
Groupe 4
Master IMAges

---

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

---

**Étudiants**
Halimi Abdelkrim
Basci Onur

December 4, 2023

# Contents

# 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 $dist(<_{\text{tot}}, P)$ 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 $w_1, w_2, \ldots, w_k$, 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.

# 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.

```python
class Graph:
    def __init__(self, list_vertex : list):
        # liste des sommets
        self.list_vertex : list = list_vertex
        # liste des arcs
        self.list_edges : list = []
        # un ordre des sommets
        self.vertex_order : list = None
        # représentation du graph sous forme de listes d'adjacence
        self.graph : dict = {x: [] for x in self.list_vertex}
        # distances pour l'algorithm de Bellman-Ford
        self.distances : np.ndarray = np.full(len(self.list_vertex), np.inf)
        # arborescence des plus courts chemins
        self.paths : list = None
        # nombre d'itérations pour l'algorithm de Bellman-Ford
        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.

```python
def bellman_ford(self, source_vertex, vertex_order=None):
    self.nb_iter = 0
    """
        Fonction qui calcule le plus court chemin entre deux sommets avec l'algorithme de Bellman-Ford
        :param source_vertex: sommet de départ
        :param vertex_order: ordre des sommets à parcourir
        :return: liste des plus court chemins, matrice des distances, nombre d'itérations
    """
    if vertex_order is None:
        vertex_order = self.list_vertex

    # Initialiser la distance du sommet source à 0
    self.distances = np.full(len(self.list_vertex), np.inf)
    self.distances[source_vertex] = 0
    self.paths = [[] for _ in range(len(self.list_vertex))]
    self.paths[0] = [0]

    # Relaxer les arcs répétitivement
    for i in range(len(self.list_vertex)):
        no_updates = True
        for vertex in vertex_order:
            for neighbor, weight in self.graph[vertex]:
                if self.distances[vertex] != np.inf and self.distances[vertex] + weight < self.distances[neighbor]:
                    self.distances[neighbor] = self.distances[vertex] + weight
                    self.paths[neighbor] = self.paths[vertex] + [neighbor]
                    no_updates = False

        self.nb_iter += 1
        if no_updates:
            break

    # Vérifier si il y a un cycle de poids négatif
    contains_negatif_cyle = i >= len(self.list_vertex)-1

    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*()

```python
def glouton_fas_v2(self):
    """
    La méthode glouton qui calcule un ordre total à partir de l'attribut graph de la classe
    :return: Un ordre
    """
    s1: list = []
    s2: list = []
    graphe = deepcopy(self)
    inverse_graphe = graphe.get_inverse_graph()
    while len(graphe.graph.keys()) > 0:
        sources = inverse_graphe.get_puits()
        while len(sources) > 0:
            u = sources[0]
```

```
            s1.append(u)
            graphe.delete_vertex(u)
            inverse_graphe.delete_vertex(u)
            sources = inverse_graphe.get_puits()
        puits = graphe.get_puits()
        while len(puits) > 0:
            u = puits[0]
            s2.insert(0, u)
            graphe.delete_vertex(u)
            puits = graphe.get_puits()
        u_max = np.argmax(np.array([graphe.get_diff_enter_exit(vertex) for vertex in
        ↪  self.graph.keys()]))

        if len(graphe.graph.keys()) > 0:
            s1.append(u_max)
            graphe.delete_vertex(u_max)
    s1.extend(s2)
    return s1
```

b) **Fonction utilitaires**

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

```
def get_inverse_graph(self):
        """
        :return: Un graphe avec les poids inversés
        """
        inverse_graph = Graph(self.list_vertex)
        inverse_list_edges = [(v, u, w) for u, v, w in self.list_edges]
        inverse_graph.add_edges(inverse_list_edges)
        return inverse_graph

def get_puits(self):
    """
        Méthode qui calcule les puits dans l'attribut graphe i.e. les sommet qui n'ont pas de
        ↪  voisins mais des précedents
        :return: Une liste de précedents
    """
    puits = []
    for vertex in self.graph.keys():
        if len(self.graph[vertex]) <= 0:
            puits.append(vertex)

    return puits

def delete_vertex(self, vertex_to_delete : int):
    """
        Methode qui supprime un sommet
        :param vertex_to_delete: sommet à supprimer
        :return: None
    """
    #remove vertex
    del self.graph[vertex_to_delete]
    #remove precedents
    for vertex in self.graph.keys():
        for arcs in self.graph[vertex]:
            if arcs[0] == vertex_to_delete:
                self.graph[vertex].remove(arcs)

def get_diff_enter_exit(self, vertex : int):
    """
        Méthode calculant la difference entre les valeurs entrant et la valeur sortant d'un sommet
        ↪  dans l'attribut graph
        :param vertex: Le sommet à calculer la difference
        :return: La différence
    """
    if not(vertex in self.graph.keys()):
        return -math.inf
    sum_enter = sum(precedent[1] for precedent in self.get_precedent(vertex))
    sum_exit = sum(neighbor[1] for neighbor in self.graph[vertex])
    return sum_exit - sum_enter
```

## Q3 Génération de graphes d'entrainement 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

```python
@staticmethod
def generate_random_graph(size_graph : int,nb_edges : int) -> 'Graph':
    """
        Fonction qui génère un graph aléatoire d'une taile donnée
    """
    liste_vertex : list = [i for i in range(size_graph)]
    liste_edges = [tuple(random.sample(liste_vertex,2)+[1]) for _ in range(nb_edges)]
    new_graph = Graph(liste_vertex)
    new_graph.add_edges(liste_edges)
    new_graph = Graph.generate_random_weights(new_graph)
    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.

```python
@staticmethod
def generate_random_weights(graph : 'Graph'):
    """
    Fonction qui génère des poids aléatoires pour un graphe donnée
    :param graph: Un graphe orienté sans poids
    :return: Creation d'un nouveau graphe avec des poids aléatoirement générés
    """

    # Génération des poids aléatoires entre -10 et 10
    random_graph = Graph(graph.list_vertex)
    list_edges = []
    for vertex in graph.list_vertex:
        for neighbor, _ in graph.graph[vertex]:
            random_weight = random.randint(-10, 10)
            list_edges.append((vertex, neighbor, random_weight))
    random_graph.add_edges(list_edges)

    # Vérification qu'il n'y a pas de cycle négatif
    paths, _, _, contains_negatif_cycle = random_graph.bellman_ford(0, None)
    path = paths[np.argmax(np.array([len(path) for path in paths]))]
    # Si il y a un cycle négatif, on le supprime
    while contains_negatif_cycle :
        # On récupère le cycle négatif
        negative_cycle = Graph._find_negative_cycle(path)
        # On convertit les poids négatifs en positif
        for i in range(len(negative_cycle)-1):
            for u,v,w in list_edges:
                if u == negative_cycle[i] and v == negative_cycle[i+1]:
                    list_edges.remove((u,v,w))
                    list_edges.append((u,v,abs(w)))

        # On fait la même chose pour le dernier arc
        for u, v, w in list_edges:
            if u == negative_cycle[-1] and v == negative_cycle[0]:
                list_edges.remove((u, v, w))
                list_edges.append((u, v, abs(w)))
        random_graph.add_edges(list_edges)

        paths, _, _, contains_negatif_cycle = random_graph.bellman_ford(0, None)
        path = paths[np.argmax(np.array([len(path) for path in paths]))]
    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.

```python
@staticmethod
def _find_negative_cycle( path : list) -> int:
    """
        Fonction qui trouve un cycle à poids négatif
        :param path: chemin
        :return: nombre d'itérations
    """
    already_seen = {}
    index_cycle = -1
    for i,value in enumerate(path):
        if value in already_seen:
            index_cycle = i
            break
        already_seen[value] = i
    i_start_cycle = already_seen[path[index_cycle]]
    cycle = path[i_start_cycle:index_cycle]
    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.

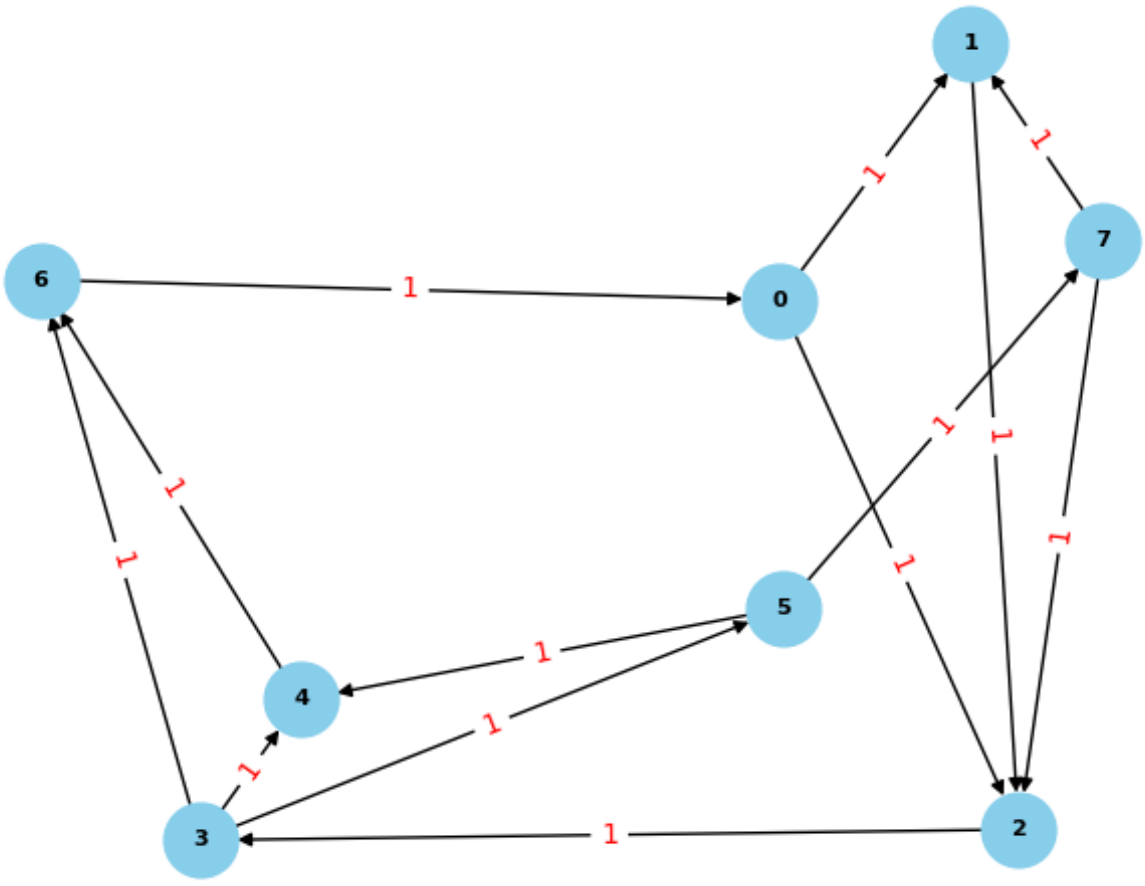


Figure 1: Graphe de base

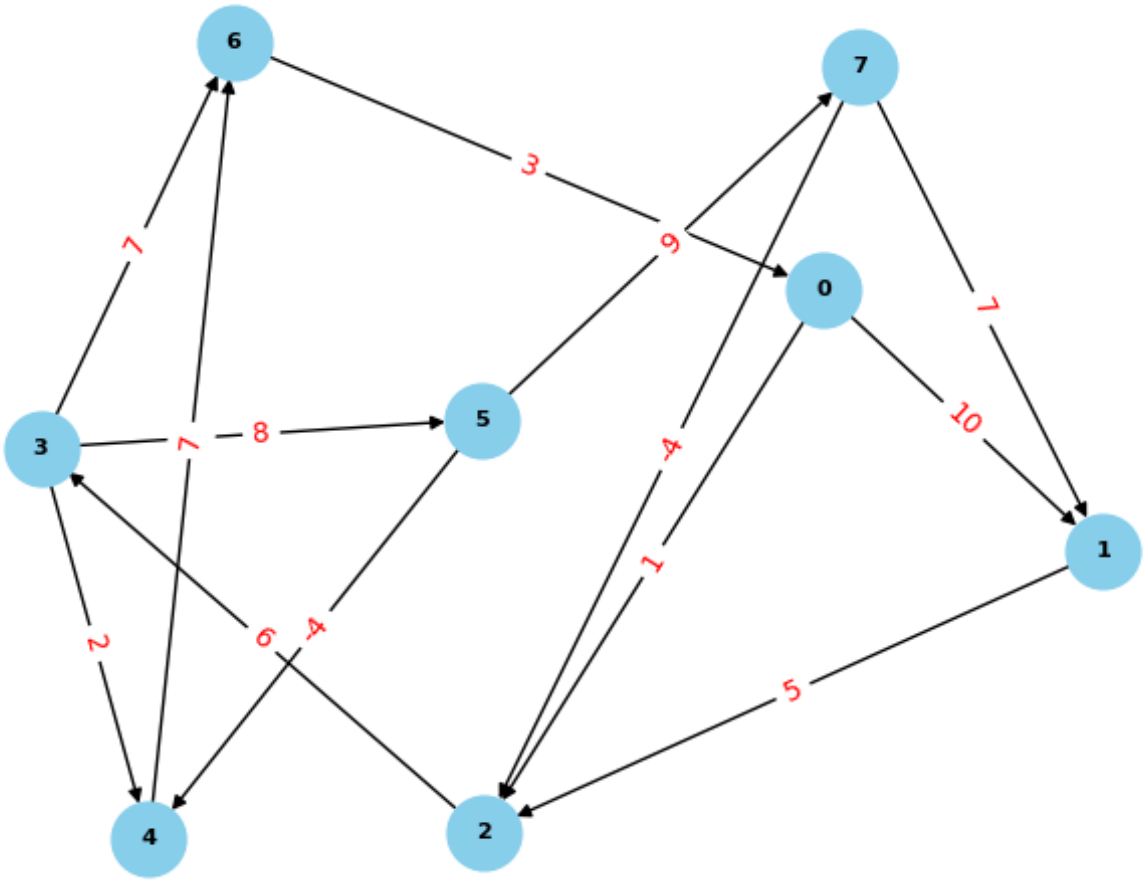


Figure 2: Graphe d'entraînement

## 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

```
def question_4(graph_1: Graph, graph_2: Graph, graph_3: Graph):
    """
        Etant donnes les 3 graphes G1,G2,G3 (g en er es dans la question 3), appliquer
        l'algorithme Bellman-Ford dans chacun de ces graphes et d eterminer l'union de leurs
        ↪  arborescences
        des plus courts chemins que l'on appelera T.
    """
    path_1,_,_,_ = graph_1.bellman_ford(0,None)
    path_2,_,_,_ = graph_2.bellman_ford(0,None)
    path_3,_,_,_ = graph_3.bellman_ford(0,None)


    union_T : Graph = Graph.unifiy_paths([path_1,path_2,path_3], graph_1.list_vertex)
    print("[INFO] Question 4 terminée\n")
    return union_T
```

b) **Fonction qui permet l'unification des arborescences**

```python
@staticmethod
def unifiy_paths(paths : list,list_vertex : list) -> 'Graph':
    """
    Fonction qui unifie les chemins pour créer un graph
    :param paths: chemins des graphes
    :param list_vertex: liste des sommets
    :return: un graph
    """
    # Initialisation des variables
    list_edges : list = []
    added_edges : dict = {}
    # Fonction qui ajoute un arc au graph si il n'existe pas déjà
    def add_edge_from_path(path):
        if len(path) <= 1:
            return
        for i in range(0,len(path) - 1):
            vertex_1,vertex_2 = path[i],path[i+1]
            if added_edges.get((vertex_1,vertex_2),None) is not None:
                continue
            list_edges.append((path[i],path[i+1],1))
            added_edges[(vertex_1,vertex_2)] = True

    # Ajout des arcs pour chaque chemin
    for p_x in zip(*paths):
        for p in p_x :
            add_edge_from_path(p)
    # Création du graph
    new_graph = Graph(list_vertex)
    new_graph.add_edges(list_edges)
    # Retourne le graph
    return new_graph
```

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

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

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

### Ordre obtenu

```
[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.

### Utlisation de l'ordre sur notre graphe de test

```
ordre = question_5(union_T)

def question_6(H, ordre):
    """
    Pour le graphe H généré à la question 3 appliquer l'algorithme Bellman-Ford en
    utilisant l'ordre <tot.
    """
    print("Question 6: Application de Bellman Ford à H")
    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

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

### Utlisation d'un ordre aléatoire sur notre graphe de test

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

## Q8 Comparaison des résultats avec les deux approche

### a) Fonction comparant les deux approche

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

```
    print(f"Nb itération, Bellman avec prétraitement: {Bellman_H[2]}")
    print(f"Nb itération, Bellman avec un ordre aléatoire: {Bellman_H_random[2]}")
    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

```
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
```

## 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

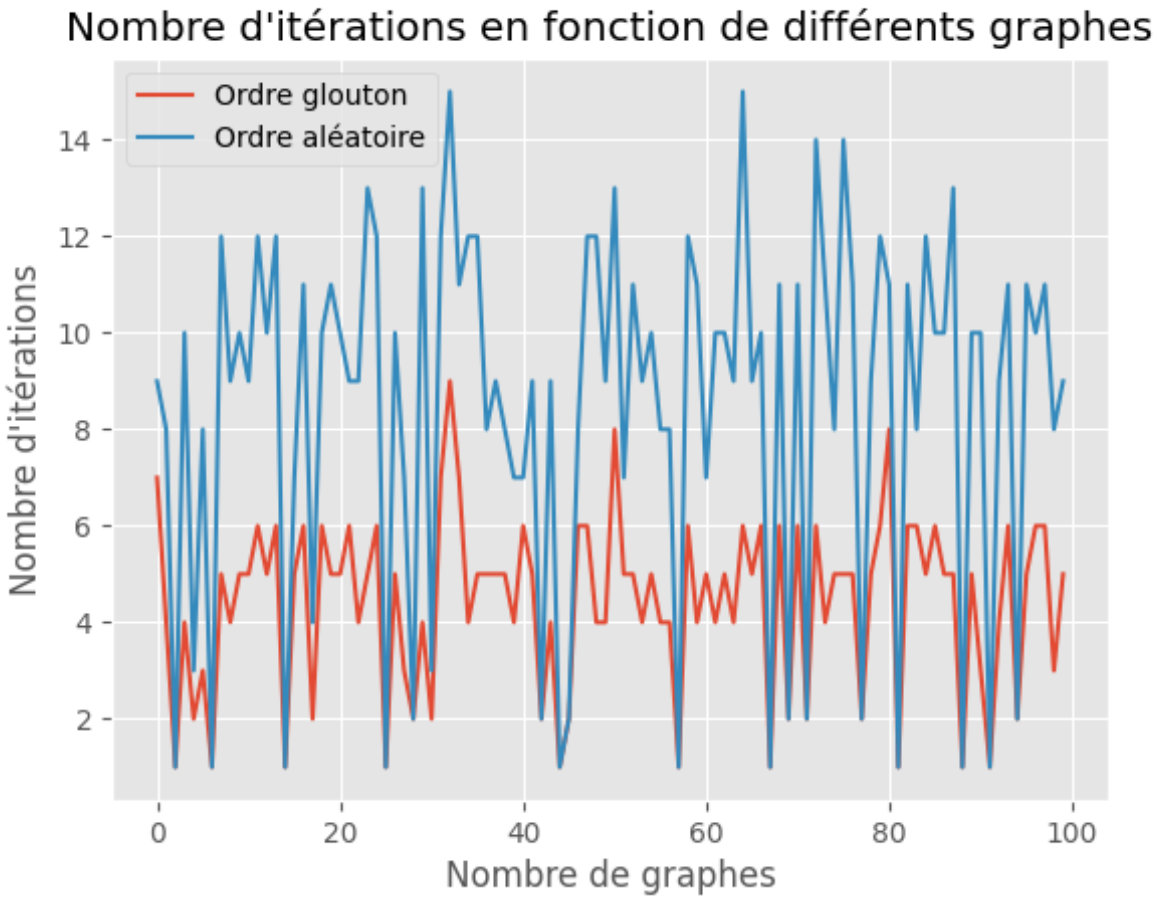


Figure 3: Comparaison des 2 approches sur plus de 100 grahes

### 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

```
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)

    plt.style.use("ggplot")
```

```
    plt.figure()
    plt.plot(liste_nb_iter_glouton, label="Ordre glouton")
    plt.plot(liste_nb_iter_random, label="Ordre aléatoire")
    plt.legend()
    plt.title("Nombre d'itérations en fonction de différents graphes")
    plt.xlabel("Nombre de graphes")
    plt.ylabel("Nombre d'itérations")
    plt.show()

    mean_glouton = sum(liste_nb_iter_glouton)/len(liste_nb_iter_glouton)
    mean_random = sum(liste_nb_iter_random)/len(liste_nb_iter_random)
    print(f"moyenne glouton {mean_glouton}")
    print(f"moyenne random {mean_random}")
    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.

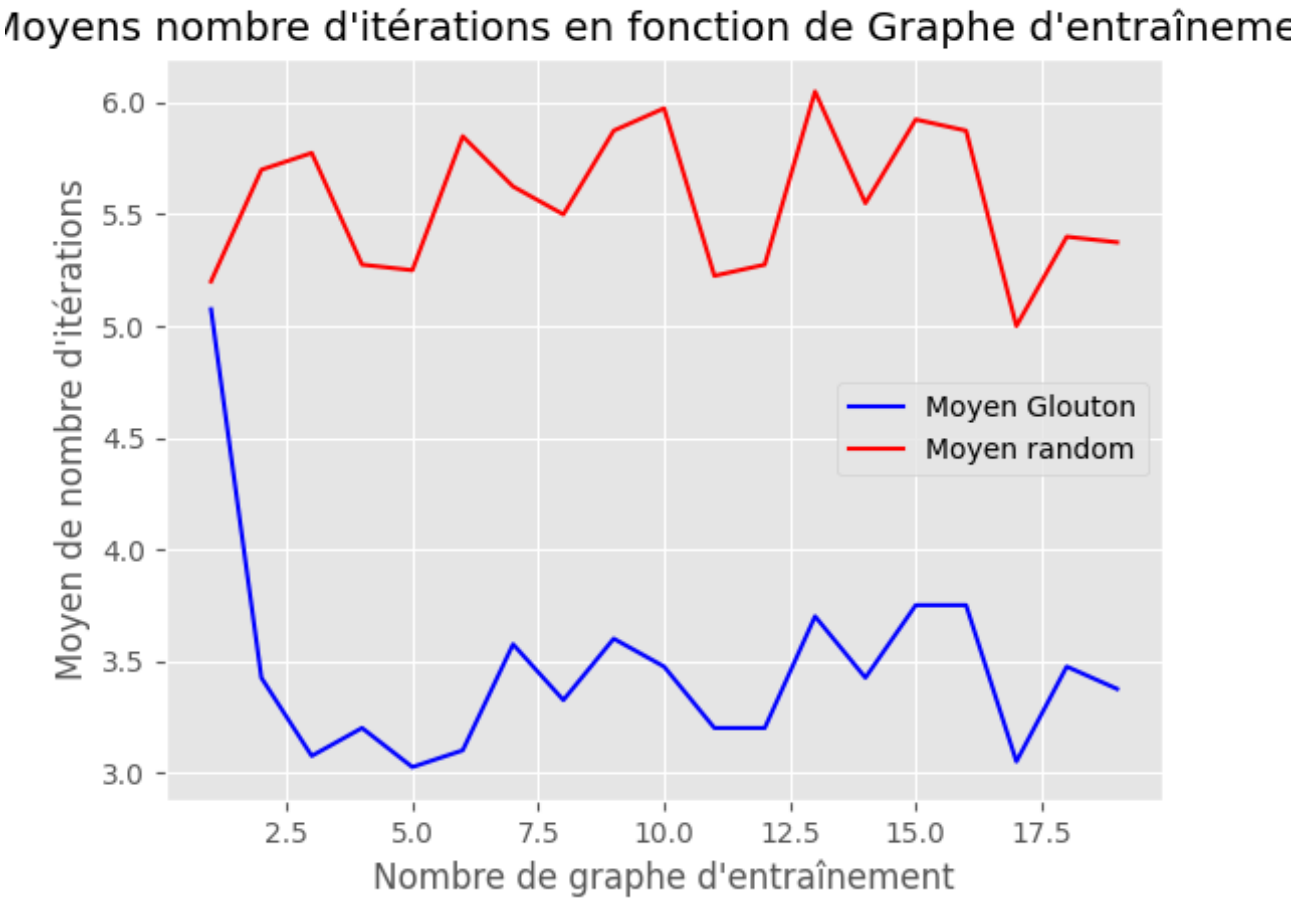


Figure 4: Comparaison des moyennes du nombre d'itérations en fonction du nombre de graphes d'entraînement. $NS = 30$, $NA = 75$, $NR = 40$

**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**

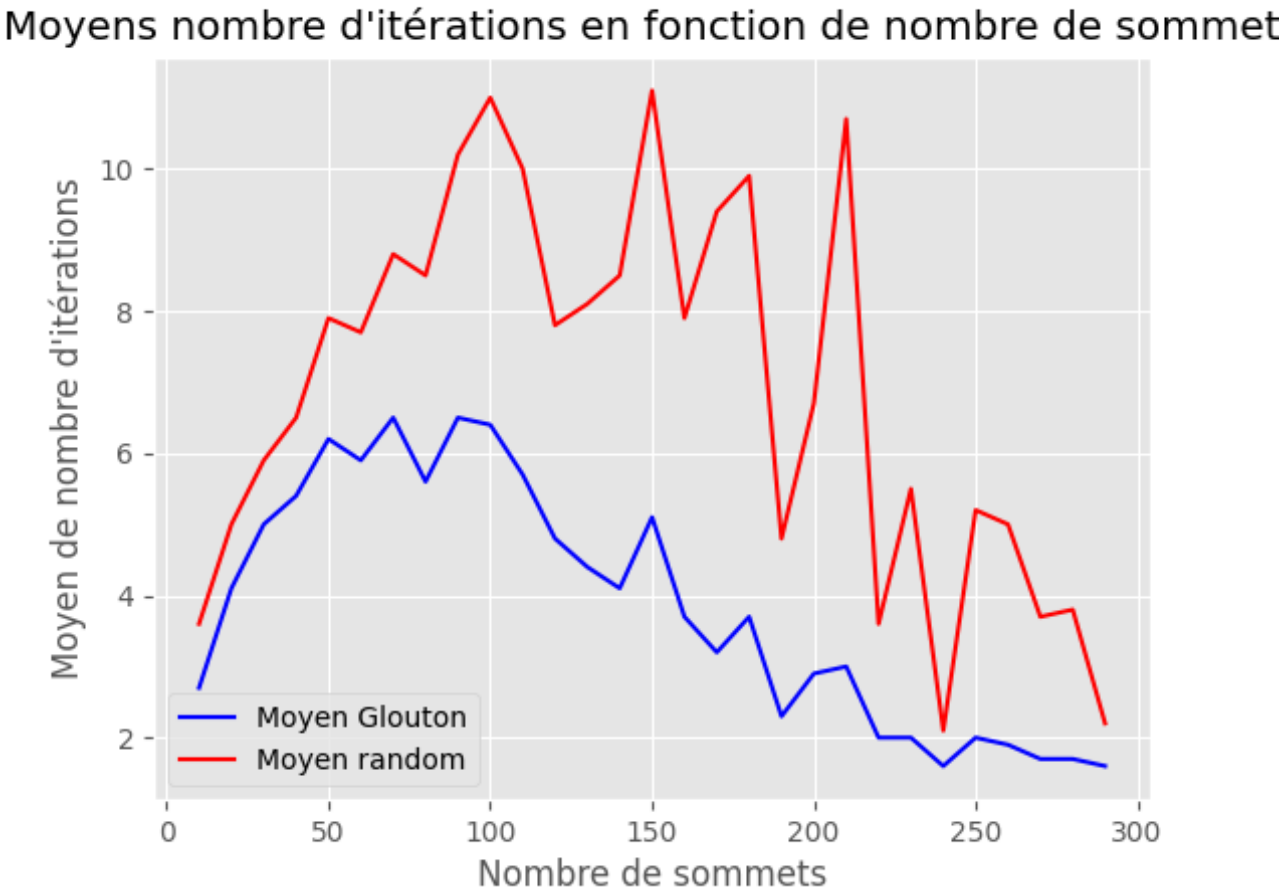


Figure 5: Comparaison de moyennes de nombre d'itérations en fonction de nombre de Sommets. $NA = 300$, $NA = 3$, $NR = 10$

**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**

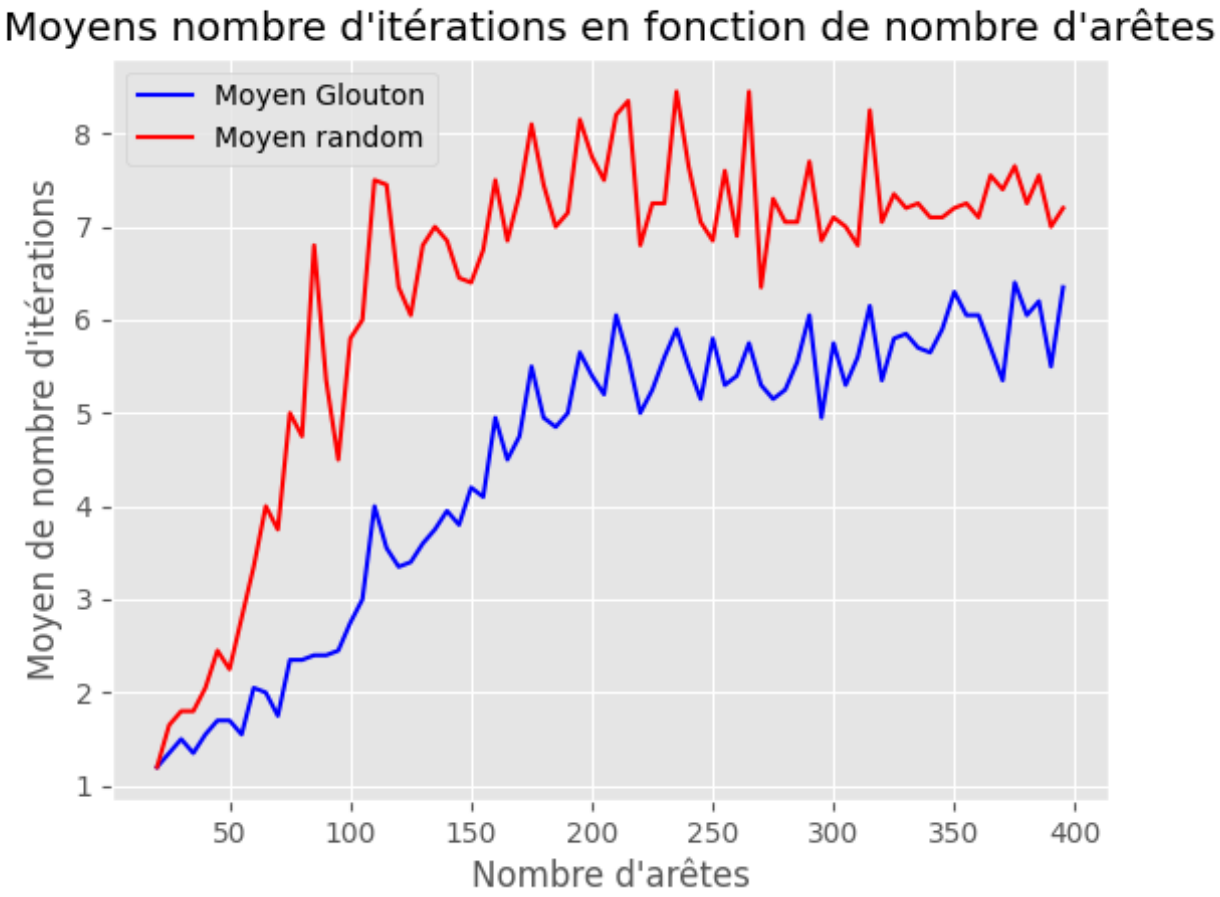


Figure 6: Comparaison des moyennes du nombre d'itérations en fonction du nombre d'arêtes. $NS = 50$, $NA = 3$, $NR = 20$

**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, 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**

```
@staticmethod
def generate_level_graph(nb_level):
    """
        Cette fonction crée un graph avec des niveau comme expliqué dans la question 11
        :param nb_level: nombre de niveau
        :level_graph: La fonction générée de la structure expliqué dans la question 11
    """
    level_graph = Graph([i for i in range(nb_level * 4)])

    edges = []
    for i in range(nb_level - 1):
        next_level_vertexes = [(i * 4) + 4 + k for k in range(4)]
```

```
            for j in range(4):
                for k in next_level_vertexes:
                    edges.append(((i * 4) + j, k, 1))
        level_graph.add_edges(edges)
        level_graph = Graph.generate_random_weights(level_graph)

        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

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

# 4 Annexes

### Fonction *get_sources*()

```
def get_sources(self):
    """
        Méthode qui calcule les sources dans l'attribut graphe i.e. les sommet qui n'ont pas de
        ↪  précedents mais des voisins
        :return: Une liste de sources
    """
    sources = []
    for vertex in self.graph.keys():
        if len(self.get_precedent(vertex)) <= 0:
            sources.append(vertex)

    return sources
```

### Fonction d'analyses

#### (a) Nombres d'itérations par graphes

```
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)
```

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

```
def anlayse_nb_iter_by_nb_trainGraph(nb_vertex, nb_edges, nb_repetitions, max_nb_trainGraph):
    """
    Faire une analyse de la peformence en fonction de nombre de graphe d'entraînement
    :param nb_vertex: nombre de sommets
    :param nb_edges: nombre d'arêtes
    :param nb_repetitions: nombre de fois qu'on calcule la moyenne par itérations
```

```
    :param max_nb_trainGraph: Maximum de nombre de graphe d'entrâinement
    :return: None
    """
    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)]
```

## (c) Nombres d'itérations par sommets

```
def anlayse_nb_iter_by_nb_sommets(nb_train_graph, nb_edges, nb_repetitions, max_nb_sommets,
↪  nb_pas = 10):
    """
    Faire une analyse de la peformence en fonction de nombre de graphe d'entraînement
    :param nb_train_graph: Nombre de graphe d'entraînement
    :param nb_edges: Nombre d'arêtes
    :param nb_repetitions: nombre de fois qu'on calcule la moyenne par itérations
    :param max_nb_sommets: Maximum de nombre de sommet
    :param nb_pas: la difference entre deux nb sommets
    :return: None
    """
    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)]
```

## (d) Nombres d'itérations par arcs

```
def anlayse_nb_iter_by_nb_edges(nb_train_graph, nb_sommet, nb_repetitions, max_nb_edges, nb_pas =
↪  5):
    """
    :param nb_train_graph: Nombre de graphe d'entraînement
    :param nb_sommet: Nombre de sommet
    :param nb_repetitions: nombre de fois qu'on calcule la moyenne par itérations
    :param max_nb_edges: Maximum de nombre d'arêtes
    :param nb_pas: la difference entre deux nombre d'arêtes.
    :return:
    """
    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)]
```