ECOLE NATIONALE
SUPÉRIEURE
D'INFORMATIQUE

الجـــمهوريـــة الجزائريـــة الديمـــقراطيـــة الشـعبيـة
وزارة التعليـــم العـــالي والبـــحث العلمـــي
People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

Optimisation Combinatoire

2nd Year Specialty SIQ G02

Titre:

Heuristiques

Réalisé par :

Bayadh Hadjer

Rahal Nour Elhouda

Medfouni Khitem

Nakib ibtihel

Yazi Lynda Mellissa

Benmachiche Khaled

1. Introduction

Dans le cadre de l'optimisation combinatoire , les heuristiques sont des méthodes utilisées pour trouver des solutions satisfaisantes à des problèmes complexes où les approches exactes deviennent impraticables en raison de la taille du problème ou du temps de calcul requis. Ces techniques offrent des solutions approximatives mais efficaces en un temps raisonnable.

On distingue principalement deux types d'heuristiques en optimisation combinatoire :

1. Les heuristiques constructives :

- Elles génèrent une solution en construisant progressivement une solution valide selon une règle prédéfinie.
- Exemples : l'algorithme glouton (greedy), l'insertion séquentielle, ou encore l'algorithme de Prim pour le problème de l'arbre couvrant minimal.

2. Les heuristiques d'amélioration (ou de recherche locale) :

- Elles partent d'une solution existante et cherchent à l'améliorer en explorant les solutions voisines.
- Exemples : la recherche tabou, le recuit simulé, ou encore les algorithmes génétiques.

Ces heuristiques sont souvent utilisées lorsqu'un compromis entre qualité de la solution et temps de calcul est nécessaire, notamment dans des problèmes NP-difficiles comme le voyageur de commerce (TSP) ou l'affectation de ressources et coloration .

2. Les types des heuristiques :

2.1. Heuristique par construction :

WelshPowell :

- **Approche 1 :**
  - Tri des sommets par degré décroissant de leur de degré
  - **Coloration :**
    - Prendre le premier sommet non coloré et lui attribuer la couleur actuelle.
    - Parcourir les autres sommets non colorés et leur attribuer la même couleur s'ils ne sont pas adjacents aux sommets déjà colorés avec cette couleur.
  - **Changement de couleur :** Si aucun autre sommet ne peut être coloré avec la couleur actuelle, passer à une nouvelle couleur (+1).

  - Répétition du processus jusqu'à ce que tous les sommets soient colorés.

- **Approche 2 :**
  - Tri des sommets par degré décroissant avant la coloration.
  - Affectation de la première couleur disponible à chaque sommet non encore coloré.
  - Vérification de la validité de la coloration après l'exécution de l'algorithme.

- **Comparaison entre les deux approches :**

Nombre de nœuds	Algorithme	Nombre de couleurs	Temps (s)
100	Algo 1	7	0.0463
	Algo 2	6	0.0033
250	Algo 1	71	2.09
	Algo 2	70	0.024
500	Algo 1	18	1.8279
	Algo 2	18	0.08
450	Algo 1	31	2.3683
	Algo 2	30	0.06
1000	Algo 1	29	8.4530
	Algo 2	29	0.34

Table 1: Comparaison des deux algorithmes en fonction du nombre de nœuds

- **Remarques :**
  - **Nombre de couleurs utilisées :**
    * L'algorithme 2 utilise généralement moins de couleurs que l'algorithme 1, sauf pour le cas des 1000 et 500 nœuds où les deux algorithmes donnent le même résultat
    * Cela indique que l'algorithme 2 est potentiellement plus efficace en termes d'optimisation de la coloration.
  - **Temps d'exécution :**
    * L'algorithme 2 est beaucoup plus rapide que l'algorithme 1 pour toutes les tailles de graphes.
    * La différence de temps devient très significative pour les grands graphes (ex. : 1000 nœuds).
  - **Conclusion :**
    * L'algorithme 2 semble mieux optimiser la répartition des couleurs tout en étant plus efficace en termes de temps de calcul.
    * L'algorithme 1 reste correct mais est plus lent et utilise parfois plus de couleurs.

D-Satur :

- **DSatur avec tas binaire (heapq)** :
  - Utilise une file de priorité (tas binaire, heapq) pour gérer les sommets triés par :
    - Degré de saturation (nombre de couleurs distinctes utilisées par les voisins)
    - Degré du sommet (nombre total de voisins)
    - Index du sommet (en cas d'égalité)
  - Sélectionne en priorité le sommet avec le plus haut degré de saturation.
  - Met à jour dynamiquement les degrés de saturation des voisins après chaque coloriage.
- **DSatur avec arbre rouge-noir (SortedDict)** :
  - Utilise un arbre équilibré (SortedDict) au lieu d'un tas (heapq) et permet un accès direct au sommet avec la plus grande priorité en O(1).
  - Mise à jour plus efficace : Retrait et réinsertion des sommets après chaque coloration. Complexité
  - Insertion/Suppression dans SortedDict : O(logn).
  - Accès au sommet avec priorité max : O(1).
  - Globalement : O(nlogn), légèrement plus rapide que la version heapq mais gourmand en memoire.
- **DSatur avec Buckets (optimisé) :** Il remplace heapq et SortedDict par une structure de Buckets : Chaque indice du tableau représente un degré de saturation
  - Accès direct au sommet avec le plus grand degré de saturation en O(1).
  - Pas besoin de mise à jour dynamique de la structure (contrairement au tas ou à l'arbre). Structure du code .
  - Tableau de buckets (buckets) : buckets[i] contient les sommets avec un degré de saturation i. Quand un sommet est colorié, il est retiré de son bucket.
  - Tableau NumPy pour les degrés/saturation : Plus rapide que les listes Python.
  - Complexité Accès au sommet prioritaire : O(1). Mise à jour après chaque coloration : O(1). Globalement : $O(n^2)$ mais plus rapide en pratique. Il est rapide en pratique sur les grands graphes mais peut être moins efficace pour les graphes très denses.
- **Comparaison entre les trois versions :**

Nombre de nœuds	Algorithme	Nombre de couleurs	Temps d'exécution (s)
100	Algo 1	6	0.0061
	Algo 2	6	0.0065
	Algo 3	6	0.0089
250	Algo 1	68	256.723
	Algo 2	68	311.842
	Algo 3	64	81.54
500	Algo 1	16	0.622
	Algo 2	16	0.727
	Algo 3	16	0.077
450	Algo 1	28	4.902
	Algo 2	28	6.636
	Algo 3	29	0.099
1000	Algo 1	27	5.519
	Algo 2	27	5.939
	Algo 3	27	0.264

Table 2: Comparaison des trois algorithmes de coloration de graphe

- **Remarques :**
  - **Nombre de couleurs utilisées :**
    - Pour la plupart des cas, les trois algorithmes produisent le même nombre de couleurs.
    - Une exception est observée pour le graphe à **250 nœuds**, où **Algo 3** utilise 64 couleurs au lieu de 68.
    - Une autre différence apparaît pour **450 nœuds**, où **Algo 3** utilise 29 couleurs contre 28 pour les autres.
    - Cela suggère que **Algo 3** peut parfois produire une solution légèrement différente, soit en réduisant, soit en augmentant le nombre de couleurs.
  - **Temps d'exécution :**
    - Pour les petits graphes (100 nœuds), les trois algorithmes ont des temps d'exécution très proches.
    - Pour les grands graphes, **Algo 3** est le plus rapide, en particulier pour **250, 450 et 500 nœuds**, où il est nettement plus efficace que les autres.
    - À **250 nœuds**, **Algo 3** est environ **3 fois plus rapide** qu'Algo 1 et **près de 4 fois plus rapide** qu'Algo 2.
    - À **500 et 450 nœuds**, **Algo 3** reste le plus performant, tandis que **Algo 2** est systématiquement le plus lent.
    - Pour **1000 nœuds**, bien que les trois algorithmes utilisent le même nombre de couleurs, **Algo 3** est environ **20 fois plus rapide** qu'Algo 1 et Algo 2.
  - **Conclusion :**

    - Pour les petits graphes (100 nœuds), les différences de performances sont négligeables.
    - Pour les grands graphes, **Algo 3** est de loin le plus rapide, mais il peut modifier légèrement le nombre de couleurs dans certains cas.
    - **Algo 1** et **Algo 2** garantissent des résultats plus stables en termes de coloration mais sont beaucoup plus lents sur les grandes instances.
    - **Algo 3** semble être le meilleur choix en termes de rapidité, avec une attention particulière aux variations de couleurs qu'il peut introduire.

Recursive Largest First :

- **Initialisation :** Tous les sommets sont considérés comme non colorés. On commence avec la première couleur color = 0.
- **Sélection du sommet initial :** Choisir le sommet avec le plus grand degré dans le graphe actuel.
- **Construction d'un ensemble indépendant :**
  - Initialiser un ensemble independent_set avec le sommet sélectionné. Ajouter récursivement les sommets qui ne sont pas adjacents aux sommets déjà dans l'ensemble.
  - À chaque itération, choisir un sommet avec le plus petit nombre de voisins dans les candidats restants pour maximiser la taille de l'ensemble.
  - **Coloration de l'ensemble :**
  - Tous les sommets de independent$_s et reoivent la couleur actuelle. Retirer ces sommets du graphe.$
  - **Répétition :**
    - Augmenter l'indice de couleur et recommencer le processus jusqu'à ce que tous les sommets soient coloriés.

Hybrid Graph Coloring Algorithm: DSATUR + RLF :

Cet algorithme combine DSATUR (degré de saturation) et RLF (Recursive Largest First) pour colorer efficacement un graphique tout en minimisant le nombre de couleurs utilisées. Étapes de l'algorithme :

- **Sélectionner le nœud de départ en utilisant la stratégie DSATUR :** Choisir le nœud non coloré ayant le degré de saturation le plus élevé (c'est-à-dire le nombre de couleurs différentes utilisées par ses voisins). Si plusieurs nœuds ont la même saturation, choisir celui qui a le degré le plus élevé (le plus grand nombre de voisins).
- **. Construire un ensemble indépendant à l'aide de la méthode RLF :**
  - Tous les nœuds de l'ensemble reçoivent la même couleur.
  - Les retirer de la liste des nœuds non colorés.
  - Mettre à jour le degré de saturation de leurs voisins.
- Répétez l'opération jusqu'à ce que tous les nœuds soient colorés.

Comparaison entre les trois algorithmes :

Nombre de nœuds	Algorithme	Nombre de couleurs	Temps d'exécution (s)
100	RLF	6	0.003
	DSATUR + RLF	6	0.0041
250	RLF	73	0.027
	DSATUR + RLF	75	0.035
500	RLF	17	0.133
	DSATUR + RLF	18	0.143
450	RLF	35	0.133
	DSATUR + RLF	36	0.149
1000	RLF	28	1.06
	DSATUR + RLF	27	1.41

Table 3: Comparaison les deux algorithmes de coloration de graphe

Remarques :

- **Nombre de couleurs utilisées :**
  - Pour les petites instances (**100 nœuds**), les deux algorithmes donnent exactement le même résultat (**6 couleurs**).
  - Pour des graphes plus grands, RLF a tendance à utiliser un nombre légèrement inférieur de couleurs par rapport à **DSATUR + RLF. - Exemple : à **250 nœuds**, RLF utilise **73 couleurs**, contre **75** pour DSATUR + RLF. - De même, à **500 nœuds**, RLF utilise **17 couleurs**, tandis que DSATUR + RLF en utilise **18**.
  - Une exception est observée à **1000 nœuds**, où DSATUR + RLF trouve une solution légèrement meilleure avec 27 couleurs, contre 28 pour RLF.
- **Temps d'exécution :**
  - RLF est plus rapide que DSATUR + RLF dans toutes les instances testées.
  - L'ajout de DSATUR augmente légèrement le temps de calcul, ce qui est visible pour les grandes instances : - À 1000 nœuds, RLF prend 1.06s, tandis que DSATUR + RLF prend 1.41s. - À 500 nœuds, RLF exécute en 0.133s, contre 0.143s pour DSATUR + RLF. - Pour les petites tailles (100 et 250 nœuds), la différence est plus faible mais reste notable.
- **Conclusion :**
  - RLF est plus rapide et donne globalement de bons résultats en termes de minimisation des couleurs.
  - DSATUR + RLF peut parfois offrir de meilleures solutions (comme à 1000 nœuds), mais au prix d'un temps de calcul légèrement plus élevé.

- Si l'objectif est la rapidité, RLF est le meilleur choix.
- Si l'on cherche à optimiser la coloration pour certains graphes spécifiques, DSATUR + RLF peut être une option intéressante.

2.2. Choisir le meilleure algorithme constructives :

Comparaison entre DSATUR + RLF et Algo3 de D-SATUR et Algo2 de welshpowell

Comparaison entre les trois algorithmes

L'analyse des résultats obtenus permet de comparer les performances des trois algorithmes en fonction du nombre de couleurs utilisées et du temps d'exécution.

Concernant le nombre de couleurs, l'algorithme **Algo 3** est le plus performant, notamment pour des graphes de 250 et 450 nœuds, où il utilise respectivement 64 et 29 couleurs, contre 75 et 36 pour **DSATUR + RLF**. Pour 1000 nœuds, **DSATUR + RLF**, **Algo 3** et **Algo 2** trouvent tous une solution avec 27 couleurs, tandis que **RLF** seul en utilise 28. Ainsi, **Algo 3** minimise le nombre de couleurs de manière plus efficace dans la plupart des cas.

En termes de temps d'exécution, **Algo 3** est également le plus rapide. Par exemple, pour 1000 nœuds, il effectue la coloration en seulement 0.264 secondes, contre 1.41 secondes pour **DSATUR + RLF** et 0.34 secondes pour **Algo 2**. Cette tendance se vérifie sur toutes les tailles de graphes, où **Algo 3** surpasse systématiquement les autres algorithmes en rapidité.

Ainsi, **Algo 3** apparaît comme le meilleur choix global, offrant à la fois une réduction du nombre de couleurs et un temps d'exécution optimisé. **DSATUR + RLF** reste un bon compromis en termes de minimisation des couleurs, bien qu'il soit plus lent. Enfin, **Algo 2** offre une alternative plus rapide que **DSATUR + RLF**, mais légèrement moins performante en minimisation des couleurs.

2.3. Heuristique par amelioration :

Algorithme 1 :

Cet algorithme combine DSATUR (Degree of Saturation) et une heuristique d'amélioration pour colorier un graphe tout en minimisant le nombre de couleurs utilisées.

Étapes de l'algorithme :

- **Initialisation du coloriage avec DSATUR ou Max Degree :**

  - **DSATUR :** Sélectionne le nœud non colorié ayant le degré de saturation le plus élevé (c'est-à-dire le plus grand nombre de couleurs différentes utilisées par ses voisins). En cas d'égalité, choisit le nœud avec le plus haut degré (le plus de voisins).
  - **Max Degree :** Trie les nœuds par degré décroissant (nombre de voisins) et leur assigne la plus petite couleur disponible.

- **Attribution des couleurs de manière gloutonne :** Pour chaque nœud sélectionné, on choisit la plus petite couleur disponible qui ne crée pas de conflit avec ses voisins déjà coloriés. On garde en mémoire le nombre total de couleurs utilisées.

- **Application d'une heuristique d'amélioration :** On parcourt chaque nœud et on tente de réduire sa couleur tout en gardant un coloriage valide.
  Si un nœud peut être colorié avec une couleur plus basse sans conflit, on met à jour sa couleur.
  Ce processus est répété jusqu'à ce qu'aucune amélioration supplémentaire ne soit possible.

Algorithme 2 :

- **L'initialisation :** est la même que dans l'algorithme précédente
- **Recherche d'une meilleure couleur pour chaque nœud :**
  - Pour chaque nœud du graphe, l'algorithme teste s'il peut être colorié avec une couleur plus petite sans provoquer de conflit avec ses voisins.
- **Fusion de couleurs (Color Merging) :**
  - Il vérifie si deux couleurs distinctes peuvent être fusionnées sans créer de conflits.
- **Répétition jusqu'à convergence :**
  - Les deux étapes précédentes sont répétées tant qu'une amélioration est possible

Algorithme 3 :

Cet algorithme repose sur l'utilisation de DSATUR et de RLF pour initialiser la solution, suivie d'une phase d'optimisation qui applique différentes stratégies de recherche locale et de perturbation pour améliorer le coloriage. **Étapes de l'algorithme :**

- **L'initialisation :**
  - **DSATUR :** Sélectionne et colore les sommets en fonction de leur degré de saturation.
  - **RLF :** Forme des sous-ensembles indépendants et les colore successivement.(comme deja expliquer)

  - **Sélection de la meilleure solution :** Compare DSATUR et RLF et choisit la solution utilisant le moins de couleurs.

- **Amélioration de la Coloration**
  - **Réduction locale des couleurs :** consiste à essayer d'attribuer une couleur plus petite à un sommet déjà colorié, sans provoquer de conflit avec ses voisins.
  - **Permutation des couleurs :** Échange des couleurs entre des ensembles de sommets indépendants (où tous les sommets d'un même ensemble ont la même couleur). Si aucun conflit n'est détecté, la couleur du premier ensemble est conservée pour les deux.
  - **Perturbation :** Applique des modifications aléatoires pour éviter les minima locaux.

- **Vérification et Critères d'Arrêt :**
  - Le nombre cible de couleurs est atteint.
  - Le nombre maximal d'itérations est atteint : il est fixé en fonction de la densité du graphe. Plusieurs tests sont effectués avec différentes valeurs d'itérations, et on choisit un seuil optimal lorsque aucune nouvelle amélioration n'est observée après un certain nombre d'itérations.

Comparaison entre les deux heuristiques par amélioration

Nombre de nœuds	Algorithme	Approche	Nombre de couleurs	Temps d'exécution
100	Algo 1	Approche 1	5	0.01
		Approche 2	5	0.0103
	Algo 2	-	5	0.0065
	Algo 3	-	5	0,02
250	Algo 1	Approche 1	66	0.297
		Approche 2	66	0.30
	Algo 2	-	66	0.04
	Algo 3	-	66	4
500	Algo 1	Approche 1	17	0.30
		Approche 2	17	0.324
	Algo 2	-	17	0.053
	Algo 3	-	16	1,48
450	Algo 1	Approche 1	28	0.3864
		Approche 2	28	0.3994
	Algo 2	-	28	0.051
	Algo 3	-	28	1,18
1000	Algo 1	Approche 1	28	1.96
		Approche 2	28	1.78
	Algo 2	-	28	0.36
	Algo 3	-	26	4,79

Table 4: Résultats des algorithmes pour différents nombres de nœuds

- **Remarques :** En se basant uniquement sur les résultats du tableau, voici la comparaison :
  - **Comparaison du nombre de couleurs utilisées**
    - Pour tous les cas de figure (100, 250, 450 et 1000 nœuds), l'Algorithme 1 (approche 1 et 2) ainsi que l'Algorithme 2 génèrent exactement le même nombre de couleurs.
    - L'Algorithme 3 se distingue uniquement pour 500 et 1000 nœuds, où il utilise une couleur de moins que les deux autres (16 au lieu de 17 pour 500 nœuds et 26 contre 28 pour 1000 nœuds).
    - Ainsi, en termes de qualité de solution (nombre de couleurs), l'Algorithme 3 semble être légèrement plus performant pour les graphes de plus grande taille.
  - **Temps d'exécution :** L'analyse des temps d'exécution révèle des écarts significatifs :
    - L'Algorithme 2 est systématiquement le plus rapide, prenant beaucoup moins de temps que l'Algorithme 1 et surtout l'Algorithme 3.

    - L'Algorithme 3 est le plus lent, en particulier pour les grandes tailles de graphes. Par exemple, pour 500 nœuds, l'Algorithme 3 prend 1.48 s contre 0.053 s pour l'Algorithme 2.
    - L'Algorithme 1 est plus lent que l'Algorithme 2, avec un temps d'exécution entre 3 et 5 fois plus long, et encore plus lent pour les grands graphes.

- **Conclusion :**
  - En termes de nombre de couleurs, l'Algorithme 1 et l'Algorithme 2 donnent systématiquement des solutions de même qualité, tandis que l'Algorithme 3 est parfois légèrement plus performant (500 et 1000 nœuds).
  - Concernant le temps d'exécution, l'Algorithme 2 est le plus rapide, surpassant largement l'Algorithme 1 et surtout l'Algorithme 3.
  - L'Algorithme 3, bien qu'il utilise moins de couleurs pour les grands graphes, est significativement plus lent que l'Algorithme 2. Cela peut poser problème dans des contextes nécessitant une exécution rapide.
  - L'Algorithme 3, bien qu'il utilise moins de couleurs pour les grands graphes, est significativement plus lent que l'Algorithme 2. Cela peut poser problème dans des contextes nécessitant une exécution rapide.
  - Si l'objectif est la rapidité, alors l'Algorithme 2 est clairement le plus efficace, avec un temps d'exécution nettement inférieur aux autres algorithmes.

2.4. comparaison des algorithme algorithme 2 par amélioration et algorithme 3 D-Satur avec brunch and bound :

Comparaison entre les trois algorithmes :

Le tableau met en évidence la performance des trois algorithmes sur différentes tailles de graphes.

- **Algo 2 (amélioré)** : Il est **le plus rapide**, avec des temps d'exécution très courts, même pour des graphes de 1000 nœuds.
- **Algo 3 (D-Satur)** : Il est **plus lent** que l'Algo 2, mais il trouve souvent **une solution avec moins de couleurs**.
- **Branch and Bound** : Il est **très lent**, devenant impraticable pour des graphes au-delà de 100 nœuds.

Conclusion :

- Si la priorité est la rapidité, **Algo 2** est le meilleur choix.
- Si l'objectif est d'utiliser le moins de couleurs possible, **Algo 3 (D-Satur)** est préférable.

Nombre de nœuds	Algorithme	Nombre de couleurs	Temps d'exécution (s)
100	Algo 2	5	0.0065
	Algo 3	6	0.0089
	Branch and Bound	5	4291.9065
250	Algo 2	66	0.04
	Algo 3	65	81.54
	Branch and Bound	65	lent
500	Algo 2	17	0.053
	Algo 3	16	0.077
	Branch and Bound	12	lent
450	Algo 2	28	0.051
	Algo 3	29	0.099
	Branch and Bound	25	lent
1000	Algo 2	28	0.36
	Algo 3	27	0.264
	Branch and Bound	20	lent

Table 5: Comparaison des algorithmes (Algo 2, Algo 3, et Branch and Bound)

- **Branch and Bound** n'est viable que pour des graphes très petits (100 nœuds) parce que le temps d'éxécution de brunch and bound prend bouceaup de temps (des heures) .

Recommandation finale : Algo 3 (D-Satur) offre un bon équilibre entre optimalité et rapidité, tandis qu'Algo 2 est plus efficace pour les très grands graphes.