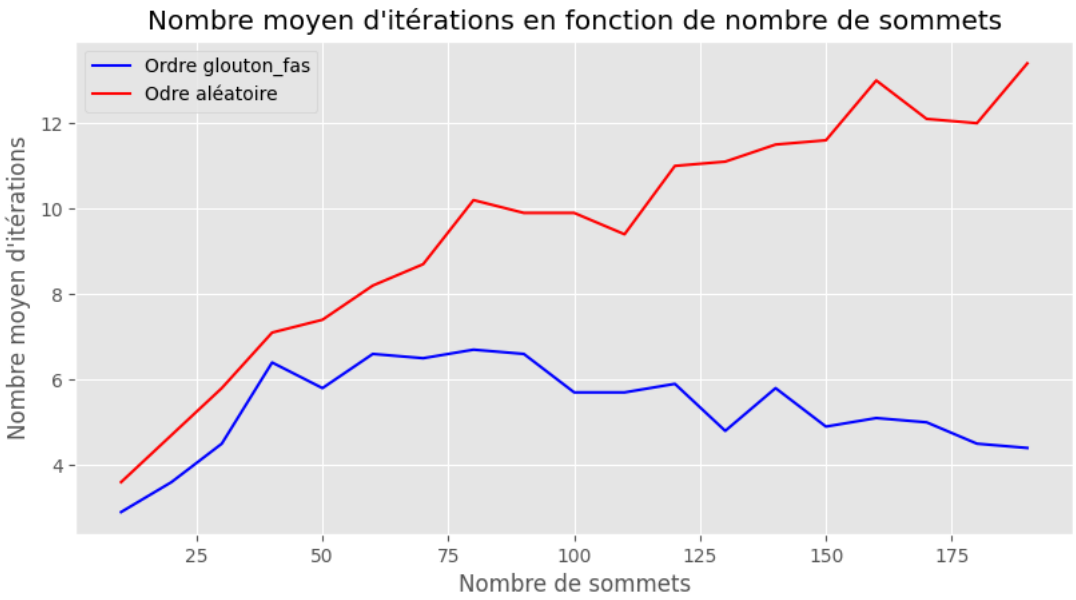

Figure 5: Comparaison de moyennes de nombre d'itérations en fonction de nombre de Sommets. $NA = 200$, $NG = 3$, $NR = 10$

Résultats : Il semble que pour l'ordre extrait avec GloutonFas, le nombre d'itérations augmente initialement avant de diminuer en fonction du nombre de sommets. Cette diminution du nombre d'itérations s'explique par la réduction du nombre de sommets accessibles à partir des points de départ, ce qui limite le nombre de chemins alternatifs, en effet on test l'algorithme sur des graphes avec 200 arêtes au maximum. Par conséquent, un nombre moindre d'itérations est nécessaire pour appliquer l'algorithme de Bellman. Cependant avec un ordre aléatoire, on constate que le nombre d'itération continue de grimper pour l'instant.

b) **Tests en fonction de nombre d'arêtes**

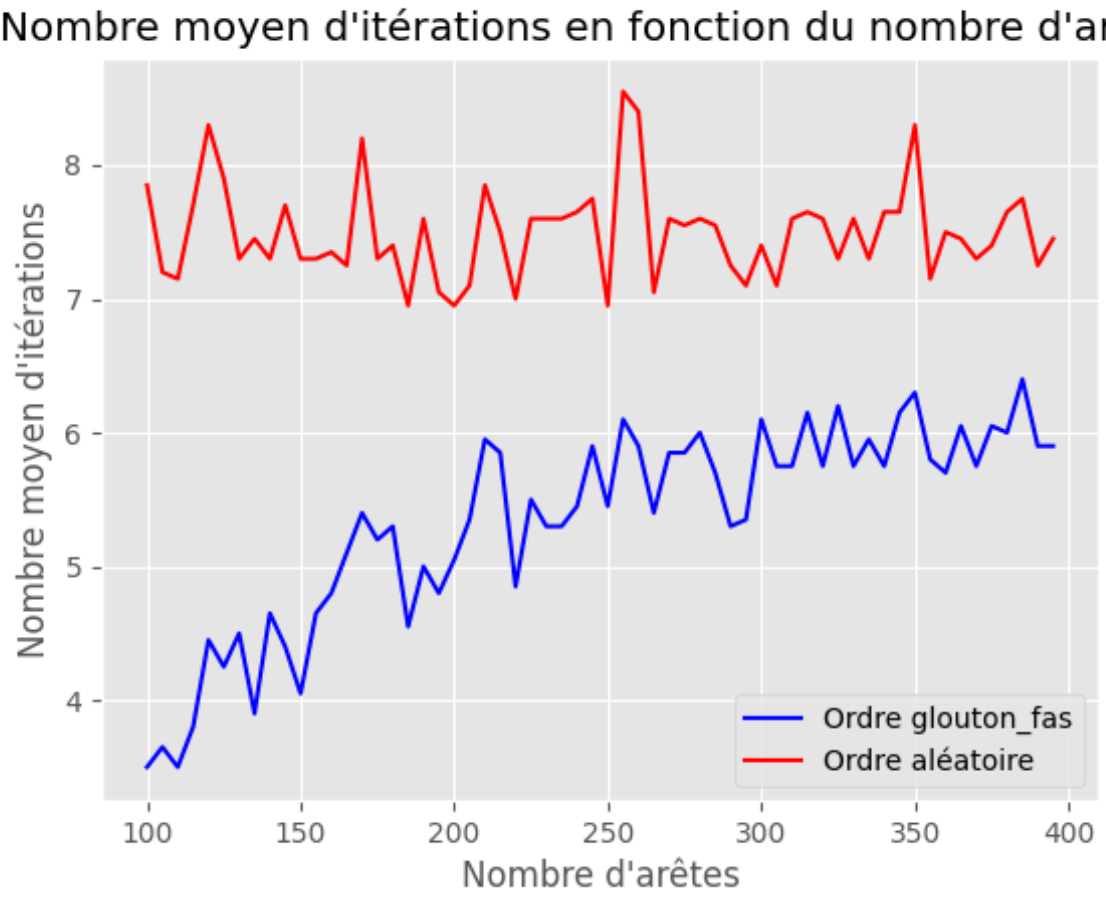

Figure 6: Comparaison des moyennes du nombre d'itérations en fonction du nombre d'arêtes. $NS = 50$, $NG = 3$, $NR = 20$

Résultats On remarque que l'ordre aléatoire fournit un résultat constant, d'environ 8 itérations tout au long du test. Cependant, avec l'ordre de GloutonFAS, on met en évidence une augmentation du nombre d'itérations en corrélation avec le nombre d'arêtes du graphe. Il semble que cette croissance atteigne un plateau au-delà d'un seuil spécifique, observé aux alentours de $NA = 250$ dans la figure 6 ci-dessus.

8) Évaluation de l'Adéquation de la Méthode avec Prétraitement sur des cas Particulier

Considérons un graphe G organisé par niveaux, avec quatre sommets par niveau et 2500 niveaux. Dans ce graphe, les sommets du niveau j précèdent tous les sommets du niveau

$j+1$, et chaque arc est associé à un poids tiré de manière uniforme et aléatoire dans l'intervalle [-10, 10].

Pour ce type de graphe, l'algorithme optimal se révèle très efficace. Pour le démontrer, examinons d'abord ce qu'est l'ordre optimal dans ce graphe, puis comparons-le avec l'ordre obtenu grâce à l'algorithme glouton.

Supposons l'application de l'algorithme de Bellman avec une source S de niveau k. L'algorithme de Bellman repose sur la formule $dv = \min_{y:(y,v)\in E}(dy + w(y,v))$. Cependant, dans le cas de G, un sommet v de niveau j a tous ses prédécesseurs dans le niveau $j-1$. Ainsi, pour déterminer la distance minimale de v dans le niveau $j > k$, il suffit de connaître la distance minimale des sommets du niveau $j-1$. Pour cela il faut que les sommets de niveau $j-1$ apparaissent avant que les sommets de niveau j dans l'ordre. Cette relation se maintient pour les sommets des niveaux $j-1, j-2, \ldots, 2$. On a donc pour tout sommet dans le niveau j, les sommets de niveau $j-1$ apparaissent avant dans l'ordre optimal.

En ce qui concerne les sommets de niveau inférieur ou égal à k, leur distance minimale est infinie car ils ne sont pas accessibles, à l'exception de S dont la distance est initialisée à 0. Par conséquent, l'ordre de ces sommets n'a pas d'influence sur le nombre d'itération, car à chaque itération, leurs valeurs restent constantes à l'infini.

On en déduit alors que l'ordre optimal, minimisant le nombre d'itérations, est de type croissant (dans le sens du niveau, où l'ordre entre les sommets du même niveau n'influence pas le nombre de répétitions) pour les sommets de niveau supérieur à k. Comme toutes les distances minimales des sommets de niveau $j-1$ sont calculées avant celles de niveau j avec cet ordre, l'algorithme détermine toutes les distances minimales en une seule itération. L'algorithme se termine alors en 2 itérations.

Démontrons à présent que l'algorithme glouton FAS nous fournit cet ordre optimal. L'algorithme glouton FAS débute en éliminant les sources dans G. Initialement, les seules sources dans G sont les sommets de niveau 1, car tous les sommets dans un niveau supérieur à 1 ont leurs prédécesseurs dans leur niveau précédent. Ainsi, il n'y a pas de source dans chaque niveau tant que le niveau précédent n'est pas supprimé. L'algorithme glouton FAS commence donc par supprimer les sommets de niveau 1, puis de niveau 2 jusqu'au niveau 2500. Pour chaque sommet supprimé, il le place à la fin de la liste. En fin de compte, cela crée un ordre croissant.

Remarque : Dans l'implémentation, en ayant connaissance de la structure du graphe, il serait possible de définir manuellement un ordre optimal croissant, afin d'éviter l'application de l'algorithme Glouton Fas, susceptible de réduire la complexité de l'algorithme.

Approche pratique
Nous avons également évalué ce scénario dans notre programme. Voici le code qui crée le graphe G

Fonction utilisé pour construire un graphe de 2500 niveau

```
Algorithme Generer_Graph_Niveaux(nb_niveaux):
    graph_niveaux <- Nouveau Graph([i pour i dans range(nb_niveaux * 4)])
    aretes <- Liste()
    Pour i de 0 à nb_niveaux-2:
        sommets_niveau_suivant <- [(i*4)+4+k pour k de 0 à 3]
        Pour j de 0 à 3:
            Pour k dans sommets_niveau_suivant:
                Ajouter (((i*4)+j), k, 1) à aretes
```

```
9
10      graph_niveaux.ajouter_aretes(aretes)
11      graph_niveaux <- Generer_Poids_Aleatoires(graph_niveaux)
12
13      Retourner graph_niveaux
14  Fin Algorithme
```

En appliquant l'algorithme de Bellman-Ford avec un ordre extrait de GloutonFas sur un graphe de 2500 niveaux, nous obtenons effectivement 2 itérations. En revanche, l'application de l'algorithme de Bellman-Ford avec un ordre aléatoire conduit à un nombre beaucoup plus élevé d'itérations, en l'occurrence 1216 ici. Voici un exemple de résultat obtenu.

Résultat de l'exécution

```
moyenne glouton 2.0
moyenne random 1216.0
```

IV Annexes

Fonction d'analyses en Python

- **Nombres d'itérations par graphes** : La fonction calculant le nombre moyen d'itération de Bellman-Ford, avec un ordre optimale, et aléatoire en fonction de différents graphes

```python
def anlayse_nb_iter_by_random_graph(nb_graphs=10, nb_vertex=100, nb_edges=400):
    nb_edges = nb_vertex*2
    liste_nb_iter_glouton = []
    liste_nb_iter_random = []
    for _ in range(nb_graphs):
        nb_iter_glouton,nb_iter_random =
        ↪  Graph.generate_compare_graph(nb_vertex,nb_edges,nb_graph_to_generate=10)
        liste_nb_iter_glouton.append(nb_iter_glouton)
        liste_nb_iter_random.append(nb_iter_random)

    mean_glouton = sum(liste_nb_iter_glouton)/len(liste_nb_iter_glouton)
    mean_random = sum(liste_nb_iter_random)/len(liste_nb_iter_random)

    return mean_glouton, mean_random
```

- **Nombres d'itérations par graphes d'entrainements** : Fonction d'analyse de la performance en fonction du nombre de graphes d'entraînement.

```python
def anlayse_nb_iter_by_nb_trainGraph(nb_vertex, nb_edges, nb_repetitions, max_nb_trainGraph):
    moyens_glouton = []
    moyens_random = []

    for i in range(1, max_nb_trainGraph):
        print(f"iteration {i}")
        moyen_glouton, moyen_random, _, _ = calclulate_mean_nb_iter(i, nb_vertex, nb_edges,
        ↪  nb_repetitions)
        moyens_glouton.append(moyen_glouton)
        moyens_random.append(moyen_random)

    nb_train_graph = [i for i in range(1, max_nb_trainGraph)]
```

- **Nombres d'itérations par sommets** : Fonction d'analyse en fonction du nombre de sommets

```python
def anlayse_nb_iter_by_nb_sommets(nb_train_graph, nb_edges, nb_repetitions, max_nb_sommets, nb_pas = 10):
    moyens_glouton = []
    moyens_random = []

    for i in range(10, max_nb_sommets, nb_pas):
        print(f"iteration {i}")
        moyen_glouton, moyen_random, _, _ = calclulate_mean_nb_iter(nb_train_graph, i, nb_edges, nb_repetitions)
        moyens_glouton.append(moyen_glouton)
        moyens_random.append(moyen_random)

    nb_sommets = [i for i in range(10, max_nb_sommets, nb_pas)]
```

- **Nombres d'itérations par arcs** : Fonction d'analyse en fonction du nombre d'arrêtes

```python
def anlayse_nb_iter_by_nb_edges(nb_train_graph, nb_sommet, nb_repetitions, max_nb_edges, nb_pas = 5):

    moyens_glouton = []
    moyens_random = []

    for i in range(20, max_nb_edges, nb_pas):
        print(f"iteration {i}")
        moyen_glouton, moyen_random, _, _ = calclulate_mean_nb_iter(nb_graphs=nb_train_graph, nb_vertex=nb_sommet, nb_edges=i, nb_repetition=nb_repetitions)
        moyens_glouton.append(moyen_glouton)
        moyens_random.append(moyen_random)

    nb_train_graphs = [i for i in range(20, max_nb_edges, nb_pas)]
```

- **Nombre moyen d'itérations** : Fonction calculant le nombre moyen d'itération obtenus à partir de l'algorithme Bellman-Ford avec l'ordre glouton et l'ordre aléatoire.

```python
def anlayse_nb_iter_by_nb_edges(nb_train_graph, nb_sommet, nb_repetitions, max_nb_edges, nb_pas = 5):

    moyens_glouton = []
    moyens_random = []

    for i in range(20, max_nb_edges, nb_pas):
        print(f"iteration {i}")
        moyen_glouton, moyen_random, _, _ = calclulate_mean_nb_iter(nb_graphs=nb_train_graph, nb_vertex=nb_sommet, nb_edges=i, nb_repetition=nb_repetitions)
        moyens_glouton.append(moyen_glouton)
        moyens_random.append(moyen_random)

    nb_train_graphs = [i for i in range(20, max_nb_edges, nb_pas)]
```