Université Iba Der THIAM de THIES
UFR Sciences et Technologies
Département Informatique

Algorithme des Graphes

Master 1 Informatique Tronc Commun

Rapport de Projet

Implémentation de l'algorithme de Bellman-Ford : Gérer les graphes avec des poids négatifs.

Exposants :

- ⇨ **Cheikh Mbacké COLY**
- ⇨ **Mouhamet DIAGNE**
- ⇨ **Mamadou MBAYE**

Professeur :

Bôa Djavph YOGANGUINA

Table des matières

1. Introduction

L'algorithme de Bellman-Ford est un algorithme de recherche de plus courts chemins à source unique dans un graphe pondéré. Il est capable de gérer des poids d'arêtes négatifs, ce qui le distingue de l'algorithme de Dijkstra. En revanche, il est moins efficace que ce dernier sur les graphes où tous les poids sont positifs.

L'algorithme a été développé indépendamment par Richard Bellman et Lester Ford. Il est largement utilisé dans le domaine des réseaux et de l'optimisation combinatoire.

2. Principe de fonctionnement

L'algorithme de Bellman-Ford suit les étapes suivantes :

2.1. Initialisation

- On assigne une distance infinie (∞) à tous les sommets, sauf la source qui reçoit la distance 0.
- Un tableau dist[] est utilisé pour stocker la plus courte distance trouvée jusqu'à chaque sommet.
- Un graphe dirigé est représenté sous forme de liste d'arêtes (u, v, w), où u est le sommet de départ, v est le sommet d'arrivée et w est le poids de l'arête.

Illustration :

Si nous avons un graphe avec les sommets {0, 1, 2, 3} et une source 0, la table des distances initiales est :

Sommet	Distance depuis 0
0	0
1	∞
2	∞
3	∞

2.2. Relâchement des arêtes

- On répète |V| - 1 fois (où |V| est le nombre de sommets) la mise à jour des distances.
- Pour chaque arête (u, v, w), on vérifie si on peut améliorer la distance vers v via u :

Si dist[u] + w < dist[v], alors dist[v] = dist[u] + w

Exemple : Si nous avons les arêtes suivantes :

- (0 → 1, poids = 4)
- (0 → 2, poids = 3)
- (1 → 2, poids = -2)
- (2 → 3, poids = 2)

L'algorithme met à jour les distances comme suit :

Itération	dist[0]	dist[1]	dist[2]	dist[3]
Initial	0	∞	∞	∞
1ère	0	4	3	∞
2ème	0	1	3	5
3ème	0	1	3	5

2.3. Détection des cycles négatifs

- Après la relaxation, on effectue une itération supplémentaire pour voir si une mise à jour est encore possible.
- Si dist[u] + w < dist[v] est encore vrai pour une arête (u, v, w), cela signifie qu'il existe un cycle de poids négatif dans le graphe.
- Dans ce cas, l'algorithme signale la présence d'un cycle.

Exemple de cycle négatif : Si nous ajoutons une arête (3 → 1, poids = -6), cela forme un cycle négatif. L'algorithme détecte alors une incohérence et affiche un avertissement : Le graphe contient un cycle de poids négatif

3. Complexité de l'algorithme

L'algorithme de Bellman-Ford a une complexité de O(VE) (où V est le nombre de sommets et E le nombre d'arêtes). Il est donc plus lent que Dijkstra (O(E log V) avec un tas de Fibonacci), mais il est plus flexible car il fonctionne même avec des poids négatifs.

4. Explication de l'implémentation

```python
import networkx as nx
import matplotlib.pyplot as plt
```

- **import networkx as nx :** Importe la bibliothèque networkx pour manipuler et analyser des graphes. Elle est renommée nx pour une utilisation simplifiée.
- **import matplotlib.pyplot as plt** : Importe la bibliothèque matplotlib.pyplot pour créer des visualisations graphiques. Elle est renommée plt pour une utilisation simplifiée.

➢ **Fonction dessiner_graphe()**

```python
def dessiner_graphe(graphe, positions, distances, aretes_surlignees=[]):
```

- **Définition de la fonction :** Cette fonction dessine un graphe avec des informations supplémentaires comme les distances des nœuds et les arêtes surlignées.
- **Paramètres :**
    - **graphe :** Le graphe à dessiner (objet networkx.Graph ou networkx.DiGraph).
    - **positions :** Un dictionnaire des positions des nœuds pour le dessin.
    - **distances :** Un dictionnaire des distances des nœuds par rapport à la source.

- **aretes_surlignees** : Une liste des arêtes à surligner (par défaut, une liste vide).

➢ **Initialisation de la figure**

Crée une nouvelle figure pour le dessin avec une taille de 8x6 pouces ***figsize***=(8, 6). ⟷

```python
plt.figure(figsize=(8, 6))
```

➢ **Récupération des arêtes et de leurs poids**

```python
aretes = graphe.edges(data=True)
labels_aretes = {(u, v): d['weight'] for u, v, d in aretes}
```

- **aretes :** Récupère toutes les arêtes du graphe avec leurs attributs (ici, le poids).
- **labels_aretes :** Crée un dictionnaire des labels des arêtes, où chaque clé est une paire de nœuds (u, v) et la valeur est le poids de l'arête.

➢ **Couleurs des nœuds**

```python
couleurs_noeuds = ['lightgreen' if distances[n] < float('inf') else 'lightblue' for n in graphe.nodes]
```

- **Attribue une couleur à chaque nœud :**
  - **lightgreen** si la distance du nœud est inférieure à l'infini (c'est-à-dire que le nœud a été atteint).
  - **lightblue** sinon.

➢ **Dessin du graphe**

```python
nx.draw(graphe, positions, with_labels=True, node_color=couleurs_noeuds, edge_color='gray', node_size=1000, font_size=12)
```

- Dessine le graphe avec les positions spécifiées (**positions**), les étiquettes des nœuds activées (**with_labels=True**), les couleurs des nœuds (**couleurs_noeuds**), les arêtes en gris (**edge_color='gray'**), une taille de nœud de 1000 (**node_size=1000**) et une taille de police de 12 (**font_size=12**).

➢ **Ajout des labels des arêtes**

```python
nx.draw_networkx_edge_labels(graphe, positions, edge_labels=labels_aretes, font_size=10)
```

➢ **Mise en évidence des arêtes**

```python
    if aretes_surlignees:
        nx.draw_networkx_edges(graphe, positions, edgelist=aretes_surlignees, edge_color='r', wi
dth=2)
```

- Si des arêtes sont spécifiées dans **aretes_surlignees**, elles sont dessinées en rouge (**edge_color='r'**) avec une épaisseur de 2 (**width=2**).

➢ **Ajout des distances près des nœuds**

```python
    positions_decalées = {n: (x, y + 0.08) for n, (x, y) in positions.items()}
    labels_distances = {n: f"{distances[n]}" if distances[n] < float('inf') else "" for n in gra
phe.nodes}
    nx.draw_networkx_labels(graphe, positions_decalées, labels=labels_distances, font_color='bla
ck', font_size=10, font_weight='bold', bbox=dict(facecolor='white', edgecolor='none', alpha=0.
7))
```

- **positions_decalées :** Déplace légèrement les positions des labels des distances vers le haut pour éviter la superposition avec les étiquettes des nœuds.
- **labels_distances :** Crée un dictionnaire des labels de distance pour chaque nœud. Si la distance est infinie, le label est vide.
- **nx.draw_networkx_labels :** Dessine les labels de distance sur le graphe.

➢ **Ajout des ∞ pour les distances infinies**

```python
    labels_infini = {n: "∞" for n in graphe.nodes if distances[n] == float('inf')}
    positions_infini = {n: (x, y + 0.15) for n, (x, y) in positions.items()}
    nx.draw_networkx_labels(graphe, positions_infini, labels=labels_infini, font_color='red', fo
nt_size=12, font_weight='bold', bbox=dict(facecolor='white', edgecolor='none', alpha=0.7))
```

- **labels_infini :** Crée un dictionnaire des labels "∞" pour les nœuds dont la distance est infinie.
- **positions_infini :** Déplace encore plus haut les positions des labels "∞".
- **nx.draw_networkx_labels :** Dessine les labels "∞" sur le graphe.

➢ **Affichage du graphe**

```python
    plt.title("Visualisation de l'algorithme de Bellman-Ford")
    plt.show()
```

- **Ajoute un titre à la figure et l'affiche.**

➢ Fonction bellman_ford()

```python
def bellman_ford(graphe, source):
```

- **Définition de la fonction** : Cette fonction applique l'algorithme de Bellman-Ford pour trouver les plus courts chemins depuis un nœud source.
- **Paramètres** :
  - **graphe** : Une liste d'arêtes pondérées sous forme de tuples (u, v, poids).
  - **source** : Le nœud source.

➢ Création du graphe oriente

```python
    G = nx.DiGraph()
    G.add_weighted_edges_from(graphe)
```

- **G = nx.DiGraph()** : Crée un graphe orienté.
- **G.add_weighted_edges_from(graphe)** : Ajoute les arêtes pondérées au graphe.

➢ Calcul des positions du nœuds

```python
    positions = nx.spring_layout(G, k=3)
```

- Calcule les positions des nœuds pour le dessin en utilisant l'algorithme de disposition **spring_layout**. Le paramètre **k=3** augmente l'espacement entre les nœuds.

➢ Initialisation des distances

```python
    noeuds = list(G.nodes)
    distances = {noeud: float('inf') for noeud in noeuds}
    distances[source] = 0
```

- **noeuds** : Liste des nœuds du graphe.
- **distances** : Dictionnaire des distances initialisées à l'infini, sauf pour la source qui est à 0.

➢ Dessin du graphe initial

```python
    dessiner_graphe(G, positions, distances)
```

- Appelle la fonction **dessiner_graphe()** pour afficher le graphe initial.

➢ **Application de l'algorithme de Bellman-Ford**

```python
    for _ in range(len(noeuds) - 1):
        for u, v, data in G.edges(data=True):
            poids = data['weight']
            if distances[u] + poids < distances[v]:
                distances[v] = distances[u] + poids
                dessiner_graphe(G, positions, distances, aretes_surlignees=[(u, v)])
```

- **Boucle principale :** Répète l'algorithme ***len(noeuds) - 1*** fois.
- **Relaxation des arêtes :** Pour chaque arête (u, v), si la distance de u plus le poids de l'arête est inférieur à la distance actuelle de v, met à jour la distance de v.
- **Dessin du graphe :** Affiche le graphe après chaque mise à jour, en surlignant l'arête relaxée.

➢ **Vérification des cycles de poids négatifs**

```python
    for u, v, data in G.edges(data=True):
        if distances[u] + data['weight'] < distances[v]:
            print("Le graphe contient un cycle de poids négatif")
            return None
```

- Vérifie la présence de cycles de poids négatifs. Si une relaxation supplémentaire est possible, le graphe contient un **cycle**.

➢ Retour des distances finales

```python
    return distances
```

✓ Retourne le dictionnaire des distances finales.

➢ **Exemple d'utilisation**

```python
graphe = [(0, 1, 4), (0, 2, 3), (1, 2, -2), (2, 3, 2), (3, 1, -6)]
source = 0
distances = bellman_ford(graphe, source)
if distances:
    print("Distances finales:", distances)
```

- **graphe :** Définit un graphe avec des arêtes pondérées.
- **source :** Définit le nœud source.
- **distances :** Appelle la fonction bellman_ford et affiche les distances finales si aucun cycle de poids négatif n'est détecté.

💡 **Résultat :**

Ce code applique l'algorithme de Bellman-Ford et visualise chaque étape de manière interactive. La documentation ligne par ligne permet de comprendre chaque partie du code en détail.

5. Illustration du code Source

Pour plus de clarté, tout le projet et son code source sont déployés sur GitHub : vous pouvez simplement cloner le projet et essayer de faire les tests.

🔗 **Lien du projet** : https://github.com/CMCode2001/Implementation-Algorithme-Bellman-Ford-sur-Python

💡 **Image Code Source**

bellman-Ford.py > bellman_ford

```python
import networkx as nx
import matplotlib.pyplot as plt

def dessiner_graphe(graphe, positions, distances, aretes_surlignees=[]):
    """
    Dessine le graphe avec les distances des nœuds et les arêtes surlignées.

    :param graphe: Le graphe à dessiner.
    :param positions: Un dictionnaire des positions des nœuds.
    :param distances: Un dictionnaire des distances des nœuds par rapport à la source.
    :param aretes_surlignees: Une liste des arêtes à surligner (par défaut, vide).
    """
    plt.figure(figsize=(8, 6))

    # Récupérer les arêtes et leurs poids
    aretes = graphe.edges(data=True)
    labels_aretes = {(u, v): d['weight'] for u, v, d in aretes}

    # Définir les couleurs des nœuds en fonction de leur distance
    couleurs_noeuds = ['lightgreen' if distances[n] < float('inf') else 'lightblue' for n in graphe.nodes]

    # Dessiner le graphe
    nx.draw(graphe, positions, with_labels=True, node_color=couleurs_noeuds, edge_color='gray', node_size=1000, font_size=12)
    nx.draw_networkx_edge_labels(graphe, positions, edge_labels=labels_aretes, font_size=10)

```

```
    # Surligner les arêtes spécifiées
    if aretes_surlignees:
        nx.draw_networkx_edges(graphe, positions, edgelist=aretes_surlignees, edge_color='r', width=2)

    # Ajouter les distances près des nœuds
    positions_decalées = {n: (x, y + 0.08) for n, (x, y) in positions.items()}  # Décalage vers le haut
    labels_distances = {n: f"{distances[n]}" if distances[n] < float('inf') else "" for n in graphe.nodes}
    nx.draw_networkx_labels(graphe, positions_decalées, labels=labels_distances, font_color='black', font_size=10, font_weight='bold',
    bbox=dict(facecolor='white', edgecolor='none', alpha=0.7))

    # Ajouter les ∞ pour les distances infinies
    labels_infini = {n: "∞" for n in graphe.nodes if distances[n] == float('inf')}
    positions_infini = {n: (x, y + 0.15) for n, (x, y) in positions.items()}  # Décalage plus haut
    nx.draw_networkx_labels(graphe, positions_infini, labels=labels_infini, font_color='red', font_size=12, font_weight='bold',
    bbox=dict(facecolor='white', edgecolor='none', alpha=0.7))

    plt.title("Visualisation de l'algorithme de Bellman-Ford")
    plt.show()

def bellman_ford(graphe, source):
    """
    Applique l'algorithme de Bellman-Ford pour trouver les plus courts chemins depuis un nœud source.

    :param graphe: Une liste d'arêtes pondérées sous forme de tuples (u, v, poids).
    :param source: Le nœud source.
    :return: Un dictionnaire des distances des nœuds par rapport à la source.
    """
    # Créer un graphe orienté
    G = nx.DiGraph()
    G.add_weighted_edges_from(graphe)
```

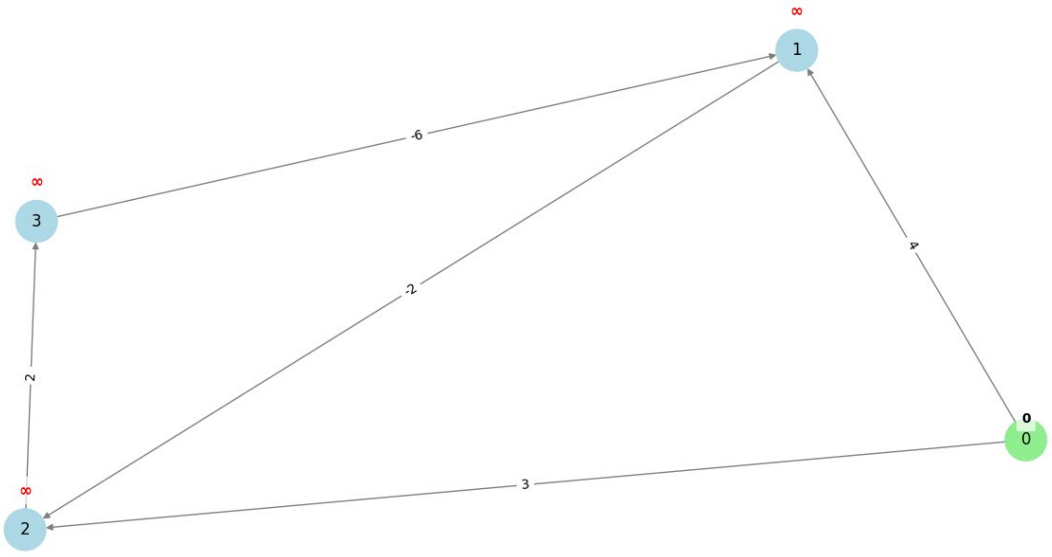
```
    # Calculer les positions des nœuds pour le dessin
    positions = nx.spring_layout(G, k=3)  # Augmentation de l'espacement entre les nœuds

    # Initialiser les distances
    noeuds = list(G.nodes)
    distances = {noeud: float('inf') for noeud in noeuds}
    distances[source] = 0

    # Dessiner le graphe initial
    dessiner_graphe(G, positions, distances)

    # Appliquer l'algorithme de Bellman-Ford
    for _ in range(len(noeuds) - 1):
        for u, v, data in G.edges(data=True):
            poids = data['weight']
            if distances[u] + poids < distances[v]:
                distances[v] = distances[u] + poids
                dessiner_graphe(G, positions, distances, aretes_surlignees=[(u, v)])

    # Vérifier la présence de cycles de poids négatifs
    for u, v, data in G.edges(data=True):
        if distances[u] + data['weight'] < distances[v]:
            print("Le graphe contient un cycle de poids négatif")
            return None

    return distances

# Exemple d'utilisation
graphe = [(0, 1, 4), (0, 2, 3), (1, 2, -2), (2, 3, 2), (3, 1, -6)]
source = 0
distances = bellman_ford(graphe, source)
if distances:
    print("Distances finales:", distances)
```

6. Résultat Graphique Obtenu

6.1. Graphe initial

📌 Affichage du graphe avant toute mise à jour des distances

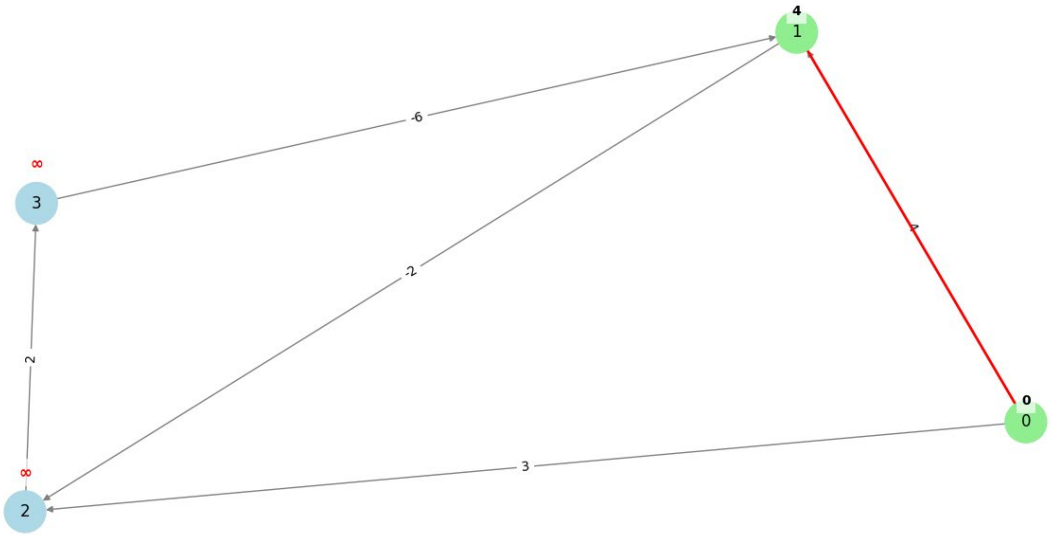
- Tous les sommets sont affichés.
- Les poids des arêtes sont indiqués sur les connexions entre les sommets.
- Les sommets sont en bleu, sauf le sommet source qui est déjà à 0 (en vert).
- Tous les autres sommets ont une distance initiale de ∞ affichée en rouge au-dessus d'eux

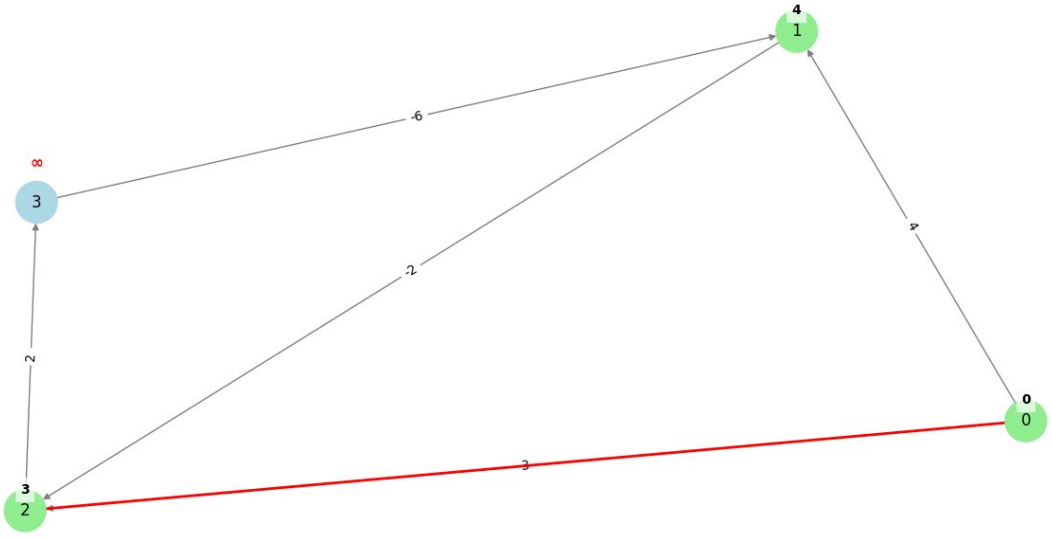


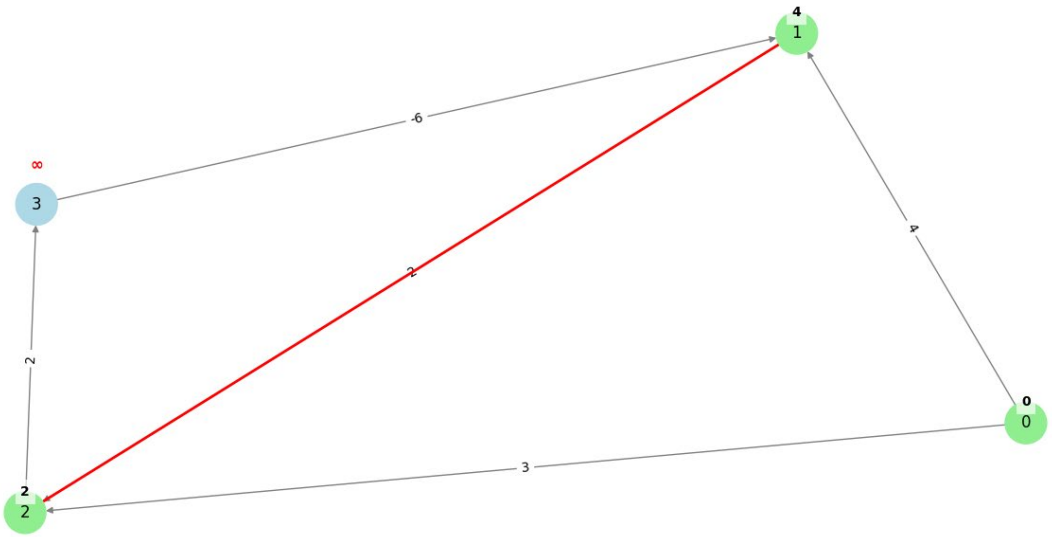
6.2. Étapes de relaxation des arêtes (répétées |V| - 1 fois)

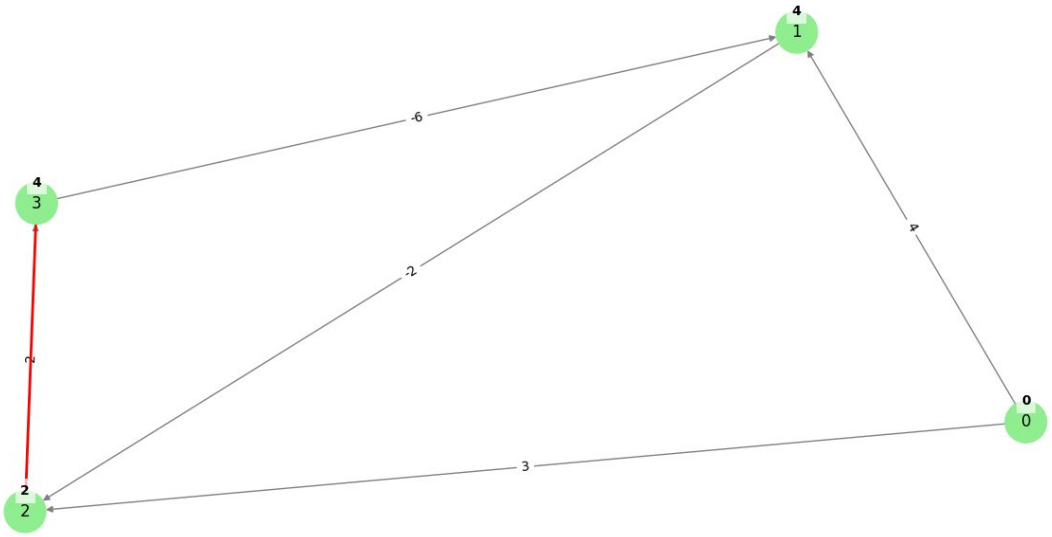
📌 À chaque mise à jour d'une distance, un nouveau graphe est affiché avec une arête mise en surbrillance

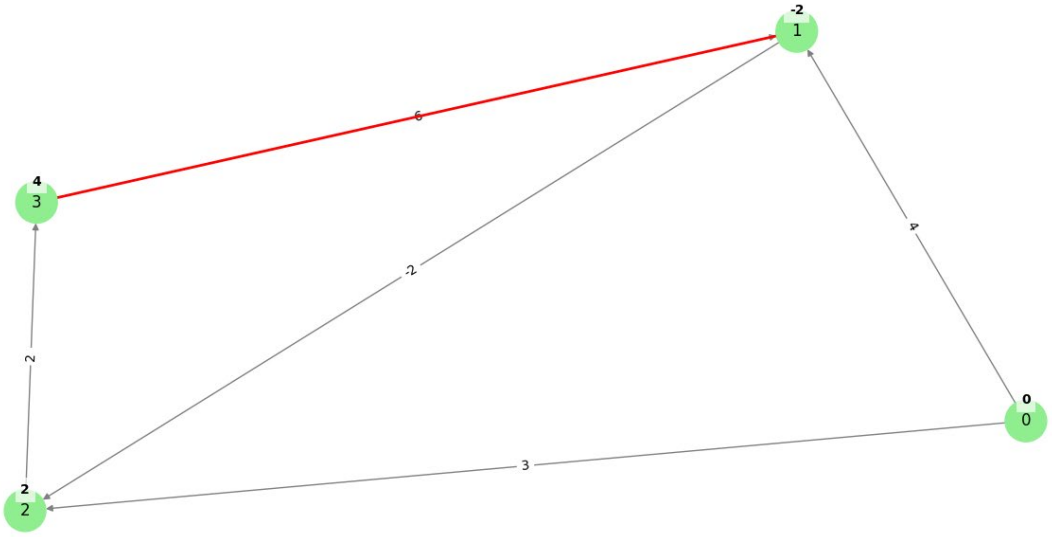
- Une arête est colorée en rouge pour montrer quelle connexion est mise à jour.
- La distance du sommet de destination est mise à jour et affichée en noir avec un fond blanc pour une meilleure lisibilité.
- Les sommets qui ont reçu une mise à jour sont colorés en vert.
- On répète cette étape jusqu'à ce qu'il n'y ait plus de mise à jour.






4
1
-6
∞
3
4
2
2
0
0
3
2

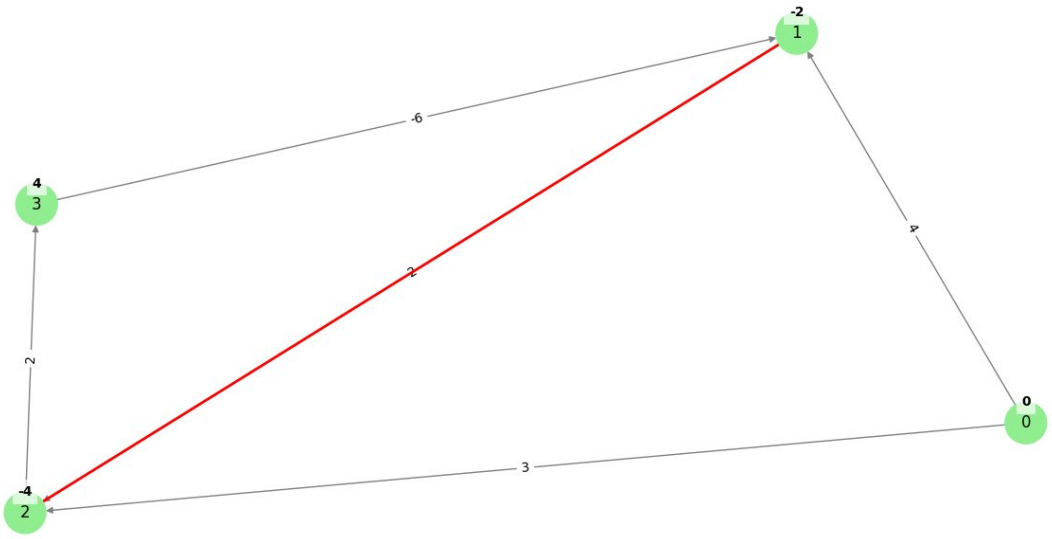

4
1
-6
4
3
4
2
2
0
0
3
2

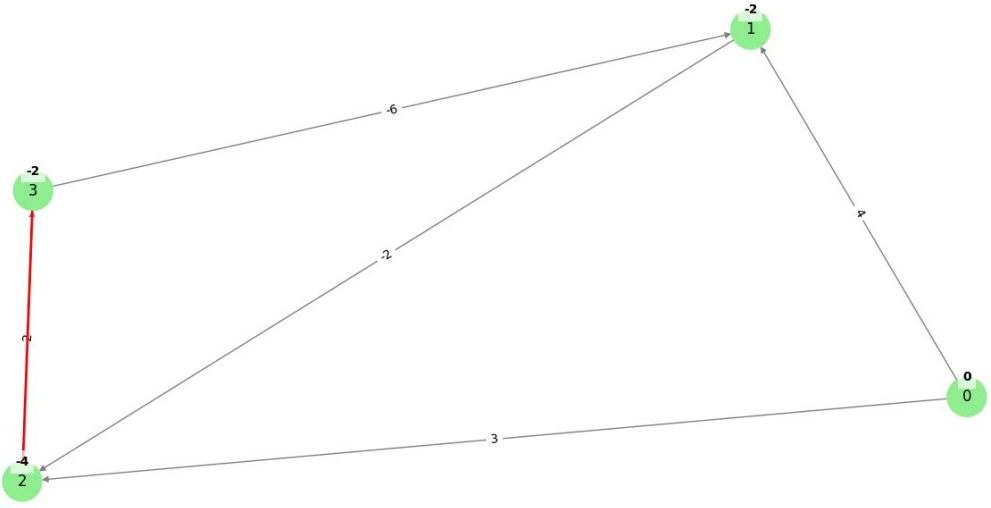

-2
1
6
4
3
4
2
2
0
0
3
2

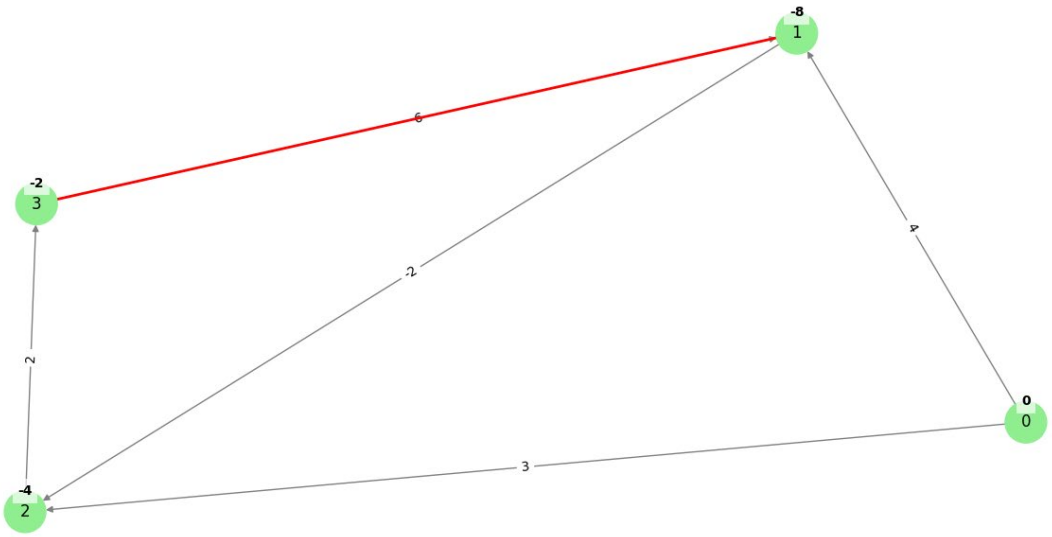
6.3. Détection des cycles négatifs (si présents)

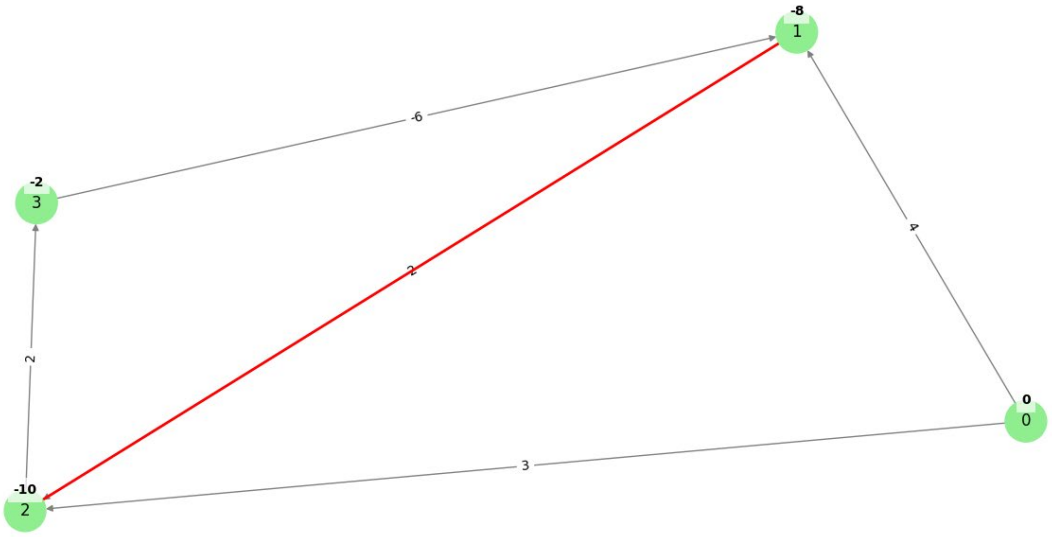
📌 **Dernière vérification pour voir si une mise à jour est encore possible**

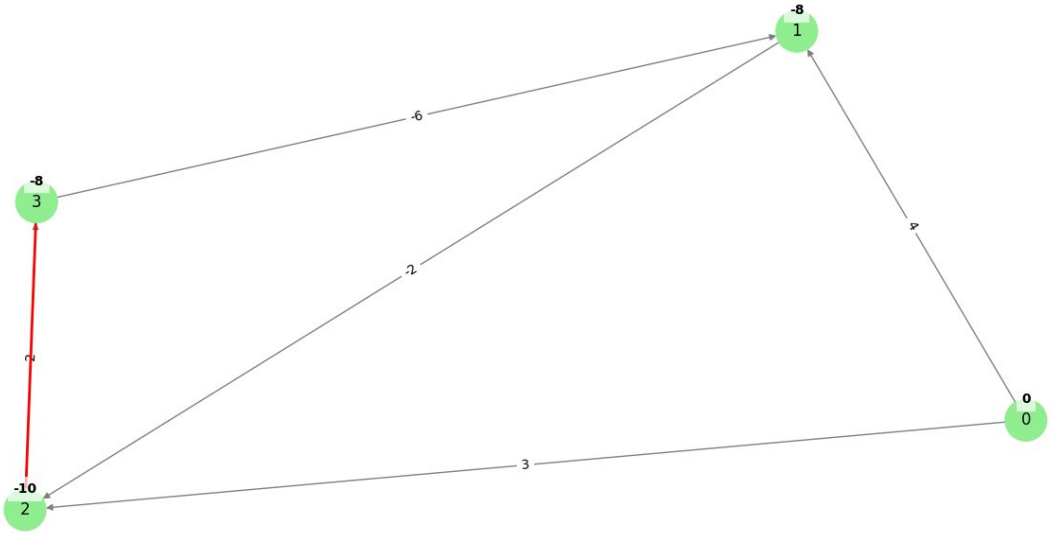
- Si une mise à jour est encore possible après |V| - 1 itérations, cela signifie qu'il existe un **cycle de poids négatif**.
- **Aucun graphe supplémentaire n'est affiché**, mais un message "Le graphe contient un cycle de poids négatif" s'affiche dans la console.
- Si aucun cycle négatif n'est trouvé, le programme affiche les **distances finales** depuis la source.






-8
1
6
-2
3
4
-2
2
0
0
3
-4
2

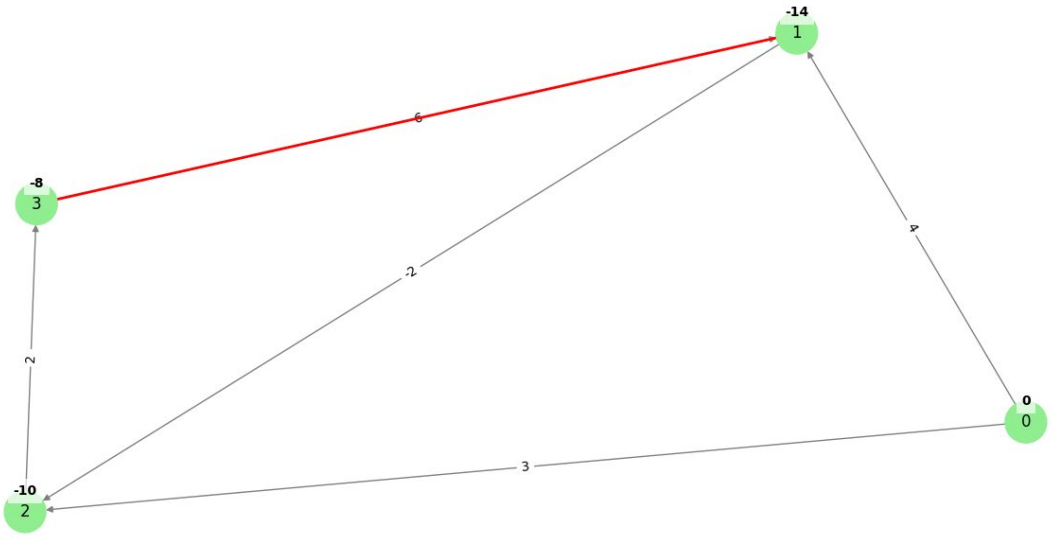

-8
1
-6
-2
3
4
2
2
0
0
3
-10
2


-8
1
-6
-8
3
4
-2
2
0
0
3
-10
2

6.4. Graphe final (distances minimales trouvées)

📌 **Affichage du graphe avec les distances les plus courtes depuis la source**

- Tous les sommets affichent maintenant leur **distance minimale** trouvée.
- Les sommets qui n'ont pas été atteints (si existants) restent marqués avec ∞.
- **Les distances affichées sont désormais stabilisées**, et aucun autre graphe n'est généré après cette étape.



7. Conclusion

L'algorithme de Bellman-Ford est un outil puissant pour résoudre les problèmes de plus courts chemins, en particulier lorsqu'il existe des poids négatifs. Son implémentation est relativement simple mais peut être coûteuse en termes de complexité. Son avantage principal reste sa capacité à détecter les cycles de poids négatif, ce qui le rend indispensable dans de nombreuses applications, notamment dans le routage réseau et l'analyse de graphes.