



GROUPE 4
MASTER IMAGES

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

Étudiants
Halimi Abdelkrim
Basci Onur

December 4, 2023

Contents

1	Problématique	2
2	Proposition	2
3	Questions	2
Q1	Algorithme de Bellman-Ford	2
Q2	Algorithme GloutonFas	3
Q3	Génération de graphes d'entraînement et de test	5
Q4	Application de l'Algorithme Bellman-Ford aux graphes de tests, et union de leurs arborescences	7
Q5	Déterminer un ordre optimale sur l'union de nos graphes d'entraînement	8
Q6	Utilisation du graphe de Test	8
Q7	Application de Bellman Ford avec un ordre aléatoires	9
Q8	Comparaison des résultats avec les deux approche	9
Q9	Test de notre approche sur différents graphes	10
Q10	L'influence du nombre de graphes sur le nombre d'itérations.	12
Q10.1	Test supplémentaires	14
Q11	Graphe de 2500 Niveaux : Évaluation de l'Adéquation de la Méthode avec Prétraitement	15
4	Annexes	17

1 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 en $n - 1$ 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 $\text{dist}(<_{\text{tot}}, P)$ pour mesurer l'inversion d'arcs dans un chemin P .

2 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, \dots, 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.

3 Questions

Q1 Algorithme de Bellman-Ford

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) Class Graph

Cette classe représente la structure de nombre graphe orienté.

```
1 class Graph:
2     def __init__(self, list_vertex : list):
3         # liste des sommets
4         self.list_vertex : list = list_vertex
5         # liste des arcs
6         self.list_edges : list = []
7         # un ordre des sommets
8         self.vertex_order : list = None
9         # représentation du graph sous forme de listes d'adjacence
10        self.graph : dict = {x: [] for x in self.list_vertex}
11        # distances pour l'algorithme de Bellman-Ford
12        self.distances : np.ndarray = np.full(len(self.list_vertex), np.inf)
13        # arborescence des plus courts chemins
14        self.paths : list = None
15        # nombre d'itérations pour l'algorithme de Bellman-Ford
16        self.nb_iter : int = 0
```

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.

```

1  def bellman_ford(self, source_vertex, vertex_order=None):
2      self.nb_iter = 0
3      """
4          Fonction qui calcule le plus court chemin entre deux sommets avec l'algorithme de
          ↪ Bellman-Ford
5          :param source_vertex: sommet de départ
6          :param vertex_order: ordre des sommets à parcourir
7          :return: liste des plus court chemins, matrice des distances, nombre d'itérations
8      """
9      if vertex_order is None:
10         vertex_order = self.list_vertex
11
12         # Initialiser la distance du sommet source à 0
13         self.distances = np.full(len(self.list_vertex), np.inf)
14         self.distances[source_vertex] = 0
15         self.paths = [[] for _ in range(len(self.list_vertex))]
16         self.paths[0] = [0]
17
18         # Relaxer les arcs répétitivement
19         for i in range(len(self.list_vertex)):
20             no_updates = True
21             for vertex in vertex_order:
22                 for neighbor, weight in self.graph[vertex]:
23                     if self.distances[vertex] != np.inf and self.distances[vertex] + weight <
                       ↪ self.distances[neighbor]:
24                         self.distances[neighbor] = self.distances[vertex] + weight
25                         self.paths[neighbor] = self.paths[vertex] + [neighbor]
26                         no_updates = False
27
28             self.nb_iter += 1
29             if no_updates:
30                 break
31
32         # Vérifier si il y a un cycle de poids négatif
33         contains_negatif_cyle = i >= len(self.list_vertex)-1
34
35         return self.paths, self.distances, self.nb_iter, contains_negatif_cyle

```

Q2 Algorithme GloutonFas

L'implémentation de l'algorithme de GloutonFas s'est faite comme suit, et nous l'avons également intégrée en tant que méthode de notre classe "Graph".

a) Algorithme principal

Soit un graphe $G = (V, E)$, nous avons choisi d'inverser le graphe pour obtenir les sources. L'inversion du graphe se fait en $O(E)$ ce qui sera crucial pour les questions suivantes. Nous avons inclus en annexe la version utilisant utilisant `get_sources()`

```

1  def glouton_fas_v2(self):
2      """
3          La méthode glouton qui calcule un ordre total à partir de l'attribut graph de la classe
4          :return: Un ordre
5      """
6      s1: list = []
7      s2: list = []
8      graphe = deepcopy(self)
9      inverse_graphe = graphe.get_inverse_graph()
10     while len(graphe.graph.keys()) > 0:
11         sources = inverse_graphe.get_puits()
12         while len(sources) > 0:
13             u = sources[0]

```

```

14         s1.append(u)
15         graphe.delete_vertex(u)
16         inverse_graphe.delete_vertex(u)
17         sources = inverse_graphe.get_puits()
18         puits = graphe.get_puits()
19         while len(puits) > 0:
20             u = puits[0]
21             s2.insert(0, u)
22             graphe.delete_vertex(u)
23             puits = graphe.get_puits()
24         u_max = np.argmax(np.array([graphe.get_diff_enter_exit(vertex) for vertex in
25             ↪ self.graph.keys()])))
26
27         if len(graphe.graph.keys()) > 0:
28             s1.append(u_max)
29             graphe.delete_vertex(u_max)
30     s1.extend(s2)
31     return s1

```

b) Fonction utilitaires

Autres fonctions utiles pour le fonctionnement de l'algorithme principal.

```

1  def get_inverse_graph(self):
2      """
3      :return: Un graphe avec les poids inversés
4      """
5      inverse_graph = Graph(self.list_vertex)
6      inverse_list_edges = [(v, u, w) for u, v, w in self.list_edges]
7      inverse_graph.add_edges(inverse_list_edges)
8      return inverse_graph
9
10 def get_puits(self):
11     """
12     Méthode qui calcule les puits dans l'attribut graphe i.e. les sommet qui n'ont pas de
13     ↪ voisins mais des précédents
14     :return: Une liste de précédents
15     """
16     puits = []
17     for vertex in self.graph.keys():
18         if len(self.graph[vertex]) <= 0:
19             puits.append(vertex)
20
21     return puits
22
23 def delete_vertex(self, vertex_to_delete : int):
24     """
25     Methode qui supprime un sommet
26     :param vertex_to_delete: sommet à supprimer
27     :return: None
28     """
29     #remove vertex
30     del self.graph[vertex_to_delete]
31     #remove precedents
32     for vertex in self.graph.keys():
33         for arcs in self.graph[vertex]:
34             if arcs[0] == vertex_to_delete:
35                 self.graph[vertex].remove(arcs)
36
37 def get_diff_enter_exit(self, vertex : int):
38     """
39     Méthode calculant la difference entre les valeurs entrant et la valeur sortant d'un sommet
40     ↪ dans l'attribut graph
41     :param vertex: Le sommet à calculer la difference
42     :return: La différence
43     """
44     if not(vertex in self.graph.keys()):
45         return -math.inf
46     sum_enter = sum(precedent[1] for precedent in self.get_precedent(vertex))
47     sum_exit = sum(neighbor[1] for neighbor in self.graph[vertex])
48     return sum_exit - sum_enter

```

Q3 Génération de graphes d'entraînement et de test

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

```
1  @staticmethod
2  def generate_random_graph(size_graph : int,nb_edges : int) -> 'Graph':
3      """
4          Fonction qui génère un graphe aléatoire d'une taille donnée
5          """
6      liste_vertex : list = [i for i in range(size_graph)]
7      liste_edges = [tuple(random.sample(liste_vertex,2)+[1]) for _ in range(nb_edges)]
8      new_graph = Graph(liste_vertex)
9      new_graph.add_edges(liste_edges)
10     new_graph = Graph.generate_random_weights(new_graph)
11     return new_graph
```

b) Création des 4 graphes avec des poids aléatoires

Lors de la création de poids aléatoires, il était possible d'obtenir des circuits négatifs. C'est pourquoi, à chaque génération, nous avons pris la décision de les détecter et de les éliminer.

```
1  @staticmethod
2  def generate_random_weights(graph : 'Graph'):
3      """
4          Fonction qui génère des poids aléatoires pour un graphe donnée
5          :param graph: Un graphe orienté sans poids
6          :return: Creation d'un nouveau graphe avec des poids aléatoirement générés
7          """
8
9      # Génération des poids aléatoires entre -10 et 10
10     random_graph = Graph(graph.list_vertex)
11     list_edges = []
12     for vertex in graph.list_vertex:
13         for neighbor, _ in graph.graph[vertex]:
14             random_weight = random.randint(-10, 10)
15             list_edges.append((vertex, neighbor, random_weight))
16     random_graph.add_edges(list_edges)
17
18     # Vérification qu'il n'y a pas de cycle négatif
19     paths, _, _, contains_negatif_cycle = random_graph.bellman_ford(0, None)
20     path = paths[np.argmax(np.array([len(path) for path in paths]))]
21     # Si il y a un cycle négatif, on le supprime
22     while contains_negatif_cycle :
23         # On récupère le cycle négatif
24         negative_cycle = Graph._find_negative_cycle(path)
25         # On convertit les poids négatifs en positif
26         for i in range(len(negative_cycle)-1):
27             for u,v,w in list_edges:
28                 if u == negative_cycle[i] and v == negative_cycle[i+1]:
29                     list_edges.remove((u,v,w))
30                     list_edges.append((u,v,abs(w)))
31
32         # On fait la même chose pour le dernier arc
33         for u, v, w in list_edges:
34             if u == negative_cycle[-1] and v == negative_cycle[0]:
35                 list_edges.remove((u, v, w))
36                 list_edges.append((u, v, abs(w)))
37         random_graph.add_edges(list_edges)
38
39     paths, _, _, contains_negatif_cycle = random_graph.bellman_ford(0, None)
40     path = paths[np.argmax(np.array([len(path) for path in paths]))]
41     return random_graph
```

c) **Elimination des circuits négatifs**

Ici, nous décrivons la méthode de détection des circuits négatifs. Cette approche nous permet de générer rapidement de très grands graphes sans circuits négatifs, ce qui facilite la réalisation de tests plus efficaces.

```
1  @staticmethod
2  def _find_negative_cycle( path : list) -> int:
3      """
4          Fonction qui trouve un cycle à poids négatif
5          :param path: chemin
6          :return: nombre d'itérations
7      """
8      already_seen = {}
9      index_cycle = -1
10     for i,value in enumerate(path):
11         if value in already_seen:
12             index_cycle = i
13             break
14         already_seen[value] = i
15     i_start_cycle = already_seen[path[index_cycle]]
16     cycle = path[i_start_cycle:index_cycle]
17     return cycle
```

d) **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.

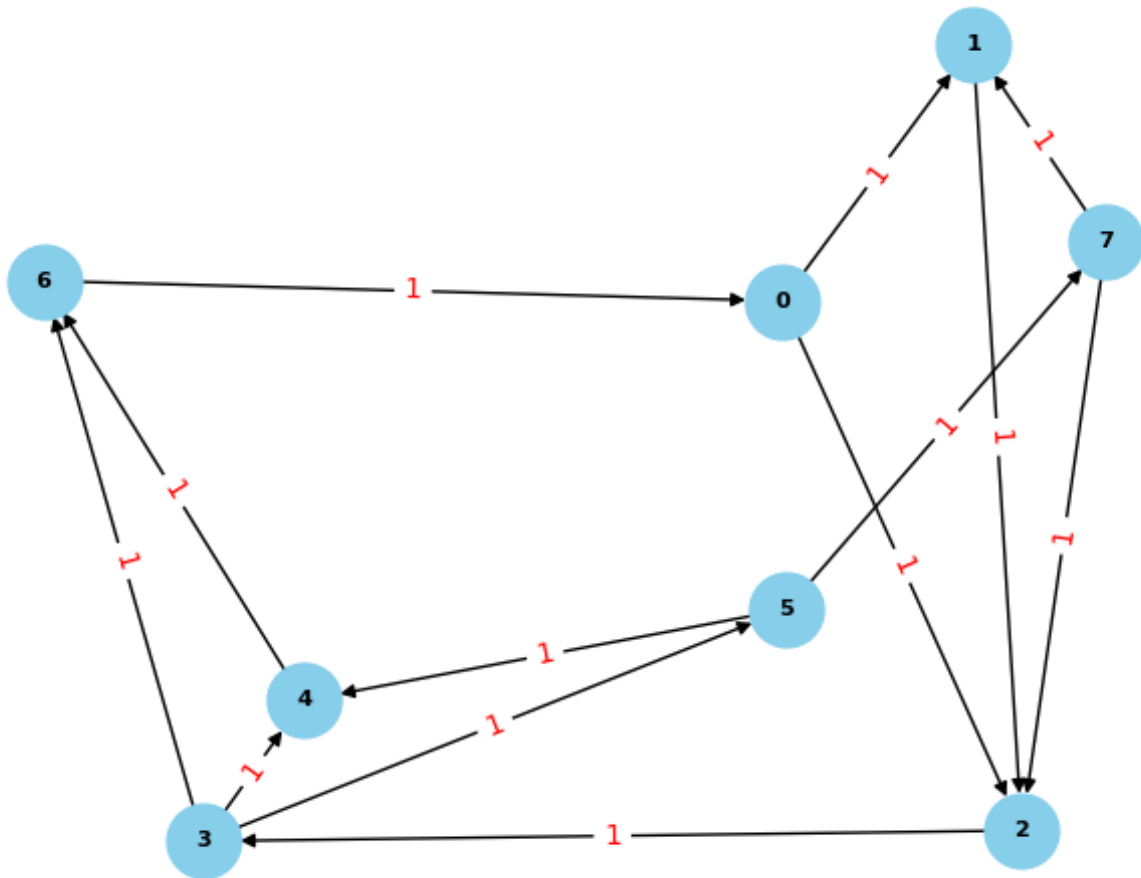


Figure 1: Graphe de base

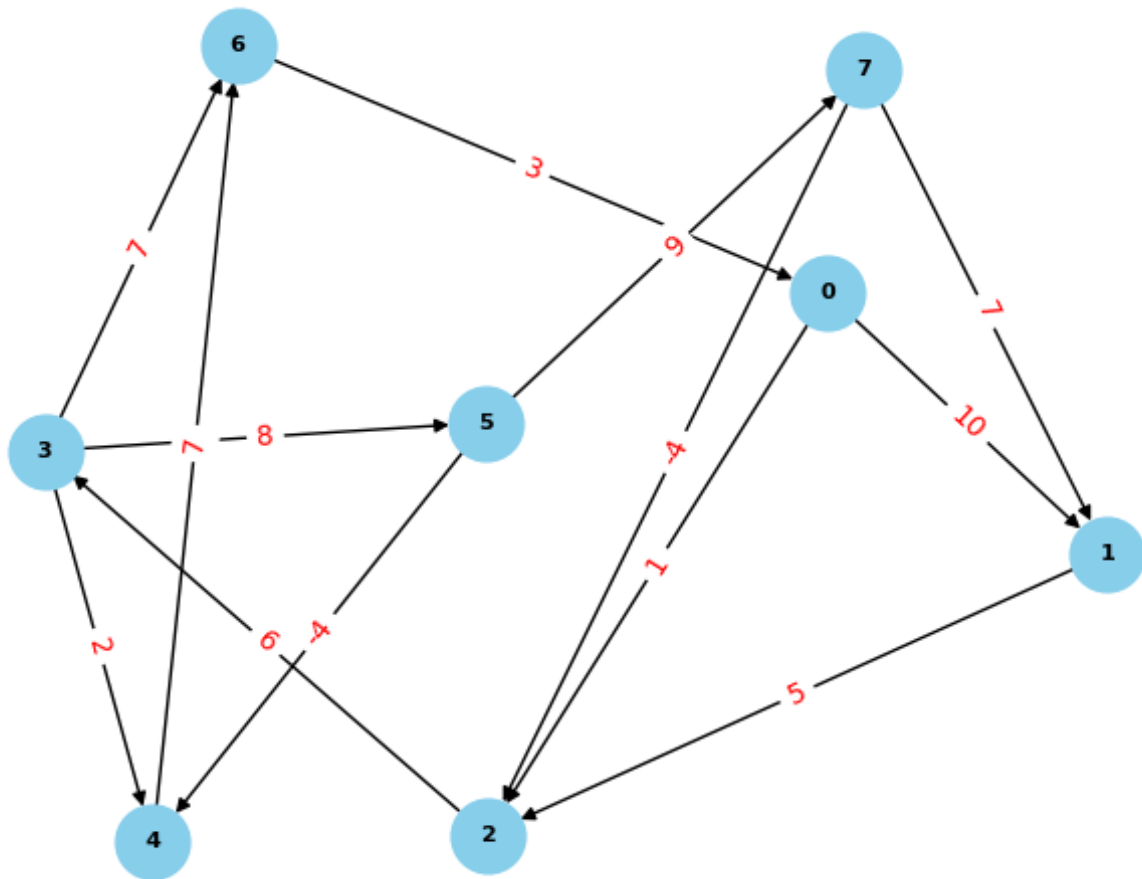


Figure 2: Graphe d'entraînement

Q4 Application de l'Algorithme Bellman-Ford aux graphes de tests, et union de leurs arborescences

Nous avons défini une fonction qui regroupe toutes ces étapes. Il est à noter que l'union de leurs arborescences définira un nouveau graphe.

a) Application de bellman-ford sur nos graphes d'entrainement

```

1 def question_4(graph_1: Graph, graph_2: Graph, graph_3: Graph):
2     """
3         Etant donnees les 3 graphes G1,G2,G3 (g en er es dans la question 3), appliquer
4         l'algorithme Bellman-Ford dans chacun de ces graphes et d eterminer l'union de leurs
5         ↪ arborescences
6         des plus courts chemins que l'on appellera T.
7     """
8     path_1,_,_,_ = graph_1.bellman_ford(0,None)
9     path_2,_,_,_ = graph_2.bellman_ford(0,None)
10    path_3,_,_,_ = graph_3.bellman_ford(0,None)
11
12    union_T : Graph = Graph.unify_paths([path_1,path_2,path_3], graph_1.list_vertex)
13    print("[INFO] Question 4 terminée\n")
14    return union_T

```

b) Fonction qui permet l'unification des arborescences

```
1  @staticmethod
2  def unify_paths(paths : list, list_vertex : list) -> 'Graph':
3      """
4      Fonction qui unifie les chemins pour créer un graph
5      :param paths: chemins des graphes
6      :param list_vertex: liste des sommets
7      :return: un graph
8      """
9      # Initialisation des variables
10     list_edges : list = []
11     added_edges : dict = {}
12     # Fonction qui ajoute un arc au graph si il n'existe pas déjà
13     def add_edge_from_path(path):
14         if len(path) <= 1:
15             return
16         for i in range(0, len(path) - 1):
17             vertex_1, vertex_2 = path[i], path[i+1]
18             if added_edges.get((vertex_1, vertex_2), None) is not None:
19                 continue
20             list_edges.append((path[i], path[i+1], 1))
21             added_edges[(vertex_1, vertex_2)] = True
22
23     # Ajout des arcs pour chaque chemin
24     for p_x in zip(*paths):
25         for p in p_x :
26             add_edge_from_path(p)
27     # Création du graph
28     new_graph = Graph(list_vertex)
29     new_graph.add_edges(list_edges)
30     # Retourne le graph
31     return new_graph
```

Q5 Déterminer un ordre optimale sur l'union de nos graphes d'entraînement

L'objectif des graphes d'entraînement est d'utiliser l'union de leurs arborescences des plus courts chemins. Cela permet d'expérimenter avec GloutonFas l'extraction d'un ordre à utiliser comme heuristique pour l'algorithme de Bellman-Ford.

Application de GloutonFas sur l'arborescence obtenu

```
1  def question_5(union_T):
2      """
3      Appliquer l'algorithme GloutonFas avec comme entrée T et retourner un ordre <tot.
4      """
5      print("Question 5: Calcul de l'ordre pour le graphe d'union")
6      ordre = union_T.glouton_fas()
7      print(f"Question 5 \n {ordre}")
8      return ordre
```

Ordre obtenu

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

Q6 Utilisation du graphe de Test

Après l'entraînement, nous obtenons un ordre que nous passerons en paramètre à l'algorithme de Bellman-Ford appliqué à notre graphe de test. La fonction est décrite ci-dessous.

Utilisation de l'ordre sur notre graphe de test

```
1  ordre = question_5(union_T)
2
3  def question_6(H, ordre):
4      """
5      Pour le graphe H généré à la question 3 appliquer l'algorithme Bellman-Ford en
6      utilisant l'ordre <tot.
7      """
8      print("Question 6: Application de Bellman Ford à H")
9      return H.bellman_ford(0,ordre)
```

Q7 Application de Bellman Ford avec un ordre aléatoires

Nous allons réaliser 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

```
1  @staticmethod
2  def generate_random_order(graph):
3      """
4      Fonction qui génère un ordre aléatoire pour un graphe donnée
5      """
6      nb_elem = len(graph.list_vertex)
7      return random.sample(range(nb_elem), nb_elem)
```

Utilisation d'un ordre aléatoire sur notre graphe de test

```
1  def question_7(graph):
2      """
3      Pour le graphe H appliquer l'algorithme Bellman-Ford en utilisant un ordre tiré
4      aléatoirement (de manière uniforme).
5      """
6      ordre = Graph.generate_random_order(graph)
7      path,dist,nb_iter,state = graph.bellman_ford(0, ordre)
8      print("[INFO] Question 7 terminée\n")
9      return path,dist,nb_iter,state
```

Q8 Comparaison des résultats avec les deux approche

a) Fonction comparant les deux approche

```
1  def question_8(Bellman_H, Bellman_H_random):
2      """
3      Comparer les résultats (nombre d'itérations) obtenus dans les questions 6 et 7.
4      """
5      print("Comparaison Bellman avec prétraitement et Bellman avec un ordre aléatoire")
6      if not(Bellman_H) or not(Bellman_H_random):
7          return
```

```
8     print(f"Nb itération, Bellman avec prétraitement: {Bellman_H[2]}")
9     print(f"Nb itération, Bellman avec un ordre aléatoire: {Bellman_H_random[2]}")
10    print("[INFO] Question 8 terminée\n")
```

b) 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

```
1  Comparaison Bellman avec prétraitement et Bellman avec un ordre aléatoire
2  Nb itération, Bellman avec prétraitement: 2
3  Nb itération, Bellman avec un ordre aléatoire: 5
4  [INFO] Question 8 terminée
```

Q9 Test de notre approche sur différents graphes

Nous avons exécuté bellman-ford sur plus de 100 graphes aléatoire, en utilisant un ordre aléatoire, et un ordre estimé avec GloutonFas.

Paramètres utilisés

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

Nombre d'itérations en fonction de différents graphes

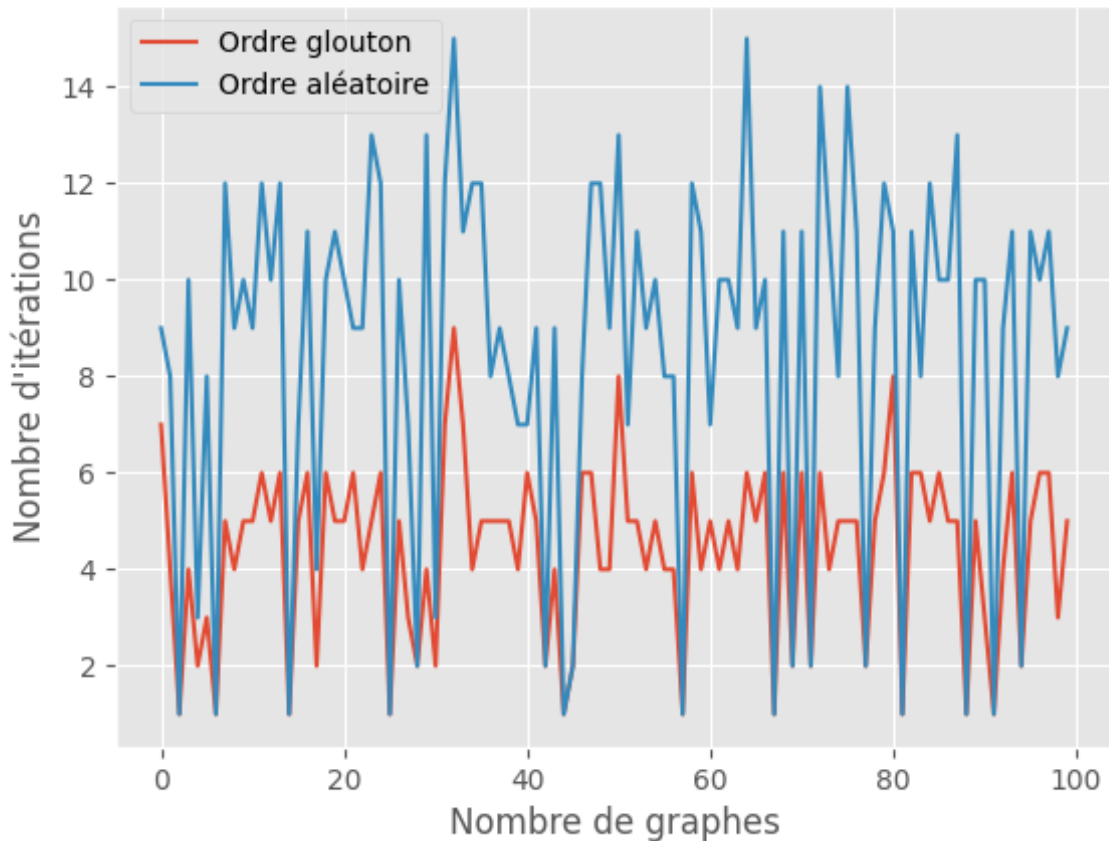


Figure 3: Comparaison des 2 approches sur plus de 100 grahes

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 similaires. 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()*

Fonction utilisé pour les résultats

```
1 def anlayse_nb_iter_by_random_graph(nb_graphs=10, nb_vertex=100, nb_edges=400):
2     nb_edges = nb_vertex*2
3     liste_nb_iter_glouton = []
4     liste_nb_iter_random = []
5     for _ in range(nb_graphs):
6         nb_iter_glouton, nb_iter_random =
7             ↪ Graph.generate_compare_graph(nb_vertex, nb_edges, nb_graph_to_generate=10)
8         liste_nb_iter_glouton.append(nb_iter_glouton)
9         liste_nb_iter_random.append(nb_iter_random)
10    plt.style.use("ggplot")
```

```

11     plt.figure()
12     plt.plot(liste_nb_iter_glouton, label="Ordre glouton")
13     plt.plot(liste_nb_iter_random, label="Ordre aléatoire")
14     plt.legend()
15     plt.title("Nombre d'itérations en fonction de différents graphes")
16     plt.xlabel("Nombre de graphes")
17     plt.ylabel("Nombre d'itérations")
18     plt.show()
19
20     mean_glouton = sum(liste_nb_iter_glouton)/len(liste_nb_iter_glouton)
21     mean_random = sum(liste_nb_iter_random)/len(liste_nb_iter_random)
22     print(f"moyenne glouton {mean_glouton}")
23     print(f"moyenne random {mean_random}")
24     print(mean_random/mean_glouton)

```

Q10 L'influence du nombre de graphes 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.

Moyens nombre d'itérations en fonction de Graphe d'entraînement

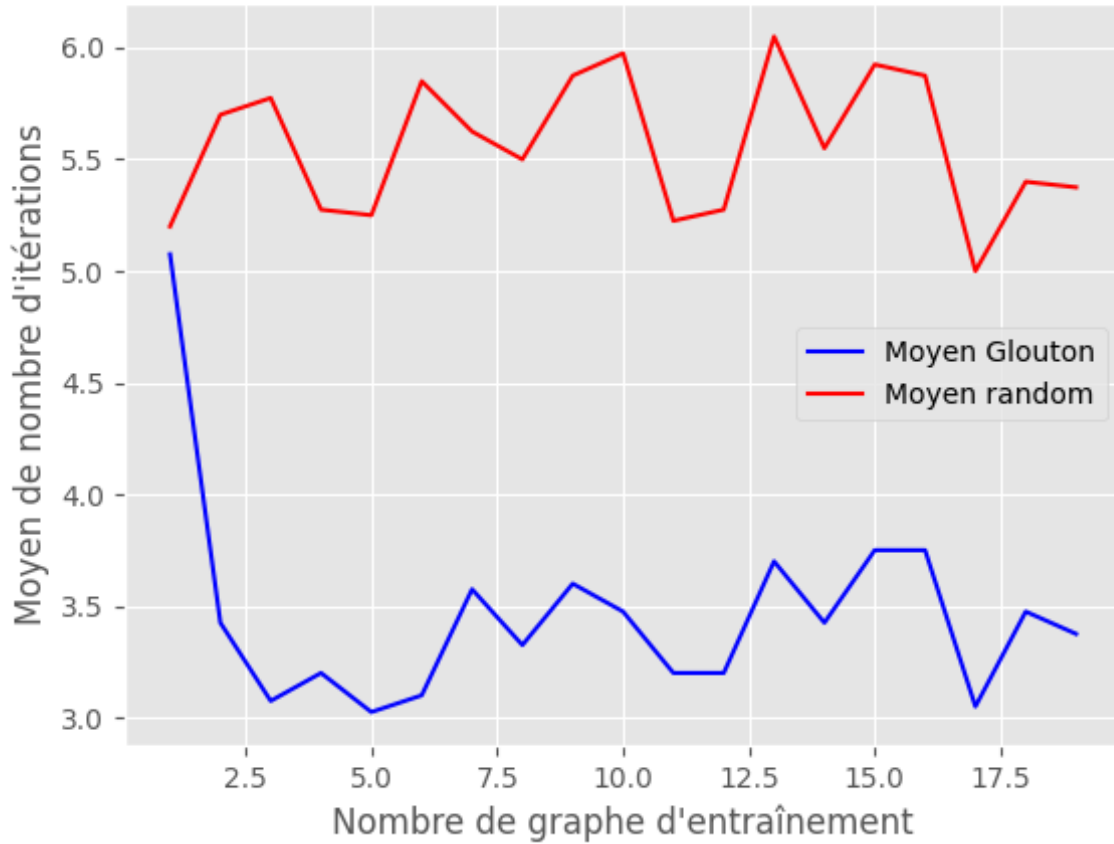


Figure 4: Comparaison des moyennes du nombre d'itérations en fonction du nombre de graphes d'entraînement. $NS = 30$, $NA = 75$, $NR = 40$

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 tests précédents, 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

Moyens nombre d'itérations en fonction de nombre de sommet

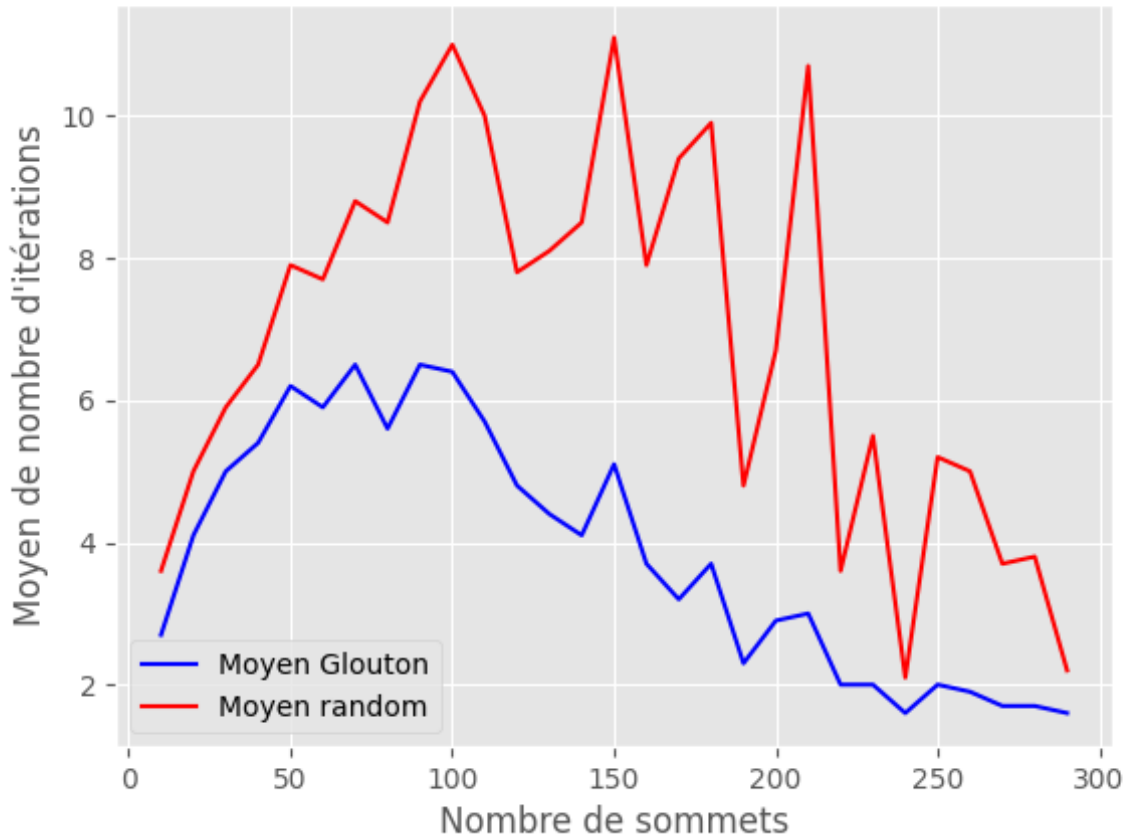


Figure 5: Comparaison de moyennes de nombre d'itérations en fonction de nombre de Sommets. $NA = 300$, $NA = 3$, $NR = 10$

Résultats Il semble que 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 300 arrêtes au maximum. Par conséquent, un nombre moindre d'itérations est nécessaire pour appliquer l'algorithme de Bellman.

b) Tests en fonction de nombre d'arêtes

Moyens nombre d'itérations en fonction de nombre d'arêtes

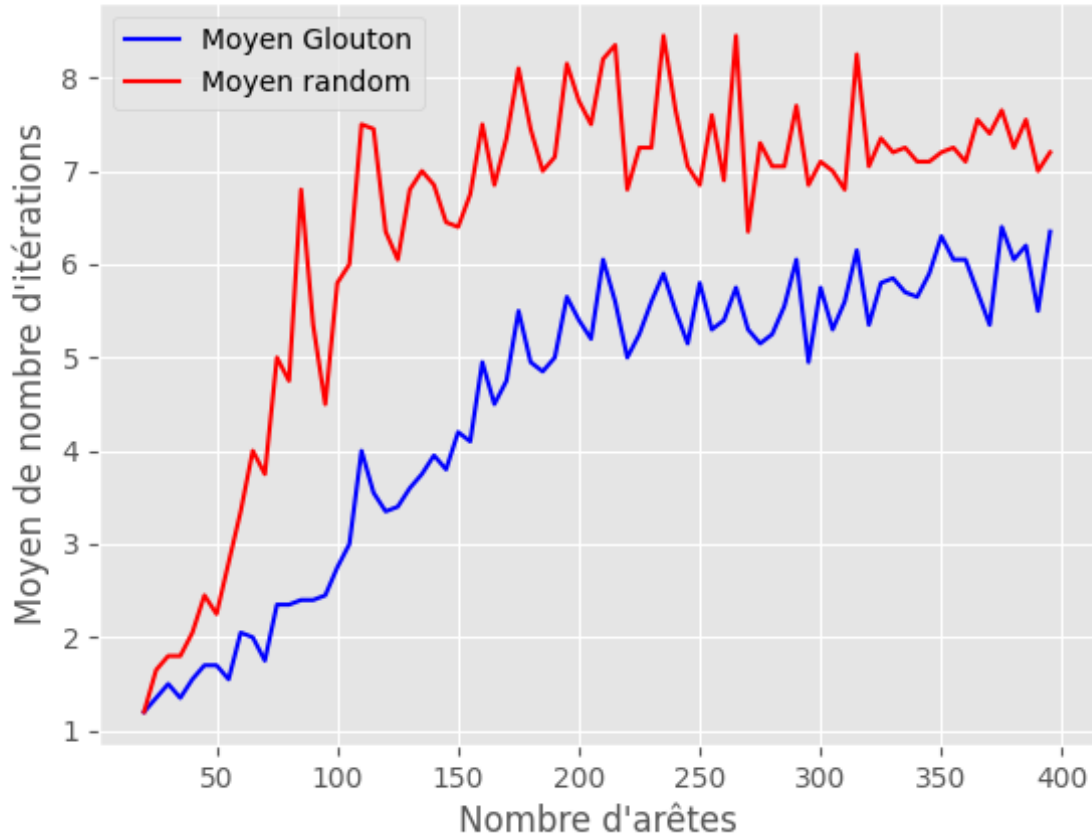


Figure 6: Comparaison des moyennes du nombre d'itérations en fonction du nombre d'arêtes. $NS = 50$, $NA = 3$, $NR = 20$

Résultats On remarque 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 $NA = 200$ dans la figure 6 ci-dessus.

Q11 Graphe de 2500 Niveaux : Évaluation de l'Adéquation de la Méthode avec Prétraitement

Considérons un graphe G organisé par niveaux, tel que les sommets du niveau j précèdent tous les sommets du niveau $j + 1$. Dans cet exemple, le graphe est constitué de 4 sommets par niveau et comporte 2500 niveaux.

En ce qui concerne les sommets de niveau k , la distance minimale de S est égale à 0 (car c'est la source), tandis que la distance minimale pour les autres sommets est infinie car ils

ne sont pas accessibles. De plus, pour les niveaux inférieurs à k , les distances minimales sont également infinies car ces niveaux ne sont plus accessibles. Il en résulte que pour minimiser le nombre d'itérations, il est préférable de suivre un ordre croissant afin de calculer les distances minimales des sommets du niveau i avant celles du niveau $i + 1$. En respectant cet ordre, lors de la mise à jour des distances des sommets de niveau $i + 1$, l'information nécessaire pour calculer la distance minimale est déjà disponible.

Ainsi, l'ordre optimal à suivre est un ordre croissant dans le sens des niveaux, et il est même possible d'affirmer que deux itérations sont suffisantes pour appliquer l'algorithme. Durant la première itération, toutes les distances minimales pour chaque niveau sont calculées, et lors de la deuxième itération, on vérifie si les distances ont changé (ce qui n'est pas le cas, comme démontré par la minimalité des distances).

Nous distinguons donc 2 cas :

1) Structure du graphe connue

Lorsque la structure décrite ci-dessus est connue à l'avance, l'algorithme GloutonFas devient superflu, car il suffit de sélectionner les sommets dans l'ordre croissant pour minimiser le nombre d'itérations.

2) Structure du graphe inconnue

Pour ce type de graphe, l'algorithme GloutonFas se révèle très efficace. Démontrons à présent que l'algorithme GloutonFas nous fournit l'ordre optimal. L'algorithme commence par éliminer les sources dans G . Initialement, seuls les sommets de niveau 1 sont des sources, car tous les sommets dans un niveau supérieur à 1 ont leurs prédécesseurs dans le niveau précédent. Par conséquent, il n'y a pas de source dans chaque niveau tant que le niveau précédent n'est pas supprimé. L'algorithme GloutonFAS commence donc par supprimer les sommets de niveau 1, puis ceux de niveau 2 jusqu'au niveau 2500. Pour chaque sommet supprimé, il le place à la fin de la liste, créant ainsi un ordre croissant.

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

```

1  @staticmethod
2  def generate_level_graph(nb_level):
3      """
4          Cette fonction crée un graph avec des niveau comme expliqué dans la question 11
5          :param nb_level: nombre de niveau
6          :level_graph: La fonction générée de la structure expliqué dans la question 11
7      """
8      level_graph = Graph([i for i in range(nb_level * 4)])
9
10     edges = []
11     for i in range(nb_level - 1):
12         next_level_vertexes = [(i * 4) + 4 + k for k in range(4)]
13         for j in range(4):
14             for k in next_level_vertexes:
15                 edges.append(((i * 4) + j, k, 1))
16     level_graph.add_edges(edges)
17     level_graph = Graph.generate_random_weights(level_graph)
18
19     return level_graph

```

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.

Génération d'un ordre aléatoire

```
1  moyenne glouton 2.0
2  moyenne random 1216.0
```

4 Annexes

Fonction `get_sources()`

```
1  def get_sources(self):
2      """
3          Méthode qui calcule les sources dans l'attribut graphe i.e. les sommet qui n'ont pas de
4          ↪ précédents mais des voisins
5          :return: Une liste de sources
6          """
7      sources = []
8      for vertex in self.graph.keys():
9          if len(self.get_precedent(vertex)) <= 0:
10             sources.append(vertex)
11
12     return sources
```

Fonction d'analyses

(a) Nombres d'itérations par graphes

```
1  def anlayse_nb_iter_by_random_graph(nb_graphs=10, nb_vertex=100, nb_edges=400):
2      nb_edges = nb_vertex*2
3      liste_nb_iter_glouton = []
4      liste_nb_iter_random = []
5      for _ in range(nb_graphs):
6          nb_iter_glouton, nb_iter_random =
7          ↪ Graph.generate_compare_graph(nb_vertex, nb_edges, nb_graph_to_generate=10)
8          liste_nb_iter_glouton.append(nb_iter_glouton)
9          liste_nb_iter_random.append(nb_iter_random)
```

(b) Nombres d'itérations par graphes d'entrainements

```
1  def anlayse_nb_iter_by_nb_trainGraph(nb_vertex, nb_edges, nb_repetitions, max_nb_trainGraph):
2      """
3          Faire une analyse de la performance en fonction de nombre de graphe d'entraînement
4          :param nb_vertex: nombre de sommets
5          :param nb_edges: nombre d'arêtes
6          :param nb_repetitions: nombre de fois qu'on calcule la moyenne par itérations
7          :param max_nb_trainGraph: Maximum de nombre de graphe d'entraînement
8          :return: None
9          """
10     moyens_glouton = []
11     moyens_random = []
12
13     for i in range(1, max_nb_trainGraph):
14         print(f"iteration {i}")
15         moyen_glouton, moyen_random, _, _ = calculatate_mean_nb_iter(i, nb_vertex, nb_edges,
16             ↪ nb_repetitions)
```

```

16         moyens_glouton.append(moyen_glouton)
17         moyens_random.append(moyen_random)
18
19     nb_train_graph = [i for i in range(1, max_nb_trainGraph)]

```

(c) Nombres d'itérations par sommets

```

1  def analyse_nb_iter_by_nb_sommets(nb_train_graph, nb_edges, nb_repetitions, max_nb_sommets,
↪  nb_pas = 10):
2      """
3      Faire une analyse de la performance en fonction de nombre de graphe d'entraînement
4      :param nb_train_graph: Nombre de graphe d'entraînement
5      :param nb_edges: Nombre d'arêtes
6      :param nb_repetitions: nombre de fois qu'on calcule la moyenne par itérations
7      :param max_nb_sommets: Maximum de nombre de sommet
8      :param nb_pas: la difference entre deux nb sommets
9      :return: None
10     """
11     moyens_glouton = []
12     moyens_random = []
13
14     for i in range(10, max_nb_sommets, nb_pas):
15         print(f"iteration {i}")
16         moyen_glouton, moyen_random, _, _ = calculate_mean_nb_iter(nb_train_graph, i, nb_edges,
↪  nb_repetitions)
17         moyens_glouton.append(moyen_glouton)
18         moyens_random.append(moyen_random)
19
20     nb_sommets = [i for i in range(10, max_nb_sommets, nb_pas)]

```

(d) Nombres d'itérations par arcs

```

1  def analyse_nb_iter_by_nb_edges(nb_train_graph, nb_sommet, nb_repetitions, max_nb_edges, nb_pas =
↪  5):
2      """
3      :param nb_train_graph: Nombre de graphe d'entraînement
4      :param nb_sommet: Nombre de sommet
5      :param nb_repetitions: nombre de fois qu'on calcule la moyenne par itérations
6      :param max_nb_edges: Maximum de nombre d'arêtes
7      :param nb_pas: la difference entre deux nombre d'arêtes.
8      :return:
9      """
10     moyens_glouton = []
11     moyens_random = []
12
13     for i in range(20, max_nb_edges, nb_pas):
14         print(f"iteration {i}")
15         moyen_glouton, moyen_random, _, _ = calculate_mean_nb_iter(nb_graphs=nb_train_graph,
↪  nb_vertex=nb_sommet, nb_edges=i, nb_repetition=nb_repetitions)
16         moyens_glouton.append(moyen_glouton)
17         moyens_random.append(moyen_random)
18
19     nb_train_graphs = [i for i in range(20, max_nb_edges, nb_pas)]

```
