Apport de la décomposition arborescente pour les méthodes de type VNS

Mathieu Fontaine

Université de Caen Basse-Normandie

École doctorale SIMEM

Thèse de doctorat

présentée et soutenue le : *04/07/2013*

par

Mathieu Fontaine

pour obtenir le

Doctorat de l'Université de Caen Basse-Normandie

Spécialité : Informatique et applications

Apport de la décomposition arborescente pour les méthodes de type VNS

Directeur de thèse : *Patrice Boizumault*

Christine Solnon	Professeur des Universités	INSA Lyon	(rapporteur)
Gérard Verfaillie	Directeur de Recherche	ONERA Toulouse	(rapporteur)
Philippe Jégou	Professeur des Universités	Université Marseille Nord	
Lakhdar Sais	Professeur des Universités	CRIL, Université d'Artois Lens	
Bertrand Neveu	Ingénieur en Chef ENPC	École des Ponts ParisTech	
Patrice Boizumault	Professeur des Universités	Université Caen Basse-Normandie	(directeur)
Samir Loudni	Maître de Conférences	Université Caen Basse-Normandie	(co-encadrant)

Mis en page avec la classe thloria.

Table des matières

Remerciements

Tous ces travaux n'auraient pas pu être menés sans toutes les personnes qui m'ont soutenu et aidé durant ces quatre années.

Je tiens tout d'abord à remercier mes encadrants, Patrice et Samir, sans qui cette thèse n'aurait pas pu avoir lieu.

Je remercie ensuite ma famille, soutien indéfectible durant toutes ces années d'études. Merci de m'avoir soutenu dans mes choix, et de m'avoir permis de devenir celui que je suis aujoud'hui. Merci à ma belle famille, qui m'a accueilli à bras ouverts, et qui m'ont aussi soutenu durant ces années.

Merci à tous mes amis du labo. Boris, qui m'a montré la voie pendant toutes ces années. Lamia, Arnaud, Nicolas, Guillaume, Abir, Benoit, Laëtitia, Grégory pour tous ces moments partagés. Merci à Cyril, Romain, et surtout à Jean-Philippe pour votre soutien et merci de m'avoir bousculé quand je le méritais bien. Merci à Pierre et Davy pour votre aide et votre patience. Merci aussi à Virginie et Agnès.

Merci à mes amis caenais. Thomas, Angèle, Wafa, Jérôme, Ludivine et Anthony, et plus particulièrement pour votre soutien dans les moments difficiles. Merci Frédéric pour m'avoir fait découvrir l'escalade et partagé ta passion, toujours avec le sourire. Merci Guillaume, Aurélien, Julien et François pour les soirées mémorables à la coloc. Merci François, Alicia, Robert et Anne-Lyne pour pour tous les bons moments ludiques que nous avons partagés. Merci à Solène et Cédrik, petits nouveaux malheureusement déjà sur le départ (on se reverra). Merci Nathalie, et tout le meilleur pour toi dans ton exil bretonnant (chanceuse).

Merci à mes amis ornais (et aux pièces rapportées). Nicolas, Nathalie, Lilian, Guillaume, Valentine, Tiphaine, Clément, Géraldine, Charles, Diego. Tous ces moments passés ensemble m'ont permis de garder les pieds sur terre et la tête froide. Merci Florie-anne, Renaud, et Ludovic, pour les parties de défouraillage de zombies et autres joyeusetés vidéoludiques.

Merci à mon meilleur ami, Fabien, qui ne m'a pas lâché depuis bientôt 25 ans. Merci aussi à Émilie. La meilleure auberge de transit de tout Paris. Vous avez toujours été là, je ne pourrai jamais assez vous remercier. Merci à toi aussi Kevin, le *good old fashioned lover boy*.

D'une manière générale, pour tous mes amis :

> Je gueule ; c'est vrai, j'suis un peu sec, tout ça, mais pour quelqu'un comme moi qui a facilement tendance à la dépression c'est très important ce que vous faites, parce que... Comment vous dire... C'est systématiquement débile mais c'est toujours inattendu. Et ça c'est très important pour la... la santé du... du cigare...
>
> *Alexandre Astier.*

Et enfin merci à Claire, qui m'inspire depuis près de 10 ans déjà, et avec qui j'espère continuer la route encore longtemps.

Introduction

1 Contexte

Le cadre des WCSP propose de modéliser les problèmes d'optimisation sous contraintes sous la forme d'un quadruplet $(X, D, W, \mathcal{S})$. X représente l'ensemble des variables x_i du problème. À chaque variable x_i est associé un domaine fini $D_i \in D$ qui contient l'ensemble des valeurs que peut prendre cette variable. L'ensemble W contient les fonctions de coût du problème et $\mathcal{S}$ est une structure de valuation. Elles modélisent les contraintes en associant, à chaque affectation, un coût entre 0 et $\top$. Le coût est nul si, et seulement si, l'affectation vérifie la contrainte. $\top$ est un entier naturel correspondant au coût maximal d'une affectation. Le but de la résolution est de trouver une affectation de toutes les variables qui minimise la somme des fonctions de coût.

1.1 Méthodes de résolution

Différentes méthodes de recherche arborescente ont été développées pour résoudre les WCSP [Verfaillie *et al.*, 1996, Larrosa et Meseguer, 1999]. La méthode de `Branch-and-Bound` proposée par [Kask et Dechter, 2001] utilise la notion de *mini-buckets*. [Larrosa et Schiex, 2003, Larrosa et Schiex, 2004, de Givry *et al.*, 2005] ont proposé une famille de méthodes utilisant une recherche de type `Depth-First Branch-and-Bound` maintenant différentes formes de cohérence locale.

Les méthodes de recherche arborescente permettent d'obtenir la ou les solutions optimales et de prouver l'optimalité de ces solutions. Cependant, pour les problèmes de grande taille, ces méthodes peuvent s'avérer trop gourmandes en temps de calcul et donc peu utiles en pratique, particulièrement si le temps de résolution est contraint.

À l'inverse, les méthodes de recherche locale visent à produire des solutions de bonne qualité en des temps de calcul raisonnables. Malheureusement, ces méthodes ne peuvent généralement pas garantir l'optimalité des solutions obtenues et ne sont pas toujours capables de s'échapper facilement des minima locaux. Les méta-heuristiques telles que `ID-Walk` [Neveu *et al.*, 2004] et `VNS/LDS+CP`, [Loudni et Boizumault, 2008] sont des méthodes qui contrôlent la recherche locale et guident celle-ci afin de sortir des minima locaux. Ces méthodes sont pour la plupart génériques et permettent la résolution approchée de problèmes combinatoires complexes.

1.2 Variable Neighborhood Decomposition Search

Variable Neighborhood Search (`VNS`, [Mladenovic et Hansen, 1997]) est une recherche à grand voisinage qui utilise plusieurs types de voisinages, dans le but de s'échapper des minima locaux. Bien que `VNS` soit très efficace, elle est toutefois moins performante sur des problèmes de grande taille.

Variable Neighborhood Decomposition Search (`VNDS`, [Hansen *et al.*, 2001]) est une extension de `VNS` qui vise à réduire l'espace de solutions parcouru. Pour cela, à chaque itération, la recherche

est effectuée uniquement sur un sous-problème déterminé heuristiquement. Une partie de la solution courante est alors désinstanciée, puis reconstruite afin d'obtenir une solution de meilleure qualité.

1.3 Décomposition arborescente

La décomposition arborescente d'un graphe, introduite par Robertson et Seymour dans [Robertson et Seymour, 1986], vise à partitionner un graphe en groupes de sommets, appelés clusters. Ces clusters forment un graphe acyclique qui constitue un arbre de jonction du graphe original.

Plusieurs méthodes de recherche arborescente telles que `BTD` [Terrioux et Jégou, 2003], `Lc-BTD`$^+$ [de Givry *et al.*, 2006], `RDS-BTD` [Sánchez *et al.*, 2009] et `DB` [Kitching et Bacchus, 2009] utilisent la notion de décomposition arborescente. L'intérêt de cette approche vient du fait que beaucoup de problèmes difficiles peuvent être résolus efficacement lorsque leur largeur de décomposition est faible.

2 Motivations et objectifs

Actuellement, la structure d'un problème est très peu prise en compte dans la résolution des problèmes de satisfaction et d'optimisation sous contraintes. Or, il existe de nombreux problèmes réels fortement structurés dont la décomposition arborescente peut s'avérer très profitable. Les travaux menés jusqu'à présent exploitent les décompositions arborescentes uniquement dans le cadre des méthodes de recherche complète.

L'objectif de cette thèse est d'étudier l'apport des techniques de décomposition arborescente dans les méthodes de recherche locale (et plus particulièrement dans les méthodes à voisinages étendus de type `VNS`), pour guider efficacement l'exploration de l'espace de recherche.

3 Contributions

1. Nous proposons, tout d'abord, **un premier schéma de recherche locale (`DGVNS`), exploitant la décomposition arborescente** pour guider efficacement l'exploration de l'espace de recherche [Fontaine *et al.*, 2011b]. Ce schéma utilise le graphe de clusters issu de la décomposition arborescente. Nous étudions et comparons trois différentes stratégies visant à mieux équilibrer l'intensification et la diversification au sein de `DGVNS` [Loudni *et al.*, 2013b].

2. Nous avons aussi constaté que :
- la plupart des instances étudiées contiennent des fonctions de coût beaucoup plus difficiles à satisfaire que d'autres.
- pour la plupart des instances étudiées, les décompositions obtenues comportent une proportion importante de clusters qui se chevauchent très fortement. Cette redondance entre clusters limite la diversification de `DGVNS`, en re-considérant des voisinages très proches.

Pour pallier à ces deux constats, **nous proposons deux raffinements de la décomposition arborescente**. Le premier, `TDTD` (*Tightness Dependent Tree Decomposition*), exploite la dureté des fonctions de coût pour identifier les parties du graphe de contraintes les plus difficiles à satisfaire. Le second raffinement cherche à augmenter la proportion de variables propres dans les clusters. À cet effet, nous proposons deux critères. Le premier consiste à fusionner les clusters partageant plus de variables qu'un seuil maximal fixé. Le second consiste à fusionner les clusters

ayant une proportion de variables partagées supérieure à un seuil maximal fixé [Fontaine *et al.*, 2011c, Fontaine *et al.*, 2013a].

3. Bien que nos propositions précédentes tirent parti de la topologie du graphe de clusters, elles ne prennent pas suffisamment en compte les séparateurs. En effet, l'affectation de toutes les variables d'un séparateur partitionne le problème initial en plusieurs sous-problèmes qui peuvent ensuite être résolus indépendamment.

Nous proposons deux extensions de `DGVNS`, notées `SGVNS` (*Separator-Guided* `VNS`) et `ISGVNS` (*Intensified* `SGVNS`), qui exploitent à la fois le graphe de clusters et les séparateurs. Notre idée est de tirer parti des affectations des variables des séparateurs pour guider `DGVNS` vers les clusters qui sont les plus susceptibles de conduire à des améliorations importantes (i.e. les clusters contenant, dans leurs séparateurs, des variables impliquées dans l'amélioration de la solution courante) [Loudni *et al.*, 2012a, Loudni *et al.*, 2012b].

4. Pour chaque contribution proposée, les expérimentations ont été menées sur différentes instances de quatre problèmes différents.

- Instances `RLFAP` : le `CELAR` (Centre d'Electronique de L'ARmement) a mis à disposition un ensemble d'instances du problème d'affectation de fréquences radio (`RLFAP`) [Cabon *et al.*, 1999]. L'objectif est d'assigner un nombre limité de fréquences à un ensemble de liens radios entre des couples de sites, afin de minimiser les interférences dues à la réutilisation de ces fréquences.

- Instances `GRAPH` : le générateur `GRAPH` a été développé par le projet CALMA [van Benthem, 1995] afin de proposer des instances aléatoires ayant une structure proche des instances `RLFAP`.

- Instances `SPOT5` : la planification quotidienne d'un satellite d'observation de la terre (`SPOT5`) consiste à sélectionner les prises de vue à effectuer dans la journée en prenant en compte les limites matérielles du satellite, tout en maximisant l'importance des photographies sélectionnées [Bensana *et al.*, 1999].

- Instances `tagSNP` : un `SNP` (*Single Nucleotide Polymorphism*) est la variation d'une seule paire de nucléotides dans l'ADN de deux individus d'une même espèce, ou dans une paire de chromosomes d'un même individu. Le problème de sélection des `tagSNP` consiste à déterminer un sous-ensemble de `SNP`, appelé `tagSNP`, qui permet de capturer le maximum d'information génétique.

Plan du mémoire

Le mémoire comporte deux parties : la première partie décrit l'état de l'art et la seconde présente nos contributions.

État de l'art. **Le chapitre 1.** présente la notion de WCSP, les différentes formes de cohérence ainsi que les méthodes de résolution associées. **Le chapitre 2.** introduit la notion de décomposition arborescente. Plusieurs méthodes exploitant la décomposition arborescente pour la résolution de CSP et de WCSP sont présentées. Notre choix d'utiliser la méthode de décomposition `MCS` (*Maximum Cardinality Search*, [Tarjan et Yannakakis, 1984a]) y est particulièrement motivé. **Le chapitre 3.** introduit la notion de recherche locale et dresse un panorama des méta-heuristiques pour la résolution des WCSP. Les méthodes `ID-Walk` et `VNS/LDS+CP`, qui nous serviront de référence pour évaluer nos contributions, y sont détaillées. Enfin, **le chapitre 4.** présente les différentes instances des problèmes `RLFAP`, `GRAPH`, `SPOT5` et `tagSNP` sur lesquelles nous avons réalisé nos expérimentations.

Contributions. **Le chapitre 5.** présente notre première contribution. Nous proposons un premier schéma de recherche locale, noté `DGVNS`, exploitant la décomposition arborescente pour guider efficacement l'exploration de l'espace de recherche (cf section 5.1). Ce schéma utilise la décomposition arborescente comme une carte du problème afin de construire des voisinages pertinents. Puis, nous étudions trois différentes stratégies visant à équilibrer l'intensification et la diversification (cf section 5.2). Les expérimentations menées sur différents jeux de test montrent la robustesse de notre approche sur des problèmes réels (cf section 5.3).

Le chapitre 6. présente notre seconde contribution. Nous proposons deux raffinements de la décomposition arborescente. Le premier raffinement, `TDTD` (cf section 6.1), exploite la dureté des foncions de coût pour identifier les parties du problème les plus difficiles à satisfaire. Le second raffinement (cf section 6.2) cherche à augmenter la proportion de variables propres dans les clusters, en fusionnant ceux ayant très peu de variables propres dans la décomposition arborescente. Nous montrons expérimentalement la pertinence et l'intérêt de ces deux raffinements (cf section 6.3).

Le chapitre 7. présente notre troisième contribution. Nous proposons deux extensions de `DGVNS` qui exploitent à la fois le graphe de clusters et les séparateurs entre ces clusters. Tout d'abord, nous présentons la méthode `SGVNS` (cf section 7.1), qui tire parti de l'évolution de la solution au cours de la recherche pour privilégier les clusters contenant, dans leurs séparateurs, des variables impliquées dans l'amélioration de la solution courante. Puis, nous détaillons la méthode `ISGVNS` (cf section 7.2), qui vise à intensifier la recherche dans les clusters contenant des variables réinstanciées. La section 7.3 présente nos résultats expérimentaux et l'intérêt d'exploiter la `TDTD` dans `SGVNS` et `ISGVNS` (cf section 7.3.5).

Le dernier chapitre présente nos conclusions et ouvre un certain nombre de perspectives dans la continuité de nos travaux.

Première partie

État de l'art

Chapitre 1

Réseaux de fonctions de coût

Sommaire

Les CSP (*Constraint Satisfaction Problems*) se limitent à prouver l'existence d'une ou plusieurs solutions, mais ne permettent pas l'expression de la qualité d'une solution, ni le calcul d'une solution dans le cas d'un problème sur-contraint. Ainsi, plusieurs extensions ont été proposées afin de définir des préférences entre les solutions lorsque une instance est satisfiable et de déterminer les contraintes à relaxer lorsque une instance est insatisfiable. Parmi celles-ci, nous pouvons citer [Fargier et Lang, 1993] permettant de modéliser les problèmes ayant des données incomplètes, [Rosenfeld *et al.*, 1976, Schiex, 1992, Ruttkay, 1994] permettant de définir des niveaux de préférences sur les tuples des contraintes, ou encore [Shapiro et Haralick, 1981, Freuder et Wallace, 1992] permettant de traiter les problèmes sur-contraints.

Le formalisme général des Problèmes de Satisfaction de Contraintes Valuées ($\mathcal{VCSP}$) [Schiex *et al.*, 1995] réunit les différents avantages des extensions citées précédemment [Bistarelli *et al.*, 1995]. Dans ce document, nous nous intéressons à une instance de ce modèle, les réseaux de fonctions de coût (*Cost Function Network* ou CFN) ou Weighted CSP (WCSP)[Larrosa, 2002].

Plan du chapitre. Nous commençons par introduire le formalisme des CSP afin de définir les notions de base. Nous présentons ensuite le cadre des $\mathcal{VCSP}$, la structure de valuations nécessaire

à l'expression des préférences, puis nous détaillons le formalisme des WCSP et les méthodes de recherche arborescente pour la résolution des WCSP. Enfin, nous terminons par les méthodes de reformulation de WCSP.

1.1 Le formalisme CSP

Un CSP est défini formellement de la façon suivante :

Définition 1 (*CSP*). *un CSP est un triplet* (X, D, C)*, avec :*
- X *l'ensemble des variables,*
- D *leurs domaines finis associés,*
- C *un ensemble de contraintes portant sur* X*.*

Nous définissons dans cette section les notions de variable, domaine et contrainte dans le cadre des CSP.

1.1.1 Variables et domaines

Soit X=$\{x_1, ..., x_n\}$ un ensemble fini de n variables. À chaque variable x_i est associée un domaine, noté D_i, représentant l'ensemble fini des valeurs pouvant être prises par cette variable. Le domaine d'une variable peut être numérique {1,3,4} ou symbolique {Matin,Soir,Repos}. On désigne l'ensemble des domaines par $D = \{D_1, ..., D_n\}$.

Définition 2 (Affectation d'une variable). *On appelle affectation d'une variable* x_i*, le fait d'attribuer à* x_i *une valeur de son domaine. L'affectation de la variable* x_i *à la valeur* a_j *est notée* (x_i, a_j)*.*

Une affectation complète $\mathcal{A}$=$\{(x_1, a_1)$,...,$(x_n, a_n)\}$ est une affectation de toutes les variables de X. Si au moins une variable n'est pas affectée, on parlera d'affectation partielle.

1.1.2 Contraintes

Soit $\mathcal{C}$ un ensemble contenant e contraintes. Chaque contrainte $c_S \in C$ porte sur un ensemble de variables $S \subseteq X$. Cet ensemble S de variables est appelé *portée* de la contrainte c_S.

Définition 3 (Voisinage d'une variable). *Le voisinage* $Vois(x_i)$ *d'une variable* x_i *est l'ensemble des variables comprises dans la portée d'une contrainte contenant* x_i*.*

$$Vois(x_i) = \{x_j | \exists\ c_S, (x_i\ \in S) \wedge (x_j \in S)\}$$

Une contrainte est une relation qui impose des conditions sur les valeurs qui peuvent être affectées aux variables de sa portée. Ces restrictions peuvent être exprimées de manière symbolique (par exemple $x_1 < x_2$), dans ce cas on parle de *contrainte en intention*. Elles peuvent aussi être exprimées sous la forme d'un ensemble de *tuples autorisés*, c'est-à-dire l'ensemble des affectations des variables satisfaisant la contrainte. On parle dans ce cas de *contrainte en extension* ou de *contrainte table*.

Définition 4 (Contrainte satisfaite). *Une contrainte* c_S *est dite* satisfaite ssi *les variables* $x_i \in S$ *sont complètement instanciées et forment un tuple vérifiant* c_S*.*

Soit les deux variables x_1 et x_2 de domaines $D_{x_1} = D_{x_2} = \{1, 2\}$, la contrainte $x_1 \neq x_2$ est satisfaite si x_1 est affectée à la valeur 1 et x_2 à 2 (ou inversement).

Un tuple autorisé est une affectation de toutes les variables de la portée d'une contrainte c_S satisfaisant celle-ci. Par opposition, un tuple interdit pour une contrainte c_S correspond à une affectation des variables de S ne satisfaisant pas c_S.

Les contraintes portant sur une seule variable sont dites unaires. Les variables portant sur deux variables, trois variables et n variables sont respectivement appelées binaires, ternaires et n-aires.

1.2 Le formalisme des WCSP

Nous présentons dans cette section les définitions de base. Nous précisons tout d'abord la notion de structure de valuations. Nous introduisons ensuite le cadre général des $\mathcal{VCSP}$. Enfin, nous définissons les particularités des WCSP par rapport à ce cadre.

1.2.1 Structure de valuations

Une structure de valuations permet de représenter les coûts (ou valuations) des tuples des contraintes ainsi que la manière de les agréger.

Définition 5 (Structure de valuations)**.** *Une structure de valuations, notée $\mathcal{S}$, est représentée par un quintuplet $(\mathcal{E}, \succ, \oplus, \bot, \top)$, avec :*
- *$\mathcal{E}$, l'ensemble des valuations associées aux tuples des contraintes, avec $\bot$ et $\top$ les valuations minimale et maximale,*
- *$\succ$, un opérateur d'ordre total sur les valuations de $\mathcal{E}$,*
- *$\oplus$, un opérateur d'agrégation des valuations possédant les propriétés suivantes :*
  - *commutativité : $\forall a, b \in \mathcal{E},\ a \oplus b = b \oplus a$,*
  - *monotonie : $\forall a, b, c \in \mathcal{E},\ a \succeq c \rightarrow (a \oplus b) \succeq (c \oplus b)$,*
  - *un élément absorbant : $\forall a \in \mathcal{E},\ a \oplus \top = \top$,*
  - *un élément neutre : $\forall a \in \mathcal{E},\ a \oplus \bot = a$.*

La commutativité de l'opérateur d'agrégation assure que l'ordre d'agrégation des valuations ne modifie pas le résultat. De même, la monotonie garantit que l'ajout d'insatisfactions (la génération d'un coût par l'affectation d'une variable) à une solution ne peut améliorer la qualité de celle-ci. $\top$ représente l'élément absorbant qui représente l'insatisfaction totale. A contrario, $\bot$ représente l'élément neutre associé à la satisfaction totale.

1.2.2 Le cadre général des $\mathcal{VCSP}$

Le formalisme des $\mathcal{VCSP}$ étend le formalisme des CSP en associant une valuation à chaque tuple de chaque contrainte. Pour un tuple, sa valuation exprime le degré d'insatisfaction engendré lorsque les variables de la contrainte sont affectées aux valeurs du tuple.

Définition 6 ($\mathcal{VCSP}$)**.** *un $\mathcal{VCSP}$ est défini par un quadruplet $(X, D, C, \mathcal{S})$, avec :*
- *X l'ensemble des variables,*
- *D leurs domaines finis associés,*
- *C un ensemble de contraintes valuées. Chaque contrainte valuée c est définie par une application c de $\prod_{x_i \in vars(c)} D_i$ dans $\mathcal{E}$ représentant les valuations des tuples de la contrainte,*

– $\mathcal{S}$ *une structure de valuations.*

Définition 7 (Projection)**.** *Étant donnée une affectation $\mathcal{A}$ et un ensemble de variables $S \subseteq X$, on note $\mathcal{A}[S]$ la projection des valeurs de $\mathcal{A}$ sur l'ensemble S.*

Définition 8 (Valuation d'une affectation complète)**.** *La valuation d'une affectation complète $\mathcal{A}$ est égale à l'agrégation par l'opérateur $\oplus$ des valuations des tuples des contraintes affectées par $\mathcal{A}$.*

$$\mathcal{V}(\mathcal{A}) = \bigoplus_{c \in \mathcal{C}} c(\mathcal{A}[vars(c)])$$

Dans la suite de ce document, nous nommerons *solution*, une affectation complète dont la valuation est strictement inférieure à $\top$. Une affectation complète est dite solution *optimale* si sa valuation est inférieure ou égale à la valuation de toute affectation complète. Une contrainte c complètement affectée pour une affectation $\mathcal{A}$, est dite satisfaite si $c(\mathcal{A}[vars(c)]) = \bot$ et violée sinon.

Pour une contrainte c_S, une affectation des variables de S dont la valuation est égale à $\bot$ est un tuple autorisé et interdit sinon.

Définition 9 (Tuple dur, tuple mou)**.** *Un tuple interdit dont la valuation est égale à $\top$ est dit dur et mou sinon.*

Définition 10 (Contrainte dure, contrainte molle)**.** *une contrainte dont tous les tuples interdits ont une valuation égale à $\top$ est dite* ***dure****; dans le cas contraire la contrainte est dite* ***molle****.*

Par abus de langage, nous dirons qu'une contrainte a une valuation égale à p lorsque tous ses tuples interdits ont une valuation de p.

1.2.3 Les WCSP

Le formalisme WCSP est une variante des $\mathcal{VCSP}$ dont l'objectif consiste à trouver une affectation complète qui minimise des fonctions de coût.

Définition 11 (Structure de valuations d'un WCSP)**.** *La structure de valuations associée à un WCSP est un quintuplet $\mathcal{S} = (E^+, \oplus, \leq, 0, \top)$ où $E^+ = [0 \dots \top]$ avec $\top \in \mathcal{N} \cup +\infty$ la valuation maximale associée à un tuple. $\oplus$ est une somme bornée dans $\mathcal{N}$:*

$$\forall a, b \in E^+, a \oplus b = min(a + b, \top)$$

Définition 12 (WCSP)**.** *Un WCSP est défini par un quadruplet $(X, D, W, \mathcal{S})$ où :*
– *$X = \{x_1, \dots, x_n\}$ est l'ensemble de variable du problème,*
– *$D = \{D_1, \dots, D_n\}$ l'ensemble des domaines finis associés aux variables,*
– *$W = \{w_{S_1}, \dots, w_{S_e}\}$ l'ensemble des fonctions de coût,*
– *$\mathcal{S}$ une structure de valuation.*

Définition 13 (Fonction de coût)**.** *Une fonction de coût w_{S_i}, définie sur un ensemble $S_i \subset X$ appelé portée, associe à chaque affectation des variables de sa portée un coût :*

$$w_{S_i} : (\prod_{x_k \in S_i} D_k) \rightarrow E^+$$

On appelle arité de w_{S_i} la taille de sa portée ($|S_i|$).

On associe au WCSP une fonction $w_\emptyset$ d'arité nulle. C'est une constante ajoutée à toute affectation et sert de minorant utilisé par DFBB (voir section 1.3.1).

Définition 14 (Coût d'une affectation dans un WCSP)**.** *Le coût d'une affectation complète* $\mathcal{A} = \{(x_1, a_1), \ldots, (x_n, a_n)\}$ *est donnée par :*

$$f(\mathcal{A}) = w_\emptyset \bigoplus_{i=1}^{e} w_{S_i}(\mathcal{A}[S_i])$$

L'objectif est de trouver une affectation complète $\mathcal{A}$ de coût $f(\mathcal{A})$ minimal.

Un WCSP peut être représenté sous la forme d'un graphe non orienté.

Définition 15 (Graphe de contraintes d'un WCSP)**.** *Soit* $\mathcal{P} = (X, D, W, \mathcal{S})$ *un WCSP, le graphe* $G = (V, E)$ *associé à* $\mathcal{P}$ *est défini par :*
- *un sommet* u_i *est associé à chaque variable* $x_i \in X$ *;*
- $\{u_i, u_j\} \in E$ *si, et seulement si,* $\exists\ w_S \in W, x_i, x_j \in S$*.*

Par abus de langage, on appelle le graphe associé à un WCSP son *graphe de contraintes*. La figure 1.1 présente un exemple de WCSP et son graphe de contraintes. Il est composé de six variables ($X = \{A, B, C, D, E, F\}$) et de cinq fonctions de coût ($W = \{w_{A,B,E}, w_{AC}, w_{C,D}, w_{B,D}, w_{D,F}\}$). L'affectation complète $\mathcal{A} = \{(A, a), (B, a), (C, a), (D, c), (E, b), (F, c)\}$ est une solution optimale de coût 20.

soit $\mathcal{P} = (X, D, W, \mathcal{S})$ avec :
- $X = \{A, B, C, D, E, F\}$,
- $D = \{\{a, b\}, \{a, b\}, \{a, c\}, \{b, c\}, \{b, c\}, \{a, c\}\}$,
- $W = \{w_{A,B,E}, w_{AC}, w_{C,D}, w_{B,D}, w_{D,F}\}$.

A	B	E	$w_{A,B,E}$
a	a	b	10
a	b	b	10
a	a	c	10
a	b	c	100
b	a	b	500
b	b	b	200
b	a	c	0
b	b	c	0

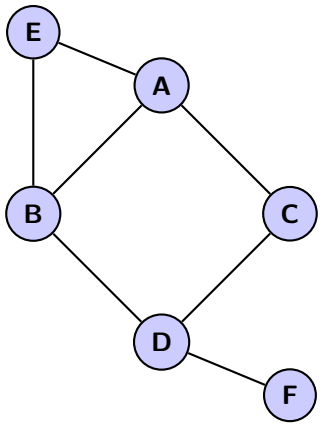
A	C	$w_{A,C}$
a	a	0
a	c	0
b	a	1,000
b	c	0

B	D	$w_{B,D}$
a	b	1,000
a	c	0
b	b	0
b	c	10

C	D	$w_{C,D}$
a	b	10
c	b	120
a	c	0
c	c	350

D	F	$w_{D,F}$
b	a	1,000
b	c	1,000
c	a	1,000
c	c	10

(a) Tables de fonctions de coût du problème.



(b) Graphe de contraintes.

FIGURE 1.1 – Exemple de WCSP.

1.3 Méthodes de recherche arborescente pour les WCSP

Nous présentons dans cette section deux méthodes de recherche arborescente. La première, *Depth First Branch and Bound*, est une recherche complète en profondeur d'abord utilisée dans

Algorithme 1 : DFBB(Y, $\mathcal{A}$, $\mathcal{UB}$)

```
1  début
2      LB ← f(A) ;
3      si LB < UB alors
4          si A est une affectation complète alors
5              retourner LB ;
6          Choisir une variable x_i ∈ Y ;
7          pour tout a_j ∈ D_i faire
8              c ← DFBB(Y\x_i, A ∪ (x_i ← a_j), UB) ;
9              si c < UB alors
10                 UB ← c ;
11     retourner UB
```

plusieurs méthodes de résolution [Larrosa et Schiex, 2003, Larrosa et Schiex, 2004, de Givry *et al.*, 2005]. La seconde, *Limited discrepancy search*, propose une recherche partielle en profondeur basée sur la notion d'écart à une heuristique fixée.

1.3.1 Depth First Branch and Bound (DFBB)

Le schéma classique de résolution d'un WCSP repose sur une recherche arborescente en profondeur d'abord basée sur le calcul d'un minorant, DFBB (Depth First Branch and Bound). À chaque noeud de l'arbre de recherche, un minorant ($\mathcal{LB}$) du coût de toutes les extensions (affectations complètes) issues de l'affectation partielle courante est calculé. Ce minorant est ensuite comparé au majorant du coût d'une solution optimale du problème. Celui ci correspond au coût de la meilleure solution trouvée à ce point de la recherche. Ainsi, si le minorant est supérieur au majorant, aucune extension de la solution partielle courante ne pourra être de meilleure qualité, et le sous-arbre peut être élagué.

L'algorithme 1 détaille le pseudo-code de DFBB. Le premier appel à DFBB est DFBB(X,$\emptyset$,$\top$). Soit un problème $\mathcal{P} = (X, D, W, \mathcal{S})$, DFBB est une fonction récursive prenant comme paramètres $Y \subseteq X$, l'ensemble des variables non affectées, une affectation partielle $\mathcal{A}$ et le majorant courant $\mathcal{UB}$. Tout d'abord, un minorant $f(\mathcal{A})$ de la solution partielle courante est calculé. Si celui est inférieur ou égal au majorant $\mathcal{UB}$, on teste tout d'abord si la solution courante est complète, auquel cas, $\mathcal{LB}$ est retourné. Sinon, une variable $x_i \in Y$ est sélectionnée, puis, pour chaque valeur a_j possible du domaine de x_i, DFBB est appelé en retirant x_i de Y et en affectant a_j à x_i. Le majorant $\mathcal{UB}$ est retourné. Il sera alors le plus petit coût trouvé sur les extensions de $\mathcal{A}$.

La complexité de DFBB est en $O(ed^n)$ avec d la taille du plus grand domaine, e le nombre de contraintes et n le nombre de variables. L'efficacité de DFBB dépend de la qualité du calcul du minorant $\mathcal{LB}$. Un minorant trivial consiste à sommer le coût des fonctions dont les variables sont affectées. Nous allons voir dans la section 1.4.2 comment calculer de bien meilleurs minorants.

1.3.2 Limited discrepancy search (LDS)

La recherche LDS a été introduite par William D. Harvey et Matthew L. Ginsberg [Harvey et Ginsberg, 1995]. C'est une recherche arborescente, tout comme DFBB, dans laquelle l'ordre

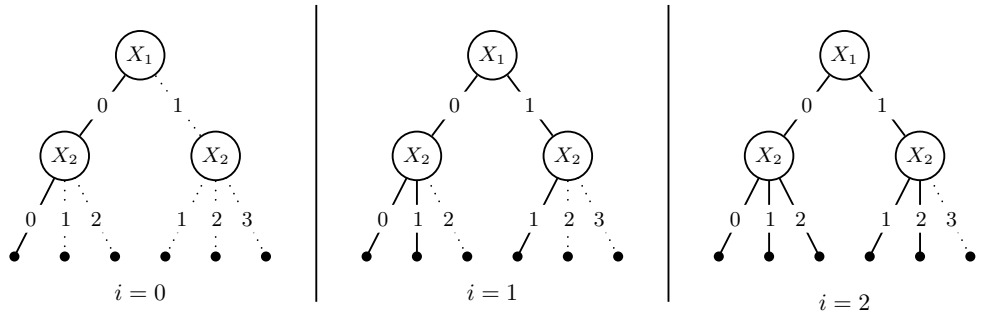


FIGURE 1.2 – Arbres de recherche explorés par LDS avec $\delta_{max} = 2$.

de visite des noeuds est modifié. Soit h une heuristique de choix de valeurs dans laquelle on a une grande confiance. Le principe de LDS est de suivre l'heuristique h lors du parcours de l'arbre de recherche, mais en permettant seulement un petit écart (δ) par rapport aux choix de h. On s'autorise donc δ écarts (ou *discrepancies*) à l'heuristique h. Pour un nombre maximal δ_{max} d'écarts autorisés, LDS explore l'arbre de recherche de manière itérative selon un nombre croissant d'écarts allant de i=0 à i=δ_{max}. À chaque itération, i est incrémenté de 1.

Dans la suite, nous ne considérons que des arbres d-aires. Dans un tel cas, l'heuristique h va ordonner toutes les valeurs du domaine d'une variable. La première valeur, selon l'ordre donné par h, n'engendre aucune augmentation de δ. La deuxième valeur augmente la valeur de δ de 1 et la troisième de 2, etc.

L'intérêt de LDS est d'explorer seulement les parties les plus intéressantes de l'arbre de recherche (selon les heuristiques) et non son intégralité comme le fait DFBB.

La figure 1.2 représente les branches de l'arbre de recherche exploré (en trait plein) lors des différentes itérations de LDS (avec δ_{max}=2).

1.4 Reformulation de $WCSP$

Les méthodes de reformulation de WCSP, aussi appelées méthodes d'inférence, ont pour but de transformer le problème afin de le rendre plus facile à résoudre. Nous présentons dans la section suivante deux approches d'inférence pour les WCSP. La première, dite exacte, consiste à éliminer des variables du problème afin de le réduire. La seconde approche, dite incomplète, est fondée sur la reformulation locale en un problème équivalent et consiste à déplacer les coûts entre les fonctions et les variables pour faire apparaître un bon minorant de l'optimum du problème.

1.4.1 *Élimination de Variable* (VE)

L'objectif de la méthode d'élimination de variable (*Variable Elimination* (VE) [Dechter, 1999]) est d'éliminer progressivement les variables du problème en préservant le coût de la solution optimale. La méthode repose sur deux opérateurs : l'addition et la projection de fonctions.

Définition 16 (Addition de deux fonctions)**.** *Soient deux ensemble de variables S_1 et S_2 et deux fonctions w_{S_1} et w_{S_2} définies respectivement sur S_1 et S_2. L'addition des deux fonctions $(w_{S_1} \oplus w_{S_2})$ définie sur l'ensemble $S_1 \cup S_2$ est définie par*

$$\forall \mathcal{A} \in D_{S_1 \cup S2}, (w_{S_1} \oplus w_{S_2})(\mathcal{A}) = w_{S_1}(\mathcal{A}[S_1]) \oplus w_{S_2}(\mathcal{A}[S_2])$$

Définition 17 (Projection d'une fonction)**.** *Soient deux ensemble de variables S et S' et une fonctions w_S définie sur l'ensemble S. La projection de la fonction w_S sur un sous-ensemble de ses variables $S' \subset S$, notée $w_S[S']$ est définie par*

$$\forall \mathcal{B} \in D_{S'}, w_S[S'](\mathcal{B}) = \min_{\mathcal{A} \in D_S,\ \mathcal{A}[S'] = \mathcal{B}} w_S(\mathcal{A})$$

Ainsi, l'élimination d'une variable x_i dans un problème $\mathcal{P} = (X, D, W, \mathcal{S})$ s'effectue en trois étapes :

1. recherche de toutes les fonctions de coût w_S telle que $x_i \in S$.
2. ajout d'une fonction de coût w'_S dans W, portant sur les variables $x_i \cup Vois(x_i)$ dans $\mathcal{P}$, résultant de la somme de toutes les fonctions identifiées en 1.
3. projection de w'_S sur l'ensemble $Vois(x_i)$.

L'algorithme se termine lorsque toutes les variables ont été supprimées. À la fin, le réseau ne contient que la fonction de coût $w_\emptyset$, égale au coût de la solution optimale du problème.

L'efficacité de cette méthode dépend fortement de l'ordre dans lequel les variables sont éliminées. En effet, chaque élimination de variable crée une nouvelle fonction de coût dont l'arité correspond au nombre de variables voisines de celle éliminée dans le réseau. La complexité est exponentielle par rapport au nombre maximal de variables voisines d'une variable éliminée. Trouver un ordre d'élimination optimal est un problème NP-difficile.

1.4.2 Mécanismes de cohérence par transformations du problème

L'arc cohérence est un mécanisme important dans la résolution des CSP puisqu'elle permet d'accélérer la recherche de solutions. Il semble alors naturel de l'étendre aux $WCSP$. Le but principal est de permettre la déduction de valeurs impossibles et de fournir à la recherche un minorant le plus précis possible afin d'arrêter au plus tôt la recherche dans un sous arbre ne pouvant conduire à une meilleure solution. Pour cela, on recherche une *reformulation* du WCSP préservant l'équivalence, dans laquelle on va transférer les coûts des fonctions de coût vers la fonction d'arité nulle $w_\emptyset$.

Cette section traite d'établissement de la cohérence par transformations préservant l'équivalence. Le principe de ces méthodes consiste à déplacer les coûts entre fonctions afin de les concentrer sur $w_\emptyset$. Nous commençons par définir une notion d'équivalence pour les $WCSP$. Puis, nous introduisons un nouvel opérateur $\ominus$ et les algorithmes de projection permettant de transférer les coûts entre les fonctions, tout en préservant l'équivalence avec le problème initial. Enfin, nous décrivons différents mécanismes de cohérence basés sur ces notions.

Nous supposons par la suite l'existence d'une fonction de coût unaire portant sur chaque variable du problème. Dans cette partie, nous noterons w_i la fonction de coût unaire portant sur x_i.

1.4.2.1 Équivalence entre $WCSP$

La notion d'équivalence entre deux $WCSP$ peut être définie de la manière suivante.

Définition 18 (Équivalence entre $WCSP$, [Larrosa et Schiex, 2003])**.** *Soient $\mathcal{P}$ et $\mathcal{P}'$ deux $WCSP$ portant sur un même ensemble de variables. $\mathcal{P}$ et $\mathcal{P}'$ sont dits équivalents si, et seulement si, toutes les affectations complètes des variables sont associées à un même coût dans $\mathcal{P}$ et $\mathcal{P}'$.*

Afin de transférer les coûts entre fonctions de coût, l'opérateur de soustraction de coûts, noté $\ominus$, permet de « retirer » un coût d'une fonction, pour l'ajouter à une autre par projection ou extension. Ces mécanismes ont pour but de concentrer les coûts sur la fonction de coût $w_\emptyset$, tout en préservant l'équivalence avec le problème originel [Schiex, 2000].

Définition 19 (Opérateur de soustraction)**.** *L'opérateur $\ominus$ de soustraction est défini par :*

$$\forall a \geq b \in \mathcal{E}, a \ominus b = \begin{cases} \top \text{ si } a = \top \\ a - b \text{ sinon} \end{cases}$$

La première opération, la projection, vise à transférer un coût d'une fonction de coût vers la fonction de coût associée à une variable.

Définition 20 (Transformation par projection)**.** *Une transformation par projection déplace un coût α d'une fonction w_S vers une fonction unaire $w_i(a_j)$, avec $x_i \in S$ et $a_j \in D_i$.*

L'algorithme 2 présente le principe de la projection. Il prend en paramètre une fonction de coût w_S, la variable x_i sur laquelle on souhaite projeter le coût, a_j la valeur associée, et α le coût à projeter. Les coûts devant rester positifs, α ne peut pas dépasser le coût minimal d'un tuple dans w_S. Tout d'abord, pour chaque affectation $\mathcal{A}$ dans laquelle x_i est affectée à a_j, on retire α au coût associé à $\mathcal{A}$ dans w_S. On ajoute ensuite le coût projeté sur $w_i(a_j)$.

Algorithme 2 : Algorithme de projection.

```
1 procedure projection(w_S, x_i, a_j, α) ;
2 début
4     pour tout A tel que A[x_i] = a_j faire
5         w_s(A) ← w_s(A) ⊖ α ;
7     w_i(a_j) ← w_i(a_j) ⊕ α ;
```

La seconde opération, l'extension, est l'opération inverse de la projection. Elle vise à déplacer un coût d'une fonction unaire vers une autre fonction de coût. L'algorithme 3 présente ce traitement.

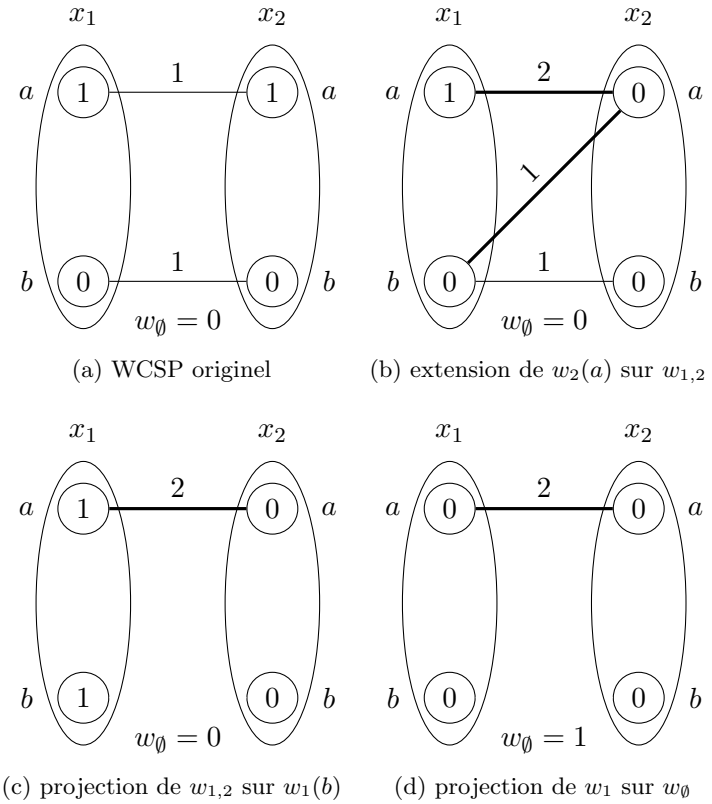
Définition 21 (Transformation par extension)**.** *Une transformation par extension déplace un coût α d'une fonction $w_i(x)$ vers une fonction w_S, avec $x_i \in S$ et $a_j \in D_i$.*

Algorithme 3 : Algorithme d'extension.

```
1 procedure extension(w_S, x_i, a_j, α) ;
2 début
4     pour tout A tel que A[x_i] = a_j faire
5         w_s(A) ← w_s(A) ⊕ α ;
7     w_i(a_j) ← w_i(a_j) ⊖ α ;
```

La figure 1.5 présente un exemple de projection et d'extension. On utilise pour cela le graphe de micro-structure du WCSP.

Définition 22 (Graphe de micro-structure)**.** *Le graphe $G = (V, E)$ de micro-structure d'un WCSP $(X, D, W, \mathcal{S})$ est un graphe construit de la manière suivante :*



(a) WCSP originel (b) extension de $w_2(a)$ sur $w_{1,2}$

(c) projection de $w_{1,2}$ sur $w_1(b)$ (d) projection de w_1 sur $w_\emptyset$

FIGURE 1.3 – Opérations d'extension et de projection.

- *Chaque sommet représente un couple variable x_i/valeur a_j ($a_j \in D_i$) ;*
- *Chaque sommet est étiqueté par $w_i(\{(x_i, a_j)\})$;*
- *Une arête est ajoutée entre deux sommets associés à (x_i, a_j) et (x_k, a_l) s'il existe une fonction $w_{i,k} \in W$ tel que $w_{i,k}(\{(x_i, a_j), (x_k, a_l)\}) > 0$;*
- *L'arête est étiquetée par $w_{i,k}(\{(x_i, a_j), (x_k, a_l)\})$.*

Soit $\mathcal{P}$ un $WCSP$ contenant deux variables x_1 et x_1 ayant pour domaine $\{a, b\}$ et 4 fonctions de coût : une fonction de coût binaire $w_{1,2}$, deux fonctions de coût unaires notées w_1 et w_2 portant respectivement sur x_1 et x_2 et la fonction de coût $w_\emptyset$. Le graphe de micro-structures de $\mathcal{P}$ est représenté figure 1.3a. Les valuations des fonctions de coût unaires sont représentées par les nombres inscrits à l'intérieur des domaines des variables. Par exemple, la fonction de coût unaire sur x_1 génère un coût de 1 pour l'affectation (x_1, a).

L'extension permet de déplacer les coûts, depuis une fonction de coût unaire, vers une fonction de coût binaire tout en préservant l'équivalence. Par exemple, l'extension de la fonction de coût unaire de x_2 vers $w_{1,2}$ permet de déplacer le coût lié à (x_2, a) sur $w_{1,2}$ (figure 1.3b).

La projection permet de déplacer les coûts minimaux d'une fonction de coût sur une fonction de coût unaire. Par exemple, la projection sur $w_1(b)$ de $w_{1,2}$ sur la figure 1.3c, permet d'incrémenter $w_1(b)$ de 1.

La projection permet aussi le déplacement des coûts minimaux d'une fonction de coût unaire vers la fonction $w_\emptyset$. Dans la figure 1.3d, le coût minimal de w_1 est transféré vers $w_\emptyset$.

1.4.3 Cohérences pour les $WCSP$

À partir de la notion d'équivalence, et des opérations de projection et d'extension, plusieurs notions de cohérence ont été proposées.

1.4.3.1 Cohérence de nœud (NC, [Larrosa, 2002])

La cohérence de noeud (*Node consistency*, NC) est une adaptation au WCSP de la cohérence de noeud des CSP.

Définition 23 (Noeud cohérence d'une valeur)**.** *Une valeur a d'une variable x_i est nœud cohérente (NC) si, et seulement si,*

$$w_i(a) + w_\emptyset < \top$$

Définition 24 (Noeud cohérence d'une variable)**.** *Une variable x_i est NC si, et seulement si,*
- *toutes les valeurs de son domaine sont nœud cohérentes ;*
- *$\exists\ a \in D_i$ telle que $w_i(a) = 0$.*

Un $WCSP$ est NC si, et seulement si, toutes ses variables le sont.

Tout $WCSP$ $\mathcal{P}$ peut être rendu nœud cohérent en projetant itérativement les valuations minimales des fonctions de coût unaires sur $w_\emptyset$. Puis, toute valeur qui n'est pas nœud cohérente est filtrée. Enfin, si aucun domaine n'est vide, $\mathcal{P}$ est alors NC. L'établissement de la nœud cohérence a une complexité temporelle de $O(nd)$.

1.4.3.2 Cohérence d'arc (AC, [Schiex, 2000])

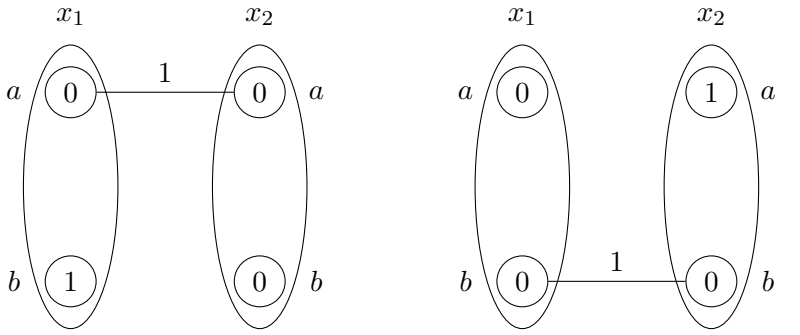
L'arc cohérence permet de filtrer les valeurs des domaines en tenant compte des fonctions de coût binaires. Elle se base sur la notion de support.

Définition 25 (Support d'une valeur pour une fonction de coût binaire)**.** *Soient w_{ij} une fonction de coût binaire et a une valeur de D_i. La valeur b de x_j est dite support de $(x_i = a)$ sur w_{ij} si, et seulement si, $w_{ij}(a, b) = 0$.*

Définition 26 (Arc cohérence d'une valeur)**.** *Une valeur a d'une variable x_i est arc cohérente pour une fonction de coût w_{ij} si, et seulement si, il existe au moins un support de a dans x_j.*

Une variable est arc cohérente (AC) si, et seulement si, elle est NC et que toutes ses valeurs sont AC. Enfin, un WCSP $\mathcal{P}$ est AC si et seulement toutes ses variables sont AC.

L'établissement de l'arc cohérence peut être effectuée en adaptant les mécanismes définis pour les CSP, afin de déplacer les coûts entre fonctions de coût par projection et extension. Ainsi, W-AC3 et W-AC2001, proposés par [Cooper et Schiex, 2004] et [Larrosa et Schiex, 2004] sont respectivement une extension de AC-3 et de AC-2001 au cadre des WCSP. Les complexités temporelles de W-AC3 et W-AC2001 sont respectivement de $O(ed^3)$ et $O(ed^2 + sd)$, avec e le nombre de fonctions de coût, d la taille maximale des domaines et s le nombre de tuples ayant une valuation différente de $\top$ et de 0. Ces deux algorithmes ont la même complexité spatiale en $O(nd)$. Toutefois, il est important de noter que, suivant l'ordre d'application des projections, le point fixe obtenu par ces algorithmes peut être différent. L'ordre d'application des projections joue donc un rôle important.



(a) WCSP $\mathcal{P}$ originel, x_1 est FAC (b) $\mathcal{P}'$, equivalent à $\mathcal{P}$, x_2 est FAC

FIGURE 1.4 – WCSP ne pouvant être rendu FAC.

1.4.3.3 Cohérence d'arc complète (FAC)

La notion de support complet permet de définir une cohérence plus forte.

Définition 27 (Support complet d'une valeur pour une fonction de coût binaire)**.** *Soient w_{ij} une fonction de coût binaire et a une valeur de D_i. La valeur b de la variable x_j est dite support complet de $(x_i = a)$ sur w_{ij} si, et seulement si, $w_{ij}(a, b) \oplus w_j(b) = 0$.*

Définition 28 (FAC d'une variable)**.** *Une variable x_i est complètement arc cohérente si, et seulement si,*

- *elle est NC ;*
- *pour toute fonction de coût binaire w_{ij}, toute valeur de son domaine possède un support complet dans x_j.*

Un WCSP $\mathcal{P}$ est complètement arc cohérent si, et seulement si, $\mathcal{P}$ est NC et que toutes ses variables sont FAC.

Cependant, un problème ne peut pas toujours être transformé en un problème FAC équivalent. En effet, sur la figure 1.4a, la variable x_2 n'est pas FAC, car la valeur a n'a pas de support complet dans x_1. En rendant x_2 FAC (figure 1.4b), c'est à dire en étendant la valuation de $w_1(b)$ sur w_{12}, puis en projetant w_{12} sur $w_2(a)$, on obtient un problème qui n'est plus FAC pour x_1 (la valeur b de x_1 n'ayant pas de support complet).

1.4.3.4 Cohérence d'arc directionnelle (DAC, [Cooper, 2003, Larrosa et Schiex, 2003])

Une solution, au problème précédemment énoncé, consiste à définir un ordre sur les variables. Pour chaque fonction de coût, la recherche d'un support complet est uniquement effectuée pour la variable la plus petite dans l'ordre.

Définition 29 (Cohérence d'arc directionnelle d'une variable)**.** *Pour un ordre α donné sur les variables d'un WCSP $\mathcal{P}$, une variable x_i est directionnelle arc cohérente (DAC) si, et seulement si,*

- *x_i est NC ;*
- *pour toute fonction de coût binaire w_{ij} telle que $x_i < x_j$ dans α, toute valeur de son domaine D_i possède un support complet dans x_j.*

Un WCSP $\mathcal{P}$ est DAC si, et seulement si, $\mathcal{P}$ est NC et que toutes ses variables sont DAC.

L'établissement de l'arc cohérence directionnelle a une complexité temporelle de $O(ed^2)$ et une complexité spatiale de $O(ed)$. Cependant, puisque l'arc cohérence directionnelle n'établit la propriété de cohérence que pour une variable de chaque fonction de coût, en la combinant avec AC pour la seconde variable, on obtient une propriété de cohérence plus forte, appelée FDAC.

1.4.3.5 Cohérence d'arc directionnelle complète (FDAC, [Cooper, 2003, Larrosa et Schiex, 2003])

Pour un ordre donné sur les variables, une variable est complètement et directionnellement arc cohérente (FDAC) si, et seulement si, elle est AC et DAC. Un $WCSP$ est FDAC si toutes ses variables le sont.

L'arc cohérence directionnelle complète peut être établie en $O(end^3)$ avec une complexité spatiale en $O(ed)$.

Dans [de Givry *et al.*, 2005], les auteurs constatent que la recherche de support complet peut être effectuée pour une fonction de coût binaire pour les deux variables à condition d'incrémenter $w_\emptyset$. Pour cela, ils définissent une nouvelle propriété de cohérence (EAC) pouvant être combinée avec les existantes.

1.4.3.6 Cohérence d'arc existentielle (EAC, [de Givry *et al.*, 2005])

Définition 30. *Une variable x_i est existentiellement arc cohérente (EAC) si, et seulement si,*
- *x_i est NC ;*
- *il existe au moins une valeur $a \in D_i$ ayant un support complet pour toute fonction de coût binaire w_{ij} portant sur x_i et telle que : $w_i(a) = 0$.*

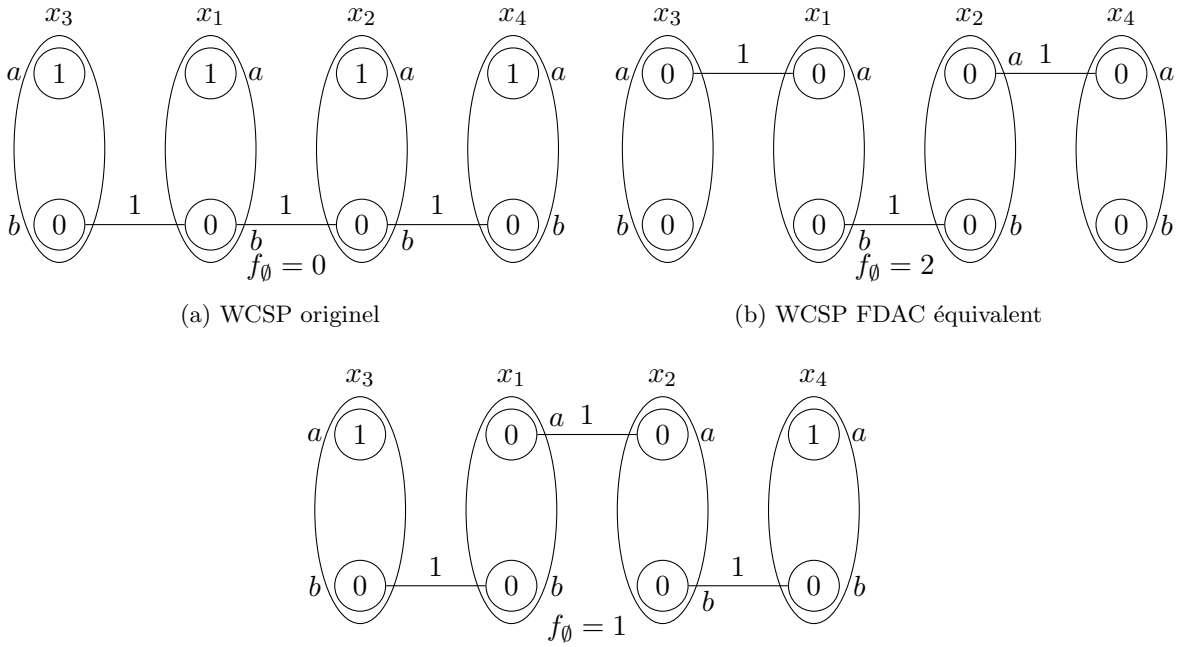
Un WCSP est EAC si, et seulement si, toutes ses variables le sont.

Les cohérences, précédemment définies, imposent que *chaque* valeur des domaines vérifie une propriété. Dans le cas de EAC, une seule valeur par domaine doit vérifier la propriété de cohérence. EAC peut être combinée avec les méthodes précédentes afin d'améliorer la propriété de cohérence de celles-ci.

1.4.3.7 Cohérence d'arc existentielle et directionnelle (EDAC, [de Givry *et al.*, 2005])

Un WCSP est EDAC si il est à la fois FDAC et EAC. La complexité temporelle de l'établissement d'EDAC est $O(ed^2 max\{nd, \top\})$.

Toutes ces cohérences peuvent être établies et maintenues heuristiquement pendant la recherche en temps polynomial dans le pire cas. Cependant, il est important de noter qu'il existe une hiérarchie entre elles. En effet, FAC et FDAC maintiennent une cohérence plus forte que AC. En théorie, FAC est plus forte que FDAC, mais il n'est pas toujours possible de rendre FAC un problème. En outre, EDAC maintient une cohérence plus forte que FDAC [Zytnicki *et al.*, 2005]. Cependant, puisqu'aucune de ces cohérences ne garantit l'unicité du point fixe, il est possible de générer une instance où FDAC serait meilleure que EDAC. En effet, le problème représenté sur la figure 1.5a, est rendu EDAC (figure 1.5c) et FDAC (figure 1.5b) pour un ordre lexicographique des variables (i.e. $x_1 < x_2 < x_3 < x_4$). Or, EDAC (resp. FDAC) prouve que toute affectation complète de $\mathcal{P}$ a un coût minimal de 1 (resp. 2).



(a) WCSP originel

(b) WCSP FDAC équivalent

(c) WCSP EDAC equivalent

FIGURE 1.5 – Exemple de cohérence EDAC (1.5c) plus faible qu'une cohérence FDAC (1.5b).

Il est à noter que DAC et EDAC ont été étendues aux fonctions de coût ternaires [Sanchez *et al.*, 2008] et à certaines fonctions de coût globales [Lee et Leung, 2009], mais la complexité augmente de manière exponentielle selon l'arité de la fonction de coût considérée.

1.4.4 Arc cohérences optimales pour les $WCSP$

Les algorithmes de cohérence précédents sont basés sur plusieurs critères heuristiques : l'ordre sur les variables et l'ordre des opérations de projections et d'extension. Suivant les heuristiques utilisées, le point fixe obtenu n'est pas toujours le même. Les méthodes suivantes permettent de s'affranchir de ces critères heuristiques et fournissent un minorant optimal (au sens de l'arc cohérence). La contrepartie est une complexité plus élevée.

1.4.4.1 Arc cohérence optimale (OSAC, [Cooper *et al.*, 2007])

Grâce à l'utilisation d'une structure de valuations autorisant les coûts en nombres rationnels, il est possible de calculer en temps polynomial, grâce à un programme linéaire, un ensemble d'opérations de projection permettant d'obtenir un problème équivalent où $w_\emptyset$ soit maximale. Cependant, l'établissement de cette cohérence ayant une complexité temporelle assez élevée ($O(poly(ed + n))$) en pratique, elle est souvent utilisée uniquement en phase de pré-traitement.

1.4.4.2 Arc cohérence virtuelle (VAC, [Cooper *et al.*, 2008])

L'arc cohérence virtuelle s'établit en appliquant itérativement trois phases. La première consiste à établir l'arc cohérence sur un CSP, noté $Bool(\mathcal{P})$ dont la structure est équivalente au WCSP $\mathcal{P}$ à résoudre et où les tuples des fonctions de coût ayant un coût de violation supérieur à 0 dans $\mathcal{P}$ sont interdits dans $Bool(\mathcal{P})$. Lorsqu'un domaine d'une variable de $Bool(\mathcal{P})$ est vide,

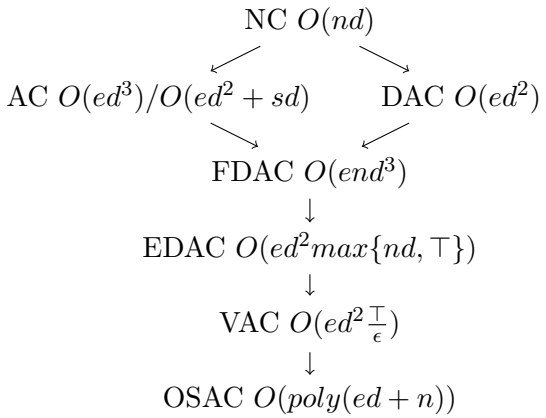


FIGURE 1.6 – Hiérarchie des cohérences d'arc pour les WCSP

une seconde phase retrace l'historique des propagations effectuées lors de la première phase, afin de déterminer un sous-ensemble de valeurs filtrées expliquant le domaine vide. La dernière phase modifie $\mathcal{P}$, en appliquant, dans l'ordre inverse, les transformations identifiées lors de la seconde phase. A chaque itération de ces trois phases, soit $\mathcal{P}$ est défini comme virtuellement arc cohérent (si $Bool(\mathcal{P})$ est arc cohérent à la fin de la première phase), soit $w_\emptyset$ est incrémenté. La complexité temporelle d'une itération des trois phases est de $O(ed^2)$. Le nombre d'itérations n'étant pas borné (l'incrémentation de $w_\emptyset$ pouvant devenir de plus en plus petite à chaque itération), un seuil minimal d'incrémentation, noté ϵ, est défini. Si ce seuil n'est pas atteint, l'algorithme s'arrête. On dit alors que $\mathcal{P}$ est VAC_ϵ. Le complexité temporelle totale de l'algorithme est $O(ed^2 \frac{\top}{\epsilon})$.

1.5 Hiérarchie des cohérences d'arc pour les WCSP

La figure 1.6 résume la hiérarchie des différentes cohérences d'arc en fonction de leur complexité pour des fonctions de coût binaires.

Nous utiliserons, pour nos expérimentations, EDAC qui fournit le meilleur compromis entre la qualité de la cohérence obtenue et la complexité temporelle de son calcul.

Chapitre 2

Méthodes de décomposition arborescente

Sommaire

De manière générale, les méthodes de résolution n'exploitent pas ou peu la structure du problème à résoudre. Cependant, l'étude de la structure du graphe de contraintes associé peut mettre en évidence des propriétés structurelles et topologiques intéressantes qui peuvent être exploitées pour guider de manière efficace la recherche de solutions. En effet, ces graphes sont souvent composés de groupes de variables formant des clusters, ces groupes n'étant eux-mêmes pas ou peu connectés entre eux.

La décomposition arborescente proposée par Robertson et Seymour dans [Robertson et Seymour, 1986] vise à partitionner un graphe en groupes de sommets, qu'on appelle clusters. Ces clusters forment un graphe acyclique qui constitue un arbre de jonction du graphe original. Plusieurs algorithmes de résolution, exploitant la structure du graphe de contraintes, utilisent la notion de décomposition arborescente. L'intérêt de cette approche vient du fait que beaucoup de problèmes difficiles peuvent être résolus efficacement si leur largeur de décomposition est faible.

Plan du chapitre. Tout d'abord, nous introduisons les notions relatives à la décomposition arborescente et les différentes méthodes de calcul de ces décompositions. Ensuite, nous présentons plusieurs méthodes exploitant les décompositions arborescentes pour la résolution de CSP et de WCSP. Enfin, nous concluons et motivons le choix de la méthode de décomposition que nous avons retenue.

2.1 Définitions

Les notions de cliques et de clusters permettent d'identifier des sous-parties du graphe de contraintes.

Définition 31 (Cluster, clique)**.** *Soit $G = (V, E)$ un graphe. Tout sous-ensemble $C \subseteq V$ de l'ensemble des sommets de G est appelé cluster de G. On dit que le cluster C est une clique ssi $\forall \{u, v\} \in C^2 \mid u \neq v, \{u, v\} \in E$. Si de plus $\forall z \in V \backslash C$, $C \cup \{z\}$ n'est pas une clique, alors C est une clique maximale de G.*

La décomposition arborescente vise à diviser un graphe en clusters et à organiser ces clusters sous la forme d'un arbre de jonction (ou arbre de clusters).

Définition 32 (Décomposition arborescente)**.** *Soit $G = (V, E)$ un graphe, on appelle décomposition arborescente de G un couple $(\mathcal{C}_T, T)$ où $T = (I, F)$ est un arbre avec I l'ensemble des sommets, F l'ensemble des arêtes et $\mathcal{C} = \{C_i | i \in I\}$ est une famille de sous-ensembles de V qui vérifie :*

- *$\bigcup_{i \in I} C_i = V$;*
- *$\forall \{v, w\}$ tel que $\{v, w\} \in E$, $\exists i \in I$ avec $v \in C_i$ et $w \in C_i$;*
- *$\forall (i, j, k) \in I^3$, si j est sur le chemin de i à k dans T alors $C_i \bigcap C_k \subseteq C_j$.*

Définition 33 (Largeur d'arbre d'un graphe G)**.** *On définit par $max_{i \in I}(|C_i| - 1)$ la largeur d'une décomposition. La largeur d'arbre d'un graphe G est la largeur minimale sur toutes ses décompositions arborescentes.*

Définition 34 (Séparateur)**.** *L'ensemble des variables partagées entre deux clusters C_i et C_j, noté $sep(C_i, C_j)$, est appelé séparateur :*

$$sep(C_i, C_j) = C_i \cap C_j$$

Définition 35 (Voisinage, degré)**.** *Soit C_i un cluster de $\mathcal{C}_T$. On appelle voisinage de C_i, noté $vois(C_i)$, l'ensemble des clusters C_j qui partagent des variables avec C_i.*

$$vois(C_i) = \{C_j | sep(C_i, C_j) \neq \emptyset\}$$

Le degrée d'un cluster C_i est le cardinal de $vois(C_i)$.

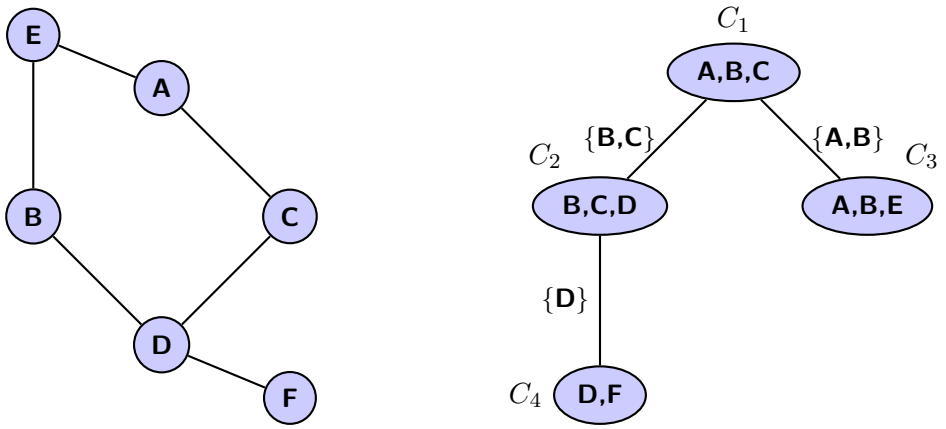
Définition 36 (Variable propre)**.** *On appelle variables propres de C_i l'ensemble $v_p(C_i)$ des variables qui n'appartiennent qu'au cluster C_i :*

$$v_p(C_i) = C_i \setminus \bigcup_{C_j \in vois(C_i)} sep(C_i, C_j)$$

Nous noterons $\texttt{parent}(C_i)$ (resp. $\texttt{fils}(C_i)$) le parent (resp. l'ensemble des fils) de C_i dans T.

La figure 2.1 donne un exemple de décomposition arborescente. Sa largeur d'arbre vaut 2. Chaque C_i correspond à un ensemble de sommets du graphe qu'on appelle **clusters**. Les arêtes entre les clusters sont étiquetées par l'ensemble des sommets communs aux clusters qu'elles relient. On appelle cet ensemble un **séparateur**.

Pour un graphe donné, il y a un grand nombre de décompositions arborescentes possibles. Calculer une décomposition de largeur minimale est un problème NP-difficile. Seules les décompositions de largeur proche de la largeur d'arbre vont nous intéresser. Il existe deux types



(a) Graphe original $G = (V, E)$.

(b) Décomposition arborescente de G de largeur 2.

FIGURE 2.1 – Exemple de décomposition arborescente.

de méthodes pour calculer une décomposition arborescente d'un graphe :

- La première méthode cherche à identifier les composants fortement connectés du graphe qui vont constituer les clusters de la décomposition, puis à former un arbre vérifiant les propriétés de la définition 32.
- La seconde méthode se base sur l'identification des séparateurs. Elle vise à trouver des ensembles de variables qui vont former des **coupes** minimales du graphe. Ces ensembles constitueront les séparateurs entre les clusters de la décomposition.

Nous allons présenter ces deux types de méthodes dans les sections suivantes.

2.2 Méthodes de décomposition basées sur la triangulation

2.2.1 Rappel du problème de triangulation

La triangulation consiste à transformer un graphe quelconque en un graphe cordal.

Définition 37 (Graphe cordal)**.** *Soit un graphe $G = (V, E)$, on dit que G est cordal ssi tout k-cycle ($k \geq 4$) de G possède une « corde », c'est-à-dire une arête reliant deux sommets non consécutifs.*

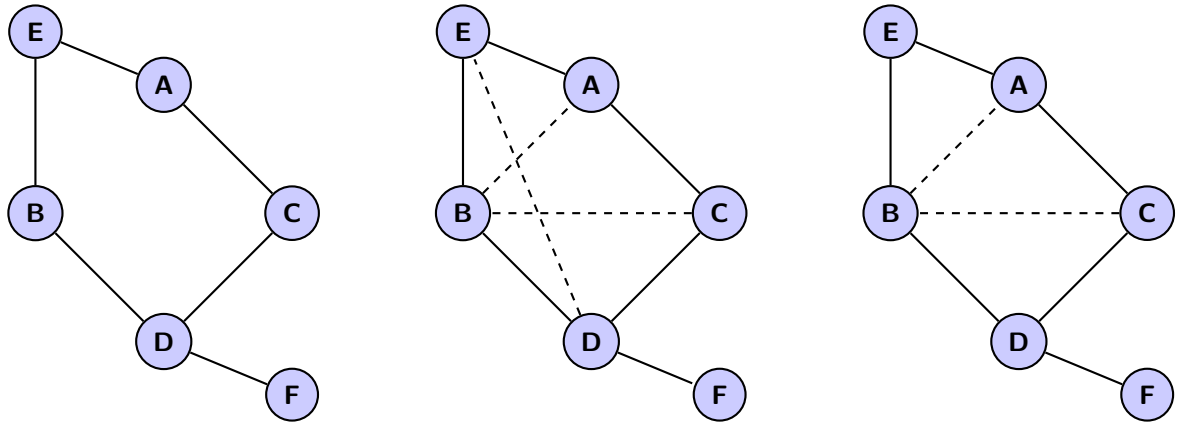
Définition 38 (Graphe triangulé)**.** *Un graphe est dit triangulé s'il est cordal.*

Une telle triangulation est notée G^*. Elle est obtenue en ajoutant des arêtes à G afin de casser tous les cycles de taille supérieure à 3. Une triangulation est dite minimum si elle ajoute un minimum d'arêtes au graphe G. Ce problème est NP-difficile [Yannakakis, 1981]. On s'intéresse donc à une variante polynomiale de ce problème, qui est la production de triangulations minimales.

Définition 39 (Triangulation minimale)**.** *Soit G^+ une triangulation de G, G^+ est minimale si, et seulement si :*

$$\nexists G'^+ \subseteq G^+, \text{ tel que } G'^+ \text{ est une triangulation de } G$$

La figure 2.2 présente, pour un graphe G (Fig. 2.2a), un exemple de triangulation G^*(Fig. 2.2b) et un exemple de triangulation minimale G^+(Fig. 2.2c).



(a) Graphe original $G = (V, E)$. (b) G^*, triangulation de G. (c) G^+, triangulation minimale de G.

FIGURE 2.2 – Exemple de triangulation.

2.2.2 De la triangulation à la décomposition arborescente

Le lien entre triangulation et décomposition arborescente est donné par la notion **d'arbre de jonction** :

Définition 40 (Arbre de jonction)**.** *Soit $G = (V, E)$ un graphe. Un arbre $T = (I, F)$, où chaque nœud $i \in I$ est associé à un sous-ensemble de sommets $V_i \subset V$, est un arbre de jonction de G ssi :*

- *$\{V_i | i \in I\}$ est l'ensemble des cliques maximales de G,*

– *pour tout sommet* $v \in V$, $T_v = \{i | v \in V_i\}$ *est un sous-arbre connexe de* T.

Le théorème suivant donne une caractérisation des graphes triangulés basée sur les arbres de jonction :

Théorème 1 ([Gavril, 1974])**.** *Un graphe G est triangulé si, et seulement si, il possède un arbre de jonction.*

Ce théorème fournit un moyen simple de trouver une décomposition arborescente dans le cas d'un graphe triangulé. En effet, un arbre de jonction d'un graphe G est une décomposition arborescente de G dont la largeur est égale à la taille de la plus grande clique de G. De plus, cette décomposition est de largeur minimale car la largeur d'arbre d'un graphe est minorée par la taille de sa plus grande clique. Ainsi, nous avons le théorème suivant :

Théorème 2. *Trouver la largeur d'arbre d'un graphe G est équivalent à trouver la triangulation de G dont la taille de la clique maximale est minimale.*

2.2.3 Décomposition arborescente d'un graphe quelconque

Le théorème 2 donne une indication pour aboutir à une décomposition à partir d'un graphe quelconque. D'après le théorème 1, un graphe non triangulé ne possède pas d'arbre de jonction. Cependant, dans [Koster *et al.*, 2001], les auteurs montrent que pour toute décomposition arborescente d'un graphe G quelconque, il existe une triangulation G^* de G qui admet la même décomposition. Ainsi, calculer la largeur de décomposition d'un graphe G quelconque revient à trouver une triangulation de G ayant une clique maximale de taille minimale. La largeur de décomposition d'une triangulation d'un graphe G est donc un majorant de la largeur de décomposition de G. On peut donc construire une décomposition arborescente d'un graphe non triangulé en produisant une triangulation de ce graphe. Cette décomposition est obtenue en trois phases :

1. triangulation du graphe,
2. construction du graphe de clusters,
3. construction de l'arbre de jonction à partir du graphe de clusters.

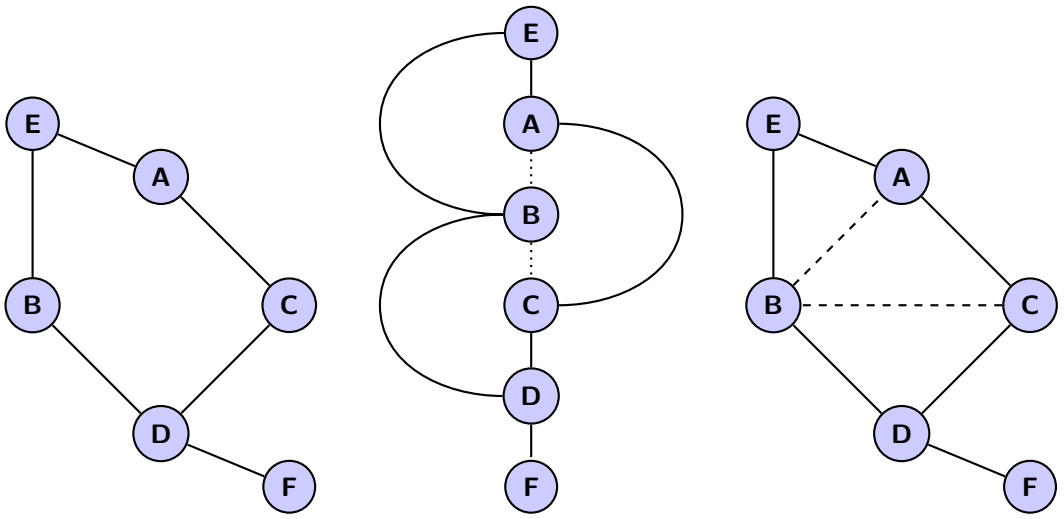
2.2.4 Algorithmes de triangulation

Comme nous l'avons indiqué dans la section précédente, pour trouver une décomposition arborescente d'un graphe, il faut calculer une triangulation de ce graphe. Un des moyens pour y parvenir est l'utilisation d'un ordre des sommets de G et la création d'un graphe d'élimination.

Définition 41 (Graphe d'élimination)**.** *Soit $G = (V, E)$ un graphe et α un ordre des sommets de G. Le graphe d'élimination de G selon α est le graphe produit en retirant successivement chaque sommet v de G dans l'ordre α et en formant une clique avec les voisins restants de v dans G (i.e. les sommets situés après v dans α).*

Ce procédé désigne le *jeu d'élimination* et les arêtes ainsi ajoutées sont appelées arêtes de *fill in*. Un ordre d'élimination de G est dit *parfait* si aucune arête n'est ajoutée lors de l'exécution du *jeu d'élimination*.

Il est montré dans [Fulkerson et Gross, 1965] que la classe des graphes possédant un ordre parfait est exactement la classe des graphes triangulés.



(a) Graphe original $G = (V, E)$. (b) Graphe d'élimination selon $\alpha = \{E, A, B, C, D, F\}$. (c) Graphe triangulé $G^* = (V, E')$.

FIGURE 2.3 – Exemple de graphe d'élimination.

Algorithme 4 : Jeu d'élimination

Données : un graphe $G = (V, E)$, un ordre α
Résultat : une triangulation G^*_α de G

1 **début**
2 $G^0 = G$;
3 **pour** $i = 1$ à n **faire**
4 soit v le sommet tel que $\alpha(v) = i$;
5 soit F^i l'ensemble des arêtes nécessaires pour faire de $N_{G^{i-1}}(v)$ une clique dans G^{i-1} ;
6 $G^i = G^{i-1}(V \setminus \{v\}, E \cup F^i)$;
7 $G^*_\alpha = (V, E \bigcup F^i)$

Théorème 3. *Un graphe est triangulé si, est seulement si, il a un ordre d'élimination parfait.*

Pour trianguler un graphe, il suffit donc de calculer son graphe d'élimination. L'algorithme 4 décrit le pseudo-code du jeu d'élimination. Il reçoit en entrée un graphe G et un ordre d'élimination α et produit en sortie un graphe G^*, triangulation de G.

Définition 42 (Ordre)**.** *Un ordre minimum (resp. minimal) est un ordre produisant une triangulation minimum (resp. minimal) par le jeu d'élimination.*

À chaque itération, l'algorithme sélectionne le sommet v d'indice i dans l'ordre α (ligne 4). Une clique est ensuite formée à partir des sommets voisins de v dans G^{i-1} (ligne 5) en ajoutant les arêtes de *fill-in* (i.e. F^i). Enfin, le sommet v est retiré du graphe G^{i-1} (ligne 6). L'algorithme s'arrête lorsque tous les sommets ont été traités et renvoie G^*_α, une triangulation de G (ligne 7).

La figure 2.3 présente une exemple de graphe d'élimination. Partant d'un graphe G (Fig. 2.3a) et d'un ordre $\alpha = \{E, A, B, C, D, F\}$, le jeu d'élimination produit un graphe d'élimination (Fig. 2.3b) en ajoutant les arêtes $\{A, B\}$ et $\{B, C\}$. On obtient alors le graphe G^* (Fig. 2.3c), triangulation de G.

Algorithme 5 : `Lex-BFS`

Données : un graphe $G = (V, E)$
Résultat : un ordre α et G^*_α

```
1 début
2     Pour tout sommet v de V, initialiser son étiquette e(v) = ∅ ;
3     pour i = n à 1 faire
4         Choisir le sommet v d'étiquette e(v) maximale dans l'ordre lexicographique ;
5         pour tout sommet non numéroté u de V faire
6             si {u, v} ∈ E alors
7                 e(u) = e(u) ∪ {i} ;
8         α(v) = i
9     jeu_élimination (G, α)
```

La qualité de la triangulation obtenue dépend uniquement de l'ordre d'élimination α. Trouver un ordre conduisant à une triangulation minimum est un problème d'optimisation à part entière. Plusieurs approches ont été proposées pour la résolution de ce problème. Elles peuvent se partitionner en 3 classes :

- Les méthodes complètes basées sur l'algorithme de *branch and bound* [Gogate et Dechter, 2004].
- Les méthodes incomplètes de type algorithme génétique [Larrañaga *et al.*, 1997], recuit simulé [Kjaerulff, 1992], etc.
- Les méthodes gloutonnes.

La complexité temporelle des deux premières classes d'algorithmes ne permet pas leur exploitation pratique. La troisième classe vise à donner une solution approchée de bonne qualité en un temps raisonnable. Ces algorithmes construisent l'ordre d'élimination en choisissant, à chaque étape, un sommet du graphe qui minimise la valeur d'une fonction heuristique. On s'intéressera donc à ces approches afin de produire des triangulations en des temps raisonnables.

2.2.4.1 `Lex-BFS`

Cet algorithme utilise une marquage lexicographique pour produire un ordre d'élimination. À chaque sommet v du graphe est associée une étiquette $e(v)$ représentant la liste de ses voisins déjà étiquetés, liste ordonnée de manière décroissante vis-à-vis de la position de ses voisins dans l'ordre d'élimination partiel. Son pseudo-code est décrit par l'algorithme 5.

Au départ chaque sommet reçoit une étiquette vide. À chaque itération, le sommet possédant la première étiquette dans l'ordre lexicographique reçoit l'indice i dans l'ordre α (i.e. $\alpha(v) = i$). L'indice i est ensuite ajouté à la fin de l'étiquette de tous les sommets non numérotés voisins de v. Enfin, pour produire une triangulation de G, on applique le jeu d'élimination en utilisant l'ordre d'élimination ainsi obtenu. La complexité de la triangulation est en $O(n + m)$, avec n le nombre de sommets et m le nombre d'arêtes.

2.2.4.2 `MCS`

`MCS` (*Maximum Cardinality Search*) [Tarjan et Yannakakis, 1984a] s'affranchit du marquage lexicographique en associant à chaque sommet un poids égal au nombre de ses voisins déjà visités. Ceci permet de réduire la complexité du tri des nœuds.

Algorithme 6 : `MCS`

```
   Données : un graphe G = (V, E)
   Résultat : un ordre minimal α et G*_α
1  début
2      F = ∅ ;
3      Pour tout sommet v de V, initialiser son poids w(v) = 0 ;
4      pour i = n à 1 faire
5          Choisir le sommet v de poids (w(v)) maximal ;
6          pour tout sommet non numéroté u de V faire
7              si {u, v} ∈ E alors
8                  incrémenter w(u) ;
9          α(v) = i
10     jeu_élimination (G, α)
```

L'algorithme 6 décrit le pseudo-code de `MCS`. À chaque itération, le sommet (noté v) qui a le plus grand nombre de voisins déjà choisis est sélectionné. Ensuite, le poids de chaque sommet non numéroté voisin de v est incrémenté de 1. L'algorithme s'arrête quand tous les sommets de G ont été numérotés. La complexité de la triangulation est en $O(n + m)$, avec n le nombre de sommets et m le nombre d'arêtes.

2.2.4.3 `Lex-M`

`Lex-M` [Rose *et al.*, 1976] est une extension de `Lex-BFS` qui vise à produire des triangulations minimales. Pour cela, il s'appuie sur la notion de *fill-path*.

Définition 43 (*fill-path*)**.** *On appelle fill-path un chemin $u, x_1, x_2, \ldots, x_k, v$ constitué uniquement de sommets non numérotés tel que $e(x_i) < e(u)$. L'arête $\{u, v\}$ est dite de fill-in.*

Comme `Lex-BFS`, à chaque itération, `Lex-M` (cf. Algorithme 7) sélectionne le sommet v ayant l'étiquette lexicographique la plus élevée (ligne 5). Toutefois, contrairement à `Lex-BFS`, `Lex-M` ajoute l'indice i dans l'ordre α (i.e. $\alpha(v) = i$) aux étiquettes de tous les sommets non numérotés voisins de v ou reliés à v par un *fill-path* (l'ensemble S, ligne 9). Ensuite chaque arête de *fill-in* $\{u, v\}$ est ajoutée à l'ensemble F. La complexité de la triangulation est en $O(n \times m)$, avec n le nombre de sommets et m le nombre d'arêtes.

2.2.4.4 `MCS-M`

Partant de `Lex-M` et `MCS`, le but de l'algorithme `MCS-M` [Berry *et al.*, 2004](algorithme 8) est de simplifier la production de triangulations en utilisant le marquage numérique des sommets. Il reprend le principe de `LEX-M` en remplaçant le tri lexicographique par une sélection par poids. Soit F l'ensemble des arêtes de *fill-in*. À chaque itération, l'algorithme sélectionne le sommet v de poids maximal (ligne 5), puis construit l'ensemble S constitué des voisins de v et des sommets reliés à v par un *fill-path* (ligne 9). Le poids de chaque sommet de S est incrémenté de 1 et l'arête $\{u, v\}$ est ajoutée à F (ligne 11). À la fin du marquage des sommets, les arêtes de F (de *fill-in*) sont ajoutées à G (ligne 15). La complexité de la triangulation est en $O(n \times m)$, avec n le nombre de sommets et m le nombre d'arêtes.

La figure 2.4 résume les liens entre les différentes heuristiques ainsi que leurs complexités.

Algorithme 7 : LEX-M

Données : un graphe $G = (V, E)$
Résultat : un ordre minimal α et G^+_α

début
  $F = \emptyset$;
  Pour chaque sommet v de V, initialiser son étiquette $e(v) = \emptyset$;
  pour $i = n$ à *1* **faire**
    Choisir le sommet v d'étiquette $(e(v))$ maximale dans l'ordre lexicographique ;
    $S = \emptyset$;
    pour tout *sommet non numéroté* u *de* V **faire**
      si $\{u, v\} \in E$ *ou* $\exists$ *un chemin* $u, x_1, .., x_k, v$ *tel que* $e(x_i) < e(u)$ *pour tout* $1 \leq i \leq k$ **alors**
        $S = S \cup \{u\}$;
    pour tout *sommet* $u \in S$ **faire**
      $e(u) = e(u) \cup \{i\}$;
      si $\{u, v\} \notin E$ **alors**
        $F = F \cup \{\{u, v\}\}$
    $\alpha(v) = i$
  $G^+_\alpha = (V, E \cup F)$

Remarque 1. *Aucun algorithme ne précise comment choisir le premier sommet de l'ordre. Cette question reste ouverte et, à notre connaissance, aucun critère n'a été proposé pour assurer une décomposition de largeur minimale.*

2.2.5 Recherche de cliques maximales

La seconde étape de l'algorithme de recherche d'une décomposition arborescente consiste à construire le graphe de jonction (le graphe de clusters) :

Définition 44 (Graphe de jonction)**.** *Soit G^* une triangulation de G et $\mathcal{C}$ l'ensemble de ses cliques maximales. Un graphe de jonction de G^* est un graphe étiqueté, noté $G_\mathcal{C} = (\mathcal{C}, E)$, dont les sommets sont les éléments de $\mathcal{C}$ et il existe une arête entre deux sommets C_i et C_j ssi $sep(C_i, C_j) \neq \emptyset$. Les arêtes sont étiquetées par les sommets en commun.*

La figure 2.5b donne le graphe de jonction associé au graphe triangulé de la figure 2.5a.

Pour identifier l'ensemble des cliques maximales, il suffit d'exploiter l'ordre calculé par l'heuristique de triangulation et le graphe triangulé G^*. En effet, pour un graphe triangulé et son ordre associé, tout sommet forme une clique avec tous ses voisins le suivant dans l'ordre. Il ne reste plus ensuite qu'à faire un test de maximalité.

2.2.6 Calcul de l'arbre de jonction

L'obtention de l'arbre de jonction à partir du graphe de jonction se fait grâce à la propriété suivante :

Théorème 4. *Un arbre couvrant de poids maximal d'un graphe de jonction est un arbre de jonction.*

Algorithme 8 : `MCS-M`

Données : un graphe $G = (V, E)$
Résultat : un ordre minimal α et G^+_α

```
1  début
2      F = ∅ ;
3      Pour tout sommet v de V, initialiser son poids w(v) = 0 ;
4      pour i = n à 1 faire
5          Choisir le sommet v de poids (w(v)) maximal ;
6          S = ∅ ;
7          pour tout sommet non numéroté u de V faire
8              si {u, v} ∈ E ou ∃ un chemin u, x_1, .., x_k, v tel que w(x_i) < w(u) pour tout 1 ≤ i ≤ k
               alors
9                  S = S ∪ {u} ;
10         pour tout u ∈ S faire
11             incrémenter w(u) ;
12             si {u, v} ∉ E alors
13                 F = F ∪ {{u, v}}
14         α(v) = i
15     G^+_α = (V, E ∪ F) ;
```

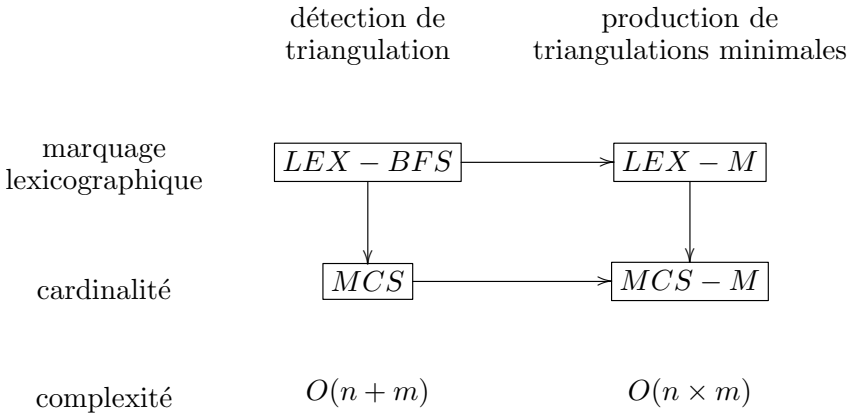
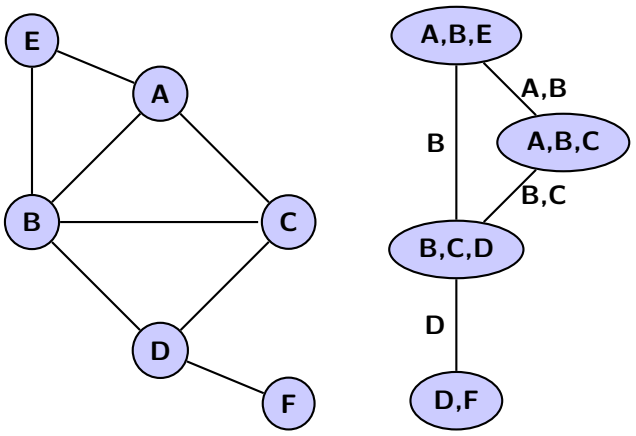


FIGURE 2.4 – Relations entre les différents algorithmes de triangulation.

La figure 2.6 présente un exemple d'arbre de jonction associé au graphe de clusters de la figure 2.5b.

Le principe de l'algorithme est de choisir, au fur et à mesure de la construction, l'arête contenant le plus grand nombre de sommets (cf. algorithme 9). On sélectionne tout d'abord un sommet u au hasard (ligne 2). Tant qu'il reste des sommets qui n'appartiennent pas à I, on sélectionne l'arête $\{v, w\}$ de poids maximal dont un seul sommet v appartient à I. On ajoute l'arête $\{v, w\}$ à F (l'ensemble des arêtes de l'arbre de jonction) et w à I. A la fin de l'algorithme, on obtient $T = (I, F)$, l'arbre de jonction de $G_\mathcal{C}$. Cet algorithme est linéaire en le nombre de sommets du graphe $G_\mathcal{C}$.



(a) graphe triangulé G^*. (b) Graphe de jonction associé a G^*.

FIGURE 2.5 – Exemple de graphe de jonction.

Algorithme 9 : Algorithme de calcul d'arbre couvrant de poids maximal

Données : un graphe étiqueté $G_{\mathcal{C}} = (\mathcal{C}, E)$
Résultat : Un arbre $T = (I, F)$ couvrant de poids maximal

```
1 début
2     soit I = {u} où u est un sommet choisi au hasard ;
3     soit F = ∅ ;
4     tant que il reste des sommets non sélectionnés dans G_C faire
5         choisir une arête {v, w} telle que v ∈ I et w ∉ I de poids maximal ;
6         ajouter {v, w} à F ;
7         ajouter w à I ;
```

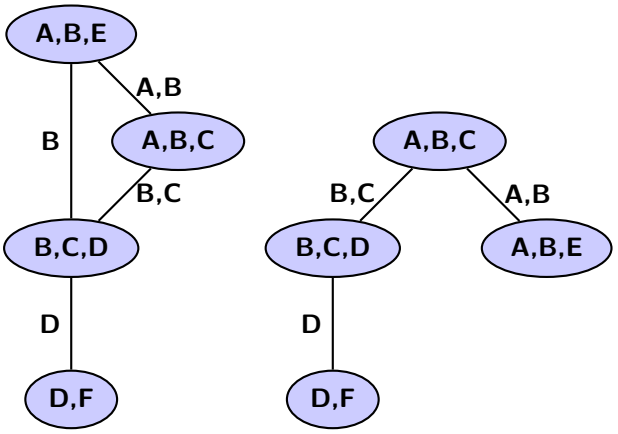


FIGURE 2.6 – Exemple de calcul d'arbre couvrant de poids maximum.

2.3 Une méthode basée sur les coupes minimales

Un second type de méthodes utilisées pour calculer la décomposition arborescente se concentre plutôt sur la recherche de séparateurs entre cliques. Les méthodes *Minimum Separating Vertex*

Set (`MSVS`) se basent sur la notion d'ensemble de sommets *st*-séparateur.

Définition 45. *Soit un graphe $G = (V, E)$ et deux nœuds $s, t \in V$, on appelle ensemble de sommets st-séparateur de G un ensemble $S \subseteq V \backslash \{s, t\}$ tel que tout chemin de s à t dans G passe par au moins un sommet de S. L'ensemble de sommets séparateur minimal de G est donné par l'ensemble st-séparateur de cardinalité minimale sur toutes les combinaisons possibles de sommets $s, t \in V$ avec $\{s, t\} \notin E$.*

Dans [Koster, 1999], l'auteur propose de rechercher des coupes minimales dans un graphe, c'est-à-dire des ensembles minimaux de sommets tels que si l'on retire les sommets d'un des ces ensembles, le graphe n'est plus connexe. L'algorithme proposé consiste, en partant d'un cluster contenant tous les sommets du graphe, à éclater récursivement les clusters jusqu'à obtenir des ensembles indivisibles, les cliques.

2.3.1 Recherche de coupes minimales dans un graphe

La recherche de coupes minimales repose sur le théorème suivant :

Théorème 5. *Soit un graphe $G = (V, E)$ et deux sommets $s, t \in V$ non adjacents, le nombre minimal de sommets d'un st-séparateur est égal au nombre maximal de chemins sommet-disjoints connectant s à t*

Définition 46 (chemins sommet-disjoints)**.** *On appelle chemins sommet-disjoints tout ensemble de chemins $\{ch_1, \ldots, ch_n\}$ reliant deux sommets u et v tel qu'aucun chemin ch_i ne passe par un sommet appartenant à un chemin ch_j.*

Il faut donc trouver le nombre maximal de chemins sommet-disjoints de G. Ce problème peut être résolu par un algorithme de flot. Pour cela, on crée à partir du graphe $G = (V, E)$ un graphe orienté $D = (V, A)$ avec $\forall \{u, v\} \in E, \{(u, v), (v, u)\} \in A$ et un graphe $D' = (V', A')$ orienté construit de la façon suivante :

- à chaque sommet v de G on associe deux sommets $v', v'' \in V'$,
- chaque arc de D dirigé vers v est redirigé vers v' avec un poids infini (∞),
- chaque arc de D partant de v est redirigé vers v'' avec un poids infini (∞),
- on ajoute un arc de v' à v'' avec un poids de 1.

On applique ensuite l'algorithme d'Edmonds-Karp [Edmonds et Karp, 1972] pour connaître le nombre de sommets *st*-séparateurs minimaux. Ce traitement est réalisé pour chaque paire de sommets non adjacents $s, t \in V$. On calcule ensuite l'ensemble *st*-séparateur de cardinalité minimale parmi toutes les paires de sommets. Selon la définition 45, cet ensemble correspond à une coupe minimale du graphe G.

2.3.2 Calcul de la décomposition arborescente à partir des coupes minimales

L'algorithme 10 présente le pseudo-code de `MSVS`. Pour construire la décomposition d'un graphe $G = (V, E)$, `MSVS` commence avec une décomposition comprenant un seul cluster contenant tous les sommets de G. Tant que tous les clusters de la décomposition ne sont pas des cliques, l'algorithme choisit un cluster C_i décomposable et calcule la coupe minimale S du graphe induit par C_i dans G. Il crée ensuite un nouveau cluster pour chaque composante connexe induite par S dans C_i. Enfin, l'arbre T est recomposé. A la fin de l'algorithme, on obtient une décomposition arborescente de G. Pour une description plus détaillée, le lecteur pourra consulter [Koster, 1999].

Algorithme 10 : `MSVS`

Données : un graphe $G = (V, E)$
Résultat : Une décomposition arborescente $(\mathcal{C}, T)$

```
1 début
2     C_0 = V ;
3     C = C_0 ;
4     T = (∅, ∅) ;
5     tant que il existe un cluster C_i décomposable faire
6         calculer la coupe minimale S de C_i ;
7         créer pour chaque composante connexe de C_i\S un cluster ;
8         recomposer l'arbre T et l'ensemble C ;
```

2.3.3 Comparaison entre `MSVS` et la méthode basée sur la triangulation

Dans [Koster *et al.*, 2001], l'auteur compare `MSVS` et plusieurs heuristiques de triangulation pour calculer une décomposition arborescente. Il ressort de cette étude que, malgré la qualité des décompositions obtenues par `MSVS` en termes de largeur de décomposition, les temps de calcul de cet algorithme le rendent peu utilisable en pratique. En revanche, la méthode basée sur la triangulation donne des résultats de bonne qualité en des temps raisonnables.

2.4 Évaluation des heuristiques de triangulation

L'évaluation des heuristiques de triangulation proposée dans [Koster *et al.*, 2001] compare uniquement les trois heuristiques `MCS`, `Lex-M` et `Lex-BFS`. Nous proposons donc d'étendre cette étude comparative en incluant l'heuristique `MCS-M`. De plus, nous élargissons les expérimentations à différentes instances de problèmes qui seront présentés au chapitre 4.

2.4.1 Protocole expérimental

Afin de comparer les performances des heuristiques de triangulation, leur implantation a été réalisée en C++. Les expérimentations ont été effectuées sur un serveur de calcul sous linux avec un AMD OPTERON 2,1 GHz et 256 Go de RAM. Les résultats sont obtenus selon le protocole suivant :

- pour chaque instance, on choisit le premier sommet de l'ordre d'élimination et on calcule la triangulation associée,
- pour chaque instance et chaque méthode, nous reportons la largeur de décomposition aborescente minimale, maximale et moyenne ainsi que le temps de calcul moyen de la décomposition sur l'ensemble des triangulations produites.

2.4.2 Résultats

Le tableau 2.1 présente les résultats sur les instances `RLFAP` (cf section 4.1). En moyenne, `MCS` obtient les meilleurs résultats aussi bien en termes de temps de calcul sur toutes les instances qu'en termes de largeur de décomposition sur les instances Scen06 et Scen08. Sur la Scen07, `MCS` reste très compétitive comparée à `Lex-BFS` qui posséde un meilleur comportement moyen. Notons également que `Lex-BFS` est légèrement meilleure en termes de largeur de décomposition minimale que `MCS`. Les deux autres heuristiques (`Lex-M` et `MCS-M`) sont moins performantes, avec respectivement des temps de calcul 3 fois et 10 fois supérieurs à `Lex-BFS` et `MCS` sur les instances

Instance	Méthode	Largeur de décomposition			Temps CPU
		min.	moy.	max	
Scen06	Lex-BFS	**11**	13.7	16	**0.1**
	MCS	**11**	**11.9**	17	**0.1**
	MCS-M	**11**	13.6	**14**	0.3
	Lex-M	**11**	13.6	**14**	0.3
Scen07	Lex-BFS	18	**22.8**	**29**	**0.2**
	MCS	19	25.6	30	**0.2**
	MCS-M	**17**	25.3	31	1.5
	Lex-M	**17**	25.3	31	1.8
Scen08	Lex-BFS	**19**	28.9	36	0.7
	MCS	22	**26.9**	**33**	**0.6**
	MCS-M	20	30.1	39	10.8
	Lex-M	20	30.1	39	8.5

TABLE 2.1 – Comparaison des heuristiques de triangulation sur les instances RLFAP.

Scen06 et Scen08 respectivement. De plus, les décompositions obtenues ont des largeurs moyennes de moins bonne qualité. Toutefois, Lex-M et MCS-M obtiennent des décompositions de largeur minimale sur 2 instances (Scen06 et Scen07).

Instance	Méthode	Largeur de décomposition			Temps CPU
		min.	moy.	max	
Graph05	Lex-BFS	30	36.6	45	0.3
	MCS	30	**34.3**	**38**	**0.3**
	MCS-M	**27**	35.3	44	0.5
	Lex-M	**27**	35.3	44	0.6
Graph06	Lex-BFS	59	71.8	84	3.2
	MCS	**56**	**62.5**	**75**	**2.4**
	MCS-M	59	70.8	85	4.7
	Lex-M	59	70.8	85	5.5
Graph07	Lex-BFS	31	38.3	47	0.5
	MCS	**30**	**37.3**	**46**	**0.4**
	MCS-M	**30**	**37.3**	48	1.1
	Lex-M	**30**	**37.3**	48	1.2
Graph11	Lex-BFS	107	121.6	139	23.6
	MCS	104	**113.6**	**127**	**15.4**
	MCS-M	**103**	120.4	140	32.1
	Lex-M	**103**	120.4	140	38
Graph12	Lex-BFS	54	63.0	73	4.1
	MCS	55	**60.8**	**70**	**2.3**
	MCS-M	**54**	62.3	74	8.3
	Lex-M	**54**	62.3	74	9.9
Graph13	Lex-BFS	149	160.8	176	109.1
	MCS	**142**	**152.2**	**164**	**45.8**
	MCS-M	**142**	160.4	180	122.3
	Lex-M	**142**	160.4	180	142

Instance	Méthode	Largeur de décomposition			Temps CPU
		min.	moy.	max	
#408	Lex-BFS	**38**	44.5	53	1.3
	MCS	43	**43.9**	**50**	**0.5**
	MCS-M	39	43.9	62	1.5
	Lex-M	39	43.9	62	2.3
#412	Lex-BFS	40	50.2	87	2.1
	MCS	43	**44.0**	**50**	**1.1**
	MCS-M	**39**	48.0	82	5.6
	Lex-M	**39**	48.0	82	8.1
#414	Lex-BFS	86	103.2	**114**	54.7
	MCS	108	111.6	118	51.8
	MCS-M	**85**	**96.0**	118	**41.4**
	Lex-M	**85**	**96.0**	118	50.9
#505	Lex-BFS	27	32.7	42	0.8
	MCS	31	33.5	**37**	**0.6**
	MCS-M	**26**	**30.0**	46	2.6
	Lex-M	**26**	**30.0**	46	3.4
#507	Lex-BFS	**55**	74.8	88	10.1
	MCS	60	69.0	83	**7.6**
	MCS-M	**55**	**68.5**	**82**	12.0
	Lex-M	**55**	**68.5**	**82**	16.0
#509	Lex-BFS	78	93.1	**103**	51.6
	MCS	87	97.0	110	**23.5**
	MCS-M	**74**	**85.9**	105	25.9
	Lex-M	**74**	**85.9**	105	33.6

TABLE 2.2 – Comparaison des heuristiques de triangulation sur les instances GRAPH et SPOT5.

Les résultats sur les instances GRAPH (cf section 4.2) sont donnés par le tableau 2.2. Une nouvelle fois, MCS obtient les meilleurs résultats moyens et les meilleurs temps de calcul sur l'ensemble des instances. MCS-M et Lex-M obtiennent des résultats identiques. Elles produisent toutes les deux les décompositions de largeur minimale sur 5 des 6 instances considérées. Lex-BFS produit des décompositions avec des largeurs de décomposition plus grandes en moyenne, en des temps de calcul supérieurs à ceux de MCS.

Sur les instances `SPOT5` (cf section 4.3), `MCS-M` et `Lex-M` obtiennent les meilleures largeurs de décompositions moyennes et minimales sur 4 des 6 instances (cf. tableau 2.2). Bien que `MCS` donne des décompositions de largeur légèrement supérieure sur ces 4 instances (en moyenne 10% supérieur), elle est toutefois plus rapide (4 fois plus rapide que `MCS-M` sur l'instance #505). Sur les instances #408 et #412, `MCS` obtient les meilleurs résultats. Enfin, `Lex-BFS` produit des décompositions de moins bonne qualité comparé à `Lex-M` et `MCS-M`, mais est plus rapide.

Instance	Méthode	Largeur de décomposition			Temps CPU
		min.	moy.	max	
#10442	Lex-BFS	108	129.8	175	23.7
	MCS	**106**	**124.1**	**140**	**11.5**
	MCS-M	108	125.3	164	106.6
	Lex-M	108	125.3	164	217.0
#13931	Lex-BFS	184	227.7	303	172.7
	MCS	204	**209.7**	**241**	**123**
	MCS-M	**159**	224.8	288	377.9
	Lex-M	**159**	224.8	288	516.5
#14007	Lex-BFS	145	155.4	229	74.7
	MCS	**116**	**121.7**	**133**	**65**
	MCS-M	136	148.0	230	722.8
	Lex-M	136	149.7	216	1531.9
#14226	Lex-BFS	125	148.4	160	14.2
	MCS	**104**	**113.6**	**121**	**11.0**
	MCS-M	125	147.2	151	98.2
	Lex-M	125	147.2	151	282.0
#15757	Lex-BFS	45	62.0	77	1.1
	MCS	46	**49.5**	**51**	**0.7**
	MCS-M	**45**	62.8	77	4.9
	Lex-M	**45**	62.8	77	7.5
#16421	Lex-BFS	82	107.0	111	4.0
	MCS	**76**	**79.6**	**95**	**3.3**
	MCS-M	82	104.9	111	12.1
	Lex-M	82	104.9	111	21.9
#16706	Lex-BFS	**30**	33.2	36	**0.6**
	MCS	**30**	**30.6**	**34**	**0.6**
	MCS-M	**30**	32.0	**34**	6.9
	Lex-M	**30**	32.0	**34**	14.2

Instance	Méthode	Largeur de décomposition			Temps CPU
		min.	moy.	max	
#17034	Lex-BFS	**156**	177.2	247	112.8
	MCS	158	**161.7**	**191**	**42.7**
	MCS-M	158	176.5	209	257.9
	Lex-M	158	176.5	209	465.6
#3792	Lex-BFS	75	82.9	134	6.6
	MCS	**72**	90.3	**97**	**4.0**
	MCS-M	73	**81.8**	119	33.2
	Lex-M	73	**81.8**	119	48.3
#4449	Lex-BFS	**78**	82.4	101	3.7
	MCS	**78**	81.8	**92**	**3.4**
	MCS-M	**78**	**81.4**	100	28.7
	Lex-M	**78**	**81.4**	100	46.1
#6835	Lex-BFS	87	93.2	143	15.2
	MCS	87	101.1	**113**	**11.1**
	MCS-M	**86**	**92.2**	119	72.9
	Lex-M	**86**	**92.2**	119	104.7
#8956	Lex-BFS	112	170.0	194	18.0
	MCS	131	**140.7**	**149**	**9.1**
	MCS-M	**111**	162.4	192	45.4
	Lex-M	**111**	162.4	192	66.2
#9150	Lex-BFS	122	190.1	241	219.8
	MCS	**115**	**129.5**	**132**	**39.8**
	MCS-M	122	190.3	250	806.8
	Lex-M	122	192.4	250	1280.4
#9319	Lex-BFS	78	87.1	101	4.5
	MCS	**72**	**73.8**	**77**	**4.1**
	MCS-M	78	86.5	100	24.8
	Lex-M	78	86.5	100	52.2

TABLE 2.3 – Comparaison des heuristiques de triangulation sur les instances `tagSNP`.

Sur les instances `tagSNP` (cf section 4.4), `MCS` surclasse les trois autres heuristiques sur 11 (resp. 9) des 14 instances en termes de largeur de décomposition moyenne (resp. minimale). Elle est aussi plus rapide. Sur les instances #6835, #4449 et #3792, `MCS` obtient des décompositions ayant respectivement une largeur moyenne 9,6%, 0,4% et 10% plus élevée que celles obtenues par `MCS-M` et `Lex-M`.

2.4.3 Bilan sur les heuristiques de triangulation

Les résultats expérimentaux confirment les bons résultats de `MCS` qui offre clairement le meilleur compromis entre qualité de la décomposition et temps de calcul. En effet, sur la plupart des instances testées, `MCS` obtient des solutions de meilleure qualité en moyenne en des temps de calcul inférieurs aux autres heuristiques. Malgré l'apport en termes de performance de `MCS-M` par rapport à `Lex-M`, la qualité des solutions obtenues par les deux méthodes est strictement identique, mais reste en deçà de celle proposée par `MCS`. `Lex-BFS` présente de meilleurs temps de calcul que les variantes minimales, mais reste toujours en retrait par rapport à `MCS`, que ce soit en temps de calcul mais aussi en largeur de décomposition produite.

2.5 Exploitation de la décomposition arborescente pour les méthodes de recherche complète

L'intérêt de la décomposition arborescente pour la résolution de CSP ou de WCSP vient d'une propriété forte des réseaux de contraintes binaires acycliques.

Théorème 6 (Résolution de CSP binaires acyclique)**.** *Un CSP binaire acyclique peut être résolu en $O(nd^2)$ avec n le nombre de variables et d la taille du plus grand domaine.*

De nombreux travaux ont donc pour but de tirer parti de la décomposition arborescente afin de réduire la complexité théorique de résolution de CSP et de WCSP. On peut distinguer deux types d'approche : les méthodes d'inférence et les méthodes énumératives. Nous décrivons dans cette section différents algorithmes proposés pour l'exploitation de la décomposition arborescente dans ces deux approches.

2.5.1 Une méthode d'inférence : *Cluster Tree Elimination* (CTE)

CTE est un algorithme dérivé de *Variable Elimination* (cf. section 1.4.1). L'algorithme CTE propose d'exploiter la décomposition arborescente afin de supprimer les variables par groupes. Il utilise pour cela les clusters et se base sur l'arbre de clusters pour déterminer l'ordre d'élimination. En partant des feuilles et en remontant jusqu'à la racine, CTE réalise la suppression groupée des variables propres des clusters.

Soit, e le nombre de contraintes, n le nombre de variable, deg le plus grand degré d'un cluster dans la décomposition, d la taille du plus grand domaine, w la largeur de la décomposition et sep la taille du plus grand séparateur. La complexité temporelle de CTE est $O((e+n)deg.k^{w+1})$ et sa complexité temporelle $O(n.k^{sep})$.

L'efficacité de CTE repose donc à la fois sur la largeur de décomposition d'un point de vue temps de calcul, mais aussi sur la taille des séparateurs pour ce qui est de l'espace mémoire nécessaire. En pratique, CTE n'est utilisable que pour des problèmes dont la décomposition possède une très faible largeur de décomposition.

2.5.2 Les méthodes énumératives

Les méthodes énumératives réalisent une exploration arborescente de l'espace de recherche. `DFBB`, que nous avons présentées dans la section 1.3.1, fait partie de cette catégorie. Nous nous intéressons, dans cette section, à deux méthodes énumératives basées sur les décompositions arborescentes.

2.5.2.1 Backtrack Bounded By Tree-Decomposition (BTD)

BTD [Jégou et Terrioux, 2004] exploite la décomposition arborescente à la fois pour guider la recherche et pour améliorer le filtrage. Pour cela, BTD définit un ordre partiel des variables du problème dans lequel toutes les variables d'un cluster fils seront placées après les variables du cluster père. BTD effectue une recherche en profondeur d'abord dans laquelle les variables seront instanciées selon cet ordre. Ainsi, BTD affecte les variables du problème de la racine aux feuilles de la décomposition. Cela lui permet de décomposer le problème en sous problèmes. En effet, pour deux cluster C_i et C_j tel que C_j soit un fils de C_i, le sous-problème induit par C_j, $\mathcal{P}_{C_j} = (C_j, F, D, \top)$ avec $F = \{f_{S_i} | S_i \subseteq C_i\}$, est uniquement influencé par l'affectation courante du séparateur de C_i et C_j. Les auteurs définissent alors la notion de *good* et *nogood* structurels :

Définition 47 (good et nogood structurels)**.** *Un good de C_i est une affectation $\mathcal{A}[sep(C_i, C_j)]$ pouvant être étendue sur C_j.*

Un nogood de C_i est une affectation $\mathcal{A}[sep(C_i, C_j)]$ ne pouvant pas être étendue sur C_j.

BTD va mémoriser au cours de la recherche les *good* et *nogood* structurels afin de réduire l'espace de recherche et ainsi d'accélerer la résolution du problème.

L'algorithme 11 décrit le pseudo-code de BTD. Soit $\mathcal{A}$ une affectation courante, C_i un cluster et V_{C_i} les variables de C_i non affectées dans $\mathcal{A}$. Si toutes les variables de C_i sont affectées, BTD vérifie si le noeud courant est une feuille (ligne 4), auquel cas, il renvoie *vrai* (l'affectation est complète et cohérente). Sinon, il va, pour chaque cluster C_j fils de C_i, vérifier si l'affectation du separateur $\mathcal{A}[sep(C_i, C_j)]$ est un *good* (resp. un *nogood*) et la déclarer cohérente (resp. incohérente) pour C_j (lignes 11 à 14). Si aucun *good* ou *nogood* n'a été enregistré pour cette affectation (ligne 15), BTD est appelé récursivement sur $\mathcal{A}$, C_j et $V_{c_j} = C_j \backslash sep(C_i, C_j)$. S'il déclare $\mathcal{A}$ cohérent (resp. incohérent), $\mathcal{A}[sep(C_i, C_j)]$ est enregistrée en tant que *good* (resp. *nogood*) de C_i/C_j (lignes 16 à 19). Si l'affectation s'avère incohérente pour un des fils, la fonction s'arrête et renvoie $faux$. L'affectation ne peut pas être étendue. Sinon, si l'affectation est cohérente pour tous les fils, la fonction renvoie *vrai*, l'affectation peut être étendue.

La seconde partie de l'algorithme (ligne 21 à 30) est une recherche arborescente. Une variable de V_{C_i} est sélectionnée (ligne 22). Ensuite, pour chaque valeur de D_i ou tant que la cohérence n'est pas établie (ligne 25), on choisit une valeur v de d_i. Si cette valeur ne viole aucune contrainte, alors BTD est appelé récursivement sur $\mathcal{A} \cup \{x \leftarrow v\}$, C_i et $V_{C_i} \backslash \{x\}$ (ligne 29).

Cet algorithme tire parti de la décomposition arborescente à la fois pour guider la recherche mais aussi pour l'accélérer en filtrant plus rapidement les affectations incohérentes. En effet, une fois un séparateur entièrement affecté, si un *good* ou un *nogood* a été enregistré pour cette affectation, il ne sera pas nécessaire de développer plus profondément l'espace de recherche, l'affectation pourra directement être déclarée cohérente ou incohérente.

La complexité temporelle de BDT est en $O(n.sep^2.e.log(d^s).d^{w+1})$ et sa complexité spatiale est en $O(n.sep.d^s)$ [Jégou et Terrioux, 2004], avec e le nombre de contraintes, n le nombre de variables, *deg* le plus grand degré d'un cluster dans la décomposition, d la taille du plus grand domaine, w la largeur de la décomposition et *sep* la taille du plus grand séparateur.

Il est à noter qu'une extension a été proposée dans le cadre des WCSP, BTD_{val}. Celle-ci s'appuie sur l'enregistrement de *nogood* valués. Chaque affectation d'un séparateur est enregistrée comme *nogood* et est associée au coût minimal de toutes les extensions de cette affectation.

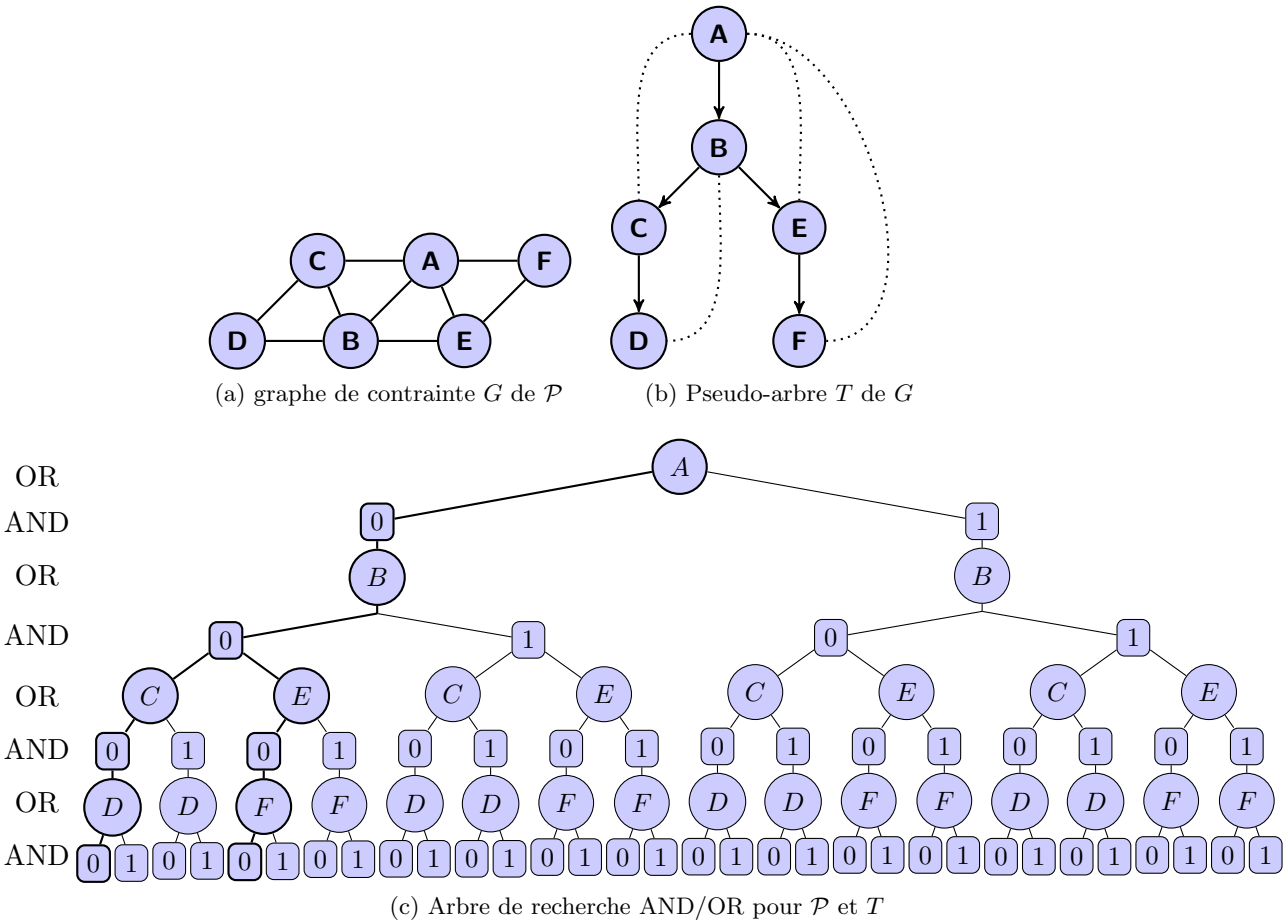
On constate une nouvelle fois l'importance de la largeur de décomposition et de la taille du plus grand séparateur, qui sont les deux facteurs clés de la complexité de BTD d'un point de vue spatial et temporel.

2.5.2.2 AND/OR tree search

La seconde méthode est basée sur une autre approche de l'exploration de l'espace de recherche, le parcours d'arbre AND/OR [Dechter et Mateescu, 2007]. Dans une recherche arborescente classique, on fait correspondre à chaque noeud de la recherche une variable et à chaque arête une affectation. Dans un arbre AND/OR, on alterne successivement des noeuds AND et OR. Les noeud AND correspondent à un choix de variable x_i et les noeuds OR à l'affectation de x_i aux différentes valeurs de son domaine D_i. Une solution est définie de la façon suivante :

Définition 48 (Solution d'un arbre AND/OR)**.** *Une solution dans un arbre AND/OR est un sous arbre T dans lequel :*

– *la racine de l'arbre de recherche appartient à T ;*



(a) graphe de contrainte G de $\mathcal{P}$ (b) Pseudo-arbre T de G

(c) Arbre de recherche AND/OR pour $\mathcal{P}$ et T

FIGURE 2.7 – espace de recherche AND/OR. figure issue de [Dechter et Mateescu, 2007].

- *si un noeud $u \in T$, tous ses fils appartiennent à T ;*
- *si un noeud OR $u \in T$, un seul de ses fils appartient à T ;*
- *toutes ses feuilles correspondent à une affectation cohérente.*

La figure 2.7 présente un exemple d'arbre de recherche AND/OR pour un problème $\mathcal{P}$, avec une solution indiquée en gras dans la figure 2.7c. Le principal avantage de cette approche est qu'à chaque noeud OR, on crée une branche pour chaque composante connexe du graphe de contraintes auquel on a retiré les variables déjà affectées. Dans l'exemple de la figure 2.7, une fois affectées les variables A et B, les composantes connexes CD et EF forment deux sous-problèmes indépendants et peuvent donc être résolues séparément. L'ordre de sélection des variables est donc primordial afin de découper efficacement l'espace de recherche. Pour cela, on utilise la notion de pseudo-arbre.

Définition 49 (Pseudo-arbre)**.** *Soit un graphe $G = (V, E)$ non orienté, on appelle pseudo-arbre de G le graphe acyclique $T = (E', V)$ si, et seulement si, toute arête $\{u, v\} \in E \backslash E'$ relie une sommet u à un de ses ancêtre dans T.*

Un exemple de pseudo-arbre est donné figure 2.7b. L'ordre de sélection des variables est donc induit par le pseudo-arbre associé à l'arbre de recherche. La complexité du parcours dépend de la

largeur d'arbre du pseudo-arbre associé, plus exactement $O(n.d^{w.log(n)})$ pour un pseudo-arbre de largeur minimale.

Plusieurs algorithmes de résolution basés sur les arbres AND/OR ont été proposés pour la résolution de WCSP. Par exemple, AOBB [Marinescu et Dechter, 2009] est basé sur un *branch and bound*. Il parcourt l'arbre AND/OR en profondeur d'abord en maintenant une borne inférieure de la solution courante. À chaque noeud est associé le coût du sous arbre dont il est la racine. Lorsque celui ci n'est pas connu, on calcule une borne inférieure grâce aux méthodes de cohérence (cf section 1.4.3). Un noeud u de l'arbre sera filtré si la borne inférieure dépasse la borne supérieure de la meilleure solution trouvée pour le sous problème formé par les noeuds AND suivant u dans l'arbre de recherche. L'algorithme alterne entre deux phases : l'extension de l'affectation courante et la révision de la borne supérieure des ancêtres pour la solution courante dans l'arbre de recherche. La complexité de cet algorithme est $O(nd^w)$. La complexité de l'algorithme repose donc sur la structure du graphe de contraintes, et plus précisément sur sa largeur de décomposition.

2.5.3 Conclusion sur l'exploitation de la décomposition arborescente pour les méthodes de recherche complète

Nous avons présenté dans cette section quelques exemples de méthodes exploitant la décomposition arborescente pour la résolution de CSP ou de WCSP.

La complexité temporelle de ces méthodes repose sur la largeur de décomposition du problème. En effet, contrairement aux algorithmes qui travaillent sur l'ensemble du problème, les méthodes basées sur la décomposition arborescente se proposent de résoudre des sous-problèmes de petite taille induits par les clusters. Ainsi, la complexité temporelle de ces algorithmes est limitée par la taille du plus grand cluster, soit la largeur de décomposition.

Cependant, du point de vue de la complexité spatiale, ces algorithmes dépendent de la taille des séparateurs de la décomposition. En effet, ils nécessitent pour la plupart de stocker les affectations des séparateurs. Ceci représente une quantité exponentielle de mémoire, et devient un point très sensible pour ces méthodes.

D'un point de vue théorique, les algorithmes exploitant la décomposition arborescente proposent donc une nette amélioration de la complexité temporelle de la résolution de problèmes d'optimisation, mais au prix d'une forte augmentation de la complexité spatiale. En pratique, les décompositions arborescentes de problèmes réels possèdent de très grands séparateurs, rendant difficile l'utilisation des algorithmes présentées dans cette section. Ils ont cependant permis la résolution de certains problèmes réels, et notamment la preuve d'optimalité pour l'instance Scen08 du CELAR, obtenue en 2010 grâce à une méthode hybride basée sur CTE [Allouche *et al.*, 2010].

2.6 Conclusions

Dans ce chapitre, nous avons présenté les notions relatives à la décomposition arborescente et les différentes méthodes permettant de la calculer. Nous avons également comparé plusieurs heuristiques de triangulation sur différents problèmes réels difficiles. De cette étude expérimentale, ils ressort que `MCS` offre le meilleur compromis entre la qualité de décomposition et le temps nécessaire pour son calcul. C'est donc cette heuristique que nous avons retenue pour nos diverses experimentations dans les chapitres 5, 6 et 7. Nous avons aussi présenté différente méthodes exploitant les décompositions arborescentes dans le cadre des recherches complètes. L'efficacité

de ces méthodes repose fortement sur la largeur de decomposition et la taille des séparateurs de la décomposition du problème.

Algorithme 11 : $BTD(\mathcal{A}, C_i, V_{C_i})$

```
1  début
2      si V_{C_i} = ∅ alors
3          si fils(C_i) = ∅ alors
4              retourner vrai ;
5          sinon
6              Coherence ← vrai ;
7              F ← fils(C_i) ;
8              tant que F ≠ ∅ et Coherence faire
9                  Prendre C_j ∈ F ;
10                 F = F\{Cj} ;
11                 si A[sep(C_j, C_i)] est un good de C_i/C_j alors
12                     Coherence ← vraie ; continue ;
13                 si A[sep(C_j, C_i)] est nogood de C_i/C_j alors
14                     Coherence ← faux ; continue ;
15                 Coherence ← BTD(A, C_j, C_j\(sep(C_j, C_i))) ;
16                 si Coherence alors
17                     enregistrer le good A[sep(C_j, C_i)] de C_i/C_j
18                 sinon
19                     enregistrer le nogood A[sep(C_j, C_i)] de C_i/C_j
20         Retourner Coherence
21     sinon
22         prendre x ∈ V_{C_i} ;
23         d_x ← D_x ;
24         Coherence ← Faux ;
25         tant que d_x ≠ ∅ et ¬Coherence faire
26             prendre v ∈ d_x ;
27             d_x ← d_x\{v} ;
28             si A ∪ {x ← v} ne viole aucune contraintes c ∈ C alors
29                 Coherence ← BTD(A ∪ {x ← v}, C_i, V_{C_i}\{x})
30         Retourner Coherence
```

Chapitre 3

Méta-heuristiques

Sommaire

Les méthodes de *recherche arborescente* permettent d'obtenir une solution optimale et de prouver son optimalité. Cependant, pour les problèmes de grande taille, ces méthodes peuvent s'avérer trop gourmandes en temps de calcul et donc peu utiles en pratique, particulièrement si le temps de résolution est contraint.

À l'inverse, les méthodes de *recherche locale* visent à produire des solutions de bonne qualité en des temps de calcul raisonnables. Malheureusement, ces méthodes ne peuvent généralement pas garantir l'optimalité des solutions obtenues et ne sont pas toujours capables de s'échapper facilement des minima locaux.

Les *méta-heuristiques* sont des mécanismes qui contrôlent les méthodes de recherche locale et guident celles-ci afin de sortir des minima locaux. Ces méthodes sont pour la plupart génériques et permettent la résolution approchée de problèmes combinatoires complexes.

Plan du chapitre. Dans ce chapitre, nous commençons par introduire les méthodes de recherche locale (section 3.1). Nous dressons ensuite un panorama non exhaustif des méta-heuristiques proposées dans la littérature (section 3.2). Puis, nous nous intéressons plus particulièrement aux méta-heuristiques proposées pour la résolution de WCSP (section 3.3). Enfin, nous terminons ce chapitre en précisant les critères que nous avons retenus pour l'évaluation des méta-heuristiques (section 3.4).

3.1 Recherches locales

Dans un premier temps, nous commençons par introduire les définitions suivantes.

Définition 50 (Problème à résoudre)**.** *Soit $\mathcal{S}$ l'ensemble des solutions à un problème d'optimisation. Soit f une fonction qui mesure le coût $f(S)$ de toute solution $S \in \mathcal{S}$. On souhaite trouver une solution $S \in \mathcal{S}$ de coût $f(S)$ minimal. Une telle solution, notée S^*, est appellée* **solution optimale** *ou* **optimum global***. Le problème à résoudre dans la suite de ce mémoire est donc le suivant :*

$$Trouver\ S^*\ tel\ que\ f(S^*) = \min_{S \in \mathcal{S}} f(S)$$

Une méthode de recherche locale est fondée sur deux éléments essentiels : la définition d'un *voisinage* (permettant d'obtenir les solutions proches d'une solution donnée) et la donnée d'une *stratégie d'exploration* du voisinage.

Définition 51 (Voisinage d'une solution)**.** *Soit $\mathcal{S}$ un ensemble de solutions. On appelle* **voisinage** *toute application $N : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ associant à toute solution $S \in \mathcal{S}$ un ensemble de solutions noté $N(S)$ dites voisines.*

Définition 52 (Mouvement)**.** *On appelle* **mouvement** *l'opération qui consiste à passer d'une solution $S \in \mathcal{S}$ à une solution voisine $S' \in N(S)$.*

Définition 53 (Minimum local pour un voisinage N)**.** *Une solution S est dite minimum localement au voisinage N si, et seulement si, pour toute solution S' dans $N(S)$: $f(S) \leq f(S')$.*

Les méthodes de recherche locale, contrairement aux méthodes arborescentes, effectuent des mouvements dans l'espace de recherche de solution en solution. À chaque mouvement, la méthode de recherche locale tente d'améliorer la solution courante S en sélectionnant heuristiquement une solution S' dans son *voisinage* $N(S)$. Le processus s'arrête lorsque la solution courante n'a pas pu être améliorée pendant un certain nombre de mouvements (fixé au départ), ou que le temps maximal de recherche autorisé (*TimeOut*) est atteint. Bien que ces méthodes ne soient pas complètes, et ne puissent par conséquent pas effectuer de preuve d'optimalité, de telles méthodes produisent très souvent des solutions de bonne qualité dans des temps de calcul raisonnables.

L'algorithme 12 détaille le pseudo-code d'une recherche locale, avec $\mathcal{P}$, le problème à résoudre, $f(S)$ une fonction d'évaluation d'une solution S, `selectVoisin`, l'heuristique de sélection dans le voisinage N et $iter_{max}$, le nombre maximal de mouvements autorisés sans amélioration de la qualité de la solution. L'algorithme part d'une solution initiale S souvent générée aléatoirement (ligne 3). À chaque itération (ou mouvement), une solution S' est sélectionnée heuristiquement dans le voisinage de S (ligne 6). Si cette solution améliore S (i.e. est de meilleure qualité que S), elle devient la solution courante (ligne 8). Dans le cas où l'algorithme n'a pas réussi à s'échapper du minimum local, le nombre de mouvements est incrémenté (ligne 11). Lorsque la solution courante n'a pas pu être améliorée pendant un certain nombre de mouvements (fixé au départ), ou que le *TimeOut* est atteint (ligne 5), l'algorithme retourne la meilleure solution trouvée (ligne 12).

Algorithme 12 : Pseudo-code d'une recherche locale.

```
1  Fonction rechercheLocale(P, iter_max, N) ;
2  début
3      S ← genSolInit(P) ;
4      iter ← 1 ;
5      while (iter < iter_max) ∧ (not TimeOut) do
6          S' ← selectVoisin(S) ;
7          si f(S') < f(S) alors
8              S ← S' ;
9              iter ← 1 ;
10         sinon
11             iter ← iter + 1 ;
12     retourner S ;
```

Algorithme 13 : Première amélioration.

```
1  Fonction selectPremierVoisin(S) ;
2  début
3      répéter
4          S' ← S, i ← 0 ;
5          répéter
6              i ← i + 1 ;
7              S ← arg min{f(S), f(S_i)}, S_i ∈ N(S) ;
8          jusqu'à (f(S) < f(S_i) ou i =| N(S) |) ;
9      jusqu'à f(S) ≥ f(S') ;
10     retourner S' ;
```

Algorithme 14 : Meilleure amélioration.

```
1  Fonction selectMeilleurVoisin(S) ;
2  début
3      répéter
4          S' ← S ;
5          S ← arg min_{y∈N(S)} f(y) ;
6      jusqu'à (f(S) ≥ f(S')) ;
7      retourner S' ;
```

La taille des voisinages influe directement sur le comportement d'une méthode de recherche locale. En effet, plus le voisinage est grand, plus il y a de chances d'améliorer la solution courante. En contrepartie, l'exploration de ce voisinage devient vite problématique car le temps nécessaire à l'exploration augmente rapidement. On distingue en général deux stratégies pour l'exploration des voisinages :

- La première stratégie, dite **première amélioration**, consiste à énumérer les voisins de S jusqu'à en rencontrer un qui améliore la solution courante. L'algorithme 13 décrit le pseudo-code de cette première stratégie.
- La seconde stratégie, dite **meilleure amélioration**, consiste à explorer le voisinage $N(S)$ de manière complète afin de rechercher le meilleur voisin (meilleure amélioration). Cette dernière solution peut sembler plus coûteuse mais a le mérite de retourner le voisin ayant le meilleur coût. L'algorithme 14 décrit le pseudo-code de cette seconde stratégie.

Remarque 2. *L'inconvénient majeur des deux stratégies précédentes est qu'elles restent bloquées au premier minimum local rencontré.*

Afin de s'échapper des minima locaux, la plupart des méthodes de recherche locale utilisent des stratégies basées sur un mécanisme dit de *diversification.*

Définition 54 (Diversification)**.** *Le but de la diversification est de parcourir un grand nombre de régions différentes de l'espace de recherche, afin d'assurer que l'espace de recherche est correctement exploré, afin de localiser la région contenant l'optimum global.*

D'autres méthodes utilisent également des stratégies permettant *d'intensifier* l'exploration dans le voisinage d'une solution.

Définition 55 (Intensification)**.** *Le but de l'intensification est de se concentrer au sein d'une région de l'espace de recherche pour converger vers un minimum local.*

La prochaine section dresse un panorama non exhaustif des méta-heuristiques proposées dans la littérature.

3.2 Méta-heuristiques

Les méta-heuristiques sont des mécanismes qui contrôlent les méthodes de recherche locale et guident celles-ci afin d'éviter qu'elles restent bloquées dans un minimum local. La figure 3.1 présente une liste non exhaustive des méta-heuristiques existantes. Celles-ci sont réparties en 3 classes suivant les mécanismes utilisés pour s'échapper ou éviter les minima locaux : les bio-inspirées, celles basées sur la notion de voisinage et celles basées sur la notion de pénalité. Le lecteur intéressé par une synthèse des méta-heuristiques pourra se référer à [Hao *et al.*, 1999], pour de plus amples détails.

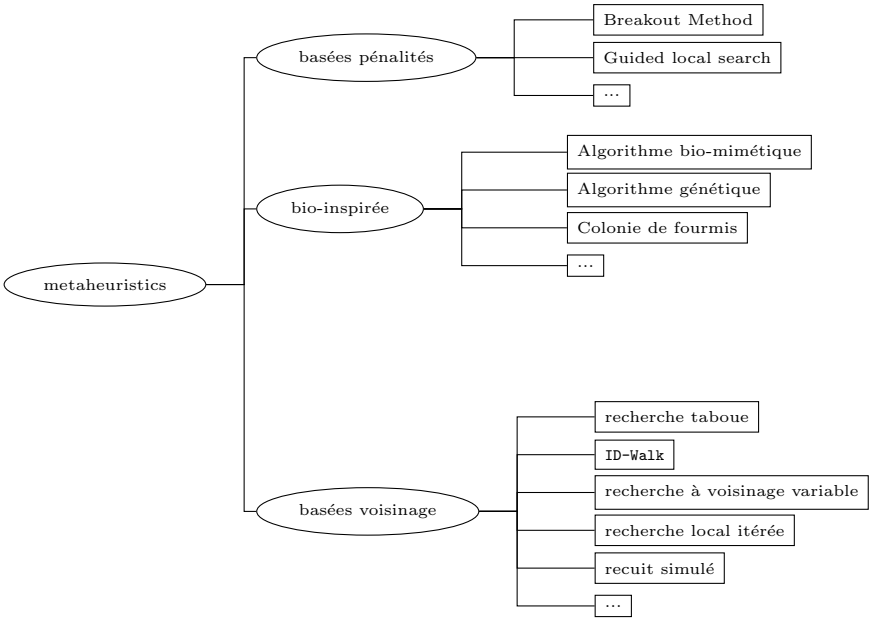


FIGURE 3.1 – Panorama non exhaustif des méta-heuristiques existantes.

3.2.1 Algorithmes basés sur les pénalités

Le principe général des méta-heuristiques basées sur les pénalités est de faire augmenter artificiellement le coût de la solution courante si celle-ci est un minimum local. À chaque itération où la recherche n'a pas permis de trouver une meilleure solution, ces pénalités sont augmentées. Ainsi, la recherche pourra s'orienter vers des solutions voisines rendues plus intéressantes et ainsi s'échapper d'un optimum local. Parmi ces méthodes, nous pouvons citer Guided Local Search (`GLS`) [Voudouris et Tsang, 1999] et Breakout Local Search (`BLS`) [Benlic et Hao, 2013].

3.2.2 Méta-heuristiques bio-inspirées

Ces méthodes cherchent à reproduire un mécanisme naturel pour l'appliquer à la recherche de solutions pour un problème d'optimisation combinatoire.

3.2.2.1 Colonies de fourmis

Cette méta-heuristique reproduit les mécanismes utilisés par les colonies de fourmis afin de trouver de la nourriture. Quand une fourmi trouve une source de nourriture, elle dépose sur son passage des phéromones qui attirent ses congénères sur ce chemin. Plus les fourmis emprunteront ce chemin, plus la quantité de phéromones augmentera, rendant le chemin de plus en plus attirant. Avec le temps, les phéromones s'évaporent, permettant de réduire l'attractivité d'un chemin.

Le premier algorithme à base de fourmis a été proposé par Dorigo [Dorigo, 1992]. La première étape de la résolution consiste à représenter le problème à résoudre par la recherche d'un chemin de coût minimal dans un graphe. Puis, une "colonie" de fourmis se déplaçant sur ce graphe, y construit itérativement des solutions. À chaque "fourmi" est associé un algorithme glouton qui construit à chaque itération une solution. Chaque fois qu'une fourmi génère une solution, elle "dépose" sur chacune des affectations de la solution, une quantité de phéromones proportionnelle à la qualité de la solution trouvée. L'*évaporation* progressive des phéromones au cours de la recherche permet d'alterner les phases d'intensification et de diversification. Le lecteur intéressé par les colonies de fourmis pourra se référer au livre [Solnon, 2008], pour de plus amples détails.

3.2.2.2 Algorithmes génétiques

Les algorithmes génétiques [Holland, 1992], miment les principes de l'évolution des espèces naturelles lors de la recherche de solutions. Leur principe est fondé sur la *sélection* des meilleurs individus au sein d'une population, sur les *mutations* intervenant sur les gènes des individus et sur les croisements entre les gènes de plusieurs individus, intervenant lors de phases de reproduction. Un individu représente une solution au problème et un gène une affectation d'une variable. Au début de la recherche, une *population* d'individus est générée (souvent de manière aléatoire). Puis des opérations de sélection, de mutation et de croisement sont appliquées itérativement sur la population d'individus. Un croisement permet de générer une nouvelle solution en combinant les affectations de deux solutions. Une mutation consiste à modifier légèrement une solution. La phase de sélection consiste à choisir un certain nombre d'individus de la population.

3.2.3 Méta-heuristiques basées sur la notion de voisinage

3.2.3.1 Recuit simulé

La méthode du recuit simulé (*Simulated Annealing*), [Kirkpatrick *et al.*, 1983], est basée sur le principe de la solidification des matériaux en physique. Quand un matériau est porté à une

température élevée, ses molécules se déplacent librement les unes par rapport aux autres. Au fur et à mesure que le matériau se refroidit, leur liberté de déplacement se réduit. Si la température baisse suffisamment lentement, les molécules adoptent alors naturellement une structure cristalline ayant un niveau d'énergie minimal. Dans le cas contraire, les molécules n'adoptent pas cette structure et le niveau d'énergie n'atteint pas ce minimum. Le niveau d'énergie de la structure des molécules représente la qualité de la solution courante, et la température reflète la limite de dégradation de la qualité de la solution courante.

La température décroît *lentement* au fur et à mesure des mouvements de l'algorithme, et lorsque celui-ci est bloqué dans un minimum local S, il est autorisé à dégrader S en S' si l'équation 3.2 est vérifiée, avec T la valeur de la température, C_b une constante physique connue sous le nom de *constante de Boltzmann* et r un nombre tiré aléatoirement entre 0 et 1.

$$\delta = f(S') - f(S) \tag{3.1}$$
$$r < \exp^{-\delta/(C_b \times T)} \tag{3.2}$$

3.2.3.2 Recherche tabou

La recherche tabou (TS), est une méthode de recherche locale introduit par [Glover, 1986]. À chaque itération, TS sélectionne la meilleure solution dans le voisinage, même si la qualité de celle-ci dégrade la solution courante. Ce mécanisme de sélection pouvant boucler sur un sous-ensemble de solutions, TS introduit la notion de liste *taboue*. Une liste taboue est une mémoire à court terme, contenant les k dernières solutions courantes, ou de manière plus générale, les k derniers attributs pertinents des solutions. Chaque solution (ou attribut) taboue S est alors *interdite* pendant les k prochaines itérations, c'est à dire que S ne peut devenir pendant k itérations la solution courante. Ce mécanisme oblige l'algorithme à sortir des minima locaux rencontrés (puisqu'ils sont tabou), mais son efficacité dépend fortement de la taille de la liste taboue. Une amélioration apportée à TS, fût l'introduction de la notion d'*aspiration* : un mouvement tabou peut être autorisé s'il conduit à une solution de meilleure qualité que la meilleure solution connue. Plusieurs autres méthodes ont été introduites et appliquées avec succès sur différents problèmes, dont les problèmes d'affectation de fréquence radio dans les réseaux mobiles ([Hao *et al.*, 1998]). Le lecteur intéressé pourra se référer au livre [Glover et Laguna, 1997], pour de plus amples détails.

3.2.3.3 Variable Neighbourhood Search

Contrairement aux méthodes de recherche locale présentées jusqu'ici, la recherche à voisinage variable (`VNS`), [Mladenovic et Hansen, 1997], est une recherche à grand voisinage qui utilise méthodiquement plusieurs types de voisinages, dans le but de s'échapper des minima locaux. Elle est basée sur les trois observations ci-dessous :

- un optimum local relativement à un voisinage donné n'est pas necessairement optimal par rapport à un autre voisinage,
- un optimum global est toujours optimal relativement à toute structure de voisinage,
- pour de nombreux problèmes, les minima locaux relativements à differents voisinages sont proches les uns des autres.

Soit $L = \{N_1, \ldots, N_{k_{max}}\}$ une liste finie de *structures de voisinage*, ordonnée par ordre croissant de taille, où $N_k(S)$ ($k \in [1 \ldots k_{max}]$) désigne l'ensemble des solutions voisines de S dans N_k. Dans la plupart des méthodes de recherche locale, nous avons $k_{max} = 1$.

Algorithme 15 : Changement de voisinage.

```
1 Procédure NeighbourhoodChange (S, S', k) ;
2 début
3     si f(S') < f(S) alors
4         S ← S' ; k ← 1 /*Make a move*/ ;
5     sinon k ← k + 1 /*Next neighbourhood*/ ;
```

L'idée centrale de la recherche VNS se trouve dans le changement systématique de la structure de voisinage, selon la procédure NeighborhoodChange. L'algorithme 15 détaille son schéma général. Soit une solution courante S et S' la nouvelle solution obtenue dans N_k ; Si S' est de meilleure qualité que S (ligne 3), alors S' devient la nouvelle solution courante et on retourne à N_1 (ligne 4). Sinon, on passe à N_{k+1} (ligne 5).

Le fait d'utiliser plusieurs structures de voisinage permet de *diversifier* l'exploration de l'espace de solutions afin d'accéder à un plus grand nombre de régions intéressantes, ce qui conduit à une méthode plus robuste que le recuit simulé ou la recherche tabou.

Algorithme 16 : Pseudo-code de la recherche VNS.

```
1  Fonction VNS (S, k_max, t_max) ;
2  début
3      répéter
4          k ← 1 ;
5          répéter
6              S' ← Shake (S,N_k) /* Shaking */ ;
7              S'' ← selectPremierVoisin (S') /* Local search */ ;
8              NeighbourhoodChange (S,S'',k) /*Change neighbourhood*/ ;
9          jusqu'à k = k_max ;
10         t ← CpuTime () ;
11     jusqu'à t > t_max ;
12     Retourner S
```

L'algorithme 16 détaille le pseudo-code de VNS, avec N_k $(k = 1, \ldots, k_{max})$ la k-ième structure de voisinage. Soit t_{max} le temps maximal alloué à la recherche ($TimeOut$). L'algorithme part d'une solution initiale S souvent générée aléatoirement. À chaque itération de la boucle des lignes (5-8), une solution S' est sélectionnée aléatoirement (phase de *shaking*) dans la k-ième structure de voisinage de S ($S' \in N_k(S)$) (ligne 6). Une méthode de recherche locale est ensuite appliquée en utilisant S' comme solution initiale et une nouvelle solution S'' est retournée (ligne 7). Ensuite, la procédure de changement de voisinage est appelée avec pour argument S, S'' et k (ligne 8). L'algorithme s'arrête dès que l'on a atteint le $TimeOut$ (ligne 3).

3.2.3.4 Variable Neighborhood Decomposition Search

Bien que VNS soit très efficace, elle est toutefois moins performante sur des problèmes de grande taille. En effet, la recherche locale sur un grand espace de recherche est très coûteuse et ralentit considérablement l'amélioration de la solution courante. Variable Neighborhood Decomposition Search (VNDS) [Hansen *et al.*, 2001] est une extension de VNS qui vise à réduire l'espace de solutions parcouru par la recherche locale. Pour cela, à chaque itération, la recherche est effectuée uniquement sur un sous-problème déterminé heuristiquement. Le principe est de fixer une sous partie du problème et de rechercher de nouvelles solutions en ne modifiant que les

Algorithme 17 : Pseudo-code de la recherche VNDS.

```
1  Fonction VNDS (S, k_max, t_max, t_d) ;
2  début
3      répéter
4          k ← 2 ;
5          répéter
6              S' ← Shake (S,N_k) ;
7              y ← S' \ S ;
8              y' ← VNS(y, k, t_d) ;
9              S'' ← (S' \ y) ∪ y' ;
10             S''' ← selectPremierVoisin (S'') ;
11             NeighbourhoodChange (S,S''',k) ;
12         jusqu'à k = k_max ;
13         t ← CpuTime () ;
14     jusqu'à t > t_max ;
15     Retourner S
```

parties « libres » du problème. Cette recherche est effectuée par la recherche `VNS` appliquée au sous-problème déterminé précédement.

L'algorithme 17 décrit le pseudo-code de `VNDS`, où t_d est un paramètre supplémentaire qui représente le temps maximal alloué pour résoudre chaque sous-problème. Soit y l'ensemble des k attributs présents dans S' mais pas dans S ($y = S' \setminus S$) (ligne 7). À chaque itération de la boucle des lignes (5-11), contrairement à `VNS`, `VNDS` sélectionne un ensemble de k attributs (k étant appelé la *dimension du voisinage*) (ligne 7), puis applique une recherche locale (`VNS`) uniquement sur le sous-problème y composé de ces k attributs (ligne 8). Soit y' la solution partielle retournée par la recherche `VNS`. La nouvelle solution S'' du problème initial est obtenue en recomposant la solution partielle y' avec la partie ayant été fixée dans S' (ligne 9). Afin de régler les éventuelles incohérences lors de la phase de recomposition de la solution globale, une recherche locale est effectuée sur l'ensemble du problème (ligne 10).

3.3 Méta-heuristiques pour la résolution des WCSP

Nous détaillons dans cette section deux approches proposées pour la résolution des WCSP. La première, `ID-Walk`, propose un algorithme générique d'intensification et diversification et a obtenu de très bonnes performances sur des instances WCSP. La seconde, `VNS/LDS+CP`, tire parti des mécanismes de filtrage liées au WCSP afin d'accélérer le parcours de l'espace de recherche.

3.3.1 `ID-Walk`

`ID-Walk` (*Intensification Diversification Walk*) est une méta-heuristique introduite dans [Neveu *et al.*, 2004], où le compromis entre intensification et diversification est réalisé grâce à l'utilisation d'une liste de mouvements candidats (*Candidate List Strategy*) [Glover et Laguna, 1997] pour contrôler le parcours d'un voisinage. Deux paramètres sont utilisés $MaxNeighbors$ et $SpareNeighbor$:

- $MaxNeighbors$ indique le nombre maximum de voisins à examiner avant d'effectuer un mouvement. Il définit l'effort d'intensification d'`ID-Walk`, et indique quand diversifier.

Algorithme 18 : Pseudo-code d'`ID-Walk`.

```
1  Fonction ID-Walk (S, MaxNeighbors, SpareNeighbor, N) ;
2  début
3      S ← genSolInit(P) ;
4      best ← S ;
5      pour iter ← 1 à Max faire
6          nbcandidat ← 1 ;
7          candidat_rejetes ← ∅ ;
8          accepte ← false ;
9          tant que (nbcandidat ≤ MaxNeighbors et not accepte) faire
10             S' ← selectVoisin(S) ;
11             si f(S') < f(S) alors
12                 accepte ← true ;
13             sinon
14                 candidat_rejetes ← candidat_rejetes ∪ {S'} ;
15             si accepte alors
16                 S ← S' ;
17             sinon
18                 S ← Select(candidat_rejetes, SpareNeighbor) ;
19             si f(S) ≤ f(best) alors
20                 best ← S ;
21     retourner best
```

– $SpareNeighbor$ précise quelle prochaine solution choisir si aucun voisin parmi les $MaxNeighbors$ visités n'a été accepté. Ce paramètre définit en quelque sorte la marge de détérioration de la solution courante pour se sortir d'un optimum local. Deux valeurs sont possibles :
  – $SpareNeighbor = 1$: le premier voisin exploré est choisi ;
  – $SpareNeighbor = MaxNeighbors$: le meilleur (moins mauvais) parmi les voisins explorés est choisi (faible détérioration).

Ainsi, à chaque mouvement, `ID-Walk` va considérer $MaxNeighbors$ solutions candidates, voisines de la solution courante, afin de trouver une nouvelle solution. Si une solution voisine est trouvée, la recherche est intensifiée autour d'elle. Sinon, la recherche est diversifiée à partir d'une des solutions candidates de moins bonne qualité.

L'algorithme 18 présente le pseudo-code d'`ID-Walk`. Tout d'abord, une solution initiale est générée aléatoirement (ligne 3). Ensuite, tant que le nombre de mouvements (fixé au départ) n'est pas atteint, on initialise la liste de candidats rejetés à $\emptyset$ et le nombre de candidats visités à 1. À chaque mouvement, `ID-Walk` considère au plus $MaxNeighbors$ voisins sélectionnés dans le voisinage de la solution courante S (ligne 10). Si un de ces voisins améliore la qualité de la solution S, le mouvement est interrompu et S' devient la solution courante (lignes 12 et 16). Sinon, le voisin est ajouté à la liste *candidat_rejetes* (ligne 14). Si le nombre de voisins candidats autorisés est atteint sans qu'aucune solution n'ait été acceptée, `ID-Walk` tente de s'échapper du minimum local en sélectionnant un des voisins rencontrés dans la liste *candidat_rejetes* (ligne 18).

Dans [Neveu *et al.*, 2004], les auteurs proposent également une méthode permettant de déterminer automatiquement la valeur de $MaxNeighbors$ lors d'une phase de tuning : initiale-

Algorithme 19 : Pseudo-code de l'algorithme VNS/LDS+CP.

```
1  Fonction VNS/LDS+CP(X, W, k_init, k_max, δ_max) ;
2  début
3      S ← genSolInit(X, W) ;
4      k ← k_init ;
5      tant que (k < k_max) ∧ (not TimeOut) faire
6          X_un ← Hneighborhood(N_k, X, W, S) ;
7          A ← S\{(x_i, a) | x_i ∈ X_un} ;
8          S' ← LDS + CP(A, X_un, δ_max, f(S), S) ;
9          si f(S') < f(S) alors
10             S ← S ;
11             k ← k_init ;
12         sinon k ← k + 1 ;
13     retourner S ;
```

ment, ID-Walk est exécuté pour différentes valeurs de $MaxNeighbors$, avec un nombre maximal d'itérations fixé à (Max/K) (K étant une constante). L'algorithme est ensuite exécuté avec $MaxNeighbors$ égal à la meilleure valeur trouvée précédemment. Si l'algorithme atteint le nombre maximal d'itérations fixé Max avant le timeout, Max est alors agrandi et un restart est effectué. Cette méta-heuristique fournit de bons résultats sur de nombreux problèmes réels, dont les instances RLFAP du CELAR (cf. section 4.1).

3.3.2 VNS/LDS+CP

La méthode VNS/LDS+CP, [Loudni et Boizumault, 2008], est une extension de VNDS. La phase de shaking est effectuée en désaffectant k variables du problème, tandis que la phase de reconstruction est réalisée grâce à une méthode arborescente partielle (LDS), combinée avec des mécanismes de propagation (CP) basés sur un calcul de minorants. L'algorithme 19 décrit son pseudo-code, avec W l'ensemble des fonctions de coût, k_{init} (resp. k_{max}) le nombre minimal (resp. maximal) de variables à désaffecter et δ_{max} la valeur maximale de la discrepancy pour LDS.

L'algorithme part d'une solution initiale S générée aléatoirement (ligne 3). Un sous-ensemble de k variables (avec k la dimension du voisinage) est sélectionné dans le voisinage N_k (i.e. l'ensemble des combinaisons de k variables parmi X) par l'heuristique de choix de voisinage Hneighborhood (ligne 6). Une affectation partielle $\mathcal{A}$ est alors générée à partir de la solution courante S, en désaffectant les k variables sélectionnées ; les autres variables (i.e. non sélectionnées) gardent leur affectation dans S (ligne 7). $\mathcal{A}$ est alors reconstruite en utilisant une recherche arborescente partielle de type LDS, aidée par une propagation de contraintes (CP) basée sur un calcul de minorants. Si LDS trouve une solution de meilleure qualité S' dans le voisinage de S (ligne 9), S' devient la solution courante et k est réinitialisé à k_{init} (lignes 10-11). Sinon, on cherche de nouvelles améliorations dans N_{k+1} (structure de voisinage où $(k+1)$ variables de X seront désaffectées) (ligne 12). L'algorithme s'arrête dès que l'on a atteint la dimension maximale des voisinages à considérer k_{max} ou le $TimeOut$ (ligne 5).

Heuristique de choix de voisinage. L'heuristique de choix de voisinage, utilisée pour sélectionner les variables à désaffecter, guide VNS/LDS+CP pour déterminer les sous-espaces à explorer afin de trouver des solutions de meilleure qualité. Les heuristiques de choix de voisinage dépendent souvent du problème traité et nécessitent, par conséquent, un temps d'expertise non

Algorithme 20 : Pseudo-code de l'heuristique `Hneighborhood`.

```
1  Fonction Hneighborhood(N_k, X, W, A) ;
2  début
3      X_un ← ∅ ;
4      X_conf ← getConflict(A, W) ;
5      tant que |X_un| < k faire
6          si X_conf ≠ ∅ alors
7              x ← randomPick(X_conf) ;
8              X_conf ← X_conf \ {x}
9          sinon x ← randomPick(X \ X_un) ;
10         X_un ← X_un ∪ {x} ;
11     retourner X_un
12 Fonction getConflict (A, W) ;
13 début
14     X_res ← ∅ ;
15     pour tout w_{S_i} ∈ W faire
16         si w(A[S_i])) > 0 alors
17             X_res ← X_res ∪ S_i
18     retourner X_res
```

négligeable. À notre connaissance, peu d'heuristiques indépendantes du problème existent. Parmi celles-ci nous pouvons citer l'heuristique ConflictVar basée sur la notion de conflit. Celle-ci est une adaptation, aux heuristiques de choix de voisinage, de l'heuristique de recherche MinConflict [Minton *et al.*, 1990].

ConflictVar est une heuristique, définie dans le cadre de la satisfaction et basée sur la notion de *conflit* définie ci-dessous.

Définition 56 (Variable en conflit)**.** *Pour une affectation complète $\mathcal{A}$, une variable est dite en conflit si elle figure dans au moins une fonction ayant un coût non nul. L'ensemble des variables en conflit pour l'affectation $\mathcal{A}$ est défini par l'équation 3.3.*

$$X_{conf} = \{x_i \in X \mid \exists w_{S_i} \in W \ t.q. \ x \in S_i \wedge w_{S_i}(\mathcal{A}[S_i]) > 0\} \tag{3.3}$$

L'heuristique `Hneighborhood` est basée sur le principe de ConflictVar. L'algorithme 20 décrit son pseudo-code. Soit `getConflict` la fonction qui retourne, pour une affectation complète, l'ensemble des variables en conflit. Pour une dimension de voisinage k, l'heuristique de choix de voisinage `Hneighborhood` sélectionne aléatoirement k variables à désaffecter parmi celles en conflit (lignes 7 et 8). S'il n'existe pas suffisamment de variables en conflit (ligne 6), l'heuristique sélectionne ensuite aléatoirement des variables parmi celles qui ne sont pas en conflit (ligne 9).

3.4 Critères d'évaluation des performances des méta-heuristiques

Dans cette section, nous détaillons les différents critères que nous avons retenus pour évaluer et comparer les performances des méthodes étudiées.

3.4.1 Taux de réussite

Afin de prendre en compte le caractère stochastique des méthodes testées, chaque instance sera résolue 50 fois par chaque méthode. Ainsi, pour chaque instance et chaque méthode, nous

reportons :
- le nombre d'essais avec succès (c'est-à-dire ayant atteint l'optimum) ;
- le temps de calcul moyen pour atteindre l'optimum ;
- le coût moyen des solutions trouvées sur les 50 essais ;

Le taux de réussite d'une méthode par rapport à une instance est défini par le rapport entre le nombre d'essais avec succès et le nombre total d'essais. Le taux de réussite permet de mesurer la capacité de la méthode à attendre le plus souvent l'optimum quand celui-ci est connu ou la meilleure solution connue. Elle traduit en quelque sorte l'efficacité de la méthode sur l'instance considérée.

3.4.2 Profil de performance

Une méthode de résolution doit être en mesure de produire des solutions intermédiaires dont la qualité s'améliore graduellement au cours du temps. Cette information sur l'évolution de la qualité en fonction du temps de calcul alloué est appelée *Profil de performance* [Boddy et Dean, 1994].

Définition 57 (Profil de performance)**.** *Un profil de performance d'une méthode de résolution est la courbe représentant l'évolution de la qualité de la meilleure solution produite en fonction du temps de calcul t.*

Le profil de performance d'un algorithme fournit une information sur sa capacité à offrir un réel compromis qualité/temps plus précise que la simple information "meilleure solution trouvée au bout d'un certain temps de calcul".

Pour nos comparaisons, nous considérons l'évolution en moyenne de la qualité des résultats intermédiaires en fonction du temps de calcul alloué. Ce type de profil est appelé *profil de performance moyen.*

Définition 58 (Profil de performance moyen)**.** *Le profil de performance moyen d'un algorithme est la courbe représentant l'évolution de la qualité de la meilleure solution produite, en moyenne, en fonction du temps de calcul t.*

3.4.3 Profils de rapport de performance

Dans la seconde partie de ce mémoire, nous serons amenés à comparer les performances des méthodes que nous proposons, que ce soit entre elles, ou que ce soit avec des méthodes existantes.

Pour cela on considère les travaux de [Dolan et Moré, 2002, Barbosa *et al.*, 2011] concernant les profils de rapport de performance. On souhaite comparer les performances d'un ensemble d'algorithmes $A = \{a_1, \ldots, a_m\}$ pour résoudre un ensemble de problèmes $P = \{p_1, \ldots, p_n\}$. Deux mesures de performance sont retenues : le temps de calcul moyen et le nombre de d'essais avec succès (noté succès) (cf. remarque 3).

La grandeur $t(p_i, a_j)$ désigne le temps de calcul moyen nécessaire à l'algorithme a_j pour atteindre l'optimum du problème p_i. Dans le calcul de la moyenne, le temps reporté pour les essais sans succès (optimum non atteint) est la valeur du $TimeOut$.

Le rapport de performance $r(p_i, a_j)$ permet de comparer la performance de l'algorithme a_j sur le problème p_i avec la meilleure performance obtenue par les autres algorithmes sur ce même problème. Ce rapport est défini par [Dolan et Moré, 2002] :

$$r(p_i, a_j) = \frac{t(p_i, a_j)}{\min\{t(p_i, a_k) \mid a_k \in A\}} \tag{3.4}$$

Quels que soient l'algorithme a_i et le problème p_j, on a toujours $r(p_i, a_j) \geq 1$. Si $r(p_i, a_j)$=1 alors a_j est considéré comme étant le meilleur algorithme pour résoudre le problème p_i. L'algorithme a_j peut être seul ou bien ex aequo avec un ou plusieurs autres algorithmes. Enfin, soit τ_{max} le maximum de tous les $r(p_i, a_j)$.

Il devient alors possible d'évaluer les performances d'un algorithme a_j sur l'ensemble de problèmes P grâce à la fonction $\rho_{a_j} : [1..\tau_{max}] \mapsto [0..1]$ définie par :

$$\rho_{a_j}(\tau) = \frac{1}{|P|}|\{p_i \in P : r(p_i, a_j) \leq \tau\}| \tag{3.5}$$

Pour un algorithme a_j fixé, $\rho_{a_j}(\tau)$ indique la proportion de problèmes $p_i \in P$ ayant un rapport de performance inférieur à τ. Pour deux algorithmes a_i et a_j, si $\rho_{a_i}(\tau) > \rho_{a_j}(\tau)$, alors, pour un facteur τ, la proportion de problèmes résolus par a_i est plus grande que celle de a_j .

Dans la suite de ce document, nous désignerons par *profil de de performance des rapports de temps d'un algorithme* a_j, la courbe associée à la fonction ρ_{a_j}. Nous noterons τ_j^* la valeur de τ tel que $\rho_{a_j}(\tau_j^*) = 1$ et τ_{min} la valeur minimale de τ_j^* parmi tous les algorithmes de A.

Plusieurs critères permettant de comparer les différents algorithmes peuvent être déduits de l'équation 3.5 :

1. $\rho_{a_j}(1)$ indique la proportion de problèmes de P où l'algorithme a été le plus performant : plus la valeur de $\rho_{a_j}(1)$ est élevée, meilleur est a_j.
2. L'aire sous la courbe ρ_{a_j} (notée $AUC_{a_j} = \int \rho_{a_j}(\tau)\, \mathrm{d}t$) donne un indicateur de la performance globale de a_j sur les problèmes de P : plus l'aire est grande, plus l'algorithme a_j est efficace.
3. Le ratio τ_{min}/τ_j^* permet de mesurer la robustesse d'un algorithme : l'algorithme le plus robuste est celui ayant un ratio de 1

Remarque 3. *Dans l'équation 3.4, permettant d'évaluer le rapport de performance d'un algorithme, outre la mesure du temps de calcul moyen passé par l'algorithme a_j pour atteindre l'optimum sur le problème p_i, nous considérons également le nombre d'essais avec succès obtenu par l'algorithme a_j sur le problème p_i. Toutefois, pour générer la courbe $\rho_{a_j}(\tau)$ mesurant l'évolution du rapport de performance du nombre d'essais avec succès de l'algorithme a_j en fonction de τ, il est nécessaire de considérer l'inverse du nombre d'essais avec succès dans l'équation 3.4.*

3.5 Conclusion sur les méta-heuristiques

Dans ce chapitre, nous avons dressé un panorama non exhaustif des méta-heuristiques proposées dans la littérature. Dans la suite de ce document, nous nous intéresserons plus particulièrement à `VNS/LDS+CP`. Cette méthode, basée sur de grands voisinages, permet en effet de traiter efficacement des problème de grande taille. De plus, `VNS/LDS+CP` permet de tirer parti des mécanisme de filtrage liés aux WCSP. Pour nos expérimentations, nous utiliserons EDAC, qui fournit le meilleur compromis entre la qualité de filtrage obtenu et la complexié temporelle de son calcul. Enfin, pour les besoins de comparaisons, nous avons aussi retenu `ID-Walk` en raison de ses bonnes performances sur différentes instances de problèmes WCSP.

Chapitre 4

Jeux de test

Sommaire

Dans ce chapitre nous présentons les quatre jeux de données (`RLFAP`, `GRAPH`, `SPOT5` et `tagSNP`) utilisés dans nos expérimentations. Pour chacun, nous indiquons ses principales caractéristiques, dont sa taille et sa difficulté de résolution ainsi que le coût de la solution optimale (si elle est connue). Pour les instances `RLFAP`, nous détaillons également la structure des réseaux de fonctions de coût de certaines instances.

4.1 Problème d'allocation de fréquences à des liens radio

Le but des instances `RLFAP` est d'assigner, à un ensemble de liens radios, une fréquence tout en minimisant les interférences entre les liens.

4.1.1 Description du problème

Le problème d'allocation de fréquences à des liens radio (Radio Link Frequency Assignment Problem, `RLFAP`) est composé d'un ensemble de liens radio, représentant chacun une communication entre deux sites. L'objectif consiste à attribuer, à chaque lien, une fréquence radio en tenant compte des contraintes suivantes :

– chaque lien radio est limité à une plage de fréquences,

- des interférences peuvent survenir lorsque qu'un même site reçoit et émet en même temps sur des fréquences proches, ou qu'une certaine distance sépare deux sites,
- lorsqu'un site A communique avec un site B sur une fréquence, le site B doit lui répondre sur une fréquence décalée d'une valeur δ. Les liens radio entre les sites A et B sont appelés *duplex links*.
- certains liens radios peuvent être initialement affectés.

4.1.2 Modélisation sous forme d'un WCSP

Ce problème peut être modélisé comme suit. L'ensemble des variables, X, représente l'ensemble des liens radio à affecter et D l'ensemble des fréquences disponibles pour chaque lien. Les fonctions de coût sont les suivantes :

- une fonction de coût unaire (équation 4.1) est associée pour chaque lien radio x_i initialement affecté à la fréquence f_i,
- une fonction de coût binaire (équation 4.2) représente les interférences possibles entre deux liens radio x_i et x_j ; d_{ij} représente l'écart minimal entre les fréquences des deux liens radio.
- une fonction de coût binaire (équation 4.3) représente les *duplex link* entre les liens x_i et x_j ; δ_{ij} représente la valeur de décalage.

$$x_i = f_i \tag{4.1}$$
$$|x_i - x_j| > d_{ij} \tag{4.2}$$
$$|x_i - x_j| = \delta_{ij} \tag{4.3}$$

4.1.3 Description des instances

Suivant l'instance à résoudre, l'objectif ainsi que le type des fonctions de coût (i.e. dures ou molles) peuvent varier. Pour les instances fournies par le Centre d'Electronique de l'Armement (CELAR), les fonctions de coût du type 4.3 sont toujours dures et les autres molles. L'objectif peut être :

- CARD : de minimiser le nombre de fréquences utilisées pour affecter les liens radios,
- SPAN : de minimiser la valeur de la fréquence la plus élevée utilisée,
- MAX : de minimiser la somme des coûts. Ce critère est pertinent lorsque toutes les fonctions de coût ne peuvent être satisfaites simultanément.

Pour tous ces cas, le problème `RLFAP` est *NP-complet*, puisque les fonctions de coût de type 4.3 permettent d'exprimer le problème de k-coloration [Cabon *et al.*, 1999]. Le tableau 4.1 résume les principales caractéristiques des instances fournies par le CELAR : nombre de variables n, la taille du plus grand domaine d, nombre de fonctions de coût e et meilleure solution connue S^*.

Scen06, Scen07 et Scen08 sont les instance réputées difficiles à résoudre. Nous allons donc limiter nos expérimentations à ces trois instances dans la suite de ce mémoire.

4.1.3.1 Pré-traitements effectués sur les instances fournies par le CELAR

Grâce aux fonctions de coût d'égalité définies dans l'équation 4.3, le nombre de variables définies dans les instances fournies par le CELAR peut être divisé par 2, puisque les contraintes d'égalité sont *bijectives* ; pour une fonction de coût portant sur deux variables x_i et x_j, celle-ci est dite *bijective* si pour toute valeur du domaine de x_i il existe une et une seule valeur compatible dans le domaine de x_j (et réciproquement). Ainsi, en remplacant toute paire de variables reliées

Scen#	n	d	e	Type	Satisfaisable	S^*
01	916	44	5548	CARD	Oui	16
02	200	44	1235	CARD	Oui	14
03	400	44	2760	CARD	Oui	14
04	680	44	3967	CARD	Oui	46
05	400	44	2598	SPAN	Oui	792
06	200	44	1322	MAX	Non	3389
07	400	44	2866	MAX	Non	343 592
08	916	44	5745	MAX	Non	262
09	680	44	4103	MAX	Non	15571
10	680	44	4103	MAX	Non	31516
11	680	44	4103	CARD	Oui	22

TABLE 4.1 – Tableau des principales caractéristiques des instances fournies par le CELAR.

par une fonction de coût bijective par une seule variable et en mettant à jour les tuples des autres fonctions de coût, il est possible pour ces instances de diviser le nombre de variables par 2 et de retirer les fonctions de coût bijectives du problème.

L'instance obtenue peut alors contenir plusieurs fonctions de coût portant sur un même ensemble de variables. Pour des raisons évidentes d'efficacité, liées essentiellement au filtrage, les tuples de ces fonctions de coût sont réunis dans une seule fonction de coût.

4.1.3.2 Graphes de contraintes des instances du CELAR

Les figures 4.1 à 4.3 détaillent les graphes de contraintes associés aux instances Scen06 à Scen08. Sur ces instances, les variables contenues dans une même ellipse forment une clique.

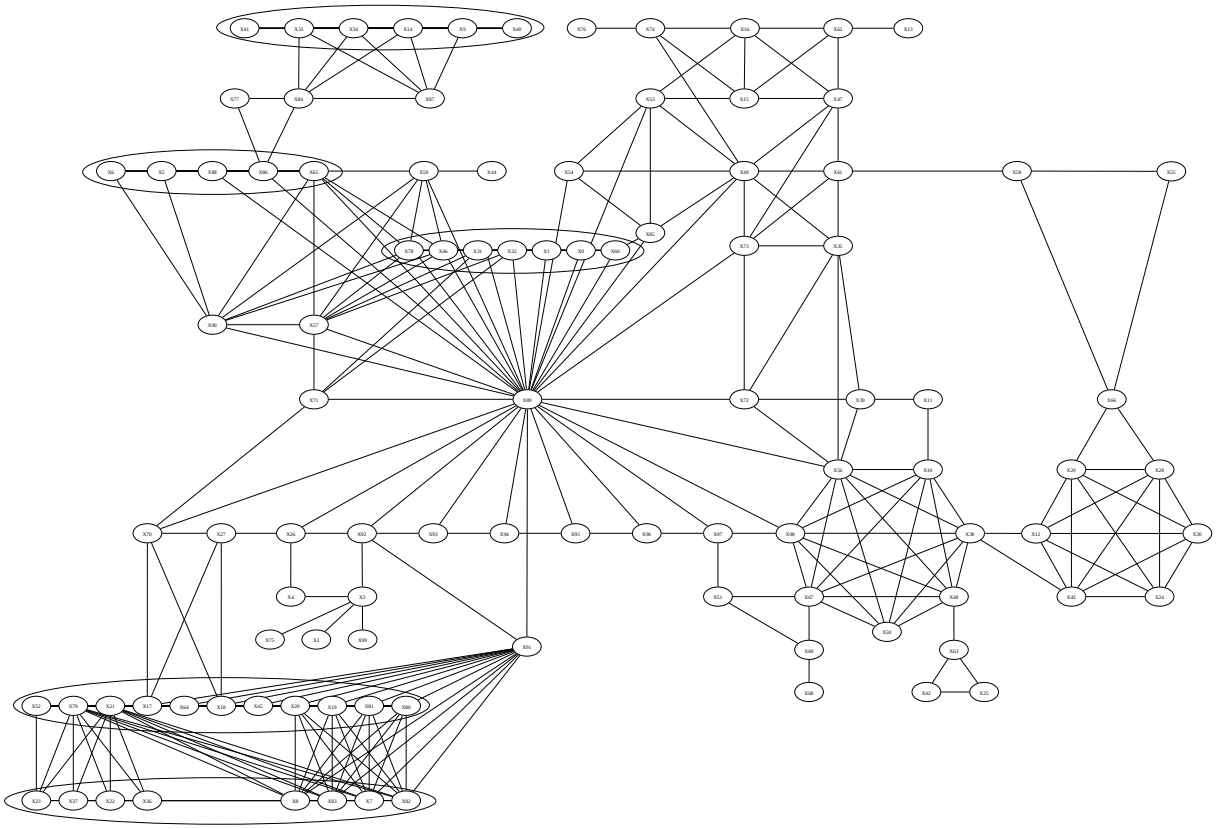


FIGURE 4.1 – Graphe de contraintes de l'instance Scen06 du CELAR.

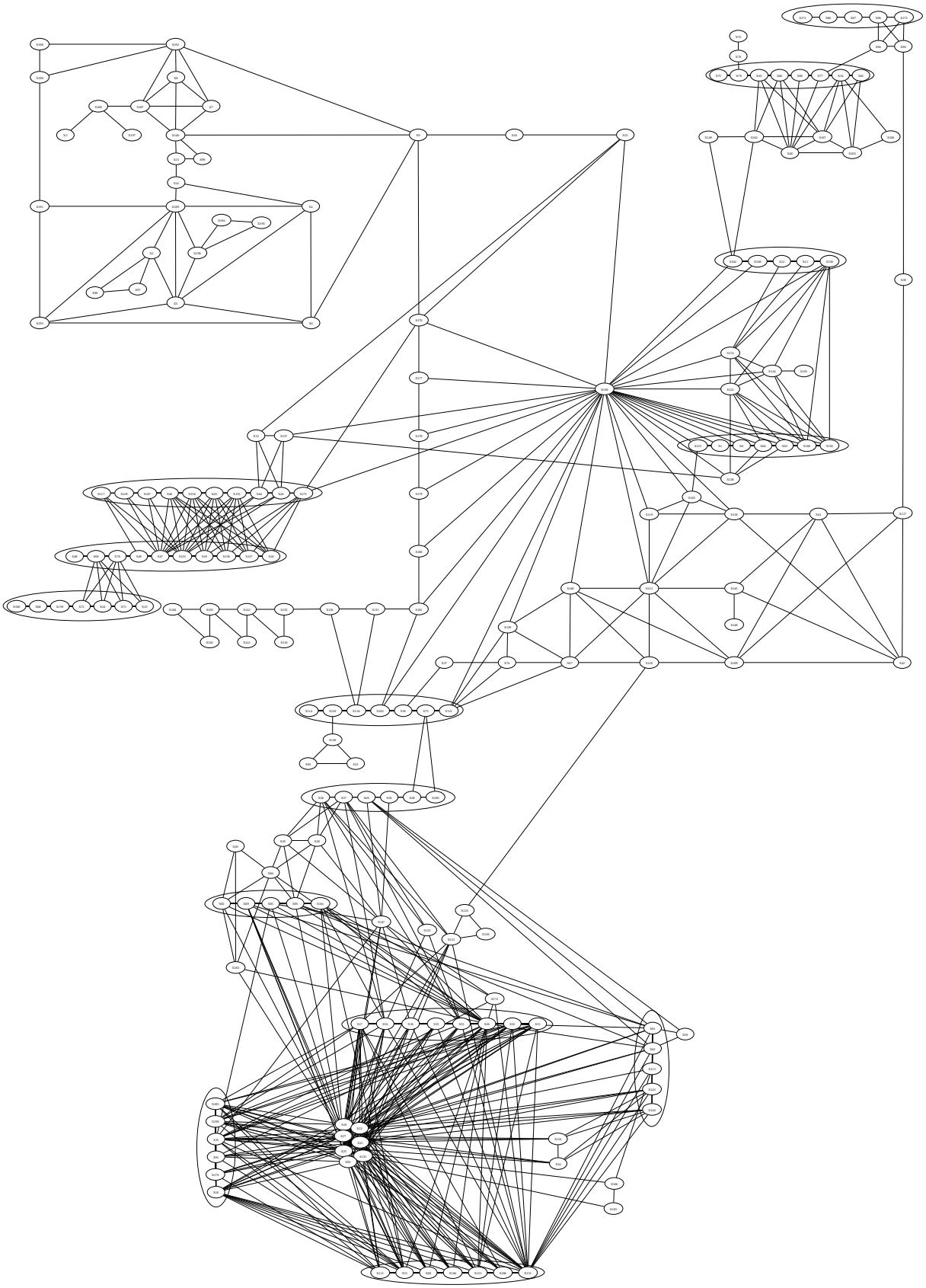


FIGURE 4.2 – Graphe de contraintes de l'instance Scen07 du CELAR.

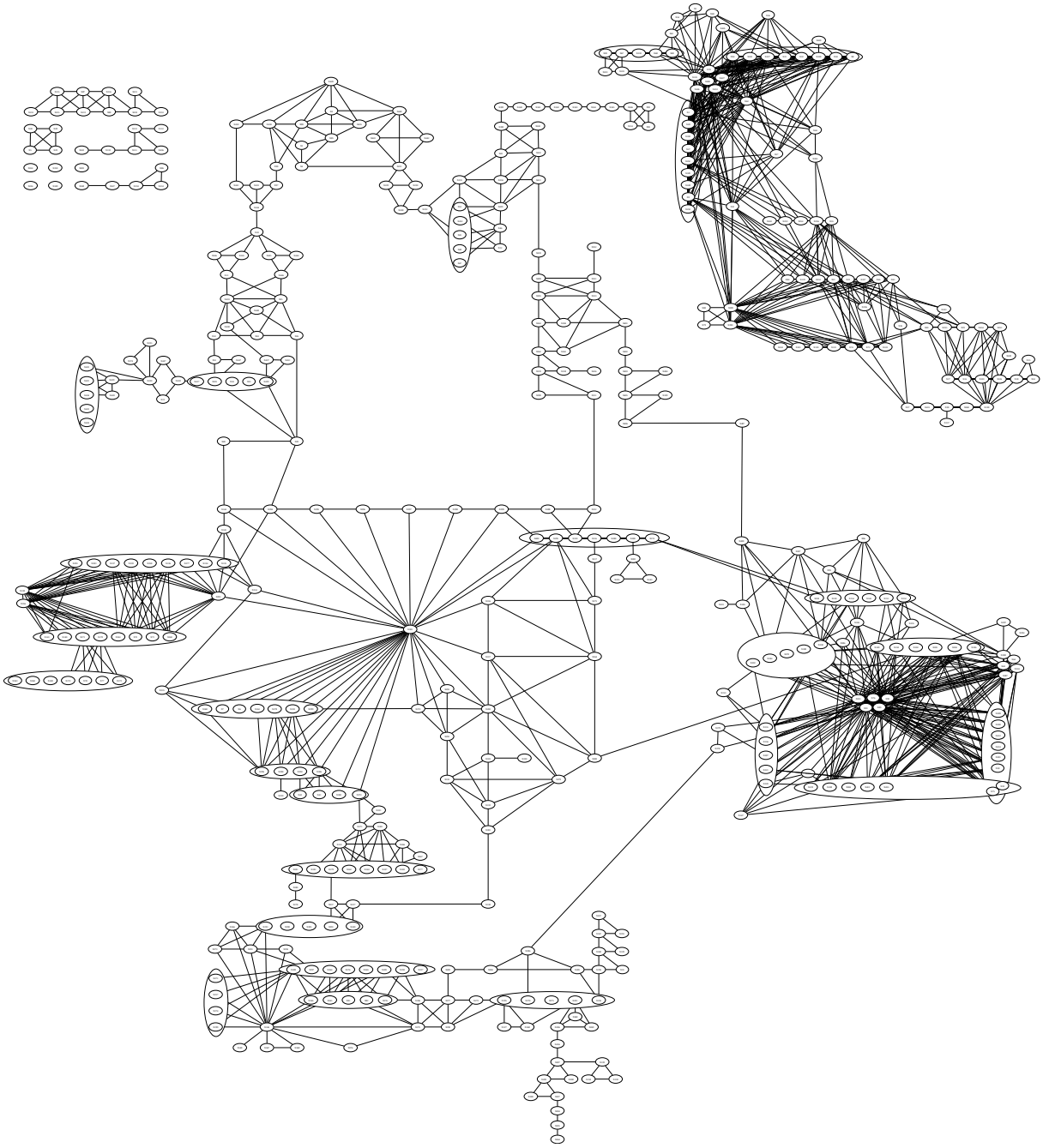


FIGURE 4.3 – Graphe de contraintes de l'instance Scen08 du CELAR.

Graph#	n	d	e	Type	Satifaisable	S^*
01	200	44	1134	CARD	Oui	18
02	400	44	2245	CARD	Oui	14
03	200	44	1134	SPAN	Oui	380
04	400	44	2244	SPAN	Oui	394
05	200	44	1134	MAX	Non	221
06	400	44	2170	MAX	Non	4123
07	400	44	2170	MAX	Non	4324
08	680	44	3757	CARD	Oui	18
09	916	44	5246	CARD	Oui	18
10	680	44	3907	SPAN	Oui	394
11	680	44	3757	MAX	Non	3080
12	680	44	4017	MAX	Non	11827
13	916	44	5273	MAX	Non	10110
14	916	44	4638	CARD	Oui	10

TABLE 4.2 – Tableau des principales caractéristiques des instances générées par GRAPH.

4.2 Instances générées par GRAPH

`GRAPH`, *Generating Radio link frequency Assignment Problems Heuristically* [Benthem, 1995], est un générateur d'instances aléatoires structurées. L'objectif de GRAPH est de générer des instances de difficulté et de structure similaires à celles fournies par le CELAR. Ces instances possèdent les caractéristiques suivantes :

- les domaines sont identiques à ceux des instances du CELAR,
- le ratio entre le nombre de liens radio et leurs interférences est approximativement le même que celui des instances du CELAR,
- la structure des instances pour les fonctions de coût de l'équation 4.3, est la même que celle des instances du CELAR.

Le type des instances générées par GRAPH peut être sélectionné parmi : Feas, CARD, SPAN ou MAX. Lors de la génération des instances de type Feas, CARD ou SPAN, GRAPH fournit les solutions optimales. Pour les instances de type MAX, le générateur fournit une borne supérieure du coût de la solution optimale. Le tableau 4.2 détaille les principales caractéristiques des instances générées par GRAPH.

4.3 Problème de planification d'un satellite de prise de vue

La gestion d'un satellite de prise de vue, tel que le satellite *Spot*, consiste à planifier chaque jour les prises de vue à réaliser le lendemain. Le but est de maximiser les gains liés à la vente de ces prises de vue tout en prenant en compte les contraintes techniques du satellite, mais aussi météorologiques ou géographiques. Une description detaillée du problème est donnée dans [Bensana *et al.*, 1999].

4.3.1 Description du problème

Un problème de planification `SPOT5` est composé d'un ensemble S de prises de vue pouvant être réalisées. On associe à chaque photo un poids, issu de l'agrégation de plusieurs critères

(urgence, importance du client, prévision météorologique du lieu...). Chaque photographie peut être réalisée sur l'un des trois appareils du satellite en mono. Si elle doit être réalisée en stéréo, cela nécessitera obligatoirement l'utilisation des appareils avant et arrière du satellite. Enfin, il faut respecter un certain nombre de contraintes dures :

- les prises de vue ne peuvent pas se chevaucher et il faut respecter un temps de transition avant de pouvoir prendre une nouvelle photographie avec un même appareil ;
- la quantité d'information transitant dans le satellite à un moment donné est limitée ;
- la capacité d'enregistrement est elle aussi limitée.

Le but est de sélectionner un sous-ensemble de prises de vue $S' \subseteq S$ qui permet de satisfaire les contraintes dures tout en minimisant la somme des poids des prises de vues qui se sont pas réalisées.

4.3.2 Modélisation sous forme d'un WCSP

La modélisation sous forme d'un WCSP des problèmes `SPOT5` est réalisée de la façon suivante :

- Une variable x_i est associée à chaque prise de vue.
- Le domaine D_i associé à x_i contient les appareils possibles pour cette prise de vue (0 désigne le fait de ne pas prendre la photographie) :
  - $\{1, 2, 3, 0\}$ pour une photographie mono, chaque valeur correspondant à un appareil ;
  - $\{13, 0\}$ pour une photographie stéréo, 13 signifiant l'utilisation des appareils 1 et 3.
- Une fonction de coût unaire w_i associée à chaque variable x_i interdisant la valeur 0 avec un coût égal au poids de la photographie (le coût engendré par le fait de ne pas prendre la photographie).
- Les contraintes de chevauchement sont traduites par des fonctions de coût binaires dures attribuant un coût $\top$ [1] si deux prises de vue ne respectent pas les temps de transition.
- Les contraintes de débit sont représentées par des fonctions de coût dures binaires ou ternaires.
- La contrainte de capacité mémoire est représentée par une fonction de coût $n - aire$ dure.

Cette modélisation assure l'existence d'une solution ne violant aucune des fonctions de coût dures. En effet, ne sélectionner aucune photographie produit une solution admissible de très mauvaise qualité.

4.3.3 Présentation des instances

Les données proposées par le CNES contiennent 498 instances réparties en deux groupes.

Le premier groupe contient 384 problèmes ne considérant qu'une orbite afin d'éliminer la limitation de capacité mémoire. Ils sont repartis en 3 groupes :

- les instances #1 à #362 ont été générées par un simulateur (LOSICORDOF) ;
- les instances #401 à #413 ont été construites à partir de la plus grande instances générée (la numéro 11) en réduisant le nombre de prises de vue ;
- les instances #501 à #509 elles aussi sont construites à partir de l'instance 11, mais cette fois-ci en réduisant le nombre de conflits entre les photographies.

Le second groupe contient 114 problèmes considérant plusieurs orbites consécutives, et donc cette fois-ci en prenant en compte la limitation de capacité mémoire. Ils sont aussi répartis en 3 groupes :

1. $\top$ est un entier supérieur à la somme des poids de toutes les prises de vue.

Instance	n	d	e	S^*
#408	200	4	2,232	6,228
#412	300	4	4,348	32,381
#414	364	4	10,108	38,478
#505	240	4	2,242	21,253
#507	311	4	5,732	27,390
#509	348	4	8,624	36,446

TABLE 4.3 – Caractéristiques des instances `SPOT5`.

- les instances #1000 à #1101 ont été générées par le simulateur ;
- les instances #1401 à #1406 ont été construites à partir de la plus grande instance générée (la numéro 1021) en réduisant le nombre de prises de vue ;
- les instances #1501 à #1509 sont elles aussi construites à partir de la même l'instance en réduisant le nombre de conflits entre les photographies.

Nous avons sélectionné 6 instances les plus difficiles, sans contraintes de mémoire, dont nous reportons les caractéristiques dans le tableau 4.3, avec n le nombre de variables, d la taille du plus grand domaine, e le nombre de fonctions de coût et S^* le coût de la solution optimale.

4.4 Problème de sélection de TagSNP

Le problème de sélection de TagSNP est issu de la recherche en génétique et plus précisément en analyse du polymorphisme. Un SNP (*Single Nucleotide Polymorphism*) est une variation ponctuelle apparaissant dans le génome d'individus d'une même espèce. Un SNP se manifeste par la modification d'une seule paire de bases de nucléotides (A, T, C ou G) du génome, et existe pour environ 2 paires sur mille, représentant ainsi 90% des variations génétique chez l'humain. L'étude des SNP est capitale pour la recherche sur les maladies génétiques héréditaires. Cependant, la taille des données les rend impossibles à manipuler en pratique. Le problème de sélection de TagSNP vise à compresser (avec perte) les données en choisissant un sous-ensemble de SNP qui représenteront les autres.

4.4.1 Description du problème

Le but est de trouver un ensemble de SNP représentant tous les SNP. Un TagSNP est considéré comme représentatif d'un autre SNP à traiter s'ils sont suffisamment corrélés dans la population étudiée. La corrélation entre deux SNP i et j est représentée par une mesure notée r^2_{ij} et doit être supérieure à un seuil θ pour que la corrélation soit significative. Cette valeur est généralement fixée à $\theta = 0.8$. On construit alors un graphe $G = (V, E)$ dans lequel chaque SNP i est représenté par une sommet u_i et $\{u_i, u_j\} \in E$ si $r^2_{ij} > \theta$. On appelle ce graphe *graphe de corrélation*. Le problème est alors réduit à un *Set Covering problem* (NP-difficile), c'est à dire trouver un ensemble de TagSNP de taille minimale.

Le nombre de solutions étant très grand, on cherche à exprimer des préférences entre les solutions. L'objectif est alors de trouver un ensemble de TagSNP maximisant deux critères. Le premier est la représentativité du groupe. Celui-ci doit en effet représenter les SNP non sélectionnés en restant le plus petit possible. Le second critère est la dispersion. On cherche à minimiser la corrélation entre deux TagSNP.

Instance	n	d	e	S^*
#3792	528	59	12,084	6,359,805
#4449	464	64	12,540	5,094,256
#6835	496	90	18,003	4,571,108
#8956	486	106	20,832	6,660,308
#9319	562	58	14,811	6,477,229
#15757	342	47	5,091	2,278,611
#16421	404	75	12,138	3,436,849
#16706	438	30	6,321	2,632,310
#6858	992	260	103,056	20,162,249
#9150	1,352	121	44,217	43,301,891
#10442	908	76	28,554	21,591,913
#14226	1,058	95	36,801	25,665,437
#17034	1,142	123	47,967	38,318,224

TABLE 4.4 – Caractéristiques des instances TagSNP.

4.4.2 Modélisation sous forme d'un WCSP

Soit un graphe pondéré $G = (V, E)$ dans lequel chaque sommet $u \in V$ représente un SNP et chaque arête $(u, v) \in E$ est pondérée par la mesure r^2 entre les SNP u et v (r^2 devant être supérieur à θ). Le problème de sélection de TagSNP est modélisé par un WCSP binaire construit de la façon suivante :

- chaque SNP i est représenté par deux variables :
  - i_s, variable booléenne indiquant la sélection du SNP en tant que TagSNP ;
  - i_r, variable de représentation ayant pour domaine l'ensemble des voisins de i dans G. Elle indique le TagSNP représentant i.
- Une fonction de coût binaire dure est associée à chaque SNP i afin d'assurer la cohérence entre i_s et i_r ($i_s \implies i_r = i$).
- Une fonction de coût binaire dure assure la cohérence entre un SNP et son représentant ($i_r = j \implies j_s$).
- Une fonction de coût unaire sur i_s génère un coût de $\lfloor 100.\frac{1-r^2_{i,i_r}}{1-\theta} \rfloor$ si $i_r \neq i$. Cette fonction de coût capture la représentativité du SNP. Plus la corrélation du SNP i avec son représentant i_r est forte, plus le coût généré est faible.
- Une fonction de coût binaire entre i_s et j_s génère un coût $\lfloor 100.\frac{r^2_{ij}-\theta}{1-\theta} \rfloor$ si les SNP i et j sont sélectionnées ($i_s = i_j = vrai$). Cette fonction de coût permet de forcer la dispersion des TagSNP. En effet, plus deux TagSNP seront corrélés, plus le coût généré sera élevé.

4.4.3 Présentation des instances

Les instances que nous traitons dans ce document ont été générées à partir des données liées au chromosome humain 1. Afin de créer des instances suffisamment difficiles, le seuil θ a été fixé à 0.5. Chacune des 19,750 instances correspond à une composante connexe du graphe de SNP obtenu. Nous avons sélectionné 13 instances parmi les plus difficiles pour nos expérimentations. Le tableau 4.4 présente leurs caractéristiques.

4.4.4 Décomposition arborescente

Pour nos expérimentations des chapitres 5, 6 et 7, nous allons utiliser, pour les instances `tagSNP`, les décompositions proposées dans [Sánchez *et al.*, 2009]. Ces décompositions sont différentes car elles utilisent un ordre `MCS` qui tient compte de la structure du graphe de contraintes découlant de la modélisation proposée. En effet, comme nous allons le voir dans le chapitre 4, chaque SNP est représenté par 2 variables : une variable de sélection i_s et une variable de représentation i_r. Chaque couple (i_s, i_r) est relié à tous les SNP voisins par leurs variables de sélection j_s. Ainsi deux variables i_r et j_r ne sont jamais directement reliées. Cependant, lors du jeu d'élimination, si une variable de sélection est retirée avant la variable de représentation i_r, une arête sera ajoutée entre i_r et les variables de représentations de tous les SNP voisins de i. Afin d'éviter cela, les auteurs proposent donc de créer l'ordre `MCS` à partir du graphe de corrélation (composé uniquement des variables i_s). On intercale ensuite dans l'ordre obtenu les variables de représentation avant leur variables de sélections associées ($\alpha = i_r, i_s, j_r, j_s, ...$). Dans le reste du document, nous considérons donc les décompositions obtenues à partir de ces ordres, car elles sont plus proches de la structure réelle du problème considéré.

Deuxième partie

Contributions

Chapitre 5

VNS guidée par la décomposition arborescente

Sommaire

Comme nous l'avons indiqué dans l'introduction, il n'existe pas de travaux permettant d'exploiter la structure d'un problème au sein des méthodes de recherche locale. Exploiter une telle information permettrait, d'une part, d'identifier des structures de voisinage pertinentes et d'autre part, de mieux guider l'exploration de l'espace de recherche. Les méthodes de décomposition arborescente vues au chapitre 2 nous semblent donc être de bons candidats pour exhiber une telle structure.

La figure 5.1 présente le graphe de contraintes associé à l'instance Scen08 du problème d'allocation de fréquence à des liens radio (RLFAP) [Cabon *et al.*, 1999]. Comme nous pouvons le constater, plusieurs parties du problème sont fortement connectées, elles-mêmes étant peu liées au reste du graphe de contraintes, voire complètement déconnectées de celui-ci. Ce sont ce type de structures (des sous-parties de problème faiblement reliées entre elles) que nous souhaitons identifier et exploiter. Plusieurs autres problèmes étudiés durant cette thèse, tels que les tagSNP en génétique [Hirschhorn et Daly, 2005] (cf section 4.4), ou encore le problème de sélection des prises de vue d'un satellite d'observation terrestre (SPOT5) [Bensana *et al.*, 1999] (cf section 4.3) exhibent ce type de structures.

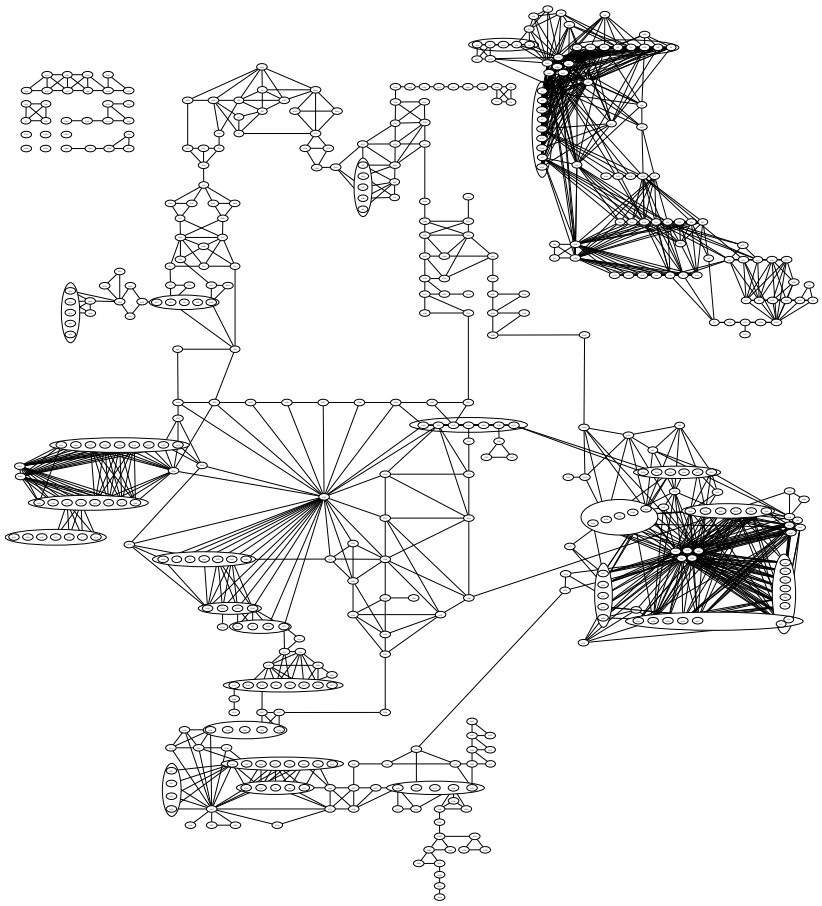


FIGURE 5.1 – Graphe de contraintes associé à l'instance Scen08 de RLFAP.

La figure 5.2 [2] présente une version simplifiée du graphe de clusters de la Scen08. On peut remarquer que la décomposition arborescente permet de bien identifier les sous-parties du problème fortement connectées et de les regrouper en clusters. Notre intuition est de sélectionner les variables à désinstancier dans un même cluster. En effet, toute affectation d'une variable du cluster impacte directement l'affectation des autres variables appartenant au même cluster. Par ailleurs, le graphe de clusters nous renseigne sur la répartition topologique des variables du problème. Cela peut nous permettre d'assurer une meilleure couverture des différentes parties du problème à considérer, dans le but de renforcer la diversification et d'assurer un meilleur compromis entre intensification et diversification.

Plan du chapitre. Nous proposons, tout d'abord, un premier schéma de recherche locale, noté DGVNS, extension de VNDS, exploitant la décomposition arborescente pour guider efficacement l'exploration de l'espace de recherche (section 5.1). Ce schéma utilise la décomposition arborescente comme une carte du problème afin de construire des voisinages pertinents. Nous étudions ensuite trois différentes stratégies visant à équilibrer l'intensification et la diversification au sein de DGVNS (section 5.2). Les expérimentations menées sur différentes instances des problèmes RLFAP, GRAPH, SPOT5 et tagSNP montrent la robustesse de notre approche sur des problèmes réels (section 5.3). Tout d'abord, nous comparons DGVNS avec deux méthodes de recherche locale dédiées à la résolution des WCSP : VNS/LDS+CP et ID-Walk (section 5.3.2). Ensuite, nous étudions l'influence

2. figure fournie par Simon De Givry.

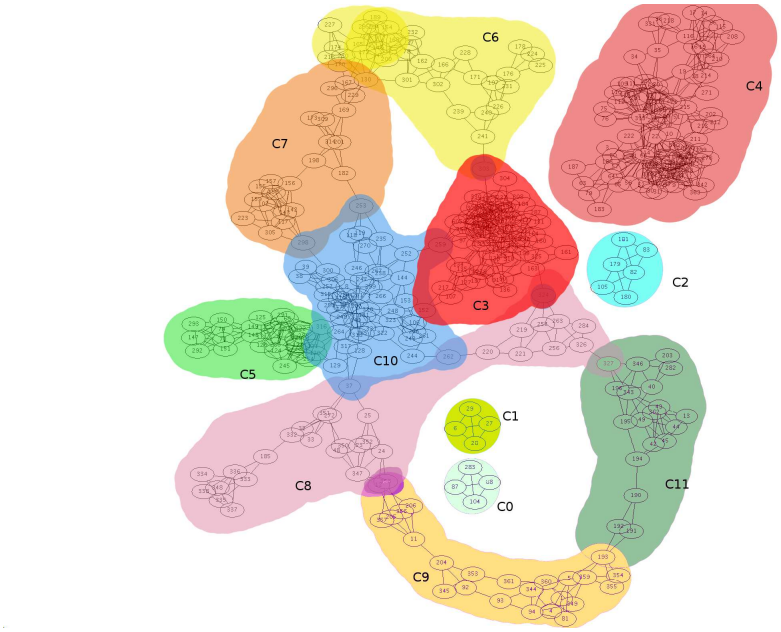


FIGURE 5.2 – Graphe de clusters associé à l'instance Scen08 de RLFAP.

de la largeur de décomposition sur les performances de notre approche (section 5.3.3). Puis, nous comparons nos différentes stratégies d'intensification et de diversification (section 5.3.4). Enfin, nous concluons.

5.1 VNS Guidée par la Décomposition (DGVNS)

Partant des constats effectués dans l'introduction, nous proposons une nouvelle méthode, notée DGVNS (*Decomposition Guided* VNS), permettant d'exploiter le graphe de clusters pour (i) **construire des structures de voisinage basées sur la notion de cluster** et pour (ii) **renforcer le schéma d'intensification et de diversification** proposé par la méthode VNS/LDS+CP. Il s'agit à notre connaissance de la première proposition permettant d'exploiter la décomposition arborescente d'un graphe de contraintes dans les méthodes de recherche locale à voisinages étendus telles que VNS.

5.1.1 Structures de voisinage basées sur la notion de cluster

Le choix de structures de voisinage est une étape critique dans les méthodes de recherche locale, puisqu'elles déterminent les sous-espaces à explorer afin de trouver des solutions de meilleure qualité. Toutefois, l'identification de structures de voisinage pertinentes dépend souvent du domaine d'application et nécessite, par conséquent, un temps d'expertise non négligeable. Nous proposons de nouvelles structures de voisinage génériques qui tiennent compte de la topologie du graphe de clusters.

Afin de tirer parti du fort lien entre les variables d'un cluster, à la place des structures de voisinage N_k dans VNS/LDS+CP (cf. section 3.3.2), DGVNS utilise des structures de voisinage $N_{k,i}$ permettant de focaliser l'exploration de l'espace de recherche autour d'un cluster.

Définition 59 (Structure de voisinage $N_{k,i}$)**.** *Soit G un graphe et G_T=($\mathcal{C}_T$,E_T) le graphe de*

Algorithme 21 : Pseudo-code de DGVNS

```
1  fonction DGVNS(X, W, k_init, k_max, δ_max) ;
2  début
3      Soit G le graphe de contraintes de (X, W) ;
4      Soit (C_T,T) une décomposition arborescente de G ;
5      Soit C_T = {C_1, C_2, ..., C_p} ;
6      S ← genSolInit() ;
7      k ← k_init ;
8      i ← 1 ;
9      tant que (k < k_max) ∧ (not TimeOut) faire
10         C_s ← CompleteCluster(C_i, k) ;
11         X_un ← Hneighborhood(N_{k,i}, C_s, W, S) ;
12         A ← S\{(x_i, a) | x_i ∈ X_un} ;
13         S' ← LDS + CP(A, X_un, δ_max, f(S), S) ;
14         NeighbourhoodChangeDGVNS(S, S', k, i) ;
15     retourner S ;
16 fonction CompleteCluster (C_i, k) ;
17 début
18     Soit C_s ← C_i ;
19     initqueue(F) ;
20     pour tout cluster C_j ∈ vois(C_i) faire
21         enqueue(F, C_j) ;
22     tant que |C_s| < k faire
23         C_f ← dequeue(F) ;
24         C_s ← C_s ∪ C_f ;
25         pour tout cluster C_j ∈ vois(C_f) faire
26             enqueue(F, C_j) ;
27     retourner C_s ;
```

clusters associé. Soit C_i un cluster de $\mathcal{C}_T$. La structure de voisinage $N_{k,i}$ désigne l'ensemble des combinaisons de k variables (k étant la dimension du voisinage) parmi C_i.

Ainsi, pour favoriser les mouvements dans des régions fortement liées, l'ensemble des k variables à désinstancier est sélectionné dans un même cluster C_i (i.e. ensemble des variables candidates). En effet, le cluster est une structure qui permet d'identifier ces régions, du fait de sa taille (plus petite que le problème original), et du lien fort entre les variables qu'il contient. De plus, grâce à la forte connectivité entre les variables, l'étape de reconstruction pourra bénéficier d'un filtrage plus efficace et d'un calcul de minorant de bien meilleure qualité. Afin de tirer parti de la topologie du graphe de clusters, nous proposons d'élargir l'ensemble des variables candidates aux clusters voisins de C_i (cf. section 5.1.2).

5.1.2 Pseudo-code de DGVNS

L'algorithme 21 décrit le pseudo-code de DGVNS. Tout d'abord, DGVNS commence par calculer une décomposition arborescente du graphe de contraintes de l'instance à résoudre (ligne 4), puis génère aléatoirement une solution initiale S (ligne 6). Soit C_s l'ensemble des variables candidates à être désinstanciées. Si le cluster courant C_i contient suffisamment de variables (i.e. $|C_i| \geq k$), alors les k variables à désinstancier sont sélectionnées dans le même cluster C_i. Sinon

(i.e. $k > |C_i|$), nous complétons l'ensemble C_s en ajoutant des variables des clusters C_j adjacents de C_i. Ainsi, nous tenons compte de la topologie du graphe de clusters. Ce traitement est assuré par la fonction `CompleteCluster`(C_i, k) (ligne 10). Ensuite, un sous-ensemble de k variables est sélectionné dans le voisinage $N_{k,i}$ (i.e. l'ensemble des combinaisons de k variables parmi C_s) par la fonction `Hneighborhood` (ligne 11). Une affectation partielle $\mathcal{A}$ est alors générée à partir de la solution courante S, en désaffectant les k variables sélectionnées ; les autres variables (i.e. non sélectionnées) gardent leur affectation dans S (ligne 12). Tout comme `VNS/LDS+CP`, la solution est alors reconstruite en utilisant une recherche arborescente partielle de type LDS, aidée par une propagation de contraintes (CP) basée sur un calcul de minorants (ligne 13). La procédure `NeighborhoodChangeDGVNS` détermine la stratégie d'intensification ou de diversification en fonction de la qualité de la nouvelle solution S' (ligne 14). Elle sera détaillée en section 5.2. Enfin, l'algorithme s'arrête dès que l'on a atteint la dimension maximale du voisinage à considérer k_{max} ou le *TimeOut* (ligne 15).

Le pseudo-code de la fonction `CompleteCluster` est détaillé par l'algorithme 21 (lignes 16-27). Au départ, l'ensemble C_s est initialisé par les variables de C_i (ligne 18) et les clusters voisins de C_i dans le graphe de clusters sont ajoutés dans une file F (ligne 20). Tant que ($|C_s| < k$), un cluster C_f est extrait de la file F (ligne 23), puis l'ensemble C_s est complété par les variables de C_f (ligne 24), et les clusters voisins de C_f sont alors ajoutés à la file F (lignes 25-26). La boucle s'arrête dès que l'ensemble C_s contient k variables ou plus, et la fonction retourne l'ensemble C_s.

5.2 Stratégies d'intensification et de diversification dans DGVNS

Nous proposons trois stratégies qui exploitent le graphe de clusters, permettant une meilleure intensification et diversification dans `DGVNS`. Elles correspondent à différents schémas de déplacement entre clusters en fonction de l'amélioration ou non de la qualité de la solution courante. Notre objectif est d'assurer une meilleure couverture des différentes parties du problème à résoudre, tout en se focalisant de manière plus intensive dans des régions plus prometteuses. Dans ce qui suit, on notera par `DGVNS-i` la méthode `DGVNS` utilisant la stratégie `NeighborhoodChangeDGVNS-i`. Tout d'abord, nous revenons sur le rôle joué par l'intensification et la diversification dans les méthodes de recherche locale (cf définition 54 et 55, section 3.1). Ensuite, nous détaillons nos trois stratégies.

5.2.1 Intensification versus diversification

Le but de l'intensification est de se concentrer plus intensivement sur une petite région de l'espace de recherche pour converger vers un minimum local. Dans `VNS/LDS+CP`, l'intensification est réalisée par la reconstruction de la solution partielle avec `LDS+CP` et par la réduction de la taille du voisinage à k_{init} à chaque amélioration. Cela permet d'accélérer la recherche d'une instanciation complète dans des voisinages de petite taille.

À l'inverse, le but de la diversification est de traiter un grand nombre de régions différentes, afin d'assurer que l'espace de recherche est largement exploré, et de parvenir à localiser la région contenant l'optimum global. Dans `VNS/LDS+CP` (cf section 3.3.2), la diversification est assurée en élargissant la dimension des voisinages à considérer (i.e. passer de k à $(k+1)$), à chaque fois que la phase de reconstruction n'arrive pas à améliorer la solution courante.

Cependant, comme récemment mentionné dans [Linhares et Yanasse, 2010], la plupart des algorithmes de recherche locale considèrent diversification et intensification comme deux

mécanismes antagonistes : à mesure qu'on effectue plus d'intensification, on peut perdre en diversification, et vice versa. Une plus grande coordination entre ces deux mécanismes est donc nécessaire.

Algorithme 22 : Pseudo-code de `NeighborhoodChangeDGVNS-1`

```
1 Procédure NeighbourhoodChangeDGVNS-1(S, S', k, i) ;
2 début
3     si f(S') < f(S) alors
4         S ← S' ;
5         k ← k_init ;
6         i ← succ(i) ;
7     sinon
8         k ← k + 1 ;
9         i ← succ(i) ;
```

5.2.2 DGVNS-1 : changer systématiquement de cluster

DGVNS-1 considère successivement tous les clusters C_i. L'algorithme 22 détaille le pseudo-code de la procédure `NeighborhoodChangeDGVNS-1`. Soit p le nombre total de clusters, *succ* une fonction successeur, qui associe à un cluster i son successeur $C_{succ(i)}$ [3] et $N_{k,i}$ la structure de voisinage courante.

Si aucune solution de meilleure qualité que la solution courante n'a été trouvée par LDS+CP dans le voisinage de S, DGVNS-1 cherche de nouvelles améliorations dans $N_{(k+1),succ(i)}$ (structure de voisinage où $(k+1)$ variables seront désinstanciées (lignes 8-9)). Premièrement, la diversification réalisée par le déplacement du cluster C_i vers le cluster $C_{succ(i)}$ permet de favoriser l'exploration de nouvelles régions de l'espace de recherche et de tenter de trouver des solutions de meilleure qualité qui peuvent se trouver dans ces régions. Deuxièmement, quand un minimum local est atteint dans le voisinage courant, passer de k à $k+1$ permet de renforcer la diversification par l'élargissement de la dimension du voisinage. En effet, plus la dimension du voisinage est grande, plus il a de chances de contenir des solutions meilleures que la solution courante.

À l'inverse, si LDS+CP trouve une solution de meilleure qualité S' dans $N_{k,i}$ (ligne 3), alors celle-ci devient la solution courante (ligne 4), k est réinitialisé à k_{init} (ligne 5) et le cluster suivant est considéré (ligne 6). En effet, en restant dans le même cluster, il est plus difficile d'obtenir une nouvelle amélioration : changer de cluster permettra de diversifier l'exploration autour de la nouvelle solution S'. Ainsi, DGVNS-1 effectue une diversification « agressive » de l'espace de recherche.

5.2.3 DGVNS-2 : changer de cluster en l'absence d'amélioration

Le pseudo-code de la procédure `NeighborhoodChangeDGVNS-2` est détaillé par l'algorithme 23. DGVNS-2 passe au cluster $C_{succ(i)}$ si aucune amélioration de la solution courante n'a été trouvée par LDS+CP (lignes 7-8). Cependant, contrairement à DGVNS-1, si LDS+CP trouve une solution de meilleure qualité S' dans le voisinage de S (ligne 3), DGVNS-2 cherche de nouvelles améliorations dans $N_{k_{init},i}$ (ligne 5). Ainsi, nous intensifions l'exploration dans le voisinage de S' lors des itérations suivantes de DGVNS-2. En effet, rester dans le même cluster permettra de propager,

3. si $i < p$ alors $succ(i) = i + 1$ sinon $succ(p) = 1$.

Algorithme 23 : Pseudo-code de `NeighborhoodChangeDGVNS-2`

```
1 Procédure NeighbourhoodChangeDGVNS-2(S, S', k, i) ;
2 début
3     si f(S') < f(S) alors
4         S ← S' ;
5         k ← k_init ;
6     sinon
7         k ← k + 1 ;
8         i ← succ(i) ;
```

durant la phase de reconstruction, les conséquences des nouvelles affectations des variables de S'. De plus réinitialiser k à k_{init} favorisera de petits mouvements dans le voisinage de S'.

Contrairement à `DGVNS-1`, `DGVNS-2` effectue un compromis entre intensification et diversification. En effet, tant qu'aucune amélioration n'a été trouvée, `DGVNS-2` considère successivement tous les clusters C_i (i.e. diversification), mais dès que la solution courante est améliorée, `DGVNS-2` passe à l'intensification.

Algorithme 24 : Pseudo-code de `NeighborhoodChangeDGVNS-3`

```
1 Procédure NeighbourhoodChangeDGVNS-3(S, S', k, i) ;
2 début
3     si f(S') < f(S) alors
4         S ← S' ;
5         k ← k_init ;
6         i ← succ(i) ;
7     sinon k ← k + 1 ;
```

5.2.4 `DGVNS-3` : changer de cluster après chaque amélioration

Afin d'étudier l'impact du changement de cluster, nous proposons une troisième stratégie `DGVNS-3` qui consiste à rester dans un même cluster tant qu'aucune amélioration n'a été trouvée (cf. algorithme 24). `DGVNS-3` restreint l'effort de diversification dans un cluster en augmentant uniquement la dimension du voisinage (ligne 7). Comme pour `DGVNS-1`, si une solution de meilleure qualité S' est trouvée, `DGVNS-3` cherche de nouvelles améliorations dans le voisinage $N_{k_{init},succ(i)}$ (lignes 5-6).

5.3 Expérimentations

5.3.1 Protocole expérimental

Les expérimentations ont été réalisées sur des instances des problèmes `RLFAP`, `SPOT5`, `tagSNP` et `GRAPH`. Chaque instance a été résolue par chaque méthode, avec une discrepancy de 3 pour `LDS`, qui est la meilleure valeur trouvée pour les instances `RLFAP` [Loudni et Boizumault, 2008]. Les valeurs de k_{min} et de k_{max} ont été fixées respectivement à 4 et n (le nombre de variables du problème). Le *TimeOut* a été fixé à 3600 secondes pour les instances `RLFAP` et `SPOT5`. Pour les

Instance	Méthode	Succ.	Temps	Moy.
Scen06	DGVNS	**50/50**	**112**	**3,389**
$n = 100, e = 1,222$	VNS/LDS+CP	15/50	83	3,399
$S^* = 3,389$	ID-Walk	NA	840	3,447 (3,389)
Scen07	DGVNS	**40/50**	**317**	**345,614**
$n = 200$, $e = 2,665$	VNS/LDS+CP	1/50	461	355,982
$S^* = 343,592$	ID-Walk	NA	360	373,334 (343,998)
Scen08	DGVNS	**3/50**	**1,811**	**275**
$n = 458$, $e = 5,286$	VNS/LDS+CP	0/50	-	394 (357)
$S^* = 262$	ID-Walk	NA	3,000	291 (267)

TABLE 5.1 – Comparaison entre DGVNS, VNS/LDS+CP et ID-Walk sur les instances RLFAP.

instances tagSNP de taille moyenne (resp. de grande taille), le *TimeOut* a été fixé à deux heures (resp. quatre heures). Un ensemble de 50 essais par instance a été réalisé sur un AMD opteron 2,1 GHz et 256 Go de RAM. Toutes les méthodes ont été implantées en C++ en utilisant la librairie toulbar2 [4].

Pour chaque instance et chaque méthode, nous reportons (cf section 3.4) :
- le nombre d'essais avec succès ;
- le temps de calcul moyen pour atteindre l'optimum ;
- le coût moyen des solutions trouvées sur les 50 essais ;
- le meilleur coût obtenu (noté entre parenthèse) si l'optimum n'a pas été atteint ;
- les *profils de performance moyens* sur les 50 essais montrant l'évolution de la qualité des solutions en fonction du temps.

Dans un premier temps, nous comparons DGVNS avec VNS/LDS+CP et ID-Walk [5], une des méthodes de recherche locale les plus efficaces sur les instances RLFAP (section 5.3.2). Dans un second temps, nous étudions l'influence de la largeur de décomposition sur notre approche (section 5.3.3). Ensuite, nous évaluons l'impact des différentes stratégies d'intensification et de diversification (section 5.3.4). Enfin, nous étudions les profils de rapport de performances de nos différentes stratégies.

Il est à noter que la comparaison des temps de calcul entre notre méthode et les méthodes de recherche complète serait peu pertinente. En effet, ces dernières vont à la fois trouver la ou les solutions optimales et prouver leur optimalité. Ces opérations peuvent nécessiter plusieurs jours sur ces instances [Sánchez *et al.*, 2009].

5.3.2 Apport de la décomposition arborescente

Afin de quantifier l'apport de la décomposition arborescente sur notre approche, nous comparons les résultats obtenus par DGVNS utilisant le schéma NeighborhoodChangeDGVNS-1 avec ceux obtenus par VNS/LDS+CP et ID-Walk sur les instances RLFAP, SPOT5 et tagSNP. Les résultats sur les instances GRAPH sont donnés tableau 5.5, section 5.3.3.

5.3.2.1 Instances RLFAP

Premièrement, DGVNS surclasse VNS/LDS+CP (cf. tableau 5.1). DGVNS atteint l'optimum sur 100% des essais pour la Scen06, 80% pour la Scen07 et 6% pour la Scen08. VNS/LDS+CP n'obtient l'optimum que dans de rares cas sur les deux premières instances et ne l'obtient pas pour la Scen08 : la meilleure solution trouvée a un coût de 357. Deuxièmement, DGVNS dépasse nettement

4. http ://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/SoftCSP
5. Pour ID-Walk, nous avons utilisé la librairie INCOP [Neveu et Trombettoni, 2003].

Instance	Méthode	Succ.	Temps	Moy.
#408	DGVNS	49/50	117	6,228
$n = 200$, $e = 2,232$	VNS/LDS+CP	26/50	149	6,228
$S^* = 6,228$	ID-Walk	**50/50**	**3**	**6,228**
#412	DGVNS	**36/50**	**84**	**32,381**
$n = 300$, $e = 4,348$	VNS/LDS+CP	32/50	130	32,381
$S^* = 32,381$	ID-Walk	10/50	2,102	32,381
#414	DGVNS	**38/50**	**554**	**38,478**
$n = 364$,$e = 10,108$	VNS/LDS+CP	12/50	434	38,481
$S^* = 38,478$	ID-Walk	0/50	-	38,481
#505	DGVNS	**50/50**	**63**	**21,253**
$n = 240$,$e = 2,242$	VNS/LDS+CP	41/50	143	21,253
$S^* = 21,253$	ID-Walk	**50/50**	**358**	**21,253**
#507	DGVNS	**33/50**	**71**	**27,390**
$n = 311$, $e = 5,732$	VNS/LDS+CP	11/50	232	27,391
$S^* = 27,390$	ID-Walk	7/50	1,862	27,391
#509	DGVNS	**40/50**	**265**	**36,446**
$n = 348$, $e = 8,624$	VNS/LDS+CP	12/50	598	36,448
$S^* = 36,446$	ID-Walk	0/50	-	36,450

TABLE 5.2 – Comparaison entre DGVNS, VNS/LDS+CP et ID-Walk sur les instances SPOT5.

ID-Walk[6], notamment sur les deux instances les plus difficiles, Scen07 et Scen08. Pour la Scen07 (resp. Scen08), DGVNS obtient des solutions ayant une *déviation moyenne* (pourcentage de déviation par rapport à l'optimum) de 0.55% (resp. 5%) à l'optimum, tandis qu'ID-Walk obtient des solutions avec une déviation moyenne de 8% (resp. 11%) à l'optimum.

5.3.2.2 Instances SPOT5

Cette tendance se confirme sur les instances SPOT5 (cf. tableau 5.2) pour lesquelles DGVNS devance VNS/LDS+CP, en taux de succès (avec un gain de 40% en moyenne) et en temps de calcul. En effet, DGVNS atteint l'optimum sur les 50 essais sur deux instances (#408 et #505). Pour les autres instances (excepté pour #507), le taux de succès est au moins de 70%. VNS/LDS+CP obtient un taux moyen de succès de 67% sur les instances #408 et #505, tandis que pour les autres instances (excepté pour #412), son taux de succès est d'au plus 24%.

Une fois de plus, DGVNS domine clairement ID-Walk sur toutes les instances (excepté pour #408) (cf. tableau 5.2), particulièrement sur les instances #414 et #509, où ID-Walk ne trouve pas l'optimum. Pour ces deux instances, les meilleures solutions trouvées ont un coût respectif de 38,479 et 36,447. Notons, toutefois, les bons résultats d'ID-Walk sur les deux instances #408 et #505.

5.3.2.3 Instances tagSNP

Le tableau 5.3 compare les résultats de DGVNS avec ceux de VNS/LDS+CP sur les instances tagSNP. Les résultats obtenus par ID-Walk sur ces instances sont de moins bonne qualité que ceux obtenus sur les instances RLFAP et SPOT5. Ils ne sont donc pas reportés. Pour les instances de taille moyenne, DGVNS donne de bien meilleurs résultats que VNS/LDS+CP. Il atteint l'optimum sur 100% des essais. VNS/LDS+CP obtient le même taux de succès sur trois instances (#6835, #15757 et #16706), mais DGVNS est 3.6 fois plus rapide. Pour les deux instances #3792 et #8956,

6. Les résultat pour les instances RLFAP sont tirés de [Neveu *et al.*, 2004]. Le nombre d'essais avec succès et les temps de calcul ne sont pas disponibles (NA, tableau 5.1), Nous ne donnons donc que le temps par essai, le coût moyen obtenu sur 10 essais et le meilleur coût (entre parenthèses). Pour les autres instances (SPOT5 et GRAPH), les résultats ont été obtenus avec le protocole expérimental décrit précédemment.

Instance	Méthode	Succ.	Temps CPU	Moy.
#3792, $n = 528$, $d = 59$, $e = 12,084$,$S^* = 6,359,805$	DGVNS	**50/50**	**954**	**6,359,805**
	VNS/LDS+CP	15/50	2,806	6,359,856
#4449, $n = 464$, $d = 64$, $e = 12,540$, $S^* = 5,094,256$	DGVNS	**50/50**	**665**	**5,094,256**
	VNS/LDS+CP	48/50	2,616	5,094,256
#6835, $n = 496$, $d = 90$, $e = 18,003$,$S^* = 4,571,108$	DGVNS	**50/50**	**2,409**	**4,571,108**
	VNS/LDS+CP	50/50	7,095	4,571,108
#8956, $n = 486$, $d = 106$, $e = 20,832$, $S^* = 6,660,308$	DGVNS	**50/50**	**4,911**	**6,660,308**
	VNS/LDS+CP	12/50	8,665	6,660,327
#9319, $n = 562$, $d = 58$, $e = 14,811$, $S^* = 6,477,229$	DGVNS	**50/50**	**788**	**6,477,229**
	VNS/LDS+CP	47/50	2,434	6,477,229
#15757, $n = 342$, $d = 47$, $e = 5,091$, $S^* = 2,278,611$	DGVNS	**50/50**	**60**	**2,278,611**
	VNS/LDS+CP	50/50	229	2,278,611
#16421, $n = 404$, $d = 75$, $e = 12,138$, $S^* = 3,436,849$	DGVNS	**50/50**	**2,673**	**3,436,849**
	VNS/LDS+CP	37/50	3,146	3,436,924
#16706, $n = 438$, $d = 30$, $e = 6,321$, $S^* = 2,632,310$	DGVNS	**50/50**	**153**	**2,632,310**
	VNS/LDS+CP	50/50	629	2,632,310
#6858, $n = 992$, $d = 260$, $e = 103,056$, $S^* = 20,162,249$	DGVNS	0/50	-	26,882,588 (26,879,268)
	VNS/LDS+CP	0/50	-	**26,815,733 (23,524,452)**
#9150, $n = 1,352$, $d = 121$, $e = 44,217$,$S^* = 43,301,891$	DGVNS	0/50	-	**44,754,916 (43,302,028)**
	VNS/LDS+CP	0/50	-	52,989,981 (51,677,673)
#10442, $n = 908$, $d = 76$, $e = 28,554$, $S^* = 21,591,913$	DGVNS	**50/50**	**4,552**	**21,591,913**
	VNS/LDS+CP	0/50	-	22,778,811 (22,490,938)
#14226, $n = 1,058$, $d = 95$, $e = 36,801$, $S^* = 25,665,437$	DGVNS	**46/50**	**7,606**	**25,688,751**
	VNS/LDS+CP	0/50	-	28,299,904 (26,830,579)
#17034, $n = 1,142$, $d = 123$, $e = 47,967$, $S^* = 38,318,224$	DGVNS	**41/50**	**8,900**	**38,563,232**
	VNS/LDS+CP	0/50	-	41,352,709 (39,850,974)

TABLE 5.3 – Comparaison entre DGVNS et VNS/LDS+CP sur les instances tagSNP.

VNS/LDS+CP obtient un faible taux de succès (respectivement 30% et 24%). Pour comparaison, DGVNS améliore très significativement ce taux de succès d'environ 70% (de 30% à 100%) sur l'instance #3792 et de 76% (de 24% à 100%) sur l'instance #8956. Pour les autres instances, VNS/LDS+CP reste moins compétitif que DGVNS en taux de succès et en temps de calcul.

Pour les instances de grande taille, DGVNS domine nettement VNS/LDS+CP (cf. tableau 5.3), particulièrement sur les instances #10442, #14226 et #17034, pour lesquelles VNS/LDS+CP n'arrive pas à atteindre l'optimum. Pour ces instances, la déviation à l'optimum des coûts des meilleures solutions trouvées est d'environ 4% en moyenne. Pour l'instance #6858, VNS/LDS+CP obtient la meilleure solution avec une déviation à l'optimum de 16% contre 33% pour DGVNS. Cependant, les résultats moyens des deux méthodes sur cette instance sont très proches. Enfin, pour l'instance #9150, DGVNS obtient de meilleurs résultats que VNS/LDS+CP en termes de coût moyen de la solution, avec une déviation moyenne à l'optimum de 3% contre 22% pour VNS/LDS+CP, et en termes de meilleure solution trouvée, avec un écart à l'optimum quasi-nul (19% pour VNS/LDS+CP).

5.3.2.4 Profils de performance

Les figures 5.3, 5.4, 5.5 et 5.6 comparent les profils de performance de DGVNS et VNS/LDS+CP. Pour les instances RLFAP et SPOT5 (figures 5.3, 5.4), DGVNS donne de meilleurs résultats que VNS/LDS+CP (excepté pour l'instance #505). Du point de vue des profils de performance, on observe deux phénomènes importants : la courbe associée au comportement de DGVNS présente une plus forte pente et la décélération de la courbe associée au comportement de VNS/LDS+CP est beaucoup plus forte que celle de DGVNS.

Pour les instances tagSNP (figures 5.5 et 5.6), on observe le même comportement que précédemment, particulièrement sur les instances de grande taille, où la décélération de la courbe

de VNS/LDS+CP est encore plus amplifiée que celle de DGVNS. De plus, la vitesse d'amélioration de la qualité des solutions par rapport au coût initial est beaucoup plus grande pour DGVNS. De son côté, VNS/LDS+CP a besoin de beaucoup plus de temps de calcul pour obtenir des solutions de qualité comparable. Ceci confirme l'importance d'exploiter la décomposition arborescente pour guider VNS.

5.3.2.5 Bilan sur l'apport de la décomposition arborescente

Ces expérimentations montrent clairement l'efficacité et la pertinence de DGVNS comparé à VNS/LDS+CP et ID-Walk sur des problèmes structurés tels que RLFAP et SPOT5. Sur les instances tagSNP, DGVNS surclasse clairement VNS/LDS+CP, particulièrement sur les instances de grande taille, où VNS/LDS+CP est incapable d'atteindre l'optimum. Enfin, notre approche permet d'améliorer très significativement le taux de succès, les temps de calcul et les profils des performances .

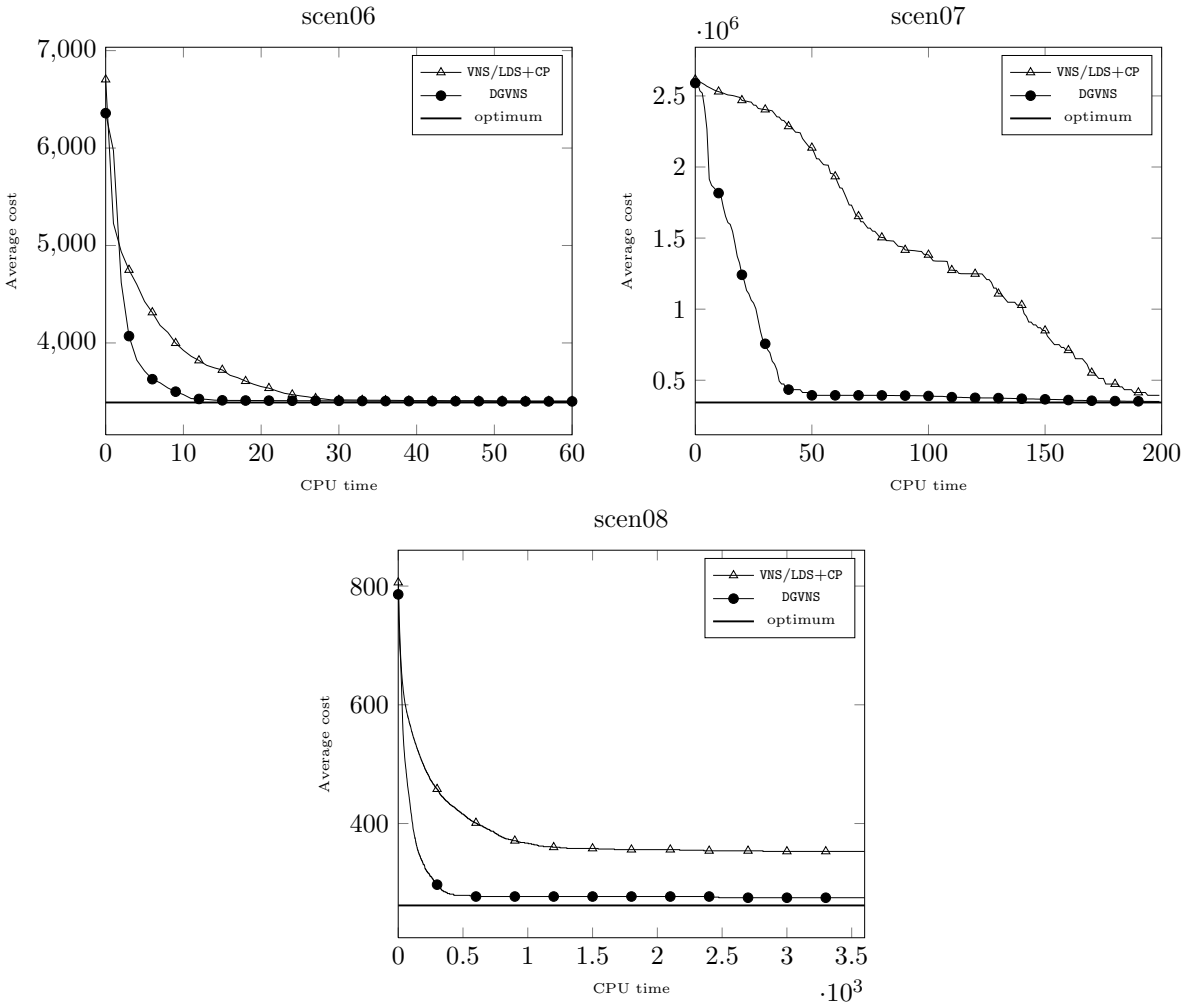


FIGURE 5.3 – Comparaison des profils de performance de VNS/LDS+CP et DGVNS sur les instances RLFAP.

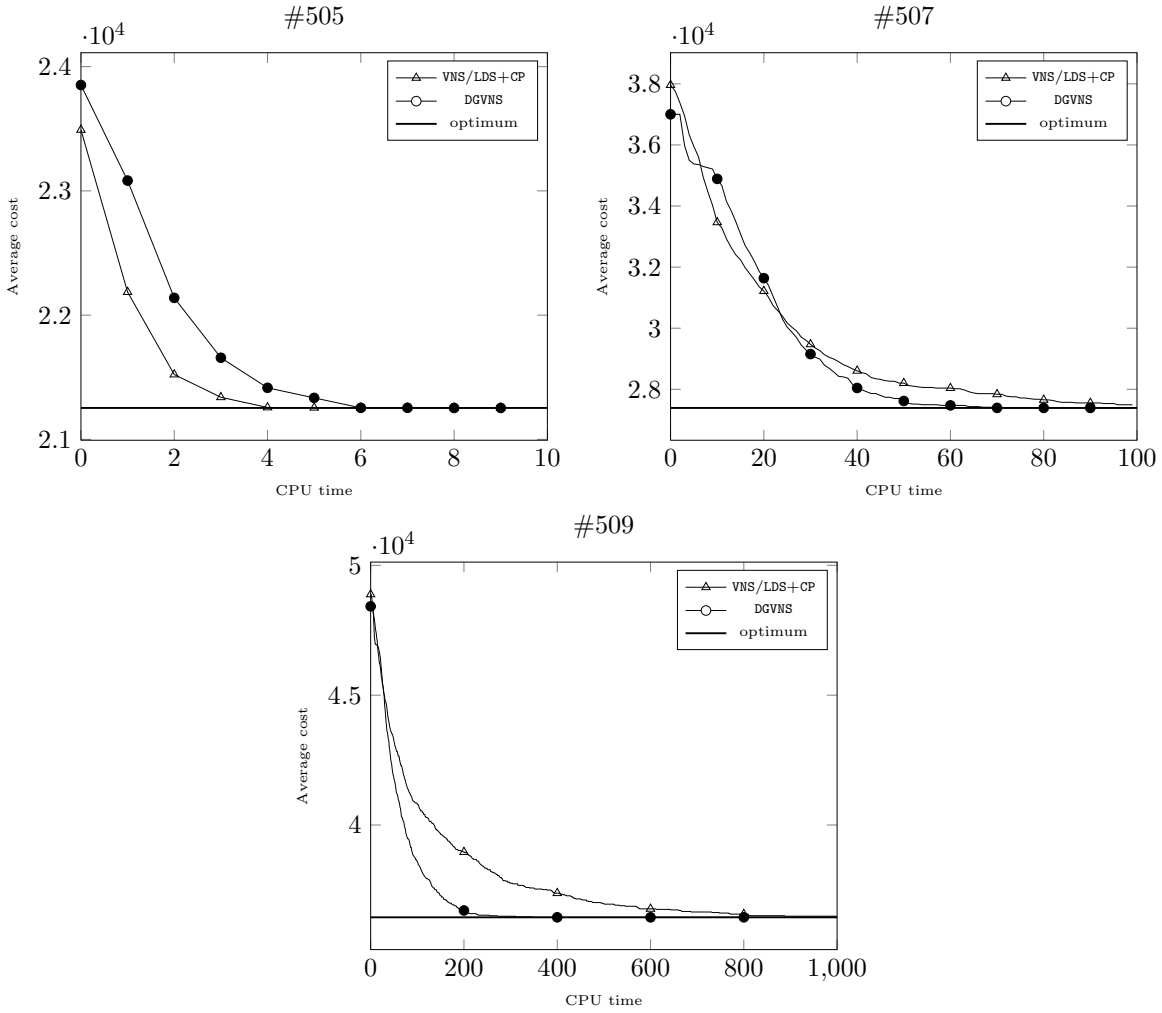


FIGURE 5.4 – Comparaison des profils de performance de VNS/LDS+CP et DGVNS sur les instances SPOT5.

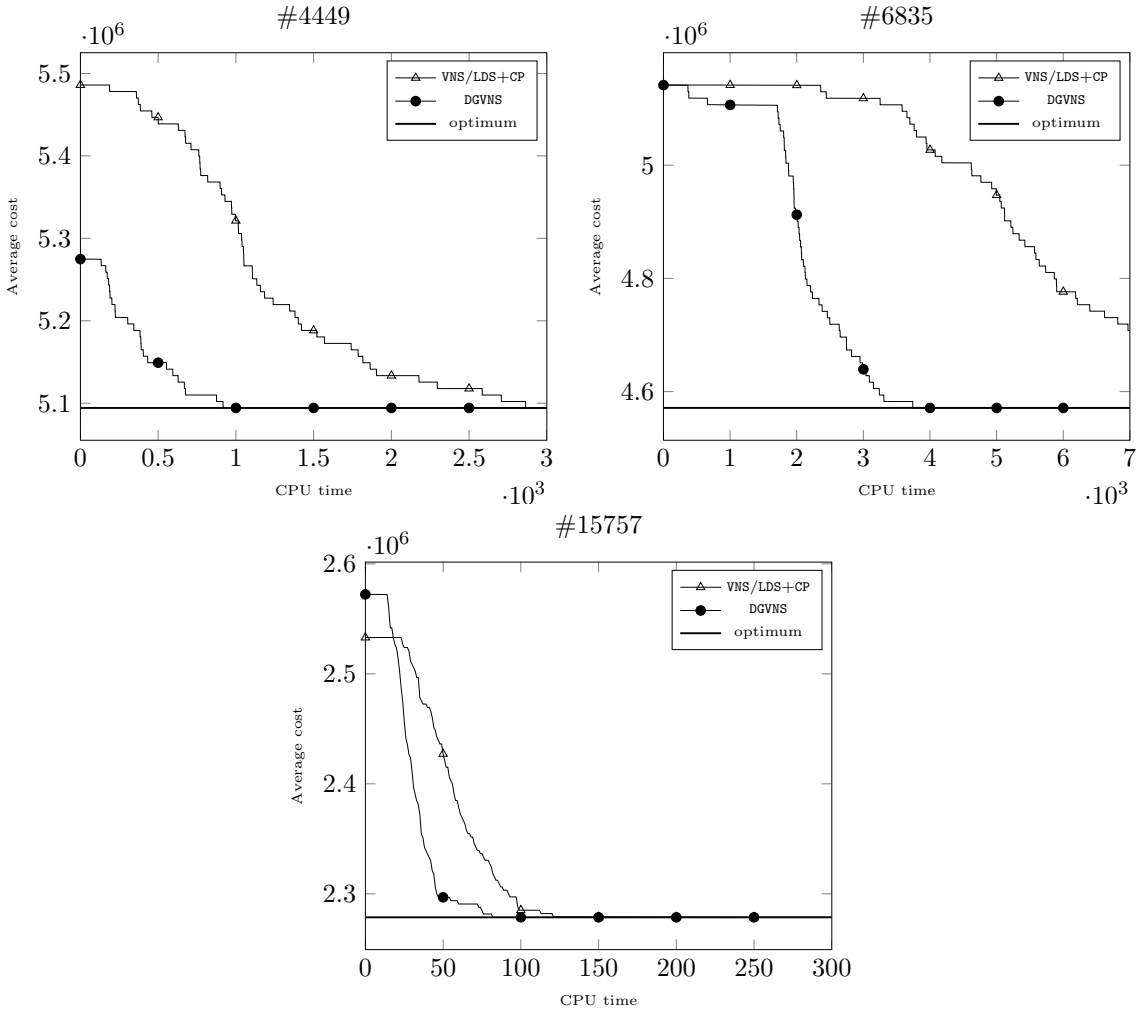


FIGURE 5.5 – Comparaison des profils de performance de `VNS/LDS+CP` et `DGVNS` sur les instances `tagSNP` de taille moyenne.

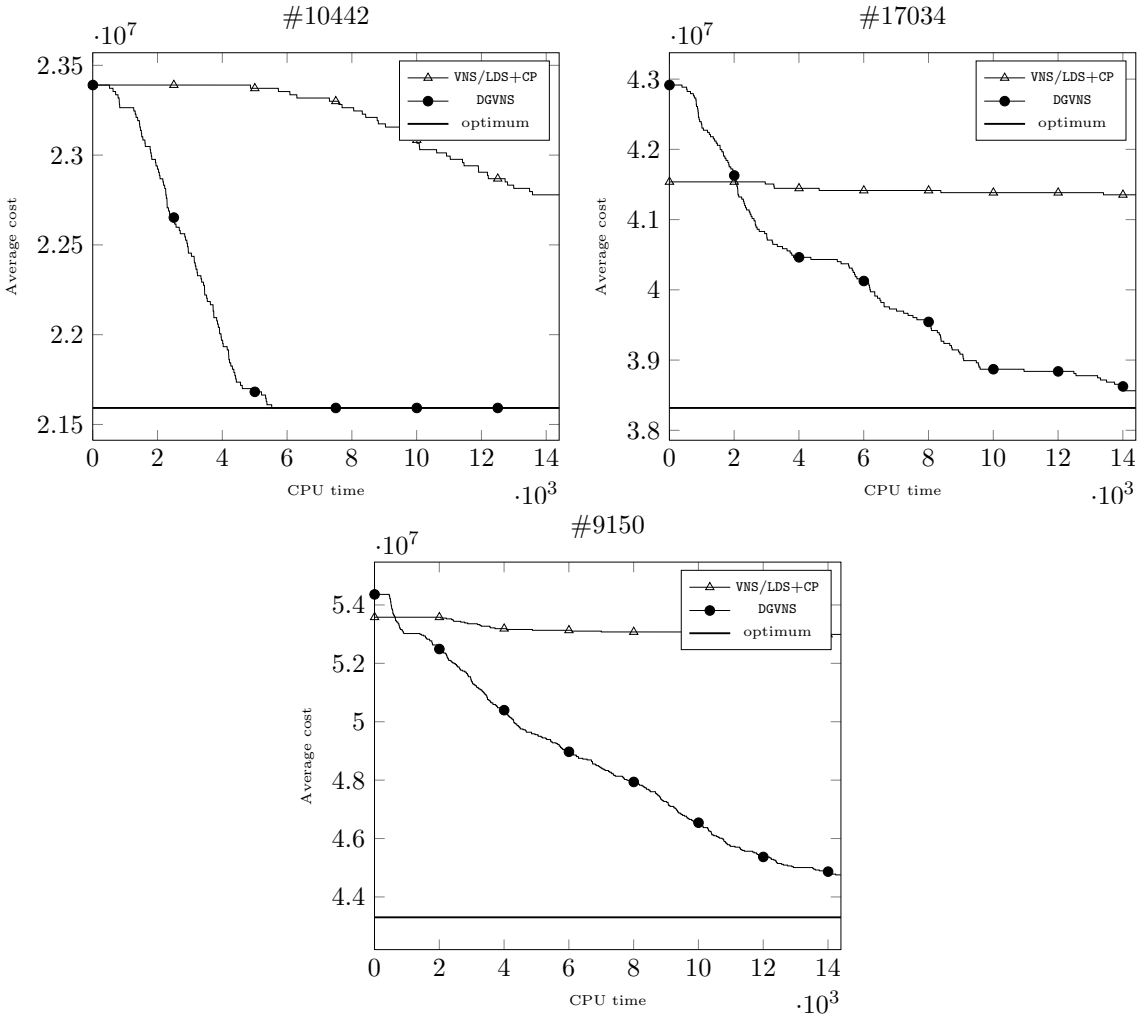


FIGURE 5.6 – Comparaison des profils de performance de VNS/LDS+CP et DGVNS sur les instances tagSNP de grande taille.

Instance	n	w^-	s_{max}	mc	$\frac{w^-}{n}$	$\frac{w^-}{mc}$	$\mid \mathcal{C}_T \mid$
Scen06	100	11	8	10	0.11	1.1	55
Scen07	200	22	22	10	0.11	2.2	110
Scen08	458	26	26	10	0.05	2.6	259
Graph05	100	32	32	8	0.32	4	58
Graph06	200	62	61	8	0.31	7.75	123
Graph11	340	117	116	7	0.34	19.5	191
Graph13	458	155	153	6	0.34	25.8	262
#3792	528	94	78	40	0.178	2.35	70
#4449	464	100	100	52	0.215	1.923	56
#6835	496	109	104	59	0.219	1.847	56
#8956	486	184	180	83	0.378	2.216	54
#9319	562	89	86	46	0.158	1.934	62
#15757	342	66	60	29	0.192	2.275	45
#16421	404	112	110	58	0.277	1.931	35
#16706	438	48	30	26	0.109	1.6	49
#10442	908	133	114	95	0.146	1.4	104
#14226	1,058	158	150	81	0.149	1.95	94
#17034	1,142	188	182	84	0.164	2.238	139
#6858	992	350	260	131	0.352	2.671	105
#9150	1,352	195	168	86	0.144	1.16	178

TABLE 5.4 – Plus petite largeur de décomposition, taille et ratios pour les instances `RLFAP`, `GRAPH` et `tagSNP`.

5.3.3 Impact de la largeur de décomposition

Comme nous l'avons précisé dans le chapitre 2, de nombreux problèmes réels présentent une structure particulière, notamment des sous-structures peu connectées entre elles (cf. figure 5.1). Ces structures reflètent des propriétés topologiques induites par la décomposition arborescente et se caractérisent par trois mesures : la largeur de la décomposition, la taille des séparateurs, et le degré des clusters. La largeur de décomposition donne une bonne indication sur la taille des sous-problèmes, tandis que les deux autres mesures permettent d'évaluer la connectivité entre clusters. En effet, plus ces valeurs sont faibles, moins les clusters sont connectés. Comme il est indiqué dans [de Givry, 2011], calculer une décomposition arborescente optimisant ces trois mesures est un problème ouvert. Nous avons sélectionné l'heuristique `MCS` [Tarjan et Yannakakis, 1984b] car celle-ci présente un bon compromis entre la qualité de la décomposition et le temps nécessaire pour son calcul (cf. chapitre 2 et [Jégou *et al.*, 2005]). Dans cette section, nous étudions l'impact de la largeur de décomposition.

5.3.3.1 Critères pour caractériser une décomposition arborescente

Pour une instance à résoudre donnée, on notera :
– mc la taille de la plus grande clique de son graphe de contraintes,
– w^- la largeur de décomposition obtenue par `MCS` [7].
Pour étudier l'impact de la largeur de décomposition sur `DGVNS`, nous définissons deux critères :

1. la décomposabilité d'un problème ($\frac{w^-}{n}$) définie par le ratio entre la largeur de décomposition et le nombre de variables du problème. Plus ce ratio est faible, plus le problème se décompose en clusters de petite taille.
2. la précision de la décomposition ($\frac{w^-}{mc}$) définie par le ratio entre la largeur de décomposition et la taille de la plus grande clique dans le graphe de contraintes du problème.

7. Cette décomposition n'est pas nécessairement de largeur minimale.

Instance	Méthode	Succ.	Temps	Moy.
Graph05	DGVNS	**50/50**	**10**	**221**
$n = 100$, $e = 1,034$	VNS/LDS+CP	50/50	17	221
$s* = 221$	ID-Walk	0/50	-	10,584 (2,391)
Graph06	DGVNS	50/50	367	4,123
$n = 200$,$e = 1,970$	VNS/LDS+CP	**50/50**	**218**	**4,123**
$s* = 4,123$	ID-Walk	0/50	-	18,949 (13,001)
Graph11	DGVNS	8/50	3,046	4,234
$n = 340$,$e = 3,417$	VNS/LDS+CP	**44/50**	**2,403**	**3,090**
$s* = 3,080$	ID-Walk	0/50	-	41,604 (27,894)
Graph13	DGVNS	0/50	-	22,489 (18,639)
$n = 458$,$e = 4,915$	VNS/LDS+CP	**3/50**	**3,477**	**14,522**
$s* = 10,110$	ID-Walk	0/50	-	58,590 (47,201)

TABLE 5.5 – Comparaison entre DGVNS, VNS/LDS+CP et ID-Walk sur les instances GRAPH.

La taille de la plus grande clique (mc) moins un donne un minorant de la largeur de décomposition. Elle permet d'évaluer la qualité d'une décomposition. En effet, plus le ratio $\frac{w^-}{mc}$ est proche de 1, plus la décomposition est proche de la structure du graphe de contraintes et peut ainsi nous renseigner efficacement sur celle-ci, nous permettant de construire des voisinages plus pertinents. Le tableau 5.4 indique, pour chaque instance RLFAP, tagSNP et GRAPH, la taille du problème, la valeur de w^-, la taille du plus grand séparateur s_{max}, la taille de la plus grande clique dans le graphe de contraintes (mc), la taille du problème (n), les ratios ($\frac{w^-}{n}$) et ($\frac{w^-}{mc}$) et le nombre de clusters. Les résultats pour les instances SPOT5 sont très similaires à ceux des instances RLFAP, ils ne sont donc pas reportés ici.

5.3.3.2 Analyse des instances

(i) Pour les instances RLFAP (cf. tableau 5.4), la plus petite largeur (w^-) augmente avec la taille du problème (n), tandis que le ratio ($\frac{w^-}{n}$) diminue. Cela signifie que pour les plus grandes instances, les clusters sont plus dispersés. De plus, ces problèmes présentent un ratio $\frac{w^-}{n}$ très faible (il varie entre 5% et 11%). Ce n'est pas le cas pour les instances GRAPH (particulièrement les instances Graph11 et Graph13), pour lesquelles les valeurs de w^- et des ratios $\frac{w^-}{n}$ et $\frac{w^-}{mc}$ sont élevées comparées aux valeurs associées aux instances RLFAP. Le tableau 5.4 montre également de grands écarts (> 100) entre les valeurs de w^- et celles de mc, tandis que sur les instances RLFAP, ces écarts sont plutôt faibles. Ces différences entre les instances RLFAP et GRAPH peuvent s'expliquer par l'origine de ces dernières : les RLFAP sont issues de problèmes réels alors que les instances GRAPH ont été générées aléatoirement.

(ii) Pour les instances tagSNP (cf. tableau 5.4), on observe une valeur très faible du ratio $\frac{w^-}{n}$ (≤ 0.22) sur les instances de taille moyenne, excepté pour les instances #8956 et #16421. De plus, l'écart entre la taille de la plus grande clique (mc) et la largeur de décomposition (w^-) est relativement petit. Ceci explique probablement les bonnes performances de DGVNS sur ces instances, qui semblent être adaptées à l'exploitation de la décomposition arborescente. Sur les instances de plus grande taille, cette tendance se confirme, excepté pour l'instance #6858, où VNS/LDS+CP obtient en moyenne de meilleures solutions que DGVNS. Les contre performances de DGVNS sur cette instance peuvent s'expliquer par une valeur de $\frac{w^-}{n}$ élevée et un écart relativement important entre mc et w^-. Il est difficile pour MCS de décomposer ce problème en clusters de petite taille faiblement connectés. Pour l'instance #9150, bien que l'écart entre mc et w^- soit faible et la valeur du ratio $\frac{w^-}{n}$ peu élevée, DGVNS n'arrive pas à atteindre l'optimum. Cette instance présente en effet une large sous-partie du graphe de contraintes qui est très dense. Une

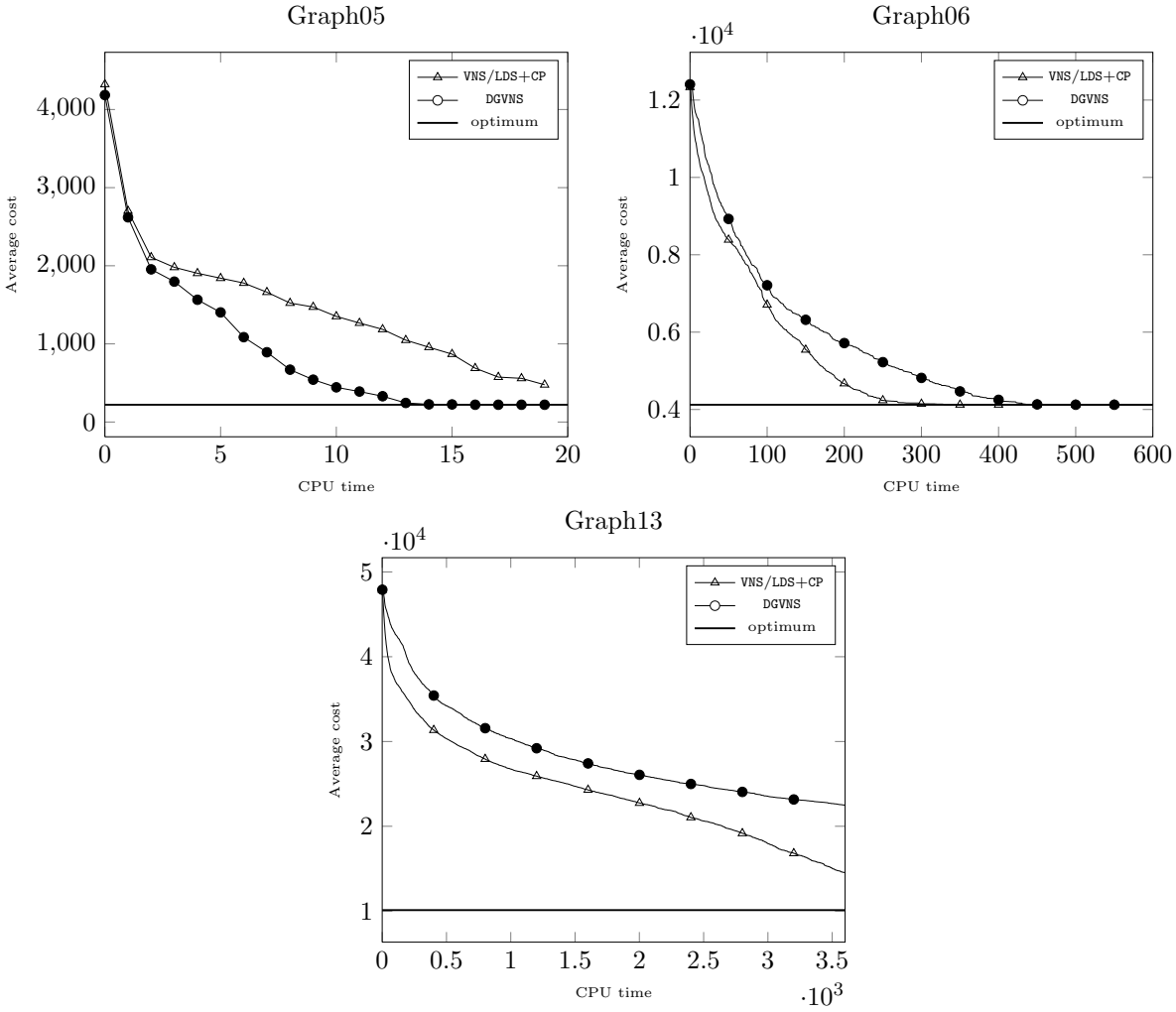


FIGURE 5.7 – Comparaison des profils de performance de VNS/LDS+CP et DGVNS sur les instances GRAPH.

telle structure, qui est proche d'une structure aléatoire, ne profite pas à DGVNS, car un grand nombre de clusters se recouvrent fortement.

(iii) Pour les instances GRAPH (cf. tableau 5.4), les résultats sont plus mitigés. Les performances de DGVNS sont bonnes sur les instances Graph06 et Graph05, que ce soit en taux de succès, en qualité de solutions (cf. tableau 5.5) et en profils de performance (cf. figure 5.7). Cependant, sur les instances Graph11 et Graph13, VNS/LDS+CP présente de meilleurs résultats. Cela est dû au fait que les clusters obtenus pour ces instances sont de grande taille et fortement connectés. Ainsi, l'impact de notre diversification devient beaucoup moins important car la plupart des clusters ont très peu de variables propres et tendent à être très similaires (cf. remarque 4). Ces résultats montrent les limites de MCS à révéler une structure pertinente sur ces instances (i.e. présence de clusters faiblement connectés et de petite taille).

Remarque 4. *Sur la plupart des instances du tableau 5.4, on peut noter que la valeur de s_{max} est très proche de celle de w^- voire égale dans certains cas. Ceci signifie que la plupart des clusters ont peu de variables propres. Nous aborderons cette problématique dans le chapitre 6.*

Instance	Méthode	Succ.	Temps	Moy.
Scen06	DGVNS-3	50/50	521	3,389
S^* =3,389	DGVNS-2	50/50	146	3,389
	DGVNS-1	**50/50**	**112**	**3,389**
Scen07	DGVNS-3	4/50	2,752	347,583
S^* =343,592	DGVNS-2	**46/50**	**901**	**344,012**
	DGVNS-1	40/50	317	345,614 (343,592)
Scen08	DGVNS-3	0/50	-	519 (500)
S^* =262	DGVNS-2	**5/50**	**595**	**277**
	DGVNS-1	3/50	1,811	275

TABLE 5.6 – Comparaison des différents DGVNS-i ($i = 1, 2, 3$) sur les instances RLFAP.

5.3.3.3 Bilan sur l'impact de la largeur de décomposition

La largeur de décomposition arborescente a un fort impact sur les performances de DGVNS. Nos expérimentations montrent clairement l'efficacité de notre méthode sur les problèmes présentant un faible ratio $\frac{w^-}{n}$ et un faible écart entre w^- et mc, c'est-à-dire des problèmes se décomposant en clusters de tailles raisonnables. Pour ces problèmes, MCS fournit des décompositions pertinentes et très proches de la structure du problème, permettant ainsi une meilleure diversification.

5.3.4 Comparaison des trois stratégies d'intensification et de diversification

Dans cette section, nous comparons nos différentes stratégies sur différentes instances des quatre problèmes considérés.

5.3.4.1 Instances RLFAP

L'impact de la stratégie de changement de voisinage est significatif, particulièrement sur les instances Scen07 et Scen08, pour lesquelles DGVNS-2 procure de fortes améliorations en comparaison avec les résultats obtenus par DGVNS-1 (cf. tableau 5.6). Pour la Scen07, DGVNS-2 améliore de 12% le taux de succès (de 80% à 92%) et réduit la déviation moyenne à l'optimum de (0.55% à 0.12%). Pour la Scen08, le gain est de 4% (de 6% à 10%) et DGVNS-2 est 3 fois plus rapide que DGVNS-1. Notons toutefois, que sur cette instance, DGVNS-1 obtient de meilleurs résultats en moyenne, avec une déviation moyenne à l'optimum de 0.5% contre 0.57% pour DGVNS-2. Pour la Scen06, les résultats de DGVNS-1 et DGVNS-2 restent très comparables, avec toutefois un léger avantage en faveur de DGVNS-1, qui est plus rapide.

L'étude des profils de performance (cf. figure 5.8) confirme les bonnes propriétés de DGVNS-1 du point de vue des profils de performance. En effet, sur les trois instances, l'amélioration de la qualité de la solution est plus rapide pour DGVNS-1 par rapport à DGVNS-2. Enfin, notons les mauvaises performances de DGVNS-3 comparé aux deux autres méthodes, particulièrement sur la Scen08, où DGVNS-3 n'atteint jamais l'optimum ; la meilleure solution trouvée a un coût 500. Ceci confirme l'importance de notre stratégie de diversification basée sur une couverture plus large des clusters du problème.

5.3.4.2 Instances SPOT5

Cette tendance se confirme sur les instances SPOT5 (cf. tableau 5.7), pour lesquelles DGVNS-2 se montre particulièrement efficace comparé à DGVNS-1. Sur les quatre instances de grande taille (#412, #414, #507 et #509), DGVNS-2 améliore le taux de succès de 19% en moyenne. En effet, sur l'instance #412 (resp. #414), DGVNS-2 permet d'augmenter le taux de succès de 24% (resp.

Instance	Méthode	Succ.	Temps	Moy.
#408, S^* =6,228	DGVNS-3	50/50	1,697	6,228
	DGVNS-2	**50/50**	**81**	**6,228**
	DGVNS-1	49/50	117	6,228
#412, S^* =32,381	DGVNS-3	0/50	-	32,384 (32,383)
	DGVNS-2	**48/50**	**484**	**32,381**
	DGVNS-1	36/50	84	32,381
#414, S^* =38,478	DGVNS-3	0/50	-	38,486 (38,482)
	DGVNS-2	**45/50**	**670**	**38,478**
	DGVNS-1	38/50	554	38,478
#505, S^* =21,253	DGVNS-3	36/50	3,094	21,253
	DGVNS-2	50/50	90	21,253
	DGVNS-1	**50/50**	**63**	**21,253**
#507, S^* =27,390	DGVNS-3	0/50	-	27,396 (27,393)
	DGVNS-2	**45/50**	**463**	**27,390**
	DGVNS-1	33/50	71	27,390
#509, S^* =36,446	DGVNS-3	0/50	-	36,454 (36,450)
	DGVNS-2	**47/50**	**509**	**36,446**
	DGVNS-1	40/50	265	36,446

TABLE 5.7 – Comparaison des différents DGVNS-i ($i = 1, 2, 3$) sur les instances SPOT5.

14%) par rapport à DGVNS-1. On peut cependant noter que DGVNS-1 est plus rapide en moyenne que DGVNS-2. Pour les autres instances (#408 et #505), les deux méthodes ont des performances similaires. Toutefois, sur l'instance #408, DGVNS-2 est 1.4 fois plus rapide que DGVNS-1, tandis que sur l'instance #505, c'est DGVNS-1 qui est très légèrement plus rapide. Enfin, les résultats obtenus par DGVNS-3 sont de piètre qualité. Elle n'obtient l'optimum que sur 2 instances parmi 6. Du point de vue des profils de performance (cf. figure 5.9), DGVNS-1 posséde le meilleur comportement sur les instances #505 et #509. Cependant, pour l'instance #507 et sur les 40 premières secondes, c'est DGVNS-2 qui présente l'amélioration de la qualité de solution la plus rapide avant d'être dépassée par DGVNS-1.

5.3.4.3 Instances GRAPH

Sur les instances faciles (Graph05 et Graph06, tableau 5.8), les trois méthodes obtiennent l'optimum sur 100% des essais. Toutefois, pour l'instance Graph05, DGVNS-3 a besoin en moyenne de 2 fois plus de temps pour obtenir des solutions de qualité comparable. Pour l'instance Graph06, DGVNS-1 obtient les meilleurs temps de calcul, avec 367 s. en moyenne contre 413 s. et 254 s. pour DGVNS-2 et DGVNS-3 respectivement. Sur les instances difficiles (Graph11 et Graph13), DGVNS-2 surclasse clairement les deux autres méthodes. Pour l'instance Graph11, DGVNS-2 multiplie par 3 le taux de succès de DGVNS-1 (de 16% à 46%) et réduit la déviation moyenne à l'optimum de 25% (de 37% à 12%). Pour l'instance Graph13, bien qu'aucune méthode n'atteint l'optimum, DGVNS-2 obtient des solutions ayant une déviation moyenne à l'optimum de 115% contre 122% pour DGVNS-1. De plus, DGVNS-2 trouve une meilleure solution de coût 18,323. Pour comparaison, la meilleure solution trouvée par DGVNS-1 est de 18,639. Enfin, les profils de performance des deux méthodes restent assez proches.

5.3.4.4 Instances tagSNP

Sur les instances tagSNP (cf. tableau 5.9), les résultats sont en faveur de DGVNS-1. Pour les instances de taille moyenne (#6835 et #8956), DGVNS-1 surclasse DGVNS-2 en taux de succès. Pour les autres instances (excepté pour #16421 et #16706), les deux méthodes atteignent l'optimum à chaque essai, mais DGVNS-1 est en moyenne 2 fois plus rapide. Pour les instances #16421 et #16706, c'est DGVNS-2 qui obtient les meilleurs temps de calcul.

Instance	Méthode	Succ.	Temps	Moy.
Graph05, $S^* = 221$	DGVNS-3	50/50	23	221
	DGVNS-2	**50/50**	**10**	**221**
	DGVNS-1	**50/50**	**10**	**221**
Graph06, $S^* = 4,123$	DGVNS-3	50/50	654	4,123
	DGVNS-2	**50/50**	413	4,123
	DGVNS-1	**50/50**	**367**	**4,123**
Graph11, S^* =3,080	DGVNS-3	1/50	3,507	6,637
	DGVNS-2	**23/50**	**3,031**	**3,480**
	DGVNS-1	8/50	3,046	3,090
Graph13, S^* =10,110	DGVNS-3	0/50	-	31,180 (28,585)
	DGVNS-2	0/50	-	**21,796 (18,323)**
	DGVNS-1	0/50	-	22,489 (18,639)

TABLE 5.8 – Comparaison des différents DGVNS-i ($i = 1, 2, 3$) sur les instances GRAPH.

Pour les instances de grande taille (excepté pour #14226, où DGVNS-2 obtient un meilleur taux de succès), les résultats sont en faveur de DGVNS-1 **qui obtient les meilleurs résultats sur** 4 **instances parmi** 5. Les profils de performance confirment l'avantage de DGVNS-1, qui devance très largement DGVNS-2.

Instance	Méthode	Succ.	Temps	Moy.
#3792	DGVNS-3	14/50	4,509	6,360,029
$S^* = 6,359,805$	DGVNS-2	50/50	1,564	6,359,805
	DGVNS-1	**50/50**	**954**	**6,359,805**
#4449	DGVNS-3	9/50	3,371	5,094,261
$S^* = 5,094,256$	DGVNS-2	50/50	1,186	5,094,256
	DGVNS-1	**50/50**	**665**	**5,094,256**
#6835	DGVNS-3	3/50	6,746	4,571,344
$S^* = 4,571,108$	DGVNS-2	34/50	3,568	4,730,484
	DGVNS-1	**50/50**	**2,409**	**4,571,108**
#8956	DGVNS-3	0/50	-	6,966,493 (6,660,336)
$S^* = 6,660,308$	DGVNS-2	34/50	5,138	6,660,383
	DGVNS-1	**50/50**	**4,911**	**6,660,308**
#9319	DGVNS-3	0/50	-	6,477,395 (6,477,242)
$S^* = 6,477,229$	DGVNS-2	50/50	1,248	6,477,229
	DGVNS-1	**50/50**	**788**	**6,477,229**
#15757	DGVNS-3	50/50	2,041	2,278,611
$S^* = 2,278,611$	DGVNS-2	50/50	127	2,278,611
	DGVNS-1	**50/50**	**60**	**2,278,611**
#16421	DGVNS-3	8/50	5,552	3,437,125
$S^* = 3,436,849$	DGVNS-2	**50/50**	**2,001**	**3,436,849**
	DGVNS-1	50/50	2,673	3,436,849
#16706	DGVNS-3	0/50	-	2,632,319 (2,632,319)
$S^* = 2,632,310$	DGVNS-2	**50/50**	**127**	**2,632,310**
	DGVNS-1	49/50	153	2,632,310
#6858	DGVNS-3	0/50	-	26,882,903 (26,882,903)
$S^* = 20,162,249$	DGVNS-2	0/50	-	26,882,824 (26,880,099)
	DGVNS-1	0/50	-	**26,882,588 (26,879,268)**
#10442	DGVNS-3	0/50	-	22,491,631 (22,491,496)
$S^* = 21,591,913$	DGVNS-2	48/50	8,257	21,591,915
	DGVNS-1	**50/50**	**4,552**	**21,591,913**
#14226	DGVNS-3	0/50	-	29,256,033 (27,996,275)
$S^* = 25,665,437$	DGVNS-2	**50/50**	**8,237**	**25,665,437**
	DGVNS-1	46/50	7,606	25,688,751
#17034	DGVNS-3	0/50	-	41,016,388 (39,852,093)
$S^* = 38,318,224$	DGVNS-2	35/50	10,288	38,777,581
	DGVNS-1	**41/50**	**8,900**	**38,563,232**
#9150	DGVNS-3	0/50	-	51,510,783 (50,281,105)
$S^* = 43,301,891$	DGVNS-2	0/50	-	46,123,257 (44,697,387)
	DGVNS-1	0/50	-	**44,754,916 (43,302,028)**

TABLE 5.9 – Comparaison des différents DGVNS-i ($i = 1, 2, 3$) sur les instances tagSNP.

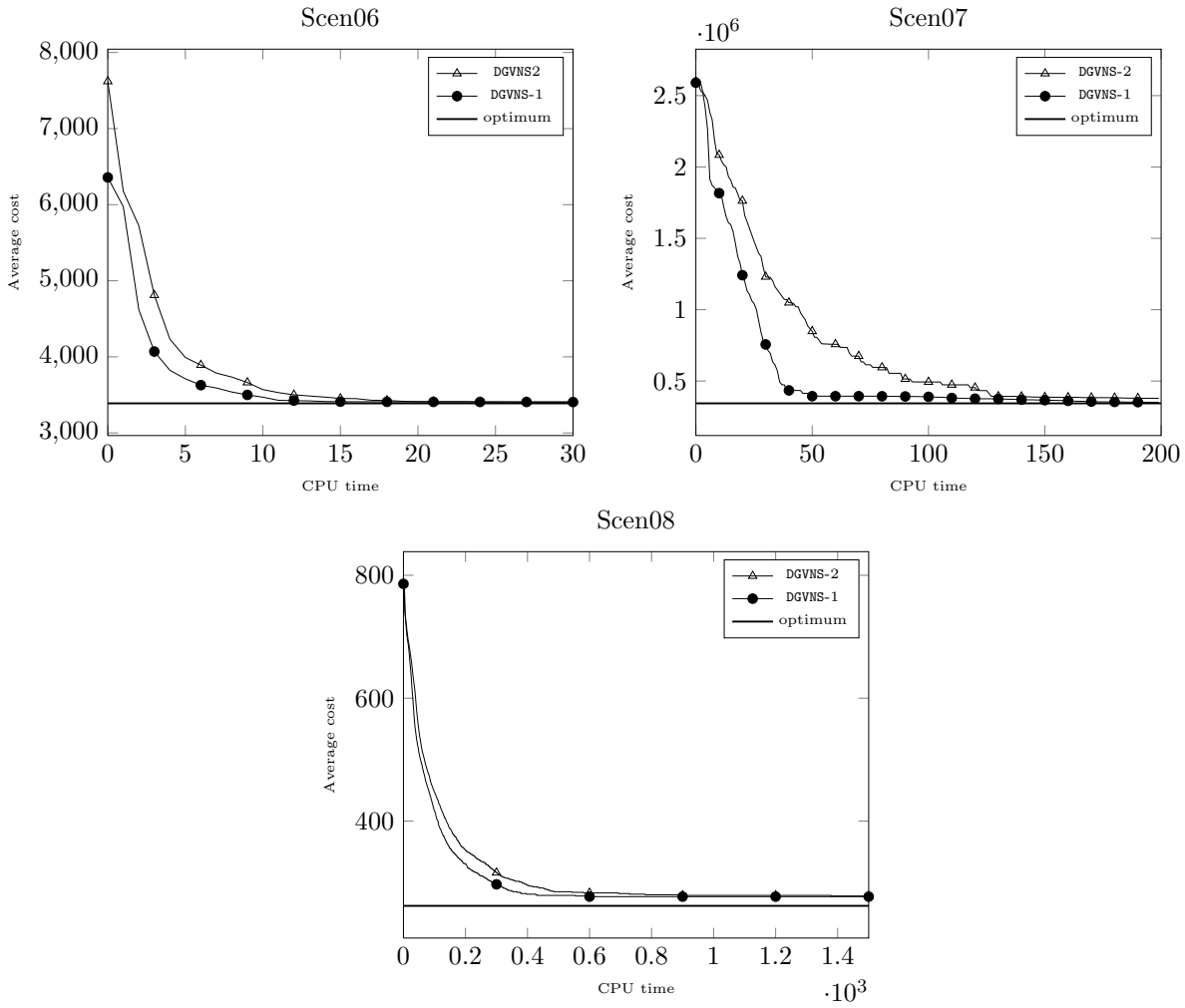


FIGURE 5.8 – Comparaison des profils de performance de DGVNS-1 et DGVNS-2 sur les instances RLFAP.

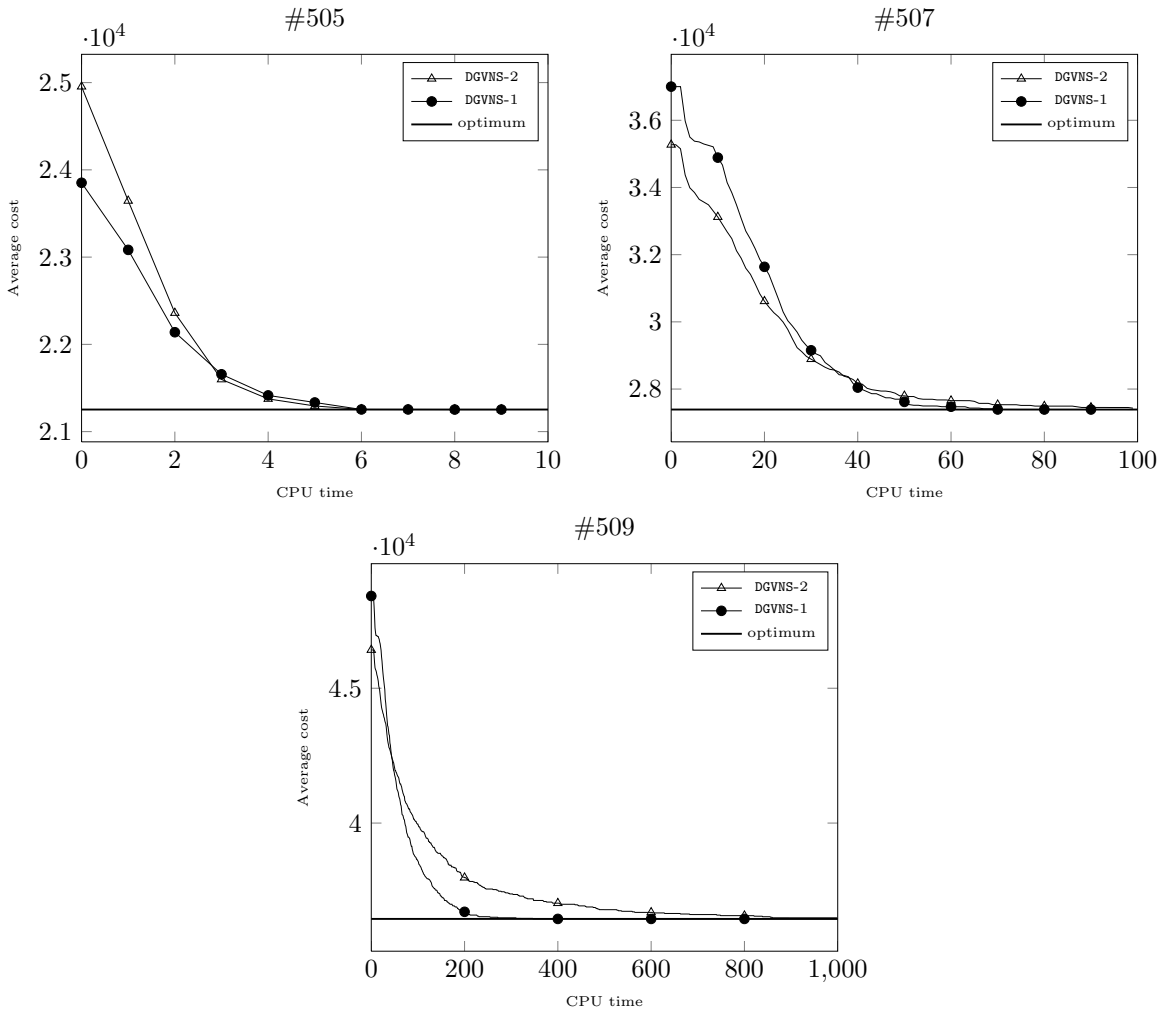


FIGURE 5.9 – Comparaison des profils de performance de `DGVNS-1` et `DGVNS-2` sur les instances `SPOT5`.

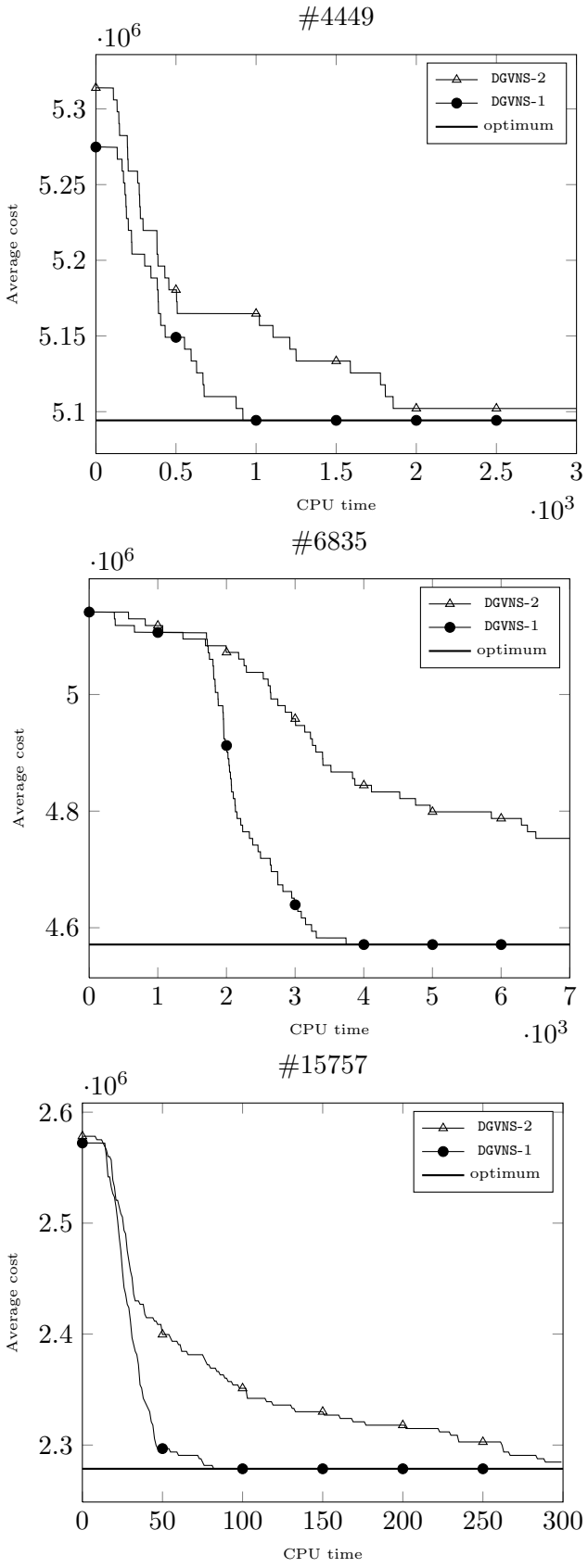


FIGURE 5.10 – Comparaison des profils de performance de DGVNS-1 et DGVNS-2 sur les instances tagSNP de taille moyenne.

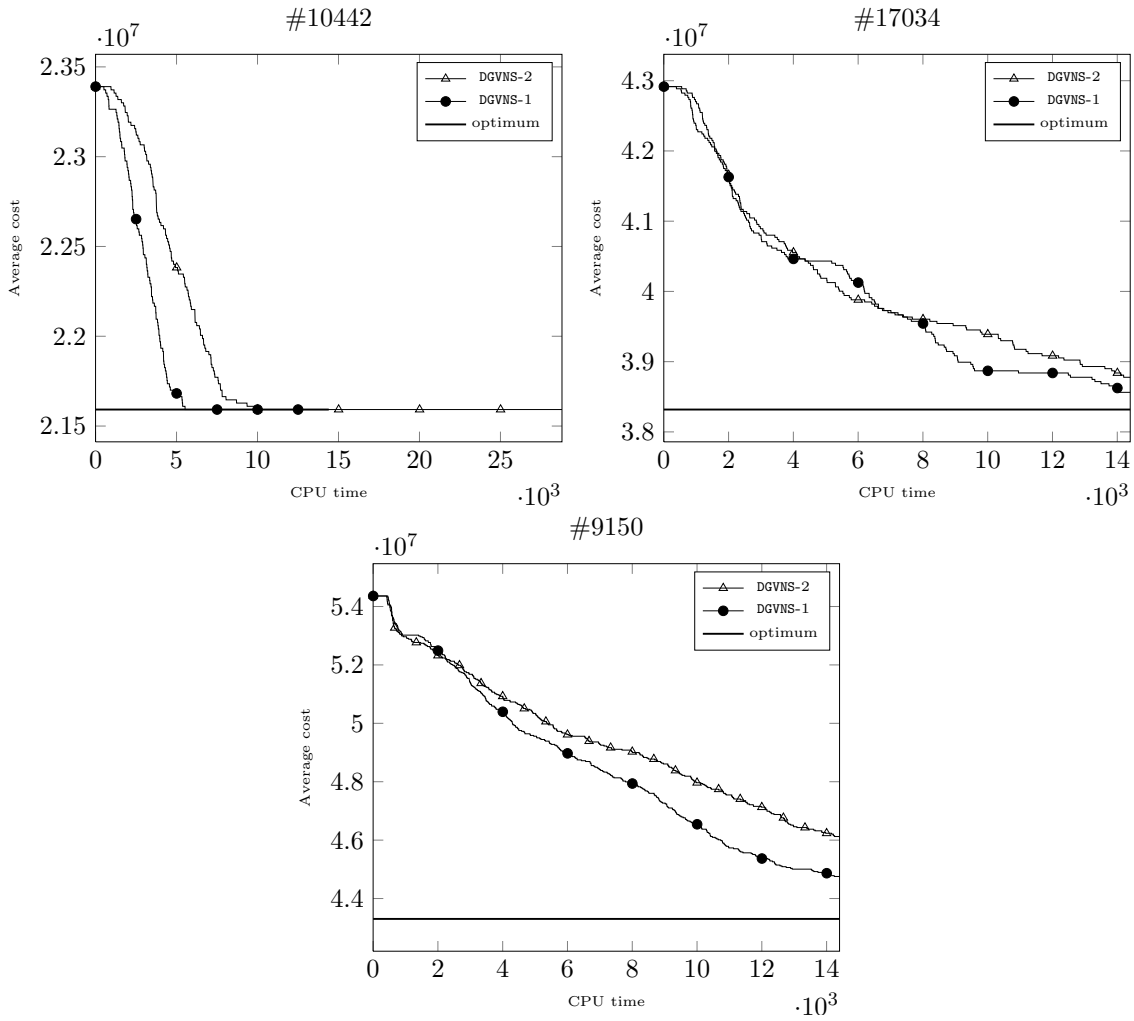


FIGURE 5.11 – Comparaison des profils de performance de `DGVNS-1` et `DGVNS-2` sur les instances `tagSNP` de grande taille.

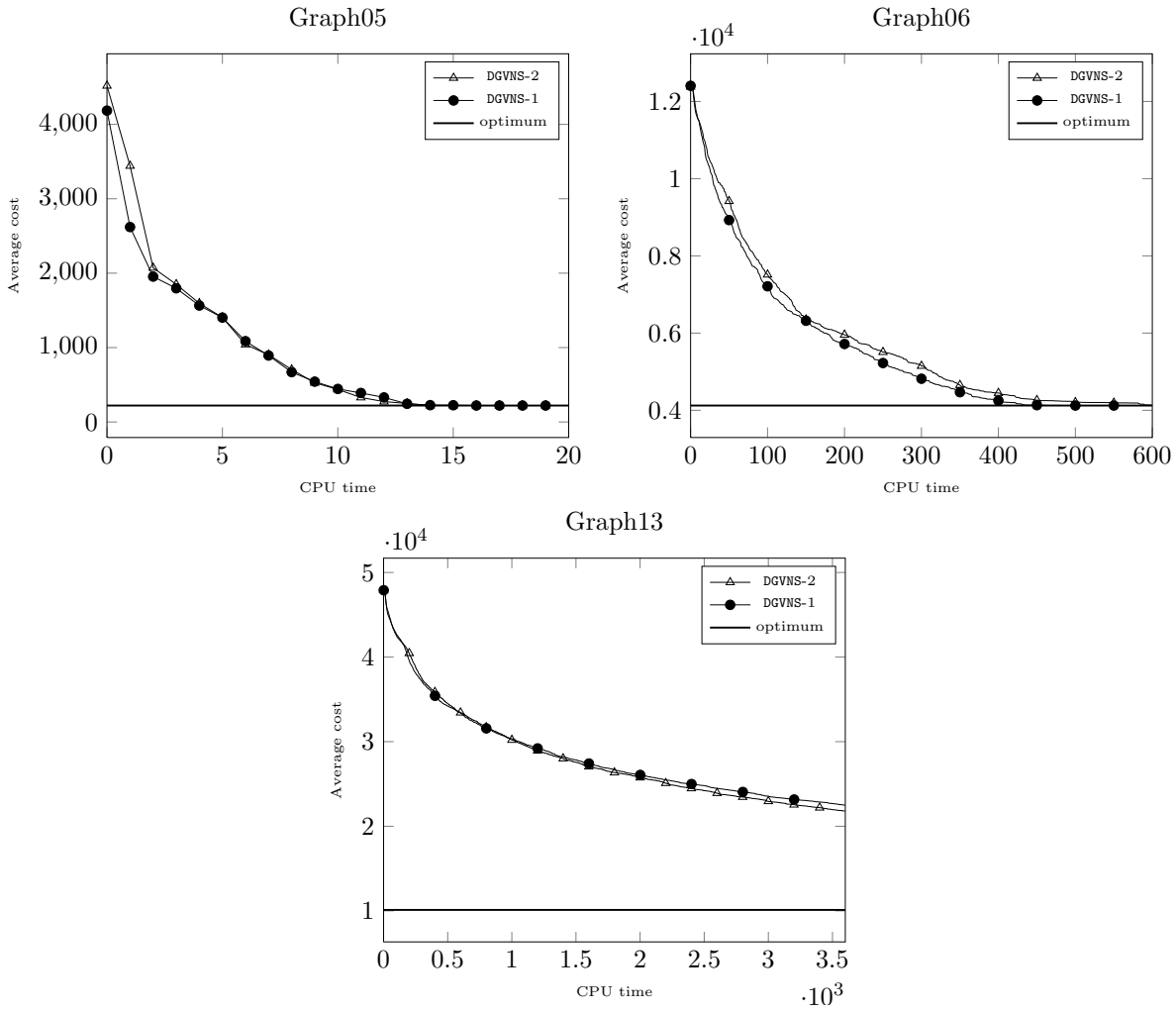


FIGURE 5.12 – Comparaison des profils de performance de DGVNS-1 et DGVNS-2 sur les instances GRAPH.

	Temps			Succès		
Méthode	AUC	$\rho(1)$	$\frac{\tau_{min}}{\tau*}$	AUC	$\rho(1)$	$\rho(\tau_{max})$
dgvns-2	0.952(2)	0.5(2)	1.0(1)	1.0(1)	0.731(1)	0.885(1)
dgvns-1	1.0(1)	0.654(1)	0.92(2)	0.935(2)	0.5(2)	0.885(1)

TABLE 5.10 – Comparaison entre DGVNS-1 et DGVNS-2 selon différents critères.

5.3.4.5 Bilan sur l'impact des stratégies d'intensification et de diversification

Nos expérimentations montrent clairement l'impact de la stratégie de changement de voisinage sur les performances de DGVNS. Premièrement, les résultats obtenus par DGVNS-3 montre l'inefficacité de l'intensification de la recherche dans un cluster ayant déjà amené à une amélioration. Deuxièmement, la diversification effectuée en considérant $C_{succ(i)}$ comme le prochain cluster est nécessaire. Troisièmement, DGVNS-2 obtient de meilleurs résultats sur la plupart des instances (excepté sur les instances tagSNP). Toutefois, DGVNS-1 est plus rapide (cf. les profils de performance). Ces résultats démontrent clairement la pertinence de favoriser un meilleur compromis entre intensification et diversification.

Enfin, pour les instances tagSNP, les performances de DGVNS-2 sont moins bonnes que celles de DGVNS-1, mais DGVNS-2 reste néanmoins très compétitif. Ces résultats peuvent probablement s'expliquer par la taille des instances et les durées de résolution qui sont 2 à 4 fois plus grandes comparées aux instances RLFAP et SPOT5. En effet, sur ces instances, rester dans un même cluster nécessite beaucoup plus d'effort pour trouver une nouvelle amélioration de la solution, en raison de la taille importante des voisinages. Ceci ralentit considérablement la vitesse d'amélioration de la qualité des solutions fournies par DGVNS-2. Ce n'est pas le cas pour DGVNS-1 qui bénéficie du changement systématique des structures de voisinage pour améliorer les solutions plus rapidement. Cela aide DGVNS-1 à renforcer le majorant utilisé par LDS dans les itérations suivantes de l'étape de reconstruction. Ces observations sont confirmées par les profils de performance, où DGVNS-1 présente des profils de performance bien meilleurs que ceux de DGVNS-2, avec une amélioration plus rapide de la qualité de la solution.

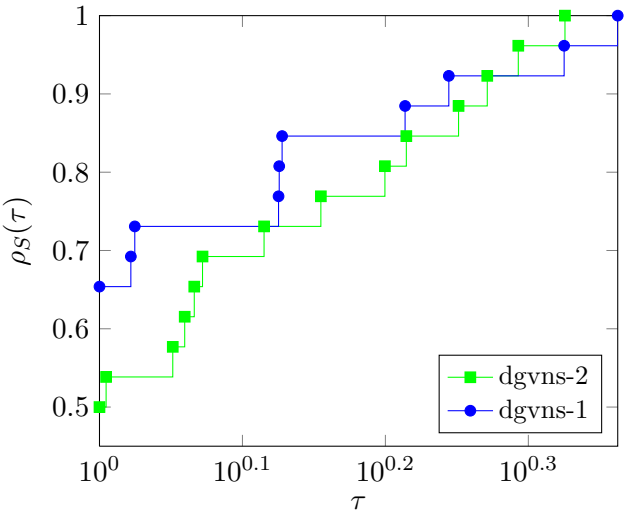
5.3.5 Profils de performance des rapports de temps et de succès

Dans cette section, nous étudions les profils de performances des rapports de temps et de succès de DGVNS-1 et DGVNS-2 sur les 26 instances considérées lors de nos expérimentations. Nous reportons également, pour chaque méthode et chaque mesure, les valeurs associées aux critères $\rho(1)$, AUC, $\rho(\tau_{max})$ et τ_{min}/τ^* (cf. tableau 5.10). Les valeurs entre parenthèses indiquent le classement de chaque méthode pour chaque critère.

La figure 5.13(a) compare les courbes des profils de performances des rapports de temps de DGVNS-1 et DGVNS-2 pour différentes valeurs de τ. Comme nous pouvons le constater, DGVNS-1 domine nettement DGVNS-2 : DGVNS-1 obtient un meilleur profil de performance dans l'intervalle $[1, 10^{0.3}]$ et $AUC_{\texttt{DGVNS}-1} > AUC_{\texttt{DGVNS}-2}$ (cf. colonne 2, tableau 5.10). Par ailleurs, DGVNS-1 est plus rapide sur environ 65% des problèmes contre seulement 50% pour DGVNS-2 (cf. colonne 3, tableau 5.10). Notons toutefois, que DGVNS-2 est plus robuste : elle résout 100% des problèmes avec un facteur 2.12 de la méthode la plus rapide contre 2.30 pour DGVNS-1 (cf. colonne 4, tableau 5.10).

En termes de succès (cf. figure 5.13(b)), la courbe de DGVNS-2 présente un meilleur profil de performance et une meilleure AUC : $AUC_{\texttt{DGVNS}-2} > AUC_{\texttt{DGVNS}-1}$ (cf. colonne 5, tableau 5.10).

(a) Temps de calcul



(b) Nombre d'essais avec succès

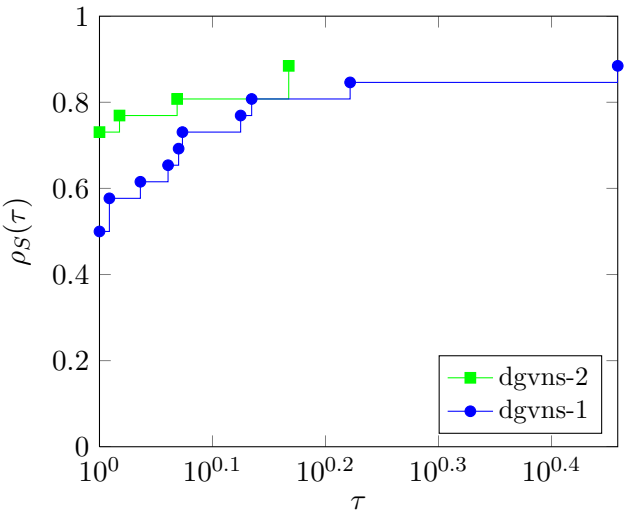


FIGURE 5.13 – Comparaison des profils de rapport de performance de DGVNS-1 et DGVNS-2.

De plus, DGVNS-2 obtient les meilleurs taux de succès sur 19 instances ($\rho(1) = 0.73$) contre 13 instances pour DGVNS-1 ($\rho(1) = 0.5$).

Ces résultats confirment donc nos conclusions dressées précédemment, à savoir que DGVNS-1 obtient un meilleur score sur les critères temps de calcul, alors que DGVNS-2 est bien meilleure en termes de taux de succès.

5.4 Conclusions

Nous avons montré que la *décomposition arborescente* permet de guider efficacement l'exploration de l'espace de recherche. Pour cela, nous avons proposé un nouveau schéma de recherche locale, noté DGVNS, permettant d'exploiter le graphe de clusters pour construire des structures de voisinage basées sur la notion de cluster. Par ailleurs, nous avons proposé plusieurs stratégies permettant une meilleure intensification et diversificaion dans DGVNS. Notre approche est générique et peut être appliquée à d'autres méthodes de recherche locale.

Nous avons montré expérimentalement que l'exploitation des décompositions arborescentes permet de surclasser VNS/LDS+CP et ID-Walk sur différents jeux de données issus de problèmes réels. Nous avons également étudié l'impact de la largeur de la décomposition sur notre approche. De cette étude, il ressort que DGVNS donne de très bons résultats sur les problèmes qui se décomposent en clusters de petite taille faiblement connectés. Enfin, nous avons comparé nos trois stratégies. Les résultats montrent que DGVNS-2 réalise un meilleur compromis entre intensification et diversification.

Chapitre 6

Raffinements de la décomposition arborescente

Sommaire

Dans le chapitre précédent, nous avons montré l'intérêt d'exploiter les clusters issus de la décomposition arborescente du graphe de contraintes, pour guider l'exploration des voisinages dans la méthode VNS. Cependant, nous avons constaté que la plupart des instances des problèmes étudiés (GRAPH, RLFAP, SPOT5, tagSNP) contiennent souvent des fonctions de coût beaucoup plus difficiles à satisfaire que d'autres. Or, il serait intéressant de tenir compte de cette information lors de la décomposition arborescente, en supprimant au préalable toutes les fonctions de coût pas assez dures. Par ailleurs, pour la plupart de ces instances, les décompositions obtenues par MCS comportent de nombreux clusters qui se chevauchent très fortement. De plus, ces clusters contiennent peu ou pas de variables propres. Cette forme de redondance entre clusters limite l'effort de diversification de DGVNS, en considérant plusieurs fois des voisinages très proches.

Pour pallier à ces deux inconvénients, nous proposons dans ce chapitre deux raffinements de la décomposition arborescente. Le premier raffinement, la *Tightness Dependent Tree Decomposition* (TDTD), exploite la dureté des foncions de coût pour identifier les parties du problème les plus difficiles à satisfaire. Le second raffinement consiste à augmenter la proportion de variables

propres dans les clusters, en fusionnant ceux ayant très peu de variables propres. À cet effet, nous proposons deux critères. Le premier consiste à fusionner les clusters partageant plus de variables qu'un seuil maximal fixé. Le second, basé sur la notion *d'absorption*, consiste à fusionner les clusters ayant un taux d'absorption (i.e. proportion de variables partagées) supérieur à un seuil maximal fixé.

Plan du chapitre. Tout d'abord, nous présentons la *Tightness Dependent Tree Decomposition* (`TDTD`) (section 6.1). Ensuite, nous détaillons le raffinement basé sur la fusion de clusters (section 6.2). À cet effet, nous étudions deux critères. Le premier consiste à fusionner les clusters partageant plus de variables qu'un seuil maximal fixé (section 6.2.1). Le second, basé sur la notion *d'absorption* (section 6.2.2), consiste à fusionner les clusters ayant un taux d'absorption (i.e. proportion de variables partagées) supérieur à un seuil maximal fixé. Nous montrons la pertinence et l'intérêt de ces deux raffinements sur les instances `RLFAP`, `SPOT5`, `GRAPH` et `tagSNP` (section 6.3). Enfin, nous concluons.

6.1 Raffinement basé sur la dureté des fonctions de coût

Les problèmes d'optimisation contiennent souvent des sous-parties plus dures à résoudre que les autres en raison de la difficulté à satisfaire les fonctions de coût qui les composent. Ces sous-parties restent cachées dans la décomposition par des fonctions de coût plus faciles à satisfaire. Afin de guider `DGVNS` vers ces sous-parties difficiles, nous proposons d'exploiter *la dureté* des fonctions de coût lors de la décomposition. Nous évaluons la dureté d'une fonction de coût selon la définition suivante :

Définition 60 (Dureté d'une fonction de coût)**.** *Soit w_S une fonction de coût et D^S le produit cartésien des domaines des variables de S. La dureté $t(w_S)$ associée à w_S est définie par :*

$$t(w_S) = \frac{|\{t \,|\, w_S(t) > 0, t \in D^S\}|}{|\{D^S\}|} \tag{6.1}$$

Plus la dureté d'une fonction de coût est forte, plus il est difficile de la satisfaire car le nombre d'affectations valides est plus faible. Ainsi, retirer toutes les fonctions de coût jugées peu dures du graphe de contraintes nous permettra de réunir, dans les clusters, les variables liées par la porté des fonctions de coût les plus dures du problème, tout en conservant un graphe suffisamment dense. De plus, cela entrainera un plus fort filtrage sur ces variables.

6.1.1 *Tightness Dependent Tree Decomposition* (`TDTD`)

Soit G le graphe de contraintes associé à un CFN et λ un seuil de dureté minimal. Pour une valeur de λ donnée, on définit les notions de fonction de coût λ-dure et de graphe de contraintes λ-dur comme suit :

Définition 61 (Fonction de coût λ-dure, graphe de contraintes λ-dur)**.** *Une fonction de coût w_S est dite λ-dure, si sa dureté est supérieure ou égale à λ. On appelle graphe de contraintes λ-dur de G, noté G^λ, le graphe obtenu en supprimant les fonctions de coût non λ-dures de G.*

Définition 62 (Décomposition λ-dure)**.** *On appelle décomposition λ-dure de G la décomposition obtenue à partir de G^λ.*

La `TDTD` d'un graphe de contraintes G, pour un seuil λ, s'effectue comme suit :

A	B	E	$w_{A,B,E}$
a	**a**	**b**	**10**
a	**b**	**b**	**10**
a	**a**	**c**	**10**
a	**b**	**c**	**100**
b	**a**	**b**	**500**
b	**b**	**b**	**200**
b	a	c	0
b	b	c	0

$t(w_{A,B,E}) = 0.75$

A	C	$w_{A,C}$
a	a	0
a	c	0
b	**a**	**1,000**
b	c	0

$t(w_{A,C}) = 0.25$

C	D	$w_{C,D}$
a	**b**	**10**
c	**b**	**120**
a	c	0
c	**c**	**350**

$t(w_{C,D}) = 0.75$

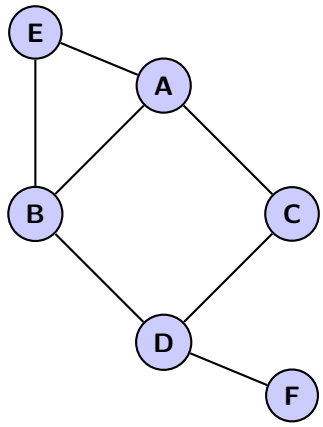
B	D	$w_{B,D}$
a	**b**	**1,000**
a	c	0
b	b	0
b	**c**	**10**

$t(w_{B,D}) = 0.5$

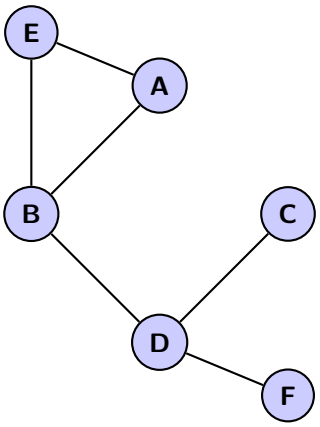
D	F	$w_{D,F}$
b	**a**	**1,000**
b	**c**	**1,000**
c	**a**	**1,000**
c	c	0

$t(w_{D,F}) = 0.75$

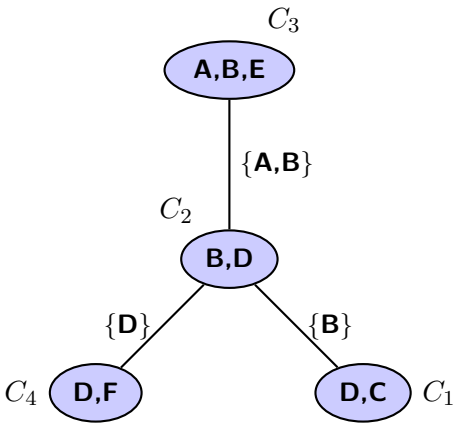
(a) Tables des fonctions de coût du problème.



(b) Graphe de contraintes original.



(c) Graphe de contraintes 0.3-dur.



(d) Décomposition arborescente 0.3-dure.

FIGURE 6.1 – Exemple de décomposition 0.3-dure d'un WCSP composé de six variables ayant pour domaines $D_A = D_B = \{a, b\}$, $D_C = \{a, c\}$, $D_D = D_E = \{b, c\}$ et $D_F = \{a, c\}$, d'une fonction de coût ternaire $w_{A,B,E}$ et de quatre fonctions de coût binaires.

1. On retire toutes les fonctions de coût non λ-dures du graphe de contraintes G ;
2. On calcule la décomposition λ-dure associée à G^λ

La figure 6.1 donne un exemple de décomposition 0.3-dure. Tout d'abord, la dureté $t(w_S)$ de chaque fonction de coût du problème est calculée (figure 6.1a). Toutes les fonctions de coût non 0.3-dures sont retirées de G. Dans notre exemple, $w_{A,C}$ est supprimée car sa dureté $t(w_{A,C})$=0.25 est inférieure à $\lambda = 0.3$ (figure 6.1c). Enfin, la décomposition arborescente 0.3-dure est calculée (figure 6.1d).

Remarque 5. *Notre définition de la dureté d'une fonction de coût ne prend pas en compte les coûts. En effet, elle dépend uniquement du nombre de n-uplets ayant un coût non nul. Soit w_1 et w_2 deux fonctions de coût, de même portée et contenant le même nombre de n-uplets de coût non nul. Supposons que w_1 et w_2 assignent respectivement les coûts 1 et 1000 à la même proportion de ces n-uplets. Les deux fonctions de coût auront la même dureté. Ainsi, si $\lambda > t(w_1)$, w_1 et w_2 seront alors retirées par la* TDTD. *Cependant w_2 peut avoir plus d'influence que w_1 sur le coût d'une solution. Il serait donc intéressant de prendre en compte les coûts, comme dans [Kitching et Bacchus, 2009], lors de la décomposition du graphe de contraintes. Nous y reviendrons dans la*

λ	% dropped	$\lvert C_T \rvert$	Taille des clusters			Degré des clusters			Taille des séparateur			
	c		min.	moy.	max.	min.	moy.	max.	min.	moy.	max.	nb
0	0	55	2	4.9	12	1	9.6	25	1	1.77	8	266
0.1	1	59	2	5.1	11	1	10.27	26	1	1.9	10	303
0.2	3	61	2	4.9	11	1	10.06	26	1	1.89	10	307
0.3	5	61	2	4.91	11	1	10.06	26	1	1.8	10	307
0.4	8	60	1	4.75	11	0	7.8	21	1	1.9	10	234
0.5	14	56	1	4.3	11	0	5.57	14	1	1.75	9	156
0.6	28	65	1	3.93	10	0	5.29	14	1	1.93	9	172
0.7	53	74	1	2.98	10	0	4.10	10	1	1.61	6	152
0.8	54	72	1	2.97	10	0	3.63	9	1	1.64	6	131
0.9	76	80	1	1.85	9	0	1.6	7	1	1.18	5	64
1	100	100	1	1	1	0	0	0	0	0	0	0

TABLE 6.1 – TDTD sur l'instance Scen06.

λ	% dropped	$\lvert C_T \rvert$	Taille des clusters			Degré des clusters			Taille des séparateur			
	c		min.	moy.	max.	min.	moy.	max.	min.	moy.	max.	nb
0	0	94	2	29.0	100	1	18.0	40	1	18.0	99	1,726
0.1	18	95	2	29.0	100	1	18.0	41	1	18.0	99	1,762
0.2	32	98	1	24.0	94	0	15.0	32	1	14.0	93	1,500
0.3	61	230	1	5.0	40	0	9.0	41	1	5.0	33	2,146
0.4	61	230	1	5.0	40	0	9.0	41	1	5.0	33	2,146
0.5	61	230	1	5.0	40	0	9.0	41	1	5.0	33	2,146
0.6	61	230	1	5.0	40	0	9.0	41	1	5.0	33	2,146
0.7	61	230	1	5.0	40	0	9.0	41	1	5.0	33	2,146
0.8	61	230	1	5.0	40	0	9.0	41	1	5.0	33	2,146
0.9	61	230	1	5.0	40	0	9.0	41	1	5.0	33	2,146
1	61	230	1	5.0	40	0	9.0	41	1	5.0	33	2,146

TABLE 6.2 – TDTD sur l'instance #509.

conclusion de ce mémoire.

6.1.2 Influence du paramètre λ sur la décomposition arborescente

Comme nous l'avons indiqué en section 5.3.3, toute décomposition arborescente peut se caractériser par trois mesures : la largeur de la décomposition, la taille des séparateurs et le degré des clusters. Afin d'évaluer l'impact du paramètre λ sur la qualité de la décomposition obtenue, les tableaux 6.1 et 6.2 reportent, pour différents seuils λ, les valeurs de ces mesures obtenues pour les instances Scen06 et #509. La col. 1 donne les différentes valeurs de λ. La col. 2 indique le pourcentage de fonctions de coût retirées. La col. 3 présente le nombre total de clusters. Les col. 4-12 reportent respectivement, pour chacune des trois mesures (*taille des clusters*, *degré des clusters* et *taille des séparateurs*), leurs valeurs minimale, moyenne et maximale. Enfin la dernière colonne indique le nombre de séparateurs. La décomposition arborescente sur le graphe initial correspond au seuil $\lambda = 0$.

Instance Scen06 (cf tableau 6.1). On peut remarquer que plus la valeur de λ augmente, plus le nombre de clusters augmente. Toutefois, pour des valeurs de λ comprises entre 0.1 et 0.6, cette augmentation reste faible. À l'inverse, les autres paramètres diminuent avec l'augmentation de la valeur de λ. En effet, plus le nombre de fonctions de coût retirées est élevé, moins le graphe de contraintes est dense, conduisant à des clusters de taille et de degré de plus en plus petit et à des séparateurs ayant des tailles relativement faibles.

Deuxièmement, pour $0.2 \leq \lambda \leq 0.5$, la `TDTD` donne le plus souvent un grand nombre de clusters de petite taille, tout en conservant les fonctions de coût les plus importantes du problème initial. De plus, les clusters ont (en moyenne) un plus haut degré [8] (une connexité plus grande) que ceux obtenus avec $\lambda = 0$. Ceci montre l'intérêt et l'importance de la `TDTD`. Cependant, pour des valeurs élevées de λ ($\lambda \geq 0.6$), la `TDTD` n'est pas pertinente car trop de fonctions de coût sont retirées, conduisant à des clusters isolés de taille négligeable. Les résultats obtenus sur les autres instances du `RLFAP` sont très similaires.

Instance #509 (cf tableau 6.2). Pour $0.1 \leq \lambda \leq 0.3$, les décompositions produites contiennent plus de clusters qui sont (en moyenne) plus petits et de plus faible degré. Cependant, pour $\lambda \geq 0.3$, la `TDTD` produit systématiquement la même décomposition. En effet, l'intégralité des fonctions de coût restantes sont 1-*dures* (i.e. toutes les affectations possibles des variables sur lesquelles elles portent sont de coût non nul). Sur les autres instances `SPOT5`, la `TDTD` est également plus pertinente pour $0.1 \leq \lambda \leq 0.3$ (cf. section 6.3 pour plus de détails).

Enfin, les instances `tagSNP` ne sont pas abordées dans cette section. En effet, ces instances sont composées de fonctions de coût de dureté comprises entre 0.1 et 0.2. Il est donc impossible d'appliquer la `TDTD` sans retirer l'intégralité des fonctions de coût ou du moins sans perdre la structure du problème.

6.1.3 Réglage de la valeur du paramètre λ

Déterminer la valeur optimale de λ dépend du problème traité. Nous avons déterminé empiriquement le meilleur réglage du seuil λ, en évaluant les performances de `DGVNS` pour différentes valeurs comprises entre 0.1 et 0.6. Les valeurs de λ supérieures à 0.6 conduisent à des décompositions constituées de clusters beaucoup plus isolés (i.e. de degré faible). Malgré son caractère empirique, notre approche reste applicable et ne nécessite pas une connaissance approfondie des instances à résoudre.

6.2 Raffinement par fusion de clusters

Comme nous l'avons indiqué au début de ce chapitre, la plupart des décompositions arborescentes, obtenues par `MCS` sur les différents problèmes étudiés (`RLFAP`, `SPOT5`, `GRAPH` et `tagSNP`), se composent de nombreux clusters contenant pas ou peu de variables propres, car se chevauchant très fortement. Cette forme de redondance entre clusters limite la diversification de `DGVNS`, en considérant, de manière répétée, des voisinages très proches.

Afin de limiter la redondance entre clusters et donc de renforcer la qualité des structures de voisinage explorées par `DGVNS`, nous proposons de fusionner les clusters qui sont redondants dans la décomposition arborescente. L'idée sous-jacente est d'augmenter la proportion de variables propres dans les clusters. À cet effet, nous proposons deux critères :

(i) *taille du plus grand séparateur* (s_{max}). Ce critère consiste à fusionner les clusters partageant plus de variables qu'un seuil maximal fixé s_{max}.

(ii) *taux d'absorption maximal* (Ab_{max}). Ce critère, basé sur la notion *d'absorption*, consiste à fusionner les clusters ayant un taux d'absorption (i.e. proportion de variables partagées) supérieur à un seuil maximal fixé Ab_{max}.

8. La suppression des fonctions de coût du graphe de contraintes initial permet de fractionner de nombreux clusters en de nouveaux clusters de plus petite taille, faisant augmenter les degrés des clusters partageant des variables avec ces nouveaux clusters.

Algorithme 25 : Fusion basée sur le critère s_{max}

```
1 function fusionSep (Cluster C, s_max) ;
2 début
3     pour tout C_i ∈ fils(C) faire
4         fusionSep (C_i, s_max) ;
5     si (| C ∩ parent(C) |> s_max) alors
6         parent(C) ← parent(C) ∪ {C} ;
```

Nous noterons DGVNS-s_{max} (resp. DGVNS-abs) la méthode DGVNS exploitant la décomposition arborescente obtenue par le critère s_{max} (resp. Ab_{max}). Ces deux approches sont détaillées dans les sections qui suivent.

6.2.1 Fusion de clusters basée sur s_{max}

La problématique de fusion de clusters est au cœur des méthodes complètes exploitant les décompositions arborescentes [Dechter et Fattah, 2001]. Dans [Jégou *et al.*, 2006], les auteurs proposent de fusionner les clusters en se basant sur le critère s_{max} afin d'augmenter le degré de liberté pour les heuristiques de choix de variable au sein des clusters. Dans [Sánchez *et al.*, 2009], les auteurs utilisent le même principe pour réduire la complexité spatiale de la méthode RDS-BTD.

Afin de générer de nouvelles décompositions à partir de celles obtenues par MCS, nous avons suivi le schéma proposé dans [Jégou *et al.*, 2006], en bornant la taille maximale des séparateurs s_{max}. Tout d'abord, nous calculons une décomposition arborescente en fixant s_{max} à $+\infty$ (cas où la taille des séparateurs n'est pas limitée). Ensuite, en partant des feuilles de la décomposition produite par MCS, nous fusionnons les clusters C_i qui partagent plus de s_{max} variables avec leur parent $\texttt{parent}(C_i)$. Nous obtenons ainsi une nouvelle décomposition, dans laquelle aucun séparateur ne contient plus de s_{max} variables. L'algorithme 25 détaille le principe de cette fusion.

Cette approche souffre cependant de sa rigidité. En effet, le critère de fusion ne permet pas de prendre en compte la structure de l'instance considérée : limiter s_{max} à 32 sur des instances de problèmes ayant des séparateurs de grande taille conduira à fusionner tous les clusters. De plus, avec ce critère, on pourra fusionner deux clusters de taille 100 partageant 32 variables mais pas ceux de taille 32 partageant 31 variables. Pour remédier à ces inconvénients nous proposons une fusion basée sur la notion d'absorption.

6.2.2 Fusion de clusters basée sur l'absorption

Afin de tenir compte des tailles respectives des clusters et de leurs séparateurs, dans le mécanisme de fusion, nous introduisons un second critère appelé *absorption*.

Définition 63 (Absorption)**.** *Soit $(\mathcal{C}, T)$ une décomposition arborescente, C_i et C_j deux clusters de $\mathcal{C}$. L'absorption de C_i par C_j, notée $Ab(C_i, C_j)$, est définie par :*

$$Ab(C_i, C_j) = \frac{|sep(C_i, C_j)|}{|C_i|}$$

Soit Ab_{max} le taux d'absorption maximal autorisé, et min_{size} la taille minimale autorisée pour un cluster. La fusion s'effectue sur la décomposition obtenue par MCS. En partant des feuilles de cette décomposition, tous les clusters $< C_i, \texttt{parent}(C_i) >$ ayant un taux d'absorption supérieur à

Algorithme 26 : Fusion basée sur le critère Ab_{max}

```
1 function fusionAbs (Cluster C, Ab_max) ;
2 début
3     pour tout C_i ∈ fils(C) faire
4         fusionAbs (C_i, Ab_max) ;
5     si Ab(C, parent(C)) > Ab_max ∨ Ab(parent(C), C) > Ab_max ∨ |C| < min_size) alors
6         parent(C) ← parent(C) ∪ {C} ;
```

Ab_{max} ou tels que $|C_i| < min_{size}$ sont fusionnés (cf. algorithme 26). Pour nos expérimentations, min_{size} a été fixée à la valeur 5 et Ab_{max} à 70%. Ainsi, les clusters partageant un peu plus de 2/3 de leurs variables seront fusionnés.

6.2.3 Influence des méthodes de fusion sur la décomposition arborescente

Cette section compare, de manière qualitative, les décompositions obtenues par les deux méthodes de fusion sur deux instances du problème `tagSNP` et sur une instance du problème `SPOT5`. Les résultats de ces comparaisons sont présentés figures 6.2, 6.3 et 6.4. Les résultats obtenus pour les instances `RLFAP` sont très proches de ceux obtenus pour les instances `SPOT5`, et ne sont donc pas présentés ici.

Pour chaque figure, les 3 premiers graphiques correspondent à la fusion s_{max} et le dernier correspond à la fusion par absorption. Chaque graphique indique, pour une valeur fixée de s_{max}, (i) la répartition des clusters de la décomposition initiale obtenue par `MCS`, et (ii) la répartition des clusters obtenus par fusion, en fonction de leur taille et de leur pourcentage de variables propres. Les points rouges représentent les clusters de la décomposition initiale, les points verts les clusters obtenus après fusion, et les points noirs les clusters communs. Chaque point vert indique aussi le nombre de clusters qui ont été fusionnés pour son obtention.

Considérons les résultats obtenus pour l'instance `tagSNP` #4449 (cf. figure 6.2). Pour (s_{max}=32), 24 clusters ont été fusionnés en 4 nouveaux clusters, dont un partageant plus de 90% de ses variables et un autre contenant plus de 120 variables. Mais, un grand nombre de clusters ayant 100% de leurs variables partagées n'ont pas été fusionnés. (s_{max}=24) permet de réduire quelque peu cette redondance. Par contre, la fusion par absorption permet de supprimer la totalité des clusters redondants et de générer une décomposition contenant des clusters ayant au moins 50% de variables propres. Par ailleurs, la taille du grand séparateur est égale à 14 (i.e. $s_{max} = 14$). On peut dresser les mêmes constats sur la très grande majorité des autres instances `tagSNP`.

Considérons les résultats obtenus pour l'instance `SPOT5` #507 (cf. figure 6.3). Pour (s_{max}>12), les décompositions produites sont très similaires à la décomposition initiale. Celles-ci sont caractérisées par des clusters qui se recouvrent très fortement. Pour ($s_{max}\leq$12), les décompositions obtenues ont des largeurs arborescentes plus élevées, mais les clusters contiennent plus de variables propres. À l'inverse, la fusion par absorption produit une décomposition constituée de beaucoup plus de clusters qui se chevauchent moins fortement (i.e., $s_{max} = 10$). Par ailleurs, la largeur de décomposition est deux fois moins élevée comparée à celle obtenue pour (s_{max}=4). On peut dresser les mêmes constats sur une grande majorité des autres instances `SPOT5`.

Pour l'instance `tagSNP` #6858 (cf. figure 6.4), les deux méthodes produisent des décompositions contenant peu de clusters. Pour (s_{max}=32), 101 clusters ont été fusionnés en 3 nouveaux clusters,

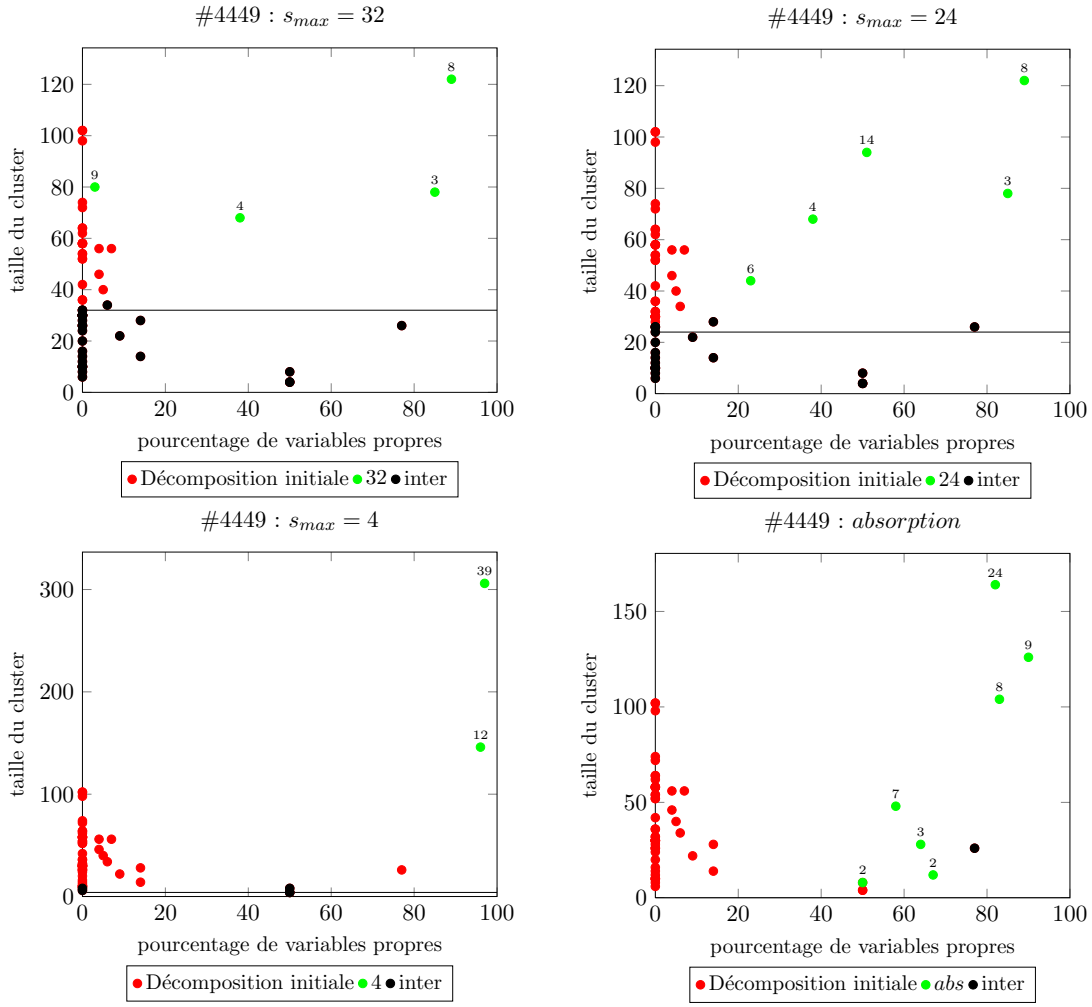


FIGURE 6.2 – Comparaison des décompositions obtenues par les deux méthodes de fusion sur l'instance `tagSNP` #4449.

dont un de très grande taille (près de 1000 variables). Pour (s_{max}<32), les décompositions produites sont très similaires à la décomposition obtenue avec (s_{max}=32). La fusion par absorption produit également une décomposition constituée de peu de clusters (103 clusters ont été fusionnés en 3 nouveaux clusters), partageant au plus 36 variables (i.e., $s_{max} = 36$). Ces résultats peuvent s'expliquer par la structure même de l'instance : la décomposition obtenue par `MCS` est constituée de clusters de taille et de degré très élevés. Par ailleurs, la quasi totalité des clusters partagent 100% de leurs variables (cf. points rouges et noirs de la figure 6.4). Ainsi, sur ce type d'instances, la différence entre les décompositions produites par les deux méthodes de fusion n'est pas vraiment discriminative.

6.2.4 Bilan des deux méthodes de fusion

Les résultats obtenus montrent clairement l'apport de la fusion par absorption. En effet, pour des valeurs de s_{max} élevées, la fusion modifie peu la décomposition initiale. En revanche, pour des valeurs de s_{max} faibles, les décompositions produites ont peu de clusters et possèdent des largeurs de décomposition très élevées. De plus, la plupart des clusters partageant 100% de leurs variables

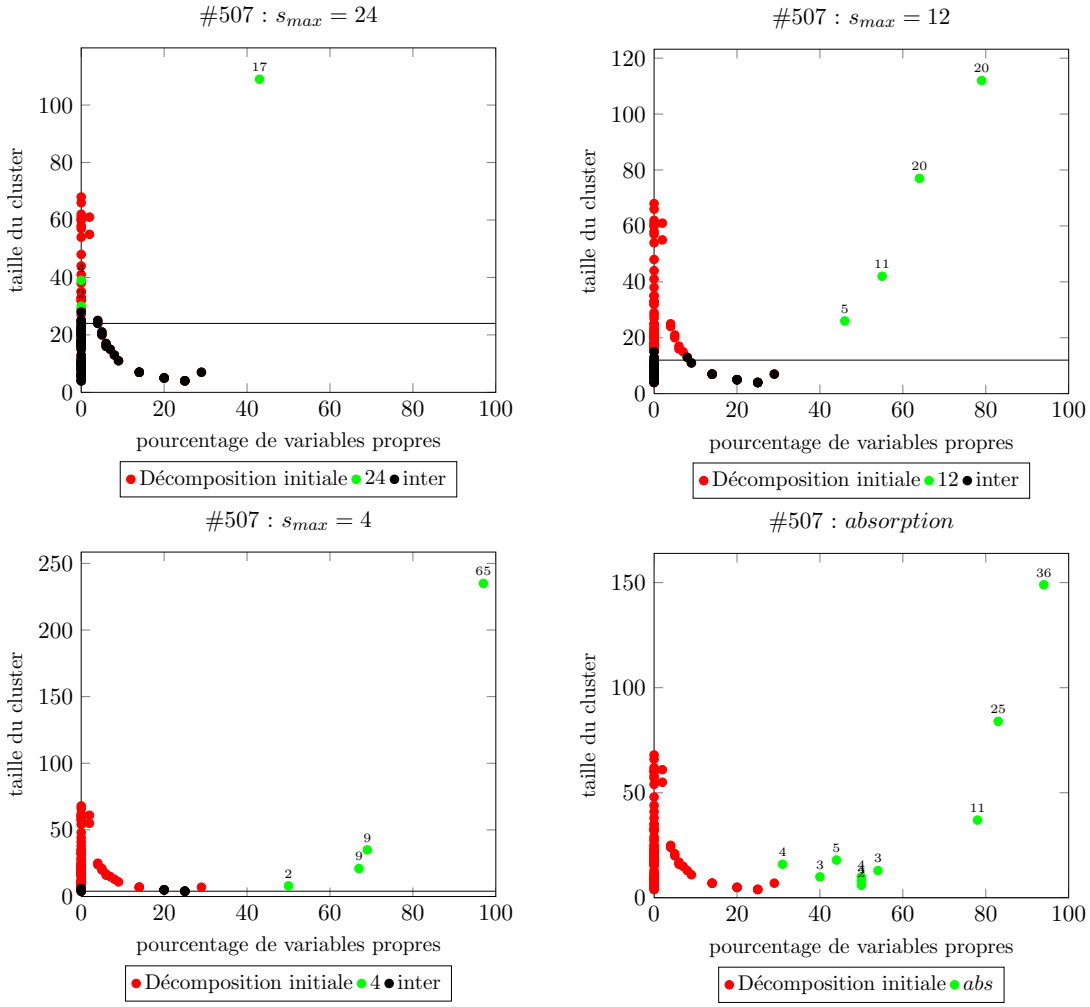


FIGURE 6.3 – Comparaison des décompositions obtenues par les deux méthodes de fusion sur une instance `SPOT5`.

sont conservés. À l'inverse, la fusion par absorption permet de produire des décompositions de bien meilleure qualité : des clusters de taille raisonnable avec une forte proportion de variables propres. Notons, toutefois, une augmentation relative de la largeur de décomposition sur certaines instances.

6.3 Expérimentations

Les expérimentations ont été menées dans les mêmes conditions qu'au chapitre 5 : 50 essais par méthode avec un *TimeOut* d'une heure pour les instances `RLFAP` et `SPOT5`. Pour les instances `tagSNP` de taille moyenne (resp. de grande taille), le *TimeOut* a été fixé à 2 heures (resp. 4 heures). Nous avons considéré différentes valeurs de s_{max} : $4, 8, 12, 16, 24$ et 32.

Pour chaque instance et chaque méthode, nous reportons :
- le nombre d'essais avec succès ;
- le temps de calcul moyen pour atteindre l'optimum ;
- le coût moyen sur les 50 essais des solutions trouvées ;

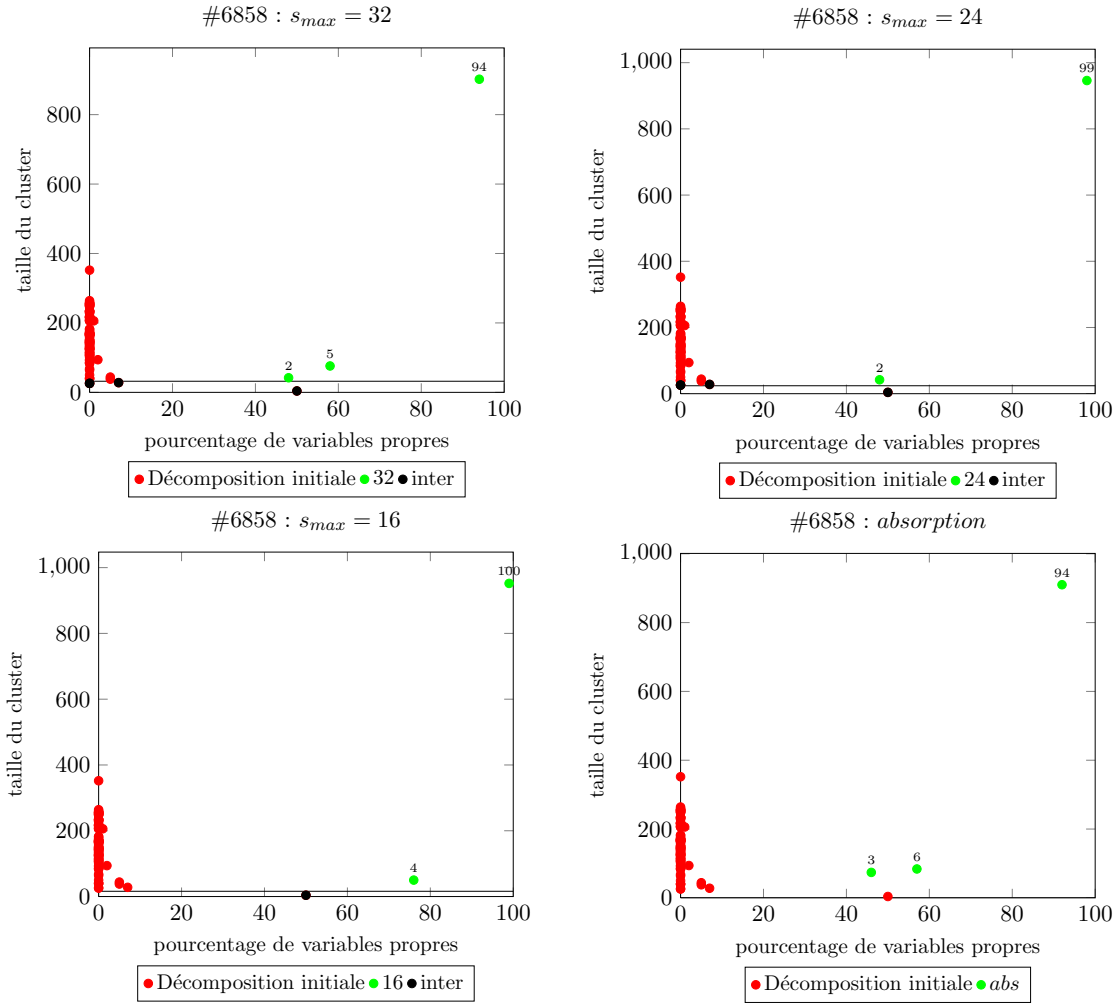


FIGURE 6.4 – Comparaison des décompositions obtenues par les deux méthodes de fusion sur l'instance `tagSNP` #6858.

– le meilleur coût (noté entre parenthèses) si l'optimum n'a pas été atteint.

Pour les méthodes basées sur la fusion, nous reportons également :
– le nombre de clusters de la décomposition obtenue ;
– la valeur moyenne et la valeur maximale de la taille des clusters ;
– si nécessaire, les valeurs de s_{max}.

Tout d'abord, nous comparons `DGVNS`-1 et `DGVNS`-2 avec les versions utilisant la `TDTD` (`TDGVNS`-1-λ et `TDGVNS`-2-λ) (cf. section 6.3.1). Ensuite, nous évaluons successivement les apports de la fusion basée sur la taille du plus grand séparateur (cf. section 6.3.2) et les apports de la fusion basée sur le taux d'absorption (cf. section 6.3.3), puis nous dressons une étude comparative des deux méthodes de fusion (cf. section 6.3.4). Enfin, nous etudions les profils de rapport de performances comparant les deux méthode de fusion, `DGVNS`-1 et `DGVNS`-2.

6.3.1 Apports de la `TDTD`

Les tableaux 6.3 et 6.4 comparent `DGVNS`-i et `TDGVNS`-i (avec $i = 1, 2$) sur les instances `RLFAP`, `SPOT5` et `GRAPH`. Nous reportons les valeurs associées aux seuils λ ayant donnés les meilleurs

Instance	Méthode	Succ.	Temps	Moy.
Scen06 $S^* = 3,389$	DGVNS-1	50/50	112	3,389
	TDGVNS-1-0.2	**50/50**	**58**	**3,389**
	DGVNS-2	50/50	146	3,389
	TDGVNS-2-0.2	**50/50**	**87**	**3,389**
Scen07 $S^* = 343,592$	DGVNS-1	40/50	317	345,614
	TDGVNS-1-0.4	**49/50**	**221**	**343,600**
	DGVNS-2	46/50	901	344,012
	TDGVNS-2-0.4	**50/50**	**331**	**343,592**
Scen08 $S^* = 262$	DGVNS-1	3/50	1,811	275
	TDGVNS-1-0.5	**9/50**	**442**	**272**
	DGVNS-2	5/50	595	277
	TDGVNS-2-0.4	**8/50**	**673**	**273**

Instance	Méthode	Succ.	Temps	Moy.
#408 S^* =6,228	DGVNS-1	49/50	117	6,228
	TDGVNS-**1**-0.1	**50/50**	**72**	**6,228**
	DGVNS-2	50/50	81	6,228
	TDGVNS-2-0.1	**50/50**	**74**	**6,228**
#412 S^* =32,381	DGVNS-1	36/50	84	32,381
	TDGVNS-**1**-0.1	**38/50**	**71**	**32,381**
	DGVNS-2	48/50	484	32,381
	TDGVNS-2-0.1	**49/50**	**244**	**32,381**
#414 S^* =38,478	DGVNS-1	38/50	554	38,478
	TDGVNS-**1**-0.1	**42/50**	**430**	**38,478**
	DGVNS-2	45/50	670	38,478
	TDGVNS-2-0.1	**46/50**	**478**	**38,478**
#505 S^* =21,253	DGVNS-**1**	**50/50**	**63**	**21,253**
	TDGVNS-1-0.1	48/50	92	21,253
	DGVNS-2	**50/50**	**90**	**21,253**
	TDGVNS-2-0.1	50/50	92	21,253
#507 S^* =27,390	DGVNS-1	33/50	71	27,390
	TDGVNS-**1**-0.1	**45/50**	**62**	**27,390**
	DGVNS-2	**45/50**	**463**	**27,390**
	TDGVNS-2-0.1	43/50	562	27,390
#509 S^* =36,446	DGVNS-**1**	**40/50**	**265**	**36,446**
	TDGVNS-1-0.2	39/50	313	36,446
	DGVNS-2	**47/50**	**509**	**36,446**
	TDGVNS-2-0.1	46/50	621	36,446

TABLE 6.3 – Comparaison entre DGVNS-i et TDGVNS-i (i=1, 2) sur les instances RLFAP et SPOT5.

résultats. Les résultats pour les autres valeurs de λ sont reportés à l'annexe A.

6.3.1.1 Instances RLFAP

Sur les instances RLFAP (cf. tableau 6.3), l'apport de la TDTD est très significatif, particulièrement sur les instances Scen07 et Scen08, pour lesquelles on observe de fortes améliorations par rapport aux résultats de DGVNS-1 et DGVNS-2. Pour la Scen07, TDGVNS-1-0.4 améliore de 18% le taux de succès de DGVNS-1 (de 80% à 98%) et réduit la déviation moyenne à l'optimum (de 0.55% à 0.002%). TDGVNS-2-0.4 et DGVNS-2 obtiennent toutes les deux 100% de succès, mais TDGVNS-2-0.4 est 3 fois plus rapide en moyenne. Pour la Scen08, le taux de succès de DGVNS-1 est amélioré de 12% (de 6% à 18%), la déviation moyenne diminue de 5% à 3.80% et TDGVNS-1-0.5 est 4 fois plus rapide. L'apport de la TDTD sur DGVNS-2 reste comparable à celui obtenu sur DGVNS-1. En effet, TDGVNS-2 améliore le taux de succès de 6% (de 10% à 16%) et la déviation moyenne à l'optimum est réduite de 5.7% à 4.2%. Pour la Scen06, TDGVNS-1-0.2 (resp. TDGVNS-2-0.2) divise par deux le temps de calcul de DGVNS-1 (resp. DGVNS-2).

Notons que la valeur de λ donnant les meilleurs résultats pour TDGVNS-1 et TDGVNS-2 est quasiment identique pour chacune des trois instances. Par ailleurs, TDGVNS-1 obtient systématiquement de meilleurs temps de calcul que TDGVNS-2. Il en était de même pour DGVNS-1 et DGVNS-2.

6.3.1.2 Instances SPOT5

La tendance se confirme sur les instances SPOT5 (cf. tableau 6.3), pour lesquelles TDGVNS-i ($i = 1, 2$) se montrent très efficaces. Sur les grandes instances (#412, #414 et #507), TDGVNS-1 améliore le taux moyen de succès ainsi que le temps de calcul de DGVNS-1 de 12% et 25% respectivement. Sur les autres instances (excepté #509), les deux méthodes ont des performances très similaires.

Instance	Méthode	Succ.	Temps	Moy.
Graph05	DGVNS-1	50/50	10	221
	TDGVNS-1-0.7	**50/50**	**8**	**221**
	DGVNS-2	50/50	10	**221**
	TDGVNS-2-0.7	**50/50**	**11**	**221**
	VNS/LDS+CP	50/50	17	221
Graph06	DGVNS-1	50/50	367	4,123
	TDGVNS-1-0.7	**50/50**	**261**	**4,123**
	DGVNS-2	50/50	413	4,123
	TDGVNS-2-0.6	**50/50**	**293**	**4,123**
	VNS/LDS+CP	**50/50**	**218**	4,123
Graph11	DGVNS-1	8/50	3,046	4,234
	TDGVNS-1-0.5	**12/50**	**2,818**	**3,643**
	DGVNS-2	23/50	3,031	3,480
	TDGVNS-2-0.7	**32/50**	**2,463**	**3,095**
	VNS/LDS+CP	**44/50**	2,403	3,090
Graph13	DGVNS-1	0/50	-	22,489 (18,639)
	TDGVNS-1-0.7	**0/50**	**-**	**11,449 (10,268)**
	DGVNS-2	0/50	-	21,796 (18,323)
	TDGVNS-2-0.7	0/50	-	**11,466 (10,145)**
	VNS/LDS+CP	**3/50**	3,477	14,522

TABLE 6.4 – Comparaison entre DGVNS-1, DGVNS-2, TDGVNS-1 et TDGVNS-2 sur les instances GRAPH.

L'apport de la TDTD sur DGVNS-2 est moins important que pour TDGVNS-1. En effet, sur la plupart des instances, TDGVNS-2 améliore le taux de succès de DGVNS-2 d'au plus un succès. Par ailleurs, TDGVNS-2 obtient de meilleurs temps de calcul **sur** 3 **instances** et est moins performante **sur** 2 **instances** (#507 et #509). Enfin, TDGVNS-2 surclasse TDGVNS-1 en taux de succès (gain de 9% en moyenne). Cependant TDGVNS-1 est 4 fois plus rapide en moyenne. Notons également que les deux méthodes obtiennent leurs meilleurs résultats pour les mêmes valeurs de λ, sur la plupart des instances (#408, #412, #414, #505 et #507).

6.3.1.3 Instances GRAPH

Sur les instances GRAPH (cf. tableau 6.4), la TDTD permet une nouvelle fois d'améliorer les performances de DGVNS-i ($i = 1, 2$). Pour les instances Graph05 et Graph06, DGVNS-i et ses versions TDTD obtiennent des résultats comparables. Pour l'instance Graph11, TDGVNS-1-0.5 améliore le taux de succès de DGVNS-1 de 8% et réduit la déviation moyenne à l'optimum (de 37% à 18%). Cette amélioration est largement amplifiée avec TDGVNS-2-0.7, qui procure un gain de 18% en de taux de succès comparé à DGVNS-2 et réduit la déviation moyenne à l'optimum par rapport à DGVNS-2 (de 12.9% à 0.48%). Ce gain atteint 40% comparé à TDGVNS-1-0.7. Toutefois, ces résultats restent en deçà de ceux obtenus par VNS/LDS+CP aussi bien en taux de succès qu'en qualité moyenne de solutions obtenues. Pour l'instance Graph13, bien que les versions TDTD de DGVNS-i ne trouvent pas l'optimum, elles obtiennent toutefois des solutions de meilleure qualité en moyenne comparé à VNS/LDS+CP. En effet, la déviation moyenne à l'optimum des solutions trouvées par TDGVNS-1-0.7 et TDGVNS-2-0.7 est d'environ 13%, contre 43% pour VNS/LDS+CP. Ceci confirme l'intérêt d'exploiter la dureté des contraintes pour mieux guider DGVNS vers les sous-parties de l'espace de recherche associées aux fonctions de coût ayant un fort impact sur la qualité des solutions produites.

6.3.1.4 Profils de performance

D'un point de vue des profils de performance (figures 6.5 et 6.6), TDGVNS-1 (resp. TDGVNS-2) présente systématiquement de meilleurs comportements que DGVNS-1 (resp. DGVNS-2), excepté

pour les instances Scen06 et #505. Cette tendance est encore amplifiée pour les instances `GRAPH` (figure 6.7), notamment pour Graph06 et Graph13, pour lesquelles la `TDTD` permet d'obtenir une amélioration bien plus rapide de la qualité des solutions comparé à `VNS/LDS+CP` et `DGVNS-`i.

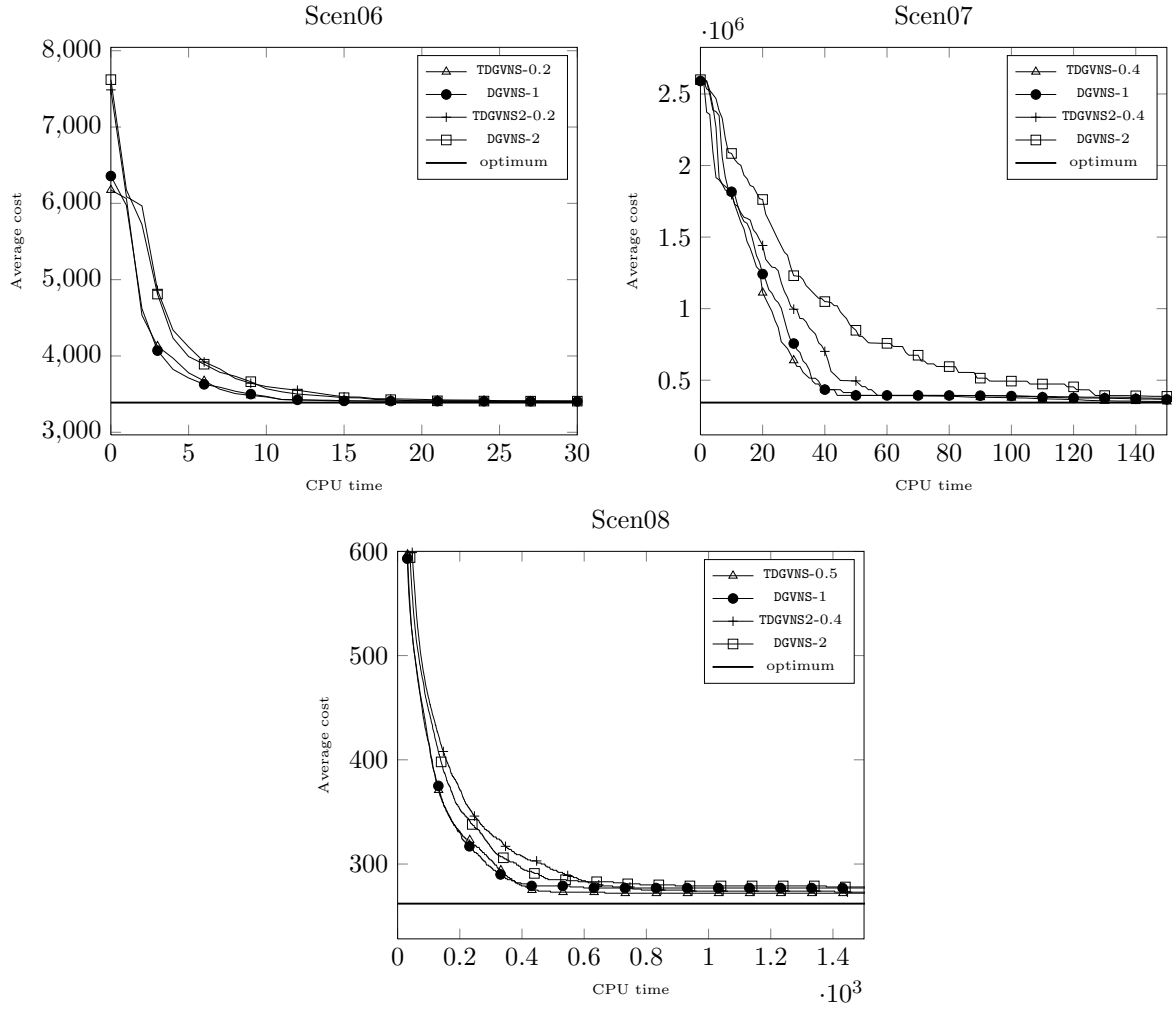


FIGURE 6.5 – Comparaison des profils de performance sur les instances `RLFAP`.

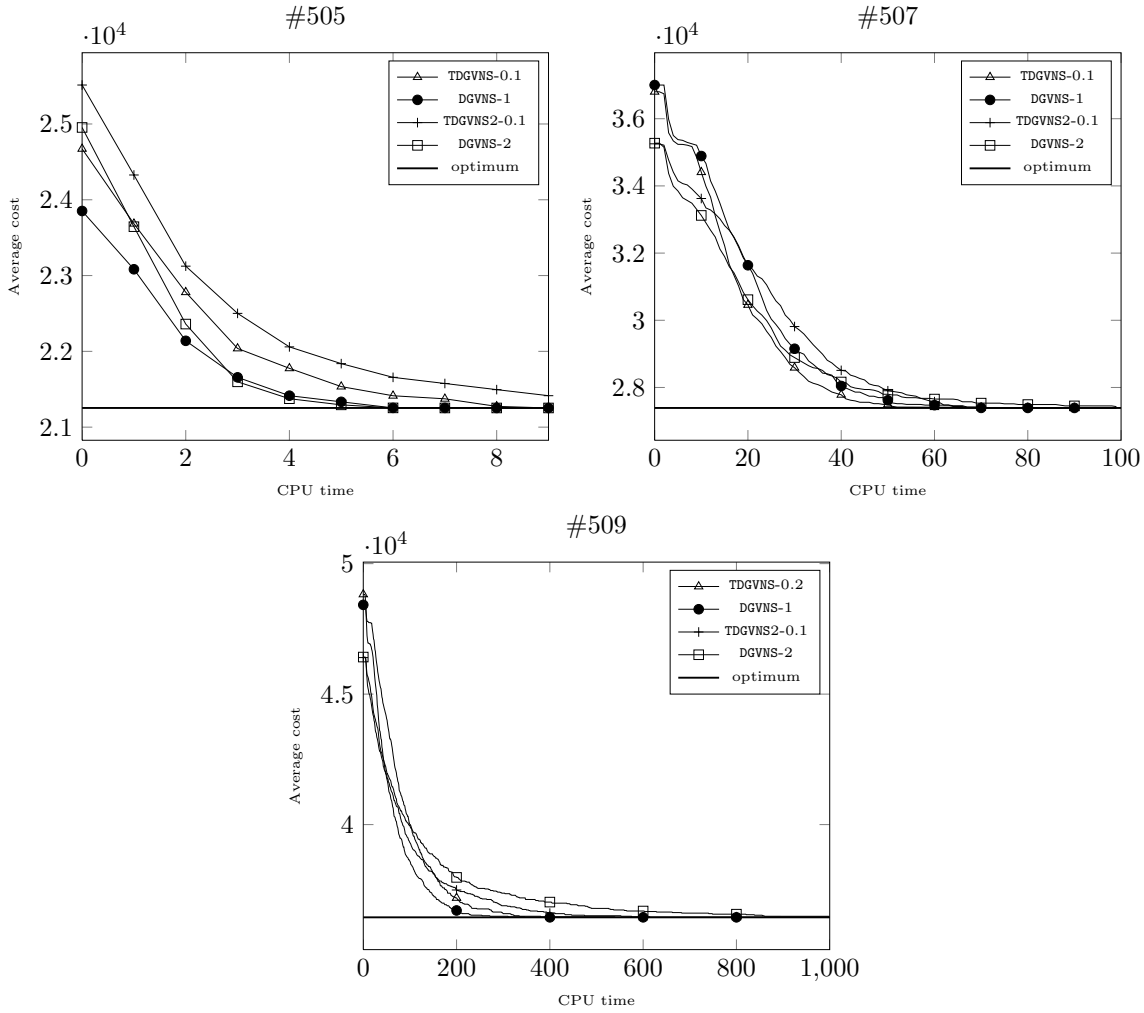


FIGURE 6.6 – Comparaison des profils de performance sur les instances `SPOT5`.

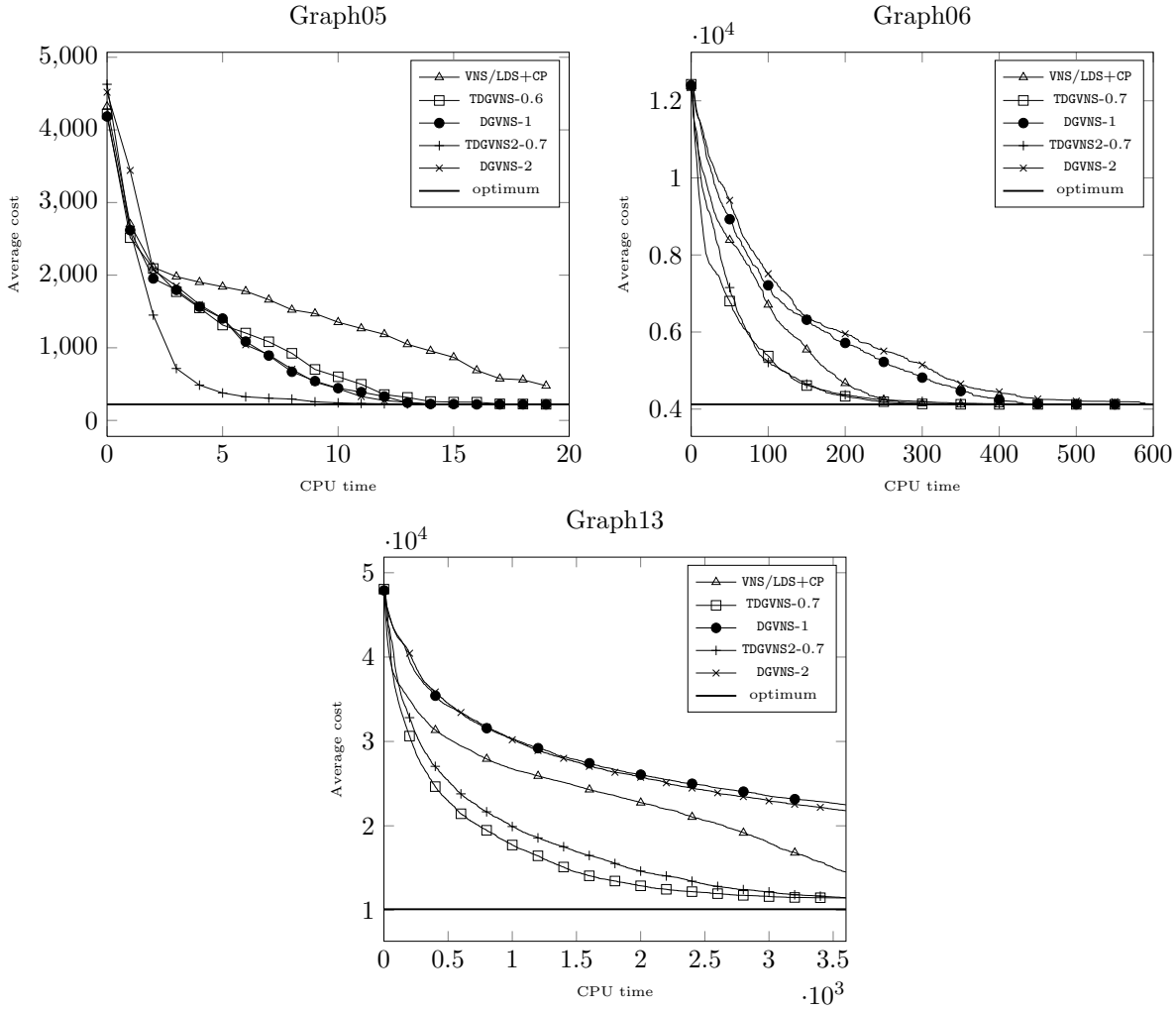


FIGURE 6.7 – Comparaison des profils de performance sur les instances `GRAPH`.

Instance	s_{max}	Succ.	Temps	Moy.	$\|C_T\|$	taille du cluster moy.	max.
Scen06	4	50/50	368	3,389	45	4.1	20
S^* =3,389	$+\infty$	**50/50**	**112**	**3,389**	55	4.9	12
Scen07	4	5/50	1,908	344,215	76	4.1	52
S^* =343,592	8	8/50	2,379	343,803	96	4.64	39
	12	7/50	2,400	343,601	100	4.85	35
	16	5/50	2,305	343,795	105	5.31	26
	24	5/50	2,272	348,640	109	5.72	23
	$+\infty$	**40/50**	**317**	**345,614**	110	5.72	23
Scen08	4	0/50	-	314 (309)	180	3.98	66
S^* =262	8	0/50	-	314 (309)	228	4.5	65
	12	0/50	-	313 (309)	243	4.88	43
	16	0/50	-	312 (309)	251	5.17	31
	24	0/50	-	312 (310)	258	5.49	27
	$+\infty$	**3/50**	**1,811**	**275**	259	5.51	27

TABLE 6.5 – Comparaison de DGVNS-1 pour différentes valeurs de s_{max} sur les instances RLFAP.

6.3.1.5 Bilan sur les apports de la TDTD

Nos expérimentations montrent clairement l'apport de la TDTD pour identifier des structures de voisinage portant sur les sous-parties du problème les plus difficiles à satisfaire. DGVNS-i peut alors tirer parti de ces informations afin de sélectionner des sous-ensembles de variables reliées par des fonctions de coût ayant un fort impact sur les solutions produites. De plus, pour les instances RLFAP, les valeurs de λ comprises entre 0.2 et 0.5 sont les plus appropriées, alors que pour les instances SPOT5, les meilleurs résultats sont obtenus avec des valeurs de λ comprises entre 0.1 et 0.3. En effet, ces instances restent très contraintes, car seulement un petit nombre de photographies sont séléctionnées dans une solution optimale. Pour les instances GRAPH, les meilleurs résultats sont obtenus avec des valeurs de λ élevées. En effet, en raison de la forte densité du graphe de contraintes, plusieurs fonctions de coût doivent être retirées afin d'obtenir des décompositions pertinentes.

6.3.2 Apports de la fusion basée sur s_{max}

Dans cette section, nous considérons différentes valeurs de s_{max} et nous comparons les performances respectives des DGVNS-1-s_{max} (notée DGVNS-s_{max}) sur les instances RLFAP, SPOT5, GRAPH et tagSNP. (s_{max}=$+\infty$) désigne le cas où aucune fusion n'est effectuée sur la décomposition initiale obtenue par MCS.

6.3.2.1 Instances RLFAP

Sur ces trois instances (cf. tableau 6.5), la fusion dégrade considérablement les performances de DGVNS-1 en taux de succès et de temps de calcul, comparé à ($s_{max} = +\infty$). Pour la Scen06, toutes les décompositions donnent 100% de succès mais c'est ($s_{max} = +\infty$) qui obtient les meilleurs temps de calcul. Notons que, pour cette instance, la taille du plus grand séparateur de la décomposition initiale est 8. Par conséquent, toutes les décompositions obtenues avec $s_{max}\geq 8$ sont identiques. Seuls les résultats de DGVNS-4 sont reportés. Pour la Scen07, DGVNS-8 obtient cinq fois moins de succès et le temps de calcul moyen est multiplié par 7. Enfin, pour la Scen08, DGVNS-s_{max} est incapable de trouver l'optimum. La meilleure solution trouvée est de coût 309.

Instance	s_{max}	Succ.	Temps	Moy.	$\|C_T\|$	taille du cluster moy.	taille du cluster max.
Graph05	4	0/50	-	6,028 (6,028)	5	22.8	96
S^* =221	8	0/50	-	3,413 (3,413)	24	10.04	76
	12	0/50	-	2,981 (2,981)	31	9.94	69
	16	0/50	-	2,981 (2,981)	32	10.13	68
	24	0/50	-	1,654 (1,654)	41	12.0	57
	32	50/50	14	221	58	17.45	33
	$+\infty$	**50/50**	**10**	**221**	58	17.45	33
Graph06	4	50/50	287	4,123	9	24.22	193
S^* =4,123	8	50/50	256	4,123	36	11.47	164
	12	50/50	262	4,123	41	11.32	160
	16	50/50	277	4,123	43	11.4	158
	24	50/50	263	4,123	49	12.43	151
	32	**50/50**	**230**	**4,123**	50	12.74	150
	$+\infty$	50/50	367	4,123	123	36.07	63
Graph11	4	0/50	-	31,286 (31,286)	8	45.5	331
S^* =3,080	8	0/50	-	24,685 (24,685)	45	13.6	292
	12	0/50	-	23,453 (23,453)	60	12.73	277
	16	0/50	-	23,353 (23,353)	63	12.75	274
	24	0/50	-	23,238 (23,186)	71	13.48	265
	32	0/50	-	23,144 (23,085)	73	13.9	263
	$+\infty$	**8/50**	**3,046**	**4,234**	191	65.04	118
Graph13	4	0/50	-	19,104 (12,132)	7	67.0	453
S^* =10,110	8	3/50	2,428	12,010	62	13.32	397
	12	2/50	2,893	12,053	74	12.77	385
	16	2/50	3,129	12,192	75	12.8	384
	24	1/50	3,473	11,999	83	13.42	376
	32	**7/50**	**3,247**	**11,766**	85	13.76	374
	$+\infty$	0/50	-	22,489 (18,639)	262	83.98	155

TABLE 6.6 – Comparaison de DGVNS-1 pour différentes valeurs de s_{max} sur les instances GRAPH.

6.3.2.2 Instances GRAPH

Les résultats varient selon les instances (cf. tableau 6.6). Pour les instances Graph05 et Graph11, la fusion dégrade considérablement les performances de DGVNS-1, plus particulièrement pour $s_{max} \leq 32$. En revanche, pour les deux autres instances, la fusion procure des gains substantiels. En effet, pour Graph06, DGVNS-4 améliore le temps de calcul de DGVNS-1 d'environ 21% comparé à (s_{max}=$+\infty$). Cette amélioration atteint 37% avec (s_{max}=32) (meilleure valeur de s_{max}). Pour Graph13, DGVNS-s_{max} atteint l'optimum pour la première fois (excepté pour s_{max}=4). Les meilleurs résultats sont obtenus avec (s_{max}=32) : 7/50 essais avec succès et une déviation moyenne à l'optimum d'environ 16%. Pour comparaison, VNS/LDS+CP (qui avait les meilleurs résultats sur cette instance) obtient 3 essais avec succès et une déviation moyenne à l'optimum de 122%.

6.3.2.3 Instances SPOT5

Sur les instances SPOT5 (cf. tableau 6.7), la fusion permet d'améliorer très sensiblement les performances de DGVNS-1 à la fois en taux de succès et en temps de calcul. Pour la plupart de ces instances (excepté pour #507), (s_{max}=8) donne les meilleurs résultats. En effet, cette valeur présente un meilleur compromis entre le nombre de clusters, la taille moyenne des clusters et la taille du plus grand cluster (cf. les colonnes 6-8, tableau 6.7). Pour les instances #412 (resp. #509), DGVNS-8 améliore le taux de succès de DGVNS-1 de 24% (resp. 14%).

Instance	s_{max}	Succ.	Temps	Moy.	$\|C_T\|$	Taille du cluster moy.	max.
#408	4	49/50	361	6,228	17	13.88	127
S^* =6,228	**8**	**49/50**	**78**	**6,228**	34	10.03	89
	12	44/50	461	6,228	48	10.17	76
	16	44/50	845	6,228	56	10.79	67
	24	43/50	460	6,228	62	11.52	65
	32	42/50	298	6,228	71	14.0	45
	$+\infty$	49/50	117	6,228	72	14.46	43
#412	**4**	**48/50**	**136**	**32,381**	16	21.0	228
S^* =32,381	8	48/50	241	32,381	34	13.18	132
	12	47/50	211	32,381	45	12.42	132
	16	43/50	365	32,381	51	12.65	101
	24	44/50	512	32,381	75	14.95	61
	32	42/50	619	32,381	89	17.02	58
	$+\infty$	36/50	84	32,381	96	19.04	45
#414	4	38/50	935	38,478	19	21.05	289
S^* =38,478	**8**	**45/50**	**532**	**38,478**	38	13.68	192
	12	45/50	721	38,478	49	12.88	192
	16	36/50	654	38,478	55	13.04	161
	24	35/50	594	38,478	73	14.53	159
	32	41/50	826	38,478	77	15.31	158
	$+\infty$	38/50	554	38,478	99	26.68	111
#505	4	50/50	99	21,253	27	11.15	174
S^* =21,253	8	50/50	212	21,253	50	9.0	71
	12	50/50	116	21,253	65	9.34	53
	16	49/50	218	21,253	82	10.38	42
	24	49/50	232	21,253	91	11.4	33
	$+\infty$	**50/50**	**63**	**21,253**	93	11.69	31
#507	4	32/50	565	27,390	20	17.55	235
S^* =27,390	8	35/50	544	27,390	39	12.05	145
	12	**38/50**	**663**	**27,390**	49	11.71	112
	16	28/50	542	27,390	60	12.32	111
	24	29/50	1,039	27,390	81	14.28	109
	32	30/50	1,248	27,390	86	15.21	108
	$+\infty$	33/50	71	27,390	101	20.48	68
#509	4	43/50	1,036	36,446	19	20.21	273
S^* =36,446	**8**	**47/50**	**438**	**36,446**	38	13.24	176
	12	45/50	814	36,446	50	12.52	145
	16	38/50	794	36,446	57	12.82	143
	24	29/50	1,039	36,446	73	14.23	143
	32	47/50	681	36,446	77	14.96	142
	$+\infty$	40/50	265	36,446	94	29.0	100

TABLE 6.7 – Comparaison de DGVNS-1 pour différentes valeurs de s_{max} sur les instances SPOT5.

6.3.2.4 Instances tagSNP

Sur les cinq instances tagSNP du tableau 6.8, la fusion dégrade considérablement les performances de DGVNS-1 aussi bien en taux de succès qu'en temps de calcul. Cette tendance s'amplifie pour les petites valeurs de s_{max} (≤ 12). Pour l'instance #8956, le taux de succès diminue de 8% (resp. 18%) pour s_{max}=32 (resp. s_{max}=24), et le temps de calcul augmente de 42% (resp. 46%) pour s_{max}=24 (resp. s_{max}=32). Pour $s_{max} \in \{12, 16\}$, le temps de calcul est multiplié par 2. On retrouve des comportements similaires sur les instances #14226 et #17034, pour lesquelles DGVNS-1 obtient les plus mauvais résultats, plus particulièrement pour les petites valeurs de s_{max}.

Les contre-performances de DGVNS-1 sur ces instances (cf. tableau 6.8) peuvent s'expliquer par le fait que la fusion basée sur s_{max} tend à augmenter les largeurs (w^-) des décompositions obtenues, en particulier pour de petites valeurs de s_{max}. Ainsi, l'impact de la de diversification de DGVNS s'en trouve fortement affaibli : les voisinages deviennent trop grands et leur nombre trop petit. Pour l'instance #6835, avec s_{max}=4, w^- passe de 108 à 468 et le nombre de clusters

Instance	s_{max}	Succ.	Temps	Moy.	$\|C_T\|$	`taille du cluster` moy.	max.
#6835	4	46/50	10,781	4,571,140	3	167	469
	8	46/50	9,925	4,571,122	3	167	469
	12	50/50	8,075	4,571,108	7	76.71	443
	16	50/50	7,966	4,571,108	7	76.71	443
	24	50/50	3,775	4,571,108	16	44.62	235
	32	50/50	3,321	4,571,108	23	39.26	229
	$+\infty$	**50/50**	**2,409**	**4,571,108**	**56**	**53.5**	**109**
#6858	4	0/50	-	26,882,903 (26,882,903)	2	496	989
	8	0/50	-	26,882,903 (26,882,903)	2	496	989
	12	0/50	-	26,882,862 (26,882,785)	3	334.33	951
	16	0/50	-	26,882,851 (26,882,785)	3	334.33	951
	24	0/50	-	26,882,836 (26,882,785)	6	177.66	945
	32	0/50	-	26,882,790 (26,882,697)	7	156.71	901
	$+\infty$	0/50	-	**26,882,588 (26,879,268)**	**105**	**156.23**	**351**
#8956	4	42/50	9,037	6,660,310	9	56.11	215
	8	46/50	7,109	6,660,309	14	38.42	215
	12	44/50	7,592	6,660,310	15	36.6	215
	16	42/50	7,580	6,660,310	16	35.12	215
	24	41/50	7,015	6,660,311	26	29.53	215
	32	46/50	7,192	6,660,309	30	29.33	215
	$+\infty$	**50/50**	**4,911**	**6,660,308**	**54**	**52.03**	**185**
#14226	4	26/50	11,973	26,131,481	20	54.7	237
	8	13/50	11,737	26,387,778	31	37.19	185
	12	28/50	11,657	25,758,768	38	32.21	185
	16	35/50	11,333	25,828,540	42	30.52	179
	24	46/50	10,930	25,712,052	58	27.48	179
	32	22/50	10,434	26,108,198	66	27.54	179
	$+\infty$	**46/50**	**7,606**	**25,688,751**	**94**	**38.19**	**159**
#17034	4	5/50	13,096	39,635,091	21	56.23	445
	8	2/50	11,979	39,665,758	30	41.26	445
	12	5/50	12,419	39,573,906	42	32.42	233
	16	14/50	12,149	39,359,507	55	27.98	231
	24	24/50	11,407	39,114,487	63	27.19	231
	32	32/50	11,058	38,838,844	79	27.12	229
	$+\infty$	**41/50**	**8,900**	**38,563,232**	**139**	**52.95**	**189**

TABLE 6.8 – Comparaison de `DGVNS`-1 pour différentes valeurs de s_{max} sur les instances `tagSNP`.

diminue de 56 à 3. Pour l'instance #6858, cette tendance est encore amplifiée : w^- augmente de 350 à 988 alors que le nombre de clusters diminue de 105 à 2.

À l'inverse, pour les huit instances du tableau 6.9, la fusion permet d'améliorer les performances de `DGVNS`-1. Pour la plupart de ces instances (excepté pour les instance #16706, #15757 et #9150), (s_{max}=32) donne les meilleurs résultats. Pour l'instance #16421, avec (s_{max}=32), le temps de calcul est divisé par 2 comparé à s_{max}=$+\infty$, tandis que pour l'instance #9315, le gain en temps de calcul est d'environ 29%. Pour les autres instances, ce gain est d'environ 12%. Enfin, pour l'instance #9150, ($s_{max} = 16$) permet une réduction de l'écart moyen à l'optimum de $3,3\%$ à $2,9\%$.

Sur ces instances, la fusion permet de supprimer les clusters qui se recouvrent fortement. Les voisinages obtenus sont plus pertinents (clusters faiblement connectés et de taille raisonnable), car la plupart des clusters fusionnés possèdent peu de variables propres. De plus, pour ces instances, la largeur de décomposition ainsi que le nombre de clusters varient peu (cf. colonnes 6-8 du tableau 6.9).

Instance	s_{max}	Succ.	Temps	Moy.	$\|C_T\|$	`taille du cluster`	
						moy.	max.
#3792	4	50/50	2,381	6,359,805	11	49.36	283
	8	50/50	1,498	6,359,805	20	29.8	255
	12	50/50	1,391	6,359,805	29	23.68	157
	16	50/50	959	6,359,805	37	21.64	139
	24	50/50	1,028	6,359,805	42	21.42	133
	32	**50/50**	**832**	**6,359,805**	**49**	**22.18**	**121**
	$+\infty$	50/50	954	6,359,805	70	30.37	95
#4449	4	50/50	2,399	5,094,256	7	67.85	305
	8	50/50	1,396	5,094,256	16	33	195
	12	50/50	1,216	5,094,256	17	31.70	195
	16	50/50	996	5,094,256	19	29.73	159
	24	50/50	663	5,094,256	26	27.07	121
	32	**50/50**	**587**	**5,094,256**	**49**	**22.18**	**121**
	$+\infty$	50/50	665	5,094,256	56	36.71	101
9150	4	0/50	-	50,699,638 (48,884,934)	27	51.7	936
	8	0/50	-	44,921,605 (43,302,157)	43	35.02	372
	12	0/50	-	44,977,064 (43,302,509)	45	33.91	372
	16	0/50	-	**44,587,379 (43,302,391)**	**52**	**31.38**	**370**
	24	0/50	-	46,178,649 (43,304,405)	62	29.68	302
	32	0/50	-	46,289,738 (43,302,644)	89	29.52	272
	$+\infty$	0/50	-	44,754,916 (43,302,028)	178	45.74	196
#9319	4	50/50	1,027	6,477,229	14	41.85	197
	8	50/50	853	6,477,229	22	28.72	173
	12	50/50	897	6,477,229	30	23.93	171
	16	50/50	698	6,477,229	35	22.48	115
	24	50/50	656	6,477,229	40	22	115
	32	**50/50**	**555**	**6,477,229**	**44**	**22.5**	**115**
	$+\infty$	50/50	788	6,477,229	62	35.42	90
#10442	4	25/50	6857	21664097	16	59.25	366
	8	42/50	5,094	21,591,975	31	34.0	326
	12	50/50	5,190	21,591,913	38	29.79	316
	16	50/50	6,220	21,591,913	45	27.47	306
	24	50/50	4,882	21,591,913	61	25.48	206
	32	**50/50**	**3,950**	**21,591,913**	**69**	**26.0**	**148**
	$+\infty$	50/50	4,552	21,591,913	104	43.12	134
#15757	4	50/50	195	2,278,611	13	27.61	215
	8	50/50	82	2,278,611	19	21	101
	12	**50/50**	**54**	**2,278,611**	**25**	**18.44**	**101**
	16	50/50	64	2,278,611	29	17.82	73
	24	50/50	72	2,278,611	41	18.31	73
	32	50/50	100	2,278,611	43	18.62	71
	$+\infty$	50/50	60	2,278,611	45	20.24	67
#16421	4	50/50	2,170	3,436,849	5	82.2	207
	8	50/50	2,273	3,436,849	7	60.14	201
	12	50/50	2,209	3,436,849	11	42.27	201
	16	50/50	2,369	3,436,849	14	36.28	133
	24	50/50	1,253	3,436,849	17	33.47	133
	32	**50/50**	**1,038**	**3,436,849**	**20**	**32.8**	**133**
	$+\infty$	50/50	2,673	3,436,849	35	42.54	113
#16706	4	50/50	131	2,632,310	11	41.36	129
	8	50/50	132	2,632,310	19	26.68	125
	12	50/50	105	2,632,310	33	19.66	55
	16	**50/50**	**99**	**2,632,310**	**37**	**19**	**51**
	24	50/50	124	2,632,310	46	18.91	49
	32	50/50	137	2,632,310	49	19.45	49
	$+\infty$	49/50	153	2,632,310	49	19.45	49

TABLE 6.9 – Comparaison de `DGVNS-1` pour différentes valeurs de s_{max} sur les instances `tagSNP`.

Instance	Méthode	Succ.	Temps	Moy.	$\|C_T\|$	Taille du cluster moy.	Taille du cluster max.	s_{max}
Scen06	DGVNS-2	50/50	146	3,389	55	4.9	12	8
S^* =3,389	DGVNS-1	50/50	112	3,389	55	4.9	12	8
	$s_{max} = 4$	50/50	368	3,389	45	4.1	20	4
	DGVNS-*abs*	**50/50**	**103**	**3,389**	**11**	**11.82**	**26**	**5**
Scen07	DGVNS-2	46/50	901	344,012	110	5.72	23	18
S^* =343,592	DGVNS-1	40/50	317	345,614	110	5.72	23	18
	$s_{max} = 8$	8/50	2,379	343,803	106	4.64	39	8
	DGVNS-*abs*	**50/50**	**514**	**343,592**	**17**	**14.82**	**49**	**8**
Scen08	DGVNS-2	5/50	595	277	259	5.51	27	25
S^* =262	DGVNS-1	3/50	1,811	275	259	5.51	27	25
	$s_{max} = 16$	0/50	-	312 (309)	251	5.17	31	16
	DGVNS-*abs*	**5/50**	**726**	**274**	**47**	**12.02**	**63**	**11**

TABLE 6.10 – Comparaison entre DGVNS-1, DGVNS-2, DGVNS-s_{max} et DGVNS-*abs* sur les instances RLFAP.

6.3.3 Apports de la fusion basée sur l'absorption

Dans cette section, nous comparons les performances de DGVNS-1, DGVNS-2 et DGVNS-1-*abs* (notée DGVNS-*abs*) sur les instances RLFAP, SPOT5, GRAPH et tagSNP.

6.3.3.1 Instances RLFAP

Sur les instances RLFAP (cf. tableau 6.10), DGVNS-*abs* surclasse DGVNS-1 et DGVNS-2. Pour la Scen06, les trois méthodes obtiennent 100% de succès ; toutefois, DGVNS-*abs* est plus rapide. Pour la Scen07, DGVNS-*abs* améliore le taux de succès de DGVNS-1 (resp. DGVNS-2) de 20% (resp. 8%). Pour la Scen08, DGVNS-*abs* obtient deux succès de plus que DGVNS-1 et le temps de calcul est divisé par 2. DGVNS-*abs* et DGVNS-2 obtiennent toutes les deux le même taux de succès, toutefois DGVNS-*abs* permet de réduire la déviation moyenne à l'optimum de (5.7% à 4.5%). Notons que DGVNS-2 est plus rapide.

6.3.3.2 Instances GRAPH

Sur les instances GRAPH (cf. tableau 6.11), l'apport de la fusion par absorption est très important, particulièrement sur les instances difficiles, Graph11 et Graph13, pour lesquelles on observe de fortes améliorations par rapport à DGVNS-1 et DGVNS-2. Pour Graph11, DGVNS-*abs* améliore le taux de succès de DGVNS-1 de 34% (de 16% à 50%) et réduit la déviation moyenne à l'optimum de (37.4% à 23%). Comparé à DGVNS-2, DGVNS-*abs* obtient deux succès de plus, toutefois, DGVNS-2 obtient des solutions de meilleure qualité en moyenne (déviation moyenne à l'optimum de 12.9% contre 23% pour DGVNS-*abs*). Pour Graph13, DGVNS-*abs* obtient un succès et réduit la déviation moyenne à l'optimum de moitié (pour DGVNS-1 : de 122% à 65%, pour DGVNS-2 : de 115% à 65%). Pour Graph06, les trois méthodes obtiennent 100% de succès, toutefois, DGVNS-*abs* est plus rapide (gain en temps de calcul de 29% et 37% par rapport à DGVNS-1 et DGVNS-2 respectivement).

6.3.3.3 Instances SPOT5

DGVNS-*abs* se montre très efficace sur les instances SPOT5 (cf. tableau 6.12), où il obtient 100% de succès sur **quatre instances parmi six**. Pour les instances de grande taille (#412, #414, #507 et #509), DGVNS-*abs* améliore le taux moyen de succès de DGVNS-1 (resp. DGVNS-2) de 25%

Instance	s_{max}	Succ.	Temps	Moy.	$\|C_T\|$	taille du cluster avg.	taille du cluster max.	s_{max}
graph05	DGVNS-2	**50/50**	**10**	**221**	**58**	**17.45**	**33**	**32**
S^* =221	DGVNS-1	**50/50**	**10**	**221**	**58**	**17.45**	**33**	**32**
	$s_{max} = 24$	0/50	-	1,654 (1,654)	41	12.0	57	21
	DGVNS-*abs*	50/50	17	221	3	41.33	81	13
graph06	DGVNS-2	50/50	413	4,123	123	36.07	63	61
S^* =4,123	DGVNS-1	50/50	367	4,123	123	36.07	63	61
	$s_{max} = 32$	**50/50**	**230**	**4,123**	**50**	**12.74**	**150**	**28**
	DGVNS-*abs*	**50/50**	**260**	**4,123**	**4**	**69.5**	**134**	**61**
graph11	DGVNS-2	23/50	3,031	3,480	191	65.04	118	116
S^* =3,080	DGVNS-1	8/50	3,046	4,234	191	65.04	118	116
	$s_{max} = 32$	0/50	-	23,144 (23,085)	73	13.9	263	31
	DGVNS-*abs*	**25/50**	**2,967**	**3,789**	**3**	**122**	**320**	**19**
graph13	DGVNS-2	0/50	-	21,796 (18,323)	262	83.98	155	153
S^* =10,110	DGVNS-1	0/50	-	22,489 (18,639)	262	83.98	155	153
	s_{max} =**32**	**7/50**	**3,247**	**11,766**	**85**	**13.76**	**374**	**29**
	DGVNS-*abs*	**1/50**	**3,520**	**16,756**	**5**	**134.2**	**292**	**153**

TABLE 6.11 – Comparaison entre DGVNS-1, DGVNS-2, DGVNS-s_{max} et DGVNS-*abs* sur les instances GRAPH.

(resp. 6%). Pour les autres instances, les trois méthodes obtiennent les mêmes taux de succès, mais DGVNS-*abs* est plus rapide.

6.3.3.4 Instances tagSNP

Une fois de plus, les résultats sont en faveur de DGVNS-*abs* (cf. tableau 6.13). Pour les instances de taille moyenne (excepté pour les instances #6835 et #8956), DGVNS-*abs* obtient de meilleurs temps de calcul que DGVNS-1. Pour les instances #16421 et #16706, DGVNS-*abs* est 2 à 3 fois plus rapide. Par ailleurs, DGVNS-*abs* surclasse nettement DGVNS-2 aussi bien en taux de succès (cf. les instances #6835 et #8956) qu'en temps de calcul. Pour les instances de grande taille #17034 et #6858, c'est encore DGVNS-*abs* qui obtient les meilleures performances. Pour l'instance #17034, DGVNS-*abs* améliore les taux succès de DGVNS-1 et DGVNS-2 de 12% et 24% respectivement. Pour l'instance #6858, bien qu'aucune des trois méthodes n'atteint l'optimum, DGVNS-*abs* obtient des solutions de meilleure qualité en moyenne : DGVNS-*abs* réduit la déviation moyenne à l'optimum de 28% à 33% pour DGVNS-1 et DGVNS-2. Pour les trois autres instances de grande taille (#10442, #9150 et #14226), les résultats de DGVNS-*abs* sont plus nuancés. Pour les instances #10442, DGVNS-*abs* dégrade le nombre de succès avec 84% d'essais à l'optimal. Pour l'instance #9150, DGVNS-*abs* obtient un meilleur écart moyen à l'optimum que DGVNS-2 (6.1 contre 6.6%) mais reste en deçà des performances de DGVNS-1 (3.3%). Enfin, pour l'instance #14226, DGVNS-*abs* dépasse nettement DGVNS-1, mais elle est moins rapide que DGVNS-2.

6.3.4 Comparaison des deux méthodes de fusion

Cette section compare DGVNS-*abs* et DGVNS-s_{max} (pour les valeurs de s_{max} ayant obtenues les meilleurs résultats).

6.3.4.1 Instances RLFAP

Les résultats sont en faveur de DGVNS-*abs* (cf. tableau 6.10). Pour la Scen06, DGVNS-*abs* est 3 fois plus rapide. Pour la Scen07, DGVNS-*abs* obtient 100% de succès contre 16% pour DGVNS-8 et divise le temps de calcul par 4. Pour la Scen08, DGVNS-abs obtient 5 essais avec succès et une déviation moyenne à l'optimum de 4% contre 19% pour DGVNS-16.

Instance	Méthode	Succ.	Temps	Moy.	$\|C_T\|$	Taille du cluster moy.	Taille du cluster max.	s_{max}
#408	DGVNS-2	50/50	81	6,228	72	14.46	43	34
S^* =6,228	DGVNS-1	49/50	117	6,228	72	14.46	43	34
	$s_{max} = 8$	49/50	78	6,228	34	10.03	89	8
	DGVNS-*abs*	**50/50**	**71**	**6,228**	**9**	**25.56**	**84**	**10**
#412	DGVNS-2	48/50	484	32,381	96	19.04	45	42
S^* =32,381	DGVNS-1	36/50	84	32,381	96	19.04	45	42
	$s_{max} = 4$	48/50	136	32,381	16	21.0	228	4
	DGVNS-*abs*	**50/50**	**84**	**32,381**	**8**	**41.25**	**137**	**10**
#414	DGVNS-2	45/50	670	38,478	99	26.68	111	110
S^* =38,478	DGVNS-1	38/50	554	38,478	99	26.68	111	110
	$s_{max} = 8$	45/50	532	38,478	38	13.68	192	8
	DGVNS-*abs*	**48/50**	**635**	**38,478**	**8**	**49.25**	**200**	**10**
#505	DGVNS-2	50/50	90	21,253	93	11.69	31	25
S^* =21,253	DGVNS-1	**50/50**	**63**	**21,253**	**93**	**11.69**	**31**	**25**
	$s_{max} = 4$	50/50	99	21,253	27	11.15	174	4
	DGVNS-*abs*	50/50	76	21,253	15	21.33	71	14
#507	DGVNS-2	45/50	463	27,390	101	20.48	68	65
S^* =27,390	DGVNS-1	33/50	71	27,390	101	20.48	68	65
	$s_{max} = 12$	38/50	663	27,390	49	11.71	112	12
	DGVNS-*abs*	**49/50**	**665**	**27,390**	**10**	**35.1**	**149**	**10**
#509	DGVNS-2	47/50	509	36,446	94	29.0	100	99
S^* =36,446	DGVNS-1	40/50	265	36,446	94	29.0	100	99
	s_{max} =**8**	**47/50**	**438**	**36,446**	**38**	**13.24**	**176**	**8**
	DGVNS-*abs*	**50/50**	**437**	**36,446**	**9**	**42.67**	**184**	**10**

TABLE 6.12 – Comparaison entre DGVNS-1, DGVNS-2, DGVNS-s_{max} et DGVNS-*abs* sur les instances SPOT5.

Les bonnes performances de DGVNS-*abs* sur ces instances peuvent s'expliquer par la qualité des décompositions produites par la fusion basée sur l'absorption. En effet, ces dernières présentent des clusters ayant une forte proportion de variables propres et contenant en moyenne une douzaine de variables. Par ailleurs, on observe une nette réduction de la valeur de s_{max}. Ceci renforce la pertinence du mécanisme de diversification de DGVNS, car les clusters sont moins redondants et se chevauchent beaucoup moins fortement. Ce n'est pas le cas pour les décompositions fournies par la fusion basée sur s_{max}, où de nombreux clusters partageant 100% de leurs variables sont encore conservés (cf. section 6.2.3). Par ailleurs, en raison du nombre peu élevé de clusters obtenus, différentes régions de l'espace de recherche sont couvertes beaucoup plus rapidement, permettant de converger vers la région contenant l'optimum global.

6.3.4.2 Instances SPOT5

Sur les instances SPOT5 (cf. tableau 6.12), DGVNS-*abs* obtient de légères améliorations aussi bien en taux de succèss qu'en temps de calcul. Notons que, pour l'instance #507, l'amélioration est beaucoup plus significative. DGVNS-*abs* obtient 98% d'essais avec succès contre 76% pour DGVNS-12 (gain de 22%). Sur les instances SPOT5, les mêmes constats dressés sur les instances RLFAP peuvent être effectués concernant la pertinence des décompositions produites par la fusion basée sur l'absorption. De plus, les largeurs des décompositions (w^-) obtenues restent comparables, voire nettement inférieures à celles produites par la fusion basée sur s_{max}. Par exemple, pour l'instance #412, w^- passe de 227 (avec $s_{max} = 4$) à 136 (avec l'absorption).

6.3.4.3 Instances GRAPH

Sur ces instances (cf. tableau 6.11), aucune des deux méthodes ne domine clairement l'autre. En effet, DGVNS-*abs* surclasse DGVNS-32 sur Graph11 (50% d'essais avec succès) et Graph05

Instance	s_{max}	Succ.	Temps	Moy.	$\|C_T\|$	Cluster size avg.	Cluster size max.	s_{max}
#3792	DGVNS-2	50/50	1,564	6,359,805	70	30.37	95	78
S^* =6,359,805	DGVNS-1	50/50	954	6,359,805	70	30.37	95	78
	$s_{max} = 32$	50/50	832	6,359,805	49	22.18	121	30
	DGVNS-*abs*	**50/50**	**602**	**6,359,805**	**12**	**50.17**	**140**	**18**
#4449	DGVNS-2	50/50	1,186	5,094,256	56	36.71	101	100
S^* =5,094,256	DGVNS-1	50/50	665	5,094,256	56	36.71	101	100
	DGVNS-32	50/50	587	5,094,256	49	22.18	121	32
	DGVNS-*abs*	**50/50**	**578**	**5,094,256**	**8**	**64.5**	**164**	**14**
#6835	DGVNS-2	34/50	3,568	4,730,484	56	54.5	110	104
S^* =4,571,108	DGVNS-1	**50/50**	**2,409**	**4,571,108**	**56**	**54.5**	**109**	**104**
	$s_{max} = 32$	50/50	3,321	4,571,108	23	39.26	229	32
	DGVNS-*abs*	50/50	2,947	4,571,108	4	140.0	262	36
#8956	DGVNS-2	34/50	5,138	6,660,383	54	52.03	185	180
S^* =6,660,308	DGVNS-1	**50/50**	**4,911**	**6,660,308**	**54**	**52.03**	**185**	**180**
	$s_{max} = 32$	46/50	7,192	6,660,309	30	29.33	215	30
	DGVNS-*abs*	50/50	6,555	6,660,308	6	88.17	216	23
#9319	DGVNS-2	50/50	1,248	6,477,229	62	35.42	90	86
S^* =6,477,229	DGVNS-1	50/50	788	6,477,229	62	35.42	90	86
	$s_{max} = \mathbf{32}$	**50/50**	**555**	**6,477,229**	**44**	**22.5**	**115**	**30**
	DGVNS-*abs*	50/50	738	6,477,229	10	59.9	181	9
#15757	DGVNS-2	50/50	127	2,278,611	45	20.24	67	60
S^* =2,278,611	DGVNS-1	50/50	60	2,278,611	45	20.24	67	60
	$s_{max} = \mathbf{12}$	**50/50**	**54**	**2,278,611**	**25**	**18.44**	**101**	**12**
	DGVNS-*abs*	**50/50**	**58**	**2,278,611**	**10**	**40.4**	**92**	**26**
#16421	DGVNS-2	50/50	2,001	3,436,849	35	42.54	113	110
S^* =3,436,849	DGVNS-1	50/50	2,673	3,436,849	35	42.54	113	110
	$s_{max} = \mathbf{32}$	**50/50**	**1,038**	**3,436,849**	**20**	**32.8**	**133**	**32**
	DGVNS-*abs*	50/50	1,114	3,436,849	6	74.0	134	16
#16706	DGVNS-2	50/50	127	2,632,310	49	19.45	49	30
S^* =2,632,310	DGVNS-1	49/50	153	2,632,310	49	19.45	49	30
	$s_{max} = \mathbf{16}$	**50/50**	**99**	**2,632,310**	**37**	**19**	**51**	**16**
	DGVNS-*abs*	**50/50**	**53**	**2,632,310**	12	41.33	94	14
#6858	DGVNS-2	0/50	-	26,882,824 (26,880,099)	105	156.23	351	260
S^* =20,162,249	DGVNS-1	0/50	-	26,881,172 (26,877,191)	105	156.23	351	260
	$s_{max} = 32$	0/50	-	26,882,790 (26,882,697)	7	156.71	901	32
	DGVNS-*abs*	0/50	-	**25,875,192 (23,524,246)**	**3**	**356**	**910**	**40**
#9150	DGVNS-2	0/50	-	46,123,257 (44,697,387)	178	45.74	196	168
S^* =43,301,891	DGVNS	0/50	-	44,754,916 (43,302,028)	178	45.74	196	168
	$s_{max} = 16$	0/50	-	**44,587,379 (43,302,391)**	**52**	**31.38**	**370**	**16**
	DGVNS-*abs*	0/50	-	45,955,657 (43,302,975)	18	79.33	380	14
#10442	DGVNS-2	48/50	8,257	21,591,915	104	43.12	134	114
$S^* = 21,591,913$	DGVNS-1	**50/50**	**4,552**	**21,591,913**	104	43.12	134	114
	$s_{max} = 32$	**50/50**	**3,950**	**21,591,913**	**69**	**26.0**	**148**	**32**
	DGVNS-*abs*	42/50	7,766	21,699,767	15	66.4	194	28
#14226	DGVNS-2	**50/50**	**8,237**	**25,665,437**	**94**	**38.19**	**159**	**150**
S^* =25,665,437	DGVNS-1	46/50	7,606	25,688,751	94	38.19	159	150
	$s_{max} = 24$	46/50	10,930	25,712,052	58	27.48	179	24
	DGVNS-*abs*	50/50	10,417	25,665,437	15	76.27	184	22
#17034	DGVNS-2	35/50	10,288	38,777,581	139	52.95	189	182
S^* =38,318,224	DGVNS-1	41/50	8,900	38,563,232	139	52.95	189	182
	$s_{max} = 32$	32/50	11,058	38,838,844	79	27.12	229	32
	DGVNS-*abs*	**47/50**	**13,108**	**38,318,226**	**15**	**80.73**	**237**	**14**

TABLE 6.13 – Comparaison entre DGVNS-1, DGVNS-2, DGVNS-s_{max} et DGVNS-*abs* sur les instances tagSNP.

(100% de succès), alors que sur Graph13, c'est DGVNS-32 qui obtient les meilleurs résultats en termes de taux de succès (gain de 12%) et en qualité des solutions obtenues (réduction de la déviation moyenne à l'optimum de 65.73% à 16.37%). Notons, que sur cette instance difficile, la fusion permet à DGVNS d'atteindre l'optimum pour la première fois. La contre-performance de DGVNS-*abs* sur Graph13 peut probablement s'expliquer par la taille moyenne des clusters qui est trop importante (134 variables en moyenne) et par un fort chevauchement entre clusters (s_{max}=153). Pour comparaison, les clusters issus de la fusion basée sur s_{max} contiennent en

moyenne 14 variables et (s_{max}=29).

6.3.4.4 Instances tagSNP

Pour les instances de taille moyenne (cf. tableau 6.13), DGVNS-*abs* obtient de meilleurs résultats **sur 5 instances parmi 8** (meilleur taux de succès sur l'instance #8956 et meilleurs temps de calcul sur quatre autres instances). Pour les trois autres instances (excepté pour #15757, où les résultats sont très similaires), c'est DGVNS-32 qui obtient les meilleurs temps de calcul. Pour deux instances de grande taille (#14226 et #17034), DGVNS-*abs* surclasse nettement DGVNS-32, avec un gain en taux moyen de succès de 12.5%. À l'inverse, pour les instances #9150 et #10442, DGVNS-*abs* est légèrement en retrait par rapport à DGVNS-32 et DGVNS-16. En effet, DGVNS-*abs* obtient 42 succès contre 50 pour l'instance #10442 et un écart moyen à l'optimal de $6,1\%$ contre 2.9% pour l'instance #9150. Enfin, pour l'instance #6858, DGVNS-*abs* obtient des solutions de meilleure qualité en moyenne, avec un écart moyen à l'optimum de 28% contre 33% pour DGVNS-32. De plus, la meilleure solution trouvée par DGVNS-*abs* présente une déviation de l'optimum de 16.6% contre 33.3% pour DGVNS-32.

6.3.4.5 Bilan sur les apports des deux méthodes de fusion

Sur l'ensemble des instances, la fusion de clusters procure des gains substantiels comparés aux résultats obtenus par DGVNS-1 et DGVNS-2 sur la décomposition initiale de MCS. Ces résultats confirment l'intérêt de réduire la redondance entre clusters pour identifier des structures de voisinage plus pertinentes et ainsi renforcer l'effort de diversification de DGVNS.

Par ailleurs, la fusion par absorption permet de produire des décompositions ayant une forte proportion de variables propres. De plus, la taille des clusters reste raisonnablement élevée. En effet, pour la plupart des instances (cf. colonnes 6-8 des différents tableaux), la taille des clusters est en moyenne 2 fois plus grande comparée à la décomposition initiale. De plus, le nombre de clusters reste relativement petit.

6.3.5 Profils de performance des rapports de temps et de succès

Les figures 6.8 et 6.9 comparent les courbes des profils de performances des rapports de temps et de succès de DGVNS-1, DGVNS-2, DGVNS-s_{max} et DGVNS-*abs* ainsi que leurs scores respectifs selon différents critères (cf. section 3.4.3). Pour chaque critère est associé un axe vertical sur lequel sont reportés les valeurs de ce critère pour différentes méthodes. De ces comparaisons, nous pouvons dresser les constats suivants :

- Sur cet ensemble de problèmes, DGVNS-*abs* surclasse nettement toutes les autres méthodes à la fois en temps et en termes de succès. En effet, la courbe du profil de performance de DGVNS-*abs* est systématiquement au-dessus de toutes les autres courbes (cf. figure 6.8(a)). Cela signifie que, pour toute valeur de $\tau \geq 1$, DGVNS-*abs* résout plus de problèmes que n'importe quel autre méthode. De plus, DGVNS-*abs* est plus rapide sur 50% des problèmes (cf. colonne 3, tableau de la figure 6.9) et obtient de meilleurs taux de succès sur environ 85% des problèmes (cf. colonne 6, tableau de la figure 6.9).
- DGVNS-*abs* obtient le meilleur score sur l'ensemble des critères (cf. figure 6.9), surclassant toutes les autres méthodes, particulièrement sur le critère robustesse (i.e. critère τ_{min}/τ^*) où DGVNS-*abs* est clairement la méthode la plus robuste.

– Comparé à DGVNS-2, DGVNS-1 obtient un meilleur profil de performance dans l'intervalle $[1, 10^{0.35}]$ (cf. figure 6.8(a)). Toutefois, pour $\tau \geq 10^{0.37}$, DGVNS-2 devient plus compétitive : DGVNS-2 met $10^{0.86}$ ($\simeq 7.24$) fois plus de temps que la méthode la plus rapide pour résoudre 100% des instances, alors que ce facteur est de $10^{1.1044}$ ($\simeq 12.72$) pour DGVNS-1. En termes de succès, la courbe du profil de performance de DGVNS-2 est largement au-dessus de celle de DGVNS-1 (cf. figure 6.8(b)). De plus, DGVNS-2 obtient le meilleure second score sur la plupart des critères (cf. figure 6.9).

– Enfin, bien que dans l'intervalle $[1, 10^{0.32}]$, le profil de DGVNS-s_{max} soit assez proche de celui de DGVNS-1 (cf. figure 6.8(a)), mais DGVNS-s_{max} reste moins efficace comparé DGVNS-1 et DGVNS-2 et obtient les plus mauvais scores sur la plupart des critères (cf. figure 6.9).

6.4 Conclusions

Nous avons présenté dans ce chapitre deux raffinements de la décomposition arborescente. Tout d'abord, nous avons proposé une première approche permettant de prendre en compte la dureté des fonctions de coût lors de la décomposition, et nous avons montré expérimentalement l'intérêt de cette approche comparé à DGVNS. Ces résultats démontrent clairement la pertinence de réunir, au sein d'un même cluster, des variables connectées par les fonctions de coût les plus difficiles à satisfaire.

Ensuite, nous avons proposé une seconde approche permettant de réduire la redondance entre clusters, dans le but de renforcer la qualité des structures de voisinage explorées par DGVNS. À cet effet, nous avons proposé deux critères. Le premier consiste à fusionner les clusters partageant plus de variables qu'un seuil maximal fixé s_{max}. Le second, basé sur la notion *d'absorption*, consiste à fusionner les clusters ayant un taux d'absorption supérieur à un seuil maximal fixé Ab_{max}.

Les expérimentations menées sur plusieurs instances difficiles montrent la pertinence et l'intérêt de la fusion par absorption pour renforcer la qualité des voisinages explorés. Par ailleurs, en limitant le recouvrement entre clusters, différentes régions de l'espace de recherche peuvent ainsi être couvertes beaucoup plus rapidement, permettant de converger vers la région contenant l'optimum global.

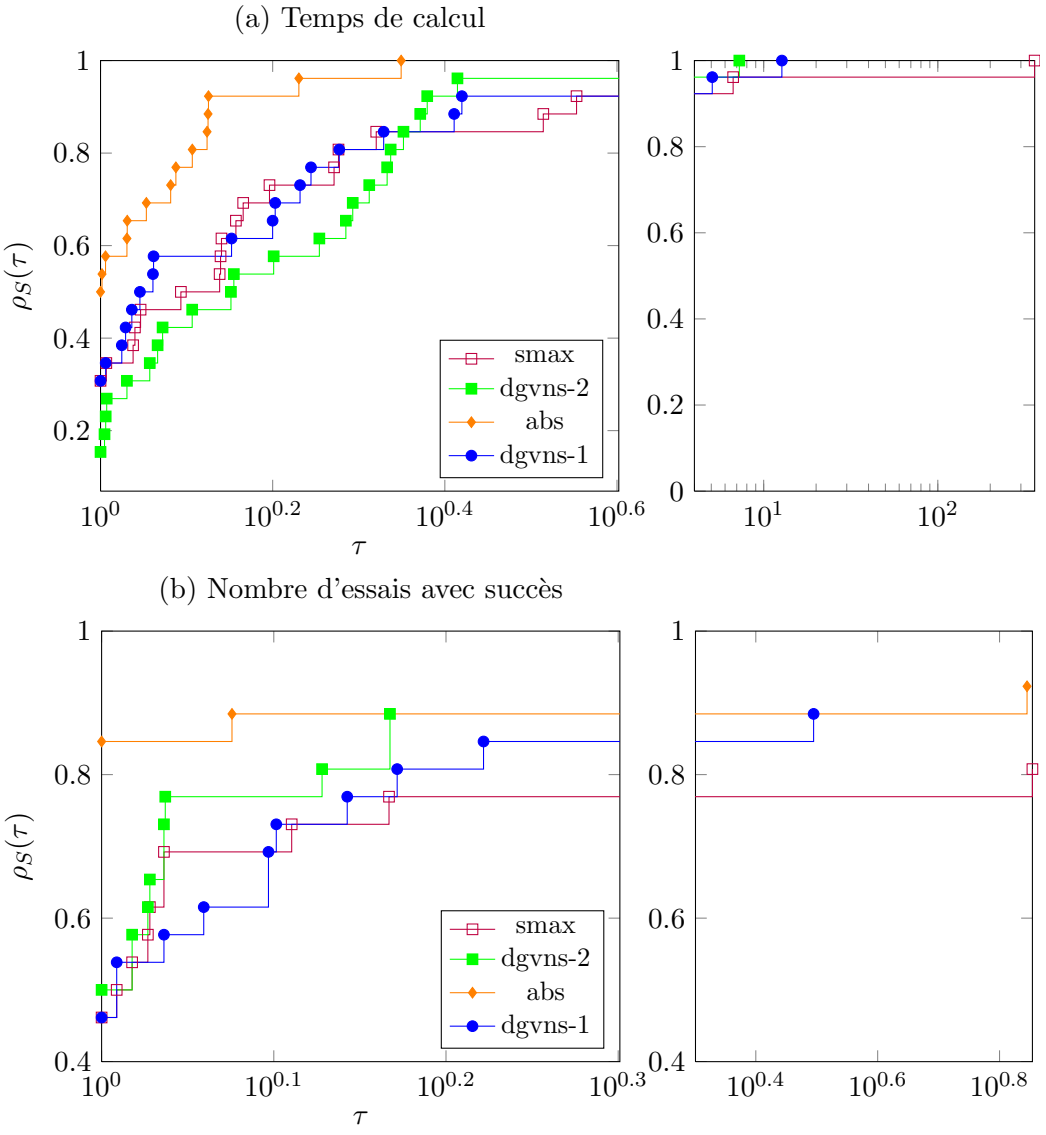


FIGURE 6.8 – Comparaison des profils de rapport de performance de DGVNS-1, DGVNS-2, DGVNS-s_{max} et DGVNS-*abs*.

	Temps			Succès		
methode	AUC	$\rho(1)$	$\frac{\tau_{min}}{\tau*}$	AUC	$\rho(1)$	$\rho(\tau_{max})$
DGVNS-2	0.998(2)	0.154(4)	0.31(2)	0.988(2)	0.5(2)	0.885(2)
DGVNS-1	0.998(2)	0.308(2)	0.18(3)	0.967(3)	0.462(3)	0.885(2)
DGVNS-s_{max}	0.96(4)	0.308(2)	0.01(4)	0.862(4)	0.462(3)	0.808(4)
DGVNS-abs	1.0(1)	0.5(1)	1.0(1)	1.0(1)	0.846(1)	0.923(1)

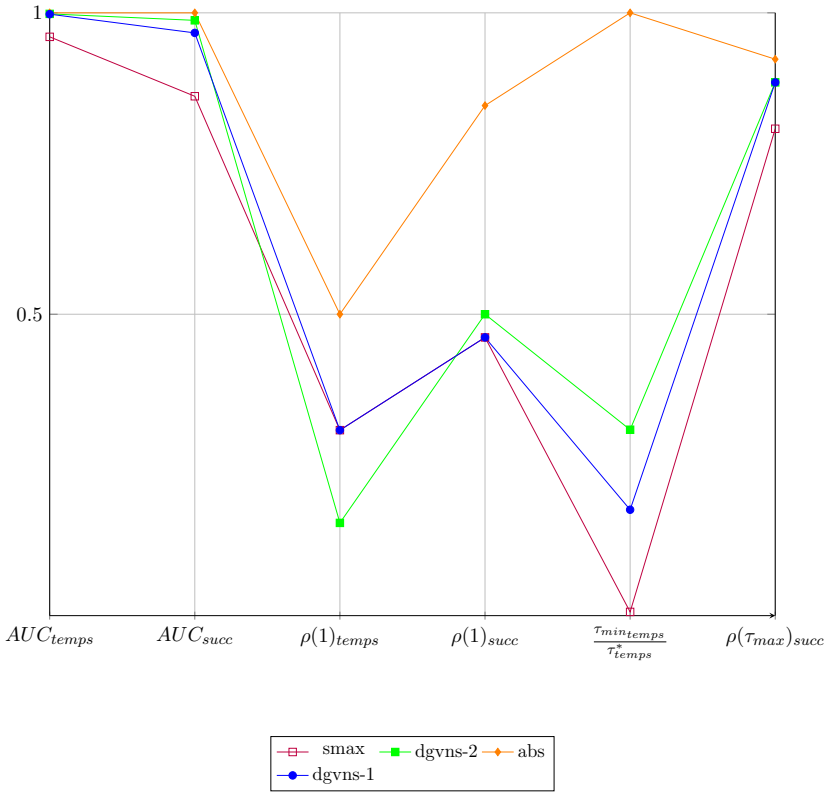


FIGURE 6.9 – Représentation graphique des valeurs des critères pour différentes méthodes.

Chapitre 7

VNS guidée par les séparateurs

Sommaire

Nous avons proposé un schéma générique (cf. chapitre 5) ainsi que plusieurs raffinements de la décomposition arborescente (cf. chapitre 6), permettant d'exploiter le graphe de clusters issu de la décomposition arborescente pour guider l'exploration des voisinages dans VNS. Bien que notre schéma tire parti de la topologie du graphe de clusters, il ne permet pas de prendre en compte les variables partagées entre ces clusters, appelées séparateurs. Ces variables jouent un rôle charnière dans la résolution du problème. En effet, l'affectation de toutes les variables des séparateurs scinde le problème initial en plusieurs sous-problèmes qui peuvent ensuite être résolus indépendamment. De plus, les compatibilités entre affectations des différents sous-problèmes dépendent uniquement des séparateurs. En effet, étant donné deux clusters C_i et C_j (avec $C_j \in \texttt{fils}(C_i)$), le sous-problème enraciné [9] en C_j dépend uniquement de l'affectation du séparateur $sep(C_i, C_j)$.

Dans ce chapitre, nous proposons deux extensions de DGVNS, notées SGVNS (*Separator-Guided* VNS) et ISGVNS (*Intensified* SGVNS), qui exploitent à la fois le graphe de clusters et les séparateurs entre ces clusters, afin de mieux cibler les régions du problème qui vont être explorées durant la recherche. Notre idée est de tirer parti des affectations des variables des séparateurs pour guider DGVNS vers les clusters qui sont les plus susceptibles de conduire à des améliorations importantes

9. Soit $(\mathcal{C}, T)$ une arborescence et $C_j \in \mathcal{C}$. Le sous-graphe $T[C_j \cup \texttt{Desc}(C_j)]$ est appelé sous-arborescence de T enraciné en C_j, avec $\texttt{Desc}(C_j)$ est l'ensemble des descendants de C_j dans T.

de la qualité de la solution courante (i.e. les clusters contenant, dans leurs séparateurs, des variables impliquées dans l'amélioration de la solution courante).

Plan du chapitre. Tout d'abord, nous présentons la méthode SGVNS (section 7.1), qui tire parti de l'évolution de la qualité de la solution au cours de la recherche pour privilégier les clusters contenant, dans leurs séparateurs, des variables impliquées dans l'amélioration de la solution courante. Puis, nous détaillons la méthode ISGVNS (section 7.2), qui vise à intensifier la recherche dans les clusters contenant des variables réinstanciées. Nous décrivons en section 7.3 les résultats de nos expérimentations et nous étudions l'intérêt d'exploiter la TDTD dans SGVNS et ISGVNS.

7.1 Separator-Guided VNS (SGVNS)

SGVNS exploite l'évolution de la qualité de la solution au cours de la recherche afin de guider l'effort de diversification de DGVNS vers des clusters contenant, dans leurs séparateurs, **au moins une variable réinstanciée** impliquée dans l'amélioration de la solution courante.

7.1.1 Pseudo-code de SGVNS

L'algorithme 27 présente le pseudo code de SGVNS. Son schéma général est similaire à celui de DGVNS (cf. algorithme 21 section 5.1). La différence principale, réside dans l'utilisation d'une **liste de prospection**, notée $\mathcal{C}and$, contenant les clusters à visiter en priorité. Cette liste est mise à jour à chaque fois que LDS+CP trouve une solution de meilleure qualité. L'objectif est de diriger l'exploration vers les clusters contenant, dans leurs séparateurs, des variables ayant conduit à l'amélioration de la solution courante.

Soit C_i le cluster courant et S' une solution de meilleure qualité trouvée par LDS+CP dans le voisinage de S. Soit V_c l'ensemble des variables qui ont été réinstanciées dans S pour obtenir S' (ligne 23) et C_w l'ensemble des clusters C_j de $\mathcal{C}and$ tel que $sep(C_i, C_j)$ partage au moins une variable avec V_c [10] ($C_j \cap V_c \neq \emptyset$, ligne 24). Au début, $\mathcal{C}and$ est initialisé à l'ensemble des clusters $\mathcal{C}_T$ de la décomposition arborescente (ligne 8).

Contrairement à DGVNS, la diversification de SGVNS est réalisée en considérant les clusters $C_j \in C_w$. En effet, à chaque fois que LDS+CP trouve une solution de meilleure qualité S' dans le voisinage de S (ligne 20), l'ensemble C_w est calculé (lignes 23-24), les clusters de C_w sont insérés en début de la liste $\mathcal{C}and$ (lignes 25-27) et le prochain cluster C_i à considérer est retiré de la liste (ligne 30). Comme pour DGVNS, S' devient la solution courante (ligne 21) et k est réinitialisé à k_{init} (ligne 22). À l'inverse, si aucune amélioration de la solution courante n'a été trouvée, k est incrémenté de 1 (ligne 29) et SGVNS considère le cluster courant (tête de la liste $Cand$ (ligne 30)).

Tout d'abord, la suppression du cluster courant C_i de $\mathcal{C}and$ permet de ne maintenir que la liste des clusters non encore visités, assurant ainsi une plus large couverture de l'espace de recherche. Ainsi, chaque cluster de $\mathcal{C}_T$ sera visité au moins une fois [11]. De plus, privilégier les clusters contenant, dans leurs séparateurs, au moins une variable réinstanciée permet de guider la recherche vers des régions pouvant mener à de plus larges améliorations de la solution courante. Enfin, cela permet de propager, au travers des séparateurs, les nouvelles affectations de S' vers les clusters $C_j \in C_w$, lors des prochaines itérations de SGVNS.

10. En effet, comme V_c contient que des variables de C_s (cf. ligne 10), si $C_j \cap V_c \neq \emptyset$, alors $sep(C_i, C_j) \cap V_c \neq \emptyset$.
11. Un cluster pourra être considéré plusieurs fois dès lors que $Cand$ devient vide, car réinitialisée à $\mathcal{C}_T$ (ligne 19).

Algorithme 27 : Pseudo-code de SGVNS

```
1  Fonction SGVNS (X, W, k_init, k_max, δ_max) ;
2  début
3      Soit G le graphe de contraintes de (X, W) ;
4      Soit (C_T, T) une décomposition arborescente de G ;
5      Cand ← C_T ;
6      S ← genSolInit() ;
7      k ← k_init ;
8      C_i ← RetirerDeTêteListe(Cand) ;
9      tant que (k < k_max) ∧ (notTimeOut) faire
10         C_s ← CompleteCluster(C_i, k) ;
11         X_un ← Hneighborhood(N_{k,i}, C_s, W, S) ;
12         A ← S\{(x_i, a) | x_i ∈ X_un} ;
13         S' ← LDS + CP(A, X_un, δ_max, f(S), S) ;
14         NeighbourhoodChangeSGVNS(S, S', k, i) ;
15     retourner S ;
16 Procédure NeighbourhoodChangeSGVNS(S, S', k, i) ;
17 début
18     si Cand = ∅ alors
19         Cand ← C_T ;
20     si f(S') < f(S) alors
21         S ← S' ;
22         k ← k_init ;
23         V_c ← {x | S'[x] ≠ S[x]} ;
24         C_w ← {C_j ∈ Cand | C_j ∩ V_c ≠ ∅} ;
25         Cand ← Cand\C_w ;
26         pour tout C_j ∈ C_w faire
27             InsérerEnTêteListe(Cand, C_j) ;
28     sinon
29         k ← k + 1 ;
30     C_i ← RetirerDeTêteListe(Cand) ;
```

7.2 Intensified Separator-Guided VNS (ISGVNS)

ISGVNS vise à intensifier l'exploration "autour" des clusters contenant des variables réinstanciées. À cet effet, une liste de propagation P_{List}, dotée d'une liste taboue dynamique T_{List} de taille L, est utilisée. La liste de propagation contient l'ensemble des clusters candidats à examiner après chaque amélioration de la solution courante. La liste taboue assure que les variables impliquées dans la sélection des clusters candidats (i.e. variables de V_c) ne seront pas reconsidérées dans $N_{k,i}$ par la fonction Hneighborhood, lors des L prochaines itérations de ISGVNS.

7.2.1 Pseudo-code de ISGVNS

L'algorithme 28 présente le pseudo-code de ISGVNS ; **L'intensification est réalisée en exploitant la liste de propagation**. Comme pour SGVNS, V_c désigne l'ensemble de toutes les variables qui ont été réinstanciées (ligne 21) et C_w est l'ensemble des clusters ayant au moins, dans leurs séparateurs, une variable de V_c (ligne 22). Contrairement à SGVNS, chaque cluster $C_j \in C_w$ est ajouté à P_{List} (ligne 24) et chaque variable $x \in V_c$ est rendue taboue pour les L

Algorithme 28 : Pseudo-code de `ISGVNS`

```
1  Fonction ISGVNS (X, W, k_init, k_max, δ_max) ;
2  début
3      Soit G le graphe de contraintes de (X, W) ;
4      Soit (C_T, T) une décomposition arborescente de G ;
5      S ← genSolInit() ;
6      k ← k_init ;
7      i ← 1 ;
8      P_List ← ∅ ;
9      tant que (k < k_max) ∧ (notTimeOut) faire
10         C_s ← CompleteCluster(C_i, k) ;
11         X_un ← Hneighborhood(N_{k,i}, C_s, W, S) ;
12         A ← S\{(x_i, a) | x_i ∈ X_un} ;
13         S' ← LDS + CP(A, X_un, δ_max, f(S), S) ;
14         NeighbourhoodChangeISGVNS(S, S', k, i) ;
15     retourner S ;
16 Procédure ChangeNeighborISGVNS(S, S', k, i) ;
17 début
18     si f(S') < f(S) alors
19         S ← S' ;
20         k ← k_init ;
21         V_c ← {x | S'[x] ≠ S[x]} ;
22         C_w ← {C_j | C_j ∩ V_c ≠ ∅, j = i + 1, ..., | C_T |} ;
23         pour tout C_j ∈ C_w faire
24             InsérerEnQueue (P_List, C_j) ;
25         rendre taboue chaque variable x ∈ V_c pour les L prochaines itérations ;
26     sinon
27         k ← k + 1 ;
28     si P_List est non vide alors
29         i ← RetirerDeTêteListe(P_List)
30     sinon
31         i ← succ(i)
```

prochaines itérations (ligne 25). La valeur de L est fixée à la taille de P_{List} afin d'éviter de réaffecter les variables de V_c tant que tous les clusters $C_j \in C_w$ n'ont pas été considérés. Enfin, le prochain cluster à examiner correpond au premier élément de P_{List}, si celle-ci n'est pas vide (ligne 29). Dans le cas contraire, le successeur de C_i dans $\mathcal{C}_T$ est considéré (ligne 31).

`ISGVNS` **permet un meilleur équilibre entre intensification et diversification**. En effet, tant qu'aucune amélioration n'est effectuée, `ISGVNS` se comporte comme `DGVNS-1`, en considérant successivement tous les clusters C_i. Cependant, dès que `LDS+CP` améliore la solution courante, `ISGVNS` bascule vers un schéma d'intensification, jusqu'à ce que tous les clusters de P_{List} aient été examinés.

7.3 Expérimentations

Dans cette section, nous reportons les résultats de nos expérimentations menées sur les instances `RLFAP`, `GRAPH`, `SPOT5` et `tagSNP`. Tout d'abord, nous comparons `SGVNS` et `ISGVNS` avec

Instance	Méthode	Succ.	Temps	Moy.
Scen06 S^* =3,389	SGVNS	50/50	111	3,389
	ISGVNS	**50/50**	**93**	**3,389**
	DGVNS-1	50/50	112	3,389
Scen07 S^* =343,592	SGVNS	48/50	652	343,802
	ISGVNS	**49/50**	**806**	**343,794**
	DGVNS-1	40/50	317	345,614
scen08 S^* =262	SGVNS	2/50	532	281
	ISGVNS	**4/50**	**1,408**	**276**
	DGVNS-1	3/50	1,811	275

TABLE 7.1 – Comparaison entre SGVNS, ISGVNS et DGVNS-1 sur les instances RLFAP.

Instance	Méthode	Succ.	Temps	Moy.
#408 S^* =6,228	SGVNS	**50/50**	**132**	**6,228**
	ISGVNS	50/50	140	6,228
	DGVNS-1	49/50	117	6,228
#412 S^* =32,381	SGVNS	48/50	351	32,381
	ISGVNS	**49/50**	**434**	**32,381**
	DGVNS-1	36/50	84	32,381
#414 S^* =38,478	SGVNS	**48/50**	**630**	**38,478**
	ISGVNS	46/50	524	38,478
	DGVNS-1	38/50	554	38,478
#505 S^* =21,253	SGVNS	50/50	99	21,253
	ISGVNS	50/50	67	21,253
	DGVNS-1	**50/50**	**63**	**21,253**
#507 S^* =27,390	SGVNS	39/50	576	27,390
	ISGVNS	**45/50**	**694**	**27,390**
	DGVNS-1	33/50	71	27,390
#509 S^* =36,446	SGVNS	**48/50**	**412**	**36,446**
	ISGVNS	46/50	499	36,446
	DGVNS-1	40/50	265	36,446

TABLE 7.2 – Comparaison entre SGVNS, ISGVNS et DGVNS-1 sur les instances SPOT5.

DGVNS-1 (cf. sections 7.3.1 et 7.3.2), puis nous comparons SGVNS avec ISGVNS (cf. section 7.3.3). Ensuite, nous étudions les apports de la TDTD sur SGVNS et ISGVNS (cf. section 7.3.5). Enfin, nous etudions les profils de rapport de performances comparant SGVNS, ISGVNS, DGVNS-1 et DGVNS-2. Nous noterons par TD-SGVNS-λ et TD-ISGVNS-λ, les versions TDTD de SGVNS et ISGVNS.

7.3.1 Comparaison entre SGVNS et DGVNS-1

7.3.1.1 Instances RLFAP

Pour les instances RLFAP (cf. tableau 7.1), SGVNS surclasse clairement DGVNS-1 sur Scen07, et reste très comparable sur les deux autres instances. Pour la Scen07, SGVNS améliore le taux de succès de DGVNS-1 de 16% (de 80% à 96%), et réduit la déviation moyenne à l'optimum de (0.58% à 0.06%).

7.3.1.2 Instances SPOT5

Sur les instances SPOT5 (cf. tableau 7.2), SGVNS se montre très efficace comparé à DGVNS-1, particulièrement sur les instances de grande taille (#412, #414, #507 et #509), pour lesquelles l'amélioration est très significative. Pour les instances #507 et #509, SGVNS procure un gain en taux de succès de 12% (de 66% à 78%) et 16% (de 80% à 96%) respectivement. Ces gains atteignent respectivement 24% et 20% pour les instances #412 et #414. Sur les autres instances, les deux méthodes obtiennent des résultats similaires.

Instance	Méthode	Succ.	Temps	Moy.
Graph05	SGVNS	50/50	19	221
S^* =221	ISGVNS	50/50	16	221
	DGVNS-1	**50/50**	**10**	**221**
	VNS/LDS+CP	50/50	17	221
Graph06	SGVNS	50/50	292	4,123
S^* =4,123	ISGVNS	50/50	240	4,123
	DGVNS-1	50/50	367	4,123
	VNS/LDS+CP	**50/50**	**218**	4,123
Graph11	SGVNS	27/50	3,026	4,238
S^* =3,080	ISGVNS	39/50	2,762	3,349
	DGVNS-1	8/50	3,046	4,234
	VNS/LDS+CP	**44/50**	2,403	3,090
Graph13	SGVNS	**6/50**	**3,260**	**14,707**
S^* =10,110	ISGVNS	1/50	3,196	17,085
	DGVNS-1	0/50	-	22,489 (18,639)
	VNS/LDS+CP	**3/50**	3,477	14,522

TABLE 7.3 – Comparaison entre SGVNS, ISGVNS et DGVNS-1 sur les instances GRAPH.

7.3.1.3 Instances GRAPH

La supériorité de SGVNS se confirme sur les instances GRAPH (cf. tableau 7.3). Pour les instances faciles (Graph05 et Graph06), les deux méthodes obtiennent les mêmes taux de succès (i.e. 100%). Toutefois, SGVNS est plus rapide que DGVNS-1 sur Graph06 (gain en temps de calcul d'environ 20%). Pour les instances difficiles, les gains obtenus par SGVNS sont plus importants. Pour l'instance Graph11, SGVNS obtient 3 fois plus de succès que DGVNS-1 (gain de 38% en termes de taux de succès). Toutefois, ces résultats restent en deçà de ceux obtenus par VNS/LDS+CP (44 succès contre 27 pour SGVNS). Pour l'instance Graph13, SGVNS atteint l'optimum (6 succès), surclassant nettement DGVNS-1 et VNS/LDS+CP (SGVNS obtenant 2 fois plus de succès que VNS/LDS+CP). Pour comparaison, les meilleurs résultas obtenus jusqu'à présent sont dûs à DGVNS-1 avec s_{max}=32 (7 succès). De plus, SGVNS n'exploite aucun raffinement de la décomposition initiale fournie par MCS.

Ceci démontre clairement la pertinence d'exploiter les séparateurs.

7.3.1.4 Instances tagSNP

Pour les instances de taille moyenne (cf. tableau 7.4), SGVNS et DGVNS-1 atteignent l'optimum sur chacun des 50 essais. Cependant, SGVNS est plus rapide sur trois instances (#4449, #9313 et #16421), plus lent sur quatre autres instance (#3792, #6835, #8956, et #15757), et très similaire à DGVNS-1 sur l'instance #16706. Pour l'instance #16421, SGVNS améliore de 24% le temps de calcul de DGVNS-1. Le meilleur résultat est obtenu sur l'instance #9313, où SGVNS améliore le temps de calcul de DGVNS-1 de 36%. Les moins bons résultats sont obtenus sur l'instance #8956, pour laquelle DGVNS-1 est plus rapide.

Pour les instances de grande taille (cf. tableau 7.5), SGVNS se montre moins compétitive en taux de succès et en temps de calcul, en particulier sur les trois instances #10442, #14226 et #17034, où DGVNS-1 est clairement le meilleur. Pour l'instance #17034, DGVNS-1 améliore le taux de succès de 16% (de 66% à 82%), la déviation moyenne à l'optimum est réduite (de 1.43% à 0.63%) et DGVNS est 1.2 fois plus rapide que SGVNS. Pour l'instance #14226, le gain en taux de succès atteint 12% (de 80% à 92%) et DGVNS-1 réduit légèrement la déviation moyenne à l'optimum (de 0.54% à 0.09%). Pour les autres instances (#6858 et #9150), les deux méthodes restent comparables.

Instance	Méthode	Succ.	Temps	Moy.
#3792	DGVNS-1	50/50	954	6,359,805
$S^* = 6,359,805$	SGVNS	50/50	1,033	6,359,805
	ISGVNS	**50/50**	**853**	**6,359,805**
#4449	DGVNS-1	50/50	665	5,094,256
$S^* = 5,094,256$	SGVNS	**50/50**	**661**	**5,094,256**
	ISGVNS	50/50	675	5,094,256
#6835	DGVNS	**50/50**	**2,409**	**4,571,108**
$S^* = 4,571,108$	SGVNS	50/50	4,061	4,571,108
	ISGVNS	50/50	3,452	4,571,108
#8956	DGVNS-1	50/50	4,911	6,660,308
$S^* = 6,660,308$	SGVNS	50/50	5,483	6,660,308
	ISGVNS	**50/50**	**4,118**	**6,660,309**
#9319	DGVNS-1	50/50	788	6,477,229
$S^* = 6,477,229$	SGVNS	**50/50**	**500**	**6,477,229**
	ISGVNS	50/50	672	6,477,229
#15757	DGVNS-1	**50/50**	**60**	**2,278,611**
$S^* = 2,278,611$	SGVNS	50/50	104	2,278,611
	ISGVNS	50/50	80	2,278,611
#16421	DGVNS-1	50/50	2,673	3,436,849
$S^* = 3,436,849$	SGVNS	**50/50**	**2,025**	**3,436,849**
	ISGVNS	50/50	5,863	3,436,849
#16706	DGVNS-1	50/50	153	2,632,310
$S^* = 2,632,310$	SGVNS	50/50	159	2,632,310
	ISGVNS	**50/50**	**89**	**2,632,310**

TABLE 7.4 – Comparison entre SGVNS, ISGVNS et DGVNS-1 sur les instances tagSNP de taille moyenne.

7.3.2 Comparaison entre ISGVNS et DGVNS-1

Instances RLFAP.

ISGVNS domine clairement DGVNS-1, notamment sur les deux instances difficiles Scen07 et Scen08. Pour Scen07, ISGVNS améliore le taux de succès de DGVNS-1 de 18% (de 80% à 98%). Pour Scen08, ISGVNS obtient un succès de plus que DGVNS-1 et réduit le temps de calcul d'environ 22%. Pour Scen06, ISGVNS est plus rapide que DGVNS-1.

Instances SPOT5.

ISGVNS est meilleur que DGVNS-1 sur la plupart des instances (cf. tableau 7.2). Pour les instances de grande taille (#412, #414, #507 et #509), ISGVNS obtient un taux moyen de succès de 93% contre 73.5% pour DGVNS-1. Pour les autres instances (#408 et #505), les deux méthodes obtiennent les mêmes taux de succès, mais DGVNS-1 est plus rapide.

Instances GRAPH.

Une fois de plus, ISGVNS surclasse nettement DGVNS-1, particulièrement sur les deux instances Graph11 et Graph13 (cf. tableau 7.3). Pour l'instance Graph11, ISGVNS obtient 5 fois plus de succès que DGVNS-1, se rapprochant ainsi des résultats de VNS/LDS+CP. En revanche, pour l'instance Graph13, bien que ISGVNS atteint une fois l'optimum, les résultats sont moins bons comparé à VNS/LDS+CP. Pour les instances faciles, ISGVNS est plus rapide que DGVNS-1.

Instances tagSNP.

Pour les instances de taille moyenne (cf. tableau 7.4), ISGVNS est plus rapide que DGVNS-1 sur quatre instances (#3792, #8956, #9313 et #16706), plus lent sur trois instances (#6858,

Instance	Méthode	Succ.	Temps	Moy.
#6858, $S^* = 20,162,249$	DGVNS-1	0/50	-	26,882,588 (26,879,268)
	SGVNS	0/50	-	26,881,890 (26,878,275)
	ISGVNS	0/50	-	**26,880,507 (26,876,673)**
#9150 $S^* = 43,301,891$	DGVNS	0/50	-	**44,754,916 (43,302,028)**
	SGVNS	0/50	-	45,425,451 (43,304,752)
	ISGVNS	0/50	-	46,179,187 (43,304,479)
#10442 $S^* = 21,591,913$	DGVNS-1	**50/50**	**4,552**	**21,591,913**
	SGVNS	50/50	7,153	21,591,913
	ISGVNS	50/50	7,291	21,591,913
#14226 $S^* = 25,665,437$	DGVNS-1	46/50	7,606	25,688,751
	SGVNS	40/50	7,646	25,805,242
	ISGVNS	**50/50**	**9,596**	**25,665,437**
#17034 $S^* = 38,318,224$	DGVNS-1	**41/50**	**8,900**	**38,563,232**
	SGVNS	33/50	10,212	38,869,514
	ISGVNS	36/50	10,579	38,746,957

TABLE 7.5 – Comparison entre SGVNS, ISGVNS et DGVNS-1 sur les instances tagSNP de grande taille.

#15757 et #16421) et similaire à DGVNS-1 sur l'instance #4449. Pour les instances #8956 et #9313, ISGVNS améliore le temps de calcul de DGVNS-1 d'environ 16% en moyenne. Les meilleurs résultats sont obtenus sur l'instance #16706, pour laquelle le gain est d'environ 41%. En revanche, pour l'instance #16421, ISGVNS est deux fois plus lent que DGVNS-1.

Pour les instances de grande taille, ISGVNS est plus efficace que DGVNS-1 sur l'instance #14226 et est moins compétitive sur l'instance #17034. Les deux méthodes obtiennent les mêmes taux de succès sur l'instance #10442, mais DGVNS-1 est plus rapide. Pour les autres instances, aucune méthode ne domine l'autre. En effet, si on compare la qualité moyenne des solutions obtenues, ISGVNS est meilleur sur l'instance #6858 alors que DGVNS-1 obtient les meilleurs résultats sur l'instance #9150.

7.3.3 Comparaison entre SGVNS et ISGVNS

Instances RLFAP.

ISGVNS devance SGVNS sur toutes les instances (cf. tableau 7.1). Pour Scen06, ISGVNS est plus rapide en moyenne. Pour Scen07, ISGVNS obtient un succès de plus que SGVNS. Pour Scen08, ISGVNS obtient **deux fois plus de succès** que SGVNS et des solutions de meilleure qualité en moyenne.

Instances SPOT5.

Sur les instances SPOT5, aucune des deux méthodes ne domine l'autre (cf. tableau 7.2). En effet, SGVNS obtient 2 succès de plus que ISGVNS sur deux instances (#414 et #509) et est plus rapide sur l'instance #408, tandis que ISGVNS devance SGVNS en taux de succès sur deux instances (#412 et #507) et est plus rapide sur l'instance #505.

Instances GRAPH.

Sur les instances faciles (Graph05 et Graph06), ISGVNS est légèrement plus rapide que SGVNS (cf. tableau 7.3). En revanche, sur les deux instances difficiles, aucune méthode ne surclasse nettement l'autre. Pour l'instance Graph11, ISGVNS obtient 24% de succès en plus par rapport à SGVNS et réduit la déviation moyenne à l'optimum de (37% à 8%). Pour l'instance Graph13,

SGVNS multiplie le nombre de succès de ISGVNS par 6 et réduit la déviation moyenne à l'optimum de (69% à 45%).

Instances tagSNP.

Pour les instances de taille moyenne, SGVNS et ISGVNS obtiennent les mêmes taux de succès. Toutefois, ISGVNS est plus rapide que SGVNS sur 5 instances (#3792, #6858, #8956, #15757 et #16706), et est moins rapide sur 3 autres instances. Pour les instances de grande taille, ISGVNS surclasse nettement SGVNS. ISGVNS améliore le taux moyen de succès de SGVNS d'environ 13% sur deux instances (#14226 et #17034) et obtient des solutions de meilleure qualité sur l'instances #6858, tandis que SGVNS est légèrement plus rapide sur l'instance #10442 et obtient des solutions de meilleure qualité sur l'instance #9150.

7.3.4 Bilan sur les apports de SGVNS et ISGVNS

Nous avons montré expérimentalement l'intérêt d'exploiter les séparateurs pour mieux guider DGVNS-1 vers des voisinages susceptibles de conduire à des améliorations importantes de la qualité de la solution. Sur la plupart des instances testées, SGVNS et ISGVNS surclassent nettement DGVNS-1, excepté sur les instances tagSNP de grande taille, où SGVNS est moins efficace que DGVNS-1. SGVNS reste toutefois plus compétitive sur les instances de taille moyenne. De ces expérimentations, il ressort également que ISGVNS donne de meilleurs résultats en moyenne que SGVNS. ISGVNS surclasse SGVNS sur 17 instances parmi 26, alors que SGVNS obtient de meilleurs résultats sur 9 instances.

7.3.5 Apports de la TDTD pour SGVNS et ISGVNS

Instances RLFAP.

L'apport de la TDTD pour SGVNS et ISGVNS est significatif (cf. tableau 7.6), particulièrement sur les instances Scen07 et Scen08, pour lesquelles on observe de fortes améliorations. Pour la Scen06, TD-ISGVNS-0.2 (resp. TD-SGVNS-0.2) améliore le temps de calcul de ISGVNS (resp. SGVNS) de 27% (resp. 14%). Pour la Scen07, TD-ISGVNS-0.4 et TD-SGVNS-0.4 obtiennent toutes les deux 100% de succès et améliorent considérablement les temps de calcul de SGVNS et ISGVNS. Enfin, pour la Scen08, TD-ISGVNS-0.4 obtient 2 succès de plus que ISGVNS et réduit la déviation moyenne à l'optimum de 5.3% à 4.5%. TD-SGVNS-0.4 obtient 4 fois plus de succès que SGVNS et réduit la déviation moyenne à l'optimum de (7.25% à 3.8%). Notons également que les seuils de λ donnant les meilleurs résultats pour TD-ISGVNS et TD-SGVNS sont identiques à ceux constatés pour les différents DGVNS-i (cf. section 6.3.1).

Instances SPOT5.

L'apport de la TDTD est plus important pour SGVNS. TD-SGVNS obtient de meilleurs résultats sur 4 instances (#408, #414, #505 et #507) (cf. tableau 7.7). Pour l'instance #408, la TDTD permet de diviser le temps de calcul de SGVNS par 2.5. Pour l'instance #507, TD-SGVNS améliore le taux de succès de SGVNS de 6%. Pour les deux instances #412 et #509, SGVNS obtient les meilleurs résultats. En revanche, l'impact de la TDTD sur ISGVNS est moins significatif. ISGVNS est meilleur que TD-ISGVNS sur 4 instances (#408, #412, #505 et #507) (cf. tableau 7.7). Pour l'instance #412, ISGVNS est 1.5 plus rapide. Pour l'instance #507, la version TDTD dégrade le taux de succès de ISGVNS de 16% (37 essais avec succès contre 45 pour ISGVNS). Notons toutefois les bonnes performances de TD-ISGVNS sur les deux instances #414, #509.

Instance	Méthode	Succ.	Temps	Moy.
Scen06	ISGVNS	50/50	93	3,389
S^* =3,389	TD-ISGVNS-0.2	**50/50**	**67**	**3,389**
	SGVNS	50/50	111	3,389
	TD-SGVNS-0.2	**50/50**	**95**	**3,389**
Scen07	ISGVNS	49/50	806	343,794
S^* =343,592	TD-ISGVNS-0.4	**50/50**	**461**	**343,592**
	SGVNS	48/50	652	343,802
	TD-SGVNS-0.4	**50/50**	**529**	**343,592**
Scen08	ISGVNS	4/50	1,408	276
S^* =262	TD-ISGVNS-0.4	**6/50**	**852**	**274**
	SGVNS	2/50	532	281
	TD-SGVNS-0.4	**8/50**	**534**	**272**

TABLE 7.6 – Comparaison entre TD-SGVNS, TD-ISGVNS, SGVNS et ISGVNS sur les instances RLFAP.

Instance	Méthode	Succ.	Temps	Moy.
#408	ISGVNS	**50/50**	**140**	**6,228**
S^* =6,228	TD-ISGVNS-0.1	50/50	183	6,228
	SGVNS	50/50	132	6,228
	TD-SGVNS-0.1	**50/50**	**50**	**6,228**
#412	ISGVNS	**49/50**	**434**	**32,381**
S^* =32,381	TD-ISGVNS-0.1	48/50	286	32,381
	SGVNS	**48/50**	**351**	**32,381**
	TD-SGVNS-0.1	47/50	392	32,381
#414	ISGVNS	46/50	524	38,478
S^* =38,478	TD-ISGVNS-0.2	**46/50**	**481**	**38,478**
	SGVNS	48/50	630	38,478
	TD-SGVNS-0.1	**49/50**	**581**	**38,478**
#505	ISGVNS	**50/50**	**67**	**21,253**
S^* =21,253	TD-ISGVNS-0.1	50/50	90	21,253
	SGVNS	50/50	99	21,253
	TD-SGVNS-0.1	**50/50**	**91**	**21,253**
#507	ISGVNS	**45/50**	**694**	**27,390**
S^* =27,390	TD-ISGVNS-0.1	37/50	765	27,390
	SGVNS	39/50	576	27,390
	TD-SGVNS-0.1	**42/50**	**526**	**27,390**
#509	ISGVNS	46/50	499	36,446
S^* =36,446	TD-ISGVNS-0.1	**48/50**	**454**	**36,446**
	SGVNS	**48/50**	**412**	**36,446**
	TD-SGVNS-0.1	43/50	741	36,446

TABLE 7.7 – Comparaison entre TD-SGVNS, TD-ISGVNS, SGVNS et ISGVNS sur les instances SPOT5.

Instances GRAPH.

L'exploitation de la TDTD améliore les performances de SGVNS et ISGVNS sur les deux instances les plus difficiles (cf. tableau 7.8). Pour l'instance Graph11, TD-SGVNS-0.7 obtient un succès de plus que SGVNS et réduit la déviation moyenne à l'optimum de (37.6% à 2.7%) ainsi que le temps de calcul d'environ 15%. Pour l'instance Graph13, bien que TD-SGVNS-0.7 ne trouve pas l'optimum, elle obtient toutefois des solutions de meilleure qualité en moyenne (la déviation moyenne à l'optimum est réduite de 45.1% à 12.1%). L'impact de la TDTD est largement amplifié avec ISGVNS. Pour l'instance Graph11, TD-ISGVNS-0.6 obtient 100% de succès, surclassant très nettement ISGVNS et VNS/LDS+CP, qui est la meilleure méthode sur cette instance (44 essais avec succès). La tendance se confirme pour l'instance Graph13, où TD-ISGVNS-0.6 obtient un taux de succès de 72% (36 essais avec succès) et une déviation moyenne à l'optimum de 9.7%. Pour comparaison, DGVNS-1 avec ($s_{max} = 32$), qui est la meilleure méthode sur cette instance, obtient un taux de succès de 14% (7 succès) et des solutions ayant une déviation moyenne à l'optimum de 13.8%.

Instance	Méthode	Succ.	Temps	Moy.
Graph05	ISGVNS	50/50	16	221
S^* =221	TD-ISGVNS-0.6	**50/50**	**10**	**221**
	SGVNS	50/50	19	221
	TD-SGVNS-0.7	**50/50**	**10**	**221**
Graph06	ISGVNS	**50/50**	**240**	**4,123**
S^* =4,123	TD-ISGVNS-0.5	50/50	288	4,123
	SGVNS	**50/50**	**292**	**4,123**
	TD-SGVNS-0.6	50/50	444	4,123
Graph11	ISGVNS	39/50	2,762	3,349
S^* =3,080	TD-ISGVNS-0.6	**50/50**	**1,558**	**3,080**
	SGVNS	27/50	3,026	4,238
	TD-SGVNS-0.7	**28/50**	**2,551**	**3,164**
Graph13	ISGVNS	1/50	3,196	17,085
S^* =10,110	TD-ISGVNS-0.6	**36/50**	**2,817**	**11,090**
	SGVNS	**6/50**	3,260	14,707
	TD-SGVNS-0.7	0/50	-	**11,510 (10,168)**

TABLE 7.8 – Comparaison entre TD-SGVNS, TD-ISGVNS, SGVNS et ISGVNSsur les instances GRAPH.

7.3.5.1 Bilan sur les apports de la TDTD

Nos expérimentations confirment les apports de la TDTD sur SGVNS et ISGVNS, et tout particulièrement sur les instances RLFAP et GRAPH. De plus, on retrouve les mêmes meilleures valeurs pour λ observées lors de nos précédentes expérimentations avec DGVNS (cf. section 6.3.1). Ceci démontre la pertinence de la TDTD pour identifier des structures de voisinage plus appropriées, indépendamment de la méthode utilisée.

7.3.6 Profils de performance des rapports de temps et de succès

Les figures 7.1 et 7.2 comparent les courbes des profils de performances des rapports de temps et de succès de DGVNS-1, DGVNS-2, SGVNS et ISGVNS ainsi que leurs scores respectifs selon différents critères. De ces comparaisons, nous dressons les conclusions suivantes :

- ISGVNS surclasse nettement SGVNS dans l'intervalle $]10^{0.01}, 10^{0.24}]$ (cf. figure 6.8(a)). Toutefois, pour $\tau > 10^{0.24}$, SGVNS est plus compétitive : SGVNS met 1.9 (i.e. $10^{0.28}$) fois plus de temps que la méthode la plus rapide pour résoudre 100% des instances, alors que ce facteur est de 2.93 (i.e. $10^{0.46}$) pour ISGVNS. Par ailleurs, SGVNS est plus rapide sur 35% des problèmes contre 30% pour ISGVNS (cf. colonne 3, tableau de la figure 7.2).
- ISGVNS obtient les meilleurs taux de succès sur 19 instances ($\rho(1) = 0.73$, colonne 6 du tableau de la figure 7.2) contre 17 pour SGVNS ($\rho(1) = 0.65$) et 14 pour DGVNS-2 ($\rho(1) = 0.53$). De plus, pour $1 < \tau \leq 10^{0.1}$, le fraction d'instances, pour lesquelles ISGVNS obtient τ fois moins de succès que la meilleure méthode, est plus élevée comparé aux autres méthodes. Notons toutefois, que sur 24 instances (cf. colonne 7, figure 7.2), SGVNS obtient $\tau = 2.5$ fois moins de succès que la meilleure méthode dans le pire cas. Pour comparaison, ISGVNS obtient 6 fois moins de succès dans le pire cas.
- ISGVNS et SGVNS dominent globalement DGVNS-1 et DGVNS-2. DGVNS-1 et SGVNS obtiennent les meilleurs résultats sur une plus grande proportion d'instances en terme de temps de calcul ($\rho(1) = 0.346$ contre 0.308 pour ISGVNS). C'est cependant ISGVNS qui obtient la meilleure AUC, 7% supérieure à DGVNS-1 (colonne 2, figure 7.2). Ceci se confirme sur la courbe 6.8(a), où, dans l'intervalle $[10^{0.01}, 10^{0.35}]$, ISGVNS domine toutes les autres méthodes. On observe pour SGVNS une courbe très proche de celle DGVNS sur l'intervalle $[1, 10^{0.2}]$. Au delà, SGVNS dépasse DGVNS-1 et obtient la meilleure valeur pour τ^* devant DGVNS-2 (cf.

colonne 4, figure 7.2). Il obtient donc globalement le plus petit écart à la meilleure méthode en termes de temps de calcul. Les deux méthodes montrent de meilleurs profils de rapport de performances en terme de nombre d'essais réussis, ce qui est confirmé par le tableau 7.2 (colonne 5-7), dans lequel ISGVNS et SGVNS surclassent clairement DGVNS-1 et DGVNS-2.

– SGVNS procure un meilleur compromis en termes de score sur l'ensemble des critères (cf. figure 7.2). Bien que ISGVNS soit la seconde meilleure méthode, les résultats des deux algorithmes sont très proches et on peut noter le bon comportement de ISGVNS en terme de temps de calcul. Enfin, les deux algorithmes dominent clairement DGVNS-1 et DGVNS-2.

7.4 Conclusions

Dans ce chapitre, nous avons proposé deux extensions de la méthode DGVNS qui tirent parti à la fois du graphe de clusters et des séparateurs entre ces clusters, pour guider efficacement l'exploration des grands voisinages dans VNS. Les expérimentations menées sur plusieurs instances difficiles ont montré que SGVNS et ISGVNS sont nettement plus performants que DGVNS, et que SGVNS est très efficace par rapport à ISGVNS. Ensuite, nous avons montré expérimentalement l'intérêt d'exploiter la TDTD dans SGVNS et ISGVNS. Nos résultats confirment la pertinence de la TDTD pour identifier des structures de voisinage appropriées.

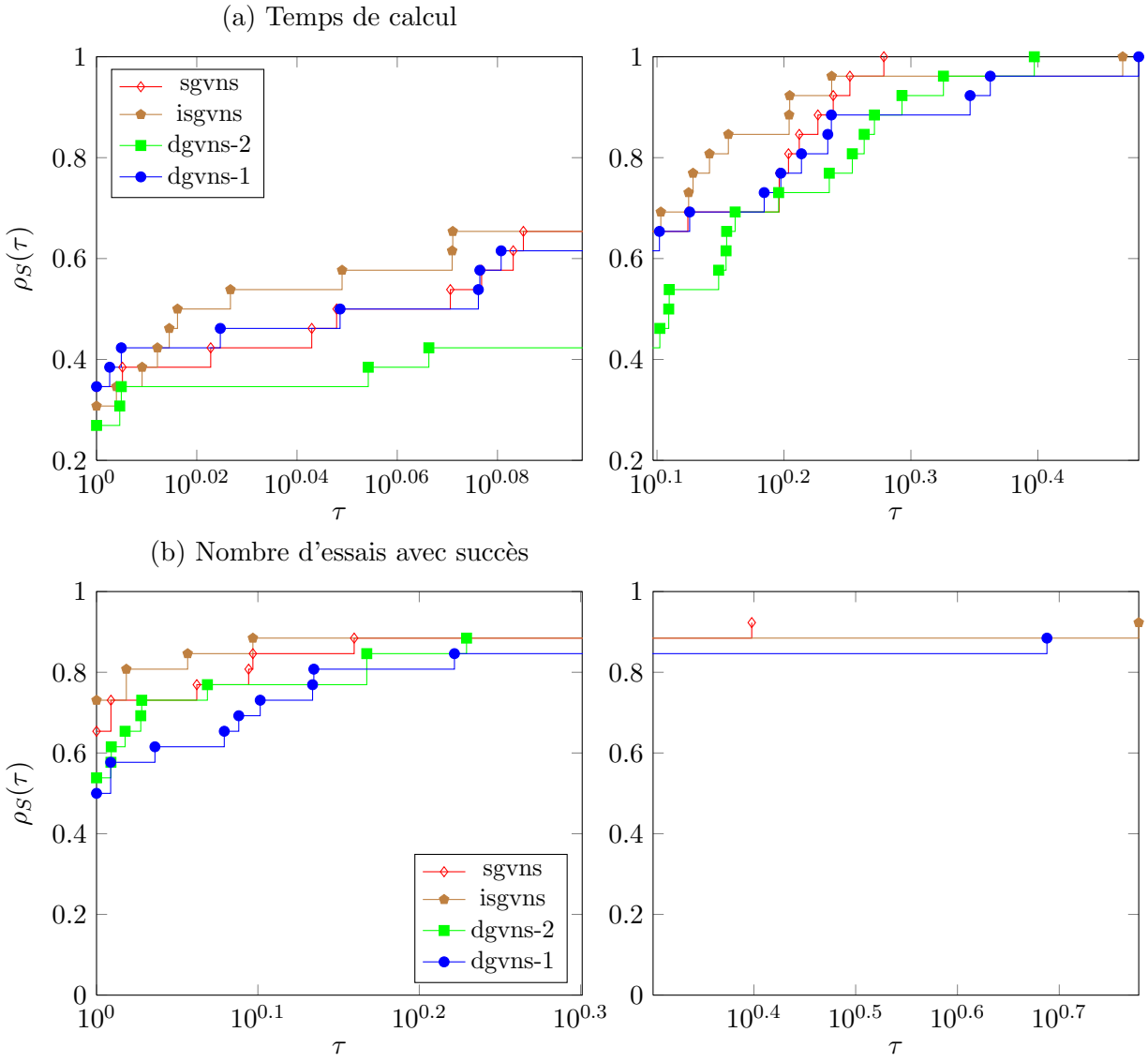


FIGURE 7.1 – Comparaison des profils de rapport de performances de `DGVNS-1`, `DGVNS-2`, `SGVNS` et `ISGVNS`.

	temps			succès		
methode	AUC	$\rho(1)$	$\frac{\tau_{min}}{\tau*}$	AUC	$\rho(1)$	$\rho(\tau_{max})$
sgvns	0.987(2)	0.346(1)	1.0(1)	1.0(1)	0.654(2)	0.923(1)
isgvns	1.0(1)	0.308(3)	0.65(3)	0.976(2)	0.731(1)	0.923(1)
dgvns-2	0.914(4)	0.269(4)	0.76(2)	0.963(3)	0.538(3)	0.885(3)
dgvns-1	0.939(3)	0.346(1)	0.63(4)	0.928(4)	0.5(4)	0.885(3)

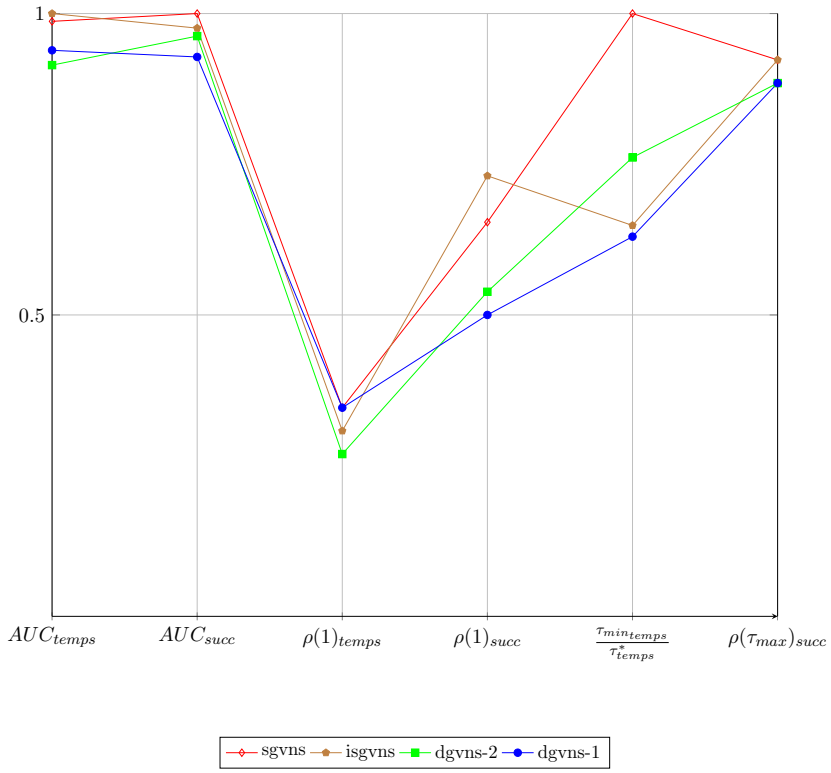


FIGURE 7.2 – Représentation graphique des valeurs des critères pour différentes méthodes.

Conclusions et Perspectives

Bilan

Les travaux présentés dans ce mémoire de thèse portent sur l'exploitation de la décomposition arborescente pour guider efficacement l'exploration de l'espace de recherche dans les méthodes de recherche locale et plus particulièrement dans les méthodes à voisinages étendus de type `VNS`. Nous nous sommes plus particulièrement intéressés à la résolution de problèmes d'optimisation sous contraintes modélisés sous forme de WCSP. Ces travaux sont, à notre connaissance, les premiers à étudier l'apport des techniques de décompositon arborescente sous cet angle. En effet, les travaux menés jusqu'à présent exploitaient les décompositions arborescentes uniquement dans le cadre des méthodes de recherche arborescente. Nos principales contributions sont les suivantes :

Un premier schéma de recherche locale (`DGVNS`), exploitant la décomposition arborescente

Dans le chapitre 5, nous avons proposé un premier schéma de recherche locale, noté `DGVNS`, exploitant la décomposition arborescente pour guider efficacement l'exploration de l'espace de recherche. Ce schéma utilise le graphe de clusters issu de la décomposition arborescente pour construire des voisinages pertinents. Nous avons aussi étudié et comparé trois différentes stratégies visant à mieux équilibrer l'intensification et la diversification au sein de `DGVNS`.

Raffinements de la décomposition arborescente

Lors de nos expérimentations, nous avons constaté que la plupart des instances étudiées contiennent des fonctions de coût beaucoup plus difficiles à satisfaire que d'autres. De plus, pour la plupart de ces instances, les décompositions obtenues comportent une proportion importante de clusters qui se chevauchent très fortement. Cette redondance entre clusters limite la diversification de `DGVNS`, en re-considérant des voisinages très proches. Pour remédier à ces problèmes, nous avons proposé, dans le chapitre 6, deux raffinements de la décomposition arborescente.

1. la `TDTD` (*Tightness Dependent Tree Decomposition*), qui exploite la dureté des fonctions de coût pour identifier les parties du graphe de contraintes les plus difficiles à satisfaire.
2. la fusion de clusters qui vise à réduire la redondance entre les clusters de la décomposition. À cet effet, nous avons proposé deux critères. Le premier consiste à fusionner les clusters partageant plus de variables qu'un seuil maximal fixé. Le second consiste à fusionner les clusters ayant une proportion de variables partagées supérieure à un seuil maximal fixé

DGVNS guidée par les séparateurs

Pour tenir compte des séparateurs entre clusters, nous avons proposé, dans le chapitre 7, deux extensions de DGVNS, notées SGVNS (*Separator-Guided* VNS) et ISGVNS (*Intensified* SGVNS), qui exploitent à la fois le graphe de clusters et les séparateurs entre ces clusters. Notre idée est de tirer parti des affectations des variables des séparateurs pour guider DGVNS vers les clusters qui sont les plus susceptibles de conduire à des améliorations importantes (i.e. les clusters contenant, dans leurs séparateurs, des variables impliquées dans l'amélioration de la solution courante).

Perspectives

Nos perspectives de recherche s'inscrivent dans la continuité de ces travaux de thèse, que ce soit pour améliorer la qualité des décompositions produites, pour proposer de nouvelles extensions de DGVNS ou pour étendre le champ d'application de nos méthodes à d'autres problèmes.

Prise en compte des coûts lors de la décomposition

Comme nous l'avons précisé dans la section 6.1.1, la TDTD ne permet pas de prendre en compte les coûts des fonctions lors de la décomposition. Or, cette information peut être exploitée pour aider la décomposition arborescente à identifier des clusters qui vont contenir des fonctions de coût ayant le plus grand impact sur la qualité des solutions. Des travaux allant dans cette direction ont été menés par [Kitching et Bacchus, 2009], dans le cadre des méthodes de recherche arborescente.

Décompositions ad-hoc pour des problèmes ayant des structures particulières

Certains problèmes ont des structures très particulières et se prêtent peu à la décomposition arborescente. C'est le cas des problèmes dont le graphe de contrainte forme une grille (instances GRID [12] ou encore les problèmes d'allocation d'emplois du temps des infirmières (*Nurse Rostering Problems*, NRP) [Cheang *et al.*, 2003]). Pour ce type de problèmes, il serait envisageable de proposer des décompositions ad-hoc afin de guider plus efficacement leur résolution.

Raffinement dynamique de la décomposition arborescente

Les deux critères de fusion que nous avons proposé (cf. section 6.2) ne permettent pas de prendre en compte l'évolution de la solution courante durant la recherche. Il serait intéressant d'envisager des critères dynamiques de fusion lors de la recherche, selon les valeurs des variables instanciées. De plus, pour la TDTD, il serait également intéressant d'étudier une variante dynamique permettant d'adapter la valeur du seuil λ au cours de la recherche.

Parallélisation de DGVNS

Avec la démocratisation des machines multi-processeurs, la parallélisation des algorithmes de résolution est devenue une problématique majeure. Plusieurs approches parallèles ont déjà été proposées pour VNS [García-López *et al.*, 2002], et de nombreux travaux dans le cadre des recherches complètes s'appuient sur la décomposition arborescente afin de paralléliser la recherche de solution (RDS-BTD [Sánchez *et al.*, 2009] ou parallel AOBB [Otten et Dechter, 2011]). Il

12. http ://graphmod.ics.uci.edu/uai08/Evaluation/Report/Benchmarks

semble donc pertinent de vouloir paralléliser DGVNS. Les travaux préliminaires, que nous avons menés, ont donné des résultats prometteurs qu'il est nécessaire d'approfondir.

Utilisation d'autres types de décomposition

Plusieurs travaux récents s'interessent à d'autres types de décomposition. Nous pouvons citer, par exemple, la décomposition en *hyper-arbre* qui est une généralisation de la décomposition arborescente [Gottlob *et al.*, 2009], ou encore la détection de communautés dans les graphes [Fortunato, 2010, Newman et Girvan, 2004], qui visent à partionner ces derniers en optimisant un critère de modularité qui est fonction de la taille des partitions et du nombre d'arêtes entre ces partitions. Notre idée serait d'exploiter ces types de décomposition au sein de DGVNS. En effet, notre approche est générique et permet l'utilisation d'autres décompositions.

Une approche hybride exploitant les séparateurs

Nous avons proposé dans [Métivier *et al.*, 2013] une nouvelle approche hybride permettant d'exploiter les séparateurs d'une décomposition arborescente d'un graphe de contraintes dans une recherche locale de type VNS combinée à une recherche arborescente. Son principe consiste à désinstancier un sous-ensemble de variables apparaissant dans les séparateurs puis celles des clusters les partageant. Ensuite, une nouvelle instanciation des variables des séparateurs est construite par une recherche gloutonne couplée à une liste tabou. Une fois les séparateurs fixés, les variables libres dans les clusters sont à leur tour fixées par une recherche arborescente. Les premiers résultats préliminaires sont très prometteurs et indiquent que les variables séparateurs peuvent exprimer un voisinage compact. Nous envisageons plusieurs pistes pour améliorer notre approche. Nous comptons également mener une étude expérimentale plus poussée avec une comparaison avec des solveurs dédiés afin de mieux apprécier l'apport de cette contribution.

Vers de nouvelles applications

Lors du challenge PIC [13] de la conférence UAI'2012, DGVNS a obtenu des résultats prometteurs sur un grand nombre d'instance du problème MPE (*Maximum Probability Explanation*) dans les réseaux Bayésiens, approchant les performances du meilleur solveur de la compétition, AOBB [Marinescu et Dechter, 2009]. Une étude expérimentale plus large est toutefois nécessaire afin d'évaluer les apports de DGVNS sur ce type d'instances.

13. http ://www.cs.huji.ac.il/project/PASCAL/index.php

Annexe A

Impact de la TDTD

Instance	Méthode	Succ.	Temps	moy.
Scen06	DGVNS-1-0.1	50/50	**94**	3389
S^* =3389	DGVNS-**1**-0.2	**50/50**	**58**	**3,389**
	DGVNS-**1**-0.3	**50/50**	**61**	**3,389**
	DGVNS-1-0.4	50/50	110	3,389
	DGVNS-1-0.5	49/50	122	3,389
	DGVNS-1-0.6	18/50	272	3,395
Scen07	DGVNS-1-0.1	38/50	273	345630
S^* =343592	DGVNS-1-0.2	45/50	271	344,603
	DGVNS-**1**-0.3	**47/50**	**229**	**344,198**
	DGVNS-**1**-0.4	**49/50**	**221**	**343,600**
	DGVNS-1-0.5	45/50	244	344,603
	DGVNS-**1**-0.6	**47/50**	**216**	**344,198**
Scen08	DGVNS-1-0.1	6/50	1,012	**273**
S^* =262	DGVNS-**1**-0.2	**7/50**	**327**	273
	DGVNS-1-0.3	5/50	323	273
	DGVNS-1-0.4	6/50	344	274
	DGVNS-**1**-0.5	**9/50**	**442**	**272**
	DGVNS-1-0.6	0/50	-	272 (263)

Méthode	Succ.	Temps	Moy
DGVNS-2-0.1	50/50	108	3,389
DGVNS-2-0.2	**50/50**	**87**	**3,389**
DGVNS-2-0.3	**50/50**	**87**	**3,389**
DGVNS-2-0.4	50/50	182	3,389
DGVNS-2-0.5	50/50	280	3,389
DGVNS-2-0.6	27/50	1,506	3,393
DGVNS-2-0.1	49/50	579	343,794
DGVNS-2-0.2	50/50	756	343,592
DGVNS-2-0.3	50/50	756	343,592
DGVNS-2-0.4	**50/50**	**331**	**343,592**
DGVNS-2-0.5	49/50	406	343,600
DGVNS-2-0.6	50/50	329	343,592
DGVNS-2-0.1	3/50	480	273
DGVNS-2-0.2	5/50	583	273
DGVNS-2-0.3	5/50	583	273
DGVNS-2-0.4	**8/50**	**673**	**273**
DGVNS-2-0.5	2/50	373	276
DGVNS-2-0.6	1/50	472	271

TABLE A.1 – Influence de λ sur les performances de DGVNS-1 et de DGVNS-2 sur les instances RLFAP.

Instance	λ	Succ.	Time	Avg
#408	DGVNS-**1-**0.1	**50/50**	**72**	**6,228**
S^* =6,228	DGVNS-**1-**0.2	**44/50**	**90**	**6,228**
	DGVNS-1-0.3	10/50	105	6,229
#412	DGVNS-**1-**0.1	**38/50**	**71**	**32,381**
S^* =32,381	DGVNS-1-0.2	32/50	135	32,384
	DGVNS-1-0.3	8/50	1337	32,384
#414	DGVNS-**1-**0.1	**42/50**	**430**	**38,478**
S^* =38,478	DGVNS-**1-**0.2	**38/50**	**471**	**38,478**
	DGVNS-1-0.3	4/50	704	38,480
#505	DGVNS-**1-**0.1	**48/50**	**92**	**21,253**
S^* =21,253	DGVNS-**1-**0.2	**18/50**	**148**	**21,253**
	DGVNS-1-0.3	4/50	319	21,256
#507	DGVNS-**1-**0.1	**45/50**	**62**	**27,390**
S^* =27,390	DGVNS-**1-**0.2	**28/50**	**134**	**27,390**
	DGVNS-1-0.3	3/50	418	27,391
#509	DGVNS-**1-**0.1	**36/50**	**286**	**36,446**
S^* =36,446	DGVNS-**1-**0.2	**39/50**	**313**	**36,446**
	DGVNS-1-0.3	2/50	583	36,448

Méthode	Succ.	Temps CPU	Moy
DGVNS-**2-0.1**	**50/50**	**74**	**6,228**
DGVNS-2-0.2	49/50	107	6,228
DGVNS-2-0.3	49/50	107	6,228
DGVNS-**2-0.1**	**49/50**	**244**	**32,381**
DGVNS-2-0.2	44/50	311	32,381
DGVNS-2-0.3	44/50	311	32,381
DGVNS-**2-0.1**	**46/50**	**478**	**38,478**
DGVNS-2-0.2	43/50	486	38,478
DGVNS-2-0.3	43/50	486	38,478
DGVNS-**2-0.1**	**50/50**	**92**	**21,253**
DGVNS-2-0.2	16/50	356	21,253
DGVNS-2-0.3	16/50	356	21,253
DGVNS-**2-0.1**	**43/50**	**562**	**27,390**
DGVNS-2-0.2	36/50	365	27,390
DGVNS-2-0.3	36/50	365	27,390
DGVNS-**2-0.1**	**46/50**	**621**	**36,446**
DGVNS-2-0.2	46/50	628	36,446
DGVNS-2-0.3	46/50	628	36,446

TABLE A.2 – Influence de λ sur les performances de DGVNS-1 et de DGVNS-2 sur les instances SPOT5.

Instance	Méthode	Succ.	Temps	Moy.
Graph05	DGVNS-1-0.1	50/50	11	221
S^* =221	DGVNS-1-0.2	50/50	11	221
	DGVNS-1-0.3	50/50	11	221
	DGVNS-1-0.4	50/50	11	221
	DGVNS-1-0.5	50/50	12	221
	DGVNS-1-0.6	50/50	13	221
	DGVNS-1-0.7	**50/50**	**8**	**221**
Graph06	DGVNS-1-0.1	50/50	343	4,123
S^* =4,123	DGVNS-1-0.2	50/50	440	4,123
	DGVNS-1-0.3	50/50	351	4,123
	DGVNS-1-0.4	50/50	457	4,123
	DGVNS-1-0.5	50/50	339	4,123
	DGVNS-1-0.6	49/50	259	4,123
	DGVNS-1-0.7	**50/50**	**261**	**4,123**
7 Graph11	DGVNS-1-0.1	3/50	3,370	4,199
S^* =3,080	DGVNS-1-0.2	10/50	3,097	4,275
	DGVNS-1-0.3	5/50	3,037	4,060
	DGVNS-1-0.4	10/50	3,132	3,881
	DGVNS-1-0.5	**12/50**	**2,818**	**3,643**
	DGVNS-1-0.6	**7/50**	**2,446**	**3,123**
	DGVNS-1-0.7	5/50	1,417	3,223
Graph13	DGVNS-1-0.1	0/50	-	22,319 (19,856)
S^* =10,110	DGVNS-1-0.2	0/50	-	24,626 (22,265)
	DGVNS-1-0.3	0/50	-	23,588 (20,477)
	DGVNS-1-0.4	0/50	-	23,170 (19,308)
	DGVNS-1-0.5	0/50	-	24,637 (18,833)
	DGVNS-1-0.6	0/50	-	19,340 (14,318)
	DGVNS-1-0.7	**0/50**	**-**	**11,449 (10,268)**

Méthode	Succ.	Temps	Moy
DGVNS-2-0.1	50/50	12	221
DGVNS-2-0.2	50/50	12	221
DGVNS-2-0.3	50/50	12	221
DGVNS-2-0.4	50/50	13	221
DGVNS-2-0.5	50/50	14	221
DGVNS-2-0.6	50/50	15	221
DGVNS-2-0.7	50/50	11	221
DGVNS-2-0.1	50/50	449	4,123
DGVNS-2-0.2	50/50	470	4,123
DGVNS-2-0.3	50/50	470	4,123
DGVNS-2-0.4	50/50	475	4,123
DGVNS-2-0.5	50/50	387	4,123
DGVNS-2-0.6	50/50	293	4,123
DGVNS-2-0.7	50/50	345	4,123
DGVNS-2-0.1	0/50	-	5,318 (3,186)
DGVNS-2-0.2	0/50	-	5,037 (3,089)
DGVNS-2-0.3	0/50	-	5,037 (3,089)
DGVNS-2-0.4	0/50	-	4,877 (3,165)
DGVNS-2-0.5	3/50	3,304	4,235
DGVNS-2-0.6	12/50	2,931	3,119
DGVNS-2-0.7	32/50	2,463	3,095
DGVNS-2-0.1	0/50	-	23,862 (20,449)
DGVNS-2-0.2	0/50	-	25,921 (23,688)
DGVNS-2-0.3	0/50	-	25,921 (23,688)
DGVNS-2-0.4	0/50	-	24,733 (20,249)
DGVNS-2-0.5	0/50	-	25,353 (22,860)
DGVNS-2-0.6	0/50	-	19,384 (13,746)
DGVNS-2-0.7	0/50	-	11,466 (10,145)

TABLE A.3 – Influence de λ sur les performances de DGVNS-1 et DGVNS-2 sur les instances GRAPH.

Table des figures

Liste des tableaux

Bibliographie

[Allouche *et al.*, 2010] Allouche, D., De Givry, S. et Schiex, T. (2010). Towards parallel non serial dynamic programming for solving hard weighted csp. *In Principles and Practice of Constraint Programming–CP 2010*, pages 53–60. Springer.

[Barbosa *et al.*, 2011] Barbosa, H. J., Bernardino, H. et Barreto, A. M. (2011). Exploring performance profiles for analyzing benchmark experiments. *In Metaheuristics International Conference (MIC) 2011*, Udine, Italy.

[Benlic et Hao, 2013] Benlic, U. et Hao, J.-K. (2013). Breakout local search for maximum clique problems. *Computers & OR*, 40(1) :192–206.

[Bensana *et al.*, 1999] Bensana, E., Lemaître, M. et Verfaillie, G. (1999). Earth observation satellite management. *Constraints*, 4(3) :293–299.

[Benthem, 1995] Benthem, H. V. (1995). Graph : Generating radio link frequency assignment problems heuristically. Mémoire de Master Recherche, Faculty of Technical Mathematics and Informatics, Delft University of Technology, Delft, The Netherlands.

[Berry *et al.*, 2004] Berry, A., Blair, J., Heggernes, P. et Peyton, B. (2004). Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 39(4) :287–298.

[Bistarelli *et al.*, 1995] Bistarelli, S., Fargier, H., Montanari, U., Rossi, F., Schiex, T. et Verfaillie, G. (1995). Semiring-based CSPs and valued CSPs : Basic properties and comparison. *In* Jampel, M., Freuder, E. C. et Maher, M. J., éditeurs : *Over-Constrained Systems*, volume 1106 de *Lecture Notes in Computer Science*, pages 111–150. Springer.

[Boddy et Dean, 1994] Boddy, M. et Dean, T. L. (1994). Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67 :245–295.

[Cabon *et al.*, 1999] Cabon, B., De Givry, S., Lobjois, L., Schiex, T. et Warners, J. (1999). Radio link frequency assignment. *Constraints*, 4(1) :79–89.

[Cheang *et al.*, 2003] Cheang, B., Li, H., Lim, A. et Rodrigues, B. (2003). Nurse rostering problems—-a bibliographic survey. *European Journal of Operational Research*, 151(3) :447–460.

[Cooper, 2003] Cooper, M. C. (2003). Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets Syst.*, 134(3) :311–342.

[Cooper *et al.*, 2008] Cooper, M. C., de Givry, S., Sánchez, M., Schiex, T. et Zytnicki, M. (2008). Virtual arc consistency for weighted CSP. *In AAAI*, pages 253–258.

[Cooper *et al.*, 2007] Cooper, M. C., de Givry, S. et Schiex, T. (2007). Optimal soft arc consistency. *In* Veloso, M. M., éditeur : *IJCAI*, pages 68–73.

[Cooper et Schiex, 2004] Cooper, M. C. et Schiex, T. (2004). Arc consistency for soft constraints. *Artif. Intell.*, 154(1-2) :199–227.

[de Givry, 2011] de Givry, S. (INRA UBIA Toulouse. 2011). Mémoire de HDR.

[de Givry *et al.*, 2005] de Givry, S., Heras, F., Zytnicki, M. et Larrosa, J. (2005). Existential arc consistency : Getting closer to full arc consistency in weighted CSPs. *In IJCAI*, pages 84–89.

[de Givry *et al.*, 2006] de Givry, S., Schiex, T. et Verfaillie, G. (2006). Exploiting tree decomposition and soft local consistency in weighted csp. *In AAAI-06*, pages 1–6.

[Dechter, 1999] Dechter, R. (1999). Bucket elimination : A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2) :41–85.

[Dechter et Fattah, 2001] Dechter, R. et Fattah, Y. E. (2001). Topological parameters for time-space tradeoff. *Artif. Intell.*, 125(1-2) :93–118.

[Dechter et Mateescu, 2007] Dechter, R. et Mateescu, R. (2007). And/or search spaces for graphical models. *Artificial intelligence*, 171(2) :73–106.

[Dolan et Moré, 2002] Dolan, E. D. et Moré, J. J. (2002). Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2) :201–213.

[Dorigo, 1992] Dorigo, M. (1992). *Optimization, Learning and Natural Algorithms (in Italian)*. Thèse de doctorat, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy.

[Edmonds et Karp, 1972] Edmonds, J. et Karp, R. (1972). Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2) :248–264.

[Fargier et Lang, 1993] Fargier, H. et Lang, J. (1993). Uncertainty in constraint satisfaction problems : a probalistic approach. *In* Clarke, M., Kruse, R. et Moral, S., éditeurs : *ECSQARU*, volume 747 de *Lecture Notes in Computer Science*, pages 97–104. Springer.

[Fontaine *et al.*, 2011a] Fontaine, M., Loudni, S. et Boizumault, P. (2011a). Apport des décompositions arborescentes dans les méthodes de recherche de type VNS. *In 12ième congrès de la société française de recherche opérationnelle et d'aide à la décision (ROADEF'11)*, pages I.89–90, Saint-Etienne.

[Fontaine *et al.*, 2011b] Fontaine, M., Loudni, S. et Boizumault, P. (2011b). Exploiting tree decomposition within VNS for solving constraint optimization problems. *In The IX Metaheuristics International Conference (MIC 2011)*, pages S12–1– S12–10, Udine, Italy.

[Fontaine *et al.*, 2011c] Fontaine, M., Loudni, S. et Boizumault, P. (2011c). Guiding VNS with tree decomposition. *In 23-rd IEEE International Conference on Tools with Artificial Intelligence, (ICTAI'11)*, pages 505–512, Boca Raton, Florida, USA.

[Fontaine *et al.*, 2012] Fontaine, M., Loudni, S. et Boizumault, P. (2012). Exploitation de la décomposition arborescente pour guider la recherche VNS. *In 8-èmes Journées Francophones de Programmation par Contraintes (JFPC'12)*, pages 117–126, Toulouse.

[Fontaine *et al.*, 2013a] Fontaine, M., Loudni, S. et Boizumault, P. (2013a). Exploiting Tree Decomposition for Guiding Neighborhoods Exploration for VNS. *RAIRO Operations Research*, 47(2) :91–123.

[Fontaine *et al.*, 2013b] Fontaine, M., Loudni, S. et Boizumault, P. (2013b). Raffinement de décompositions arborescentes par fusion de clusters pour guider DGVNS. *In 9-èmes Journées Francophones de Programmation par Contraintes (JFPC'13)*, pages 1–10, Aix en Provence.

[Fortunato, 2010] Fortunato, S. (2010). Community detection in graphs. *Physics Reports*, 486(1) :75–174.

[Freuder et Wallace, 1992] Freuder, E. C. et Wallace, R. J. (1992). Partial constraint satisfaction. *Artif. Intell.*, 58(1-3) :21–70.

[Fulkerson et Gross, 1965] Fulkerson, D. et Gross, O. (1965). Incidence matrices and interval graphs. *Pacific J. Math*, 15(3) :835–855.

[García-López *et al.*, 2002] García-López, F., Melián-Batista, B., Moreno-Pérez, J. A. et Moreno-Vega, J. M. (2002). The parallel variable neighborhood search for the p-median problem. *Journal of Heuristics*, 8(3) :375–388.

[Gavril, 1974] Gavril, F. (1974). The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1) :47–56.

[Glover, 1986] Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & OR*, 13(5) :533–549.

[Glover et Laguna, 1997] Glover, F. et Laguna, F. (1997). *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA.

[Gogate et Dechter, 2004] Gogate, V. et Dechter, R. (2004). A complete anytime algorithm for treewidth. *In Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 201–208. AUAI Press.

[Gottlob *et al.*, 2009] Gottlob, G., Miklós, Z. et Schwentick, T. (2009). Generalized hypertree decompositions : Np-hardness and tractable variants. *J. ACM*, 56(6).

[Hansen *et al.*, 2001] Hansen, P., Mladenović, N. et Perez-Britos, D. (2001). Variable neighborhood decomposition search. *Journal of Heuristics*, 7(4) :335–350.

[Hao *et al.*, 1998] Hao, J.-K., Dorne, R. et Galinier, P. (1998). Tabu search for frequency assignment in mobile radio networks. *J. Heuristics*, 4(1) :47–62.

[Hao *et al.*, 1999] Hao, J.-K., Galinier, P. et Habib, M. (1999). Méthaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes. *Revue d'Intelligence Artificielle*, 13 :1–39.

[Harvey et Ginsberg, 1995] Harvey, W. D. et Ginsberg, M. L. (1995). Limited discrepancy search. *In IJCAI (1)*, pages 607–615.

[Hirschhorn et Daly, 2005] Hirschhorn, J. et Daly, M. (2005). Genome-wide association studies for common diseases and complex traits. *Nature Reviews Genetics*, 6(2) :95–108.

[Holland, 1992] Holland, J. H. (1992). *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA.

[Jégou *et al.*, 2005] Jégou, P., Ndiaye, S. et Terrioux, C. (2005). Computing and exploiting tree-decompositions for solving constraint networks. *In CP*, pages 777–781.

[Jégou *et al.*, 2006] Jégou, P., Ndiaye, S. et Terrioux, C. (2006). Strategies and Heuristics for Exploiting Tree-decompositions of Constraint Networks. *In Inference methods based on graphical structures of knowledge (WIGSK'06)*, pages 13–18.

[Jégou et Terrioux, 2004] Jégou, P. et Terrioux, C. (2004). Decomposition and good recording for solving max-csps. *In ECAI*, volume 16, page 196.

[Kask et Dechter, 2001] Kask, K. et Dechter, R. (2001). A general scheme for automatic generation of search heuristics from specification dependencies. *Artif. Intell.*, 129(1-2) :91–131.

[Kirkpatrick *et al.*, 1983] Kirkpatrick, S., Jr., D. G. et Vecchi, M. P. (1983). Optimization by simmulated annealing. *Science*, 220(4598) :671–680.

[Kitching et Bacchus, 2009] Kitching, M. et Bacchus, F. (2009). Exploiting decomposition on constraint problems with high tree-width. *In IJCAI*, pages 525–531.

[Kjaerulff, 1992] Kjaerulff, U. (1992). Optimal decomposition of probabilistic networks by simulated annealing. *Statistics and Computing*, 2(1) :7–17.

[Koster *et al.*, 2001] Koster, A., Bodlaender, H. et Van Hoesel, S. (2001). *Treewidth : computational experiments*. METEOR, Maastricht research school of Economics of Technology and Organizations ; University Library, Universiteit Maastricht [Host].

[Koster, 1999] Koster, A. M. C. A. (1999). *Frequency assignment : Models and Algorithms*. Thèse de doctorat, University of Maastricht, The Netherlands. Available at www.zib.de/koster/thesis.html.

[Larrañaga *et al.*, 1997] Larrañaga, P., Kuijpers, C., Poza, M. et Murga, R. (1997). Decomposing bayesian networks : triangulation of the moral graph with genetic algorithms. *Statistics and Computing*, 7(1) :19–34.

[Larrosa, 2002] Larrosa, J. (2002). Node and arc consistency in weighted csp. *In AAAI/IAAI*, pages 48–53.

[Larrosa et Meseguer, 1999] Larrosa, J. et Meseguer, P. (1999). Partition-based lower bound for max-csp. *In CP*, pages 303–315.

[Larrosa et Schiex, 2003] Larrosa, J. et Schiex, T. (2003). In the quest of the best form of local consistency for weighted CSP. *In IJCAI*, pages 239–244.

[Larrosa et Schiex, 2004] Larrosa, J. et Schiex, T. (2004). Solving weighted CSP by maintaining arc consistency. *Artif. Intell.*, 159(1-2) :1–26.

[Lee et Leung, 2009] Lee, J. H.-M. et Leung, K. L. (2009). Towards efficient consistency enforcement for global constraints in weighted constraint satisfaction. *In Proceedings of the 21st international jont conference on Artifical intelligence*, pages 559–565. Morgan Kaufmann Publishers Inc.

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

[Loudni et Boizumault, 2008] Loudni, S. et Boizumault, P. (2008). Combining VNS with constraint programming for solving anytime optimization problems. *EJOR*, 191 :705–735.

[Loudni *et al.*, 2012a] Loudni, S., Fontaine, M. et Boizumault, P. (2012a). Exploiting separators for guiding VNS. volume 39 de *Electronic Notes in Discrete Mathematics*, pages 265–272.

[Loudni *et al.*, 2012b] Loudni, S., Fontaine, M. et Boizumault, P. (2012b). Exploiting separators for guiding VNS. *In EURO Conference XXVIII on Variable Neighbourhood Search (MEC VNS'12)*, page 24 (abstract), Herceg Novi, Montenegro.

[Loudni *et al.*, 2013a] Loudni, S., Fontaine, M. et Boizumault, P. (2013a). DGVNS guidée par les séparateurs. *In 9-èmes Journées Francophones de Programmation par Contraintes (JFPC'13)*, pages 1–9, Aix en Provence.

[Loudni *et al.*, 2013b] Loudni, S., Fontaine, M. et Boizumault, P. (2013b). Intensification/Diversification in Decomposition Guided VNS. *In 8th International Workshop on Hybrid MetaHeuristics (HM'13)*, volume 7919 de *LNCS*, pages 22–36, Napoli, Italy.

[Marinescu et Dechter, 2009] Marinescu, R. et Dechter, R. (2009). And/or branch-and-bound search for combinatorial optimization in graphical models. *Artificial Intelligence*, 173(16) :1457–1491.

[Minton *et al.*, 1990] Minton, S., Johnston, M. D., Philips, A. B. et Laird, P. (1990). Solving large-scale constraint-satisfaction and scheduling problems using a heuristic repair method. *In AAAI*, pages 17–24.

[Mladenovic et Hansen, 1997] Mladenovic, N. et Hansen, P. (1997). Variable neighborhood search. *Computers And Operations Research*, 24 :1097–1100.

[Métivier *et al.*, 2013] Métivier, J., Fontaine, M. et Loudni (2013). Exploiter les séparateurs : une approche hybride combinant recherche locale et arborescente. *In 9-èmes Journées Francophones de Programmation par Contraintes (JFPC'13)*, pages 1–10, Aix en Provence.

[Neveu et Trombettoni, 2003] Neveu, B. et Trombettoni, G. (2003). Incop : An open library for incomplete combinatorial optimization. *In Principles and Practice of Constraint Programming–CP 2003*, pages 909–913. Springer.

[Neveu *et al.*, 2004] Neveu, B., Trombettoni, G. et Glover, F. (2004). Id walk : A candidate list strategy with a simple diversification device. *In* Wallace, M., éditeur : *CP*, volume 3258 de *Lecture Notes in Computer Science*, pages 423–437. Springer.

[Newman et Girvan, 2004] Newman, M. E. J. et Girvan, M. (2004). Finding and evaluating community structure in networks. *Phys. Rev. E*, 69(2) :026113.

[Otten et Dechter, 2011] Otten, L. et Dechter, R. (2011). Finding most likely haplotypes in general pedigrees through parallel search with dynamic load balancing. *In Pac Symp Biocomput*, volume 16, pages 26–37. World Scientific.

[Robertson et Seymour, 1986] Robertson, N. et Seymour, P. (1986). Graph minors. ii. algorithmic aspects of tree-width. *J. Algorithms*, 7(3) :309–322.

[Rose *et al.*, 1976] Rose, D., Tarjan, R. et Lueker, G. (1976). Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on computing*, 5(2) :266–283.

[Rosenfeld *et al.*, 1976] Rosenfeld, A., Hummel, R. A. et Zucker, S. W. (1976). Scene labeling by relaxation operations. *Systems, Man and Cybernetics, IEEE Transactions on*, 6(6) :420–433.

[Ruttkay, 1994] Ruttkay, Z. (1994). Fuzzy constraint satisfaction. *In In Proc. 3rd IEEE International Conference on Fuzzy Systems*, pages 1263–1268.

[Sánchez *et al.*, 2009] Sánchez, M., Allouche, D., de Givry, S. et Schiex, T. (2009). Russian doll search with tree decomposition. *In IJCAI*, pages 603–608.

[Sanchez *et al.*, 2008] Sanchez, M., De Givry, S. et Schiex, T. (2008). Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. *Constraints*, 13(1-2) :130–154.

[Schiex, 1992] Schiex, T. (1992). Possibilistic constraint satisfaction problems or "How to handle soft constraints ?". *In* Dubois, D. et Wellman, M. P., éditeurs : *UAI*, pages 268–275. Morgan Kaufmann.

[Schiex, 2000] Schiex, T. (2000). Arc consistency for soft constraints. *In* Dechter, R., éditeur : *CP*, volume 1894 de *Lecture Notes in Computer Science*, pages 411–424. Springer.

[Schiex *et al.*, 1995] Schiex, T., Fargier, H. et Verfaillie, G. (1995). Valued constraint satisfaction problems : Hard and easy problems. *In IJCAI (1)*, pages 631–639.

[Shapiro et Haralick, 1981] Shapiro, L. et Haralick, R. (1981). Structural descriptions and inexact matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3 :504–519.

[Solnon, 2008] Solnon, C. (2008). Optimisation par colonies de fourmis (Collection programmation par contraintes). Editions HermÃšs-Lavoisier.

[Tarjan et Yannakakis, 1984a] Tarjan, R. et Yannakakis, M. (1984a). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13 :566.

[Tarjan et Yannakakis, 1984b] Tarjan, R. E. et Yannakakis, M. (1984b). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3) :566–579.

[Terrioux et Jégou, 2003] Terrioux, C. et Jégou, P. (2003). Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146(1) :43–75.

[van Benthem, 1995] van Benthem, H. (1995). Graph : Generating radiolink frequency assignment problems heuristically.

[Verfaillie *et al.*, 1996] Verfaillie, G., Lemaître, M. et Schiex, T. (1996). Russian doll search for solving constraint optimization problems. *In AAAI/IAAI, Vol. 1*, pages 181–187.

[Voudouris et Tsang, 1999] Voudouris, C. et Tsang, E. P. K. (1999). Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, 113(2) :469–499.

[Yannakakis, 1981] Yannakakis, M. (1981). Computing the minimum fill-in is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1) :77–79.

[Zytnicki *et al.*, 2005] Zytnicki, M., Heras, F., de Givry, S. et Larrosa, J. (2005). Cohérence d'arc existentielle : un pas de plus vers la cohérence d'arc complète. *In JFPC*, pages 139–148.

Apport de la décomposition arborescente pour les méthodes de type VNS

Actuellement, la résolution de problèmes d'optimisation sous contraintes tire rarement parti de la structure du problème trait. Or, il existe de nombreux problèmes réels fortement structurés dont la décomposition arborescente pourrait s'avérer très profitable. Les travaux menés jusqu'à présent exploitent les décompositions arborescentes uniquement dans le cadre des méthodes de recherche complète. Dans cette thèse, nous étudions l'apport des décompositions arborescentes pour les méthodes de recherche locale de type VNS (Variable Neighborhood Search), dont l'objectif est de trouver une solution de très bonne qualité en un temps limité.

Cette thèse apporte trois contributions. La première est un schéma générique (DGVNS), exploitant la décomposition arborescente pour guider efficacement l'exploration de l'espace de recherche. Trois différentes stratégies visant à équilibrer l'intensification et la diversification de DGVNS sont étudiées et comparées. La seconde contribution propose deux raffinements de la décomposition arborescente. Le premier exploite la dureté des fonctions de coût pour identifier les parties du graphe de contraintes les plus difficiles à satisfaire. Le second raffinement cherche à augmenter la proportion de variables propres dans les clusters. La troisième contribution consiste en deux extensions de DGVNS qui exploitent à la fois le graphe de clusters et les séparateurs. Chaque contribution proposée est évaluée et comparée au travers d'expérimentations menées sur de multiples instances de quatre problèmes réels.

Exploiting Tree Decomposition for Guiding Neighborhoods Exploration for VNS.

Many real-life constraints optimization problems are very large and exhibit a highly structured constraints graph. Exploiting such structural properties may lead these problems to be tractable. Tree decomposition aims to decompose the problem to be solved into subproblems constituting an acyclic graph. As each subproblem is significantly smaller in size than the original one, it can be solved more efficiently. Actually, there are few papers exploiting tree decomposition for complete search methods. In this thesis, we propose different ways to exhibit and exploit structural properties for local search methods using VNS (Variable Neighborhood Search).

This thesis makes three contributions. First, we introduce a novel local search scheme, DGVNS (Decomposition Guided VNS), that exploits the graph of clusters to guide VNS. Three different strategies of intensification and diversification are then proposed and compared. Second, we introduce two ways to refine a tree decomposition. The purpose of the first one is to exhibit the hardest part of the problem using the constraints tightness. The second one aims to reduce the redundancy between clusters and increase the proportion of proper variables within those clusters. Finally, we propose two extensions of DGVNS, that make use of both the graph of clusters and the separators between the clusters. Each contribution has been evaluated and compared using experiments conducted on various benchmarks of four real life problems.

Mots-clés :

- **indexation Rameau : Optimisation combinatoire ; Programmation par contraintes ; Informatique ; Intelligence Artificielle ; Algorithmes ; Recherche Opérationnelle.**
- **indexation libre : Métaheuristique.**

Keywords : Combinatorial Optimisation ; Constraint Programming (Computer Science) ; Computer Science ; Artificial Intelligence ; Algorithms ; Operations Research.

Discipline : Informatique et applications

Groupe de Recherche en Informatique, Image, Automatique et Instrumentation de Caen,
CNRS UMR 6072,Université de Caen Basse-Normandie BP 5186,14032 Caen Cedex, FRANCE