Intensification/Diversification VNS Application au Problème de Coloration de Graphe

Samir Loudni
GREYC (CNRS - UMR 6072) – Université de Caen
Boulevard du Maréchal Juin
14000 Caen
Samir.Loudni@unicaen.fr

Abstract

Étant donné un graphe non-orienté $G = (V, E)$, le problème de coloration consiste à déterminer le nombre minimal de couleurs nécessaires pour colorier les sommets de G de telle sorte que deux sommets adjacents n'aient pas la même couleur. Dans cet article nous présentons une extension de l'algorithme de base VNS, appelé `ID-VNS`, qui intègre des mécanismes simples pour réaliser un bon compromis entre intensification et diversification. Ensuite, nous montrons comment `ID-VNS` peut être appliqué avec succès au problème de coloration en définissant des voisinages génériques qui bénéficient à la fois des avantages des conflits et de la topologie du graphe de contraintes. Des expérimentations réalisées sur les instances difficiles de DIMACS montrent que notre méthode est très compétitive avec les meilleurs algorithmes connus de coloration.

1 Introduction

Étant donné un graphe non-orienté $G = (V, E)$, avec un ensemble V de sommets et un ensemble E d'arêtes, le *problème de coloration* des sommets d'un graphe (PCSG) consiste à déterminer le nombre minimum de couleurs (*nombre chromatique* $\chi(G)$) nécessaires pour colorier les sommets de G tel qu'il n'existe pas deux sommets adjacents de la même couleur. Déterminer une k-coloration est un problème NP-complet pour un nombre de couleurs $k \geq 3$ [13]. Ce problème a été largement étudié dans la littérature, non seulement pour ses applications directes dans des domaines variés, tels que l'allocation de registre [6], l'affectation de fréquences [12], les emplois du temps [4, 7] et l'ordonnancement [11, 19], mais aussi pour ses aspects théoriques et pour sa difficulté. En effet, bien que de nombreux algorithmes exactes ont été proposés pour ce problème [3, 5, 16, 24, 25], ces derniers ne peuvent résoudre que de petites instances (jusqu'à 100 sommets). Par conséquent, les méthodes heuristiques sont nécessaires pour traiter les grandes instances. Les approches heuristiques les plus performantes sont les méthodes de recherche locale (e.g., [1, 2, 17, 18, 27]) et les méthodes *hybrides à base de populations* (e.g. [8, 9, 22]).

La recherche à voisinages variables (VNS) [26] exploite l'idée de changer systématiquement de structure de voisinage, tant dans la descente aux minima locaux, mais aussi pour s'échapper des vallées qui les contiennent. Elle consiste en l'application successive d'une étape de *perturbation*, suivie d'une recherche locale. L'étape de perturbation joue un rôle primordial, car elle permet à VNS d'explorer différentes régions de l'espace de recherche, afin de s'échapper du bassin d'attraction des optima locaux (*effort de diversification*), alors que l'objectif de la recherche locale est de se concentrer plus intensivement au sein de chaque région pour converger vers un minimum local (*effort d'intensification*). Cependant, comme récemment mentionné dans [20], la plupart des algorithmes de RL considèrent diversification et intensification comme deux objectifs antagonistes : à mesure qu'on effectue plus d'intensification, on peut perdre en diversification, et vice-versa. Une plus grande coordination entre ces deux objectifs est donc nécessaire, en équilibrant entre (i) *degré d'exploration*, correspondant à la quantité d'efforts consacrées pour conduire l'exploration vers des régions distinctes de l'espace de recherche, et (ii) *degré d'exploitation*, définie par la quantité d'efforts déployée en direction de la recherche locale pour explorer une région prometteuse.

La liste de mouvements candidats (*Candidate List Strategy*) de type `Aspiration Plus` [15] est un mécanisme simple qui a été proposé pour la recherche tabou. Elle permet de restreindre le nombre de voisins examinés afin d'équilibrer entre l'effort de recherche dans le voisinage et la qualité du meilleur voisin trouvé. `IDWalk` [28] est une autre métaheuristique qui utilise une CLS pour gérer l'exploration d'un voisinage.

Dans cet article, nous présentons une extension de l'algorithme de base VNS, appelé `ID-VNS`, qui intègre des mécanismes simples pour réaliser un contrôle efficace de balance diversification/intensification au sein de VNS. Pour atteindre cet objectif, nous définissons tout d'abord une *mesure de diversité*, appelée *degré de perturbation*, pour contrôler la quantité de changements nécessaires pour transformer une solution en une autre. Deuxièmement, comme dans [28], nous utilisons une CLS pour contrôler l'effort d'exploration d'un voisinage par la recherche locale. Ensuite, nous montrons comment `ID-VNS` peut être appliquée au problème de coloration de graphe, en définissant des voisinages génériques qui bénéficient à la fois des avantages des conflits et de la topologie du graphe de contraintes.

Nous avons effectué des expérimentations sur plusieurs familles de graphes du deuxième "challenge" de DIMACS [29], et nous avons comparé nos résultats à ceux obtenus par six méthodes de recherche locale, ainsi que deux algorithmes hybrides à base de populations `HCA` [9] et `MMT` [22]. Ces résultats montrent que notre méthode domine nettement les méthodes de recherche locale et est très compétitive avec les approches hybrides.

La section 2 donne un aperçu synthétique du problème de coloration et présente les meilleurs algorithmes de coloration. Dans la section 3, nous décrivons notre méthode de résolution `ID-VNS`. Ensuite, nous détaillons (section 4) nos structures de voisinages. La section 5 est consacrée aux expérimentations. Enfin, nous concluons et donnons quelques perspectives.

2 Problème de coloration de graphe

2.1 Définitions et notations

Soit $G = (V, E)$ un graphe connexe avec un ensemble V de sommets et un ensemble E d'arêtes. Étant donné un entier positif k, une k-coloration de G est une fonction $c : V \rightarrow \{1, \ldots k\}$, qui attribue une valeur $c(u)$, appelée *couleur* de u, à chaque sommet $u \in V$. Si deux sommets adjacents u et v ont la même couleur, on dit qu'ils sont en *conflit* et que l'arête $(u, v) \in E$ est une arête *conflictuelle*.

Une k-coloration sans conflit est dite *légale*, sinon elle est *illégale*. Soit $s = [c(1), \ldots, c(|V|)]$ une k-coloration légale, s peut être représentée par une *partition* de V en k sous-ensembles disjoints V^1, ..., V^k. Nous notons par V^r la classe de couleur r induite par s (i.e. l'ensemble de sommets ayant la couleur r dans s).

Soit f la fonction objectif qui associe à chaque coloration s le nombre d'arêtes conflictuelles induites par s. Le problème de coloration des sommets d'un graphe (PCSG) consiste à déterminer le plus petit entier k, appelé *nombre chromatique* $\chi(G)$, tel qu'il existe une k-coloration légale de G.

2.2 Approches métaheuristiques

Dans cette section, nous décrivons brièvement les meilleurs algorithmes de coloration (voir [23] pour plus de détails).

Un des premiers algorithmes de recherche locale proposé pour ce problème est `TABUCOL` [17]. Depuis, il a été amélioré et incorporé avec succès dans plusieurs algorithmes complexes de coloration [9, 18].

Morgenstern [27] a proposé, dans le cadre d'un Recuit Simulé, un algorithme distribué complexe, noté `MOR`, utilisant un codage à base de colorations légales *partielles* (i.e., certains sommets peuvent ne pas être colorés). Chaque sommet non coloré induit une pénalité et l'objectif est de minimiser la somme des degrés des sommets non colorés.

Galinier et Hao [9] ont proposé une approche hybride évolutionniste, notée `HCA`, incorporant `TABUCOL` comme recherche locale, et avec un croisement glouton très efficace à base de partitions.

Dans [2], deux algorithmes Tabou (`DYN-P.COL` et `FOO-P.COL`), utilisant un codage sous forme de colorations légales partielles ont été proposés. `DYN-P.COL` (resp. `FOO-P.COL`) utilise un réglage dynamique (resp. réactif) de la durée Tabou.

Algorithm 1: Pseudo-code de `ID-VNS`

```
   fonction ID-VNS(maxIter, maxneigh, maxMoves, nextneigh) ;
   début
       Soit N_i, (i = 1,...,i_max), un ensemble prédéfini de structures de
       voisinages ;
1      s ← genRandomSol();
2      iter ← 1, i ← 1, b ← b^i_max;
3      tant que (iter ≤ maxIter) faire
4          iter ← iter + 1;
5          s' ← Perturbation(s, i, b) ;
6          s'' ← LS(s', maxneigh, maxMoves, nextneigh) ;
7          si f(s'') < f(s) alors
8              s ← s'', i ← 1;
9              b ← b^i_max, iter ← 1;
           sinon
10             si f(s'') = f(s) alors s ← s'' ;
11             si (iter mod α = 0) alors i ← i + 1 ;
12             b ← getB(iter, i);
13     retourner s;
```

Enfin, Hertz et al. [18] ont proposé une extension de l'algorithme VNS, appelé *Variable Search Space* (`VSS-Col`), dans lequel le processus de recherche commute itérativement entre trois espaces de recherche, chacun ayant son propre codage, sa propre fonction objective et son voisinage.

3 Intensification/Diversification VNS

3.1 Schéma de base de `ID-VNS`

`ID-VNS` étend l'algorithme de base VNS en incluant des mécanismes permettant de réaliser un contrôle efficace de balance diversification/intensification au sein de VNS. Pour parvenir à ce compromis, deux objectifs doivent être équilibrés : (i) *degré d'exploration* et (ii) *degré d'exploitation*. Pour le premier objectif, nous définissons une mesure de diversité, appelée *degré de perturbation* (b), pour contrôler la quantité de changements effectuées sur la solution courante, tout en conduisant la recherche locale vers les régions de l'espace de recherche non encore visités (cf. sec. 3.2). Pour le second objectif, nous utilisons une *liste de mouvements candidats* [15, 28] pour équilibrer l'intensification de la recherche dans le voisinage et la possibilité de s'échapper rapidement des optima locaux (cf. sec. 3.3).

Le pseudo-code de `ID-VNS` est décrit par l'algorithme 1. Nous désignons par i l'indice du voisinage courant N_i, et par b^i_{min} (resp. b^i_{max}) le nombre minimum (resp. maximum) de perturbations effectuées dans chaque N_i. L'algorithme part d'une solution initiale s générée aléatoirement. Le voisinage est initialisé à N_1 en ligne 2. Chaque itération de la boucle des lignes 3 à 12 consiste en l'application d'une perturbation de degré b à s dans le voisinage N_i (ligne 5), suivie d'une recherche locale appliquée à s' (ligne 6). Si la solution voisine obtenue s'' est de meilleure qualité que la solution courante s, alors s'' devient la solution courante et le voisinage est réinitialisé à N_1 (lignes 8). La perturbation dans N_i prend fin lorsque (`iter`$/\alpha$) itérations sont effectuées sans améliorer s (ligne 11). L'algorithme considère alors une autre perturbation dans le voisinage suivant N_{i+1}. La valeur de α a été fixée à $(2 \times |V|/i_{max})$. l'algorithme s'arrête dès que `maxIter` itérations consécutives sont effectuées sans améliorer s.

3.2 Étape de perturbation

Chaque perturbation consiste à choisir une variable en conflit dans s et une valeur pour cette variable minimisant le nombre de conflits. Le choix de ces variables dépend de chaque N_i (cf. sec. 4). Une solution voisine s' est générée dans N_i, en réaffectant b variables en conflit.

Le paramètre b permet de contrôler le nombre d'éléments de la solution courante s qui sont modifiés. Il représente une mesure simple de *diversification*. En effet, la solution s' constitue un point de départ à partir duquel la recherche locale va explorer des régions de l'espace des solutions non encore visitées. Toutefois, le nombre de modifications que la perturbation devrait effectuer reste un point crucial. Il doit être suffisamment grand pour éviter à la recherche locale de retomber dans un optimum local déjà visité, tout en empêchant celle-ci à dégénérer en un algorithme multi-restart.

Nous proposons d'exploiter l'information sur l'effort d'amélioration de la solution courante pour ajuster dynamiquement la valeur du paramètre b. Dans notre approche, b diminue proportionnellement en fonction du nombre d'itérations consécutives effectuées sans amélioration de la solution courante (i.e., `iter`). Initialement, $b = b^i_{max}$. Durant la recherche, à chaque fois que s est améliorée b est réinitialisé à sa valeur de départ (ligne 9) ; sinon il est décrémenté à chaque fois que `iter` augmente, jusqu'à atteindre une valeur minimale b^i_{min} (ligne 12). L'objectif étant d'effectuer de fortes perturbations à chaque fois que s est améliorée, et de favoriser progressivement de petites perturbations lorsque s n'a pas été améliorée pendant une longue période. Après quelques expérimentations préliminaires, nous avons initialisé b^i_{min} à 20 pour $(i = 1, \ldots, 5)$ et b^i_{max} à 100 et 50, respectivement pour $i = 1$ et $i = 2 \ldots, 5$.

3.3 Étape de recherche locale

Comme recherche locale, nous avons utilisé un algorithme *tabou*. Son pseudo-code est décrit par l'algorithme 2. Étant donné une solution courante s, à chaque itération (boucle des lignes 8 à 19), l'algorithme génère une solution voisine s', en modifiant une variable en conflit v dans s et en calculant une nouvelle affectation x_v pour cette variable qui minimise les conflits (lignes 9 et 10). L'exploration du voisinage de s s'arrête dès qu'un voisin est accepté ou que `maxneigh` voisins ont été examinés. Pour évaluer les mouvements, nous utilisons des structures de données incrémentales [8, 10]. L'idée est de maintenir pour chaque paire (v, x_v) un *compteur de conflits* qui enregistre le nombre de variables dans s qui sont en conflit avec l'affectation $(v = x_v)$. À Chaque fois qu'un mouvement est effectué, uniquement les compteurs des variables affectées par ce mouvement sont mis à jour. Une fois le mouvement effectué, la variable v est interdite de revenir à sa précédente valeur pour les prochaines T itérations (ligne 22). Après quelques expériences préliminaires, la *durée tabou* T a été fixée à 20. L'algorithme s'arrête dès qu'un nombre fixé d'itérations (`stopIter`) a été effectué. Chaque fois qu'une meilleure solution est trouvée, `stopIter` est mis à jour en ligne 24 (effort d'intensification).

Pour contrôler finement le compromis *intensification/diversification* dans l'exploration des voisinages, il est nécessaire de contrôler (i) le nombre maximum de voisins à explorer pour choisir un mouvement et (ii) le choix du prochain voisin. En effet, le nombre de voisins à considérer devrait être assez grand pour se focaliser plus intensivement sur des régions jugées prometteuses (i.e., effort d'intensification). Toutefois, il devrait être suffisamment petit pour permettre à la recherche de s'échapper des optima locaux plus rapidement (i.e., effort de diversification). Dans ce cas, la manière de choisir le prochain voisin devrait guider la recherche vers des régions inexplorées de l'espace des solutions. Comme pour `IDWalk` [28], nous proposons d'intégrer à notre recherche tabou une *liste de mouvements candidats* pour contrôler le parcours d'un voisinage. Deux paramètres sont définis : `maxneigh` qui indique le nombre maximum de voisins à examiner avant d'effectuer un mouvement, et `nextneigh` qui précise quelle prochaine solution choisir si aucun voisin parmi les `maxneigh` visités n'a été accepté. Deux valeurs sont possibles pour ce paramètre : `first` : on choisit le premier voisin (`s_premier`) parmi les `maxneigh` candidats non acceptés, `best` : on choisit le voisin (`s_meilleur`) qui détériore le moins la fonction d'évaluation.

La boucle des lignes 8 à 19 s'arrête dès qu'une première solution s' qui améliore la meilleure solution connue s^* est obtenue ou aucune amélioration n'ait été trouvée après `maxneigh` itérations. Si la solution voisine obtenue s' n'est pas tabou ou qu'elle satisfait un *critère d'aspiration*[1], s' de-

Algorithm 2: Pseudo-code de LS

```
   fonction LS (s, maxneigh, maxMoves, nextneigh) ;
   début
1    stopIter ← maxMoves;
2    s_meilleur ← ∅,  f(s_meilleur) ← ⊤;
3    iter ← 1,  s* ← s;
4    tant que (iter ≤ stopIter) faire
5        iter ← iter + 1;
6        nbtries ← 1;
7        premierTrouvé ← faux, trouvé ← faux;
8        tant que (nbtries ≤ maxneigh) et non(trouvé) faire
9            v ← randomConflictVertex(s);
10           s' ← getNeighbor(s, v);
11           si non(premierTrouvé) alors
12               premierTrouvé ← vrai;
13               s_premier ← s';
14           si (v, s'[v]) ∉ tabulist) ou (f(s') < f(s*) alors
15               s ← s';
16               trouvé ← vrai;
             sinon
17               si non(trouvé) et (nextneigh = best) et
                 (f(s') < f(s_meilleur)) alors
18                   s_meilleur ← s' ;
19           nbtries ← nbtries + 1;
20       si  non(trouvé) et (nextneigh = first) alors
         s ← s_premier ;
21       si  non(trouvé) et (nextneigh = best) alors
         s ← s_meilleur ;
22       insérer (v, c(v)) dans tabulist et rendre (v, c(v)) tabou
         pour T itérations;
23       si f(s) < f(s*) alors
24           stopIter ← iter + maxMoves;
25           s* ← s;
26   retourner s*
```

1. Il s'agit d'éliminer le caractère tabou d'un mouvement lorsque celui-ci conduit à une solution meilleure que celle obtenue jusqu'ici.

vient la nouvelle solution courante (lignes 14 à 16) ; sinon, `s_meilleur` est mis à jour (ligne 18). Selon la valeur de `nextneigh`, la prochaine solution est sélectionnée parmi les candidats rejetés (lignes 20 et 21).

4 Structures de voisinages

Dans cette section, nous détaillons nos structures de voisinages génériques pour le problème de coloration de graphe. Ces voisinages tiennent compte de la topologie du graphe de contraintes et des conflits. On notera par $neighbors(u)$ l'ensemble des sommets adjacents à u et par $\mathcal{X}(s)$ l'ensemble des sommets en conflit dans la solution courante s.

4.1 ConflictVar

ConflictVar (`CV`) sélectionne aléatoirement un sommet v en conflit dans $\mathcal{X}(s)$, puis choisit une nouvelle classe de couleur V^i pour ce sommet qui *minimise* les conflits. Soit s_1 la solution obtenue. Si $f(s_1) > f(s)$, `CV` sélectionne aléatoirement un nouveau sommet parmi ceux en conflit dans $\mathcal{X}(s_1) \setminus \mathcal{X}(s)$ et lui assigne une nouvelle meilleure classe de couleur. Ce processus est répété jusqu'à ce qu'un mouvement *non-détériorant* (i.e., n'augmentant pas le nombre de conflits) est trouvé. Dans notre implémentation, nous évitons de changer la couleur d'un sommet plus d'une fois. Cette séquence de mouvements est appliquée avec b différents sommets en conflit. Un tel voisinage qui repose principalement sur un choix aléatoire, permet de diversifier la recherche.

4.2 Extensions de ConflictVar

`CV` choisit aléatoirement les sommets en conflit indépendamment de la topologie du graphe de contraintes. Les sommets sélectionnés peuvent être complètement isolées dans le graphe et peuvent avoir des degrés élevés. Dans un tel cas, il est peu probable de modifier la couleur d'un sommet sans créer de nombreux conflits, et donc de trouver une solution de meilleure qualité. Ainsi, nous proposons d'exploiter la topologie du graphe de contraintes pour favoriser la sélection de sommets qui sont le plus liés [21].

1. *ConflictVar-Connected* (`CV_C`) choisit aléatoirement un sommet origine v_{init} dans s et lui assigne une nouvelle meilleure couleur V^i possible. Soit s_1 la solution obtenue. Si $f(s_1) > f(s)$, `CV_C` sélectionne aléatoirement un nouveau sommet u parmi ceux en conflit dans $(\mathcal{X}(s_1) \setminus \mathcal{X}(s)) \cap V^i$ et lui assigne une nouvelle meilleure couleur V^j. Soit s_2 la nouvelle solution obtenue. Si $f(s_2) > f(s_1)$, `CV_C` sélectionne aléatoirement un nouveau sommet z parmi ceux en conflit dans $(\mathcal{X}(s_2) \setminus \mathcal{X}(s_1)) \cap V^j$ et lui assigne une nouvelle meilleure couleur V^r. Cette séquence de mouvements est répétée jusqu'à ce qu'un mouvement *non-détériorant* est trouvé, formant ainsi un *sous-graphe connexe* dans G. Nous évitons également de changer la couleur d'un sommet plus d'une fois. Ce processus est est réitéré avec b différents v_{init}. Le prochain v_{init} est choisi parmi ceux en conflit dans V^i.
2. *ConflictVar-Star* (`CV_S`) choisit aléatoirement un premier sommet "centre" v_c dans s et lui assigne une nouvelle meilleure couleur V^i possible. Ensuite, pour chaque sommet de $neighbors(v_c) \cap V^i$, une nouvelle couleur qui minimise les conflits est alors trouvée. Ce processus est est réitéré avec b différents v_c. Un nouveau centre est déterminé parmi les sommets en conflit dans $neighbors(v_c) \cap V^i$.
3. *ConflictVar-ConnectedStar* (`CV_CS`) choisit aléatoirement un premier sommet "centre-connexe" v_{cc} dans s et lui assigne une nouvelle meilleure couleur V^i possible. Ensuite, `CV_CS` considère chaque sommet en conflit dans $neighbors(v_{cc}) \cap V^i$ comme un centre à partir duquel `CV_S` est appliquée. Ce processus est est réitéré avec b différents v_{cc}. Un nouveau "centre-connexe" est déterminé parmi les sommets en conflit voisins des centres.
4. *Conflict ColorClass* (`C_CC`) sélectionne la classe de couleur V^i ayant le plus grand nombre de sommets en conflit (les égalités sont cassées aléatoirement), puis choisit, pour chacun de ces sommets, une autre meilleure couleur possible. Soit s_1 la nouvelle solution obtenue. Ensuite, b sommets nouvellement en conflit (i.e., ceux en conflit dans $\mathcal{X}(s_1) \setminus \mathcal{X}(s)$) sont alors sélectionnés, puis réaffectés à leur meilleure couleur possible. Une *liste tabou* est utilisée pour interdire la sélection de la classe V^i pour les prochaines l itérations de `ID-VNS`, avec $l = 0.6 \times |\mathcal{X}(s)| + rand(0, 9)$ [9], où $rand(0, 9)$ est une fonction retournant un nombre aléatoire dans $\{0, \ldots, 9\}$.

5 Résultats expérimentaux

Dans cette section, nous présentons des résultats détaillés sur différents types de graphe du challenge de DIMACS. Pour certains graphes difficiles, nous considérons un ensemble d'instances (G, k), avec différentes valeurs de k. En effet, la difficulté de colorier un graphe est strictement corrélée avec la valeur de k. Ainsi, le niveau de difficulté pour trouver une k-coloration légale peut varier de trivial à très difficile. Pour chaque instance, **nous nous comparons avec les meilleurs algorithmes de coloration**. Comme les expérimentations ont été menées sur diverses machines, nous rapportons (lorsque cela est possible), les temps CPU normalisés [2].

5.1 Instances DIMACS

Nous avons expérimenté notre algorithme sur les graphes difficiles suivants [29] :

- 12 graphes aléatoires classiques **DSJC**$n.y$ ayant n sommets et de densité égale à $0.y$. Nous avons considéré les graphes avec $n \in \{125, 250, 500, 1000\}$ et $y \in \{1, 5, 9\}$.
- 2 graphes géométriques aléatoires **DSJR**$n.r$ construits à partir d'un ensemble de n points aléatoires (dans le carré unité) en joignant chaque couple de points distants de moins d'un seuil de longueur (indiqué par r). Nous avons considéré les deux graphes avec $n = 500$ et $r \in \{1, 5\}$.
- 6 graphes **flat**n_x_0, qui sont des graphes quasi-aléatoires avec un nombre chromatique connu x. Les graphes flat sont générés en partitionnant l'ensemble des sommets en x classes et en ajoutant des arêtes uniquement entre les sommets de classes différentes. Nous avons

2. Pour une machine κ fois plus lente que la nôtre, les temps CPU rapportés seront divisés par κ.

Instance	k	N_i	MaxN.	Succ.	Time	Avg
DSJC250.5		N_1	200	18/20	113.5	0.1
n=250		N_2	175	18/20	148.1	0.1
m=15668	28	N_3	100	15/20	83.2	0.25
k^*=28		N_4	150	20/20	118.9	0
		N_5	**150**	**20/20**	**76.4**	**0**
DSJC250.9		**N_1**	50	20/20	8.5	0
n=250		N_2	100	20/20	9.8	0
m=27897	72	N_3	150	20/20	9.9	0
k^*=72		**N_4**	**50**	**20/20**	**8.4**	**0**
		N_5	50	20/20	10.86	0
DSJC500.1		N_1	175	17/20	142.4	0.15
n=500		N_2	175	17/20	448.9	0.2
m=24916	12	N_3	175	15/20	325.7	0.4
k^*=12		N_4	100	13/20	303.4	0.5
		N_5	**200**	**20/20**	**121.9**	**0**
		N_1	**100**	**16/20**	**965.4**	**0.3**
	49	N_2	175	14/20	889	1.15
		N_3	100	11/20	1140.2	0.75
DSJC500.5		N_4	200	16/20	1778.6	1.05
n=500		N_5	200	12/20	1894.3	0.95
m=125248		N_1	200	1/10	2077.6	1.5
k^*=48		**N_2**	**100**	**2/10**	**4762.6**	**1.2**
	48	N_3	100	2/10	6658.2	1.2
		N_4	100	2/10	6445.6	1.6
		N_5	50	1/10	2820.6	2.1
		N_1	**50**	**20/20**	**153.10**	**0**
	127*	N_2	100	20/20	173.3	0
		N_3	100	19/20	203.3	0.05
DSJC500.9		N_4	100	20/20	208.5	0
n=500		N_5	50	19/20	194.2	0.05
m=224874		N_1	**150**	**18/20**	**2451**	**0.15**
k^*=126	126	N_2	50	11/20	3513	0.45
		N_3	150	10/20	6072.4	0.5
		N_4	175	14/20	5348.8	0.3
		N_5	175	12/20	6093.4	0.4
DSJR500.1c		N_1	150	0/10	-	2
n=500		N_2	200	0/10	-	2
m=121275	85	N_3	50	0/10	-	2
k^*=85		N_4	50	1/10	152.2	1.8
		N_5	**50**	**10/10**	**1713.5**	**0**
		N_1	150	0/10	-	2.7
		N_2	100	0/10	-	3.1
	125	N_3	50	6/10	4407.9	0.4
DSJR500.5		N_4	100	0/10	-	1.4
n=500		**N_5**	**100**	**9/10**	**2702.6**	**0.15**
m=58862		N_1	175	0/10	-	3.6
k^*=122		N_2	100	0/10	-	3.9
	124	N_3	175	0/10	-	1.7
		N_4	175	0/10	-	1.8
		N_5	**175**	**1/10**	**297.1**	**1.4**
DSJC1000.1		N_1	50	20/20	3.67	0
n=1000		N_2	50	20/20	3.1	0
m=49629	21*	N_3	100	20/20	3.8	0
k^*=20		N_4	50	20/20	3.3	0
		N_5	**50**	**20/20**	**3.08**	**0**
		N_1	150	20/20	1546.4	0
	88	N_2	50	20/20	1322.9	0
		N_3	50	20/20	2253.4	0
		N_4	**100**	**20/20**	**852.6**	**0**
		N_5	50	20/20	1116.1	0
DSJC1000.5		N_1	150	0/10	-	5.5
n=1000	86	**N_2**	**100**	**2/10**	**4998.6**	**1.7**
m=499652		N_3	100	0/10	-	12.7
k^*=83		N_4	50	0/10	-	19.3
		N_5	175	2/10	27,071.2	4.8
le450_15c		N_1	25	20/20	0.6	0
n=450		N_2	25	20/20	0.6	0
m=16680	15*	N_3	25	20/20	0.5	0
k^*=**15**		N_4	25	20/20	0.4	0
		N_5	**25**	**20/20**	**0.3**	**0**

Instance	k	N_i	MaxN.	Succ.	Time	Avg
le450_15d		N_1	150	20/20	2.9	0
n=450		N_2	50	20/20	6.4	0
m=16750	15*	**N_3**	**50**	**20/20**	**1.3**	**0**
k^*=**15**		N_4	150	19/20	6.7	0.05
		N_5	50	20/20	3.2	0
le450_25c		N_1	100	7/25	19.9	0.8
n=450	26*	**N_2**	**100**	**25/25**	**10**	**0**
m=17343		N_3	50	18/25	8	0.28
k^*=**25**		N_4	150	18/25	12	0.32
		N_5	100	12/25	19.4	0.68
le450_25d		N_1	100	11/20	61	0.56
n=450	26*	**N_2**	**100**	**20/20**	**11.3**	**0**
m=17425		N_3	150	13/20	6.5	0.52
k^*=**25**		N_4	100	17/20	9	0.4
		N_5	150	7/20	17.2	0.88
flat300_26_0		N_1	50	20/20	4.6	0
n=300		N_2	50	20/20	4.2	0
m=21633	26	N_3	50	20/20	5	0
k^*=**26**		**N_4**	**50**	**20/20**	**3.2**	**0**
		N_5	50	20/20	10.3	0
		N_1	150	15/20	192	0.25
	31*	N_2	150	17/20	160.9	0.15
		N_3	100	17/20	127.8	0.15
		N_4	175	17/20	239.1	0.15
		N_5	**200**	**20/20**	**185**	**0**
Flat300_28_0		N_1	150	10/20	1399.43	1.7
n=300	30	**N_2**	**100**	**10/20**	**1273.6**	**2**
m=21695		N_3	100	6/20	1008.4	2.45
k^*=**28**		N_4	150	9/20	1932.9	2
		N_5	150	8/20	1344.8	2.2
		N_1	50	2/20	547.3	9.35
	29	N_2	200	4/20	2075.8	8.3
		N_3	150	2/20	1876.4	9.35
		N_4	**200**	**4/20**	**1612.7**	**8.2**
		N_5	200	3/20	1190.9	8.75
flat1000_50_0		N_1	50	0/20	2379.8	-
n=1000		N_2	50	0/20	2305.1	-
m=224874	50*	N_3	50	0/20	2226.7	-
k^*=**50**		N_4	50	0/20	1977.4	-
		N_5	**50**	**20/20**	**2858.1**	**0**
flat1000_60_0		N_1	50	0/20	-	-
n=1000		N_2	50	0/20	-	-
m=245830	60	N_3	50	0/20	-	-
k^*=**60**		N_4	50	0/20	-	-
		N_5	**50**	**20/20**	**13,854**	**0**
		N_1	100	20/20	1359.2	0
	87	N_2	50	20/20	1780.5	0
		N_3	150	20/20	3722.4	0
flat1000_76_0		**N_4**	**50**	**20/20**	**859.9**	**0**
n=1000		N_5	50	20/20	1204.7	0
m=246708		**N_1**	**100**	**17/20**	**5750.6**	**0.15**
k^*=82	86	N_2	100	1/20	29,765	2.8
		N_3	50	0/20	-	5.05
		N_4	150	0/20	-	9.88
		N_5	100	2/20	20,579	1.77
		N_1	50	20/20	1803	0
	225*	**N_2**	**50**	**20/20**	**1546**	**0**
		N_3	100	20/20	5467	0
DSJC1000.9		N_4	50	20/20	2542	0
n=1000		N_5	175	20/20	20,816	0
m=449449		N_1	175	8/10	31,461	0.2
k^*=224	224	N_2	50	5/10	13,384	0.7
		N_3	150	5/10	24,483	0.55
		N_4	50	3/10	20,671	1.2
		N_5	**150**	**9/10**	**32,598**	**0.1**
		N_1	50	0/20	-	3.35
latin_square_10		N_2	50	0/20	-	3.35
n=900	100	N_3	100	2/20	16,878.5	2.3
m=307350		N_4	50	0/20	-	3.75
k^*=98		**N_5**	**100**	**15/20**	**12,812.1**	**0.3**

TABLE 1 – Comparaison des différents voisinages : `CV` : N_1, `CV_C` : N_2, `CV_S` : N_3, `CV_CS` : N_4 and `C_CC` : N_5. Les cellules marquées par * indiquent les valeurs de k pour lesquelles de courtes exécutions sont effectuées.

considéré les $flat300_x_0$, avec $x \in \{20, 26, 28\}$ et les $flat1000_x_0$, avec $x \in \{50, 60, 76\}$.

- 4 graphes de Leighton **le**n_x issus de l'ordonnancement, ayant 450 sommets et un nombre chromatique connu et égal à la taille de la plus grande clique. Nous avons considéré ceux avec $x \in \{15, 25\}$ et avec l'extension c ou d (i.e., les plus difficiles).
- un graphe de carré latin (latin_square_10).

5.2 Protocole expérimental

Deux séries de tests ont été réalisées avec différents réglages des paramètres `maxMoves` et `maxIter`, correspondants à des exécutions *courtes* et des exécutions plus *longues*. Les exécutions courtes sont utilisées sur les k-coloration "faciles", alors que des exécutions longues sont utilisées sur les grandes denses, avec des valeurs de k proches ou égales aux meilleures colorations connues :

(i) **exécutions courtes** (`maxIter`=n) : `maxMoves`=4500 pour $n = 450$ et `maxMoves` $= 50,000$ pour $n \in \{300, 500, 1000\}$;

(ii) **exécutions longues** (`maxIter`=$2 * n$) : `maxMoves` $= 50,000$ pour $n = 250$ et `maxMoves` $= 100,000$ pour $n \in \{300, 500\}$;

(iii) **exécutions longues** (`maxIter`=$3 * n$) : `maxMoves`=$200,000$ pour $n = 1000$.

Mentionnons que dans la littérature sur la coloration de graphe, il est courant de faire tourner un algorithme de coloration de plusieurs heures à plusieurs jours pour tenter de résoudre une instance difficile [2, 18].

Notre objectif est d'obtenir un *réglage "générique"* de ces deux paramètres en s'appuyant sur les caractéristiques générales des instances.

Pour chaque paire (G,k), nous avons lancé 20 (ou 10) exécutions sur un Intel Core 2 DUO à 2GHz et à 2GB de RAM. Nous donnons (au plus) les trois plus petites valeurs de k pour laquelle il existe une k-coloration légale [3]. Pour chaque algorithme, nous rapportons le taux de succès ("succ. runs/total runs"), le temps moyen requis pour trouver une solution ainsi

3. Les meilleurs résultats sont disponibles à l'adresse : http ://www.info.unicaen.fr/~loudni/Coloring.html.

que le coût moyen sur l'ensemble des exécutions. ID-VNS a été implémenté en C++.

5.3 Comparaison des différents voisinages

Pour évaluer les performances de chacun des voisinages, nous avons exécuté ID-VNS avec une seule structure de voisinage ($i_{max} = 1$), avec nextneigh = first et maxneigh $\in \{50, 100, 150, 175, 200\}$.

Le Tableau 1 présente les différents résultats obtenus. La colonne 1 indique les caractéristiques de chaque graphe : son nom, le nombre de sommets (n), le nombre d'arêtes (m) et la meilleure coloration connue dans la littérature (k^*) (en gras quand elle est prouvée optimale). La colonne 2 indique le nombre de couleurs k utilisé. La colonne 3 désigne les différents voisinages. La colonne 4 indique le meilleur réglage pour maxneigh.

Pour comparer les différents voisinages, nous donnons à chaque N_i un score noté (b-s-w), correspondant au nombre de k-colorations pour lesquels N_i obtient respectivement le meilleur (second meilleur entre parenthèses), le même (100%) et le plus mauvais taux de succès. De ces résultats, nous pouvons dresser les remarques suivantes :

- Les voisinages basés sur la topologie du graphe de contraintes (excepté CV_S) sont nettement plus pertinents. Si l'on compare CV et CV_S, le score est de 5(2)-8-11 en faveur de CV ; le score de CV_S est de 1(5)-8-7. Ceci peut s'expliquer par le fait que CV_S effectue des perturbations seulement sur une partie très limitée du graphe (i.e., les voisins d'un sommet centre) alors que le caractère aléatoire de CV permet une plus grande diversification dans l'étape de perturbation, ce qui contribue à trouver des solutions plus facilement.
- CV_CS (avec un score de 5(6)-8-9) est plus performant que CV_S sur la plupart des graphes (excepté DSJC500.1, DSJR500.5 et latin_square_10). Cette comparaison est très instructive et montre l'importance de considérer des sommets plus liés.
- CV_C (avec un score de 8(5)-8-7) et C_CC (avec un score de 11(2)-8-3) surclassent CV_CS. Les deux voisinages trouvent des solutions avec un taux de succès de 100%, respectivement pour 11 et 16 instances, alors que CV_CS atteint le même taux de succès pour 9 instances. En effet, CV_C et C_CC permettent une diversification plus "agressive" en effectuant des perturbations sur différentes sous-parties connexes du graphe de contraintes.
- Enfin, C_CC est légèrement meilleur que CV_C, mais aucun des deux voisinages domine strictement l'autre. En effet, CV_C obtient de meilleurs résultats que C_CC sur 7 instances : (DSJC500.5, $k = 48, 49$), (DSJC1000.5, $k = 89$), le450_25c&d et (flat300_28_0, $k = 29, 30$), alors que C_CC surclasse CV_C sur 9 instances : DSJC250.5, DSJC500.1, DSJR500, (DSJC1000.9, $k = 224$), flat1000_50&60, (flat300_28_0, $k = 31$) et latin_square_10.

5.4 Performance de ID-VNS par rapport à TABUCOL, ID-Walk et VSS-Col

Nous avons comparé ID-VNS à trois algorithmes de recherche locale :

(i) IDWalk [28] : nous avons utilisé une variante basée sur notre recherche tabou. Nous avons effectué des expérimentations avec nextneigh = first et maxneigh $\in \{50, 100, 150, 175, 200\}$. Pour chaque valeur de k, et chaque essai, IDWalk est exécuté 20 fois avec un nombre maximal d'itérations (maxMoves) égal à $1,500,000$. Si aucune k-coloration n'est trouvée, alors il est exécuté 10 fois avec maxMoves égal à $5,000,000$. Un ensemble de 5 essais par k-coloration a été réalisé. Ceci donne une bonne base pour évaluer ID-VNS.

(ii) Deux versions de TABUCOL avec plusieurs restarts [17] : *Short Tabu* et *Long Tabu*. Le choix de TABUCOL réside dans le fait que ID-VNS utilise une variante proche de TABUCOL comme procédure de recherche locale. En outre, il peut être considéré comme une variante de IDWalk avec nextneigh = best [28].

(iii) VSS-Col qui est l'un des algorithmes de recherche locale les plus performants en coloration de graphe [18]. La raison de comparer ID-VNS à VSS-Col est que les deux méthodes étendent VNS. Toutefois, VSS-Col commute itérativement entre plusieurs espaces de recherche, chacun ayant sa propre fonction objective et son voisinage.

Le tableau 2 compare les performances des quatre méthodes. Les résultats de VSS-Col proviennent de [18] et correspondent à ceux obtenus avec une limite de temps d'une heure sur un Pentium 4 à 2 GHz et à 512 MB de RAM. Les résultats de TABUCOL proviennent de [17]. Pour TABUCOL, nous indiquons uniquement la meilleure valeur de k trouvée, tandis que pour ID-Walk, nous donnons aussi le taux de succès (entre parenthèses), le temps moyen en secondes et le coût moyen sur les cinq essais. Les trois dernières lignes donnent un aperçu des comparaisons. Les lignes *better*, *equal* et *worse* donnent le nombre de graphes pour lesquels notre méthode obtient respectivement une meilleure, égale et mauvaise coloration que les autres algorithmes.

De ces résultats, on observe que ID-VNS domine nettement les deux versions de TABUCOL ainsi que ID-Walk, particulièrement sur les graphes de grande taille. En effet, ID-VNS trouve de meilleures colorations que ID-Walk sur 10 graphes (DSJC500.5, DSJC500.9, DSJR500, DSJC1000.5, DSJC1000.9, flat1000 et flat300_28_0). Pour les deux graphes flat1000_50_0 et flat1000_60_0, ID-Walk n'a pas été en mesure de trouver une coloration légale, même pour des valeurs élevées de k. Sur les neuf graphes restants, ID-VNS et ID-Walk trouvent des solutions de même qualité, mais ID-VNS obtient un meilleur taux de succès.

Lorsqu'on compare avec les résultats de VSS-Col, ID-VNS obtient de meilleures solutions sur trois graphes (DSJR500.5, DSJC1000.5 et flat1000_76_0), avec des colorations utilisant respectivement 2, 2 et 1 couleurs en moins. Notons toutefois que ID-VNS n'a pas été capable de trouver une coloration légale avec 20 couleurs pour le graphe DSJC1000.1. Les deux méthodes obtiennent des colorations légales avec le même nombre de couleurs sur 12 graphes. Cependant, si on compare les taux de succès, ID-VNS est bien plus performant que VSS-Col sur trois graphes (DSJC500.9, DSJC1000.9 et flat300_28_0). Les deux méthodes obtiennent des colorations légales avec le même nombre de couleurs et le même taux de succès sur 9 graphes, mais VSS-Col est généralement plus rapide, sauf pour le450_15c et le450_15d, où ID-VNS obtient une coloration légale avec un nombre de couleurs égale au

Instance	k^*	k	ID-VNS		VSS-Col		IDWalk			ShortTabu	LongTabu
			Succ.	Time	Succ.	Time	Best k	Time	AVG	Best k	Best k
DSJC250.1	8	8	20/20	0	-	-	8 (5)	0	0	8	8
DSJC250.5	28	28	20/20	76	-	-	28 (1)	1241	0.8	30	29
DSJC250.9	72	72	20/20	8	-	-	72 (5)	15	0	73	72
DSJC500.1	12	12	20/20	121	10/10	97	12 (5)	1465	0	13	13
DSJC500.5	48	48	2/10	4762	**3/10**	1331	50 (3)	2378	0.4	53	50
		49	9/20	186	**10/10**	162					
DSJC500.9	126	126	18/20	2451	8/10	1686	127 (1)	3435	1	133	130
		127	20/20	153	10/10	169					
DSJR500.1c	85	85	10/10	1713	9/10	736	85 (0)	-	1.4	87	86
DSJR500.5	122	124	**1/10**	297	0/10	-	125 (0)	-	3.4	128	128
		125	**9/10**	2702	0/10	-					
DSJC1000.1	20	20	-	-	**3/10**	2396	21 (5)	4	0	22	22
		21	20/20	3	10/10	11					
DSJC1000.5	83	86	**2/10**	4998	-	-	90 (1)	2711	1.2	95	89
		88	**20/20**	852	8/10	2028					
		89	20/20	1608	10/10	820					
DSJC1000.9	224	224	**9/10**	32,598	1/10	3326	228 (1)	5707	1.2	248	245
		225	**20/20**	1546	5/10	1484					
		226	20/20	2227	10/10	1751					
le450_15c	15	15	20/20	0	10/10	6	15 (5)	3	0	15	15
le450_15d	15	15	20/20	1	10/10	44	15 (5)	5	0	22	16
le450_25c	25	26	25/25	10	10/10	1	26 (3)	6	0.4	27	26
le450_25d	25	26	25/25	11	10/10	1	26 (1)	279	0.8	27	27
flat300_28_0	28	29	**4/20**	1612	1/10	867	31 (1)	486	1.4	-	-
		30	**10/20**	1273	2/10	2666					
		31	20/20	185	10/10	39					
flat1000_50_0	50	50	20/20	2858	10/10	318	60 (0)	-	484.4	-	-
flat1000_60_0	60	60	20/20	13,854	10/10	694	70 (0)	-	223.8	-	-
flat1000_76_0	82	86	**17/20**	5750	-	-	90 (5)	1190	0	-	-
		87	**20/20**	859	4/6	1689					
		88			10/10	1155					
Better					**3**		**10**			**13**	**12**
Equal					12		9			2	3
Worse					**1**		**0**			**0**	**0**

TABLE 2 – Comparaison entre TABUCOL, IDWalk et VSS-Col.

Instance	k^*	MOR	HCA		FOO-P.COL		DYN-P.COL		MMT	ID-VNS	
		k_{best}	Succ.	k	Succ.	k	Succ.	k	k_{best}	Succ.	k
DSJC125.1	5								**5**	20/20	**5**
DSJC125.5	17	**17**							**17**	20/20	**17**
DSJC125.9	44								**44**	20/20	**44**
DSJC250.1	8								**8**	20/20	**8**
DSJC250.5	28	**28**	9/10	**28**					**28**	20/20	**28**
DSJC250.9	72								**72**	20/20	**72**
DSJC500.1	12	**12**			23/50	**12**	50/50	**12**	**12**	20/20	**12**
DSJC500.5	48	49	5/10	**48**	50/50	(49)50	1/50	(48)49	**48**	2/10	**48**
DSJC500.9	126	128			48/50	128	1/50	127	127	18/20	**126**
DSJR500.1c	85	**85**			50/50	**85**	3/50	**85**	**85**	10/10	**85**
DSJR500.5	122	123			24/50	128	28/50	(125)126	**122**	1/10	124
DSJC1000.1	20	21	-	**20**	50/50	21	3/50	**20**	**20**	20/20	21
DSJC1000.5	83	88	-	**83**	5/50	89	6/50	89	84	2/10	86
DSJC1000.9	224	226	-	**224**	30/50	228	30/50	(226)228	225	9/10	**224**
le450_15c	15	**15**	6/10	**15**	50/50	**15**	50/50	**15**	**15**	20/20	**15**
le450_15d	15	**15**			50/50	**15**	50/50	**15**	**15**	20/20	**15**
le450_25c	25	**25**	10/10	26	50/50	(26)27	50/50	(25)27	**25**	20/20	26
le450_25d	25	**25**			50/50	(25)27	50/50	(25)27	**25**	20/20	26
flat300_26_0	26	**26**							**26**	20/20	**26**
flat300_28_0	28	31	6/10	31	35/50	**28**	13/50	**28**	31	4/20	29
flat1000_50_0	50	**50**			50/50	**50**	50/50	**50**	**50**	20/20	**50**
flat1000_60_0	60	**60**			50/50	**60**	50/50	**60**	**60**	20/20	**60**
flat1000_76_0	82	89	4/5	**83**	10/50	88	9/50	88	(82) **83**	17/20	86
latin_square	98								101	15/20	**100**
Better		**6**		**1**		**8**		**8**	**4**		
Equal		10		5		7		6	14		
Worse		**3**		**3**		**1**		**2**	**6**		

TABLE 3 – Comparison aux meilleurs résultats sur les graphes les plus étudiés . Entre parenthèses, certains meilleurs résultats obtenus avec une limite de temps de 10h (non pris en compte dans nos comparaisons).

nombre chromatique très rapidement. Par conséquent, ID-VNS peut être considéré comme plus efficace que VSS-Col, même si notre méthode utilise des voisinages simples au sein d'un même espace de recherche.

Nous avons aussi comparé ID-VNS à deux encodages SAT du problème de coloration (*direct* et avec *support* [14]). MiniSAT, aidé par des techniques de cassage de symétrie, n'a pas été capable de résoudre même les instances denses de petites taille ; les résultats ne sont donc pas rapportés.

5.5 Comparaison avec les meilleurs algorithmes

Dans cette section, nous comparons notre méthode aux résultats des meilleurs algorithmes (voir sec. 2) : **trois algorithmes de recherche locale** (MOR [27] et DYN/FOO-P.COL [2]) et **deux algorithmes hybrides évolutionnistes** (HCA [9] et MMT [22]). Toutefois, nous ne rapportons pas les temps CPU des différentes méthodes, car les conditions d'expérimentation ne sont pas équivalentes. En effet, en plus des différentes machines utilisées, certains algorithmes (MOR, HCA et MMT) nécessitent un réglage spécifique de leurs divers paramètres, qui changent d'une instance à une autre. Les résultats sont donnés afin que le lecteur puisse avoir une base de comparaison pour évaluer les performances de ID-VNS.

Si on compare les résultats de ID-VNS à ceux des méthodes de recherche locale, on peut observer que ID-VNS domine nettement ces algorithmes (voir les trois dernières lignes du tableau 3). En effet, ID-VNS obtient de meilleurs résultats pour au moins six graphes, alors que de moins bons résultats sont obtenus pour au plus trois graphes. En outre, pour les graphes DSJC500.9 (resp. DSJC1000.9), ID-VNS et VSS-Col (resp. HCA) sont les seuls algorithmes capables d'atteindre une 126-coloration (resp. 224-coloration). Le détail des comparaisons est donné ci-dessous :

- ID-VNS est meilleur que MOR sur 6 graphes (DSJC500.5, DSJC500.9, DSJC1000.5, DSJC1000.9, flat300_28_0, et flat1000_76_0) et obtient de moins bons résultats sur 3 graphes (DSJR500.5, le450_25c et le450_25d).
- ID-VNS est meilleur que DYN/FOO-P.COL sur 8 graphes (DSJC500.5, DSJC500.9, DSJC1000.5, DSJC1000.9, le450_25c, le450_25d et flat1000_76_0) et obtient de moins bons résultats pour au plus 2 graphes (flat300_28_0 et/ou DSJC1000.1). Pour flat300_28_0, seul P.COL trouve facilement une 28-coloration.
- Si on considère le taux de succès, ID-VNS est meilleur que VSS-Col sur 6 graphes (DSJC500.9, DSJR500.5,DSJC1000.5, DSJC1000.9, flat300_28_0 et flat1000_76_0) et obtient de moins bons résultats sur 2 graphes (DSJC500.5 et DSJC1000.1).

Lorsqu'on se compare aux deux méthodes hybrides, on observe que ID-VNS est très compétitif. En effet, ID-VNS est meilleur que HCA sur flat300_28_0 et moins bon sur 3 graphes (DSJC1000.1, DSJC1000.5 et flat1000_76_0), meilleur que MMT sur 4 graphes (DSJC500.9, DSJC1000.9, flat300_28_0 et latin_square_10) et moins bon sur 6 graphes.

Cependant, ID-VNS doit être considéré comme une méthode de recherche locale simple qui utilise des mécanismes simples (sans aucun réglage spécifique des paramètres) pour échapper aux minima locaux, alors que HCA et MMT sont plus complexes, avec des ingrédients sophistiqués et finement réglés, qui changent d'une instance à une autre.

5.6 Analyse des paramètres de ID-VNS

Le but de cette section est d'analyser l'influence des paramètres les plus importants de ID-VNS : la liste de mouvements candidats (CLS), les paramètres netxneigh, maxneigh, maxIter et maxMoves.

5.6.1 Influence de nextneigh

Le tableau 4 compare les résultats de ID-VNS avec deux réglages du paramètre nextneigh : first et best. L'impact sur les performances de (nextneigh = first) est très significatif, particulièrement sur les graphes le450_25 et DSJC500, pour lesquels plusieurs ordres de grandeur sont obtenus. Les mauvais résultats de ID-VNS(best) sont probablement dues au fait que la sélection du meilleur voisin (i.e., de plus petit coût) parmi les maxneigh candidats rejetés conduit ID-VNS à rester coincé dans un profond optimum local en bouclant sur des zones déjà visitées. Ce surprenant résultat souligne l'importance de ce paramètre, pour lequel une plus grande diversification est obtenue avec nextneigh = first.

5.6.2 Influence de la liste de mouvements candidats

Pour montrer l'importance de la liste de mouvements candidats (CLS), nous avons comparé ID-VNS avec une version de VNS utilisant une recherche tabou n'exploitant aucune CLS (notée VNS/TS). Comme le montre le tableau 4, ID-VNS(first) est clairement le meilleur. À l'inverse, nous pouvons observer que l'impact de la CLS pour nextneigh=best n'est pas systématique. En effet, comparé à VNS/TS, ID-VNS(best) est très performant sur le450_15c&d. Cependant, sur les autres instances, VNS/TS est bien plus efficace. Ces résultats confirment, une fois de plus, l'importance du mécanisme CLS qui offre un moyen simple et efficace pour parvenir à un meilleur compromis intensification/diversification, en choisissant first.

5.6.3 Influence de maxneigh.

Pour mesurer l'impact du paramètre maxneigh, nous déterminons pour chaque valeur de maxneigh $\in \{50, 100, 150.175\}$ et chaque voisinage N_i, le nombre k-colorations pour lesquels maxneigh obtient respectivement le meilleur (b), second meilleur (entre parenthèses), le même (s) et le plus mauvais taux de succès (w). Le tableau 5 présente le résumé des comparaisons. Comme on pouvait s'y attendre, la valeur optimale de maxneigh dépend de l'instance, ainsi que du voisinage utilisé. Sur les instances "faciles" (i.e., DSJC250.9, le450_15c et le450_15d), l'impact sur les performance (i.e., taux de succès) de maxneigh semble négligeable. Toutefois, il est préférable de privilégier de petites valeurs (50 dans notre cas), probablement parce que les grandes valeurs sont plus coûteuses. À l'inverse, sur les instances difficiles, le gain en performance est très clair pour les valeurs élevées de maxneigh, qui favorisent une *forte intensification* dans l'étape de recherche locale, car plus de voisins sont examinés. Notons toutefois que maxneigh = 100 offre un comportement plus stable pour les cinq voisinages.

Instance	k	N_i	ID-VNS (first)			ID-VNS (best)			VNS/TS		
			Succ.	Time	Avg.	Succ.	Time	Avg.	Succ.	Time	Avg.
le450_15c	15	N_1	20/20	0.6	0	16/20	15	0.9	0/20	-	15.5(7)
		N_2	20/20	0.6	0	10/20	9.1	2.15	5/20	286	7.5
		N_3	20/20	0.5	0	4/20	12.9	17.05	1/20	670	11.5
		N_4	20/20	0.4	0	11/20	11.8	1.2	11/20	377.7	5.3
		N_5	20/20	0.3	0	7/20	17.4	3.1	0/20	-	24.7(2)
le450_15d	15	N_1	20/20	2.9	0	12/20	27.2	1.35	0/20	-	16.1(6)
		N_2	20/20	6.4	0	11/20	76.4	1.05	2/20	468.5	11.7
		N_3	20/20	1.3	0	3/20	99.4	5.7	0/20	-	15.6(4)
		N_4	19/20	6.7	0.05	10/20	64.5	0.95	6/20	447	7.74
		N_5	20/20	3.2	0	5/20	43.6	2.7	0/20	-	31.5(13)
le450_25c	26	N_1	7/25	19.8	0.8	0/20	-	5.6(4)	8/20	246.2	2.05
		N_2	25/25	10	0	0/20	-	6.1(4)	5/20	295.6	2.2
		N_3	18/25	8	0.28	0/20	-	4.4(3)	3/20	330.6	2.8
		N_4	18/25	12	0.32	0/20	-	7(4)	6/20	279.8	2
		N_5	12/25	19.4	0.68	0/20	-	6.6(5)	0/20	-	6.6(2)
le450_25d	26	N_1	11/20	61	0.56	0/20	-	4.95(3)	2/20	198	3
		N_2	20/20	11.3	0	0/20	-	5.4(3)	7/20	233	2.2
		N_3	13/20	6.5	0.52	0/20	-	4.9(3)	6/20	107.7	2.3
		N_4	17/20	9	0.4	0/20	-	6.8(6)	10/20	365.5	1.8
		N_5	7/20	17.2	0.88	0/20	-	6.6(5)	2/20	79.5	5.6
DSJC250.5	28	N_1	18/20	113.5	0.1	0/20	-	4.05(2)	2/20	1946	2
		N_2	18/20	148.1	0.1	0/20	-	3.9(2)	3/20	1436	2.7
		N_3	15/20	83.2	0.25	0/20	-	4(2)	4/20	1318	2.4
		N_4	20/20	118.9	0	0/20	-	4.15(2)	3/20	1494.3	2.6
		N_5	20/20	76.4	0	0/20	-	5(4)	1/20	149	5.5
DSJC500.9	127	N_1	20/20	153.1	0	0/20	-	3.85(2)	3/20	2015.3	3.35
		N_2	20/20	173.3	0	0/20	-	3.95(2)	1/20	5523	3.85
		N_3	19/20	203.3	0.05	0/20	-	3.7(2)	2/20	7686	3.95
		N_4	20/20	208.5	0	0/20	-	4.2(3)	2/20	11,626	3.45
		N_5	19/20	194.2	0.05	0/20	-	5.05(3)	1/20	687	5.3

TABLE 4 – Influence de la liste des mouvements candidats et du paramètre `nextneigh` sur les performances. Pour les essais infructueux, nous rapportons entre parenthèses la meilleure violation trouvée.

5.6.4 Exécutions courtes vs. Longues

Pour mieux analyser l'influence des exécutions courtes et longues sur notre algorithme, nous avons comparé `ID-VNS` avec les deux réglages sur six instances. Le tableau 6 rapportent les résultats obtenus. `ID-VNS` est capable d'améliorer très significativement les taux de succès obtenus avec les exécutions courtes, en particulier sur trois graphes (DSJC250.5, DSJC250.9 et DSJC500.1). Ce comportement montre l'importance des mécanismes de diversification utilisés (degré de perturbation et le paramètre `nextneigh`), qui forcent `ID-VNS` à découvrir des régions inexplorées de l'espace de recherche. Avec plus de temps de calcul, `ID-VNS` est capable de trouver de nouvelles meilleures solutions. Cette importante caractéristique n'est vérifiée que par peu d'algorithmes existants. Très souvent, ces méthodes se bloquent dans des optima locaux et passent leur temps à ré-explorer les mêmes régions dans une boucle, même si on leur donne plus de temps de calcul.

6 Conclusions

Dans ce paper, nous avons proposé une nouvelle extension de l'algorithme de base VNS, appelé `ID-VNS`, qui intègre des mécanismes simples pour réaliser un contrôle efficace de balance diversification/intensification au sein de VNS. Pour atteindre cet objectif, nous avons défini tout d'abord une mesure de diversité, appelée *degré de perturbation*, pour contrôler la quantité de changements nécessaires pour transformer une solution en une autre. Puis, nous avons utilisé une *une liste de mouvements candidats* pour contrôler l'effort d'exploration des voisinages par la recherche locale. Ensuite, pour le problème de coloration de graphe, nous avons défini des structures de voisinages génériques qui bénéficient à la fois des avantages des conflits et de la topologie du graphe de contraintes. Les résultats montrent que les voisinages les plus efficaces sont ceux qui effectuent une diversification "plus agressive".

Des expérimentations réalisées sur les instances difficiles de DIMACS [29] montrent que notre méthode domine nettement les meilleurs algorithmes de recherche locale de coloration de graphe. En outre, malgré sa simplicité, `ID-VNS` semble être très compétitive avec les approches hybrides évolutionnistes, qui sont beaucoup plus complexes. Une première analyse des paramètres de `ID-VNS`, montre l'importance de la CLS ainsi que du paramètre `nextneigh` pour atteindre un meilleur compromis entre l'intensification et diversification.

Comme travaux futurs, doter `ID-VNS` de mécanismes pour adapter automatiquement le choix du voisinage lors de la recherche et aussi d'étudier l'impact du degré de perturbation sur le compromis intensification/diversification.

Références

[1] Cédric Avanthay, A. Hertz, and N. Zufferey. A variable neighborhood search for graph coloring. *European Journal of Operational Research*, 151(2) :379–388, 2003.

[2] I. Blöchliger and N. Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers & OR*, 35(3) :960–975, 2008.

[3] J.R. Brown. Chromatic scheduling and the chromatic number problem. *Management Science*, 19(4) :456–463, 1972.

		maxneigh											
		50			100			150			175		
		b	s	w	b	s	w	b	s	w	b	s	w
Hard instances	N_1	3(5)	5	9	**5(3)**	5	**8**	**8(2)**	5	**8**	3(3)	5	10
	N_2	3(4)	5	12	**6(5)**	5	**8**	3(3)	5	10	5(2)	5	12
	N_3	6(7)	1	11	**9(3)**	1	**10**	**7(5)**	1	**11**	5(9)	1	8
	N_4	3(4)	6	12	5(4)	6	9	4(4)	6	11	**7(4)**	6	**7**
	N_5	5(5)	8	8	**5(7)**	8	**4**	6(4)	8	8	7(2)	8	6
Easy instances	N_1	0	4	0	0	4	0	0	4	0	0	4	0
	N_2	**1**	2	**1**	0(1)	3	0	0	3	0	0	3	1
	N_3	**1**	3	**0**	0(1)	3	0	0	3	0	0	3	1
	N_4	0	3	1	0(1)	2	1	**1**	3	**0**	**1**	3	**0**
	N_5	0	4	0	0	4	0	0	4	0	0	4	0

TABLE 5 – Influence du paramètre `maxneigh`.

Instance	k	N_i	MaxN.	Succ.	Time	Succ.Iter.	Avg
DSJC250.5		N_1	200	18/20	113.5	176	0.1
n=250		N_2	175	18/20	148.1	246	0.1
m=15668	28	N_3	100	15/20	83.2	178	0.25
k^*=28		N_4	150	20/20	118.9	190	0
		N_5	**150**	**20/20**	**76.4**	**138**	**0**
DSJC250.9		**N_1**	50	20/20	8.5	19	0
n=250		N_2	100	20/20	9.8	19	0
m=27897	72	N_3	150	20/20	9.9	15	0
k^*=72		**N_4**	**50**	**20/20**	**8.4**	**20**	**0**
		N_5	50	20/20	10.86	26	0
DSJC500.1		N_1	175	17/20	142.4	136	0.15
n=500		N_2	175	17/20	448.9	429	0.2
m=24916	12	N_3	175	15/20	325.7	318	0.4
k^*=12		N_4	100	13/20	303.4	372	0.5
		N_5	**200**	**20/20**	**121.9**	**102**	**0**
DSJC500.5		**N_1**	**100**	**16/20**	**965.4**	**719**	**0.3**
n=500		N_2	175	14/20	889	547	0.35
m=125248	49	N_3	100	11/20	1140.2	862	0.5
k^*=48		N_4	200	16/20	1778.6	700	0.25
		N_5	200	12/20	1894.3	704	0.45

Instance	k	N_i.	MaxN.	Succ.	Time	Succ.Iter.	Avg
DSJC250.5		**N_1**	**50**	**8/20**	**10.46**	**539**	**0.7**
n=250		N_2	100	8/20	11.4	519	1.1
m=15668	28	N_3	175	1/20	3.8	135	2.4
k^*=28		N_4	175	2/20	13.9	494	2.5
		N_5	200	7/20	15.9	458	0.85
DSJC250.9		**N_1**	**200**	**19/20**	**6.27**	**163**	**0.05**
n=250		N_2	50	15/20	5.4	262	0.25
m=27897	72	N_3	50	5/20	3.5	172	1.35
k^*=72		N_4	50	5/20	2.7	133	1.15
		N_5	175	14/20	7.8	232	0.3
DSJC500.1		N_1	150	7/20	88.5	182	1.05
n=500		N_2	50	7/20	100.8	338	1.35
m=24916	12	N_3	100	3/20	164.2	424	1.45
k^*=12		N_4	50	6/20	88.7	303	1.40
		N_5	**50**	**18/20**	**78.4**	**233**	**0.1**
DSJC500.5		**N_1**	**100**	**9/20**	**186.2**	**278**	**0.55**
n=500		N_2	50	5/20	270.1	519	1.15
m=125248	49	N_3	100	7/20	249.4	396	0.75
k^*=48		N_4	100	7/20	246.2	376	1.05
		N_5	150	6/20	494.6	660	0.95

TABLE 6 – Comparaison entre exécutions courtes (à droite) et exécutions longues (à gauche) pour `ID-VNS`.

[4] E. K. Burke, J. Marecek, A. J. Parkes, and H. Rudová. A supernodal formulation of vertex colouring with applications in course timetabling. *Annals OR*, 179(1) :105–130, 2010.

[5] C.Desrosiers, P.Galinier, and A. Hertz. Efficient algorithms for finding critical subgraphs. *Disc. Appl. Math.*, 156(2) :244–266, 2008.

[6] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN Symp. on Compiler Construction*, pages 98–105, 1982.

[7] Dominique de Werra. An introduction to timetabling. *European Journal of Operational Research*, 19(2) :151–162, 1985.

[8] C. Fleurent and J.A. Ferland. Genetic and hybrid algorithms for graph coloring. *Annals of Oper. Research*, 63(3) :437–461, 1996.

[9] P. Galinier and J.-K. Hao. Hybrid evolutionary algorithms for graph coloring. *J. Comb. Optim.*, 3(4) :379–397, 1999.

[10] P. Galinier and A. Hertz. A survey of local search methods for graph coloring. *Computers & OR*, 33 :2547–2562, 2006.

[11] M. Gamache, A. Hertz, and J. O. Ouellet. A graph coloring model for a feasibility problem in monthly crew scheduling with preferential bidding. *Computers & OR*, 34(8) :2384–2395, 2007.

[12] A. Gamst. Some lower bounds for a class of frequency assignment problems. *IEEE Trans. on Vehicular Technology*, 35 :8–14, 1986.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[14] Ian P. Gent. Arc consistency in SAT. In *ECAI*, pages 121–125, 2002.

[15] F. Glover and M. Laguna. *Tabu Search*. Kluwer, 1997.

[16] F. Herrmann and A. Hertz. Finding the chromatic number by means of critical graphs. *ACM J. of Exp. Algorithm.*, 7 :10, 2002.

[17] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4) :345–351, 1987.

[18] A. Hertz, M. Plumettaz, and N. Zufferey. Variable space search for graph coloring. *Disc. Appl. Math.*, 156(13) :2551–2560, 2008.

[19] F.T. Leighton. A graph coloring algorithm for large scheduling problems. *J. Res. Nat. Bureau of Standards*, 84(6) :489–503, 1979.

[20] A. Linhares and H. Yanasse. Search intensity versus search diversity : a false trade off ? *Appl. Intell.*, 32(3) :279–291, 2010.

[21] S. Loudni, P. Boizumault, and N. Levasseur. Advanced generic neighborhood heuristics for VNS. *Eng. Appl. of AI*, 23(5) :736–764, 2010.

[22] E. Malaguti, M. Monaci, and P. Toth. A metaheuristic approach for the vertex coloring problem. *INFORMS Journal on Computing*, 20(2) :302–316, 2008.

[23] E. Malaguti and P. Toth. A survey on vertex coloring problems. *Intl. Trans. in Op. Res.*, 17(1) :1475–3995, 2010.

[24] A. Mehrotra and M. Trick. A column generation approach for graph coloring. *INFORMS Journal on Computing*, 8(4) :344–354, 1996.

[25] I. Méndez-Díaz and P. Zabala. A branch-and-cut algorithm for graph coloring. *Disc. Appl. Math.*, 154(5) :826–847, 2006.

[26] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers And Operations Research*, 24 :1097–1100, 1997.

[27] C. Morgenstern. Distributed coloration neighborhood search. volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 335–357. American Mathematical Society, Providence, RI, USA, 1996.

[28] B. Neveu, G. Trombettoni, and Fred Glover. ID Walk : A candidate list strategy with a simple diversification device. In *Proceedings of CP'04, LNCS 3258*, pages 423–437, 2004.

[29] M. Trick. Computational symposium : Graph coloring and its generalizations. In *Cornell University, Ithaca, NY*, 2002. http ://mat.gsia.cmu.edu/COLOR02/.