Rapport Mini Projet :
Graph Algorithms Online

Filière : Cycle d'Ingénieure en Génie Informatique s2

Dirigé par : EZ-ZINE HAMZA & DEBBAGH AYOUB & GRICH ALI

Resp. : Pr. Youssef QARAAI

Module : Recherche Opérationnelle

Faculté des sciences et techniques d'Errachidia

Université Moulay Ismaïl

2022/2023

Table de matieres

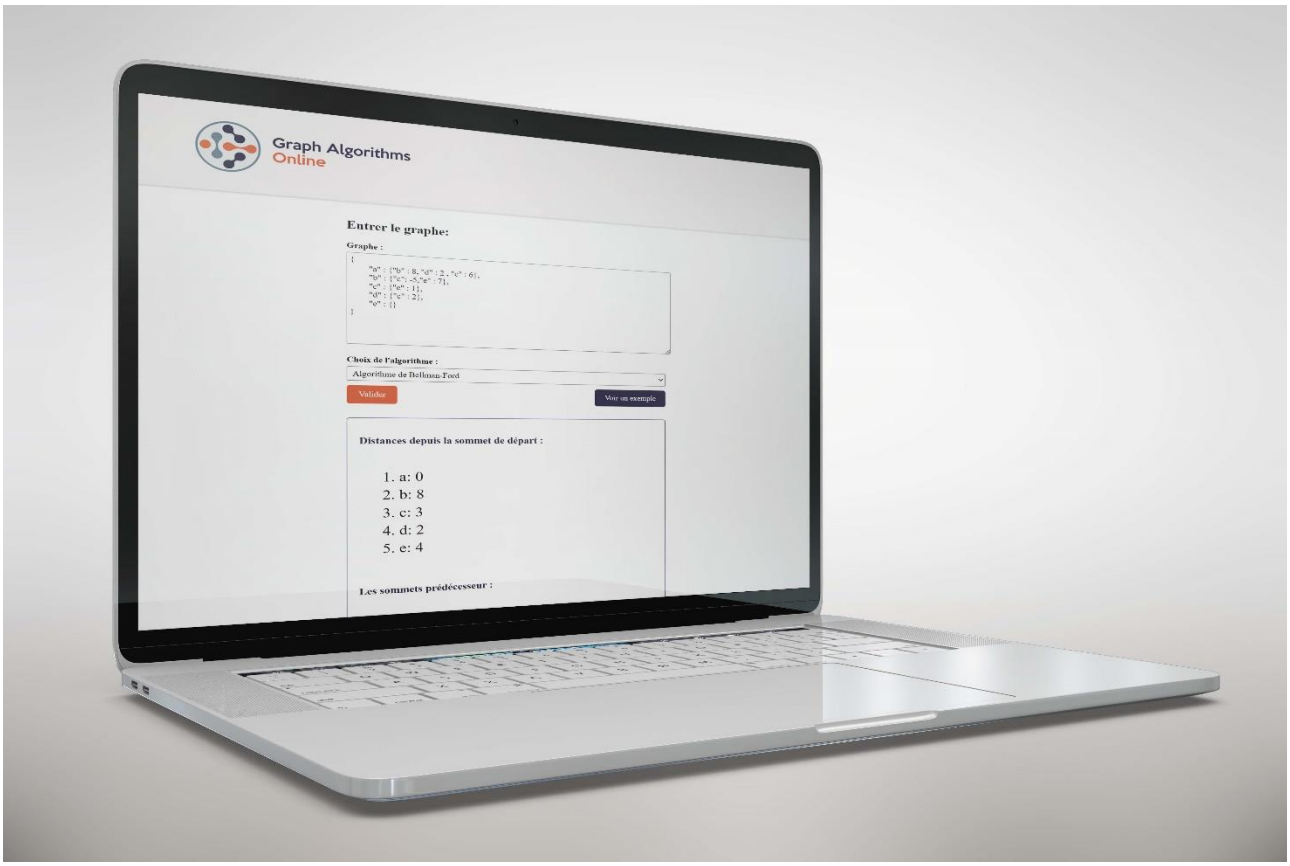
1. Introduction

Le domaine des graphes et de leurs algorithmes joue un rôle essentiel dans de nombreux domaines tels que la science des données, la recherche opérationnelle et la théorie des réseaux. Les graphes sont des structures de données puissantes pour représenter des relations entre des objets, et les algorithmes qui y sont associés permettent d'analyser, d'explorer et d'optimiser ces relations. Aujourd'hui, avec l'avènement du web et de la technologie, les ressources en ligne fournissent un accès rapide et pratique à des outils et des plateformes pour explorer et exécuter ces algorithmes.

Le site web "Graph Algorithms Online" a été conçu dans le but de faciliter l'apprentissage et l'expérimentation des algorithmes de graphes en fournissant une interface conviviale et des outils puissants. Il offre une collection complète d'algorithmes populaires, allant des algorithmes de parcours de graphes aux algorithmes de recherche de chemins les plus courts et d'arbres couvrants minimaux. Les utilisateurs ont la possibilité d'interagir avec ces algorithmes en fournissant leurs propres graphes personnalisés ou en utilisant des exemples prédéfinis pour comprendre leur fonctionnement et évaluer leurs performances.

Dans ce rapport, nous nous plongerons en détail 8 populaires algorithmes clés proposés par "Graph Algorithms Online". Nous explorerons leur fonctionnement et les etapes de chaque algorithme temporelle pour mieux comprendre leur efficacité et leur applicabilité dans divers scénarios. Nous étudierons également exemple pour chaque algorithme et leur resultat

Notre objectif est d'offrir une vision approfondie de ces algorithmes et de démontrer la valeur ajoutée que le site web "Graph Algorithms Online" apporte à ceux qui souhaitent explorer et comprendre ce domaine fascinant. Nous présenterons une analyse détaillée des algorithmes sélectionnés,ainsi que des exemples pratiques pour illustrer leur mise en œuvre.



Notre mini projet est une application web créée par les langages de programmation : HTML, CSS et JAVASCRIPT

Lorsque vous ouvrez la page, vous verrez un titre "Entrer le graphe" et un espace de texte où vous pouvez saisir les détails du graphe. Vous devez entrer le graphe au format JSON.

Juste en dessous, il y a une liste déroulante intitulée "Choix de l'algorithme". Vous pouvez sélectionner l'algorithme que vous souhaitez appliquer au graphe.

Une fois que vous avez saisi le graphe et sélectionné l'algorithme, vous pouvez cliquer sur le bouton "Valider" pour traiter le graphe avec l'algorithme sélectionné.

Si vous souhaitez voir un exemple de graphe, vous pouvez cliquer sur le bouton "Voir un exemple". Cela affichera une fenêtre contextuelle avec une image représentant un exemple de graphe.

Les résultats de l'algorithme appliqué au graphe seront affichés dans la zone de résultat située en dessous du formulaire.

2. Parcours et coloration de graphes

2.1 Parcous

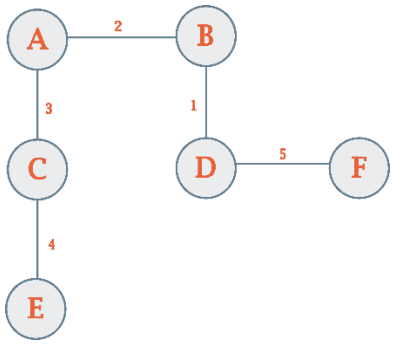
2.1.1 Parcours en largeur

les étapes de la fonction **parcoursLargeur(G, d)**:

1. La fonction crée une file de sommets à traiter appelée fileSommets, initialement vide. Elle crée également un tableau visiter contenant le sommet de départ d.
2. La boucle while s'exécute tant que la file visiter n'est pas vide. Cela signifie qu'il reste des sommets à visiter.
3. À chaque itération de la boucle, on extrait le premier sommet s de la file visiter à l'aide de la méthode shift().
4. Si le sommet s est déjà présent dans la file fileSommets, cela signifie qu'il a déjà été visité, donc on passe à l'itération suivante de la boucle avec continue.
5. Sinon, on ajoute le sommet s à la file fileSommets, ce qui signifie qu'il a été visité.
6. On récupère les voisins du sommet s dans le graphe G et on les stocke dans la variable voisins.
7. Si des voisins existent (c'est-à-dire si voisins n'est pas null ou undefined), on itère sur les voisins pour les ajouter à la file visiter.
    - Si les voisins sont représentés sous forme d'un tableau (Array.isArray(voisins) renvoie true), on les ajoute un par un à la file visiter, à condition qu'ils ne soient pas déjà présents dans la file fileSommets ou dans la file visiter.
    -
    - Si les voisins sont représentés sous forme d'un objet, on itère sur les propriétés de l'objet et on ajoute les voisins à la file visiter, à condition qu'ils ne soient pas déjà présents dans la file fileSommets ou dans la file visiter.
8. Une fois que tous les voisins de s ont été traités, la boucle while continue jusqu'à ce que la file visiter soit vide.
9. Enfin, la fonction renvoie la file fileSommets, qui contient tous les sommets visités dans l'ordre du parcours en largeur.

Exemple :

```
{
 "A": {"B": 2, "C": 3},
 "B": {"D": 1},
 "C": {"E": 4},
 "D": {"F": 5},
 "E": {},
 "F": {}
}
```



1. FileSommets: ["A"] (étape 1)
   Sommets visités: []

2. FileSommets: [] (étape 2, "A" retiré de la file)
   Sommets visités: ["A"]
   Voisins de "A": {"B": 2, "C": 3}
   Ajout de "B" et "C" à la fileSommets.

3. FileSommets: ["B", "C"] (étape 2)
   Sommets visités: ["A"]
   Voisins de "B": {"D": 1}
   Ajout de "D" à la fileSommets.

4. FileSommets: ["C", "D"] (étape 2)
   Sommets visités: ["A"]
   Voisins de "C": {"E": 4}
   Ajout de "E" à la fileSommets.

5. FileSommets: ["D", "E"] (étape 2)
   Sommets visités: ["A"]
   Voisins de "D": {"F": 5}
   Ajout de "F" à la fileSommets.

6. FileSommets: ["E", "F"] (étape 2)
   Sommets visités: ["A"]
   Voisins de "E": {}
   Aucun voisin à ajouter à la fileSommets.

7. FileSommets: ["F"] (étape 2)
   Sommets visités: ["A"]
   Voisins de "F": {}
   Aucun voisin à ajouter à la fileSommets.

8. FileSommets: [] (étape 3, la fileSommets est vide)
   Sommets visités: ["A", "B", "C", "D", "E", "F"]

Le parcours en largeur à partir du sommet "A" dans le graphe donné donne les sommets visités dans l'ordre suivant: ["A", "B", "C", "D", "E", "F"].

2.1.2 Parcours en profondeur

les étapes de la fonction **parcoursProfondeur(G, d, pileSommets)**:

1. Ajouter le sommet de à la liste des sommets visités (pileSommets).
2. Récupérer les voisins du sommet d à partir du graphe (G) et les stocker dans la variable voisins.
3. Parcourir tous les voisins du sommet d.
4. Pour chaque voisin v dans la liste voisins :
    - Vérifier si le voisin v n'est pas déjà présent dans la liste des sommets visités (pileSommets).
    - Si le voisin v n'a pas été visité :
        - Appeler récursivement la fonction parcoursProfondeur avec les paramètres G, v et pileSommets.
        - Cela permettra d'explorer en profondeur à partir du voisin v.
5. Répéter les étapes jusqu'à ce que tous les voisins du sommet d aient été visités.

Cela garantit que tous les sommets accessibles depuis le sommet de départ sont visités en profondeur.

Exemple :

En utilise la même graph précédente :

1. Appel initial : parcoursProfondeur(G, "A", [])

Étape 1:

- Ajouter "A" à la liste des sommets visités : pileSommets = ["A"]
- Récupérer les voisins du sommet "A" : voisins = {"B": 2, "C": 3}
- Parcours des voisins de "A" :
- Voisin 1 : "B"
    - Vérifier si "B" n'est pas déjà dans visited (non trouvé).
    - Appel récursif : parcoursProfondeur(G, "B", pileSommets)

Étape 2:

- Ajouter "B" à la liste des sommets visités : pileSommets = ["A", "B"]
- Récupérer les voisins du sommet "B" : voisins = {"D": 1}
- Parcours des voisins de "B" :
    - Voisin 1 : "D"
    - Vérifier si "D" n'est pas déjà dans pileSommets (non trouvé).
    - Appel récursif : parcoursProfondeur(G, "D", pileSommets)

Étape 3:

- Ajouter "D" à la liste des sommets visités : pileSommets = ["A", "B", "D"]

- Récupérer les voisins du sommet "D" : voisins = {"F": 5}
- Parcours des voisins de "D" :
    - Voisin 1 : "F"
    - Vérifier si "F" n'est pas déjà dans pileSommets (non trouvé).
    - Appel récursif : parcoursProfondeur(G, "F", pileSommets)

Étape 4:

- Ajouter "F" à la liste des sommets visités : pileSommets = ["A", "B", "D", "F"]
- Aucun voisin pour "F", donc la boucle se termine.

- Fin de l'appel récursif pour "D"
- Fin de l'appel récursif pour "B"

- Voisin 2 : "C"
- Vérifier si "C" n'est pas déjà dans pileSommets (non trouvé).
- Appel récursif : parcoursProfondeur(G, "C", pileSommets)

Étape 5:

- Ajouter "C" à la liste des sommets visités :
  pileSommets = ["A", "B", "D", "F", "C"]
- Récupérer les voisins du sommet "C" : voisins = {"E": 4}
- Parcours des voisins de "C" :
    - Voisin 1 : "E"
        - Vérifier si "E" n'est pas déjà dans pileSommets (non trouvé).
        - Appel récursif : parcoursProfondeur(G, "E", pileSommets)

- Ajouter "E" à la liste des sommets visités :
  pileSommets = ["A", "B", "D", "F", "C", "E"]
- Aucun voisin pour "E", donc la boucle se termine.

- Fin de l'appel récursif pour "E"
- Fin de l'appel récursif pour "C"
- Fin de la boucle des voisins de "A"

Parcours terminé. La liste des sommets visités est : ["A", "B", "D", "F", "C", "E"]

Cela représente l'ordre dans lequel les sommets ont été visités lors du parcours en profondeur à partir du sommet "A".

Voici un résumé du parcours en profondeur du graphe donné :

- On commence par le sommet "A" et on l'ajoute à la pile des sommets visités.
- On explore les voisins de "A" dans l'ordre : "B" et "C".
- On choisit le premier voisin, "B", et on l'ajoute à la pile des sommets visités.
- On explore les voisins de "B" : "D".
- On choisit le seul voisin de "B", "D", et on l'ajoute à la pile des sommets visités.
- On explore les voisins de "D" : "F".
- On choisit le seul voisin de "D", "F", et on l'ajoute à la pile des sommets visités.
- On n'a plus de voisins à explorer à partir de "F".
- On revient en arrière et on explore le deuxième voisin de "A", "C".
- On ajoute "C" à la pile des sommets visités.
- On explore le seul voisin de "C", "E", et on l'ajoute à la pile des sommets visités.
- On n'a plus de voisins à explorer à partir de "E".

On a terminé l'exploration de tous les sommets du graphe.

Le résultat final est la pile des sommets visités : ["A", "B", "D", "F", "C", "E"].

2.2 Coloration

2.2.1 Algorithme glouton de coloriage d'un graphe

La fonction **`coloriageGlouton(graph)`** est une implémentation de l'algorithme de coloration gloutonne (greedy coloring) en JavaScript. Voici les étapes de cette fonction :

1. **`const sommets = Object.keys(graph);`:** Cette ligne extrait les sommets du graphe en utilisant la fonction **`Object.keys()`.** Les sommets sont stockés dans un tableau appelé **`sommets`**.

2. **`const resultat = {};`**: Crée un objet vide appelé **`resultat`** qui sera utilisé pour stocker les couleurs attribuées à chaque sommet.

3. La boucle **`for (const sommet of sommets)`** parcourt tous les sommets du graphe.

4. **`const attribue = new Set()`**;`: Crée un ensemble vide appelé `attribue` pour stocker les couleurs déjà attribuées aux voisins du sommet actuel.

5. La boucle `**for (const voisin in graph[sommet])**` parcourt tous les voisins du sommet actuel.

6. `**if (resultat[voisin]) { attribue.add(resultat[voisin]); }`:** Vérifie si le voisin a déjà une couleur attribuée. Si c'est le cas, la couleur du voisin est ajoutée à l'ensemble `**attribue**`.

7. `**let coleur = 1;`:** Initialise la couleur du sommet actuel à 1.

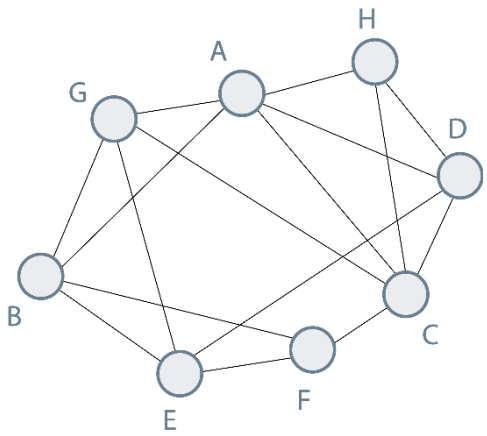
8. La boucle `**while (attribue.has(coleur)) { coleur++; }**` vérifie si la couleur actuelle est déjà attribuée à l'un des voisins du sommet. Si c'est le cas, on incrémente la couleur jusqu'à ce qu'une couleur non attribuée soit trouvée.

9. `**resultat[sommet] = coleur;`:** Attribue la couleur `**coleur**` au sommet actuel en l'ajoutant à l'objet `resultat`.

10. Une fois que tous les sommets ont été traités, la fonction `**return resultat**;` renvoie l'objet `**resultat**` contenant les couleurs attribuées à chaque sommet du graphe.

Exemple :

```
{
        "A":{"G":1,"H":1,"D":1,"B":1,"C":1},
        "B":{"G":1,"A":1,"E":1,"F":1},
        "C":{"G":1,"A":1,"D":1,"F":1,"H":1},
        "D":{"A":1,"C":1,"E":1,"H":1},
        "E":{"B":1,"F":1,"D":1,"G":1},
        "F":{"E":1,"C":1,"B":1},
        "G":{"A":1,"B":1,"E":1,"C":1},
        "H":{"D":1,"C":1,"A":1}
}
```



1. On extrait les sommets du graphe :

sommets = ["A", "B", "C", "D", "E", "F", "G", "H"]

2. On crée un objet vide pour stocker les couleurs attribuées :

resultat = {}

3. On commence la boucle pour chaque sommet :
    a. Pour le sommet "A" :
        i. On crée un ensemble vide pour les couleurs attribuées aux voisins.
        ii. On parcourt les voisins (G, H, D, B, C) et si un voisin a une couleur attribuée, on l'ajoute à l'ensemble.
        iii. Ensemble attribue : {1}
        iv. On commence avec la couleur 1.

v. On vérifie si la couleur 1 est déjà attribuée aux voisins, mais elle n'est pas dans l'ensemble attribué.
vi. Donc, la couleur 1 est attribuée au sommet A.
vii. resultat = {"A": 1}

b. Pour le sommet "B" :
   i. On répète les mêmes étapes que pour le sommet A.
   ii. Ensemble attribue : {1}
   iii. La couleur 1 est déjà attribuée aux voisins G et A, donc on incrémente la couleur à 2.
   iv. La couleur 2 est attribuée au sommet B.
   v. resultat = {"A": 1, "B": 2}

c. Pour le sommet "C" :
   i. On répète les mêmes étapes que pour le sommet A.
   ii. Ensemble attribue : {1, 2}
   iii. La couleur 1 est déjà attribuée aux voisins G et A, et la couleur 2 est déjà attribuée au sommet B.
   iv. On incrémente la couleur à 3.
   v. La couleur 3 est attribuée au sommet C.
   vi. resultat = {"A": 1, "B": 2, "C": 3}

d. Pour le sommet "D" :
   i. On répète les mêmes étapes que pour le sommet A.
   ii. Ensemble attribue : {1, 2, 3}
   iii. La couleur 1 est déjà attribuée aux voisins G et A, la couleur 2 est attribuée au sommet B, et la couleur 3 est attribuée au sommet C.
   iv. On incrémente la couleur à 4.
   v. La couleur 4 est attribuée au sommet D.
   vi. resultat = {"A": 1, "B": 2, "C": 3, "D": 4}

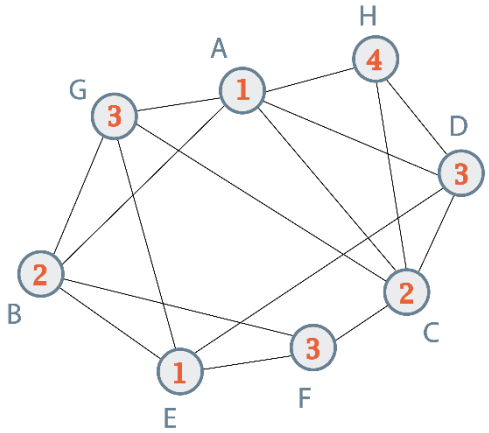
e. On continue ainsi pour les autres sommets.

4. Après avoir parcouru tous les sommets et attribué les couleurs, le résultat final est :

resultat = {"A": 1, "B": 2, "C": 2, "D": 3, "E": 1, "F": 3, "G": 3, "H": 4}

Le résultat de la coloration avec l'algorithme de Welsh-Powell pour le graphe donné est :

A - 1
C - 2
B - 2
D - 3
E - 1
G - 3
F - 3
H - 4



Chaque sommet est suivi de le nombre attribuée.

2.2.2 Algorithme de Welsh-Powell pour colorier un graphe

La fonction **welshPowell(graph)** est une implémentation de l'algorithme de Welsh-Powell en JavaScript. Voici les étapes de cette fonction :

1. On extrait les noms des sommets du graphe en utilisant la fonction **Object.keys()** et on initialise les objets **sommetColore** et **degreeSommet** :
    - **nomSommets** contient les noms des sommets du graphe.
    - **sommetColore** est un objet utilisé pour suivre si un sommet a déjà été coloré (initialisé à **false** pour tous les sommets).
    - **degreeSommet** est un objet qui stocke le degré de chaque sommet, c'est-à-dire le nombre de voisins qu'il a.
2. On initialise le compteur à 0 et l'objet **resultatColoriage** vide.
3. On définit une fonction récursive **colorer(sommet)** pour attribuer les couleurs aux sommets.
    - On incrémente le compteur à chaque appel de la fonction pour suivre le nombre de sommets déjà colorés.
    - On recherche le sommet non coloré ayant le plus grand degré (**degreeTemp**) et on le stocke dans la variable **grande**.
    - On initialise **coleur** à 1 et **voisinsColores** comme un tableau vide.
    - On parcourt les voisins de **grande** et si un voisin est déjà coloré, on ajoute sa couleur à **voisinsColores**.

- On itère sur les valeurs de 1 à **coleur** jusqu'à ce qu'on trouve une couleur qui n'est pas présente dans **voisinsColores**.
- On attribue cette couleur à **grande** dans l'objet **resultatColoriage** et on marque **grande** comme coloré en mettant **sommetColore[grande]** à **true**.
- On vérifie si tous les sommets ont été colorés en utilisant **Object.values(sommetColore).every((i) => i)**. Si c'est le cas, on retourne **resultatColoriage**.
- Sinon, on rappelle la fonction **colorer** en utilisant un sommet non coloré pour continuer la coloration récursive.

4. Enfin, on appelle **colorer** avec le premier sommet de **nomSommets[0]** et on retourne le résultat de la coloration.

Exemple :

En utilise la même graph précédente :

1. On initialise les variables :

- `nomSommets = ["A", "B", "C", "D", "E", "F", "G", "H"]`
- `sommetColore = {}`
- `degreeSommet = {}`

2. On parcourt les sommets et initialise `sommetColore` à `false` pour chaque sommet, et `degreeSommet` au nombre de voisins de chaque sommet :

- `sommetColore = {"A": false, "B": false, "C": false, "D": false, "E": false, "F": false, "G": false, "H": false}`
- `degreeSommet = {"A": 5, "B": 4, "C": 5, "D": 4, "E": 4, "F": 3, "G": 4, "H": 3}`

3. On appelle la fonction `colorer` avec le premier sommet, "A":

- `compteur = 0`
- `resultatColoriage = {}`

a. Dans la fonction `colorer` :
- `compteur` devient 1
- `grande = "C"` (sommet non coloré avec le plus grand degré)
- `degreeTemp = 5`
- On cherche les voisins colorés de "C" (G, A, D, F, H) :
  - `voisinsColores = []`

- On incrémente `coleur` à 1 et vérifie si la couleur est déjà présente dans `voisinsColores`. La couleur 1 n'est pas présente.

- On attribue la couleur 1 à "C" dans `resultatColoriage` et marque "C" comme coloré.
- On vérifie si tous les sommets sont colorés, mais ce n'est pas le cas.
- On rappelle `colorer` avec le prochain sommet non coloré, qui est "B".

b. Dans la fonction `colorer` :
- `compteur` devient 2
- `grande = "D"`
- `degreeTemp = 4`
- On cherche les voisins colorés de "D" (A, C, E, H) :
  - `voisinsColores = [1]`
- On incrémente `coleur` à 2 et vérifie si la couleur est déjà présente dans `voisinsColores`.La couleur 2 n'est pas présente.
- On attribue la couleur 2 à "D" dans `resultatColoriage` et marque "D" comme coloré.
- On vérifie si tous les sommets sont colorés, mais ce n'est pas le cas.
- On rappelle `colorer` avec le prochain sommet non coloré, qui est "E".

c. Dans la fonction `colorer` :
- `compteur` devient 3
- `grande = "G"`
- `degreeTemp = 4`
- On cherche les voisins colorés de "G" (A, B, E, C) :
  - `voisinsColores = [1, 2]`
- On incrémente `coleur` à 3 et vérifie si

la couleur est déjà présente dans `voisinsColores`. La couleur 3 n'est pas présente.
- On attribue la couleur 3 à "G" dans `resultatColoriage` et marque "G" comme coloré.
- On vérifie si tous les sommets sont colorés, mais ce n'est pas le cas.
- On rappelle `colorer` avec le prochain sommet non coloré, qui est "E".

d. Dans la fonction `colorer` :
- `compteur` devient 4
- `grande = "E"`
- `degreeTemp = 4`
- On cherche les voisins colorés de "E" (B, F, D, G) :
  - `voisinsColores = [2, 3]`
- On incrémente `coleur` à 4 et vérifie si la couleur est déjà présente dans `voisinsColores`. La couleur 4 n'est pas présente.
- On attribue la couleur 4 à "E" dans `resultatColoriage` et marque "E" comme coloré.
- On vérifie si tous les sommets sont colorés, mais ce n'est pas le cas.
- On rappelle `colorer` avec le prochain sommet non coloré, qui est "F".

e. Dans la fonction `colorer` :
- `compteur` devient 5
- `grande = "F"`
- `degreeTemp = 3`
- On cherche les voisins colorés de "F" (E, C, B) :
  - `voisinsColores = [4, 2]`
- On incrémente `coleur` à 5 et vérifie si la couleur est déjà présente dans `voisinsColores`. La couleur 5 n'est pas présente.

- On attribue la couleur 5 à "F" dans `resultatColoriage` et marque "F" comme coloré.
- On vérifie si tous les sommets sont colorés, mais ce n'est pas le cas.
- On rappelle `colorer` avec le prochain sommet non coloré, qui est "H".

f. Dans la fonction `colorer` :

- `compteur` devient 6
- `grande = "H"`
- `degreeTemp = 3`
- On cherche les voisins colorés de "H" (D, C, A) :
  - `voisinsColores = [2, 3]`
- On incrémente `coleur` à 4 et vérifie si la couleur est déjà présente dans `voisinsColores`. La couleur 4 est déjà présente.
- On incrémente `coleur` à 5 et vérifie si la couleur est déjà présente dans `voisinsColores`. La couleur 5 n'est pas présente.
- On attribue la couleur 5 à "H" dans `resultatColoriage` et marque "H" comme coloré.
- On vérifie si tous les sommets sont colorés, et c'est le cas.
- On retourne `resultatColoriage` : `{"A": 1, "B": 2, "C": 2, "D": 3, "E": 4, "F": 5, "G": 3, "H": 5}`

Le résultat de la

coloration du graphe selon l'algorithme de Welsh-Powell est :

1. A - 1
2. B - 2
3. C - 2
4. D - 3
5. E - 4
6. F - 5
7. G - 3
8. H - 5

3. Optimisation dans les graphes

3.1 Arbres couvrants à poids minimum

3.1.1 Algorithme de Prim

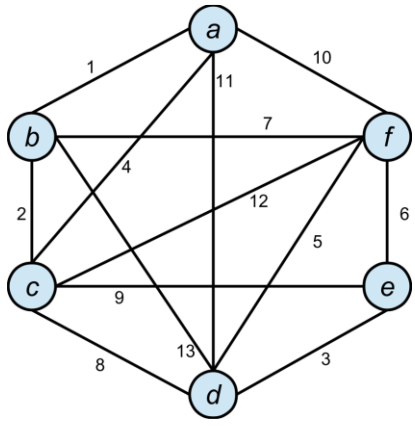
les étapes de la fonction **prim(graph, debutSommet)** :

1. Initialiser la variable **visiter** comme un tableau vide.
2. Ajouter le **debutSommet** au tableau **visiter**.
3. Initialiser les variables **arbre** et **aretes** comme des tableaux vides.
4. Tant que la longueur du tableau **visiter** n'est pas égale au nombre de clés dans le **graph** :
    - Déclarer les variables **minPoids** avec une valeur infinie, **minSommet** à null et **minArete** à null.
    - Parcourir les éléments du tableau **visiter** à l'aide d'une boucle **for**.
    - Vérifier si le **graph** a des voisins pour le sommet courant **visiter[i]**.
    - Pour chaque voisin dans les voisins du sommet courant :
        - Vérifier si le voisin n'est pas déjà dans **visiter** et si le poids de l'arête entre le sommet courant et le voisin est inférieur à **minPoids**.
        - Si la condition est satisfaite, mettre à jour **minPoids**, **minSommet** et **minArete** avec les valeurs correspondantes.
    - Ajouter **minSommet** à **visiter**.
    - Ajouter **minArete** à la fin des tableaux **arbre** et **aretes**.
5. Retourner le tableau **aretes**.

Ces étapes illustrent comment la fonction de l'algorithme de Prim parcourt le graphique donné pour construire un arbre couvrant minimum en sélectionnant à chaque étape l'arête de poids minimum qui connecte un sommet visité à un sommet non visité.

Exemple :

```
{
   "a" : { "b" : 1, "c" : 4 , "d" : 11 , "f" : 10},
   "b" : { "a" : 1 , "f" : 7 , "d" : 13,"c" : 2},
   "c" : { "b" : 2,"a" : 4 , "f" : 12,"e" : 9,"d" : 8},
   "d" : { "c" : 8,"b" : 13,"a" : 11,"f" : 5,"e" : 3},
   "e" : { "d" : 3,"c" : 9,"f" : 6},
   "f" : { "a" : 10,"b" : 7,"c" : 12,"d" : 5}
}
```



Étape 1 :

- Sélection du sommet de départ "a".
- Ajout du sommet "a" à la liste des sommets visités.
- Initialisation des listes "arbre" et "aretes" vides.

Étape 2 :

- Recherche de l'arête de poids minimum reliant un sommet visité à un sommet non visité.
- Parmi les sommets visités, les voisins de "a" sont "b", "c", "d" et "f".
- L'arête de poids minimum est "a -> b" avec un poids de 1.
- Ajout de l'arête "a -> b" à la liste "arbre" et "aretes".

Étape 3 :

- Ajout du sommet "b" à la liste des sommets visités.
- Recherche de l'arête de poids minimum reliant les sommets visités aux sommets non visités.
- Parmi les sommets visités, les voisins de "b" sont "a", "f", "d" et "c".
- L'arête de poids minimum est "b -> c" avec un poids de 2.
- Ajout de l'arête "b -> c" à la liste "arbre" et "aretes".

Étape 4 :

- Ajout du sommet "c" à la liste des sommets visités.
- Recherche de l'arête de poids minimum reliant les sommets visités aux sommets non visités.
- Parmi les sommets visités, les voisins de "c" sont "b", "a", "f", "e" et "d".
- L'arête de poids minimum est "b -> f" avec un poids de 7.
- Ajout de l'arête "b -> f" à la liste "arbre" et "aretes".

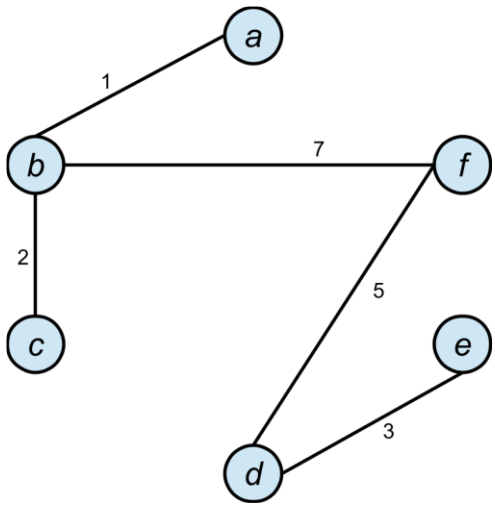
Étape 5 :

- Ajout du sommet "f" à la liste des sommets visités.
- Recherche de l'arête de poids minimum reliant les sommets visités aux sommets non visités.
- Parmi les sommets visités, les voisins de "f" sont "a", "b", "c" et "d".
- L'arête de poids minimum est "f -> d" avec un poids de 5.
- Ajout de l'arête "f -> d" à la liste "arbre" et "aretes".

Le résultat final de l'algorithme de Prim est le suivant :

Arbre couvrant minimal trouvé avec l'algorithme de Prim :

a -> b (Poids : 1)
b -> c (Poids : 2)
b -> f (Poids : 7)
f -> d (Poids : 5)
d -> e (Poids : 3)



3.1.2 Algorithme de Kruskal

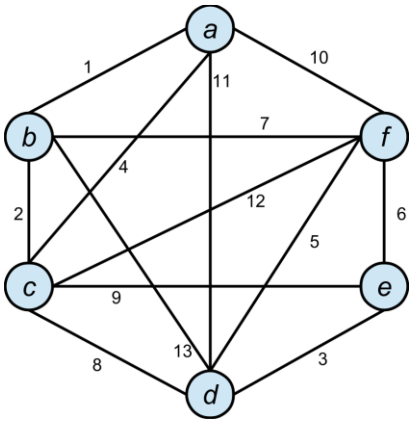
les étapes de la fonction **kruskal(graph)** :

1. Création de l'objet "**TrouverUnion**" avec une propriété "**parent**" initialisée comme un objet vide.
2. Définition de la fonction "**trouver**" dans l'objet "**TrouverUnion**". Cette fonction prend un paramètre "**x**" et effectue les actions suivantes :
    a. Si la clé "**x**" n'existe pas dans l'objet "**parent**", assigner "**x**" comme sa propre racine en l'ajoutant à l'objet "parent".
    b. Sinon, si la valeur associée à la clé "**x**" dans l'objet "**parent**" n'est pas égale à "**x**" (ce qui signifie que "**x**" n'est pas sa propre racine), récursivement appeler la fonction "trouver" en utilisant la valeur associée à la clé "**x**" dans l'objet "parent" comme nouvel argument.
    c. Retourner la valeur associée à la clé "**x**" dans l'objet "**parent**".
3. Définition de la fonction "union" dans l'objet "**TrouverUnion**". Cette fonction prend deux paramètres "**x**" et "**y**" et effectue les actions suivantes :
    a. Trouver les racines des ensembles contenant "**x**" et "**y**" en appelant la fonction "**trouver**" pour chaque paramètre.
    b. Si les racines "**racineX**" et "**racineY**" sont différentes, assigner "**racineY**" comme nouvelle racine de l'ensemble contenant "**racineX**" en mettant à jour la valeur associée à la clé "**racineX**" dans l'objet "parent".
4. Déclaration des variables "**aretes**" (liste des arêtes), "**result**" (résultat final) et "**tu**" (objet TrouverUnion).
5. Parcourir le graphe pour créer une liste de toutes les arêtes, en utilisant les boucles "**for-in**" pour itérer sur les sommets et leurs voisins.
6. Chaque arête est ajoutée à la liste "**aretes**" avec les propriétés "**u**" (sommet de départ), "v" (sommet d'arrivée) et "**poids**" (poids de l'arête).
7. Trier la liste des arêtes dans l'ordre croissant en fonction de leur poids.
8. Itérer sur les arêtes triées.
9. Pour chaque arête, extraire les sommets "**u**" et "**v**".

10. Vérifier si l'ajout de l'arête crée un cycle en appelant la fonction "**trouver**" pour les sommets "**u**" et "**v**" et en comparant leurs racines.
11. Si les racines sont différentes, cela signifie que l'ajout de l'arête ne crée pas de cycle. Ajouter l'arête au résultat final et effectuer l'opération d'union en appelant la fonction "union" avec les sommets "**u**" et "**v**".
12. Retourner le résultat final, qui est l'arbre couvrant minimal obtenu par l'algorithme de Kruskal.

Exemple :

```
{
   "a" : { "b" : 1, "c" : 4 , "d" : 11 , "f" : 10},
   "b" : { "a" : 1 , "f" : 7 , "d" : 13,"c" : 2},
   "c" : { "b" : 2,"a" : 4 , "f" : 12,"e" : 9,"d" : 8},
   "d" : { "c" : 8,"b" : 13,"a" : 11,"f" : 5,"e" : 3},
   "e" : { "d" : 3,"c" : 9,"f" : 6},
   "f" : { "a" : 10,"b" : 7,"c" : 12,"d" : 5}
}
```



Voici les étapes appliquées à l'exemple donné :

1. Création de l'objet "TrouverUnion" avec une propriété "parent" initialisée comme un objet vide.
2. Définition de la fonction "trouver" dans l'objet "TrouverUnion".
3. Définition de la fonction "union" dans l'objet "TrouverUnion".
4. Déclaration des variables "aretes", "result" et "tu".
5. Parcours du graphe pour créer une liste de toutes les arêtes.

a -> b (poids : 1)	c -> b (poids : 2)	d -> f (poids : 5)
a -> c (poids : 4)	c -> a (poids : 4)	d -> e (poids : 3)
a -> d (poids : 11)	c -> f (poids : 12)	e -> d (poids : 3)
a -> f (poids : 10)	c -> e (poids : 9)	e -> c (poids : 9)
b -> a (poids : 1)	c -> d (poids : 8)	e -> f (poids : 6)
b -> f (poids : 7)	d -> c (poids : 8)	f -> a (poids : 10)
b -> d (poids : 13)	d -> b (poids : 13)	f -> b (poids : 7)
b -> c (poids : 2)	d -> a (poids : 11)	f -> c (poids : 12)
		f -> d (poids : 5)

6. Tri de la liste des arêtes par ordre croissant de poids.

a -> b (poids : 1) b -> a (poids : 1) d -> e (poids : 3) e -> d (poids : 3) b -> c (poids : 2) c -> b (poids : 2) f -> b (poids : 7) b -> f (poids : 7)	f -> d (poids : 5) d -> f (poids : 5) c -> e (poids : 9) e -> c (poids : 9) c -> d (poids : 8) d -> c (poids : 8) c -> a (poids : 4) a -> c (poids : 4) f -> a (poids : 10)	a -> f (poids : 10) f -> c (poids : 12) c -> f (poids : 12) d -> b (poids : 13) b -> d (poids : 13) a -> d (poids : 11) d -> a (poids : 11) e -> f (poids : 6) f -> e (poids : 6)

7. Itération sur les arêtes triées.

Étape 1 : a -> b (poids : 1) - Pas de cycle, ajout de l'arête à result. Étape 2 : b -> a (poids : 1) - Cycle détecté, l'arête est ignorée. Étape 3 : d -> e (poids : 3) - Pas de cycle, ajout de l'arête à result. Étape 4 : e -> d (poids : 3) - Cycle détecté, l'arête est ignorée. Étape 5 : b -> c (poids : 2) - Pas de cycle, ajout de l'arête à result. Étape 6 : c -> b (poids : 2) - Cycle détecté, l'arête est ignorée. Étape 7 : f -> b (poids : 7) - Pas de cycle, ajout de l'arête à result. Étape 8 : b -> f (poids : 7) - Cycle détecté, l'arête est ignorée. Étape 9 : f -> d (poids : 5) - Pas de cycle, ajout de l'arête à result. Étape 10 : d -> f (poids : 5) - Cycle détecté, l'arête est ignorée. Étape 11 : c -> e (poids : 9) - Pas de cycle, ajout de l'arête à result. Étape 12 : e -> c (poids : 9) - Cycle détecté, l'arête est ignorée. Étape 13 : c -> d (poids : 8) - Pas de cycle, ajout de l'arête à result.	Étape 14 : d -> c (poids : 8) - Cycle détecté, l'arête est ignorée. Étape 15 : c -> a (poids : 4) - Cycle détecté, l'arête est ignorée. Étape 16 : a -> c (poids : 4) - Cycle détecté, l'arête est ignorée. Étape 17 : f -> a (poids : 10) - Cycle détecté, l'arête est ignorée. Étape 18 : a -> f (poids : 10) - Cycle détecté, l'arête est ignorée. Étape 19 : f -> c (poids : 12) - Cycle détecté, l'arête est ignorée. Étape 20 : c -> f (poids : 12) - Cycle détecté, l'arête est ignorée. Étape 21 : d -> b (poids : 13) - Cycle détecté, l'arête est ignorée. Étape 22 : b -> d (poids : 13) - Cycle détecté, l'arête est ignorée. Étape 23 : a -> d (poids : 11) - Cycle détecté, l'arête est ignorée. Étape 24 : d -> a (poids : 11) - Cycle détecté, l'arête est ignorée. Étape 25 : e -> f (poids : 6) - Pas de cycle, ajout de l'arête à result. Étape 26 : f -> e (poids : 6) - Cycle détecté, l'arête est ignorée.

Le résultat final est :

[

{ u: 'a', v: 'b', poids: 1 },

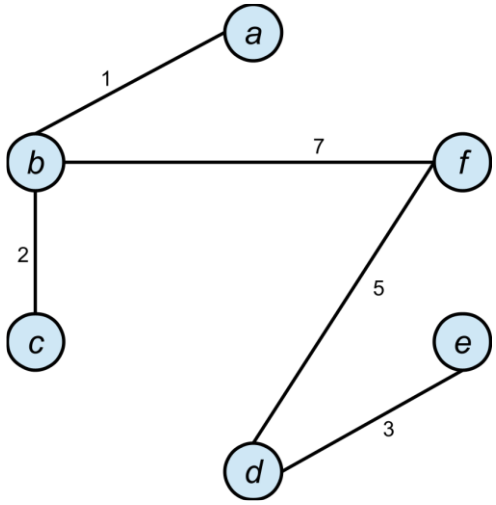
{ u: 'd', v: 'e', poids: 3 },

{ u: 'b', v: 'c', poids: 2 },

{ u: 'f', v: 'b', poids: 7 },

{ u: 'f', v: 'd', poids: 5 }

]



Ces arêtes forment l'arbre couvrant minimal du graphe donné.

3.2 Problème de plus court chemin

3.2.1 Algorithme de Bellman-Ford

Voici les étapes de la fonction **bellmanFord(graph, debutSommet)** :

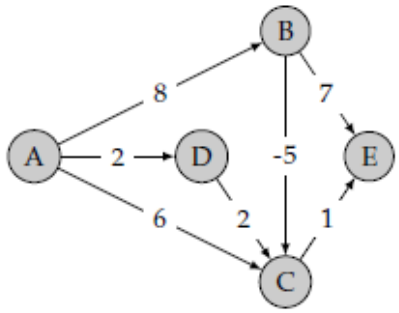
1. Obtenez le nombre total de sommets dans le graphe en utilisant **Object.keys(graph).length**.
2. Initialisez deux tableaux : **distances** et **predecesseurs**. Ces tableaux sont utilisés pour stocker les distances minimales et les prédécesseurs de chaque sommet.
   - **distances** est un objet avec chaque sommet comme clé et la distance minimale depuis le sommet de départ comme valeur initiale.
   - **predecesseurs** est un objet avec chaque sommet comme clé et le prédécesseur du sommet sur le chemin le plus court depuis le sommet de départ comme valeur initiale.
3. Initialisez les distances et les prédécesseurs pour tous les sommets sauf le sommet de départ :
   - Parcourez tous les sommets du graphe à l'aide d'une boucle **for...in**.
   - Définissez la distance initiale pour chaque sommet à **Number.MAX_VALUE** (représentant l'infini) sauf pour le sommet de départ, que vous définissez à 0.
   - Définissez le prédécesseur initial pour chaque sommet à null.
4. Effectuez l'itération principale **sommets - 1** fois, où **sommets** est le nombre total de sommets :
   - Parcourez tous les sommets du graphe à l'aide d'une boucle **for...in**.
   - Pour chaque sommet, parcourez tous ses voisins à l'aide d'une autre boucle **for...in**.

- Pour chaque voisin, mettez à jour la distance minimale si la distance actuelle depuis le sommet de départ plus le poids de l'arête vers le voisin est inférieure à la distance minimale enregistrée pour le voisin.
- Mettez à jour le prédécesseur du voisin si la distance a été mise à jour.

5. Vérifiez les cycles de poids négatifs :
   - Parcourez à nouveau tous les sommets et leurs voisins.
   - Si la distance actuelle depuis le sommet de départ plus le poids de l'arête vers le voisin est inférieure à la distance minimale enregistrée pour le voisin, cela signifie qu'il y a un cycle de poids négatif dans le graphe.
   - Affichez un message d'erreur indiquant la présence d'un cycle de poids négatif.
6. Retournez un objet contenant les distances minimales (**distances**) et les prédécesseurs (**predecesseurs**).

Exemple :

```
{
        "a" : {"b" : 8, "d" : 2 , "c" : 6},
        "b" : {"c": -5,"e" : 7},
        "c" : {"e" : 1},
        "d" : {"c" : 2},
        "e" : {}
}
```



Étape 1:

- distances = {"a": 0, "b": ∞, "c": ∞, "d": ∞, "e": ∞}
- prédécesseurs = {"a": null, "b": null, "c": null, "d": null, "e": null}

Étape 2:

- Itération 1 :
  - a -> b : distance[a] + poids(a -> b) = 0 + 8 = 8
    - 8 < distances[b] (∞), donc mettre à jour distances[b] = 8 et prédécesseurs[b] = a
  - a -> d : distance[a] + poids(a -> d) = 0 + 2 = 2
    - 2 < distances[d] (∞), donc mettre à jour distances[d] = 2 et prédécesseurs[d] = a
  - a -> c : distance[a] + poids(a -> c) = 0 + 6 = 6
    - 6 < distances[c] (∞), donc mettre à jour distances[c] = 6 et prédécesseurs[c] = a

- Itération 2 :
    - b -> c : distance[b] + poids(b -> c) = 8 + (-5) = 3
        - 3 < distances[c] (6), donc mettre à jour distances[c] = 3 et prédécesseurs[c] = b
    - b -> e : distance[b] + poids(b -> e) = 8 + 7 = 15
        - 15 > distances[e] (∞), donc ne pas mettre à jour distances[e] et prédécesseurs[e]
- Itération 3 :
    - c -> e : distance[c] + poids(c -> e) = 3 + 1 = 4
        - 4 < distances[e] (∞), donc mettre à jour distances[e] = 4 et prédécesseurs[e] = c
- Itération 4 :
    - d -> c : distance[d] + poids(d -> c) = 2 + 2 = 4
        - 4 > distances[c] (3), donc ne pas mettre à jour distances[c] et prédécesseurs[c]

Étape 3:

- Pas de cycle de poids négatif détecté.

Étape 4:

- Aucune itération supplémentaire n'est nécessaire car nous avons déjà effectué 4 itérations.

Après avoir terminé ces étapes, voici les résultats obtenus :

- distances = {"a": 0, "b": 8, "c": 3, "d": 2, "e": 4}
- prédécesseurs = {"a": null, "b": "a", "c": "b", "d": "a", "e": "c"}

Cela signifie que le chemin le plus court depuis le sommet "a" vers les autres sommets est le suivant :

- a -> b -> c -> e avec une distance totale de 4
- a -> d -> c -> e avec une distance totale de 4

Distances depuis la sommet de départ :

1. a: 0
2. b: 8
3. c: 3
4. d: 2
5. e: 4

Les sommets prédécesseur :

1. a: -
2. b: a
3. c: b
4. d: a
5. e: c

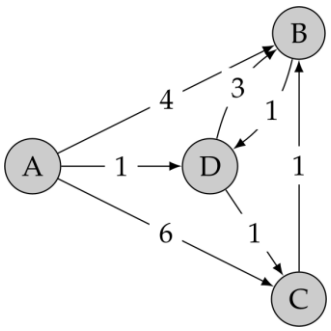
3.2.2 Algorithme de Dijkstra

Voici les étapes de la fonction **bellmanFord(graph, debutSommet)** :

1. Créez un ensemble de sommets non visités et initialisez toutes les distances de départ à l'infini, sauf pour le sommet source dont la distance est de 0.
2. Tant que l'ensemble de sommets non visités n'est pas vide, répétez les étapes suivantes :
    a. Sélectionnez le sommet non visité avec la plus petite distance. Au début, cela sera le sommet source avec une distance de 0.
    b. Marquez ce sommet comme visité pour ne plus le considérer dans les prochaines itérations.
    c. Pour chaque voisin du sommet sélectionné, calculez la distance totale depuis le sommet source en passant par le sommet sélectionné. La distance totale est la somme de la distance du sommet sélectionné et du poids de l'arête reliant le sommet sélectionné à son voisin.
    d. Si cette distance totale est plus petite que la distance actuellement enregistrée pour le voisin, mettez à jour la distance du voisin avec cette nouvelle distance totale.
    e. Enregistrez le sommet sélectionné comme prédécesseur du voisin. Cela permettra de reconstruire le chemin le plus court une fois l'algorithme terminé.
3. Une fois toutes les étapes effectuées pour tous les sommets accessibles, vous aurez les distances les plus courtes depuis le sommet source vers tous les autres sommets accessibles du graphe, ainsi que les prédécesseurs correspondants pour chaque sommet.
4. Vous pouvez retourner les distances et les prédécesseurs sous la forme d'un objet ou d'une structure de données appropriée pour une utilisation ultérieure.
5. Si vous souhaitez reconstruire le chemin le plus court entre le sommet source et un autre sommet, vous pouvez utiliser les prédécesseurs en remontant de ce sommet jusqu'au sommet source en suivant les prédécesseurs enregistrés.

Exemple :

```
{
        "a" : {"b" : 4,"d" : 1 , "c" : 6},
        "b" : {"d" : 1},
        "c" : {"b" : 1},
        "d" : {"b" : 3, "c" : 1}
}
```



Voici les étapes détaillées de l'algorithme de Dijkstra appliqué au graphe donné :

Étape 1 :

- Initialisez toutes les distances à l'infini, sauf la distance du sommet source à 0.
  - distances = {a: Infinity, b: Infinity, c: Infinity, d: Infinity}
  - distances[a] = 0

Étape 2 :

- Sélectionnez le sommet avec la plus petite distance non visitée (dans ce cas, c'est "a").
  - Sommet sélectionné : a

Étape 3 :

- Parcourez tous les voisins du sommet sélectionné ("a") et mettez à jour les distances si une distance plus courte est trouvée.
  - Voisin "b" : distance[a] + poids(a, b) = 0 + 4 = 4
    - Comme 4 < distances[b], mettez à jour distances[b] à 4.
    - Mettez à jour le prédécesseur de b à a : predecesseur[b] = a
  - Voisin "d" : distance[a] + poids(a, d) = 0 + 1 = 1
    - Comme 1 < distances[d], mettez à jour distances[d] à 1.
    - Mettez à jour le prédécesseur de d à a : predecesseur[d] = a
  - Voisin "c" : distance[a] + poids(a, c) = 0 + 6 = 6
    - Comme 6 > distances[c], ne mettez pas à jour distances[c].

Après cette étape, les distances et les prédécesseurs sont les suivants :

Distances depuis le sommet de départ :
- a: 0
- b: 4
- c: Infinity
- d: 1

Prédécesseurs des sommets :
- a: null
- b: a
- c: null
- d: a

Étape 4 :

- Marquez le sommet "a" comme visité.

Étape 5 :

- Répétez les étapes 2 à 4 pour tous les sommets non visités.

Étape 2 :

- Sélectionnez le sommet avec la plus petite distance non visitée (dans ce cas, c'est "d").
- Sommet sélectionné : d

Étape 3 :

- Parcourez tous les voisins du sommet sélectionné ("d") et mettez à jour les distances si une distance plus courte est trouvée.
    - Voisin "b" : distance[d] + poids(d, b) = 1 + 3 = 4
        - Comme 4 > distances[b], ne mettez pas à jour distances[b].
    - Voisin "c" : distance[d] + poids(d, c) = 1 + 1 = 2
        - Comme 2 < distances[c], mettez à jour distances[c] à 2.
        - Mettez à jour le prédécesseur de c à d : predecesseur[c] = d

Après cette étape, les distances et les prédécesseurs sont les suivants :

Distances depuis le sommet de départ :
- a: 0
- b: 4
- c: 2
- d: 1

Prédécesseurs des sommets :

- a: null
- b: a
- c: d
- d: a

Étape 4 :

- Marquez le sommet "d" comme visité.

Étape 5 :

- Répétez les étapes 2 à 4 pour tous les sommets non visités.

Étape 2 :

- Sélectionnez le sommet avec la plus petite distance non visitée (dans ce cas, c'est "c").
    - Sommet sélectionné : c

Étape 3 :

- Parcourez tous les voisins du sommet sélectionné ("c") et mettez à jour les distances si une distance plus courte est trouvée.
    - Voisin "b" : distance[c] + poids(c, b) = 2 + 1 = 3
        - Comme 3 < distances[b], mettez à jour distances[b] à 3.
        - Mettez à jour le prédécesseur de b à c : predecesseur[b] = c

Après cette étape, les distances et les prédécesseurs sont les suivants :

Distances depuis le sommet de départ :
- a: 0
- b: 3
- c: 2
- d: 1

Prédécesseurs des sommets :
- a: null
- b: c
- c: d
- d: a

Étape 4 :

- Marquez le sommet "c" comme visité.

Étape 5 :

- Répétez les étapes 2 à 4 pour tous les sommets non visités.

Étape 2 :

- Sélectionnez le sommet avec la plus petite distance non visitée (dans ce cas, c'est "b").
    - Sommet sélectionné : b

Étape 3 :

- Parcourez tous les voisins du sommet sélectionné ("b") et mettez à jour les distances si une distance plus courte est trouvée.
    - Voisin "d" : distance[b] + poids(b, d) = 3 + 1 = 4
        - Comme 4 > distances[d], ne mettez pas à jour distances[d].

Après cette étape, les distances et les prédécesseurs restent les mêmes :

Distances depuis le sommet de départ :
- a: 0
- b: 3
- c: 2
- d: 1

Prédécesseurs des sommets :
- a: null
- b: c
- c: d
- d: a

Étape 4 :
- Marquez le sommet "b" comme visité.

Étape 5 :
- Répétez les étapes 2 à 4 pour tous les sommets non visités.

Étant donné que tous les sommets ont été visités, l'algorithme de Dijkstra est terminé.

Les résultats finaux sont les suivants :

Distances depuis le sommet de départ :
- a: 0
- b: 3
- c: 2
- d: 1

Prédécesseurs des sommets :
- a: null
- b: c
- c: d
- d: a

4. Conclusion

Dans ce rapport, nous avons exploré en détail les algorithmes clés proposés par "Graph Algorithms Online", une plateforme en ligne dédiée à l'apprentissage et à l'expérimentation des graphes. Les graphes et leurs algorithmes jouent un rôle essentiel dans de nombreux domaines de l'informatique, et cette plateforme offre une interface conviviale et des outils puissants pour faciliter leur exploration.

Nous avons étudié huit algorithmes majeurs, couvrant des domaines tels que le parcours de graphes, la coloration, l'optimisation et la recherche de chemins les plus courts. Chaque algorithme a été présenté avec une explication détaillée de son fonctionnement et de ses étapes temporelles, permettant une meilleure compréhension de leur efficacité et de leur applicabilité dans différents scénarios.

En outre, des exemples pratiques ont été fournis pour chaque algorithme, illustrant leur mise en œuvre et les résultats obtenus. Cela a permis de mettre en évidence la valeur ajoutée de la plateforme "Graph Algorithms Online" pour ceux qui souhaitent explorer et approfondir leur compréhension de ce domaine fascinant.

L'application web développée pour ce mini-projet a permis aux utilisateurs d'interagir avec les algorithmes en fournissant leurs propres graphes personnalisés ou en utilisant des exemples prédéfinis. En saisissant le graphe au format JSON et en sélectionnant l'algorithme souhaité, les utilisateurs ont pu obtenir les résultats de l'algorithme appliqué au graphe, affichés de manière claire et compréhensible.

En conclusion, ce rapport a offert une vision approfondie des algorithmes clés disponibles sur la plateforme "Graph Algorithms Online". Il a démontré la pertinence et l'utilité de ces algorithmes dans divers contextes, ainsi que la valeur ajoutée de la plateforme pour l'apprentissage et l'expérimentation des graphes. En continuant à explorer et à étudier ces algorithmes, les utilisateurs pourront développer leurs compétences en résolution de problèmes complexes et en optimisation, contribuant ainsi au progrès de la science des données, de la recherche opérationnelle et de la théorie des réseaux.