



# Documentation

## TP2 SECTION

### Vérification

Fichier : `verificateur.py` dans le dossier `src`

#### Description du problème :

Ce code résout le problème de la vérification de la validité d'une 3-coloration d'un graphe. Une 3-coloration d'un graphe est une attribution de trois couleurs (par exemple : Rouge, Bleu et Vert) aux sommets du graphe de manière à ce que deux sommets adjacents ne partagent pas la même couleur. La fonction `is_valid_3_coloration` parcourt les sommets du graphe pour vérifier si la 3-coloration est valide.

#### Analyse de la complexité temporelle :

La complexité temporelle de cette fonction dépend de la taille du graphe (le nombre de sommets et d'arêtes) et peut être décomposée comme suit :

Parcours de tous les sommets du graphe :  $O(V)$ , où  $V$  est le nombre de sommets.

Pour chaque sommet, parcours de ses voisins :  $O(E)$ , où  $E$  est le nombre d'arêtes.

Vérification de la couleur du sommet et de ses voisins :  $O(1)$ .

La complexité totale de la fonction est donc  $O(V * (E + 1))$ , où  $V$  et  $E$  sont respectivement le nombre de sommets et d'arêtes du graphe.

---

## SolvBackTracking

Fichier : `SolvBackTracking.py` dans le dossier `src`

### Description du problème :

Ce code résout le problème de la coloration de graphe en utilisant un algorithme de backtracking. Le but est d'attribuer une couleur (parmi Rouge, Bleu et Vert) à chaque sommet du graphe de manière à ce que deux sommets adjacents n'aient pas la même couleur. L'algorithme explore les différentes combinaisons de couleurs de manière récursive pour trouver une solution.

### Analyse de la complexité temporelle :

L'analyse de la complexité temporelle de cet algorithme peut être décomposée comme suit :

Parcours de tous les sommets du graphe :  $O(V)$ , où  $V$  est le nombre de sommets.

Pour chaque sommet, il y a une boucle qui parcourt les trois couleurs possibles (Rouge, Bleu, Vert). Dans le pire cas, cela donne 3 itérations.

À chaque itération, la fonction `is_safe` est appelée, qui parcourt les voisins du sommet. Le nombre de voisins dépend du graphe, mais il est généralement proportionnel au nombre d'arêtes.

La complexité totale dépend donc du nombre de sommets, du nombre d'arêtes et du nombre de couleurs. Dans le pire cas, la complexité est exponentielle, mais elle peut être considérablement réduite avec des heuristiques d'optimisation ou d'autres techniques.

---

## SolvRandom

Fichier : `SolvRandom.py` dans le dossier `src`

## Description du problème :

Ce code propose une approche aléatoire pour résoudre le problème de la coloration de graphe. L'objectif est d'attribuer aléatoirement une couleur (parmi Rouge, Bleu et Vert) à chaque sommet du graphe de manière à ce que deux sommets adjacents n'aient pas la même couleur. L'algorithme génère une coloration aléatoire, puis modifie de manière aléatoire les couleurs des sommets problématiques jusqu'à trouver une coloration valide ou atteindre un nombre maximum d'itérations.

## Analyse de la complexité temporelle :

L'analyse de la complexité temporelle de cet algorithme est la suivante :

La fonction `random_coloring` attribue une couleur à chaque sommet du graphe, ce qui a une complexité linéaire  $O(V)$ , où  $V$  est le nombre de sommets.

La fonction `find_problematic_edges` parcourt les arêtes du graphe, ce qui a une complexité proportionnelle au nombre d'arêtes ( $O(E)$ , où  $E$  est le nombre d'arêtes).

La boucle principale de la fonction `random_coloring_solver` effectue des itérations, jusqu'à un maximum de `max_iterations`. Dans chaque itération, on effectue des opérations qui dépendent du graphe et du nombre de sommets.

La complexité totale dépend donc de plusieurs facteurs, y compris le nombre de sommets, d'arêtes et le nombre d'itérations. Dans le pire cas, si aucune solution n'est trouvée, la complexité est de l'ordre de  $O(\text{max\_iterations} * (V + E))$ .

---

## SolvLawler

Fichier : `SolvLawler.py` dans le dossier `src`

## Description du problème :

Ce code résout le problème de détermination de la bipartition dans un graphe, en recherchant un ensemble indépendant biparti. L'objectif est de diviser les sommets du graphe en deux ensembles tels que les sommets à l'intérieur d'un ensemble ne sont pas connectés entre eux. L'algorithme explore différentes combinaisons de sommets pour trouver une telle bipartition.

## Analyse de la complexité temporelle :

L'analyse de la complexité temporelle de cet algorithme peut être décomposée comme suit :

Parcours de tous les sommets du graphe pour initialiser visited et color :  $O(V)$ , où  $V$  est le nombre de sommets.

Double boucle pour parcourir les combinaisons de sommets (combinations) :  $O(C(V, 2))$ , où  $C(V, 2)$  est le nombre de combinaisons de deux sommets parmi  $V$  sommets, ce qui est proportionnel à  $V^2$ .

Pour chaque combinaison, vérification de l'ensemble indépendant et de la bipartition en utilisant is\_independent\_set et is\_bipartite.

La complexité totale dépend donc du nombre de sommets, et la complexité est généralement  $O(V^2)$  dans le pire cas.

## **Conclusion :**

Après avoir examiné l'algorithme de recherche de la bipartition, nous avons constaté que sa mise en œuvre n'était pas complète et qu'il n'était pas possible de fournir une solution complète. La recherche d'un ensemble indépendant biparti dans un graphe est un problème complexe qui peut nécessiter des algorithmes plus élaborés pour trouver une solution efficace.

## **Comparaison entre l'algorithme BackTracking et Random**

Pour comparer les deux algorithmes, le solvRandom et le solvBacktracking, nous allons examiner leurs approches respectives pour résoudre le problème de coloration de graphe. Chacun a ses avantages et inconvénients, et il est important de choisir l'algorithme approprié en fonction des besoins spécifiques et des contraintes du problème.

### **1. solvRandom (Résolution Aléatoire)**

Approche : L'algorithme solvRandom utilise une approche de résolution aléatoire pour générer une coloration aléatoire du graphe. Ensuite, il vérifie si cette coloration est valide en parcourant les arêtes du graphe. Si la coloration n'est pas valide, il effectue des modifications aléatoires pour tenter de résoudre le problème.

### **Avantages :**

Peut être rapide pour les graphes de petite à moyenne taille.  
Ne nécessite pas de recherche exhaustive.  
Peut être utile pour obtenir une solution approximative rapidement.

**Inconvénients :**

Ne garantit pas toujours de trouver une solution valide.  
Peut ne pas être efficace pour les graphes complexes ou les instances difficiles.

**2. solvBacktracking (Résolution par Backtracking)**

Approche : L'algorithme solvBacktracking utilise une approche de recherche exhaustive (backtracking) pour explorer toutes les possibilités de coloration du graphe. Il commence par attribuer une couleur à un sommet, puis passe au suivant, en vérifiant à chaque étape si la coloration est valide. En cas d'invalidité, il effectue un retour en arrière (backtrack) pour explorer d'autres possibilités.

**Avantages :**

Garantit de trouver une solution si elle existe.  
Convient aux graphes complexes et aux instances difficiles.  
Fournit une solution optimale si le graphe est colorable.  
Inconvénients :

Peut être très lent pour les graphes de grande taille.  
La complexité en temps peut être élevée dans le pire cas.

**Comparaison et Recommandations :**

Utilisez solvRandom lorsque vous avez besoin d'une solution rapide pour des graphes de petite à moyenne taille et que la précision n'est pas critique.

Utilisez solvBacktracking lorsque la garantie de trouver une solution valide est essentielle, même si cela signifie une recherche exhaustive. Cela convient aux graphes plus complexes, mais attendez-vous à des temps d'exécution plus longs.

**Exemples :**

Exemple d'utilisation de solvRandom : Vous avez un petit graphe avec peu de sommets et d'arêtes, et vous voulez une solution approximative rapidement.

Exemple d'utilisation de solvBacktracking : Vous devez résoudre le problème de coloration de graphe pour un graphe plus complexe où une solution exacte est requise,

même si cela prend plus de temps.

Gardez à l'esprit que la performance des algorithmes dépendra de la taille et de la complexité du graphe, ainsi que des paramètres spécifiques des algorithmes (comme le nombre maximal d'itérations pour `solvRandom` ou les heuristiques de sélection de sommet initial pour `solvBacktracking`). Il peut être judicieux d'expérimenter avec ces paramètres pour obtenir les meilleurs résultats dans des situations spécifiques.

---

## Complexité de l'algorithme de Lawler

La complexité de l'algorithme `lawler_solve` implémenté dans le code dépend principalement des boucles `for` imbriquées, des vérifications et de la taille du graphe. Voici une analyse de la complexité de cet algorithme :

Boucle externe (`size`) : Cette boucle parcourt toutes les tailles d'ensembles possibles, de 2 à `len(graph)`. La complexité de cette boucle est  $O(N)$ , où  $N$  est le nombre de sommets dans le graphe.

Boucle interne (`combinations`) : Dans cette boucle, toutes les combinaisons de deux sommets sont générées. Le nombre total de combinaisons possibles est  $C(N, 2)$ , où  $N$  est le nombre de sommets. La complexité de cette boucle est  $O(N^2)$ .

Fonctions `is_independent_set` et `is_bipartite` : Ces fonctions parcourent le graphe et effectuent des vérifications. Dans le pire cas, elles parcourent tous les sommets et leurs voisins. La complexité est donc de l'ordre de  $O(N)$ .

En prenant en compte ces éléments, la complexité totale de l'algorithme est  $O(N * N^2 * N) = O(N^4)$ . Cela signifie que la complexité de l'algorithme est polynomiale en fonction du nombre de sommets dans le graphe. Plus le graphe est grand, plus l'algorithme sera lent.

Il est important de noter que cette analyse de complexité ne tient pas compte de

facteurs constants ou d'optimisations potentielles. La complexité réelle peut être meilleure dans la pratique, mais elle reste exponentielle en fonction de la taille du graphe.

---

— Important, nous n'avons pas réussi à mettre en place l'algorithme de Lawler, mais nous avons quand même laissé le code dans le fichier SolvLawler.py

---

## TP3 SECTION

### Fonction `solve_3_coloration` et `translate_solution`

#### Description du Problème :

Le problème de la 3-Coloration consiste à attribuer une couleur parmi trois couleurs différentes (par exemple, Rouge, Bleu et Vert) à chaque sommet d'un graphe de telle manière que deux sommets adjacents n'aient pas la même couleur. L'objectif est de déterminer s'il existe une telle coloration pour un graphe donné.

#### Exemple :

Considérons un graphe avec 4 sommets et les arêtes suivantes : (1, 2), (2, 3), (3, 4), et (4, 1). Ce graphe est un cycle de 4 sommets. Le problème de la 3-Coloration consiste à déterminer s'il est possible de colorer ces sommets avec trois couleurs de telle sorte que les sommets adjacents n'aient pas la même couleur. La réponse est oui, et une possible 3-coloration de ce graphe serait {1: 'Rouge', 2: 'Bleu', 3: 'Rouge', 4: 'Bleu'}.

#### Analyse de la Complexité Temporelle :

1. Initialisation ( $O(1)$ ) : L'initialisation du nombre de sommets  $n$  est en temps constant.
2. Création des Clauses ( $O(n)$ ) : La création des clauses est

linéaire par rapport au nombre de sommets. Les trois boucles imbriquées ajoutent des clauses pour chaque sommet, chaque couleur et chaque paire de sommets voisins.

3. Affichage ( $O(1)$ ) : Les affichages du graphe d'entrée et des entrées pour le solveur SAT sont en temps constant.

4. Solveur SAT (Varie) : La complexité dépend du solveur SAT utilisé. Le temps d'exécution du solveur SAT peut varier en fonction de la taille et de la complexité du problème, mais il est généralement de l'ordre de  $O(2^n)$  dans le pire cas.

5. Traduction de la Solution ( $O(n)$ ) : La traduction de la solution SAT en une liste de couleurs pour les sommets est linéaire par rapport au nombre de sommets.

La complexité globale du programme dépend principalement de la complexité du solveur SAT, ce qui peut varier considérablement d'un problème à l'autre. Dans le pire cas, la complexité totale serait dominée par le solveur SAT, ce qui est exponentiel. Cependant, dans la pratique, le temps d'exécution dépendra de la taille et de la structure du graphe.

---

## Fonction `get_independent_set`

### Description du Problème :



Le problème de l'ensemble indépendant consiste à trouver un ensemble de sommets dans un graphe tel que ces sommets ne soient pas reliés par une arête. Plus précisément, un ensemble indépendant de taille donnée consiste à sélectionner un certain nombre de sommets du graphe de telle manière que, pour tout sommet sélectionné, aucun de ses voisins ne soit sélectionné.

### Exemple :

Considérons un graphe avec les sommets {A, B, C, D} et les arêtes {(A, B), (A, C), (B, C), (C, D)}. Pour un ensemble indépendant de taille 2, une solution possible serait {A, D} car ni A ni D ne sont voisins, formant ainsi un ensemble indépendant de taille 2.

#### Analyse de la Complexité Temporelle :

1. Initialisation ( $O(1)$ ) : L'initialisation du nombre de sommets  $n$  est en temps constant.
2. Création des Clauses ( $O(n)$ ) : La création des clauses est linéaire par rapport au nombre de sommets. Les deux boucles imbriquées ajoutent des clauses pour chaque sommet et chaque paire de sommets voisins.
3. Affichage ( $O(1)$ ) : Les affichages de l'entrée pour le solveur SAT sont en temps constant.
4. Solveur SAT (Varie) : La complexité dépend du solveur SAT utilisé. Le temps d'exécution du solveur SAT peut varier en fonction de la taille et de la complexité du problème, mais il est généralement de l'ordre de  $O(2^n)$  dans le pire cas.

5. Traduction de la Solution ( $O(n)$ ) : La traduction de la solution SAT en un ensemble de sommets est linéaire par rapport au nombre de sommets.

La complexité totale du programme dépend principalement de la complexité du solveur SAT, qui peut varier considérablement d'un problème à l'autre. Dans le pire cas, la complexité totale serait dominée par le solveur SAT, ce qui est exponentiel. Cependant, dans la pratique, le temps d'exécution dépendra de la taille et de la structure du graphe.

## **Fonction solve\_vertex\_cover\_3\_clique**

### **Description du Problème :**

Le problème de couverture de sommets par des cliques de taille 3 consiste à trouver une coloration des sommets d'un graphe tel que chaque clique (ensemble de sommets complètement connectés) ait au plus 3 couleurs différentes, de manière à minimiser le nombre de couleurs utilisées.

### **Analyse de la Complexité Temporelle :**

1. Inversion du Graphe ( $O(n + m)$ ) : L'inversion du graphe implique de parcourir toutes les arêtes du graphe et de créer un nouveau graphe complémentaire. La complexité est linéaire par rapport au nombre de sommets ( $n$ ) et d'arêtes ( $m$ ).


2. Résolution de 3-Coloration (Varie) : La résolution du problème de 3-coloration est effectuée en utilisant la fonction ``solve_3_coloration``. La complexité dépend du solveur SAT utilisé, qui peut varier en fonction de la taille et de la complexité du problème, mais il est généralement de l'ordre de  $O(2^n)$  dans le pire cas.

3. Traduction de la Solution ( $O(n)$ ) : La traduction de la solution SAT en une coloration pour le graphe original est linéaire par rapport au nombre de sommets.

4. Application de la Coloration ( $O(n)$ ) : L'application de la coloration au graphe original est également linéaire par rapport au nombre de sommets.

La complexité totale du programme dépend principalement de la complexité du solveur SAT, qui peut varier considérablement d'un problème à l'autre. Dans le pire cas, la complexité totale serait dominée par le solveur SAT, ce qui est exponentiel. Cependant, dans la pratique, le temps d'exécution dépendra de la taille et de la structure du graphe.

---

 Important, nous n'avons pas réussi à mettre en place le dernier algorithme 3Col est NP-complet.

---

Lien du repo Github :

[https://github.com/Yuss9/TP2\\_ANALYSE\\_ALGO](https://github.com/Yuss9/TP2_ANALYSE_ALGO)