

Coloration de Graphes Géants

Application des algorithmes DSATUR glouton, Branch and Bound et LNS

ANEFAOUI Anouar, BENHALLOUK Oumaima,
CHAUVIN Téo, REMOND Etienne

9 janvier 2023

Sommaire

1	Introduction	3
2	Traitement des instances	4
2.1	Conversion des graphes en graphes d'intersections	4
2.2	Format de stockage et sauvegarde des graphes d'intersections	5
3	Une heuristique efficace : DSATUR glouton	6
3.1	Principe général	6
3.2	Utilisation d'un tas binaire modifié	7
3.3	Application de l'algorithme	8
4	Un algorithme exact : DSATUR Branch and Bound	9
4.1	Principe et limites	9
4.2	Implémentation	10
5	Des recherches locales : Large Neighborhood Search	11
5.1	Principe et implémentation	11
5.2	Exploration de l'arbre de recherche	12
6	Méthodologie et conclusion	14

1 Introduction

Les problèmes de coloration de graphes sont des problèmes d'optimisation combinatoire appartenant au champ de la théorie des graphes, ils consistent à attribuer une couleur à chaque noeud de manière à ce que deux noeuds adjacents soient colorés différemment. L'objectif est d'obtenir une coloration en un minimum de couleurs, chaque graphe G possède une coloration optimale en $\chi(G)$ couleurs, le nombre chromatique. La résolution de tels problèmes a un immense intérêt pratique, parmi eux l'attribution des fréquences en télécommunication, la résolution de conflits aériens, ou encore la planification et l'ordonnancement de tâches incompatibles. Il existe différentes classes d'algorithmes qui permettent d'obtenir une coloration valable optimale ou approchée : en particulier nous traiteront une méthode gloutone (DSATUR), une méthode exacte (DSATUR Branch and Bound) et une méthode par voisinnage (Large Neighborhood Search).

Notre travail est à l'origine The CG :SHOP Challenge 2022 [3] qui ambitionnait de trouver des méthodes performantes pour les problèmes de coloration de graphes planaires. Nous nous sommes limité à l'application de méthodes connues, citées plus haut, sur des graphes comportant un grand nombre de noeuds (entre 2500 et 75000) et dont la densité peut être élevée. Trouver le nombre chromatique d'un graphe est un problème NP-difficile dans le cas général, ce qui signifie qu'il n'existe a priori pas d'algorithmes polynomiaux ($P \neq NP$). On comprend alors l'intérêt pratique des méthodes approchées comme DSATUR ou LNS.

Les instances que l'on utilise ont été générées avec des processus variés [4] afin d'éviter les biais liés à la structure des graphes. Voici quelques exemples sélectionnés parmi les 225 instances proposées.

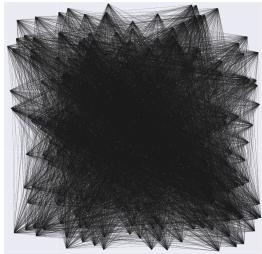


FIGURE 1 – rsqrp7320

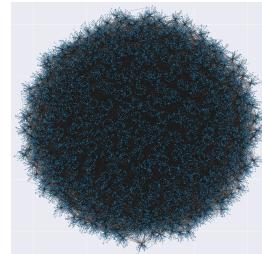


FIGURE 2 – reecn12588



FIGURE 3 – visp31334

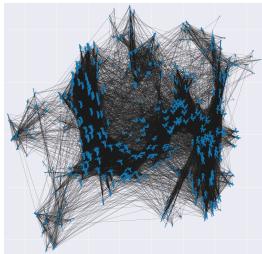


FIGURE 4 – rvispecn13421

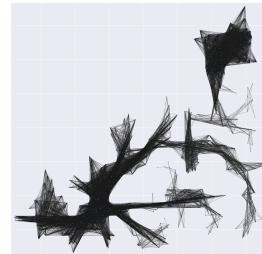


FIGURE 5 – rvisp10374



FIGURE 6 – sqrpecn62891

Dans un premier temps nous allons détailler le traitement des instances et le format sous lequel nous avons choisi de représenter les graphes. En effet les instances de graphes fournies ne sont pas directement les graphes d'intersections, il faut réaliser une conversion. Dans un second temps nous allons présenter nos implémentations des algorithmes dans l'ordre suivant : DSATUR, DSATUR Branch and Bound et LNS. Chacun de ces algorithmes utilisant le précédent, on enrichi ainsi progressivement notre méthode de coloration.

2 Traitement des instances

2.1 Conversion des graphes en graphes d'intersections

Comme précisé en introduction, il est nécessaire de convertir les graphes fournis par le challenge en graphes d'intersections. Vis à vis du challenge, ce sont ces derniers qu'il s'agit de colorer. La conversion consiste, pour chaque segment, à trouver tous les segments sécants. Cela permet de construire le graphe d'intersections car on peut alors donner pour chaque noeud la liste de ses adjacents.

Les données de départ nous sont fournies dans le format json, en voici un exemple représentant la manière dont ces graphes sont stockés.

```
1 {"type": "Instance_CGSHOP2022", "id": "exemple", "meta": {}, "n": 6, "m": 8, "x": [2, 3, 4, 5, 3, 1], "y": [3, 4, 3, 2, 1, 2], "edge_i": [0, 1, 0, 3, 2, 1, 0, 1], "edge_j": [2, 5, 4, 5, 4, 3, 3, 4]}
```

On peut ainsi récupérer les coordonnées de chaque noeuds (listes x et y) ainsi que les paires des noeuds adjacents (listes edge_i et edge_j). La méthode que l'on a utilisé pour la conversion est la suivante : pour chaque paire de segment on calcule si oui ou non les deux segments s'intersectent. La fonction Counter clockwise (CCW) prend comme paramètres trois points A , B et C et regarde si le point C est à droite ou à gauche du segment $[A, B]$. Dans la fonction suivante on calcule si les segments s'intersectent. Si A et B sont respectivement à droite et à gauche du segment $[C, D]$ et que les points C et D sont respectivement à droite et à gauche du segment $[A, B]$ alors il y a intersection. En réalité il faut également éliminer le cas où l'un des triplets de points est aligné. Les pseudo-codes sont présentés ci-dessous :

Algorithm 1 Counter clockwise CCW

Require: $(A, B, C) \in (\mathbb{R}^2)^3$

```
let  $\overrightarrow{AB}$  and  $\overrightarrow{AC}$ 
 $D \leftarrow \det(\overrightarrow{AB}, \overrightarrow{AC})$ 
if  $D < 0$  then
    return  $-1$ 
else
    return  $1$ 
end if
```

Algorithm 2 Intersect two segments

Require: $s_1 = [A, B]; s_2 = [C, D]$ distinct points

```
if CCW(C,D,A) ≠ CCW(C,D,B) and CCW(A,B,C) ≠ CCW(A,B,D) then
    if triplets not alligned then
        return true
    end if
else
    return false
end if
```

Bien que suffisante pour notre problème en terme de complexité temporelle, cette méthode n'est pas la plus efficace car on teste deux fois chaque paire de segment. On aurait également pu envisager une amélioration de cet algorithme qui teste uniquement les segments qui peuvent s'intersecter en projetant les coordonnées des points sur l'axe des abscisses et des ordonnées, deux segments qui sont possiblement sécants verront leurs projections se superposer sur ces axes. En notant $m = |E|$, notre algorithme est en $\mathcal{O}(m^2)$.

2.2 Format de stockage et sauvegarde des graphes d'intersections

Pour les besoins de notre projet il nous a fallu représenter un graphe d'intersections dans la structure idoine pour le manipuler, et en particulier le colorer. Cela signifie que l'on doit pouvoir identifier chaque noeud, changer sa couleur, accéder à ses adjacents, son degré et son degré de saturation. Il faut se rappeler que chacune de ces opérations va être effectuée un grand nombre de fois sur un grand nombre de noeuds. Voilà pourquoi nous avons décidé de pré-calculer et stocker les degrés, les identifiants et les adjacents qui sont des invariants et d'initialiser une variable représentant le degré de saturation à 0 quand le graphe est vierge. Évidemment on représente également dans une variable la couleur que prend le noeud. Pour ce faire on crée un type produit qui stocke ces informations.

Plusieurs structures de données sont envisageables pour stocker les informations d'un graphe, notre attention s'est portée sur les *Array* et *Hashtbl* de OCaml. La raison étant que l'on va très régulièrement chercher à accéder aux informations d'un noeud, une simple liste nous obligeraient à la parcourir à chaque fois, ce qui est inenvisageable du point de vue de la complexité temporelle. En revanche les *Array* et les *Hashtbl* peuvent accéder à un élément en respectivement $\mathcal{O}(1)$ et $\mathcal{O}(1)$ en moyenne. En réalité ici, puisque la taille du graphe ne varie pas, il n'y a pas de problème de collisions des clés de hashage, ce qui nécessiterait alors de toutes les recalculer, on peut donc considérer qu'il s'agit d'un accès en $\mathcal{O}(1)$. Nous avons d'abord pensé qu'une table de hashage nous donnerait également plus de flexibilité, c'est pourquoi nous avons choisi cette dernière. On ne s'intéresse qu'aux complexités temporelles de l'accès à une donnée car on n'a pas besoin d'ajouter, supprimer ou fusionner des graphes entre eux au cours du processus de coloration.

Maintenant que nous avons converti notre graphe, il est intéressant de le sauvegarder dans un fichier pour pouvoir accéder au graphe d'intersection à notre guise sans avoir besoin de passer par le processus de conversion. Une première idée qui nous ait venu est d'effectuer cette sauvegarde en créant un objet json fidèle à la structure que l'on utilise pour la coloration. Après avoir essayé cette méthode nous avons constaté que le temps de traitement était bien trop long (environ 2min 30s pour un graphe de 3000 noeuds) et les fichiers json obtenus trop volumineux. C'est alors que nous avons eu l'idée de stocker directement notre objet *Hashtbl* dans un fichier binaire. C'est le choix que nous avons conservé étant donné que l'on obtenait des fichiers de tailles raisonnables, des temps de sauvegarde et d'accès au graphe quasi-nuls.

3 Une heuristique efficace : DSATUR glouton

DSATUR est un algorithme séquentiel de coloration par heuristique créé par Daniel Brélaz en 1979 [1]. L'heuristique en question est basée sur les degrés de saturation, c'est à dire le nombre de couleurs différentes utilisées par les voisins d'un noeud donné, c'est de là que l'algorithme tient son nom. DSATUR fournit une solution approchée de coloration d'un graphe $G = (V, E)$ connexe et non orienté en $\mathcal{O}(n^2)$ avec $n = |V|$.

3.1 Principe général

Le principe de l'algorithme DSATUR glouton est de systématiquement colorer le noeud qui subit le plus de contraintes de la part de ses voisins avec la couleur la plus basse possible. Cela nécessite de pouvoir rapidement trouver le noeud avec le degré de saturation le plus élevé et la couleur la plus basse. De plus, à chaque itération on doit mettre à jour les degrés de saturation des k noeuds adjacents. Le pseudo-code associé est présenté ci-dessous :

Algorithm 3 DSATUR glouton

Require: $G = (V, E)$: graph to color ; $C = \{\emptyset\}$: colored nodes

```

let  $v_{max\_degree} \in \bar{C}$ 
 $C \leftarrow \{v_{max\_degree}\}$  with color 1}  $\cup C$ 
while  $|C| < |V|$  do
    let  $v_{max\_dsat} \in \bar{C}$  {in case of equality, select the node of maximum degree}
    let  $c$  {lowest color possible for  $v_{max\_dsat}$ }
     $C \leftarrow \{v_{max\_degree}\}$  with color  $c\cup C$ 
end while
return  $G$ 
```

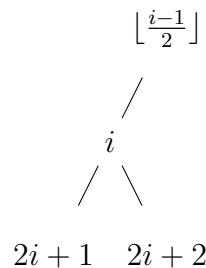
Avec notre manière de stocker le graphe, on peut colorer un élément en $\mathcal{O}(1)$ et vérifier si il est complètement coloré en testant simplement le nombre de coloration que l'on a effectué, puisqu'on colore un noeud donné qu'une seule fois. Naïvement, pour récupérer le noeud avec le degré de saturation le plus élevé il suffit de chercher le maximum dans la liste des noeuds non colorés, c'est une opération en $\mathcal{O}(|\bar{C}|)$. Pour trouver la couleur minimale disponible, il faut récupérer les couleurs des k noeuds adjacents et trier pour déduire la couleur la plus petite possible, c'est donc au mieux en $\mathcal{O}(k \log k)$. Finalement on effectue nécessairement n itérations. En faisant l'approximation que chaque noeud a en moyenne \hat{k} voisins, on a pour complexité :

$$T(n) = n\hat{k} \log(\hat{k}) + \sum_{i=0}^n i = \frac{n(n+1)}{2} + n\hat{k} \log(\hat{k}).$$

Pour des graphes très denses le nombre moyen de voisins s'approche du nombre de noeuds du graphe, ils sont du même ordre de grandeur alors la complexité devient $\mathcal{O}(n^2 \log(n))$. Mais dans le cas général on peut considérer que le nombre de voisins moyen d'un noeud est négligeable devant n . Il suit que la complexité est en $\mathcal{O}(n^2)$. Cette complexité peut être réduite en améliorant la manière dont on trouve le noeud de degré de saturation maximal, on introduit alors un tas-max annexe qui va stocker les noeuds non colorés ordonnés par degrés de saturation et par degrés.

3.2 Utilisation d'un tas binaire modifié

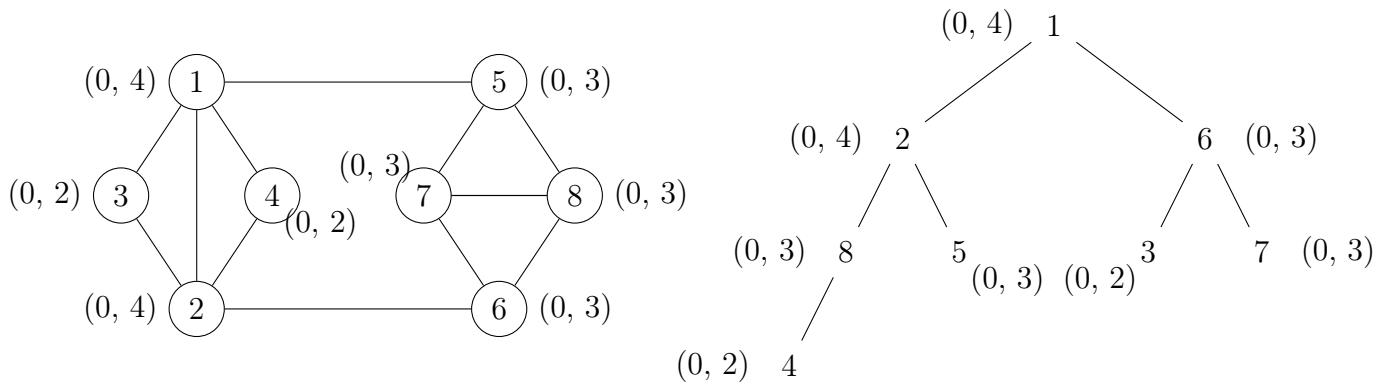
Le tas que nous avons implémenté n'est pas simplement un tas binaire, en effet cela suffirait pour extraire le noeud de degré de saturation maximal, mais en revanche nous serions obligé de reconstruire le tas à chaque actualisation des degrés de saturation, ce qui est une opération en $\mathcal{O}(|\bar{C}|)$. On ne réduit alors pas la complexité. Pour gagner du temps il faut connaître à tout instant la position d'un noeud donné dans le tas, ainsi actualiser sa position revient à effectuer des permutations dans le tas, ce qui est une opération en temps constant. Nous avons choisi de construire le tas dans un *Array*, nous aurions pu réaliser un type arbre binaire en OCaml mais il nous aurait été plus coûteux d'effectuer des permutations et d'accéder à ses éléments. Ainsi en partant d'un noeud à l'indice i du tableau on peut accéder à son fils droit et gauche avec les indices $2i + 1$ et $2i + 2$. On peut accéder au père d'un noeud d'indice i à l'indice $\lfloor \frac{i-1}{2} \rfloor$. Un autre *Array* permet de mémoriser les positions des noeuds dans le tas, les indices correspondent à l'indice du noeud et la valeur stockée correspond à la position de ce noeud dans le tas. Une permutation dans le tas doit être également effectuée dans le tableau d'association.



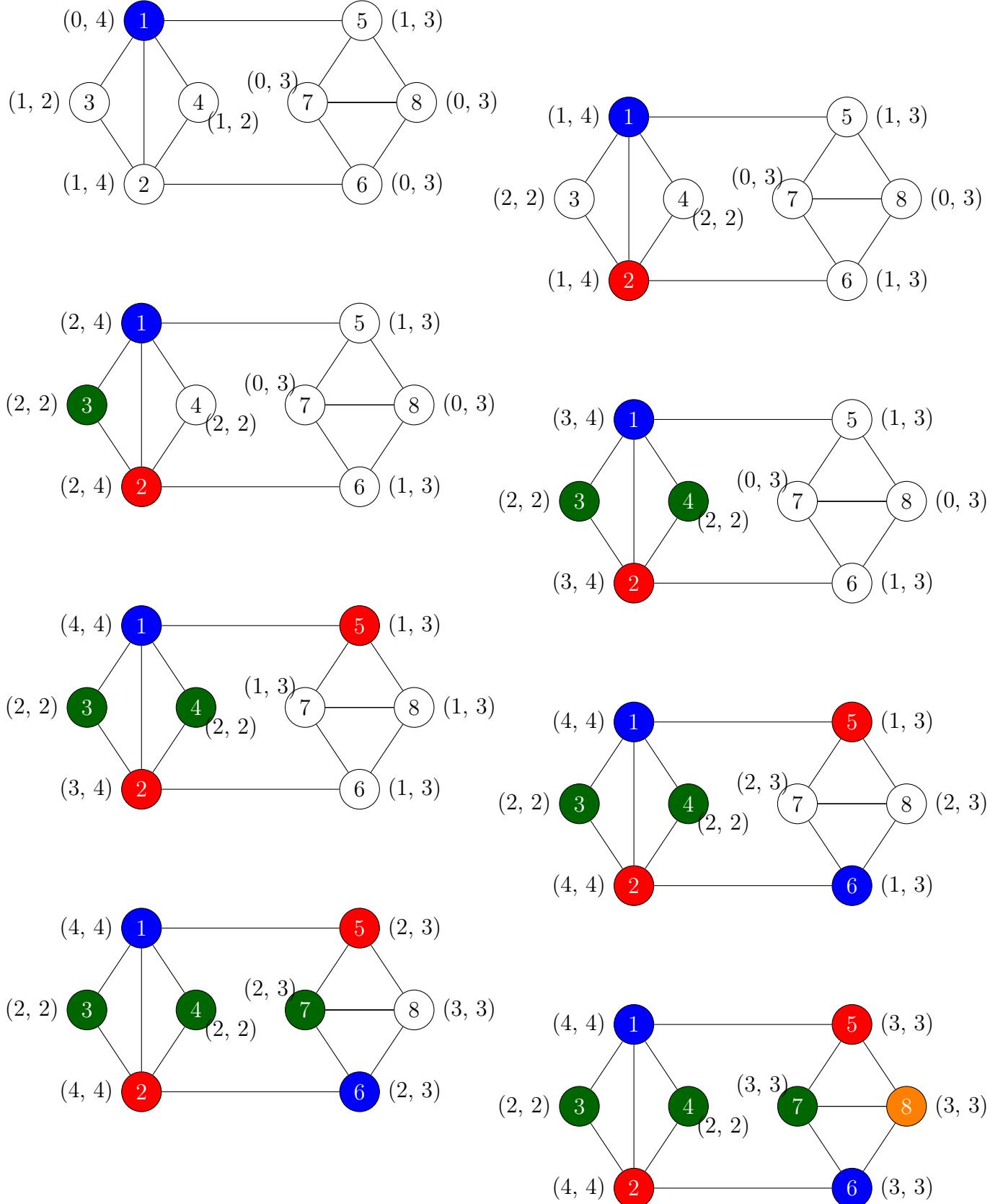
Désormais nous pouvons extraire le noeud de degré de saturation maximal en $\mathcal{O}(1)$, on peut actualiser le tas en temps constant. Pour fusionner les deux sous tas après avoir extrait la racine du tas, on injecte le dernier élément à la racine et on effectue les permutations adéquates jusqu'à équilibrer le tas, c'est une opération en $\mathcal{O}(\log_2(n))$. Ainsi

$$T(n) = n\hat{k} \log(\hat{k}) + \sum_{i=0}^n \log(n) = n \log(n) + n\hat{k} \log(\hat{k}).$$

La complexité de DSATUR glouton est en $\mathcal{O}(n \log(n))$ et approche $\mathcal{O}(n^2 \log(n))$ pour des graphes très denses. Voici deux représentations de ce à quoi ressemblent le tas et le graphe associé, suivi d'un exemple d'application de l'algorithme sur notre graphe sur lequel il trouve une coloration valide en 4 couleurs.



3.3 Application de l'algorithme

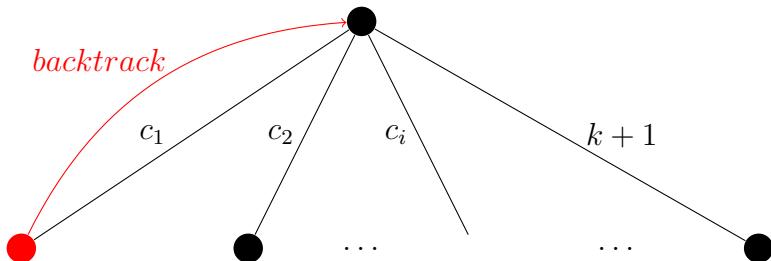


4 Un algorithme exact : DSATUR Branch and Bound

4.1 Principe et limites

DSATUR Branch and Bound [2] est un algorithme de coloration de graphes exact, son principe est simplement de colorer à chaque étape un noeud (non coloré) avec une couleur existante si possible, ou une nouvelle couleur sinon. Dès qu'une coloration atteint son terme, on garde en mémoire le nombre de couleurs nécessaires, on fait un retour sur traces dans nos étapes de coloration pour en essayer d'autres. La différence étant qu'au fur et à mesure que l'on améliore notre solution, l'algorithme va devenir plus sélectif sur les colorations qu'il va explorer. À terme l'algorithme n'aura plus d'options de coloration et renverra la meilleure solution trouvée, à savoir la coloration optimale.

Quand on parle de garder en mémoire la meilleure coloration, on décrit le concept de la borne supérieure UB . En effet l'algorithme Branch and Bound utilise des bornes supérieure et inférieure pour orienter ses décisions au sein de l'arbre de recherche. La borne inférieure LB décrit le nombre minimal de couleurs qu'il faudra pour colorer le graphe. Il s'agit idéalement de la taille de la clique maximum. La borne inférieure reste alors fixe durant l'exécution de l'algorithme. La borne supérieure est actualisée dès qu'une meilleure coloration est trouvée. Ces deux bornes encadrent la coloration optimale alors inconnue. Sans ce système de bornes, l'algorithme consisterait simplement à essayer toutes les colorations possibles et à retourner le meilleur résultat rencontré. Quoi qu'il en soit, l'algorithme Branch and Bound est un algorithme de complexité temporelle exponentielle, une limite évidente pour nous, cet algorithme est absolument inexploitable sur nos graphes de tailles importantes.



De la même manière que pour l'algorithme DSATUR glouton, lorsque l'on décide du noeud qui va être coloré, l'algorithme choisit le noeud de degré de saturation maximal ; c'est à dire celui sur lequel s'exerce le plus de contraintes. Le parcours de l'arbre de recherche s'effectue d'abord en profondeur, en colorant en priorité les noeuds qui ont le moins de degrés de liberté, on force l'algorithme à utiliser souvent des couleurs différentes. On se rend potentiellement plus vite compte qu'une coloration est mauvaise, car le nombre de couleurs utilisées dépassera plus vite la borne supérieure. La complexité temporelle moyenne s'en voit améliorée. Techniquement, on utilise encore une fois le tas binaire modifié pour y parvenir, la différence étant que lors du backtrack le noeud décoloré doit être replacé dans le tas. C'est une opération en $\mathcal{O}(\log_2(n))$.

Un des problèmes majeurs de DSATUR Branch and Bound est son incapacité à casser la symétrie des colorations, c'est à dire que pour une coloration à k couleurs données, il est possible de permute deux couleurs en gardant des colorations équivalentes. En d'autres termes, colorer un graphe en k couleurs est un problème équivalent à en trouver une k -partition. Pour une k -partition donnée il existe alors $k!$ colorations possibles, toutes présentes dans l'arbre de recherche. L'algorithme est alors susceptible d'explorer, au moins partiellement, $k!$ colorations équivalentes, ce qui représente une perte d'efficacité temporelle. L'intérêt d'une heuristique comme DSATUR est qu'elle casse cette symétrie en choisissant uniquement et à chaque étape la plus petite couleur possible.

4.2 Implémentation

Dans notre implémentation nous n'initialisons pas LB à la taille de la clique maximum mais plutôt à la taille de la première clique rencontrée lors de la coloration avec l'algorithme glouton. En effet, lors de l'exécution des premières itérations de ce dernier, les premiers noeuds colorés appartiennent tous à la même clique. Il est donc aisément de mettre en évidence cette clique. De plus, trouver la clique maximum dans un graphe est un problème de complexité exponentielle. La borne supérieure est quant à elle initialisée avec le nombre de couleurs distinctes $\kappa(G)$ utilisées lors de la coloration du graphe G par DSATUR glouton.

Les pseudocodes ci-dessous correspondent à l'algorithme DSATUR Branch and Bound et à sa boucle de récursion.

Algorithm 4 DSATUR_rec

Require: $G = (V, E)$: graph to color ; k : current coloration ; LB ; UB ; C : colored nodes

```

if  $G$  is fully colored then
    if  $k < UB$  then
         $UB \leftarrow k$ 
        update best solution
    end if
else
    let  $v_{max\_dsat} \in \bar{C}$  {in case of equality, select the node of maximum degree}
    for  $c$  in available colors plus one do
        if  $\max(LB, k) < UB$  then
             $C \leftarrow \{v_{max\_degree} \text{ with color } c\} \cup C$ 
             $DSATUR\_rec(G)$ 
            uncolor last colored node
        end if
    end for
end if
```

Algorithm 5 DSATUR Branch and Bound

Require: $G = (V, E)$: graph to color

```

 $G^* \leftarrow DSATUR(G)$ 
 $LB \leftarrow |F|$  {F is the first clique found while coloring with DSATUR}
 $UB \leftarrow \kappa(G^*)$ 
if  $LB=UB$  then
    return  $G^*$ 
else
     $G^* \leftarrow DSATUR\_rec(G^*)$ 
    return  $G^*$ 
end if
```

5 Des recherches locales : Large Neighborhood Search

5.1 Principe et implémentation

L'algorithme Large Neighborhood Search (LNS) est une heuristique de recherche utilisée en optimisation pour fournir des solutions approchées sur de grands espaces de recherche. Il fonctionne par itération, partant d'une solution initiale sous optimale qu'il améliore en explorant non pas tout l'espace de recherche mais un voisinage seulement. La notion de voisinage est à comprendre au sens de toutes les solutions proches d'une solution donnée. En considérant le problème équivalent du nombre de partition minimum d'un graphe, les solutions voisines sont celles qui réalisent presque les mêmes partitions. En d'autres termes, à chaque itération on définit notre voisinage, on l'exploré, et on réitère sur la meilleure solution trouvée. L'algorithme s'arrête avec un critère d'arrêt à définir, un nombre fini d'itérations par exemple. Cet algorithme est indépendant de la technologie utilisée, plusieurs interprétation d'un voisinage sont possibles, plusieurs algorithmes de recherche peuvent être utilisés sur les graphes partiellement colorés et d'autres critères d'arrêts sont envisageables. Notre tâche sera de préciser tous ces choix et leurs implémentations.

Nous utilisons l'heuristique de DSATUR glouton pour obtenir une première coloration approchée du graphe d'entrée G . Cela nous permet alors de définir les bornes LB et UB de la même manière que dans l'algorithme de DSATUR Branch and Bound . Dans notre cas, l'implémentation du tas est utile pour récupérer l'ordre avec lequel les noeuds ont été colorés. On se sert de cette donnée pour définir un voisinage \mathcal{N}_p de G dont p noeuds sont décolorés. Les noeuds que l'on décolore correspondent aux p derniers noeuds colorés. Il est intéressant de définir \mathcal{N}_p de cette manière étant donné qu'il s'agit des noeuds dont le degré de liberté est le plus important ; c'est-à-dire les noeuds dont la couleur est la plus susceptible de changer. Lors d'une itération de LNS , on applique DSATUR Branch and Bound à \mathcal{N}_p et nous réitérons avec le meilleur graphe renvoyé.

Algorithm 6 Large Neighborhood Search

Require: $G = (V, E)$: graph to color ; i_{limit} : number of iterations ; p : number of nodes to uncolor

```

 $G^* \leftarrow DSATUR(G)$ 
 $LB_{G^*} \leftarrow |F|$  {F is the first clique found while coloring with DSATUR}
 $UB_{G^*} \leftarrow \kappa(G^*)$ 
if  $LB_{G^*} = UB_{G^*}$  then
    return  $G^*$ 
else
    for  $i = 1$  to  $i_{limit}$  do
        let  $N_i \in \mathcal{N}_p(G^*)$  {Neighborhood}
         $\tilde{G} \leftarrow DSATUR\_rec(N_i)$ 
        if  $UB_{\tilde{G}} < UB_{G^*}$  then
             $G^* \leftarrow \tilde{G}$ 
        end if
    end for
    return  $G^*$ 
end if
```

5.2 Exploration de l'arbre de recherche

Nous sommes passé rapidement sur ce qu'était véritablement le voisinage d'un graphe. Il faut comprendre que nous considérons l'espace des graphes colorés avec au plus k couleurs, un voisinage correspond à l'ensemble des graphes à p décoloration du graphe initial. On peut donc affirmer que $|\mathcal{N}_p(G)| = \binom{n}{p}$. On se demande alors comment choisir un bon élément du voisinage, c'est à dire un graphe susceptible d'être recoloré avec moins de couleurs. Une première idée est de choisir aléatoirement l'un de ces graphes, cependant en procédant ainsi, les noeuds décolorés sont susceptibles d'avoir comme voisins uniquement des noeuds colorés. Cela revient à choisir des noeuds qui ont peu de degrés de liberté, ils seront alors recolorés de la même manière. Pour que cette méthode montre des résultats il faudrait probablement décolorer un nombre important de noeuds, chaque noeud décoloré augmentant exponentiellement le temps de calcul. On souhaite alors choisir des noeuds qui ont un nombre de degrés de liberté important, laissant dès lors à l'algorithme un grand nombre de configurations à explorer et donc une plus grande chance de trouver une meilleure solution. Pour ce faire nous utilisons l'ordre de coloration, les noeuds colorés en dernier seront ceux qui ont grand degré de liberté, de plus nos algorithmes choisissent de colorer les noeuds de grands degrés de saturation. Ils sont susceptibles de colorer des cliques, ce qui est particulièrement intéressant quand on sait qu'on recolore avec un algorithme exact.

N'oublions pas qu'à chaque itération il faut changer les paramètres, au choix : le nombre d'éléments à décolorer, les noeuds à décolorer, ou les deux. L'idée étant que si l'on décolore systématiquement les même noeuds, DSATUR Branch and Bound étant déterministe, nous obtiendrons toujours la même solution ; nous n'explorons alors pas correctement l'arbre de recherche et restons potentiellement bloqué sur une solution sous-optimale. En faisant varier p , on augmente le nombre de configurations à explorer, et en choisissant d'autres noeuds, par exemple en choisissant les noeuds suivants l'ordre de coloration, on explore une toute autre partie de l'arbre de recherche. La figure 7 montre la forme de notre arbre de recherche, la base du triangle représente les colorations complètes du graphe, et son sommet, le graphe vierge. On voit qu'en considérant un graphe complètement coloré G^* , on peut représenter l'ensemble des graphes à p décolorations $\mathcal{N}_p(G^*)$; parmi eux se trouve le graphe qui nous intéresse à l'itération i , noté ici N_i . La figure 8 illustre l'exploration de l'arbre de recherche par DSATUR Branch and Bound une fois notre graphe N_i initialisé. On visualise alors l'ensemble des colorations que l'on peut explorer. Si un graphe \tilde{G} est une meilleure solution, alors l'algorithme le trouvera. On remarque que G^* et \tilde{G} sont relativement proches au sens de l'exploration de l'arbre. On comprend désormais l'importance d'effectuer plusieurs itérations, de manière à trouver la meilleure solution possible.

L'objectif de LNS est d'explorer les colorations possibles, c'est-à-dire la base de l'arbre de recherche, de manière à trouver une bonne solution en s'économisant des calculs sur l'exploration de l'arbre. Cependant, si notre meilleure solution G^* est trop éloignée des potentielles meilleures solutions, on risque de rester bloquer sur cette solution sous optimale. On peut imaginer un algorithme adaptatif qui fait varier la taille du voisinage pour atteindre des solutions plus distantes.

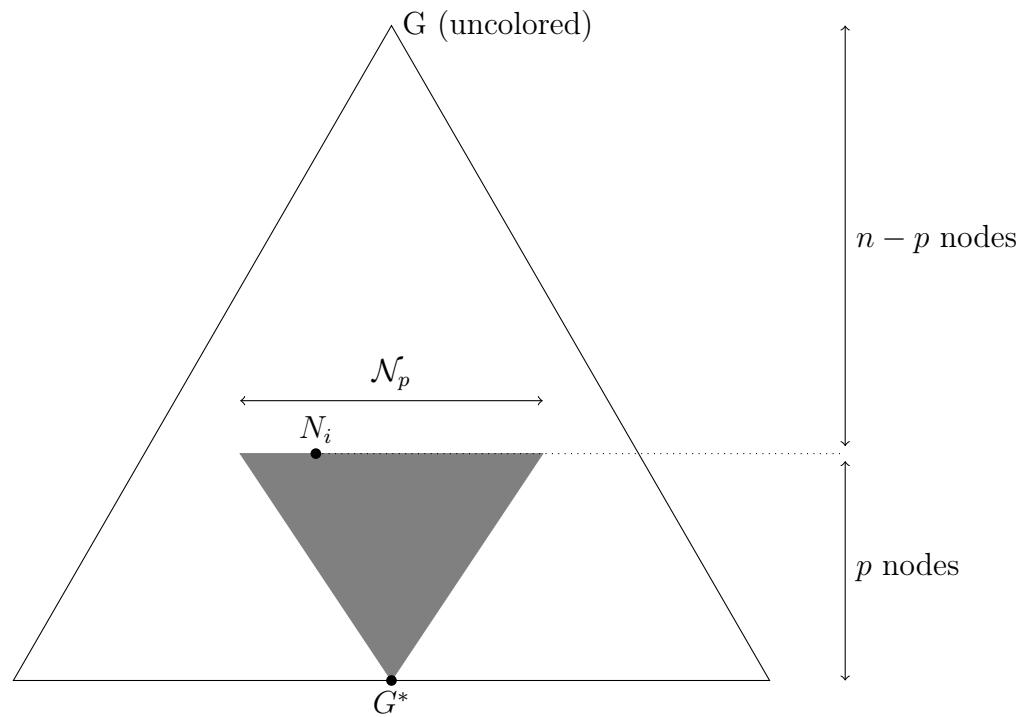
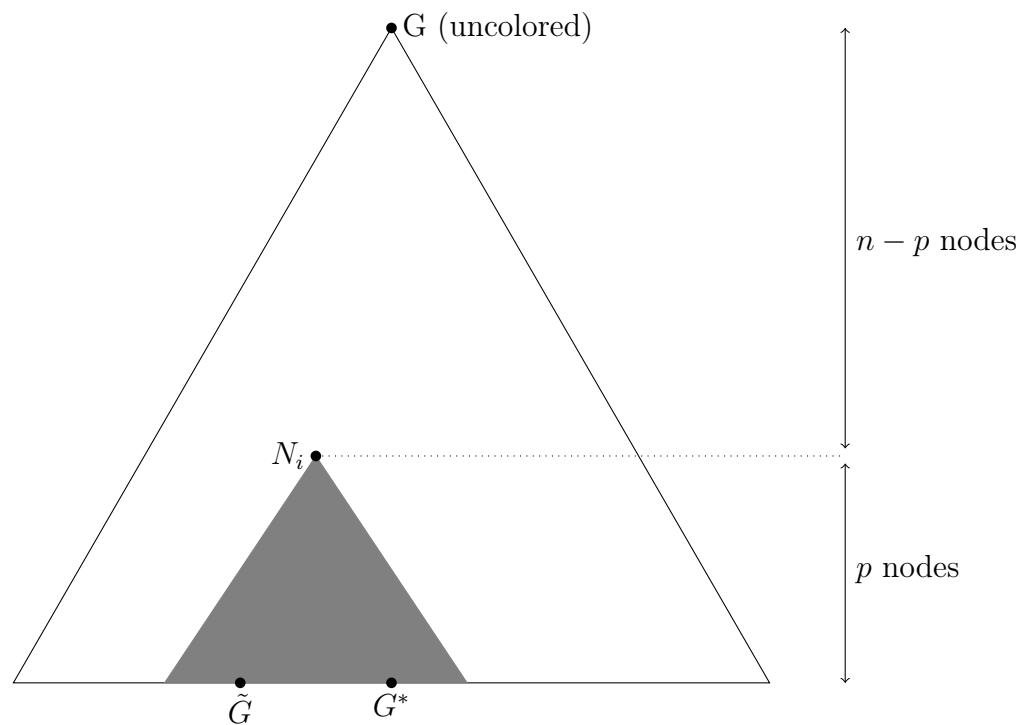


FIGURE 7 – Initialisation d'un voisinage

FIGURE 8 – DSATUR Branch and Bound sur N_i

6 Méthodologie et conclusion

Du point de vue de l'organisation de notre travail, nous avons débuté par mettre en place nos moyens de communication. Après la réunion de lancement avec notre référent Mr. Alexandre GONDTRAN, nous avons convenu de la mise en place d'un répertoire de travail Github pour la mise en commun de nos contributions aux codes. Nous restions en étroite communication durant le projet via une messagerie web, ainsi que diverses réunions en autonomie pour notre groupe.

La première réunion nous a permis de réaliser une première distribution des tâches. Une personne à alors été désignée comme chef de projet, une autre comme responsable de l'harmo-nisation du Github et de ses codes. Nous avons débuté par prendre en main le sujet ; c'est-à-dire une lecture approfondie de l'article [4] ainsi qu'un survol de l'état de l'art. Après avoir défini les méthodes de manipulations des graphes, chacun a du travailler sur une des tâches suivantes : la lecture des fichiers d'instances et la sauvegarde des graphes, la conversion des instances en graphes d'intersections, la généralisation du code via le foncteur, et une première implémentation des algorithmes de coloration.

La gestion des instances a représenté une première difficulté. En effet il nous a fallu trouver une manière efficace de stocker les graphes pour les manipuler. Nous nous sommes beaucoup questionné sur la forme que devait prendre la structure de données, nous avions des contraintes évidentes imposées par la taille des instances. C'est alors que nous avons fait le choix de gé-néraliser le module de gestion des graphes à l'aide d'un foncteur qui nous permettrait à terme de changer à notre guise la structure de données (*Hashtbl* ou *Array*). S'en sont suivies les réflexions concernant la sauvegarde des graphes que l'on détaillera en 2.1.

Notre première implémentation de DSATUR glouton n'étant pas optimisée et donc inefficace sur des instances de tailles conséquentes, notre référent nous a conseillé l'utilisation d'un tas pour récupérer les noeuds à colorer. L'implémentation de DSATUR Branch and Bond a été quant à elle été plus problématique. La difficulté résidait dans la mise en place du backtrack et la propagation des meilleures solutions. Étant donné que nous avions attaché de l'importance dans la généralisation des algorithmes précédemment cités, l'implémentation de LNS nous a semblé assez rapide.

Pour terminer, nous avons constaté, à l'usage, que DSATUR glouton donnait une approxima-tion correcte du nombre chromatique d'un graphe. En revanche, bien que DSATUR Branch and Bound soit un algorithme exact, il nous est impossible de l'appliquer à des instances aussi volumineuses. L'utilisation de LNS nous permet alors de corriger localement l'approximation faite par DSATUR. Nous pourrions envisager maintenant d'appliquer d'autres heuristiques à LNS pour réaliser un comparatif complet de leur efficacité. De la même manière nous pourrions implémenter d'autres méthodes comme la recherche locale de tabucol ou des algorithmes génétiques et / ou hybrides.

Références

- [1] Daniel BRÉLAZ. “New Methods to Color the Vertices of a Graph”. In : (1979).
- [2] Ian-Christopher Ternier FABIO FURINI Virginie Gabrel. “An improved DSATUR-based Branch-and-Bound algorithm for the Vertex Coloring Problem”. In : (2016).
- [3] Dominik Krupke Stefan Schirra SÁNDOR P. FEKETE Phillip Keldenich. *Minimum Partition into Plane Subgraphs : The CG :SHOP Challenge 2022*. 2022. URL : <https://cgshop.ibr.cs.tu-bs.de/competition/cg-shop-2022/#problem-description>.
- [4] Dominik Krupke Stefan Schirra SÁNDOR P. FEKETE Phillip Keldenich. “Minimum Partition into Plane Subgraphs : The CG :SHOP Challenge 2022””. In : (2022).