N^o Année 2007

UNIVERSITÉ DE DROIT, D'ECONOMIE ET DES SCIENCES
D'AIX-MARSEILLE

THÈSE

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ Paul CEZANNE
Faculté des Sciences et Techniques

Spécialité Informatique

ECOLE DOCTORALE DE MATHEMATIQUES
ET D'INFORMATIQUE DE MARSEILLE

par

Samba Ndojh NDIAYE

Calcul et exploitation de recouvrements acycliques pour la résolution de (V)CSP

Soutenue le 11 décembre 2007 devant le jury composé de :

M.	Boi Faltings	Rapporteur
M.	Philippe Jégou	Directeur de thèse
M.	Pedro Meseguer	Examinateur
M.	Thomas Schiex	Rapporteur
M.	Cyril Terrioux	Directeur de thèse
M.	Gérard Verfaillie	Examinateur
Mme.	Marie-Catherine Vilarem	Examinatrice

Laboratoire des Sciences de l'Information et des Systèmes

Table des matières

1 Introduction **5**

2 Etat de l'art **9**
2.1 Introduction . . . 9
2.2 Graphes, hypergraphes et décompositions . . . 9
2.2.1 Rappels sur les graphes . . . 9
2.2.2 Rappels sur les hypergraphes . . . 12
2.2.3 Décompositions de (hyper)graphes . . . 15
2.3 Problème de satisfaction de contraintes : CSP . . . 22
2.3.1 Présentation du formalisme . . . 22
2.3.2 Les méthodes de résolution . . . 25
2.4 VCSP . . . 38
2.4.1 Présentation du formalisme . . . 38
2.4.2 Les méthodes de résolution . . . 40
2.5 Conclusion . . . 52

3 Calcul de décompositions arborescentes **55**
3.1 Introduction . . . 55
3.2 La classe des graphes triangulés . . . 56
3.3 Les différentes classes de méthodes de triangulation . . . 59
3.3.1 Les algorithmes exacts . . . 59
3.3.2 Les algorithmes d'approximation avec garanties . . . 61
3.3.3 Les algorithmes heuristiques . . . 62
3.3.4 Les algorithmes minimaux . . . 63
3.4 Définition de nouvelles stratégies de triangulation. . . . 65
3.4.1 Stratégies basées sur les séparateurs minimaux du graphe. . . . 65
3.4.2 Minesp : une stratégie basée sur l'espérance mathématique du nombre de solutions d'un CSP. . . . 68
3.5 Etude expérimentale . . . 68
3.5.1 Qualité de l'approximation . . . 68
3.5.2 Qualité des triangulations par rapport à la résolution pratique de (V)CSP 71
3.5.3 Limitation de la taille des séparateurs : une nouvelle comparaison . . . 76
3.6 Conclusion . . . 80

4 Exploitation d'hypergraphes acycliques recouvrants **83**
4.1 Introduction . . . 83
4.2 Une nouvelle définition des hypergraphes acycliques . . . 84
4.3 Recouvrements par hypergraphes acycliques . . . 91
4.4 Exploitation Algorithmique des CAHs : BDH . . . 95
4.4.1 Construction incrémentale des arbres . . . 95

4.4.2 Description de l'algorithme BDH 98
4.5 Extension de BDH aux VCSP : BDH-val 103
4.6 Ordre sur les variables : choix hypergraphes recouvrants 107
4.6.1 Les Classes d'ordres 108
4.6.2 Heuristiques de construction des arbres. 109
4.6.3 Heuristiques de fusion d'hyperarêtes. 109
4.6.4 Heuristiques de choix d'hyperarêtes. 110
4.6.5 Heuristiques de choix de variables. 111
4.7 Etude expérimentale 112
4.7.1 Comparaison des heuristiques sur les CSP 112
4.7.2 Comparaison des heuristiques sur les VCSP 116
4.8 Conclusion 117

5 SBBT : un cadre générique pour la résolution de CSP 121
5.1 Introduction 121
5.2 SBBT : un schéma générique d'algorithmes énumératifs 122
5.2.1 Formalisation et justifications 122
5.2.2 Algorithme SBBT 124
5.3 Analyse de complexité 127
5.3.1 SBBT avec un ordre d'affectation des variables et un ensemble de séparateurs induits par une décomposition arborescente 128
5.3.2 SBBT avec un ordre d'affectation des variables et un ensemble de séparateurs induits par un recouvrement acyclique 128
5.3.3 SBBT avec un ordre d'affectation des variables et un ensemble de séparateurs induits par une décomposition en bicomposantes 129
5.3.4 SBBT avecun ordre d'affectation des variables et un ensemble de séparateurs induits par un dtree 130
5.3.5 SBBT avec un ordre d'affectation des variables et un ensemble de séparateurs induits par une décomposition Hinge 130
5.3.6 SBBT avec un ordre d'affectation des variables et un ensemble de séparateurs induits par un pseudo-tree ou un arrangement en arbre orienté 131
5.3.7 Complexités de SBBT avec un ordre d'affectation des variables et un ensemble de séparateurs quelconques 131
5.4 Discussion 132
5.5 Conclusion 134

6 Conclusion 137

7 Notations 149

8 Annexes 153

Chapitre 1

Introduction

L'Intelligence Artificielle, dans l'imaginaire populaire, fait référence à des robots plus ou moins évolués, dotés d'une certaine intelligence cherchant à égaler celle de l'Homme, en copiant ses comportements. Il est clair que cet aspect est beaucoup plus attrayant tant les médias, à intervalles réguliers, se font échos des avancées dans ce domaine des japonais qui proposent depuis quelques décennies des robots de ce type. Dire que je travaille dans le domaine de l'Intelligence Artificielle a suscité beaucoup d'interrogations durant les trois dernières années. Face à ces questions multiples et à la complexité d'expliquer exactement un sujet de thèse à des personnes ne travaillant pas dans le même domaine, il a fallu faire toute une gymnastique pour donner quelques éclaircissements. Ces derniers se révélant souvent vains. J'ai donc essayé de réfléchir à un exemple qui serait en mesure de dire l'objet de mes travaux de recherche. Je me suis alors souvenu de celui de l'emploi du temps. Il était en effet facile de se rendre compte que pour une petite école, cela doit être relativement facile. Cependant, il est tout aussi facile de voir que pour une entreprise avec des dizaines de milliers d'employés, la tâche est beaucoup plus ardue. Elle nécessite parfois l'utilisation de logiciels appropriés pour faire finalement le même raisonnenment que le directeur d'école. Le raisonnement des logiciels est certes imparfait comparé à celui du directeur mais l'avantage est surtout de pouvoir traiter des problèmes de très grande taille non accessibles aux humains. Voilà donc un problème d'intelligence artificielle facilement compréhensible du point de vue de l'utilité et de la nécessité d'utiliser l'ordinateur. Mais, il faut remarquer que, pour résoudre ce problème, l'ordinateur n'étant pas aussi efficace que l'Homme au niveau du raisonnement mais tellement plus au niveau du calcul, va schématiquement essayer toutes les possibilités jusqu'à en trouver une bonne ou démontrer qu'il n'y a pas de solution. Autant, sa capacité de calcul est élevée, autant il est utile de voir que si le problème est de taille considérable, le nombre de possibilités est très grand. Tellement grand, que même pour un ordinateur, il faut beaucoup de temps. Combien exactement ? Cela dépend de la méthode utilisée. Les méthodes usuelles en théorie mettent beaucoup de temps. Une durée souvent inacceptable dans le monde de l'industrie. On ne peut sans doute pas se permettre d'attendre l'emploi du temps des employés d'une entreprise pendant 10 ans. Il existe d'autres méthodes qui proposent des durées de calcul plus courtes en théorie. Néanmoins, ces méthodes sont souvent catastrophiques en pratique. Mes travaux se situent donc dans ce contexte. Nous cherchons à proposer des solutions pour que la pratique rejoigne la théorie : rendre opérationnelles les méthodes qui sont les plus efficaces en théorie. L'expérience a pu montrer que cette stratégie d'intéressement ne rencontrait pas un franc succès au niveau des personnes étrangères à ce domaine. Je me suis donc résigné, concernant mes travaux, à dialoguer de manière presque exclusive avec les membres de la communauté contraintes. Cela n'empêche aucunement de présenter les choses de la manière la plus claire possible et la plus accessible donc à un grand nombre, ou du moins, à un peu plus de trois personnes. Cependant, je vais tenter d'être très formel afin de présenter ce travail rigoureusement.

L'Intelligence Artificielle recouvre un grand nombre de domaines tels que la simulation, la

planification, etc. Mes travaux se situent plutôt au niveau de l'aide à la décision et de l'optimisation sous contraintes. Depuis quelques dizaines d'années, des chercheurs ont essayé de construire des mécanismes assez sophistiqués qui permettraient de raisonner sur un ensemble de données. L'objectif étant de pouvoir acquérir de la connaissance sur ces données pour répondre à un certain nombre de questions ou résoudre des problèmes portant sur cet ensemble. La difficulté majeure est le choix d'une représentation permettant un traitement efficace de l'information. Plusieurs représentations ont vu le jour : la logique des propositions (SAT) et ses extensions, le formalisme CSP et les siennes, etc. Il existe de nombreuses relations entre ces différents formalismes et des outils qui dans plusieurs cas permettent de passer d'une représentation à une autre. Nos travaux ont été effectués avec le formalisme CSP mais pourraient assez naturellement s'étendre à d'autres formalismes tels que SAT. Le choix porté sur le formalisme CSP (problème de satisfaction de contraintes) n'est pas fortuit. En effet, il permet une représentation simple des données et regorge de méthodes très puissantes proposées depuis des dizaines d'années pour raisonner dans ce formalisme. Une instance de CSP est donnée par un ensemble de variables, associées chacune à un domaine de valeurs de taille finie et par un ensemble de contraintes qui mettent en relation les variables et définissent des combinaisons de valeurs compatibles. Une solution est une affectation de l'ensemble des variables qui respecte toutes les contraintes. Le problème de décision associé à un CSP, qui consiste à déterminer l'existence ou non d'une solution, est NP-complet. La méthode de résolution de base est le *Backtrack standard* qui schématiquement procède à l'énumération de toutes les affectations possibles jusqu'à trouver une solution. Dans le pire des cas, cette méthode peut explorer la totalité de l'espace de recherche dont la taille est bornée par d^n avec d la taille maximale des domaines du problème et n le nombre de variables. Elle peut donc s'avérer très coûteuse. Elle se révèle totalement inefficace en pratique à cause du grand nombre de redondances inhérentes à cette technique. Plusieurs voies ont été proposées pour l'améliorer que ce soit en pratique ou en théorie : les améliorations effectives de la technique du backtracking et la décomposition des problèmes. Le *filtrage* permet de réduire la taille du problème ou de le modifier de sorte à construire un problème équivalent mais généralement plus facile à résoudre. Il existe différents degrés de filtrage ([Wal75, Mac77, Fre78],...), plus ou moins efficaces et plus ou moins coûteux, qui se basent sur les notions de *cohérence locale* ou générale et de propagation de contraintes. Associées à des techniques de *retour en arrière non chronologique* ([Gas79, Dec90],...), de *consistance avant* ([HE80, SF94],...)et d'*apprentissage* ([Dec90, FD94, SV93, SV94],...), ces améliorations ont mené à des méthodes telles que *FC* ([HE80]) et *MAC* ([SF94]) qui sont bien plus efficaces en pratique, mais dont la complexité théorique demeure en $O(md^n)$, où m est le nombre de contraintes. De meilleures bornes de complexité sont l'œuvre de méthodes basées sur la décomposition du problème ([Fre85, DP89, BM96, Dar01, BT01a, JT03, DM07, GLS00, CJG05]). Cette décomposition est essentiellement un *recouvrement acyclique* du problème permettant de profiter des indépendances entre différentes parties. L'une des meilleures bornes est en $O(exp(w+1))$, où w est la *treewidth* de la décomposition arborescente ([RS86]) du problème. En effet cette complexité peut considérablement améliorer la complexité de base, à savoir $O(exp(n))$ avec $w < n$, et parfois $w \ll n$. Il existe d'autres méthodes avec des complexités encore meilleures ([GLS00, CJG05]), mais difficilement exploitables en pratique car ces bornes sont souvent obtenues au détriment de l'efficacité pratique. Généralement, on observe une inadéquation entre les excellentes bornes de complexité théorique et les résultats médiocres voire inexistants des méthodes structurelles. La méthode BTD ([JT03]) est une méthode structurelle qui obtient parmi les meilleurs résultats pratiques tout en gardant une excellente complexité théorique en $O(exp(w+1))$. Ces résultats ont été obtenus sans une étude approfondie sur la qualité des décompositions et des stratégies d'exploitation de ces décompositions. Nous nous proposons ici de faire cette étude et de définir des stratégies de décomposition et d'exploitation de qualité pour une résolution efficace en pratique. En effet, les méthodes de décomposition ont été étudiées intensivement d'un point de vue théorique avec une recherche d'une décomposition de largeur réduite. Mais, notre but est de remédier à l'inadéquation entre les bornes de complexité théorique et l'efficacité pratique. De ce point

de vue, la treewidth n'est pas le seul critère pertinent dans l'évaluation de la qualité d'une décomposition. La taille des intersections (séparateurs) entre les différentes composantes est d'une importance cruciale dans la majeure partie de ces méthodes dans la mesure où l'espace mémoire requis en dépend. Une limitation de la taille des séparateurs s'avère incontournable pour éviter un échec de la résolution pour une insuffisance d'espace mémoire. De même, les méthodes structurelles imposent généralement une grande rigidité dans l'ordre d'affectation des variables. Or, les ordres statiques obtiennent généralement des résultats catastrophiques par rapport aux heuristiques dynamiques. Nous proposons ainsi des stratégies permettant de s'affranchir de cette rigidité par une exploitation dynamique de la structure du problème. Cela donne une liberté accrue dans le choix d'heuristiques efficaces d'ordonnancement des variables qui sont dynamiques. Les premiers résultats obtenus montrent des gains considérables au niveau du temps de résolution.

Cette étude sera étendue au formalisme *CSP valués* (VCSP [SFV95]) qui est une extension du formalisme CSP permettant de prendre en compte la notion de préférence sur les solutions. Contrairement au formalisme CSP, il autorise l'utilisation de contraintes dites *molles*. Ces dernières font référence à des possibilités, préférences, souhaits qu'il faut satisfaire si possible. Le problème VCSP consiste donc à trouver une affectation totale de coût optimum suivant un critère exprimé par les contraintes. Le problème VCSP est NP-difficile.
La recherche d'une solution optimale est beaucoup plus difficile que celle d'une solution dans le cadre CSP. En effet, elle nécessite le parcours de la totalité de l'espace de recherche. Généralement, les méthodes de résolution sont des adaptations des méthodes du cadre décision au cadre optimisation. La méthode classique est le *branch and bound*. La méthode du branch and bound consiste à parcourir l'espace de recherche de la même manière que le Backtracking, en maintenant deux bornes : un majorant de l'optimum et un minorant du coût de l'affectation courante. Durant la recherche, si le minorant dépasse le majorant, on sait que le coût de l'affectation en cours de construction est supérieure à l'optimum. L'extension de cette affectation est stoppée et un backtrack réalisé.

Le minorant et le majorant sont primordiaux dans l'efficacité de la méthode.

Pour calculer un minorant de qualité et permettre par la même occasion une réduction de l'espace de recherche, des techniques de filtrages basées sur des cohérences locales plus ou moins fortes ont été proposées ([FW92, SFV95, SFV97, Lar02, LS03, Coo03, CS04, dGHZL05, CdGS07]). Ces dernières sont pour la plupart des adaptations de la consistance d'arc définie dans le cadre classique. Elles ont mené à des méthodes telles que *PFC-MRDAC* ([LMS99]), très efficaces en pratique, mais dont la complexité théorique demeure en $O(md^n)$. Les méthodes basées sur la décomposition du problème ou les techniques de programmation dynamique ([VLS96, Kos99, LD03, LMS02, JT03, dGSV06, DM07]) offrent de meilleures bornes de complexité théorique. On peut ainsi citer *BTD-val* ([JT03]), une adaptation de BTD aux VCSP, qui obtient de bons résultats en pratique avec complexité théorique en $O(exp(w+1))$. Notre étude sur les stratégies de calcul et d'exploitation des décompositions de VCSP a permis d'améliorer les performances des méthodes structurelles en pratique.

Le prochain chapitre de cette thèse présente un état de l'art sur les différentes décompositions de graphes et d'hypergraphes qui ont été utilisées pour la résolution de CSP et de VCSP. Il présente également les formalismes CSP et VCSP et des méthodes de résolution développées dans ce cadre. Le troisième chapitre est consacré à une large étude sur les algorithmes de *triangulation* qui sont très largement utilisés pour calculer des recouvrements acycliques de (V)CSP. Nous définissons dans ce cadre un nouvel algorithme de triangulation TSep, modulable avec différentes stratégies et basé sur la complétion d'un ensemble de séparateurs. Les recouvrements ainsi obtenus améliorent significativement la résolution en pratique. Le chapitre 4 est consacré à la méthode *BDH* qui permet une exploitation dynamique des recouvrements acycliques et une liberté accrue concernant les heuristiques de choix de variables. Enfin, le cadre générique SBBT, basé sur un ensemble quelconque de séparateurs du problème et des composantes paramétrables, recouvre un spectre très large d'algorithmes allant de méthodes énumératives à des techniques

structurelles avec de meilleures bornes de complexité théorique.

Chapitre 2

Etat de l'art

2.1 Introduction

La littérature abonde de travaux concernant les formalismes CSP et VCSP. Un nombre considérable de méthodes de résolution ont été développées dans ces cadres. Nous allons essayer de synthétiser une petite partie incontournable de ces recherches pour situer nos travaux. L'exhaustivité n'est pas notre objectif principal dans cet état de l'art. Le livre [Dec03] peut être consulté pour avoir de plus amples détails sur les techniques de résolution structurelles.

Le formalisme CSP est présenté dans la section 2.3, de même que quelques techniques majeures de résolution de problèmes représentés dans ce formalisme. Un accent particulier sera mis sur les méthodes structurelles et notamment la méthode BTD. Ceci est justifié par notre souhait de rendre opérationnelle ce type d'approches qui, dans la plupart des cas, obtient des résultats médiocres en pratique. Cette présentation des méthodes structurelles est étendue au formalisme VCSP et plus particulièrement à la méthode BTD-val dans la section 2.4.

Dans les deux cas, ces méthodes reposent sur des techniques de décomposition du graphe (ou hypergraphe) de contraintes des (V)CSP. Ces dernières correspondent à des recouvrements acycliques du problème permettant d'avoir de meilleures bornes de complexité théorique par rapport à celle du backtrack standard et des méthodes énumératives classiques. Dans la section 2.2, nous présentons les techniques de décomposition de graphes ou d'hypergraphes qui ont servi dans l'élaboration de méthodes structurelles de résolution de (V)CSP.

Nous allons donc commencer par faire des rappels sur les graphes et hypergraphes pour ensuite présenter des décompositions sur lesquelles sont basées plusieurs méthodes de résolution.

2.2 Graphes, hypergraphes et décompositions

Dans cette partie, nous allons rappeler des notions incontournables dans la suite de la théorie des graphes. La logique voudrait sans doute que les notions de graphe et hypergraphe soient présentées en même temps sachant qu'un graphe est un cas particulier d'hypergraphes. Cependant pour des raisons pédagogiques voire même historique, il est préférable de commencer par la définition de graphe.

2.2.1 Rappels sur les graphes

Dans toute la suite de ce travail, nous allons manipuler un ensemble fini d'éléments appelés sommets, noté $X = \{x_1, x_2, \ldots, x_n\}$.

Définition 2.2.1 *Un graphe orienté G est une paire (X, C) avec X l'ensemble des sommets du graphe et C l'ensemble de ses arcs.*

Les arcs du graphe sont des paires de sommets orientées (x_i, x_j) (arc de x_i vers x_j), $1 \leq i, j \leq n$.

Définition 2.2.2 *Un graphe non orienté G est une paire (X, C) avec X l'ensemble des sommets du graphe et C l'ensemble de ses arêtes.*

Une arête est un ensemble de deux sommets $\{x_i, x_j\}$, $1 \leq i \neq j \leq n$. Nous pouvons constater qu'il existe une orientation dans un arc alors qu'il en existe aucune dans une arête.

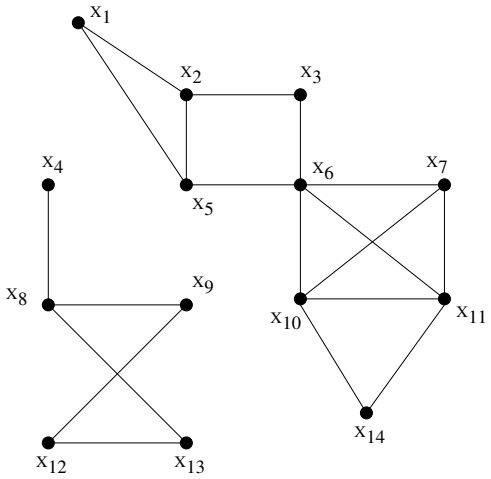

Fig. 2.1 – Un graphe non orienté.

Dans la suite, nous allons uniquement considérer des graphes non orientés. La figure 2.1 est une représentation graphique du graphe non orienté $G = (X, C)$ avec :
$X = \{x_1, x_2, \ldots, x_{14}\}$,
$C = \{\{x_1, x_2\}, \{x_1, x_5\}, \{x_2, x_3\}, \{x_2, x_5\}, \{x_3, x_6\}, \{x_4, x_8\}, \{x_5, x_6\}, \{x_6, x_7\}, \{x_6, x_{10}\}, \{x_6, x_{11}\}, \{x_7, x_{10}\}, \{x_7, x_{11}\}, \{x_8, x_9\}, \{x_8, x_{13}\}, \{x_9, x_{12}\}, \{x_{10}, x_{11}\}, \{x_{10}, x_{14}\}, \{x_{11}, x_{14}\}, \{x_{12}, x_{13}\}\}$.

Soit V un sous-ensemble de X.

Définition 2.2.3 *Le sous-graphe de G induit par V (noté $G[V]$) est le graphe (V, C') tel que $C' = \{\{x_i, x_j\} \in C | x_i \in V, x_j \in V\}$.*

Définition 2.2.4 *Deux sommets x_i et x_j sont voisins dans G si $\{x_i, x_j\} \in C$.*

Définition 2.2.5 *On appelle voisinage d'un sommet x_i dans G, l'ensemble*
$N(x_i) = \{x_j \in X |$ x_j et x_i sont voisins dans $G\}$.
On appelle voisinage fermé de x_i, l'ensemble $N[x_i] = N(x_i) \cup \{x_i\}$.
Pour un sous-ensemble de sommets V, le voisinage de V est l'ensemble $N(V) = \bigcup_{x \in V} N(x) - V$ et son voisinage fermé est l'ensemble $N[V] = \bigcup_{x \in V} N[x]$.

Définition 2.2.6 *Une chaîne entre deux sommets x_i et x_j dans G est une séquence $(x_{u_1}, x_{u_2}, \ldots, x_{u_p})$ de sommets de G telle que :*
- *$x_{u_1} = x_i$ et $x_{u_p} = x_j$*
- *$\forall v, 1 \leq v \leq p-1$, x_{u_v} et $x_{u_{v+1}}$ sont voisins.*

Deux sommets sont donc mutuellement accessibles s'il existe une chaîne qui les relie. La définition suivante va introduire une notion centrale : la connexité.

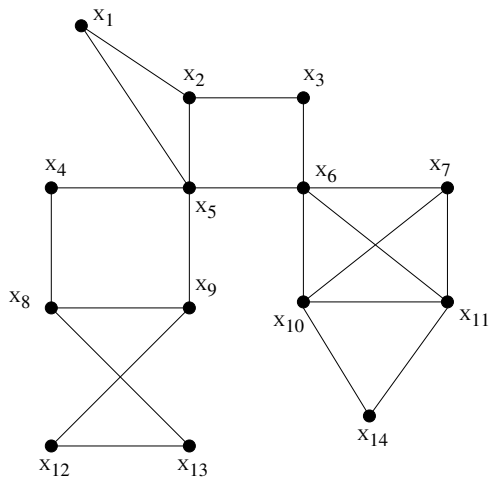

Fig. 2.2 – Un graphe connexe.

Définition 2.2.7 *G est connexe ssi tous ses sommets sont mutuellement accessibles.*

Le graphe de la figure 2.1 n'est pas connexe contrairement à celui de la figure 2.2.

Un graphe non connexe peut être décomposé en plusieurs sous-graphes connexes donnés par les composantes connexes.

Définition 2.2.8 *$V \subset X$ est une composante connexe de G ssi $G[V]$ est un graphe connexe et il n'existe aucun ensemble V' tel que $V \subsetneq V'$ et $G[V']$ est un graphe connexe.*

La relation d'accessibilité mutuelle étant une relation d'équivalence, les composantes connexes représentent donc les composantes maximales de cette relation.

Les ensembles $V = \{x_4, x_8, x_9, x_{12}, x_{13}\}$ et $V' = \{x_1, x_2, x_3, x_5, x_6, x_7, x_{10}, x_{11}, x_{14}\}$ sont les composantes connexes du graphe non connexe de la figure 2.1.

Dans le cadre de cette étude, un graphe ayant plusieurs composantes connexes est traité comme un ensemble de graphes équivalents aux composantes. Nous allons donc supposer que les graphes considérés dans la suite sont connexes.

Définition 2.2.9 *Soient $G = (X, C)$ un graphe, x_i et x_j deux sommets de G. $S \subset X$ est un $\{x_i, x_j\}$-séparateur ssi x_i et x_j sont dans deux composantes connexes différentes de $G[X - S]$.*
S est un $\{x_i, x_j\}$-séparateur minimal ssi il n'existe aucun ensemble S' tel que $S' \subsetneq S$ et S' est également un $\{x_i, x_j\}$-séparateur.
On appelle séparateur de G tout $\{x_i, x_j\}$-séparateur.
On appelle séparateur minimal de G tout $\{x_i, x_j\}$-séparateur minimal.

Donc S est séparateur de G ssi $G[X - S]$ admet au moins deux composantes connexes. Les composantes connexes de $G[X - S]$ sont dites induites par S. Un séparateur minimal de G peut contenir un autre séparateur de G. En effet, S est un séparateur minimal de G si S est un séparateur de G qui n'en contient pas un autre dont les composantes connexes induites contiennent celles de S. Cette observation donne une caractérisation différente de la minimalité des séparateurs : S est un séparateur minimal de G s'il induit au moins deux composantes connexes CC_1 et CC_2 telles que : $N(CC_1) = N(CC_2) = S$.

L'ensemble de sommets $\{x_5, x_6\}$ du graphe de la figure 2.2, est un séparateur minimal qui induit trois composantes connexes $\{x_1, x_2, x_3\}$, $\{x_4, x_8, x_9, x_{12}, x_{13}\}$ et $\{x_7, x_{10}, x_{11}, x_{14}\}$. $\{x_4, x_5, x_6\}$ est également un séparateur, mais il n'est pas minimal. Ses composantes connexes induites $\{x_1, x_2, x_3\}$, $\{x_8, x_9, x_{12}, x_{13}\}$ et $\{x_7, x_{10}, x_{11}, x_{14}\}$ sont contenues dans celles de $\{x_5, x_6\}$. Par contre $\{x_7, x_{10}\}$ n'est pas un séparateur car son retrait du graphe le laisse connexe.

Une des notions les plus importantes dans cette thèse est sans nul doute l'acyclicité. Elle permet en effet d'utiliser des algorithmes très efficaces pour résoudre des problèmes difficiles en général.

Définition 2.2.10 *Un cycle dans G est une chaîne $(x_{u_1}, x_{u_2}, \ldots, x_{u_p}), p \geq 4$ qui contient au moins trois sommets distincts, et tel que $x_{u_1} = x_{u_p}$.*

Définition 2.2.11 *Un graphe G est acyclique ssi il ne contient pas de cycle.*

Un arbre est un graphe connexe et acyclique. Si on retire la propriété de connexité à un arbre, on se retrouve avec la définition d'une forêt.

2.2.2 Rappels sur les hypergraphes

Comme dans la section précédente, $X = \{x_1, x_2, \ldots, x_n\}$ représente un ensemble fini de sommets. Une hyperarête est un sous-ensemble non vide de X. La différence entre une arête et une hyperarête réside dans l'arité, c'est-à-dire le nombre de sommets dans l'ensemble. Une arête est d'arité 2, alors que celle d'une hyperarête est quelconque. Une arête est donc un cas particulier d'hyperarête.

Définition 2.2.12 *Un hypergraphe H est une paire (X, C) avec X l'ensemble des sommets du graphe et C l'ensemble de ses hyperarêtes.*

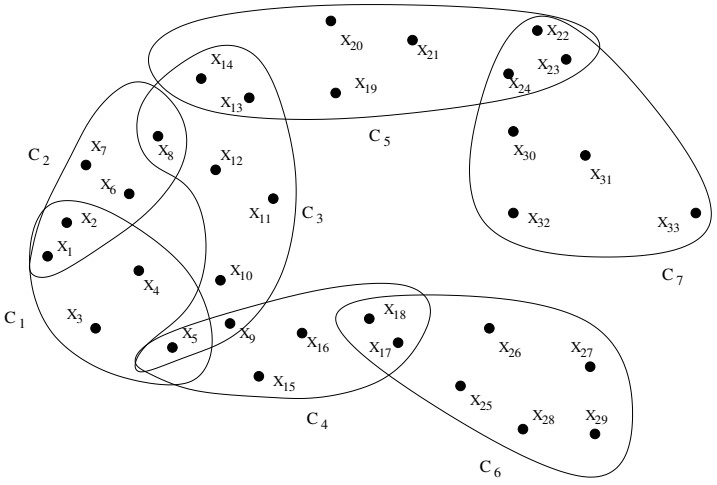

Fig. 2.3 – Un hypergraphe.

Un hypergraphe qui ne contient que des hyperarêtes d'arité 2 est un graphe. La notion de graphe est donc un cas particulier d'hypergraphe. Dans certains cas, il peut être beaucoup plus facile de travailler sur une représentation d'un hypergraphe en graphe. C'est le cas, comme nous le verrons dans le prochain chapitre, du calcul de décompositions arborescentes ou plus généralement de recouvrements acycliques. Dans ce cas, nous préférons utiliser la 2-section de l'hypergraphe.

Définition 2.2.13 *[Ber70] La 2-section d'un hypergraphe $H = (X, C)$ est un graphe $G = (X, C')$ ayant le même ensemble de sommets et il existe une arête entre deux sommets du graphe s'il existe une hyperarête de H contenant les deux sommets.*

Définition 2.2.14 *Soit V un sous-ensemble de X. L'ensemble des hyperarêtes partielles induites par V dans H est $C' = \{c' | c' = c \cap V, c \in C\} - \{\emptyset\}$.*

Dans la suite, nous allons uniquement considérer des hypergraphes réduits, c'est-à-dire qui ne possèdent aucune hyperarête contenue dans une autre.

La figure 2.3 est une représentation graphique d'un hypergraphe $H = (X, C)$ avec : $X = \{x_1, x_2, \ldots, x_{33}\}$ et $C = \{c_1, c_2, \ldots, c_7\}$ où $c_1 = \{x_1, x_2, x_3, x_4, x_5\}$, $c_2 = \{x_1, x_2, x_6, x_7, x_8\}$, $c_3 = \{x_5, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}\}$, $c_4 = \{x_5, x_9, x_{15}, x_{16}, x_{17}, x_{18}\}$, $c_5 = \{x_{13}, x_{14}, x_{19}, x_{20}, x_{21}, x_{22}, x_{23}, x_{24}\}$, $c_6 = \{x_{17}, x_{18}, x_{25}, x_{26}, x_{27}, x_{28}, x_{29}\}$, $c_7 = \{x_{22}, x_{23}, x_{24}, x_{30}, x_{31}, x_{32}, x_{33}\}$.

Définition 2.2.15 *Soit V un sous-ensemble de X. L'ensemble des hyperarêtes partielles induites par V dans H est $C' = \{c', c' = c \cap V, c \in C\} - \{\emptyset\}$.*

On dit de cet ensemble qu'il est généré par des sommets. Comme dans le cas des graphes, il est possible de définir la notion de sous-hypergraphe induit.

Définition 2.2.16 *Soit $V \subset X$. Le sous-hypergraphe de H induit par V est l'hypergraphe $H' = (V, C')$ avec C', l'ensemble des hyperarêtes partielles induites par V dans H.*

Définition 2.2.17 *Une chaîne entre deux sommets x_i et x_j dans G est une séquence d'hyperarêtes $(c_{u_1}, c_{u_2}, \ldots, c_{u_p})$ de G telle que :*
- *$x_i \in c_{u_1}$ et $x_j \in c_{u_p}$*
- *$\forall v, 1 \leq v \leq p-1$, $c_{u_v} \cap c_{u_{v+1}} \neq \emptyset$.*

De même, une chaîne entre deux hyperarêtes c_i et c_j dans G est une séquence d'hyperarêtes $(c_{u_1}, c_{u_2}, \ldots, c_{u_p})$ de G telle que :
- *$c_{u_1} = c_i$ et $c_{u_p} = c_j$*
- *$\forall v, 1 \leq v \leq p-1$, $c_{u_v} \cap c_{u_{v+1}} \neq \emptyset$.*

Deux hyperarêtes sont mutuellement accessibles dans H s'il existe une chaîne dans H qui les relie. Deux sommets $x_i \in c_i$ et $x_j \in c_j$ sont mutuellement accessibles si $c_i = c_j$ ou s'il existe une chaîne qui relie c_i et c_j. La définition de connexité est similaire au cas des graphes.

Définition 2.2.18 *Soit $V \subset X$. V est connexe ssi tous ses sommets sont mutuellement accessibles. De même, soit $\mathcal{C} \subset C$, $\mathcal{C}$ est connexe ssi toutes ses hyperarêtes sont mutuellement accessibles.*

Définition 2.2.19 *Un hypergraphe H est connexe ssi toutes ses hyperarêtes sont mutuellement accessibles.*

Les composantes connexes sont les composantes maximales de la relation d'équivalence d'accessibilité mutuelle.

Définition 2.2.20 *$CC \subset C$ est une composante connexe de H ssi $H[V]$, avec $V = \bigcup_{c \in CC} c$ est un hypergraphe connexe et il n'existe aucun ensemble CC' tel que $CC \subsetneq CC'$ et $H[V']$, avec $V' = \bigcup_{c \in CC} c$, est un hypergraphe connexe.*

Elles constituent donc des ensembles maximaux d'hyperarêtes mutuellement accessibles. Nous allons donc supposer comme dans le cas des graphes que les hypergraphes considérés sont connexes.

Définition 2.2.21 *Soient $\mathcal{C}$ un ensemble connexe, réduit, d'hyperarêtes partielles, c_1 et c_2 deux éléments de $\mathcal{C}$ et $q = c_1 \cap c_2$. q est une articulation de $\mathcal{C}$ si le retrait de q de toutes les hyperarêtes de $\mathcal{C}$ décompose $\mathcal{C}$ en au moins deux composantes connexes.*

Dans l'hypergraphe $H = (X, C)$ de la figure 2.3, $q = c_4 \cap c_6 = \{x_{17}, x_{18}\}$ est une articulation de C. En effet, son retrait de H induit deux composantes connexes :
$\{\{x_1, x_2, x_3, x_4, x_5\}, \{x_1, x_2, x_6, x_7, x_8\}, \{x_5, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}\}, \{x_5, x_9, x_{15}, x_{16}\}$, $\{x_{13}, x_{14}, x_{19}, x_{20}, x_{21}, x_{22}, x_{23}, x_{24}\}, \{x_{22}, x_{23}, x_{24}, x_{30}, x_{31}, x_{32}, x_{33}\}\}$ et $\{\{x_{25}, x_{26}, x_{27}, x_{28}, x_{29}\}\}$.

La notion d'acyclicité définie de manière très naturelle à partir de la notion de cycle dans les graphes, admet de manière surprenante des généralisations multiples au niveau des hypergraphes. Toutes ces définitions, qui ont été introduites essentiellement dans le cadre de travaux sur les Bases de Données Relationnelles (on peut citer également les travaux sur la cyclicité de Berge [Ber70]), cherchent à capturer des propriétés sur ces dernières. La plus connue et sans doute la plus utilisée est celle d'α-acyclicité. Nous allons nous contenter de rappeler cette définition dans un premier temps pour mieux y revenir dans la suite et voir les applications possibles de cette propriété.

Définition 2.2.22 *Un hypergraphe H est $\alpha-$acyclique si tout ensemble d'hyperarêtes partielles, qui est connexe, réduit, induit par un ensemble de sommets et qui n'admet pas d'articulation, est trivial (ne contient qu'un unique élément).*

L'hypergraphe H de la figure 2.3 n'est pas acyclique car l'ensemble d'hyperarêtes partielles $\{\{x_1, x_2, x_3, x_4, x_5\}, \{x_1, x_2, x_6, x_7, x_8\}, \{x_5, x_8, x_{10}, x_{11}, x_{12}\}\}$ est connexe, réduit, non trivial et n'admet pas d'articulation.
Par contre celui de la figure 2.4 est acyclique.

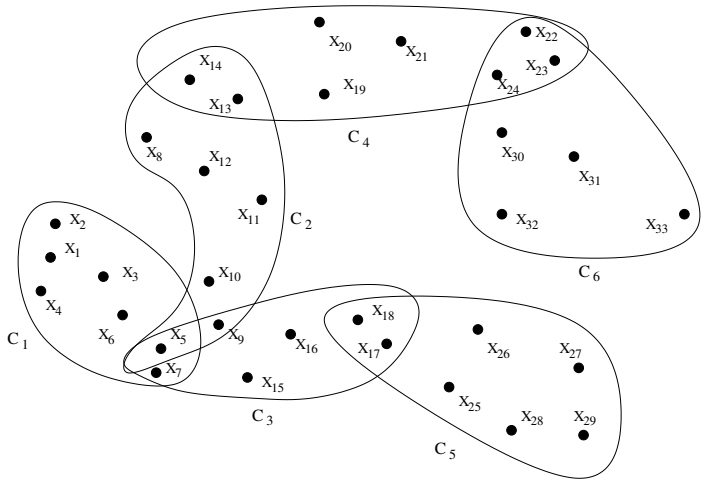

Fig. 2.4 – Un hypergraphe acyclique.

Cette définition admet un grand nombre de définitions équivalentes présentées dans [BFMY83]. Nous allons en relever deux qui revêtent une importance certaine dans la suite.

Définition 2.2.23 *Un arbre des jointures d'un hypergraphe H est un arbre connexe T dont les nœuds sont les hyperarêtes de H tels que si un sommet x de H est contenu dans deux hyperarêtes c_i et c_j de H alors il est contenu dans l'ensemble des nœuds de l'unique chaîne reliant c_i et c_j dans T.*

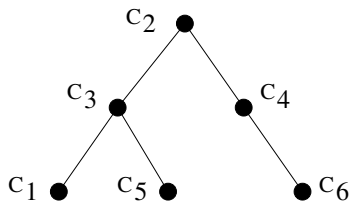

Fig. 2.5 – Un arbre de jointure de l'hypergraphe acyclique de la figure 2.4.

En d'autres termes, l'ensemble des nœuds contenant x induit un sous-arbre connexe de T.

La figure 2.5 présente un arbre de jointure de l'hypergraphe de la figure 2.4. H vérifie la condition de running intersection s'il existe un ordre σ sur ses hyperarêtes tel que pour tout $2 \leq \sigma(i) \leq m$, il existe $\sigma(i_0) < \sigma(i)$ tel que $c_{\sigma(i)} \cap (\bigcup_{\sigma(j)<\sigma(i)} c_{\sigma(j)}) \subset c_{\sigma(i_0)}$. L'intersection d'une hyperarête donnée avec celles qui la précèdent dans l'ordre est contenue dans l'une de ces hyperarêtes.

L'hypergraphe de la figure 2.4 vérifie la running intersection. Avec l'ordre $\sigma = (c_2, c_3, c_1, c_5, c_4, c_6)$ sur ses hyperarêtes, nous avons bien l'intersection d'une hyperarête avec ces prédécesseurs dans l'ordre qui est contenue dans l'un d'entre eux.

Théorème 2.2.1 *[BFMY83] H est α-acyclique ssi :*
- *H admet un arbre de jointures*
- *H vérifie la condition de running intersection*

Un hyperarbre est un hypergraphe connexe et α-acyclique.

2.2.3 Décompositions de (hyper)graphes

Cette partie a pour objectif de rappeler quelques décompositions de graphes et hypergraphes proposées lors de ces dernières années. Ces décompositions donnent pour la plupart des arrangements en hypergraphe acyclique du graphe ou de l'hypergraphe de départ. Ces arrangements s'effectuent par regroupement de sommets ou d'hyperarêtes. Le but principal est de profiter des propriétés très intéressantes des hypergraphes acycliques au niveau de la résolution de certains problèmes difficiles. Aussi, nous allons nous restreindre aux décompositions pertinentes dans le cadre de notre étude, essentiellement celles qui ont conduit à des méthodes de résolution de (V)CSP. Nous verrons dans les parties de cet état de l'art consacrées aux (V)CSP que ces décompositions permettent d'avoir des bornes de complexité théorique différentes dans le cadre de la résolution de ces problèmes. La puissance d'une décomposition fait référence donc à la qualité de la borne de complexité qu'elle induit. En utilisant ce critère, [GLS00] définit une hiérarchie sur les décompositions.

2.2.3.1 Décompositions de graphes

Décompositions arborescentes ([RS86]).

Définition 2.2.24 *Etant donné un graphe $G = (X, C)$, une décomposition arborescente de G est un couple $(E, \mathcal{T})$ où $\mathcal{T} = (I, F)$ est un arbre avec I l'ensemble des nœuds et F celui des arêtes et $E = \{E_i : i \in I\}$ une famille de sous-ensembles de X, qui vérifie :*
- *$\cup_{i \in I} E_i = X$,*
- *pour chaque arête $\{x, y\} \in C$, il existe $i \in I$ avec $\{x, y\} \subset E_i$, et*
- *pour tout $i, j, k \in I$, si k est sur une chaîne allant de i à j dans $\mathcal{T}$, alors $E_i \cap E_j \subset E_k$.*

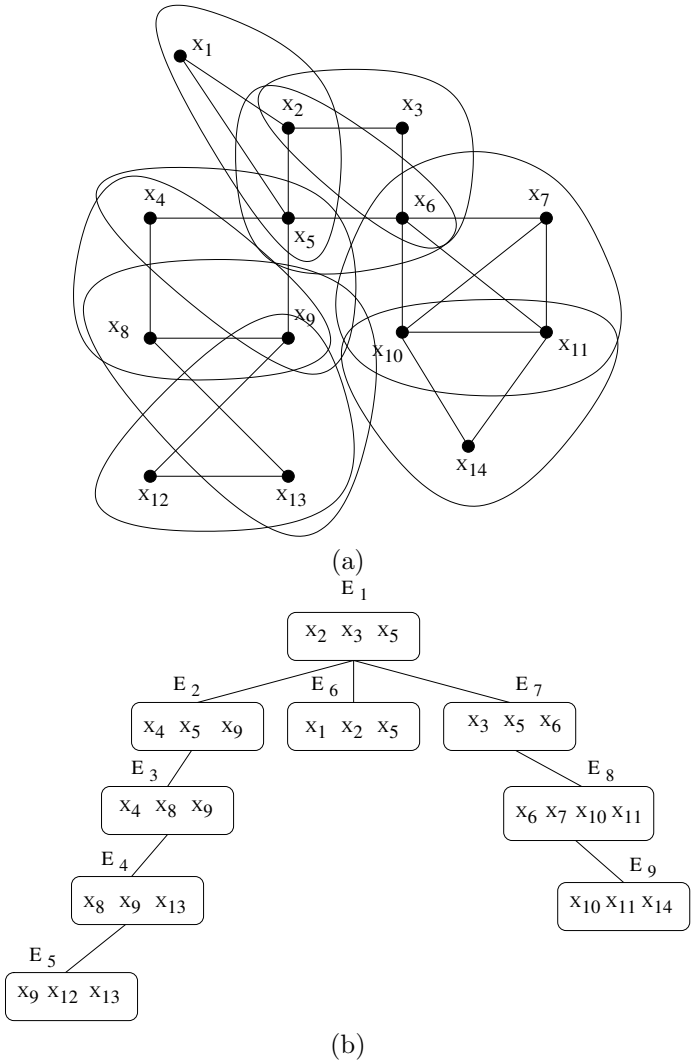

Fig. 2.6 – (a) Un recouvrement acyclique du graphe de la figure 2.2 et (b) une décomposition arborescente basée sur ce recouvrement.

Les nœuds E_i de $\mathcal{T}$ sont appelés clusters. La largeur d'une décomposition arborescente $(E, \mathcal{T})$ est égale à $max_{i \in I}|E_i| - 1$. La *largeur d'arbre* (ou *tree-width*) de G, notée w, est la largeur minimale sur toutes les décompositions arborescentes de G. La figure 2.6(a) présente le recouvrement des arêtes du graphe de la figure 2.2, qui définit la décomposition arborescente de ce même graphe dans la figure 2.6(b).

La décomposition arborescente est une des plus puissantes dans la hiérarchie des décompositions.

Composantes biconnexes ([Eve79]. La décomposition de graphes en composantes biconnexes repose sur une généralisation de la définition de connexité.

Définition 2.2.25 *Un graphe $G = (X, C)$ est k-connexe si G reste connexe après la suppression d'un ensemble quelconque de $k - 1$ de ses sommets.*

Une composante k-connexe V est un sous-ensemble maximal de X qui induit un sous-graphe de G k-connexe ($G[V]$ est k-connexe). Le graphe des composantes k-connexes, obtenu en représentant

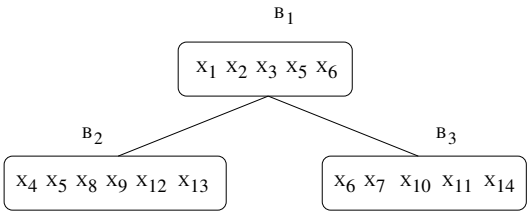

Fig. 2.7 – Une décomposition biconnexe du graphe de la figure 2.2.

chaque composante comme un nœud du graphe et en ajoutant une arête entre deux composantes si elles partagent un sommet, est un arbre. Ici nous allons considérer le cas où $k = 2$. Dans ce cas, les composantes 2-connexes (ou biconnexes) sont appelées bicomposantes. On parle ainsi de graphe biconnexe et de bicomposantes qui demeurent connexes après la suppression de n'importe lequel de leurs sommets. Le graphe des bicomposantes est donc appelé arbre BCC de G.
La figure 2.7 est une représentation graphique d'une décomposition BCC du graphe de la figure 2.2.

La décomposition biconnexe est moins puissante que la décomposition arborescente dans la hiérarchie des décompositions.

Pseudo-arbre ([FQ85]). Un pseudo-arbre $\mathcal{T} = (X, C')$ de $G = (X, C)$, est une arborescence enraciné tel que toute arête de C n'appartenant pas à C' relie un sommet de X avec un de ses ancêtres dans $\mathcal{T}$. La notion de pseudo-arbre permet de détecter les indépendances entre sous-problèmes.

La décomposition induite par un pseudo-arbre est au même niveau que la décomposition arborescente dans la hiérarchie des décompositions.

Arrangement en arbre ([Gav77]). Un arrangement en arbre orienté d'un graphe $G = (X, C)$, est un arborescence enraciné $\mathcal{T} = (X, C')$ tel que deux sommets adjacents de G soient dans une même branche de $\mathcal{T}$ qui est une chemin de la racine à une feuille de l'arbre. Cette notion est très proche de celle de pseudo-arbre et rend également compte des indépendances entre sous-problèmes.

Comme pour le pseudo-arbre, la décomposition induite par un arrangement est au même niveau que la décomposition arborescente dans la hiérarchie des décompositions.

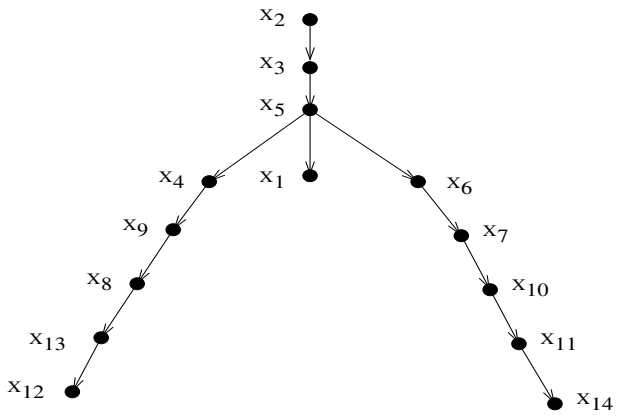

Fig. 2.8 – Un pseudo-arbre et un arrangement en arbre orienté du graphe de la figure 2.2.

2.2.3.2 Décompositions d'hypergraphes

Décomposition Hinge ([GP84]). La décomposition Hinge repose sur une définition de connexité différente de la définition usuelle. Cette nouvelle définition restreint la connexité usuelle en considérant un sous-ensemble d'hyperarêtes qui fait parfois office d'articulation.

Soient $H = (X, C)$ un hypergraphe connexe, C' et C'' deux sous-ensembles disjoints de C.

Définition 2.2.26 *C'' est a-connecté par rapport à C' si $\forall e \in C'', \forall f \in C''$, il existe une séquence $c_1, \ldots, c_n$ d'hyperarêtes de C'' telle que $c_1 = e, c_n = f$ et $c_i \cap c_{i+1}$ n'est pas incluse dans $var(C')$ ($var(C')$ représente l'ensemble des sommets contenues dans les hyperarêtes de C').*

La a-connexité est plus forte que la connexité sur les hypergraphes. C'' est connexe, mais en plus les intersections entre les éléments d'une chaîne entre deux de ses hyperarêtes ne sont pas contenues dans C', ce dernier jouant un peu le rôle de séparateur.

Définition 2.2.27 *Soit $C' \subset C$. $CC \subset C - C'$ est une composante a-connexe induite par C' si CC est a-connexe par rapport à C' et il n'existe pas un autre $CC' \subset C - C'$ tel que $CC \subsetneq CC'$ et CC' est a-connexe par rapport à C'.*

Définition 2.2.28 *Soient $C' \subset C$ contenant au moins deux hyperarêtes et $CC_1, \ldots, CC_m$ les composantes a-connexes induites par C'. C' est un hinge si pour tout $i = 1, \ldots, m$, il existe une hyperarête $c_i \in C'$ telle que $var(CC_i) \cap var(C') \subset c_i$. Un hinge est minimal s'il ne contient aucun autre hinge.*

Un Hinge est donc un ensemble d'hyperarêtes dont l'intersection avec une composante a-connexe qu'il induit est contenue dans une de ses hyperarêtes. Cette hyperarête contient donc une articulation qui sépare le reste du Hinge et le reste de l'hypergraphe.

Définition 2.2.29 *Une décomposition hinge de H est un arbre $\mathcal{T}$ vérifiant :*
- *les nœuds de $\mathcal{T}$ sont des hinges minimaux de G,*
- *chaque hyperarête de C est contenue dans au moins un nœud de $\mathcal{T}$,*
- *deux nœuds adjacents A et B de $\mathcal{T}$ partagent exactement une hyperarête $c_i \in C$, $c_i = var(A) \cap var(B)$,*
- *les variables dans l'intersection de deux nœuds de l'arbre $\mathcal{T}$ sont contenues dans chaque nœud sur la chaîne les reliant.*

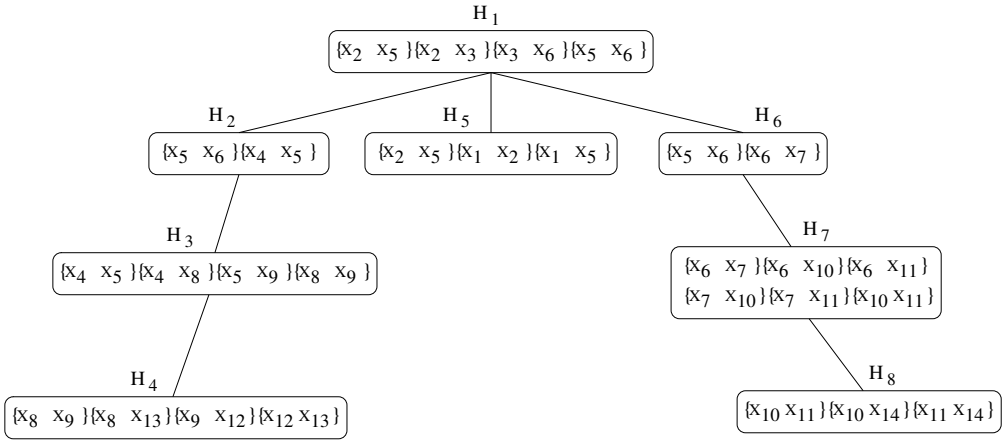

Fig. 2.9 – Une décomposition hinge du graphe de la figure 2.2.

On appelle la Hinge width w_H d'un hypergraphe H la taille maximale des nœuds d'une décomposition hinge : c'est un invariant de H appelé degré de cyclicité. Pour une décomposition hinge donnée, les nœuds de l'arbre sont des hinges minimaux. Dans le cas des graphes, une décomposition hinge peut être vue comme une décomposition arborescente en remplaçant chaque nœud C' de l'arbre par $var(C')$.
Une décomposition hinge du graphe de la figure 2.2 est donnée dans la figure 2.9.

La décomposition Hinge se situe au même niveau que la décomposition arborescente dans la hiérarchie des décompositions.

La décomposition en hyperarbre ([GLS00]). La décomposition en hyperarbre peut être vue comme une généralisation de la décomposition arborescente et de la Hinge décomposition.

Définition 2.2.30 *Soit $H = (X, C)$ un hypergraphe. Un hyperarbre de H est un triplet (T, χ, λ), avec $T = (N, B)$ un arbre enraciné, χ et λ des applications qui à chaque sommet $p \in N$ de T associent deux ensembles $\chi(p) \subset X$ et $\lambda(p) \subset C$.*

Un arbre enraciné est un arbre dans lequel un sommet est différencié et appelé racine de l'arbre. Ce sommet racine donne une orientation à l'arbre et donne ainsi naissance à une arborescence. Dans le cas de la définition d'hyperarbre de H, T_p désigne le sous-arbre enraciné en un sommet p de T .

Définition 2.2.31 *Une décomposition en hyperarbre HD d'un hypergraphe $H = (X, C)$ est un hyperarbre (T, χ, λ) de H qui vérifie les conditions suivantes :*
- *Pour toute hyperarête $c_i \in C$ de H, il existe un sommet $p \in N$ de T tel que $c_i \subset \chi(p)$ (p recouvre c)*
- *Pour tout sommet $x_i \in X$ de H, l'ensemble $\{p \in N, x_i \in \chi(p)\}$ induit un sous-arbre de T (donc graphe connexe)*
- *Pour tout $p \in N$, $\chi(p) \subset var(\lambda(p))$*
- *Pour tout $p \in N$, $var(\lambda(p)) \cap \chi(T_p) \subset \chi(p)$.*

La dernière inclusion est en fait une égalité, sachant que la troisième implique l'inclusion inverse. Une hyperarête $c_i \in C$ est fortement recouverte dans HD s'il existe un sommet p de T tel que $c_i \subset \chi(p)$ et $c_i \in \lambda(p)$. Dans ce cas, on dit que p recouvre fortement c_i. Une décomposition en hyperarbre HD de H est une décomposition complète de H si chaque hyperarête de H est fortement recouverte dans HD. Les décompositions complètes seront les seules pertinentes dans notre étude.

La largeur d'une décomposition en hyperarbre (T, χ, λ) est $max_{p \in N}|\lambda(p)|$. La hypertree-width $hw(H)$ de H est la largeur minimum sur l'ensemble de ses décompositions en hyperarbre.

La figure 2.10 est une représentation graphique d'une décomposition en hyperarbre du graphe de la figure 2.2. En (a), nous avons à la fois T et les ensembles $\chi(p)$, p étant un sommet de T. (b) donne également T et les ensembles $\lambda(p)$.

La décomposition en hyperarbre est supérieure à la décomposition arborescente dans la hiérarchie des décompositions.

Décompositions encadrées et Spread cut décomposition ([CJG05]). Les décompositions encadrées donnent un cadre générique de décompositions qui capturent les différentes décompositions déjà présentées moyennant quelques propriétés additionnelles. Ce cadre permet également de définir une nouvelle décomposition : la Spread cut décomposition.

Définition 2.2.32 *Un bloc encadré d'un hypergraphe $H = (X, C)$ est une paire (λ, χ) telle que le cadre λ est un sous-ensemble de C et le bloc χ est un sous-ensemble des sommets du cadre.*

Définition 2.2.33 *Un bloc encadré (λ, χ) d'un hypergraphe H recouvre une hyperarête c_i si c_i est contenue dans χ.*

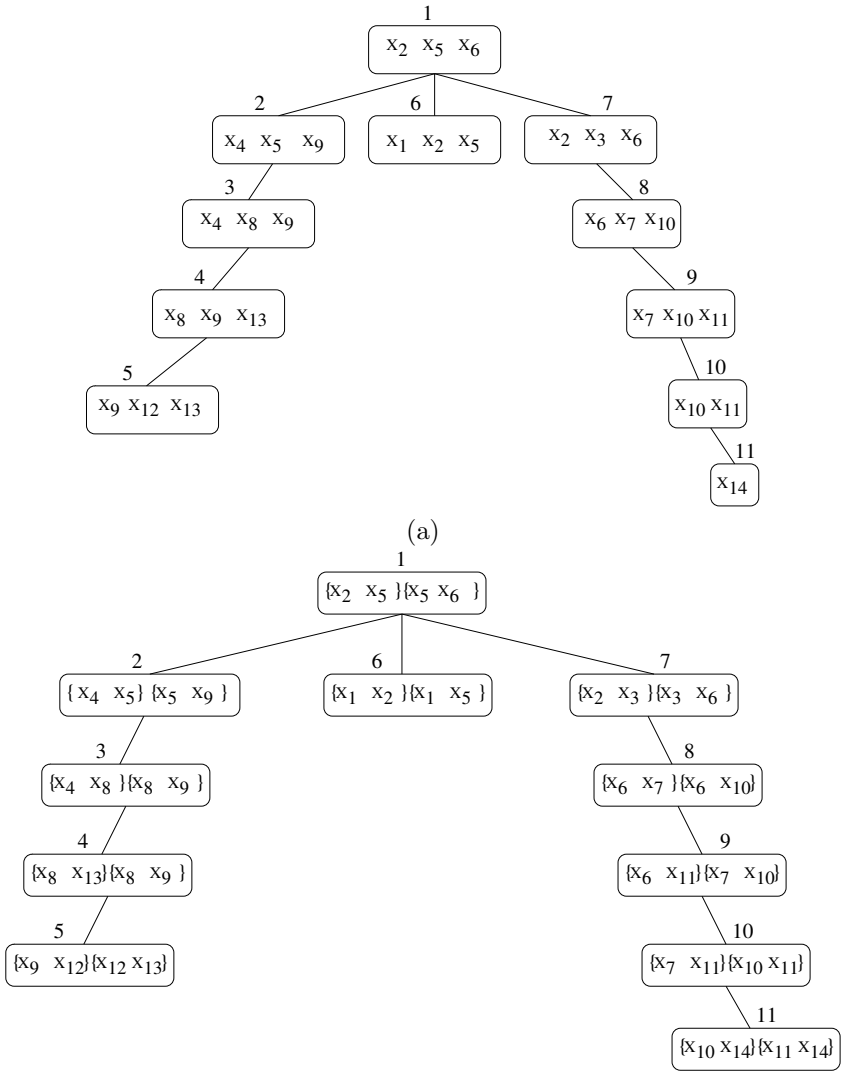

Fig. 2.10 – Une décomposition en hyperarbre du graphe de la figure 2.2 avec en (a) l'arbre défini par χ et en (b) celui défini par λ.

Un ensemble de blocs encadrés GB d'un hypergraphe H est appelé recouvrement encadré de H si toute hyperarête de H est recouverte par un bloc encadré de GB.
Un ensemble de blocs encadrés GB d'un hypergraphe H est appelé recouvrement encadré complet de H si toute hyperarête c_i de H apparaît dans le cadre d'un bloc encadré de GB recouvrant c_i.

Définition 2.2.34 *Une décomposition encadrée d'un hypergraphe H est un ensemble de blocs encadrés GB de H qui définit un recouvrement encadré complet de H.*

Comme dans le cas des hypergraphes, l'arbre des jointures d'un ensemble de blocs encadrés GB d'un hypergraphe H, est un arbre connexe T dont les nœuds sont les éléments de GB, et l'ensemble des nœuds dont le bloc contient un sommet x induit un sous-arbre connexe de T. Un ensemble de blocs encadrés GB d'un hypergraphe H est acyclique s'il admet un arbre des jointures (théorème 2.2.1).

Théorème 2.2.2 *Si l'ensemble GB de blocs encadrés est un recouvrement encadré acyclique de H alors l'ensemble $GB \cup \{(\{c\}, c) | c \in C\}$ est une décomposition encadrée acyclique de H.*

Définition 2.2.35 *Soit $H = (X, C)$ un hypergraphe. Un recouvrement encadré de $(X, C \cup \{\{x\} | x \in X\})$ est appelé recouvrement encadré étendu de H.*

Définition 2.2.36 *Un recouvrement encadré GB d'un hypergraphe H est défini par les hyperarêtes si le bloc de chaque bloc encadré de GB contient exactement l'ensemble des sommets reliés par les hyperarêtes de son cadre.*

Avec ce schéma générique, il est possible de capturer simplement plusieurs décompositions déjà définies.

- Une décomposition BCC d'un hypergraphe est un recouvrement encadré défini par les hyperarêtes, complet, acyclique GB qui vérifie une condition d'articulation :
  $\forall (\lambda_1, \chi_1), (\lambda_2, \chi_2) \in GB, |\chi_1 \cap \chi_2| \leq 1$
- Une cycle-hypercutset décomposition d'un hypergraphe $H = (X, C)$ est un recouvrement encadré défini par les hyperarêtes, complet, acyclique GB qui vérifie une condition de simplicité :
  $\exists C' \subset C, \forall (\lambda, \chi) \in GB, |\lambda - C'| \leq 1$
- Une décomposition Hinge d'un hypergraphe est un recouvrement encadré défini par les hyperarêtes, complet, acyclique GB qui vérifie une condition de séparation :
  $\forall (\lambda_1, \chi_1), (\lambda_2, \chi_2) \in GB, \exists c \in \lambda_1 \cap \lambda_2, \chi_1 \cap \chi_2 \subset c$
- Une cycle-cutset décomposition d'un hypergraphe $H = (X, C)$ est un recouvrement encadré étendu, défini par les hyperarêtes, acyclique GB tel que chaque hyperarête de chaque bloc encadré de GB est un unique sommet, et GB vérifie une condition de simplicité :
  $\exists C' \subset C, \forall (\lambda, \chi) \in GB, \exists c \in C, (\chi - C') \subset c$
- Une query décomposition d'un hypergraphe $H = (X, C)$ est une paire (GB, T) avec GB un recouvrement encadré étendu, complet, défini par les hyperarêtes, acyclique de H et T un arbre de jointures de GB qui vérifie une condition de connexité :
  $\forall c \in C, \{(\lambda, \chi) \in GB | c \in \lambda\}$ est connexe dans T.
- Une décomposition en hyperarbre d'un hypergraphe $H = (X, C)$ est une paire (GB, T) avec GB un recouvrement encadré acyclique de H et T un arbre de jointures enraciné de GB qui vérifie cette condition de descendance :
  $\forall (\lambda, \chi) \in GB, ((\bigcup_{c \in \lambda} c) \cap \bigcup_{(\lambda_i, \chi_i) \in D(\lambda, \chi)} \chi_i) \subset \chi$ avec $D(\lambda, \chi)$ l'ensemble des descendants de (λ, χ) dans T.

Définition 2.2.37 *Soient $H = (X, C)$ un hypergraphe et $\chi \subset X$ un ensemble de sommets. Deux hyperarêtes e et f de H sont dites χ-adjacentes si $e \cap f \not\subset \chi$.*
Une χ-chaîne reliant deux hyperarêtes e et f est une séquence d'hyperarêtes $e = c_0, c_1, \ldots, c_r = f$

telle que c_i est χ-adjacente à c_{i+1} pour tout $i = 0, \ldots, r-1$.
Un ensemble d'hyperarêtes $C' \subset C$ est χ-connexe s'il existe une chaîne reliant toute paire d'hyperarêtes de C'.
Un ensemble d'hyperarêtes $C' \subset C$ est une χ-composante d'hyperarêtes de H s'il est un sous-ensemble non vide, maximal, χ-connexe de C (il existe une chaîne reliant toute paire d'hyperarêtes de C').
Un ensemble non vide, C' de sommets de H est une χ-composante de sommets de H s'il existe une hyperarête χ-composante C'_χ pour laquelle $C' = \bigcup C'_\chi - \chi$.

Définition 2.2.38 *Un bloc encadré (λ, χ) d'un hypergraphe H admet des composantes compactes si chaque χ-composante de H intersecte au plus une $\bigcup\lambda$-composante de H et $\{c_1 \cap c_2 | c_1, c_2 \in \lambda\} \subset \chi$.*
Un spread cut de H est un recouvrement encadré acyclique de H dans lequel tout bloc encadré a des composantes compactes.

Le spread cut est au même niveau que la décomposition en hyperarbre dans la hiérarchie des décompositions.

2.3 Problème de satisfaction de contraintes : CSP

La notion de contraintes donne un cadre de formalisation très expressif de plus en plus largement utilisé dans de nombreux travaux en intelligence artificielle, notamment ceux concernant les problèmes de satisfaction de contraintes (Constraint Satisfaction Problem : CSP). Le formalisme CSP, tout en permettant de représenter très simplement des problèmes, est extraordinairement puissant dans la mesure où il est possible de représenter un éventail très large de problèmes. En outre, l'utilité de ce formalisme, en dehors de son expressivité, réside dans l'efficacité des méthodes de résolution qui y ont été développées durant les 30 dernières années.
Une contrainte est une propriété sur différents objets qui donnent des restrictions sur les valeurs simultanées de ces objets. On peut ainsi représenter des problèmes académiques purement combinatoires, des problèmes d'analyse de scènes, de contraintes temporelles, de recherche opérationnelle, d'ordonnancement, de planification, de configuration,..., grâce à des contraintes algébriques, temporelles, géométriques...

2.3.1 Présentation du formalisme

Un problème de satisfaction de contraintes est défini par un ensemble de variables, associée chacune à un domaine discret de valeurs de taille finie et par un ensemble de contraintes qui mettent en relation les variables et définissent des combinaisons de valeurs compatibles.

Définition 2.3.1 *[Mon74] Un Problème de Satisfaction de Contraintes $\mathcal{P} = (X, D, C, R)$ est défini par :*
- *un ensemble de variables $X = \{x_1, \ldots, x_n\}$,*
- *un ensemble de domaines finis et discrets $D = \{D_{x_1}, \ldots, D_{x_n}\}$ où D_{x_i} est le domaine associé à la variable x_i,*
- *un ensemble de contraintes $C = \{C_1, \ldots, C_m\}$ où C_i est définie sur un ensemble de variables $\{x_{i_1}, \ldots, x_{i_{n_i}}\}$,*
- *un ensemble de relations $R = \{R_1, \ldots, R_m\}$ où R_i est l'ensemble des combinaisons de valeurs qui satisfont la contrainte C_i, il s'agit de la relation associée à C_i ; R_i est un sous-ensemble du produit cartésien $D_{x_{i_1}} \times \ldots \times D_{x_{i_{n_i}}}$.*

Cette définition donne la possibilité d'avoir une représentation des relations en extension ou en intention. Une représentation en extension peut occasionner des difficultés au travers de

$R_{18} : x_1 > x_8$		$R_{25} : x_2 \leq x_5$		$R_{27} : x_2 > x_7$		$R_{34} : x_3 \neq x_4$		$R_{36} : x_3 \neq x_6$	
x_1	x_8	x_2	x_5	x_2	x_7	x_3	x_4	x_3	x_6
2	1	1	1	2	1	1	2	1	2
3	1	1	2	3	1	1	3	1	3
3	2	1	3	3	2	2	1	2	1
		2	2			2	3	2	3
		2	3			3	1	3	1
		3	3			3	2	3	2

Tab. 2.1 – Relations associées aux contraintes de $\mathcal{P}$ données en extension (1/3).

$R_{38} : x_3 + x_8 = 3$		$R_{46} : x_4 \neq x_6$		$R_{56} : x_5 \geq x_6$		$R_{58} : x_5 \geq x_8$	
x_3	x_8	x_4	x_6	x_5	x_6	x_5	x_7
1	2	1	2	1	1	1	1
2	1	1	3	2	1	2	1
		2	1	2	2	2	2
		2	3	3	1	3	1
		3	1	3	2	3	2
		3	2	3	3	3	3

Tab. 2.2 – Relations associées aux contraintes de $\mathcal{P}$ données en extension (2/3).

l'espace mémoire nécessaire à cause du nombre de tuples qui peut, dans certains cas, être très important. S'il existe ainsi une formule qui traduit la même relation en intention, son utilisation offre un gain significatif en terme d'espace mémoire. En outre les tests de consistance durant la résolution vont se traduire par la vérification de cette formule en lieu et place d'une recherche de tuples dans une table. Les difficultés de ces deux opérations ne sont pas toujours comparables. Avec une implémentation appropriée, les tests de consistances par la recherche de tuples dans une table peuvent être faits en temps constant. Les complexités des méthodes de résolution que nous allons présenter sont basées sur cette hypothèse. Néanmoins, nous allons nous autoriser, mais c'est déjà le cas pour une bonne partie de la communauté contraintes, à définir les relations en intention. Dans ce cas, le coût du test de consistance doit englober celui de la résolution de la formule associée à la contrainte. Celui-ci n'étant pas forcément constant, il faut rajouter ce facteur dans les bornes de complexité.

Dans l'exemple suivant, nous avons une instance de CSP binaire. Nous utiliserons souvent, dans le cas de CSP binaire, une notation légèrement différent des contraintes et relations : C_{ij} est une contrainte qui porte sur les variables X_i et x_j, et R_{ij} est la relation associée à cette contrainte.

Exemple 2.3.1 *Considérons le CSP $\mathcal{P}$ défini comme suit :*
- $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$,
- $D = \{D_{x_1}, D_{x_2}, D_{x_3}, D_{x_4}, D_{x_5}, D_{x_6}, D_{x_7}, D_{x_8}\}$ *avec* $D_{x_i} = \{1, 2, 3\}, \forall 1 \leq i \leq 8$,
- $C = \{C_{18}, C_{25}, C_{27}, C_{34}, C_{36}, C_{38}, C_{46}, C_{56}, C_{58}, C_{68}, C_{78}\}$ *avec* $C_{ij} = \{x_i, x_j\}, \forall C_{ij} \in C$,
- $R = \{R_{18}, R_{25}, R_{27}, R_{34}, R_{36}, R_{38}, R_{46}, R_{56}, R_{58}, R_{68}, R_{78}\}$.

Les contraintes sont des relations algébriques entre les variables. Les équations algébriques définissent en intention les relations et les tables de compatibilités les définissent en extension dans les tables 2.1, 2.2 et 2.3.

Définition 2.3.2 *Une affectation d'une variable x est l'assignation d'une valeur v de son domaine D_x à x. Elle est notée $x \leftarrow v$.*
Soit $Y \subset X$. Une affectation (ou instanciation) sur Y est l'affectation de chacune de ses variables. Elle est notée $(y_1 \leftarrow v_{y_1}, \ldots, y_{|Y|} \leftarrow v_{y_{|Y|}})$, avec $\forall 1 \leq i \leq |Y|, y_i \in Y$ et $v_{y_1} \in D_{y_i}$.

$R_{68} : x_6 + x_8 = 3$		$R_{78} : x_7 + x_8 = 4$	
x_5	x_8	x_6	x_8
1	2	1	3
2	1	2	2
		3	1

Tab. 2.3 – Relations associées aux contraintes de $\mathcal{P}$ données en extension (3/3).

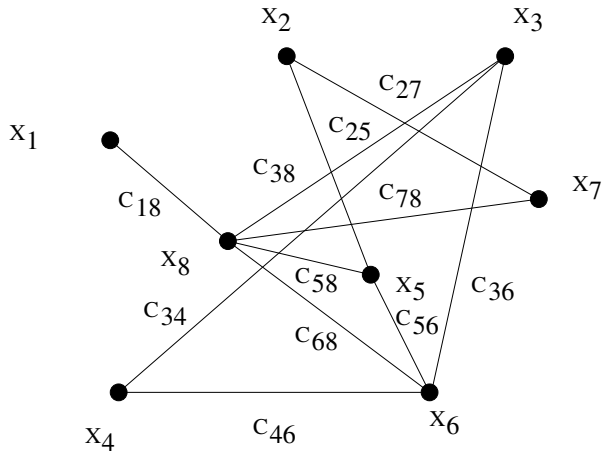

Fig. 2.11 – Graphe de contraintes de $\mathcal{P}$.

Une affectation est très souvent notée simplement $(v_{y_1}, \ldots, v_{y_{|Y|}})$. Dans la suite, nous préférerons utiliser une notation ensembliste $\{y_1 \leftarrow v_{y_1}, \ldots, y_{|Y|} \leftarrow v_{y_{|Y|}}\}$ pour une affectation par soucis de simplification. Cela permet de noter très simplement l'extension d'une affectation $\mathcal{A}$ à une nouvelle variable x avec une valeur v de son domaine par $\mathcal{A} \cup \{x \leftarrow v\}$.

Soit $\mathcal{A}$ une affectation d'un ensemble de variables. On note $X_{\mathcal{A}}$ l'ensemble des variables affectées dans $\mathcal{A}$.

Définition 2.3.3 *Soit $\mathcal{A}$ une affectation et $Y \subset X_{\mathcal{A}}$. La projection de $\mathcal{A}$ sur Y (notée $\mathcal{A}[Y]$) est la restriction de $\mathcal{A}$ aux variables de Y.*

Une contrainte C_i est satisfaite par une affectation $\mathcal{A}$ si $C_i \subset X_{\mathcal{A}}$ et $\mathcal{A}[C_i] \in R_i$. Si $C_i \subset X_{\mathcal{A}}$ et $\mathcal{A}[C_i] \notin R_i$, alors on dit que C_i n'est pas satisfaite ou C_i est violée. Une affectation sur Y est consistante si toutes les contraintes contenues dans Y sont satisfaites.

Définition 2.3.4 *Soient $\mathcal{P} = (X, D, C, R)$ et $Y \subset X$, une instanciation $\mathcal{A}$ sur Y est consistante si :*
$\forall C_i \in C, C_i \subset Y,\ \mathcal{A}[C_i] \in R_i$

Définition 2.3.5 *Une solution de $\mathcal{P} = (X, D, C, R)$ est une instanciation consistante de toutes les variables de X. L'ensemble des solutions de $\mathcal{P}$ est noté $Sol_{\mathcal{P}}$.*

La jointure naturelle, $\bowtie$, de l'ensemble des tuples autorisés par les contraintes dans C donne l'ensemble des solutions du CSP : $Sol_{\mathcal{P}} = \bowtie_{1 \leq i \leq m} R_i$.

Définition 2.3.6 *Un CSP $\mathcal{P} = (X, D, C, R)$ est consistant si $Sol_{\mathcal{P}} \neq \emptyset$.*

Un CSP est consistant s'il admet au moins une solution et inconsistant sinon.

Définition 2.3.7 *Deux CSP* $\mathcal{P} = (X, D, C, R)$ *et* $\mathcal{P}' = (X, D', C', R')$ *sont équivalents ssi* $Sol_{\mathcal{P}} = Sol_{\mathcal{P}'}$.

Cette notion d'équivalence entre CSP permet de simplifier un CSP (en diminuant la taille des domaines par exemple) tout en gardant la satisfiabilité du problème.

Définition 2.3.8 *Un CSP* $\mathcal{P} = (X, D, C, R)$ *est globalement consistant ssi* $\forall C_i \in C, Sol_{\mathcal{P}}[C_i] = R_i$.

La consistance globale traduit la participation de tout tuple autorisé par une contrainte dans au moins à une solution.
Pour un CSP donné, il est possible de s'intéresser à plusieurs problèmes :

- Existence d'une solution,
- Recherche d'une solution,
- Calcul du nombre de solutions,
- Recherche de l'ensemble des solutions,
- Recherche d'une valeur figurant dans l'ensemble des solutions
- ...

Le problème qui nous intéresse dans le cadre de ce travail est le problème de l'existence d'une solution : consistance du CSP. Il a été montré que ce problème est NP-complet [GJ79].

Il est possible de représenter la structure du problème par un graphe : le graphe des contraintes. Toutefois, ce terme n'est pas approprié pour le cas général. En effet, il a été introduit pour faire référence à la représentation graphique des CSP binaires qui sont des CSP dont les contraintes portent sur au plus deux variables. Dans ce cas, le graphe de contraintes est un graphe dont les sommets sont les variables du CSP et il existe une arête entre deux sommets s'il existe une contrainte liant les variables correspondantes. Concernant les CSP dont l'arité des contraintes n'est pas limitée à deux (appelés CSP n-aires), l'objet approprié est un hypergraphe avec, comme dans le cas des graphes de contraintes, les sommets qui sont les variables et les hyperarêtes les contraintes. Il faut noter que la majeure partie des études sur les CSP est limitée aux CSP binaires dans la mesure où il existe des transformations polynomiales des CSP n-aires en problèmes binaires moyennant une augmentation de la taille du problème et la perte de la structure du CSP original.

2.3.2 Les méthodes de résolution

Il existe différentes approches et méthodes pour résoudre un CSP. La méthode de base est le Backtrack standard. Schématiquement, elle procède à l'énumération de toutes les affectations possibles jusqu'à trouver une solution.

2.3.2.1 Backtrack classique (BT) (Algorithme 1)

C'est une méthode du type essais/erreurs. Elle consiste à instancier les variables successivement suivant un ordre prédéfini statique. A chaque nouvelle instanciation d'une variable, des tests de compatibilité par rapport aux variables affectées sont effectués afin de vérifier la consistance de l'instanciation courante. Pour cela, il faut vérifier que toutes les contraintes dont toutes les variables sont instanciées et qui contiennent la variable courante sont satisfaites. Si, la nouvelle affectation satisfait toutes les contraintes alors l'instanciation courante est consistante. Si, elle contient toutes les variables, le problème est donc consistant puisqu'elle constitue une solution. Dans le cas contraire, on passe à la prochaine variable suivant l'ordre. Si, par contre, la nouvelle affectation viole une contrainte, on choisit une nouvelle valeur dans le domaine de la variable courante jusqu'à trouver une qui satisfait toutes les contraintes. Si, toutes les valeurs ont été essayées sans succès, un retour en arrière (backtrack) est réalisé sur la dernière variable affectée avant la variable courante afin d'essayer une nouvelle valeur. Dans le pire des cas, cette

Algorithme 1 : BT($\mathcal{A}, V$)

```
1  if V = ∅ then
2  |  return true
3  else
4  |  Choisir x ∈ V
5  |  d_x ← D_x
6  |  Consistance ← false
7  |  while d_x ≠ ∅ and ¬Consistance do
8  |  |  Choisir v ∈ d_x
9  |  |  d_x ← d_x\{v}
10 |  |  if A ∪ {x ← v} ne viole aucune contrainte then
   |  |  Consistance ←BT(A ∪ {x ← v}, V\{x})
11 |  return Consistance
```

méthode peut explorer la totalité de l'espace de recherche dont la taille est bornée par d^n avec d la taille maximale des domaines du problème. Elle peut donc être très coûteuse avec une complexité temporelle en $O(md^n)$ (notée $O(exp(n))$. Elle se révèle totalement inefficace en pratique à cause du grand nombre de redondances inhérentes à cette technique. C'est une démarche qui explore de manière aveugle l'espace de recherche en ne tenant pas compte des particularités du problème et son évolution durant la résolution, à cause notamment d'un ordre d'affectation des variables prédéfini et arbitraire. Plusieurs voies ont été proposées pour l'améliorer que ce soit en pratique ou en théorie : les améliorations effectives de la technique du backtracking et la décomposition des problèmes.

Au niveau des améliorations de BT, on peut citer le calcul d'ordres d'affectations de bonne qualité tenant compte de la structure et des propriétés du problème. Il y a également la simplification du problème (réduction de la taille) avant et durant la résolution. Le retour en arrière non chronologique (intelligent) permet, de même, d'améliorer la méthode en général.

Dans tous les cas, la réduction de l'espace de recherche se traduit plus particulièrement par une réduction l'arbre de recherche associé à la méthode de résolution. Un arbre de recherche représente les affectations générées par la méthode et l'ordre sur les variables dans lequel elles sont construites. C'est un arbre orienté dont les nœuds sont les variables du problème et les arcs les différentes valeurs des domaines. Pour un nœud donné associé à la variable x, chaque arc partant de x correspond à l'affectation de la valeur v associée à cet arc à la variable x. Ainsi, une branche de l'arbre correspond à une affectation des variables associées à ses nœuds.

2.3.2.2 Le filtrage

Le filtrage permet de réduire la taille du problème ou de le modifier de sorte à construire un problème équivalent mais plus facile à résoudre. Il existe différents degrés de filtrage, plus ou moins efficaces et plus ou moins coûteux, qui se basent sur les notions de consistance locale ou générale et de propagation de contraintes. La première stratégie consiste à supprimer des domaines, les valeurs inconsistantes. Une valeur est inconsistante si elle ne participe à aucune solution du problème et de ce fait sa suppression conduit à un CSP équivalent de plus petite taille. Il est également possible de supprimer des tuples autorisés au niveau des relations dont il est sûr qu'ils ne mènent pas à une solution. Cette technique s'inscrit plus largement dans l'explicitation de contraintes cachées qui facilite la résolution. Pour réduire le coût de ces traitements, la solution utilisée est leur restriction à des sous-parties du CSP, d'où le terme de consistance locale. La propagation de contraintes consiste donc à supprimer des valeurs ou des tuples inconsistants, de proche en proche jusqu'à obtenir un problème localement consistant. Ce dernier est un point fixe de cette opération de propagation de contraintes. Il est appelé fermeture consistante locale

du problème.

La consistance d'arc ([Wal75]). La consistance d'arc est un exemple de consistance locale : chaque valeur v du domaine D_{x_i} d'une variable x_i doit être compatible avec toute contrainte portant sur x_i. Cela veut dire que v a au moins un support (une valeur compatible) sur chaque domaine de chaque variable partageant une contrainte avec x_i. La détection puis l'élimination des valeurs qui ne vérifient pas cette consistance est une opération de filtrage qui réduit la taille des domaines et donc celle du problème.

Définition 2.3.9 *Soit $\mathcal{P} = (X, D, C, R)$ un CSP, un domaine D_{x_i} vérifie la consistance d'arc ssi $D_{x_i} \neq \emptyset$ et $\forall v \in D_{x_i}, \forall C_j \in C, x_i \in C_j, v \in R_j[x_i]$. Un CSP vérifie la consistance d'arc ssi chacun de ses domaines vérifie la consistance d'arc.*

Plusieurs algorithmes de consistance d'arc sur les CSP binaires ont été proposés tels que AC-3 [Mac77], AC-4 [MH86], AC-6 [Bes94], AC-8 [CJ98], AC-2001 [BR01] et AC-3.1 [ZY01]. Les deux derniers algorithmes sont équivalents [BRYZ05]. Il est important de noter que AC-6, AC-2001 et AC-3.1 ont une complexité en temps optimale de $O(md^2)$ avec une complexité spatiale de $O(md)$. La consistance d'arc est utilisée à la fois en prétraitement, mais également durant la recherche pour maintenir une certaine cohérence locale entre l'affectation courante et la partie du problème non affectée.
Un CSP arc-consistant n'est pas forcément consistant. Des consistances plus fortes ont été définies pour supprimer plus d'inconsistances que la consistance d'arc.

La consistance de chemin La définition suivante est restreinte aux CSP binaires.

Définition 2.3.10 *Soit $\mathcal{P} = (X, D, C, R)$ un CSP, une paire de variable (x_i, x_j) est chemin consistant si $\forall (v_i, v_j) \in R_{ij}, \exists v_k \in D_{x_k}, (v_i, v_k) \in R_{ik}$ et $(v_j, v_k) \in R_{jk}$. Un CSP vérifie la consistance de chemin si toute paire de variables (x_i, x_j) est chemin consistante.*

La définition suppose l'existence d'une contrainte entre chaque paire de variables. Dans le cas où le CSP ne vérifie pas cette condition, il est possible de rajouter des contraintes universelles (qui autorisent toutes les combinaisons de valeurs) entre les variables non contraintes. Cela donne un nouveau CSP équivalent. Cette consistance permet de supprimer des couples de valeurs dans les relations voire même de rajouter des contraintes. En effet, si des couples de valeurs sont retirés des contraintes universelles rajoutées, cela traduit l'exhibition de cette nouvelle contrainte dont l'existence est induite par les contraintes du CSP de départ. Dans ce cas, il se produit donc une modification de la structure du problème même si ce dernier va demeurer binaire. A l'image de la consistance d'arc, plusieurs algorithmes de consistance de chemin ont été définis : PC-2 [Mac77], PC-4 [HL88], PC-5 [Sin95, Sin96], PC-6 [Chm96], PC-8 [CJ98] et PC-2001 [ZY01]. Il a été montré par ailleurs que PC-5 et PC-6 sont équivalents. La complexité optimale en temps de $O(n^3d^3)$ est obtenue par PC-4 dont la complexité en espace est identique, PC-5, PC-6 et PC-2001 avec une complexité en espace de $O(n^3d^2)$. Les algorithmes de consistance de chemin ont un coût non négligeable qui explique son utilisation uniquement en prétraitement et pas durant la recherche. La consistance de chemin étant toujours une consistance locale, Freuder [Fre78] en donne une généralisation.

La k-consistance ([Fre78]).

Définition 2.3.11 *Un CSP est k-consistant si toute affectation consistante sur $k-1$ variables peut être étendue en une instanciation consistante sur k variables. Un CSP est fortement k-consistant s'il est i-consistant pour tout $i \leq k$.*

La 2-consistance correspond à la consistance d'arc alors que la 3-consistance correspond à la chemin consistance. Dans le cas de cette dernière, cette définition de Freuder donne une caractérisation sur les CSP n-aires. Pour $k > 3$ la k-consistance peut induire des contraintes d'arité $k-1$ supérieure à 2 et transformer ainsi un CSP binaire en CSP n-aire. En outre, le coût de ces consistances est très élevé : $O(n^k d^k)$ [Coo89]. Plus k est élevé, plus on se rapproche de la consistance globale. Néanmoins, le coût de calcul devient prohibitif. Ce qui semble tout à fait normal, vu la nature du problème (NP-complet). Un compromis est donc nécessaire entre la résolution et le filtrage. De ce fait, les consistances utilisées ne dépassent pas, en général, la 3-consistance.

2.3.2.3 Le retour en arrière non chronologique

BT effectue un retour chronologique sur la dernière variable affectée dans le cas où toutes les valeurs de la variable courante x ont été essayées sans succès pour étendre de manière consistante l'affectation courante. Cependant, il est possible que cette variable ne soit pas en cause dans cet échec. Par exemple, si cette variable n'est pas liée à x par une contrainte, elle ne peut être en cause dans l'échec. Alors, une nouvelle valeur pour cette variable va mener au même échec. Pour éviter ces échecs répétés, il est possible de revenir sur une variable en cause plus en amont dans l'ordre d'affectation. Ce retour en arrière non chronologique est appelé Backjump. Plusieurs méthodes ont été définies dans cette optique. Elles diffèrent dans leur façon d'analyser les échecs pour permettre de revenir aux véritables causes. [DF02] peut être consulté pour avoir plus d'informations sur les algorithmes de retour en arrière non chronologiques.

Backjumping (BJ [Gas79]). Si toutes les valeurs de la variable courante x sont inconsistantes avec l'affectation courante, alors BJ revient sur la variable y la plus récemment affectée dont la valeur est en conflit avec une des valeurs de x. Cependant, dans le cas où cette valeur de la variable y était la dernière à être essayée, BJ se contente de revenir à la variable qui la précède dans l'ordre d'affectation. Ainsi, les sauts en arrière sont limités aux affectations consistantes qui ne peuvent être prolongées de manière consistante sur la variable suivante dans l'ordre (elles sont généralement appelées les feuilles de l'arbre de recherche).

Conflict-directed Backjumping (CBJ [Pro93]). L'algorithme maintient un ensemble de conflits pour chaque variable instanciée. Cet ensemble contient toutes les variables instanciées avant la variable courante x qui sont en conflit pour au moins une valeur de x déjà testée ou qui sont en cause dans l'échec d'une extension d'une affectation contenant x. En cas d'échec, CBJ revient sur la dernière variable affectée de l'ensemble des conflits de la variable courante. Contrairement à BJ, les sauts en arrière dans CBJ ne sont pas limités aux feuilles de l'arbre de recherche.

Graph-based Backjumping (GBJ [Dec90]). L'algorithme est basé sur le graphe des contraintes. En cas d'échec, GBJ revient sur la variable y la dernière variable instanciée dans le voisinage de x. Si toutes les valeurs de y ont été déjà essayées, il revient à la variable la plus profonde appartenant au voisinage de y ou au voisinage de toute variable instanciée après y, qui soit une cause potentielle de l'échec. Cet algorithme calcule, à travers les voisinages des variables, une approximation de la région du problème qui contient les variables en cause dans l'échec. Il est donc moins précis que CBJ qui calcule effectivement les raisons de l'échec. Cependant, ses sauts en arrière n'étant pas limités aux feuilles de l'arbre de recherche, il est en général plus performant que BJ en termes de réduction des redondances.

Algorithme 2 : FC($\mathcal{A}, V$)

```
1  if V = ∅ then
2  |  return true
3  else
4  |  Choisir x ∈ V
5  |  d_x ← D_x
6  |  Consistance ← false
7  |  while d_x ≠ ∅ and ¬Consistance do
8  |  |  Choisir v ∈ d_x
9  |  |  d_x ← d_x\{v}
10 |  |  if Forward-check(A ∪ {x ← v}, x) then Consistance ←FC(A ∪ {x ← v}, V\{x})
11 |  |  Unforward(x)
12 |  return Consistance
```

2.3.2.4 La consistance avant : la technique du look-ahead

Dans BT, le test de consistance est effectué en fonction des variables déjà instanciées (look-back). La technique du look-ahead, quant à elle, procède, dès l'affectation de la variable x, à la suppression de toutes les valeurs des variables non instanciées, incompatibles avec cette valeur de x. Le niveau de filtrage utilisé détermine les valeurs supprimées. Si un domaine devient vide, cela veut dire que l'affectation courante n'a pas d'extension consistante. Une nouvelle valeur de x est essayée. Si elles ont toutes été essayées, l'algorithme effectue un backtrack. Si aucun domaine n'est vide, la recherche continue sur la prochaine variable. Cette technique ne construit que des affectations consistantes, les autres étant supprimées par le filtrage.

Forward Checking (FC [HE80]) (Algorithme 2). L'algorithme Forward Checking utilise cette technique. La version présentée ici se limite aux CSP binaires. Mais des versions adaptées aux CSP n-aires existent [BMFL02]. Le filtrage de FC supprime du domaine de toute variable non instanciée y dans le voisinage de x, toutes les valeurs qui violent la contrainte entre x et y pour la valeur courante de x.

Forward-Check réalise le filtrage après l'affectation de x et Unforward annule le filtrage à chaque retour en arrière. FC a une complexité identique à celle de BT. Cependant, il obtient généralement des résultats bien meilleurs tant au niveau du nombre de nœuds de l'arbre de recherche associé qu'au niveau du nombre de tests de consistance sur le contraintes ou tout simplement du temps.

Maintaining Arc Consistency (MAC [SF94]) FC limite le filtrage par consistance d'arc aux contraintes qui lie la variable courante x à celles non affectées qui se trouvent dans son voisinage alors que MAC va plus loin. En effet il applique le même filtrage à l'ensemble des contraintes qui ne sont pas totalement instanciées. Avec ce niveau de filtrage plus élevé, MAC développe généralement moins de nœuds que BT et FC. Néanmoins, le maintien de la consistance durant la recherche est d'un certain coût qui peut entraîner au final un temps de calcul plus long.

FC et MAC sont connus comme étant les meilleures méthodes de résolution énumératives. Malgré tout, il demeure un grand nombre de redondances liées à l'approche. De ce fait, leur complexité temporelle théorique est toujours en $O(md^n)$. Elles sont incapables de résoudre des problèmes de taille considérable en un temps raisonnable.

2.3.2.5 Algorithmes avec mémorisation

Durant la recherche, il existe certaines parties du problème qui sont résolues à plusieurs reprises. Ces redondances peuvent être évitées en enregistrant le travail effectué pour une réutilisation ultérieure. Généralement, les informations mémorisées sont des affectations sur des sous-ensembles de variables. Dans le cas où une affectation de ce type ne peut être étendue en une solution, le terme de nogood est assez souvent utilisé. Si par contre, elle peut être étendue de manière consistante sur une partie du problème, on parle de good. Ces deux notions permettent d'éviter un grand de redondances et ainsi d'avoir parfois de meilleures bornes de complexités. Plusieurs méthodes ont été proposées telles que : Value-based shallow learning, Graph-based shallow learning et Deep learning [Dec90], Jump-back learning [FD94], Nogood Recording [SV93, SV94], Dynamic Backtracking [Gin93], Learning Tree-Solve [BM96], AndOr Search Graph [DM07], Recursive Conditioning [Dar01], Tree-Clustering [DP89], BCC [BT01a, Fre85], BTD [JT03], etc.
Nous reviendrons plus en détails sur certains de ces algorithmes par la suite. La grande faiblesse de ces techniques se situent au niveau de leur complexité en espace. Le nombre de (no)goods étant potentiellement exponentiel, cette complexité est exponentielle en la taille de ces (no)goods. Elles peuvent donc requérir un espace mémoire considérable empêchant la résolution d'aller à son terme. Les solutions apportées consistent généralement à limiter la taille des informations enregistrées ou à en oublier (l'information est effacée) un certain nombre durant la recherche pour réduire l'espace mémoire requis.

2.3.2.6 Les méthodes structurelles

Ces techniques tirent profit de la structure du problème pour améliorer la résolution. Cette structure est capturée à travers une repésentation du problème en (hyper)graphe (appelé (hyper)graphe de contraintes). Les sommets de l'hypergraphe de contraintes sont les variables du problème et les arêtes représentent les contraintes.

La plupart des méthodes structurelles utilise les décompositions de graphes et d'hypergraphes présentées dans la première section de ce chapitre.

Tree Clustering (TC [DP89]). TC est basé sur une décomposition arborescente du graphe de contraintes du problème. Les différents clusters sont résolus indépendamment et leurs solutions mémorisées. Ensuite, le CSP acyclique obtenu est résolu en un temps polynomial. La difficulté revient de ce fait à la résolution des clusters. La complexité en temps est de $O(exp(w+1))$, $w+1$ étant la taille du plus grand cluster de la décomposition. Le problème majeur est la complexité en espace qui est aussi en $O(exp(w+1))$. A cause de cette masse trop importante d'informations stockées, TC est inopérant en pratique.
Une amélioration de cette borne a été proposé qui restreint les informations mémorisées aux intersections entre clusters (séparateurs). La complexité spatiale passe ainsi à $O(exp(s))$ avec s la taille maximale des séparateurs.

Malgré tout, un compromis espace/temps entre l'espace mémoire requis et la borne de complexité temporelle est incontournable. [DF01] propose une famille d'algorithmes basés sur la notion de décomposition arborescente et qui permettent de réaliser ce compromis. L'idée principale consiste à réduire l'espace mémoire requis par une diminution de la taille des séparateurs en fusionnant des clusters qui ont des intersections de grande taille.

Backtracking on Tree-Decomposition (BTD [JT03]) (Algorithme 3). Cette méthode sera évoquée très souvent dans la suite, nous allons donc la présenter plus en détails. Elle est basée sur une décomposition arborescente du graphe de contraintes du CSP. Cette dernière induit un ordre d'affectation partiel sur les variables qui permet de tirer profit de la structure du problème pour éviter beaucoup de redondances grâce aux notions de goods et nogoods structurels. Soit $(E, \mathcal{T})$ une décomposition arborescente du graphe de contraintes du CSP.

Algorithme 3 : BTD($\mathcal{A}, E_i, V_{E_i}$)

```
1  if V_{E_i} = ∅ then
2      Consistance ← true
3      F ← Fils(E_i)
4      while F ≠ ∅ and Consistance do
5          Choisir E_j ∈ F
6          F ← F\{E_j}
7          if A[E_j ∩ E_i] est un good then Consistance ← true
8          else
9              if A[E_j ∩ E_i] est un nogood then Consistance ← false
10             else
11                 Consistance ←BTD(A, E_j, E_j\(E_j ∩ E_i))
12                 if Consistance then Enregistrer le good A[E_j ∩ E_i]
13                 else Enregistrer le nogood A[E_j ∩ E_i]
14     return Consistance
15 else
16     Choisir x ∈ V_{E_i}
17     d_x ← D_x
18     Consistance ← false
19     while d_x ≠ ∅ and ¬Consistance do
20         Choisir v ∈ d_x
21         d_x ← d_x\{v}
22         if A ∪ {x ← v} ne viole aucune contrainte then
           Consistance ←BTD(A ∪ {x ← v}, E_i, V_{E_i}\{x})
23     return Consistance
```

Définition 2.3.12 *Une numérotation sur E compatible avec une numérotation préfixe de $\mathcal{T}$ dont E_1 est la racine est appelée numérotation compatible.*

Cette numérotation sur E induit un ordre partiel sur les variables.

Définition 2.3.13 *Un ordre $\preceq$ sur X tel que $\forall x \in E_i, \forall y \in E_j$, avec $i < j$ et $x \preceq y$ est un ordre d'énumération compatible.*

La descendance de E_j, notée $Desc(E_j)$, est l'ensemble des variables contenues dans le sous-arbre enraciné en E_j. Par construction $E_i \cap E_j$, avec E_j un fils de E_i, est un séparateur du graphe. Ainsi, les seules contraintes liant la descendance de E_j au reste du problème sont au niveau de cette intersection : les compatibilités entre instanciations passent uniquement au niveau de cet ensemble. BTD utilise un ordre d'énumération compatible comme ordre d'affectation des variables. Quand un cluster E_i est totalement instancié de façon consistante, l'affectation se poursuit sur un de ses fils E_j et sa descendance $Desc(E_j)$. Dès lors, deux cas peuvent se présenter. Dans le premier cas, l'affectation courante n'admet pas d'extension consistante sur $Desc(E_j)$. Les raisons de cette inconsistance se situent au niveau de cet ensemble puisque $E_i \cap E_j$ a été affectée de manière consistante auparavant et qu'il constitue un séparateur entre $Desc(E_j) - (E_i \cap E_j)$ et le reste du problème. Les contraintes violées se trouvent ainsi dans $Desc(E_j)$. On peut en déduire qu'une nouvelle affectation égale à l'affectation courante sur $E_i \cap E_j$ va mener au même résultat : un échec. Fort de ce constat, cette affectation sur $E_i \cap E_j$ peut être enregistrée comme un nogood pour éviter ces redondances. Ainsi, toute affectation future égale au nogood sur cette intersection sera immédiatement stoppée dans la mesure où elle ne peut pas mener à une solution.

Dans le second cas, par contre, l'affectation courante admet une extension consistante sur $Desc(E_j)$ alors l'affectation sur $E_i \cap E_j$ est enregistrée comme un good. En effet, par un raisonnement similaire, on pourra conclure que toute nouvelle affectation égale à l'affectation courante sur $E_i \cap E_j$ pourra être étendue de manière consistante sur la descendance de E_j. Si dans le futur, une affectation est égale au good sur $E_i \cap E_j$, un forward jump (saut en avant dans l'ordre d'affectation) pourra être accompli puisqu'on a l'assurance qu'elle pourra être étendue de manière consistante sur cette partie. La recherche sera ainsi poursuivie sur le reste du problème.

Définition 2.3.14 *Soient E_i un cluster et E_j l'un de ses fils. Un good (resp. nogood) structurel de E_i par rapport à E_j est une affectation consistante sur $E_i \cap E_j$ telle qu'il existe (resp. n'existe pas) d'extension consistante de cette affectation sur $Desc(E_j)$.*

L'algorithme BTD a 3 paramètres : $\mathcal{A}$ l'affectation courante, E_i le cluster courant et V_{E_i} les variables du cluster E_i qui ne sont pas encore instanciées. Au cours de la recherche, il choisit une variable non instanciée du cluster courant E_i (ligne 16), s'il en reste, et lui affecte une valeur v (ligne 20). Si l'extension $\mathcal{A} \cup \{x \leftarrow v\}$ est inconsistante, BTD choisit une nouvelle valeur pour x (boucle while : lignes 19-22). Sinon, un appel récursif (ligne 22) : BTD($\mathcal{A} \cup \{x \leftarrow v\}, E_i, V_{E_i} \backslash \{x\}$) va essayer d'étendre $\mathcal{A} \cup \{x \leftarrow v\}$ sur $V_{E_i} \backslash \{x\}$. En cas d'échec, une nouvelle valeur de x est essayée. Si toutes les valeurs ont été essayées, un backtrack est opéré sur la variable précédente. Si toutes les variables de E_i ont été instanciées de manière consistante, alors BTD choisit un fils E_j de E_i (ligne 5). Si $\mathcal{A}[E_i \cap E_j]$ est un good (ligne 7), on sait que $\mathcal{A}$ peut être étendue de manière consistante sur le sous-problème enraciné en E_j. Si $\mathcal{A}[E_i \cap E_j]$ est un nogood (ligne 9), $\mathcal{A}$ ne peut être étendue en une solution, alors un backtrack est opéré. Si $\mathcal{A}[E_i \cap E_j]$ n'est un ni un good, ni un nogood, BTD va tenter d'étendre $\mathcal{A}$ sur le problème enraciné en E_j (ligne 11). En cas de succès, BTD enregistre le good $\mathcal{A}[E_i \cap E_j]$ (ligne 12), sinon ce dernier est enregistré comme un nogood (ligne 13). Si $\mathcal{A}$ a été étendue sur toute la descendance de E_i alors True est retourné et False dans le cas contraire (boucle while : lignes 4-13). L'appel initial BTD($\emptyset, E_1, V_{E_1}$) retourne donc True si $\mathcal{P}$ a une solution et False, sinon. Grâce à l'exploitation de la structure du problème au travers de la décomposition arborescente et aux redondances évitées en exploitant les goods et nogoods, BTD atteint la borne de complexité théorique en temps de TC.

Théorème 2.3.1 *La complexité en temps de BTD est en $O(exp(w+1))$ tandis que sa complexité spatiale est en $O(n.s.exp(s))$, avec s la taille de la plus grande intersection entre clusters.*

Il faut noter que BTD ne retourne pas de solution si le problème est consistant. Il se borne à déterminer la consistance ou l'inconsistance du problème. Dans le cas où une solution est recherchée, un travail supplémentaire devra être effectué pour la calculer. A l'aide des (no)goods enregistrés, ce travail peut être effectué efficacement à l'image de la technique présentée dans [JT04b]. Les résultats de BTD sont parmi les meilleurs obtenus par une méthode structurelle. Elle combine l'efficacité de l'énumération et les bornes de complexités issues de l'exploitation de la structure du CSP. Elle parvient à résoudre des CSP structurés (avec une décomposition arborescente de largeur réduite par rapport au nombre de variables) qui sont inaccessibles pour FC et MAC. Dans le cas des problèmes ne possédant de structure intéressante, son comportement est similaire à celui de ces deux méthodes.

Cependant, la version standard utilise un ordre d'affectation des variables statique. Sachant l'inefficacité de ces heurisiques comparées aux heuristiques dynamiques, cela constitue un handicap important. D'ailleurs, les résultats de BTD ont été obtenus avec des ordres d'affectation dynamique au sein de chaque cluster de la décomposition. Cette liberté accrue n'est cependant pas suffisant en général pour profiter pleinement (ou du moins le plus possible) de la puissance des ordres dynamiques.

De même, la complexité spatiale exponentielle de BTD impose une restriction sur l'étendue des décompositions arborescentes exploitables. Celles dont la valeur de la taille de la plus grande

intersection entre les clusters est trop importante mènent à un échec de la résolution à cause d'un espace mémoire requis qui déborde celui qui est disponible.

Recursive Conditioning (RC [Dar01]). Darwiche a défini une méthode similaire qui améliore la compléxité de BT avec un espace linéaire. Le Recursive Conditioning utilise la notion de dtree qui un arbre binaire complet construit sur le graphe des contraintes du CSP. Cet arbre donne en fait une décomposition récursive du problème (qui au final est équivalente à une décomposition arborescente). En effet, le dtree est construit récursivement par le choix d'un cutset (séparateur) qui décompose le graphe en plusieurs composantes connexes. Ensuite le même travail est effectué sur les différentes composantes réunies avec le cutset. L'ordre d'instanciation des variables suit cette décomposition récursive. Chaque composante connexe est résolue de manière indépendante des autres puisque le cutset est instancié au préalable, définissant des sous-problèmes indépendants. La première version a une complexité en espace linéaire et une complexité en temps de $O(n.exp(w.log(n)))$ avec w la largeur de l'ordre d'élimination associé au dtree et donc la largeur du dtree qui est la taille maximale de ses clusters moins un (la largeur de la décomposition arborescente équivalente). Cependant, une seconde version avec mémorisation évite les résolutions multiples d'un même sous-problème, en enregistrant leur consistance (inconsistance). Sa complexité est en $O(n.exp(w+1))$ en espace et en temps. Le compromis espace-temps est proposé avec une estimation de l'espace utilisé si la mémoire utilisée pour enregistrer les informations sur les cutsets est limitée à un certain pourcentage de la mémoire totale. Les complexités en temps et en espace seront ainsi comprises entre les deux extrêmes. Dans [DH01], les auteurs montrent l'équivalence existant entre dtree et décomposition arborescente et de quelle manière, il est possible de calculer un dtree de qualité en utilisant une technique de partitionnement d'hypergraphe. Construire un dtree de qualité revient donc à construire une décomposition de qualité qui limite la largeur. Il est possible également comme le dit Darwiche d'utiliser les résultats sur les séparateurs pour construire des dtrees équilibrés qui reviennent à des décompositions équilibrées. Cette méthode a obtenu de bons résultats.

En outre, avec son ordre statique, la seconde version de RC (avec mémorisation) fonctionne de la même manière que BTD. On retrouve de ce fait, une des faiblesse de BTD à savoir la rigidité de l'ordre d'affectation des variables. Au niveau de l'espace mémoire requis, RC offre la possibilité de faire un compromis avec l'exploitation de la structure pour éviter les redondances. Certaines informations qui auraient dû être enregistrées au niveau de certains cutsets ne le seront plus si cela doit entraîner une demande d'espace mémoire trop importante. Le choix de ces cutsets n'est pas sans conséquence sur l'efficacité de la méthode. Il n'existe pas d'étude à ce niveau pour intégrer directement à la décomposition, une heuristique efficace de choix de cutsets sur lesquels RC peut enregistrer le maximum d'informations pertinentes pour une résolution efficace sans déborder de l'espace mémoire disponible.

Variable Elimination (VE [DP87a, Dec99]). La méthode Variable Elimination est une technique sans backtrack. Elle repose sur un ordre sur les variables dit ordre d'élimination. Les variables sont éliminées une à une suivant l'ordre inverse. Lors de l'élimination d'une variable, une contrainte est ajoutée au reste du problème qui caractérise son effet sur ce dernier et maintient l'équivalence du CSP. Cette contrainte est calculée par une jointure d'un ensemble de contraintes liant la variable à éliminer à d'autres du problème courant. Cet ensemble de contraintes est appelé bucket. La complexité en temps est en $O(n.exp(w+1))$ et celle en espace en $O(n.exp(w))$, avec w la largeur de l'ordre. Cette complexité en espace vient des contraintes ajoutées dont l'arité peut atteindre w. Une table représentant une contrainte d'arité w peut contenir jusqu'à d^w tuples. Pour réduire l'espace mémoire requis, [Lar00] propose une hybridation de VE avec BT. Il définit un facteur a qui va limiter l'arité des contraintes ajoutées. En cas de dépassement, une recherche énumérative de type backtracking est effectuée au niveau de la variable en lieu et place d'une élimination. Cette méthode, notée VES, limite la complexité en espace à $O(n.exp(a))$, celle en temps passe à $O(exp(a+z))$, avec z le nombre de variables dont

la largeur dépasse a. $O(exp(a))$ est le coût de l'élimination des variables de largeur inférieure à k dans l'ordre. Les résultats de VES améliorent ceux de VE et BT.

Le compromis rendu possible par VES souffre des mêmes maux que celui de RC. La séparation du choix entre la valeur de a et l'ordre d'élimination peut entraîner un accroissement significatif de la complexité temporelle à travers une valeur de z qui n'est pas maîtrisée. Il est possible qu'avec un autre ordre d'élimination, la valeur de z soit largement plus petite.

BCC ([BT01a, Fre85]). La méthode BCC repose sur un arbre BCC du graphe de contraintes. Les nœuds de l'arbre sont naturellement ordonnés tel que les fils soient placés après leur père. Cet ordre naturel peut être obtenu aussi bien par un parcours en largeur que par un parcours en profondeur de l'arbre. BCC instancie les variables selon un ordre statique *BCC-compatible* (compatible avec l'ordre naturel de l'arbre BCC considéré) : les variables de E_i sont instanciées avant celles de E_j si E_i et E_j sont des bicomposantes telles que $i < j$. Etant donné un ordre BCC-compatible, l'accesseur d'une bicomposante est sa plus petite variable dans l'ordre. Cette variable est un séparateur du graphe de contraintes. La méthode BCC évite certaines redondances. En fait, certaines valeurs des accesseurs sont marquées si on a déjà établi qu'elles pouvaient être étendues de façon consistante sur les prochaines variables dans l'ordre. Aussi, la prochaine fois que ces valeurs seront affectées à ces variables, un forward-jump sera effectué vers une partie non explorée de l'espace de recherche. Si une valeur d'un accesseur ne peut être étendue de façon consistante sur une partie des prochaines variables selon l'ordre considéré, cette valeur est supprimée. De plus, lorsqu'un échec se produit, BCC opère un retour en arrière non chronologique vers la dernière variable instanciée pouvant expliquer l'échec. Sa complexité en temps est en $O(exp(k))$ avec k la taille de la plus grande bicomposante.
La méthode BCC procède de la même manière que BTD sur une décomposition arborescente dont la taille des intersections est limitée à 1. Certes cela permet d'éviter les problèmes d'espace mémoire, mais la taille des clusters peut être considérable. BCC ne profite que trop peu de la structure du problème pour améliorer la résolution. En outre, l'ordre statique d'affectation des variables est une autre faiblesse importante.

AndOrSearchGraph ([DM07]). La méthode AND/OR Search Tree repose sur le calcul d'un espace de recherche AND/OR défini suivant un pseudo-arbre du graphe de contraintes. Les indépendances entre sous-problèmes ainsi produits permettent de réduire exponentiellement la taille de l'espace de recherche. Soit $\mathcal{T} = (X, C')$ un pseudo-arbre de $G = (X, C)$. L'arbre de recherche AND/OR associé $S_{\mathcal{T}}(\mathcal{P})$ admet des niveaux alternés de nœuds AND et OR. Les nœuds OR x_i correspondent aux variables alors que les AND $< x_i, v_i >$ (ou v_i) sont les valeurs affectées aux variables dans leur domaine respectif. La racine de l'arbre AND/OR est le nœud OR donné par la racine de $\mathcal{T}$. Un nœud OR x_i a pour fils un nœud AND $< x_i, v_i >$ ssi $< x_i, v_i >$ est consistante avec l'affectation partielle définie sur le chemin de la racine de l'arbre au nœud x_i. Un nœud AND $< x_i, v_i >$ a pour fils un nœud OR x_j ssi x_j est un fils de x_i dans le pseudo-arbre. Une solution de $\mathcal{P}$ est un sous-arbre de l'arbre de recherche AND/OR contenant la racine de ce dernier et qui vérifie : s'il contient un nœud OR alors il contient au moins un de ses fils, s'il contient un nœud AND alors il contient tous ses fils et toutes ses feuilles sont consistantes. La méthode de résolution AND/OR Search Tree consiste donc à calculer un pseudo-arbre du graphe de contrainte et construire l'arbre de recherche AND/OR associé. De ce fait, une recherche en profondeur d'abord pour trouver un sous-arbre solution suffit pour résoudre le problème. La complexité en temps du AndOrSearch Tree est en $O(exp(h))$ avec h la hauteur du pseudo-arbre. Sa complexité spatiale est linéaire.
La méthode AND/OR Search Tree peut être améliorée en enregistrant des informations qui permettent d'éviter certaines redondances et réduire ainsi la taille de l'espace de recherche. La notion de séparateurs-parents définie dans [DM07] pour un pseudo-arbre, donne un ensemble de séparateurs du graphe de contraintes. En effet, pour un nœud x_i de $\mathcal{T}$, l'ensemble des séparateurs-

parents de x_i est formé par x_i et ses ancêtres dans $\mathcal{T}$ qui sont voisins dans G de ses descendants dans $\mathcal{T}$. Il a été prouvé dans [DM07] que deux nœuds avec le même ensemble de séparateurs-parents sont racines de deux sous-problèmes identiques si les affectations sur les variables de l'ensemble séparateurs-parents sont identiques. On peut dès lors les fusionner et cette opération mène à un point fixe appelé graphe de contexte minimal de recherche AND/OR (AND/OR Search Graph). Cela donne une approche similaire à celle de BTD. La complexité en temps du AndOrSearch Graph est en $O(exp(w^*))$, w^* étant la largeur induite de l'arbre. La largeur induite par $\mathcal{T}$ est égale à la largeur de $G = (X, C')$. Pour un ordre donné sur les nœuds de l'arbre, la largeur d'un nœud est donnée par le nombre de ses voisins le précédant dans l'ordre. La largeur de l'ordre est la largeur maximum des nœuds alors que celle d'un graphe est la largeur minimum sur l'ensemble des ordres possibles. La complexité spatiale de la méthode est également en $O(n.s.exp(s))$, avec s la taille du plus grand ensemble séparateurs-parents.

La méthode du AND/OR Search Graph obtient parmi les meilleurs résultats pratiques fournis par des techniques structurelles. Les auteurs ont commencé une étude sur une définition d'un ordre sur les variables plus dynamique qui a permis d'améliorer les performances de la méthode. Le cadre AND/OR permet de faire un compromis entre le temps et l'espace. Néanmoins, ce cadre ne propose pas de stratégie pour faire ce compromis de manière efficace.

Pseudo-tree search (PTS [FQ85]). La méthode Pseudo-Tree Search (PTS) est également basée sur un pseudo-arbre du graphe de contraintes. Dès que des parties du problème deviennent indépendantes, alors elles sont résolues indépendamment. Les variables sont affectées selon un ordre induit par $\mathcal{T}$: la racine est le point de départ et les sous-problèmes enracinés aux fils de la variable courante sont résolus récursivement et séparément. La complexité en temps de PTS est en $O(exp(h))$ avec h la hauteur du pseudo-arbre et sa complexité spatiale est linéaire.

PTS, malgré la détection des indépendances entre sous-problèmes, produit un cetain nombre de redondances par des résolutions répétées d'un même sous-problème. Cela est dû à l'absence d'apprentissage et explique la complexité en temps exponentielle en h.

Tree-Solve ([BM96]). La méthode Tree-Solve repose sur la notion d'arrangement en arbre orienté, similaire à celle de pseudo-arbre. Elle procède de la même manière que PTS sur un arrangement en arbre orienté $\mathcal{T}$ du graphe de contraintes. Les variables sont affectées en partant de la racine de $\mathcal{T}$. Ensuite, les sous-problèmes enracinés aux fils sont résolus récursivement et indépendamment. La complexité en temps de Tree-Solve est en $O(exp(h))$, avec h la hauteur de $\mathcal{T}$ et sa complexité spatiale est linéaire.
La méthode Learning Tree-Solve est une amélioration du Tree-Solve, qui enregistre des goods et nogoods sur les séparateurs du problème donnés par l'arrangement en arbre. Ces derniers sont les ensembles de définition des nœuds de $\mathcal{T}$. Pour un nœud x_i de $\mathcal{T}$, l'ensemble de définition de x_i est constitué de ses ancêtres dans $\mathcal{T}$ qui sont voisins dans G de ses descendants dans $\mathcal{T}$. La complexité en temps du Learning Tree-Solve est en $O(exp(w^*))$, où w^* est la largeur induite de l'arrangement en arbre. Soit s la taille du plus grand séparateur. La complexité en espace est en $O(n.s.exp(s))$.

Tree-Solve, en l'absence d'apprentissage, propose une moins bonne complexité en temps que la méthode Learning Tree-Solve. Cependant, cette dernière a une complexité spatiale exponentielle qui peut requérir un espace mémoire considérable. Un compromis entre ces deux extrêmes est nécessaire pour améliorer les performances.

Cycle (hyper)cutset ([DP87b, GLS00]. Ces méthodes déterminent un sous-ensemble de variables S (appelé coupe-cycle) dont le retrait donne un (hyper)graphe de contraintes acyclique. Ainsi, une résolution énumérative de S induit un CSP acyclique. La complexité en temps de ces méthodes sont en $O(exp(|S| + 2))$ tandis que celle en espace est linéaire. L'efficacité théorique dépend donc de la taille de l'ensemble coupe-cycle. Cependant, la taille de l'ensemble coupe-cycle est assez importante en pratique, d'où des résultats de qualité moyenne.

Cyclic-clustering ([JT04a]). La méthode du Cyclic-clustering combine celle du Tree Clustering avec celle de Cycle (hyper)cutset. Elle essaie de tirer profit de leurs avantages respectifs. La première phase consiste à calculer un ensemble de cliques maximales de la 2-section de l'hypergraphe de contraintes du CSP. Ces cliques maximales définissent un hypergraphe qui recouvre les contraintes du problème. Cet hypergraphe induit une décomposition plus fine que celle qui est donnée par une décomposition arborescente du problème dans laquelle les clusters recouvrent les cliques maximales. Cependant, l'hypergraphe n'est pas forcément acyclique. De ce fait, la seconde phase va consister à appliquer la méthode du Cycle (hyper)cutset sur cet hypergraphe. On obtient donc un cutset de taille réduite par rapport à une application directe de la méthode sur le CSP. La complexité en temps de cette méthode est en $O(exp((w^- + 1) + |S^-|))$ et celle en espace en $O(n.s.d^s)$, avec $w^- + 1$ la taille maximum des cliques maximales, s la taille maximum des intersections entre les cliques maximales et $|S^-|$ la taille du coupe-cycle. Les auteurs montrent que cette borne de complexité en temps est meilleure que celle de la méthode Cycle (hyper)cutset et moins bonne que celle de TC en général. Le Cyclic-clustering est donc un compromis entre TC et Cycle (hyper)cutset dont les performances sont meilleures que celles de ces derniers.

Les méthodes basées sur des recouvrement encadrés acycliques ([CJG05]). Le cadre des recouvrements encadrés présenté à la section 2.2.3 permet de définir des méthodes d'une grande efficacité théorique, telles que celle donnée par la décomposition en hyperarbre [GLS00] et la Spread cut décomposition [CJG05]. Ces décompositions recouvrent à la fois les variables par des blocs et les contraintes par des cadres. Ensuite, la résolution consiste à calculer l'ensemble des solutions sur les différents blocs par une jointure des hyperarêtes des cadres associés, projetée sur les variables des blocs. Cela donne naissance à un CSP acyclique qui peut être résolu en un temps polynomial. Les complexités en temps et en espace de ces méthodes est en $O(r^k)$, avec k le nombre maximum d'hyperarêtes dans un cadre et r le nombre maximum de tuples dans une des contraintes correspondantes aux hyperarêtes. De ce fait, une décomposition de qualité a un nombre maximum d'hyperarêtes réduit dans ses cadres, et donc une complexité limitée. Les auteurs de [GLS00] ont proposé des outils de comparaison, en terme d'efficacité théorique de résolution des CSP, et mené une étude sur la puissance de ces différentes décompositions. Il en est ressorti que la décomposition en hyperarbre et la Spread cut décomposition sont les plus puissantes. En outre, il a été prouvé dans [CJG05] qu'il n'existe aucune autre classe de recouvrements encadrés acycliques plus puissante. Même si ces deux décompositions se trouvent au même niveau dans la hiérarchie de Gottlob, Leone et Scarcello, il a été montré dans [CJG05] que sur certains graphes les spread cut décompositions peuvent être de meilleure qualité au sens de la largeur des recouvrements (nombre d'hyperarêtes dans les cadres). Les bornes de complexités théoriques de ces méthodes sont parmi les meilleures voire même, souvent les meilleures obtenues. Cependant, on peut penser qu'elles sont d'un intérêt purement théorique. En effet, elles souffrent de plusieurs faiblesses très fortes pour une application pratique. Une première est la nécessité d'avoir des contraintes définies en extension pour calculer les jointures. Cela peut s'avérer impossible pour certaines contraintes avec un nombre de tuples trop grand pour être représentés en extension. Une seconde faiblesse est l'espace requis trop important. Comme dans le cas de TC, il faut calculer des jointures d'hyperarêtes qui donnent de nouvelles contraintes dont les tuples sont les solutions des blocs. Ces nouvelles contraintes peuvent avoir une grande arité et contenir ainsi un nombre important de tuples. De ce fait, aucun résultat pratique n'est disponible pour ces méthodes dans la littérature. En outre, on peut penser que le calcul d'une jointure sur un ensemble d'hyperarêtes n'est pas plus coûteux que celui de l'ensemble de ses solutions par une méthode énumérative telle que FC ou MAC. Il est possible donc que les bornes de complexité des méthodes structurelles comme BTD, RC, et autres, soient largement surestimées. Une étude plus précise à ce niveau devrait nous permettre de disposer d'un cadre uniforme pour comparer plus précisément ces différentes techniques en théorie.

2.3.2.7 Heuristiques

L'ordre d'affectation des variables a un impact considérable dans l'efficacité des méthodes de résolution. Il dépend essentiellement du problème à résoudre. Trouver un ordre permettant la résolution la plus efficace possible n'est pas aisé. Cependant, il existe un certain nombre d'heuristiques de choix de variables définissant des ordres de qualité. Elles sont essentiellement basées sur le principe du "first-fail" qui consiste à faire les choix les plus contraints d'abord pour essayer de rencontrer les échecs le plus tôt possible et réduire par la même occasion la taille de l'arbre de recherche.

Ces heuristiques peuvent être statiques. L'ordre d'affectation des variables est calculé avant le début de la résolution. Dans ce cas, elles sont basées sur les propriétés structurelles du problème. On peut citer entre autres :

- maxdeg [DM89] : elle ordonne les variables suivant un ordre décroissant sur les degrés des variables. Le degré d'une variable est le nombre de contraintes qui portent sur elle. Ainsi, les variables les plus contraintes sont affectées en premier dans le but de rencontrer les échecs le plus tôt possible.
- mc [DM89] : la première variable est choisie de manière arbitraire. Ensuite, la prochaine dans l'ordre est une variable qui a un nombre maximum de voisines déjà ordonnées. Puisqu'elle partage un certain nombre de contraintes avec les variables déjà affectées, son intanciation peut conduire assez rapidement vers un échec.
- mw [Fre82] : les variables sont ordonnées dans l'ordre inverse d'un ordre de largeur minimum. La largeur d'un ordre est le nombre maximum de variables voisines d'une variable donnée la précédant dans l'ordre. A l'image de l'heuristique précédente, on essaie d'instancier les variables ayant le plus de contraintes avec celles déjà affectées pour découvrir les échecs le plus tôt possible.

Les heuristiques statiques ont des résultats souvent médiocres comparées aux heuristiques dynamiques. Les ordres de ces dernières sont calculés durant la recherche. Elles tiennent compte de l'évolution du problème pour faire des choix mieux éclairés. La technique de consistance avant qui permet de filtrer les domaines des variables non instanciées, entraînent en général de fortes modifications de la partie non explorée du problème suivant l'affectation courante. Donc, la définition d'un ordre d'affectation qui s'adapte à ces modifications, donne en général une amélioration significative des performances.

On peut citer parmi les heuristiques dynamiques :

- dom [HE80] : elle choisit comme prochaine variable à instancier une de taille de domaine courant minimum, en cas d'égalité le choix est arbitraire. En affectant les variables de taille de domaines minimum d'abord, on réduit le nombre de possibilités au début de l'arbre de recherche. De ce fait, on peut espérer une diminution importante de la taille de l'arbre.
- dom+deg [FD95] : elle choisit comme prochaine variable à instancier une de taille de domaine courant minimum, en cas d'égalité, elle choisit une de degré maximum. C'est une combinaison de l'heuristique précédente avec maxdeg dont l'objectif est d'amplifier la réduction de l'arbre de recherche.
- dom/deg [BR96] : elle choisit comme prochaine variable à instancier une qui minimise le rapport entre la taille du domaine courant et le degré. Elle cherche en même temps à minimiser les branches possibles au début de l'arbre de recherche et à découvrir les échecs le plus tôt possible.
- dom/futdeg [BR96] : elle choisit comme prochaine variable à instancier une qui minimise le rapport entre la taille du domaine courant et le nombre de variables voisines non instanciées. Elle raffine le critère de sélection de l'heuristique précédente.

[BCS01] propose des heuristiques multi-niveaux qui appliquent les heuristiques précédentes à différents régions définies par la variable courante.

Les heuristiques d'ordonnancement des valeurs ont un impact limité et ont été très peu étudiées. [FD95] a proposé la meilleure heuristique connue min-conflicts. Elle compte pour

chaque valeur v de la variable courante, le nombre de valeurs des variables non instanciées avec lesquelles v est incompatible. Les valeurs sont ordonnées dans l'ordre croissant des nombres de conflits.

2.4 VCSP

Une des principales faiblesses du formalisme CSP réside dans l'incapacité de prendre en compte des contraintes dites molles. Ces dernières font référence à des possibilités, préférences, souhaits qu'il faut satisfaire si possible. Malheureusement, les contraintes (dures) dans un CSP doivent impérativement être satisfaites et ne permettent donc pas d'exprimer ces notions. En outre pour un problème inconsistant, trouver une affectation complète qui viole le moins de contraintes possible peut être souhaitable. Tout cela crée une préférence au niveau des affectations complètes (solutions). Pour prendre en compte cette notion de préférence, plusieurs extensions du formalisme CSP ont été introduites : les CSP pondérés (WCSP), les CSP possibilistes [Sch92], les CSP flous [RHZ76, DFP93, Rut94], les CSP à pénalités (Max-CSP [FW92] et WCSP), probabilistes [FL93, FLMCS95], lexicographiques [FLS93], Semiring CSP (SCSP [BMR95]) , CSP valués (VCSP [SFV95])... Nous allons nous focaliser sur l'extension CSP valués qui est un cadre générique qui englobe de multiples formalismes. Cependant, ce travail peut être facilement adapté aux SCSP qui constituent un cadre équivalent [BFM$^+$96].

2.4.1 Présentation du formalisme

L'objectif des problèmes d'optimisation est de trouver une affectation complète de coût optimum (minimum ou maximum) suivant un certain critère exprimé par les contraintes (recherche de la solution préférée). La première définition du cadre VCSP introduit la notion de structure de valuation associée aux contraintes du problème pour caractériser le critère de préférence sur les affectations.

Définition 2.4.1 *[SFV95, SFV97] Une structure de valuation est un triplet $(E, \preceq, \oplus)$ avec un ensemble de valuations (coûts) totalement ordonné par $\preceq$, muni d'un élément minimum (noté $\bot$), d'un élément maximum (noté $\top$) et d'une loi de composition interne (notée $\oplus$) commutative, associative, monotone et telle que $\bot$ soit un élément neutre pour $\oplus$ et $\top$ un élément absorbant.*

Les valuations caractérisent le degré de violation des critères de préférence associés aux contraintes. $\bot$ caractérise une absence de violation et $\top$ une violation inacceptable. La loi de combinaison interne des valuations $\oplus$ permet d'agréger les valuations si une affectation viole plusieurs contraintes. Cette structure de valuation permet de définir la notion de CSP valué.

Définition 2.4.2 *[SFV95, SFV97] Un CSP valué (VCSP) est sextuplet $\mathcal{P} = (X, D, C, R, S, \phi)$ avec (X, D, C, R) un CSP classique, doté d'une structure de valuation $(E, \preceq, \oplus)$ et d'une application ϕ de C dans E qui associe une valuation à chaque contrainte du CSP.*

Une définition équivalente à cette première et plus en adéquation avec la résolution pratique, a été proposée par Larrosa dans [Lar02]. Cette fois-ci, les valuations portent non plus sur les contraintes mais sur les tuples. De ce fait, les contraintes deviennent des fonctions de valuations qui associent à chaque tuple une valuation. En plus, des contraintes unaires sont ajoutées pour chaque variable et donne une valuation à chaque valeur de son domaine (ces contraintes peuvent être définies de manière naïve au départ : toutes les valeurs ont $\bot$ pour valuation). Une dernière contrainte ne portant sur aucune variable est rajoutée avec pour but de mémoriser un minorant de la valuation optimale du VCSP.

Définition 2.4.3 *[Lar02] Un CSP valué est un quadruplet $\mathcal{P} = (X, D, C, S)$ avec :*

- $X = \{x_1, \ldots, x_n\}$, *un ensemble de variables*
- $D = \{D_{x_1}, \ldots, D_{x_n}\}$, *un ensemble de domaines finis et discrets tel que* $\forall 1 \leq i \leq n$, D_{x_i} *est le domaine associé à la variable* x_i
- S, *une structure de valuation*
- $C = \{C_\emptyset\} \cup C_{unaires} \cup C^+$, *un ensemble d'applications, avec*
  - $C_\emptyset$ *un minorant de la valuation optimale*
  - $C_{unaires} = \{C_1, \ldots, C_n\}$ *(C_i est associée à x_i)*
    où $C_i : D_{x_i} \rightarrow E$ *associe à chaque valeur* $v \in D_{x_i}$ *une valuation* $C_i(v) \in E$.
  - C^+ *est l'ensemble des autres contraintes ($C_{i_1,\ldots,i_{n_r}} \in C^+$ est définie sur un ensemble de variables $\{x_{i_1}, \ldots, x_{i_{n_r}}\}$)*
    où $C_{i_1,\ldots,i_{n_r}} : D_{x_{i_1}} \times \ldots \times D_{x_{i_{n_r}}} \rightarrow E$ *associe à chaque tuple* $t \in D_{x_{i_1}} \times \ldots \times D_{x_{i_{n_r}}}$ *une valuation* $C_{i_1,\ldots,i_{n_r}}(t) \in E$.

On note X_c l'ensemble des variables sur lequel est défini la contrainte c.

Définition 2.4.4 *[SFV95, SFV97] Soient $\mathcal{P} = (X, D, C, S)$ un VCSP et $\mathcal{A}$ une affectation totale sur X. La valuation de $\mathcal{A}$ est définie par :*
$\mathcal{V}_\mathcal{P}(\mathcal{A}) = \bigoplus_{c \in C} c(\mathcal{A}[X_c])$.

Dans la suite, nous allons utiliser cette définition de VCSP.

Dans tous les cas, le problème VCSP consiste donc à trouver une affectation totale de valuation minimum au sens de $\preceq$. Cette valuation optimale est appelée **valuation du VCSP**. Nous la noterons $\alpha^*_\mathcal{P}$ dans la suite. Le problème VCSP est NP-difficile.
Pour avoir des méthodes de résolution efficaces, il est nécessaire de disposer de la notion de valuation d'une affectation partielle.

Définition 2.4.5 *[SFV95, SFV97] Soient $\mathcal{P} = (X, D, C, S)$ un VCSP et $\mathcal{A}$ une affectation sur $Y \subset X$. La valuation de $\mathcal{A}$ est définie par :*
$v_\mathcal{P}(\mathcal{A}) = \bigoplus_{c \in C | X_c \subset Y} c(\mathcal{A}[X_c])$.

Cette valuation d'affectation partielle va permettre de calculer un minorant du coût de l'affectation en cours de construction grâce au résultat suivant.

Théorème 2.4.1 *[SFV95, SFV97] Soient $\mathcal{P} = (X, D, C, S)$ un VCSP, $\mathcal{A}$ une affectation complète et $\mathcal{B} \subset \mathcal{A}$. On a :*
$v_\mathcal{P}(\mathcal{B}) \preceq v_\mathcal{P}(\mathcal{A}) = \mathcal{V}_\mathcal{P}(\mathcal{A})$.

Il est donc possible de calculer de manière incrémentale la valuation de l'affectation courante.

Exemple 2.4.1 *Considérons le VCSP $\mathcal{P}$ défini comme suit :*
- $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$,
- $D = \{D_{x_1}, D_{x_2}, D_{x_3}, D_{x_4}, D_{x_5}, D_{x_6}, D_{x_7}, D_{x_8}\}$ *avec* $D_{x_i} = \{1, 2, 3\}, \forall 1 \leq i \leq 8$,
- $C_\emptyset = 0$
- $C_{unaires} = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8\}$ *avec* $C_i(v) = 0, \forall 1 \leq i \leq 8$ *et* $v \in D_{x_i}$.
- $C^+ = \{C_{18}, C_{25}, C_{27}, C_{34}, C_{36}, C_{38}, C_{46}, C_{56}, C_{58}, C_{68}, C_{78}\}$, C_{ij} *porte sur* $\{x_i, x_j\}$,
- $S = (\mathbb{N}, \leq, +)$.

Les contraintes unaires sont définies de manières naïves : la valuation de chaque valeur est 0. Les tables 2.4, 2.5 et 2.6 donnent les valuations des tuples associées aux contraintes C^+ du VCSP.

C_{18}			C_{25}			C_{27}			C_{34}			C_{36}		
x_1	x_8	coût	x_2	x_5	coût	x_2	x_7	coût	x_3	x_4	coût	x_3	x_6	coût
1	1	2	1	1	1	1	1	2	1	1	∞	1	1	∞
1	2	3	1	2	1	1	2	3	1	2	1	1	2	1
1	3	4	1	3	0	1	3	4	1	3	0	1	3	0
2	1	1	2	1	2	2	1	1	2	1	1	2	1	1
2	2	2	2	2	1	2	2	2	2	2	∞	2	2	∞
2	3	3	2	3	1	2	3	3	2	3	1	2	3	1
3	1	0	3	1	3	3	1	0	3	1	0	3	1	0
3	2	1	3	2	2	3	2	1	3	2	1	3	2	1
3	3	2	3	3	1	3	3	2	3	3	∞	3	3	∞

Tab. 2.4 – Tables des valuations des contraintes C^+ de $\mathcal{P}$ (1/3).

C_{38}			C_{46}			C_{56}			C_{58}			C_{68}		
x_3	x_8	coût	x_4	x_6	coût	x_5	x_6	coût	x_5	x_7	coût	x_5	x_8	coût
1	1	2	1	1	∞	1	1	1	1	1	1	1	1	2
1	2	0	1	2	1	1	2	2	1	2	2	1	2	0
1	3	2	1	3	0	1	3	3	1	3	3	1	3	2
2	1	0	2	1	1	2	1	1	2	1	1	2	1	0
2	2	2	2	2	∞	2	2	1	2	2	1	2	2	2
2	3	3	2	3	1	2	3	2	2	3	2	2	3	3
3	1	2	3	1	0	3	1	0	3	1	0	3	1	2
3	2	3	3	2	1	3	2	1	3	2	1	3	2	3
3	3	4	3	3	∞	3	3	1	3	3	1	3	3	4

Tab. 2.5 – Tables des valuations des contraintes C^+ de $\mathcal{P}$ (2/3).

C_{78}		
x_6	x_8	coût
1	1	3
1	2	2
1	3	0
2	1	2
2	2	0
2	3	2
3	1	0
3	2	2
3	3	3

Tab. 2.6 – Tables des valuations des contraintes C^+ de $\mathcal{P}$ (3/3).

Le formalisme VCSP permet de généraliser simplement le formalisme classique et de capturer différentes extensions très utilisées telles que : les CSP possibilistes [Sch92] et flous [RHZ76, DFP93, Rut94], à pénalités (Max-CSP [FW92]), probabilistes [FL93, FLMCS95], lexicographiques [FLS93].

2.4.2 Les méthodes de résolution

La recherche d'une affectation complète de valuation minimum (une solution optimale) est beaucoup plus difficile que celle d'une solution dans le cadre CSP. En effet, la preuve de l'optima-

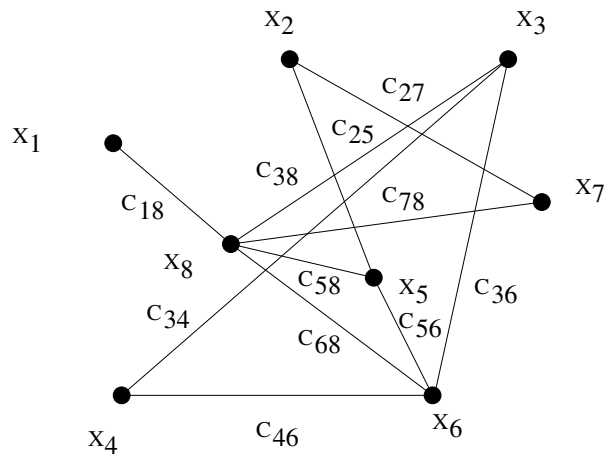

FIG. 2.12 – Graphe de contraintes de $\mathcal{P}$.

lité d'une solution (l'inexistence d'une meilleure affectation) nécessite le parcours de la totalité de l'espace de recherche. Néanmoins, ce parcours peut être plus ou moins efficace dans la mesure où certaines affectations partielles de qualité moyenne n'ont pas besoin d'être complétées pour déterminer que leur valuation sera supérieure à l'optimum. Généralement, les méthodes de résolution sont des adaptations des méthodes du cadre décision au cadre optimisation. La méthode classique est le branch and bound (séparation / évaluation).

2.4.2.1 Le Branch and Bound (BB) (Algorithme 4).

La méthode du branch and bound consiste à parcourir l'espace de recherche de la même manière que le Backtracking, en maintenant deux bornes : un majorant de la valuation optimale (noté ub) et un minorant de la valuation de l'affectation courante (un minorant de la valuation de l'affectation complète qui est en cours de construction) (noté lb). $\mathcal{A}$ est l'affectation courante et V l'ensemble des variables à instancier. Durant la recherche, si le minorant dépasse le majorant, on sait que la valuation de l'affectation en cours de construction est supérieure à l'optimum. L'extension de cette affectation est stoppée et un backtrack réalisé sur la dernière variable instanciée pour tester une nouvelle valeur (boucle while : lignes 7-16). Si toutes les valeurs du domaine de la variable ont déjà été essayées, il y a un retour à la variable qui la précède. Par contre, si le minorant est plus petit que le majorant, on passe à la variable suivante (ligne 5). Si toutes les variables ont été affectées (ligne 1), une affectation de valuation inférieure au majorant vient d'être construite. Elle est mémorisée comme la meilleure solution courante et une mise à jour du majorant est réalisée (lignes 2-3). La recherche continue avec un backtrack sur la dernière variable. Lorsque tout l'espace de recherche a été exploré, la valuation de la meilleure solution rencontrée constitue donc la valuation optimale du VCSP.

Le minorant et le majorant sont primordiaux dans l'efficacité de la méthode : plus leur qualité est bonne (majorant est petit et le minorant est grand), plus le nombre de branches élaguées sera élevé assurant ainsi une méthode efficace. Dans l'algorithme standard, le majorant utilisé, est la valuation de la meilleure solution rencontrée jusque-là et le minorant est la valuation locale de l'affectation courante.

Pour améliorer cette méthode (plus précisément son minorant), des techniques de filtrages basées sur des cohérences locales plus ou moins fortes ont été proposées. Ces dernières sont pour la plupart des adaptations de l'arc cohérence définie dans le cadre classique. La méthode FC a été adaptée aux VCSP [FW92, SFV95, SFV97] donnant naissance à une première consistance locale

Algorithme 4 : BB($\mathcal{A}, V, lb, ub$)

```
1  if V = ∅ then
2      Mémoriser la solution A
3      return lb
4  else
5      Choisir x ∈ V
6      d_x ← D_x
7      while d_x ≠ ∅ and lb ≺ ub do
8          Choisir v ∈ d_x
9          d_x ← d_x\{v}
10         L ← {c ∈ C|X_c ⊂ (X\(V\{x}))}
11         lb_v ← ⊥
12         while L ≠ ∅ and lb ⊕ lb_v ≺ ub do
13             Choisir c in L
14             L ← L\{c}
15             lb_v ← lb_v ⊕ c((A ∪ {x ← v})[X_c])
16         if lb ⊕ lb_v ≺ ub then ub ← min(ub,BB(A ∪ {x ← v}, V\{x}, lb ⊕ lb_v, ub)
17     return ub
```

sur les VCSP. Ensuite, des niveaux de consistance supérieurs ont été définis sur des restrictions de la classe des VCSP.

2.4.2.2 Le filtrage

Les techniques de filtrage dans le cadre CSP classique ont permis une très nette amélioration de l'efficacité des méthodes de résolution. Les tentatives d'extension des techniques de filtrage de ce cadre vers les VCSP ont été nombreuses et généralement restreintes aux Max-CSP et WCSP binaires. Cependant, la plupart des techniques présentées ici, a été généralisée aux contraintes n-aires ([CS04]). Mais pour des raisons de simplicité, nous préférons la présentation du cas binaire. Ici, le filtrage a deux objectifs : une réduction de la taille du problème par réduction des domaines (quand une valeur ne peut pas participer à l'extension de l'affectation courante car la valuation résultante dépasserait le majorant, elle est supprimée) et le calcul d'un minorant de qualité permettant un élagage important dans l'espace de recherche.

Le minorant de base des algorithmes de branch and bound est donné par la valuation locale de l'affectation partielle courante. L'adaptation de FC aux VCSP (FC-val, Algorithmes 5 et 6) a conduit à l'amélioration de ce minorant. Ce nouveau minorant tient compte des variables non affectées, en plus de celles déjà affectées. FC-val effectue une recherche du type branch and bound avec $\mathcal{A}$ l'affectation courante, V l'ensemble des variables à instancier, ub le majorant (valuation de la meilleure solution connue), lb la valuation locale de l'affectation courante et lb_{FC}, le minorant amélioré. Le calcul de lb_{FC} repose sur l'observation suivante. Soit $\mathcal{A}' = \mathcal{A} \cup \{x \leftarrow v\}$, l'extension de $\mathcal{A}$ sur la variable x par la valeur v. La valuation locale de $\mathcal{A}'$ est donnée par celle de $\mathcal{A}$ combinée avec tous les coûts de la forme $c(\mathcal{A}'[X_c]) \neq \bot$, c étant une contrainte qui relie x à une variable déjà affectée dans $\mathcal{A}$. La différence éventuelle entre la valuation de $\mathcal{A}$ et de $\mathcal{A}'$ réside en totalité dans le coût induit par ces contraintes c reliant x aux variables affectées. Sachant cela, FC-val va anticiper cette augmentation éventuelle entre ces deux valuations en considérant tout d'abord cet ensemble de contraintes pour chaque variable x non instanciée : $L(x) = \{c \in C | x \in X_c \subset (X_{\mathcal{A}} \cup \{x\})\}$. Si $L(x) = \emptyset$, les valuations des deux affectations $\mathcal{A}$ et $\mathcal{A}'$ sont égales. Si, par contre, $L(x) \neq \emptyset$, la différence éventuelle (plus précisément l'augmentation) dépend de la valeur v affectée à x. Pour une valeur v du domaine de x, elle est donnée par la formule :

Algorithme 5 : FC-val($\mathcal{A}, V, lb, lb_{FC}, ub$)

```
1  if V = ∅ then
2      Mémoriser la solution A
3      return lb
4  else
5      Choisir x ∈ V
6      d_x ← D_x
7      while d_x ≠ ∅ and lb_FC ≺ ub do
8          Choisir v ∈ d_x
9          d_x ← d_x\{v}
10         Push-Domains
11         lb_FC,v ← Mise-à-jour(A ∪ {x ← v}, V\{x}, lb ⊕ ic(x, v), x)
12         if lb_FC,v ≺ ub then
               ub ← min(ub,FC-val(A ∪ {x ← v}, V\{x}, lb ⊕ ic(x, v), lb_FC,v, ub))
13         Pop-Domains
14     return α
```

Algorithme 6 : Mise-à-jour($\mathcal{A}, V, lb, x$)

```
1  Pour tout y L ← {c ∈ C|X_c ⊂ (X\(V\{y}))}
2  while L ≠ ∅ do
3      Choisir c ∈ L
4      L ← L\{c}
5      for tout v' ∈ d_y do
6          ic(y, v') ← ic(y, v') ⊕ c((A ∪ {y ← v'})[X_c])
   return lb ⊕ ⊕_{y∈V} (min_{v'∈d_y} ic(y, v'))
7
```

$ic(x,v) = \bigoplus_{c \in L(x)} c(\mathcal{A} \cup \{x \leftarrow v\}[X_c])$. L'augmentation minimum est donc $min_{v \in D_x} ic(x,v)$. On sait, dès lors, que l'augmentation entre la valuation locale de $\mathcal{A}$ et celle de son extension à la variable x est au minimum de $min_{v \in D_x} ic(x,v)$. Il faudra payer au moins ce coût pour toute affectation de la variable x. En tenant compte de cela, FC-val rajoute directement cette valeur au minorant pour toute variable non affectée. Ainsi, lb_{FC} est la valuation locale de l'affectation courante $v(\mathcal{A})$ à laquelle on rajoute $\bigoplus_{x \in V} min_{v \in D_x} ic(x,v)$.

Au début de la recherche, $ic(x,v) = \bot, \forall x \in V, v \in D_x$, ensuite les $ic(x,v)$ sont mises à jour incrémentalement à chaque affectation d'une variable. Les procédures Push-Domains et Pop-Domains permettent la sauvegarde et la restauration des $ic(x,v)$ durant la résolution.

Ainsi, lb_{FC} est un minorant de plus grande qualité que celui de BB ce qui permet un élagage plus important. En outre, toute valeur w d'une variable non instanciée y telle que $v(\mathcal{A}) \oplus ic(y,w) \oplus \bigoplus_{x \in V - \{y\}} min_{v \in D_x} ic(x,v)$ dépasse le majorant est supprimée du domaine D_y car l'affectation de w à y conduit forcément à une affectation de valuation supérieure au majorant.

Le challenge pendant des années a été de définir une extension du filtrage par consistance d'arc aux VCSP qui permettrait donc de prendre en compte des contraintes entre des variables non instanciées. Cette extension était évidente pour les VCSP avec une loi de composition interne idempotente qui fait que si on considère à plusieurs reprises une même contrainte, la combinaison des valuations n'est pas augmentée pour autant. Cependant, elle était beaucoup plus complexe pour les lois non idempotentes. En effet, avec une adaptation de MAC

aux VCSP similaire à FC-val, il est possible de considérer une même contrainte plusieurs fois. Si la loi n'est pas idempotente cela mène à une sur-évaluation du minorant faussant ainsi le résultat obtenu. Une solution consiste donc à faire une partition des contraintes et calculer un minorant sur les différentes partitions. Pour éviter de prendre en compte des contraintes à plusieurs reprises, [Wal94, LM96] ont proposé une orientation des contraintes avec la consistance d'arc directionnelle (DAC). Cette méthode utilise un ordre d'affectation statique sur les variables. Le minorant est donné par la combinaison de la valuation locale de l'affectation courante et le coût minimum qui sera induit par l'affectation ultérieure des variables non encore instanciées : $lb = v(\mathcal{A}) \oplus \bigoplus_{x \in V} min_{v \in d_x}(ic(x,v) \oplus dac(x,v))$. $ic(x,v)$, déjà défini dans le cadre de la méthode FC-val, concerne les contraintes qui lient x à une variable déjà affectée et donne ainsi avec $v(\mathcal{A})$ la valuation de l'extension $\mathcal{A} \cup \{x \leftarrow v\}$ de $\mathcal{A}$. $dac(x,v)$ prend en compte les contraintes qui lie x aux variables non affectées y. Elle donne le coût minimum qui sera engendré par l'extension $\mathcal{A} \cup \{x \leftarrow v\}$ sur les variables non affectées. Elle est calculée en prétraitement, avant le début de la recherche. En plus, les valeurs, w telles que $v(\mathcal{A}) \oplus ic(y,w) \oplus dac(y,w) \bigoplus_{x \in V - \{y\}} min_{v \in d_x}(ic(x,v) \oplus dac(x,v))$ dépasse le majorant sont supprimées des domaines, car ne pouvant mener à une meilleure solution. Nous avons là, une amélioration du minorant de FC-val et donc une amélioration de cette méthode appelée PFC-DAC. Dans [LMS99], il a été montré qu'il est possible de s'affranchir de l'ordre statique d'une part et d'autre part il est possible d'utiliser un graphe des contraintes orienté quelconque. En outre, cette orientation des contraintes peut également être modifiée durant la résolution définissant ainsi une nouvelle consistance valuée locale : RDAC (DAC réversible). Cela a conduit à la méthode PFC-MRDAC qui maintient RDAC au cours de la recherche avec une mise à jour des $dac(x,v)$. Elle a constitué ces dernières années, l'algorithme de référence dans la résolution de VCSP.

Des consistances plus fortes encore ont été proposées pour une sous-classe de VCSP : les VCSP justes.

Définition 2.4.6 *Un VCSP $\mathcal{P} = (X, D, C, S)$ est juste si S vérifie la condition suivante : si $u \preceq v$ alors il existe w maximale (et donc unique) telle que $u \oplus w \preceq v$, notée $v \ominus u$.*

Les WCSP sont des VCSP justes qui permettent de représenter et résoudre un grand nombre de problèmes. Dans [Coo05], il est montré que ce cadre permet de capturer la complexité essentielle des VCSP non idempotents. Les consistances locales suivantes ont été définies dans le cadre des WCSP binaires auquel nous allons très souvent nous restreindre. Mais, les résultats présentés sont valables pour la totalité de la classe des VCSP justes binaires. Avec l'opération de soustraction induite dans les WCSP, il est possible de déplacer des valuations à travers les contraintes pour ainsi calculer un minorant de plus grande qualité et aussi filtrer plus de valeurs des domaines. Cela suppose une représentation légérement différente mais équivalente des WCSP, donnée par Larrosa dans [Lar02]. Un WCSP est un VCSP avec la structure de valuation suivante : $S(E = [0,k], \leq, \oplus)$ avec $[0,k]$ dans $\mathbb{N}$, $k \in \mathbb{N} \cup \{\infty\}$ et $a \oplus b = min(k, a+b)$. Si $k = \infty$, on retrouve la représentation usuelle des WCSP. Cependant pour un k fini, même si toutes les propriétés de la structure de valuations sont respectées, nous obtenons une propriété supplémentaire qui fait qu'il est possible d'atteindre $\top$ par addition d'un nombre fini de valuations. Cela permet d'avoir une représentation plus en adéquation avec la résolution pratique. En effet, la représentation du problème va permettre directement le filtrage de valeurs pour cause d'égalité avec $\top$ lors de l'application de la consistance locale. Les notions de consistance de nœud ([Mac77]) et d'arc ([Wal72]), introduites dans le cadre classique, ont été adaptées aux WCSP dans [Lar02] et des définitions équivalentes (NC^* et AC^*) sont présentées dans [LS03]. Dans ce dernier article, les auteurs proposent des adaptations de la consistance d'arc directionnelle (DAC) et la consistance d'arc directionnelle complète (FDAC) introduites par [Coo03] : DAC^* et $FDAC^*$.

Définition 2.4.7 *On suppose que les variables sont totalement ordonnées.*

- *Consistance de nœud. Une valeur $a \in D_{x_i}$ est nœud étoile consistante si $C_\emptyset \oplus C_i(a) < \top$. La variable x_i est nœud étoile consistante si :*
  i)toutes les valeurs de son domaine sont NC^* et
  ii) il exite une valeur $a \in D_{x_i}$ telle que $C_i(a) = \bot$ (a est un support de x_i). Un WCSP est NC^* si toutes ses variables sont NC^*.
- *Consistance d'arc. Une valeur $a \in D_{x_i}$ est arc consistante (AC) par rapport à la contrainte C_{ij} s'il existe une valeur $b \in D_{x_j}$ telle que $C_{ij}(a, b) = \bot$ (b est un support de a). La variable x_i est AC si toutes ses valeurs sont AC par rapport à toutes les contraintes binaires contenant x_i. Un WCSP est AC^* s'il est AC et NC^*.*
- *Consistance d'arc directionnelle. Une valeur $a \in D_{x_i}$ est arc consistante directionnelle (DAC) par rapport à la contrainte $C_{ij}, j > i$, s'il existe une valeur $b \in D_{x_j}$ telle que $C_{ij}(a, b) \oplus C_j(b) = \bot$ (b est un support total de a). La variable x_i est DAC si toutes ses valeurs sont DAC par rapport à toutes les contraintes binaires contenant x_i. Un WCSP est DAC^* s'il est DAC et NC^*.*
- *Consistance d'arc directionnelle totale. Un WCSP est $FDAC^*$ s'il est DAC^* et AC^*.*

AC^* et $FDAC^*$ reviennent à la consistance d'arc dans le cadre classique. Des méthodes pour transformer un problème en un problème équivalent NC^*, AC^*, DAC^* ou $FDAC^*$ ont été proposées. Elles sont basées sur l'application d'une succession d'opérations élémentaires qui consistent à filtrer des domaines (les valeurs qui ne sont pas NC^* sont supprimées), transférer des coût de contraintes unaires vers le minorant ou vers des contraintes binaires. L'objectif majeur étant de faire augmenter le plus possible le minorant, des coûts sont déplacés vers une variable donnée et puis de cette variable vers le minorant dans le but de trouver ou construire des supports (totaux) pour les valeurs des domaines. W-AC^*2001 a été proposée par [Lar02] et est basée sur AC2001 [BR01]. [LS03] ont proposé des algorithmes de consistance d'arc directionnelle (DAC^*) (complexités en temps en $O(md^2)$ et en espace en $O(md)$) et de consistance d'arc directionnelle totale ($FDAC^*$) (complexités en temps en $O(mnd^3)$ et en espace en $O(md)$) avec une structure proche de W-AC^*2001. Les auteurs ont comparé des techniques qui maintiennent ces consistances durant la recherche avec la méthode de référence PFC-(M)RDAC sur des problèmes aléatoires. Il en ressort que $MFDAC^*$ malgré sa complexité est la meilleure méthode suivie de près par $MDAC^*$. Le problème d'optimisation étant beaucoup plus difficile que la décision, le gain de ces filtrages a une importance accrue dans ce cadre. Le temps passé donc à calculer ces consistances locales, même complexes, n'est pas très significatif face au temps de résolution du problème dans sa totalité. Pour abonder dans le même sens, [dGHZL05] proposent $EDAC^*$ qui se rapproche plus encore que $FDAC^*$ de la consistance globale dans les CSP classiques.

Définition 2.4.8 *Une variable x_i est arc consistante existentielle (EAC^*) s'il existe au moins une valeur $a \in D_{x_i}$ telle que $C_i(a) = \bot$ et x_i a un support total dans chaque C_{ij}. Un WCSP est EAC^* si toutes ses variables sont EAC^* et NC^*. Un WCSP est $EDAC^*$ si toutes ses variables sont $FDAC^*$ et EAC^*.*

Un algorithme de consistance d'arc existentielle ($EDAC^*$) est proposé, avec une complexité en temps en $O(md^2max(nd, \top))$ et en espace en $O(md)$. $MFDAC^*$ a été comparée avec la méthode de résolution $MEDAC^*$ qui maintient $EDAC^*$, sur des WCSP aléatoires ou réels, binaires ou n-aires. Dans le cas des WCSP n-aires, la propagation sur une contrainte est retardée jusqu'à ce qu'elle devienne binaire, c'est-à-dire qu'elle ne contienne plus que deux variables non instanciées. $MEDAC^*$ obtient toujours les meilleurs résultats.

Un problème sur ces consistances se trouve au niveau du nombre de fermetures consistantes qui contrairement au cadre CSP classique, peut être supérieur à 1. En effet, suivant l'ordre dans lequel les coûts sont déplacés à travers les contraintes, on se retrouve avec des WCSP équivalents localement arc consistants valués différents. [CS04] ont prouvé que trouver une fermeture arc consistante optimale (qui maximise le minorant) avec ces consistances locales, est un problème

NP-difficile. Remplacer la notion de support par support totale, plus forte, semble intéressante mais impossible au niveau algorithmique (on ne peut pas la forcer) donc il faut introduire des restrictions. Les consistances valuées définies sont des heuristiques pour approcher le plus possible la fermeture arc consistante optimale sans garantie. Dans [CdGS07], les auteurs proposent de relaxer la définition de WCSP en autorisant des coûts de valeur rationnelle ($\mathbb{Q}$) alors que la définition de base était sur les entiers naturels. Une nouvelle structure $S_{\mathbb{Q}}(k) = ([0,k], \oplus, \leq)$ avec $[0,k]$ dans $\mathbb{Q}$. En plus de cette structure, ils s'autorisent à déplacer des coûts en simultanée (dans les cas précédents on avait des séquences d'opérations) avec les mêmes opérations. Ils prouvent qu'avec ces conditions, il est possible de transformer le problème en un problème équivalent avec un minorant optimal si $k = \infty$ en un temps polynomial. On a ainsi une consistance plus forte que celles déjà définies. Mais puisqu'il faut un $k = \infty$, cette consistance est utilisée en prétraitement. Malgré son coût non négligeable, elle permet d'améliorer de manière très nette $MEDAC^*$ sur les problèmes particulièrement difficiles. Leur difficulté permet donc d'amortir le coût du prétraitement.

2.4.2.3 La programmation dynamique

Russian Dolls Search (RDS [VLS96]). Cette méthode résout successivement plusieurs sous-problèmes emboités (poupées russes) en utilisant une technique de branch and bound et un ordre sur les variables pré-établi. A chaque étape i, RDS résout le sous-problème induit par les i dernières variables. L'affectation de valuation optimum est enregistrée de même que sa valuation pour ce sous-problème. Ainsi lors de la résolution du problème $i+1$, il sera possible de combiner la valuation de l'affectation partielle avec cet optimum pour avoir un minorant de très grande qualité, mais également d'utiliser la solution du problème i pour guider le choix des valeurs dans l'ordre d'affectation. La complexité de cette méthode est $O(\frac{d}{d-1}.exp(n))$. SRDS [MS01] est une spécialisation de RDS qui résout les sous-problèmes pour chaque valeur de la nouvelle variable permettant ainsi d'avoir un minorant encore meilleur. OSRDS [MSV02] est une extension de SRDS qui fournit un minorant de qualité encore meilleure. TRDS [MS00] (Tree based RDS) est une extension de RDS qui exploite les indépendances entre sous-problèmes pour construire un minorant meilleur que celui de RDS et aussi réduire la taille des sous-problèmes. Elle repose sur un arbre des sous-problèmes construit à l'aide des points d'articulation du problème (séparateurs de taille 1). RDS et SDRS obtiennent de bons résultats grâce à la qualité du minorant et malgré l'ordre statique et les résolutions multiples des différents sous-problèmes.

Programmation dynamique de Koster ([Kos99]). [Kos99] propose une méthode de résolution de VCSP basée sur une décomposition arborescente du graphe de contraintes du problème. Elle commence par résoudre les clusters feuilles et mémorise toutes les affectations possibles sur ces clusters. La résolution d'un cluster est possible dès que tous ses clusters fils ont été résolus. Celle-ci instancie les variables de ce cluster tout en combinant ces affectations avec celles enregistrées sur les clusters fils. A ce niveau, elle considère le cluster courant comme un séparateur. A l'image de BTD-val, elle cherche la meilleure extension possible de chaque affectation sur ce cluster sur le sous-problème dont il est la racine. Ensuite, elle enregistre les combinaisons entre les affectations des clusters du sous-problème ainsi produites. Cette technique est répétée jusqu'au cluster racine, pour lequel, on calcule uniquement la solution optimale. La compexité en temps de cette méthode est en $O(nd^{3w})$ et $O(nd^{w+1})$ en espace, avec w la largeur de la décomposition arborescente. Des prétraitements permettent de réduire la taille du problème et ainsi de résoudre des problèmes d'allocation de fréquences (archive FullRLFAP ([CdGL$^+$99]) inaccessibles jusqu'alors. Cependant la quantité de mémoire requise est un frein pour beaucoup de problèmes. Une amélioration proposée consiste à décomposer le problème par décomposition récursive des domaines. Elle obtient de meilleurs résultats pratiques sur les instances qui explosaient en mémoire mais pour certaines la mémoire requise demeure trop importante.

Algorithme 7 : BTD-val($\mathcal{A}, E_i, V_{E_i}, lb_{E_i}, ub_{E_i}$)

```
1  if V_{E_i} = ∅ then
2      F ← Fils(E_i)
3      lb ← lb_{E_i}
4      while F ≠ ∅ and lb ≺ ub_{E_i} do
5          Choisir E_j ∈ F
6          F ← F\{E_j}
7          if (A[E_j ∩ E_i], α) est un good then lb ← α ⊕ α
8          else
9              α ←BTD-val(A, E_j, E_j\(E_j ∩ E_i), ⊥, ⊤)
10             lb ← lb ⊕ α
11             Enregistrer le good (A[E_j ∩ E_i], α)
12     return lb
13 else
14     Choisir x ∈ V_{E_i}
15     d_x ← D_x
16     while d_x ≠ ∅ and lb_{E_i} ≺ ub_{E_i} do
17         Choisir v ∈ d_x
18         d_x ← d_x\{v}
19         L ← {c ∈ E_{P,E_i} | X_c = {x, y} avec y ∉ V_{E_i}}
20         lb_v ← ⊥
21         while L ≠ ∅ and lb_{E_i} ⊕ lb_v ≺ ub_{E_i} do
22             Choisir c ∈ L
23             L ← L\{c}
24             lb_v ← lb_v ⊕ c(A ∪ {x ← v})
25         if lb_{E_i} ⊕ lb_v ≺ ub_{E_i} then
           ub_{E_i} ← min(ub_{E_i},BTD-val(A ∪ {x ← v}, E_i, V_{E_i}\{x}, lb_{E_i} ⊕ lb_v, ub_{E_i}))
26     return ub_{E_i}
```

Une maîtrise de l'espace mémoire requis est incontournable pour assurer la terminaison de la résolution. En outre la complexité en temps de cette méthode, exponentielle en $3w$, peut être assez proche de n dans de nombreux cas. Elle pourrait même dépasser la complexité de BB. Mais, cela est peut-être dû à une surestimation de la complexité.

2.4.2.4 Les méthodes structurelles

BTD-val ([JT03]) (Algorithme 7). Comme dans le cadre CSP, nous ferons référence à cette méthode très souvent. C'est une extension de BTD aux VCSP. Elle est basée sur une décomposition arborescente du CSP qui induit un ordre d'affectation partiel sur les variables. Soit (E, T) une décomposition arborescente du CSP. BTD-val utilise également un ordre d'énumération compatible comme ordre d'affectation des variables. En outre, quand un cluster E_i est totalement instancié l'affectation se poursuit sur un de ses fils E_j et sa descendance $Desc(E_j)$ (ordre d'énumération compatible). BTD-val résout optimalement le sous-problème induit par $Desc(E_j)$ avec l'hypothèse de l'affectation courante $\mathcal{A}[E_i \cap E_j]$ sur $E_i \cap E_j$. Cet optimum, associé avec l'affectation $\mathcal{A}[E_i \cap E_j]$, peut être enregistré comme un good valué. Ainsi, toute affectation future égale à $\mathcal{A}[E_i \cap E_j]$ sur $E_i \cap E_j$ sera poursuivie sur le reste du problème et l'optimum (du good) directement rajouté au minorant. En effet, toute nouvelle affectation égale à l'affectation $\mathcal{A}[E_i \cap E_j]$ sur $E_i \cap E_j$ induira la résolution d'un même sous-problème.

Dans le papier [JT03], les auteurs utilisaient le terme good valué pour faire référence à ce

Algorithme 8 : FC-BTD-val($\mathcal{A}, E_i, V_{E_i}, lb_{E_i}, lb_{FC}, ub_{E_i}$)

```
1  if V_{E_i} = ∅ then
2      F ← Fils(E_i)
3      lb ← lb_{E_i}
4      while F ≠ ∅ and lb ≺ ub_{E_i} do
5          Choisir E_j ∈ F
6          F ← F\{E_j}
7          if (A[E_j ∩ E_i], α) est un good then lb ← lb ⊕ α
8          else
9              α ←FC-BTD-val(A, E_j, E_j\(E_j ∩ E_i), ⊥, ⊥, ⊤)
10             lb ← lb ⊕ α
11             Enregistrer le good (A[E_j ∩ E_i], α)
12     return lb
13 else
14     Choisir x ∈ V_{E_i}
15     d_x ← D_x
16     while d_x ≠ ∅ and lb_{FC} ≺ ub_{E_i} do
17         Choisir v ∈ d_x
18         d_x ← d_x\{v}
19         Push-Domains
20         lb_{FC,v} ←Mise-à-jour2(E_i, V_{E_i}\{x}, A ∪ {x ← v}, lb_{E_i} ⊕ ic(x, v), x)
21         if lb_{FC,v} ≺ ub_{E_i} then ub_{E_i} ← min(ub_{E_i},FC-BTD-
           val(A ∪ {x ← v}, E_i, V_{E_i}\{x}, lb_{E_i} ⊕ ic(x, v), lb_{FC,v}, ub_{E_i}))
22         Pop-Domains
23     return ub_{E_i}
```

que nous appelons ici good valué. Etant donné que nous faisons référence à la solution optimale sur un sous-problème donné avec sa valuation, le terme good valué semble plus approprié. Aussi, ils ont utilisé ce terme dans leurs récentes communications concernant BTD-val.

Définition 2.4.9 *Soient E_i un cluster et E_j l'un de ses fils. Un good valué structurel de E_i par rapport à E_j est le coût de l'affectation optimale sur $Desc(E_j)$ pour une instanciation sur $E_i \cap E_j$.*

Pour résoudre efficacement les sous-problèmes enracinés aux différents clusters, [JT03] définit la notion de VCSP induit qui repose sur les contraintes propres des clusters. Cette partition des contraintes permet d'éviter de comptabiliser plusieurs fois le coût induit par une contrainte entrainant ainsi une surestimation de la valuation locale d'une affectation.

Définition 2.4.10 *Soit E_i un cluster. L'ensemble des contraintes propres à E_i est $\mathcal{E}_{\mathcal{P},E_i} = \{c \in C | X_c \subset E_i, c \not\subset Pere(E_i)\}$, où $Pere(E_i)$ désigne le cluster père de E_i.*

Définition 2.4.11 *Soient E_i un cluster, E_j l'un de ses fils et $\mathcal{A}$ une affectation sur $E_i \cap E_j$. $\mathcal{P}_{\mathcal{A},E_i/E_j} = (X_{\mathcal{P}_{\mathcal{A},E_i/E_j}}, D_{\mathcal{P}_{\mathcal{A},E_i/E_j}}, C_{\mathcal{P}_{\mathcal{A},E_i/E_j}}, S)$ est le CSP induit par $\mathcal{A}$ sur $Desc(E_j)$, avec :*

- $X_{\mathcal{P}_{\mathcal{A},E_i/E_j}} = Desc(E_j)$
- $D_{\mathcal{P}_{\mathcal{A},E_i/E_j}} = \left\{D_{x,\mathcal{P}_{\mathcal{A},E_i/E_j}} = D_x | x \in Desc(E_j)\backslash(E_i \cap E_j)\right\} \cup \left\{D_{x,\mathcal{P}_{\mathcal{A},E_i/E_j}} = \{\mathcal{A}[x]\} | x \in E_i \cap E_j\right\}$
- $C_{\mathcal{P}_{\mathcal{A},E_i/E_j}} = \mathcal{E}_{\mathcal{P},E_j} \cup \bigcup_{E_d \text{ descendant de } E_j} \mathcal{E}_{\mathcal{P},E_d}$

Algorithme 9 : Mise-à-jour2($E_i, V_{E_i}, \mathcal{A}, lb, x$)

$L \leftarrow \{c \in \bigcup_{E_k \subset Desc(E_i)} \mathcal{E}_{\mathcal{P},E_k} | X_c = \{x, y\} \text{ avec } y \in V_{E_i}\}$
1
2 **while** $L \neq \emptyset$ **do**
3 Choisir $c \in L$
4 $L \leftarrow L \backslash \{c\}$
5 **for** *tout* $v' \in d_y$ **do**
6 $ic(y, v') \leftarrow ic(y, v') \oplus c(\mathcal{A} \cup \{y \leftarrow v'\})$
return $l \oplus \bigoplus_{y \in Desc(E_i) \backslash (E_i \backslash V_{E_i})} (min_{v' \in d_y} ic(y, v'))$
7

Le VCSP induit $\mathcal{P}_{\mathcal{A},E_i/E_j}$ correspond au VCSP de départ restreint aux variables du sous-arbre enraciné en E_j et dont les domaines des variables de $E_i \cap E_j$ sont restreintes à leur valeur dans $\mathcal{A}$. En outre, [JT03] définit la notion de valuation locale à un cluster.

Définition 2.4.12 *Soient E_i un cluster et $\mathcal{A}$ une affectation sur $Y \subset X$ avec $Y \cap E_i \neq \emptyset$. La valuation locale $v_{\mathcal{P},E_i}(\mathcal{A})$ de $\mathcal{A}$ au cluster E_i est la valuation locale de $\mathcal{A}$ restreinte aux contraintes de $E_{\mathcal{P},E_i}$.*
$v_{\mathcal{P},E_i}(\mathcal{A}) = \bigoplus_{c \in \mathcal{E}_{\mathcal{P},E_i} | X_c \subset Y} c(\mathcal{A}[X_c])$.

Ces valuations locales aux clusters possèdent une propriété intéressante : pour toute affectation totale $\mathcal{A}$ sur X, $\mathcal{V}_{\mathcal{P}}(A) = \bigoplus_{E_i \subset X} v_{\mathcal{P},E_i}(\mathcal{A})$. En outre, il a été démontré dans [JT03] le théorème suivant.

Théorème 2.4.2 *Soient E_i un cluster et $\mathcal{A}$ une affectation sur $E_i \cap Pere(E_i)$.*
$\alpha^*_{\mathcal{P}_{\mathcal{A},Pere(E_i)/E_i}} = min_{\mathcal{B}|X_{\mathcal{B}}=E_i \text{ et } \mathcal{B}[E_i \cap Pere(E_i)]=\mathcal{A}}(v_{\mathcal{P},E_i}(\mathcal{B}) \oplus \bigoplus_{E_j \in Fils(E_i)} \alpha^*_{\mathcal{P}_{\mathcal{B}[E_i \cap E_j],E_i/E_j}})$.

Avec ce résultat, le calcul de la valuation optimale du VCSP induit $\mathcal{P}_{\mathcal{A},Pere(E_i)/E_i}$ est beaucoup plus facile si on dispose des valuations optimales des VCSP induits enracinés aux fils de E_i. Cela permet d'éviter un grand nombre de redondances avec l'enregistrement et l'exploitation des goods valués.
L'algorithme BTD-val a 5 paramètres : $\mathcal{A}$ l'affectation courante, E_i le cluster courant, V_{E_i} les variables du cluster E_i à instancier, ub_{E_i} le majorant actuel du sous-problème enraciné en E_i et lb_{E_i} son minorant actuel. L'appel initial BTD-val($\emptyset, E_1, V_{E_1}, \bot, \top$) permet de calculer l'optimum. Durant la recherche, BTD-val choisit une variable non instanciée du cluster courant E_i (ligne 14), s'il en reste (ligne 1), et lui affecte une valeur v (ligne 17). Ensuite, il met à jour le minorant qui est la valuation locale de l'affectation courante sur le sous-problème $\mathcal{P}_{\mathcal{A},Pere(E_i)/E_i}$ (lignes 19-24). Si le nouveau minorant atteint ou dépasse la valuation de la meilleure solution actuelle de $\mathcal{P}_{\mathcal{A},Pere(E_i)/E_i}$ (ligne 25), BTD-val choisit une nouvelle valeur pour x. Sinon, un appel récursif est effectué : BTD-val($\mathcal{A} \cup \{x \leftarrow v\}, E_i, V_{E_i} \backslash \{x\}, lb_{E_i} \oplus lb_v, ub_{E_i}$) (ligne 25) qui va poursuivre l'instanciation des variables de E_i. Quand, toutes les valeurs de x ont été essayées (ligne 16), un backtrack est opéré sur la variable précédente dans l'ordre d'affectation. Quand V_{E_i} devient vide (ligne 1), toutes les variables ont été affectées, alors BTD-val choisit un fils E_j de E_i (ligne 5). Si $\mathcal{A}[E_i \cap E_j]$ est un good valué alors BTD-val utilise la valuation de la solution optimale du problème $\mathcal{P}_{\mathcal{A},E_i/E_j}$, associée au good valué, qu'il rajoute directement au minorant actuel du sous-problème enraciné en E_i lb (ligne 7). Si $\mathcal{A}[E_i \cap E_j]$ n'est pas un good valué alors BTD-val résout récursivement le sous-problème enraciné en E_j (ligne 9), puis enregistre le good valué $\mathcal{A}[E_i \cap E_j]$ associé à la valuation optimale de ce problème (ligne 11). Cette valuation est rajoutée au minorant lb (ligne 10) et la résolution, poursuivie par le choix d'un autre fils de E_i

Algorithme 10 : LC-BTD-val$^+(\mathcal{A}, E_i, V_{E_i}, lb_{E_i}, ub)$

```
1  if V_{E_i} = ∅ then
2      F ← Fils(E_i)
3      lb ← lb_{E_i}
4      while F ≠ ∅ and lb ≺ ub do
5          Choisir E_j ∈ F
6          F ← F\{E_j}
7          if (A[E_j ∩ E_i], lb_{E_j}, opt) est un good valué then lb ← lb ⊕ lb_{E_j}
8          else
9              ub' ← ub − lb_{E_i} + lb_{E_j}
10             lb'_{E_j} ← LC-BTD-val+(A, E_j, V_{E_j}, lb_{E_j}, ub')
11             LB_{E_j} ← lb'_{E_j} + ΔC(E_j)
12             opt' ← (lb'_{E_j} ≺ ub')
13             Enregistrer (A[E_j ∩ E_i], lb'_{E_j}, opt')
14     return lb
15 else
16     Choisir x ∈ V_{E_i}
17     d_x ← D_x
18     while d_x ≠ ∅ and lb_{E_i} ≺ ub do
19         Choisir v ∈ d_x
20         d_x ← d_x\{v}
21         lb_v ←LC(A ∪ {x ← v}, E_i)
22         if lb_{E_i} ⊕ lb_v ≺ ub then
           ub ← min(ub,LC-BTD-val+(A ∪ {x ← v}, E_i, V_{E_i}\{x}, lb_{E_i} ⊕ lb_v, ub))
23     return ub
```

(lignes 4-5). Si tous les fils de E_i ont été considérés alors lb, la valuation optimale du problème enraciné en E_i, est retournée (ligne 12).

Théorème 2.4.3 *La complexité en temps de BTD-val est en $O(n.m.d^{w+1})$ tandis que sa complexité spatiale est en $O(nsd^s)$, avec s la taille maximale des intersections entre clusters de la décomposition arborescente.*

Nous retrouvons les faiblesses de BTD classique dans cette extension aux VCSP. L'ordre d'affectation des variables est statique et la complexité spatiale est exponentielle en la taille maximale des intersections entre clusters. Mais également, une autre faiblesse majeure se situe au niveau de la résolution optimale des différents sous-problèmes. Cela peut être totalement inutile si la valuation optimale d'un sous-problème combinée à la valuation locale de l'affectation courante dépasse le majorant courant. En outre, le coût de cette opération est considérable dans la mesure où BTD-val commence la résolution du sous-problème avec un majorant égale à $\top$. Ceci entraîne une recherche de type branch and bound de piètre qualité avec un élagage assez faible au départ.

Plusieurs améliorations ont été définies pour BTD-val. En effet, il est possible d'utiliser les consistances locales définies dans la section précédente, pour améliorer le branch and bound à l'intérieur des clusters de la décomposition. Les algorithmes 8 et 9 présentent l'extension FC-BTD-val qui opère à l'intérieur des clusters de la même manière que FC-val, avec ainsi un meilleur minorant. LC-BTD-val est un cadre pour ce type d'extensions. En effet, à l'intérieur de chaque cluster, la consistance locale LC est utilisée pour calculer un meilleur minorant dans la technique du branch and bound. L'algorithme LC-BTD-val$^+$ présenté ici ne procède pas exac-

tement de la même manière que BTD-val. Dans le BTD-val classique, au début de la résolution d'un sous-problème enraciné en un cluster, le majorant est de piètre qualité. Il est égal à $\top$. Néanmoins, il permet de résoudre optimalement le sous-problème. Mais cette résolution peut s'avèrer totalement inutile, si cela entraîne un dépassement du coût de la meilleure solution connue. Dans [dGSV06], les auteurs proposent de partir avec un majorant qui est la différence entre la valuation de la meilleure solution connue et la valuation locale de l'affectation courante. L'extension de l'affectation courante dans le sous-problème est stoppée dès qu'on découvre que sa valuation va dépasser celle de la meilleure solution. LC-BTD-val^+ (Algorithme 10) procède à un élagage plus important durant la recherche à l'intérieur des sous-problèmes. Cependant, ce nouveau majorant, dans le cas où, il est supérieur à la valuation optimale du sous-problème, empêche la résolution optimale de ce dernier. Dans ce cas, on ne dispose pas de la valuation optimale du sous-problème, mais d'un minorant donné par ce majorant de départ. De ce fait, il est possible que ce sous-problème soit résolu à nouveau contrairement au cas où on dispose de l'optimum. La définition de good valué structurel est donc légèrement différente. Dans ce cas, un good valué est composé d'une affectation $\mathcal{A}[E_i \cap E_j]$ sur une intersection entre un cluster E_i et un de ses fils E_j et d'une valuation qui est soit un minorant de la valuation optimale du problème enraciné en E_j (si $opt = 1$), soit cette valuation optimale (si $opt = 0$). Ainsi, LC-BTD-val^+ a une moins bonne complexité théorique en $O(\alpha^*_{\mathcal{P}}(exp(w+1)))$ qui reste bornée par $O(exp(h))$, avec h le nombre maximum de variables dans une branche de la décomposition arborescente. Mais, il obtient en pratique des résultats généralement meilleurs. Cependant, le défaut de maîtrise de la complexité spatiale demeure.

Bucket Elimination- BB (BE-BB [LD03]). La méthode Bucket Elimination-Branch and Bound (BE-BB) est une adaptation de VES aux WCSP. C'est une hybridation de BE avec BB. La méthode BE élimine à chaque étape une variable suivant un ordre statique calculé au départ. L'élimination de cette variable donne l'extension optimum sur cette variable pour tout tuple sur les variables restantes du problème. Cela rajoute une nouvelle contrainte qui résume son influence sur les variables restantes. Cependant, pour limiter l'espace mémoire requis, un facteur a est défini pour limiter l'arité des contraintes ajoutées. En cas de dépassement, un BB est effectué au niveau de la variable à la place de l'élimination. Cette technique limite l'espace à $O(nexp(a))$, et le temps revient à $O(exp(a+z))$, avec z le nombre de variables dont la largeur dépasse a. Cette méthode obtient de meilleurs résultats que BB et BE.

Nous retrouvons également les faiblesses de VES à savoir une résolution sur un ensemble de buckets dont la qualité peut être très moyenne au point de vue de l'efficacité pratique et mais aussi théorique avec une valeur de z qui peut être assez grande. Cela est principalement causé par la séparation du choix entre la valeur de a et l'ordre d'élimination.

Pseudo-Tree Search ([LMS02]). C'est une adaptation de PTS [LMS02] au cadre VCSP. Elle utilise une technique de branch and bound avec exploitation des indépendances entre parties du problème données par le pseudo-tree. Dès qu'une variable est intanciée, les sous-arbres enracinés sur ses fils forment des sous-problèmes indépendants. Ils sont alors résolus séparément et leurs valuations optimales sont combinées pour déterminer la valuation optimale du sous-problème enraciné en la variable courante. Dans la mesure où plusieurs sous-problèmes seront résolus, il est important de se doter d'un majorant de qualité en tenant compte du travail déjà effectué sur des sous-problèmes déjà résolus et du majorant du sous-problème contenant le problème courant. En effet, avec des majorants médiocres, la résolution peut être totalement inefficace. Une combinaison de cette méthode avec RDS avec des sous-problèmes donnés par les sous-arbres du pseudo-tree, est simple et conduit d'ailleurs à des résultats meilleurs en général que ceux de SRDS sur les instances aléatoires. En effet cette décomposition engendre de véritables indépendances alors que dans RDS et SRDS les décompositions en sous-problèmes sont totalement artificielles.

La complexité en temps reste exponentielle en h, la hauteur du pseudo-arbre. Il demeure donc un grand de redondances qui limitent l'efficacité pratique.

AndOrSearchGraph ([DM07]). Les méthodes AND/OR Search Tree et Graph [DM07] possèdent des adaptations dans le cadre WCSP. Une solution de $\mathcal{P}$ devient un sous-arbre de l'arbre de recherche AND/OR contenant la racine. En outre s'il contient un nœud OR alors il contient au moins un de ses fils et s'il contient un nœud AND alors il contient tous ses fils. La valuation de l'affectation complète induite par le sous-arbre, est associée à chacune de ses feuilles. Une recherche en profondeur d'abord permet de calculer les valuations des affectation complètes induites par les feuilles de l'arbre, mais également de déterminer une solution optimale. La complexité en temps du AndOrSearch Tree est en $O(exp(h))$ avec h la hauteur du pseudo-arbre. A l'image du cadre CSP [DM07], deux nœuds avec le même ensemble de séparateurs-parents sont racines de deux sous-problèmes identiques si les affectations sur les variables de l'ensemble séparateurs-parents sont identiques. La fusion de ces sous-problèmes conduit à un graphe de contexte minimal de recherche AND/OR. La complexité en temps du AndOrSearch Graph est en $O(exp(w^*))$, w^* étant la largeur induite de l'arbre. La complexité spatiale de la méthode est également en $O(n.s.exp(s))$, avec s la taille du plus grand ensemble séparateurs-parents. Le AndOrSearch Graph est très voisin de BTD-val.

Comme, nous l'avons déjà observé dans le cadre CSP, la méthode du AND/OR Search Graph étendue aux WCSP obtient parmi les meilleurs résultats pratiques. Malheureusement, l'absence de stratégies efficaces pour réaliser un bon compromis espace/temps empêche la résolution de problèmes assez difficiles qui demeurent inaccessibles. En outre, le coût de résolution des différents sous-problèmes est considérable à cause du majorant de départ fixé à $\top$.

2.4.2.5 Heuristiques

Les heuristiques définies dans le cadre CSP sont généralement utilisées dans le cadre VCSP. Le principe recherché est toujours le même (first-fail principle) : faire les choix les plus contraints d'abord pour essayer de réduire la taille de l'arbre de recherche. Pour cela, il faut faire croître le minorant le plus rapidement possible. Le but est d'atteindre le majorant assez vite et d'élaguer le maximum de branches possibles. L'observation qui prévaut dans le cadre CSP sur la plus grande efficacité des heuristiques dynamiques par rapport à celles statiques, reste également valable dans le formalisme VCSP. L'explication vient de l'utilisation d'algorithmes de filtrage et de consistances locales valuées qui font évoluer le problème à travers la taille des domaines et le coût des tuples dans les contraintes.

2.5 Conclusion

Nous avons fait un état de l'art sur les méthodes de décomposition de graphes et hypergraphes, les cadres CSP et VCSP. Le formalisme CSP permet de représenter facilement un éventail très large de problèmes. Beaucoup de méthodes de résolution de CSP ont été développées. Les méthodes énumératives sont relativement efficaces en pratique, mais leurs bornes de complexité théorique en temps, exponentielles en n (le nombre de variables du CSP), sont très médiocres. Les méthodes structurelles proposent de meilleures garanties théoriques. Elles utilisent pour la plupart un recouvrement acyclique du problème. La résolution des différentes parties ainsi construites, donne un CSP acyclique qui peut être résolu en un temps polynomial. Les excellentes bornes de complexités de ces méthodes sont obtenues généralement au détriment de l'efficacité en pratique. L'espace mémoire requis est souvent inaccessible et rend inopérant la majeure partie de ces techniques. Cependant, certaines d'entre elles, comme BTD, associent ces garanties théoriques avec la souplesse de l'énumération. Elles arrivent à résoudre des problèmes inaccessibles aux meilleures méthodes énumératives telles FC et MAC, moyennant une bonne décomposition du problème en terme de taille (réduite) des différentes parties

définies. On retrouve exactement le même cas de figure au niveau des VCSP. On peut noter également, que dans les deux cas, la plupart des méthodes définies sont très proches voire même équivalentes. D'ailleurs, nous présenterons dans la suite un cadre qui permet d'en capturer une grande partie. Dans le prochain chapitre, nous allons aborder une étude sur ces méthodes structurelles. Nous avons choisi de nous focaliser sur BTD et BTD-val qui sont parmi les meilleures méthodes, que ce soit d'un point de vue théorique ou pratique. Leurs bons résultats ont été obtenus sans la recherche de bonnes heuristiques pour le calcul et l'exploitation des décompositions en pratique. Or, les heuristiques sont un point crucial dans l'efficacité d'algorithmes tels que FC et MAC. Une heuristique de choix de variables statique a souvent des résultats désastreux par rapport aux heuristiques dynamiques qui tiennent compte de l'évolution du problème durant la recherche. Les résultats de BTD et BTD-val ont été obtenus avec un choix de variables dynamiques au sein des clusters. Nous voulons aller encore plus loin. Dans un premier temps, nous allons mener une étude sur les méthodes de calcul de décompositions arborescentes et leur efficacité du point de vue pratique (chronométrique). Dans un second temps, nous allons définir des stratégies efficaces d'exploitation des décompositions arborescentes.

Chapitre 3

Calcul de décompositions arborescentes

3.1 Introduction

Le calcul d'une décomposition arborescente de qualité revêt une haute importance dans l'efficacité des méthodes de résolution de (V)CSP reposant sur cette structure. Il a fait l'objet de nombreuses études. L'objectif de ces travaux était purement graphique. La qualité d'une décomposition est appréciée par rapport à la taille maximale de ses clusters. Une décompostion de largeur, w, très petite par rapport au nombre de variables du problème, n ($w \ll n$) engendre un gain théorique exceptionnel par rapport à la borne de complexité en temps des méthodes énumératives. Il existe des méthodes de résolution basées sur de telles décompositions et dont la complexité en temps est en $O(exp(w+1))$. Cette borne est largement meilleure que celle en $O(exp(n))$ des méthodes énumératives de type BT. Malheureusement, le calcul d'une décomposition optimale est un problème NP-difficile. Ainsi, dans la pratique, on se contente de décompositions non optimales, mais le plus proche de l'optimum possible. Pour cela, on recourt généralement à la notion de triangulation de graphes qui est quasiment équivalente à celle de décomposition arborescente. Une triangulation optimale d'un graphe donne, à travers ses cliques maximales, une décomposition optimale. Les graphes triangulés ont été intensivement étudiés. Ici, nous allons faire une étude sur les différentes approches de triangulations définies avec le but d'atteindre l'optimum ou de s'en approcher le plus. Nous allons également, définir de nouvelles méthodes basées sur les séparateurs du graphe et des propriétés sémantiques du (V)CSP. Une comparaison pratique sera présentée sur des graphes structurés, issus de domaines très variés. La bibliothèque Treewidth-Lib de Bodlaender (http ://people.cs.uu.nl/hansb/treewidthlib/index.php), consacrée aux méthodes de calcul de treewidth exactes ou approchées, nous a permis de disposer d'une masse incroyable d'informations. Ces informations permettent d'observer le comportement des triangulations sur les graphes structurés. Nous nous focalisons sur ces derniers parce que cette thèse porte sur les méthodes structurelles dont le domaine de définition est la classe des problèmes structurés. On dit d'un problème qu'il est structuré, si la largeur de son graphe de contraintes est bien inférieure au nombre de ses variables. De même, un graphe est structuré si sa treewidth est bien plus petite que le nombre de ses sommets. Certes, la largeur des décompositions constitue un critère très important dans l'efficacité des méthodes structurelles, mais il existe d'autres critères incontournables pour une bonne résolution en pratique. Notre but étant d'améliorer ces techniques au niveau chronométrique, une seconde étude est présentée. L'élément déterminant à ce niveau, est le temps de résolution. Cette étude, qui a déjà fait l'objet de publications ([JNT05a, JNT05b]), permet de constater que la largeur n'est pas le seul critère pertinent dans l'évaluation de la qualité d'une décomposition.

Dans la prochaine section de ce chapitre, nous présentons la classe des graphes triangulés. La section 3.3 est dédiée aux différentes approches de triangulation existantes. Ensuite, une étude expérimentale révèle la pertinence de ces techniques.

3.2 La classe des graphes triangulés

Dans toute cette section, $G = (X, C)$ est un graphe.

Définition 3.2.1 *La longueur d'une chaîne est le nombre d'arêtes qui relient deux sommets.*

Définition 3.2.2 *G est triangulé si tout cycle de longueur supérieure strictement à 3 possède une corde c'est-à-dire une arête qui joint deux sommets non consécutifs.*

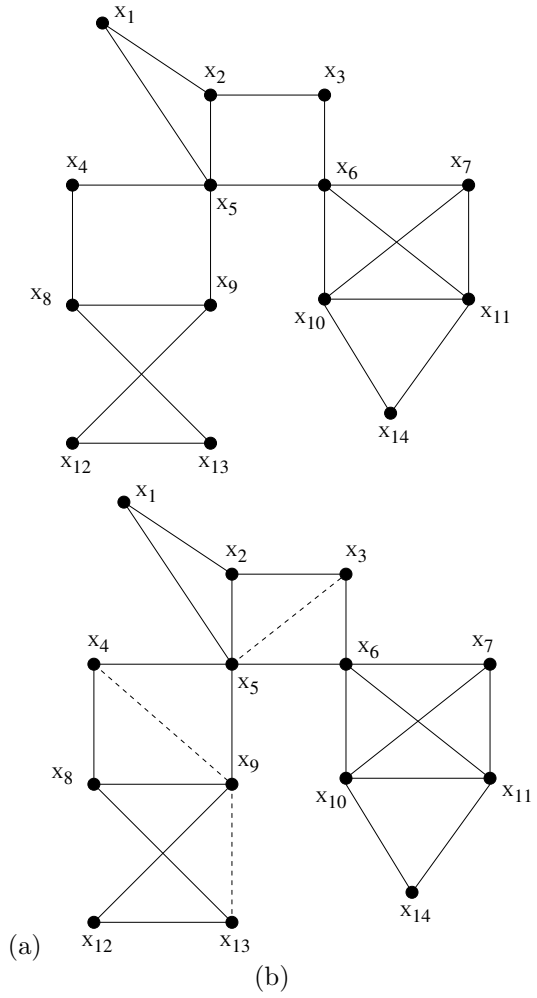

Fig. 3.1 – (a) Un graphe non triangulé, (b) une triangulation du graphe.

La figure 3.1(a) présente un graphe non triangulé car (x_2, x_3, x_6, x_5) est un cycle de longueur 4 sans corde. La figure (b) représente une transformation du premier graphe dans laquelle trois cordes ont été ajoutées pour éliminer les trois cycles de longueur 4 sans corde. De ce fait, ce nouveau graphe est triangulé.

Les graphes triangulés font partie de la classe des graphes parfaits. Cette notion de graphe parfait fait référence à des propriétés structurelles qui sont vérifiées par ces graphes. Une clique de G est un ensemble de sommets de G qui sont tous adjacents, deux à deux. On note $\omega(G)$, la taille de la plus grande clique maximale dans G. Le problème de coloration d'un graphe G consiste à déterminer le nombre minimum de couleurs suffisantes pour colorier G de telle sorte que deux sommets adjacents aient des couleurs différentes. Le nombre chromatique, noté $\chi(G)$, est le plus petit nombre de couleurs suffisantes pour colorier G. On a $\omega(G) \leq \chi(G)$.

Définition 3.2.3 *Un graphe est parfait si* $\omega(G[V]) = \chi(G[V]), \forall V \subset X$.

Théorème 3.2.1 *[Ber60, HS58] Les graphes triangulés sont parfaits.*

On déduit de ce théorème l'existence dans la classe des graphes triangulés, d'algorithmes polynomiaux pour la résolution de problèmes NP-complets ou NP-difficiles dans le cas général tels que : la recherche de la clique maximale, le recouvrement par un ensemble de cliques, la coloration des sommets du graphe, etc.
Du fait de leurs propriétés exceptionnelles dans la théorie des graphes, les graphes triangulés ont naturellement, fait l'objet d'un grand nombre d'études. Dans [Gol80], on trouve une présentation assez complète de cette classe et de ses propriétés.
Nous allons tout de même rappeler quelques définitions et résultats importants la concernant. De nombreuses caractérisations ont été proposées pour les graphes triangulés. Tout d'abord, il est possible de les caractériser par la notion de sommet simplicial.

Définition 3.2.4 *Un sommet x d'un graphe G est simplicial si $G[N(x)]$ est un graphe complet ($N(x)$ est une clique).*
Un sommet x d'un graphe G est quasi-simplicial s'il existe $y \in N(x)$ tel que $G[N(x)\backslash\{y\}]$ est un graphe complet.

Dirac [Dir61] et plus tard Lekkerkerker et Boland [LB62] ont prouvé qu'un graphe triangulé possède au moins un sommet simplicial. En utilisant cette propriété, Fulkerson et Gross [FG65] ont écrit un algorithme pour la reconnaissance des graphes triangulés. Il s'agit de repérer un sommet simplicial de l'éliminer du graphe, et répéter cette opération jusqu'à ce qu'il n'y ait plus aucun sommet auquel cas le graphe est triangulé, ou qu'il n'y ait plus de sommet simplicial ce qui veut dire que le graphe n'est pas triangulé.
L'ordre dans lequel les sommets du graphe sont éliminés dans cet algorithme constitue un schéma parfait d'élimination.

Définition 3.2.5 *Soit $\sigma = [x_1, \ldots, x_n]$ un ordre sur les sommets du graphe G. On appelle voisinage ultérieur d'un sommet x_i l'ensemble $N^+(x_i) = \{x_j \in N(x_i) | j > i\}$.*

Définition 3.2.6 *Un ordre $\sigma = [x_1, \ldots, x_n]$ sur les sommets du graphe G est un schéma parfait d'élimination si chaque x_i est simplicial pour le sous-graphe $G[N^+(x_i)]$ c'est-à-dire $G[N^+(x_i)]$ est complet.*

Le résultat suivant donne une équivalence entre les graphes triangulés et les schémas parfaits d'élimination.

Théorème 3.2.2 *[FG65] Un graphe G est triangulé ssi G possède un schéma parfait d'élimination. De plus, tout sommet simplicial peut constituer le début d'un schéma parfait.*

Prouver donc qu'un graphe est triangulé revient à trouver un ordre sur ses sommets qui constitue un schéma parfait d'élimination.
Une autre caractérisation se base sur les sommets LB-simpliciaux.

Définition 3.2.7 *[Ber98] Un sommet x d'un graphe G est LB-simplicial si pour toute composante connexe CC induite par le voisinage fermé $N[x]$, $G[N(CC)]$ est un graphe complet.*

Théorème 3.2.3 *[LB62] Un graphe G est triangulé ssi tous ses sommets sont LB-simpliciaux.*

Ce résultat admet une formulation différente.

Théorème 3.2.4 *[LB62] : Un graphe G est triangulé ssi on peut prendre n'importe quel sommet, vérifier qu'il est LB-simplicial, le retirer du graphe et recommencer jusqu'à ce qu'il ne reste plus qu'un sommet.*

Cette caractérisation est basée sur les séparateurs minimaux du graphe. Un graphe est triangulé ssi, pour tout sommet x, tous les séparateurs minimaux inclus dans $N(x)$ sont des cliques. Berry ([Ber98]) en déduit que dans un graphe triangulé tout séparateur minimal est dans le voisinage d'un sommet.

Grâce à ces différentes caractérisations, il existe des algorithmes linéaires en temps pour la reconnaissance des graphes triangulés ([Lue74, RTL76]).
Cependant, il existe un nombre assez restreint de graphes vérifiant les conditions très fortes pour appartenir à cette classe. Mais pour profiter des propriétés exceptionnelles de ces graphes, il est possible de procéder à la triangulation d'un graphe non triangulé au départ. Trianguler un graphe consiste à rajouter des arêtes de sorte à construire un graphe (surgraphe) triangulé.

Définition 3.2.8 *Une triangulation de G est un ensemble d'arêtes T tel que $G' = (X, C \cup T)$ est triangulé.*

Par abus de langage, nous appelerons également G' une triangulation de G dans la suite. Il est possible de calculer des triangulations différentes d'un même graphe en rajoutant des ensembles différents d'arêtes. Cependant, toutes les triangulations n'ont pas la même qualité ou le même intérêt. Cette notion de qualité va dépendre fortement de l'utilisation ultérieure de cette triangulation. Les premiers objectifs furent de rajouter le moins d'arêtes possible ou de réduire au minimum la taille maximum des cliques maximales de la triangulation. Ces deux problèmes minimum fill-in (ajout d'un nombre minimum d'arêtes) et treewidth (triangulation avec une taille maximale de cliques maximales minimum) sont NP-difficiles ([ACP87]). Néanmoins, il existe des algorithmes linéaires de calcul de triangulations qui, évidemment, ne vérifient pas nécessairement ces propriétés [Kja90a].
En se basant sur le théorème 3.2.2, calculer une triangulation consiste à prendre un ordre sur les sommets du graphe et le transformer en schéma parfait d'élimination en complétant les voisinages ultérieurs de tous les sommets du graphe. De même, avec le théorème 3.2.3, une triangulation revient à compléter tous les séparateurs minimaux du graphe qui sont dans le voisinage des sommets. L'ordre dans lequel les séparateurs sont complétés peut modifier la triangulation obtenue. Dans tous les cas, la qualité de la triangulation suivant les objectifs à atteindre va dépendre de l'ordre choisi.

Les graphes triangulés sont intéressants dans notre étude car il existe un lien étroit entre la notion de décomposition arborescente de graphe et celle de triangulation. Le moyen le plus simple pour calculer une décomposition arborescente d'un graphe consiste à calculer une triangulation de ce graphe.

Théorème 3.2.5 *[RS86] : Soient G un graphe et $k \in \mathbb{N}^+$. La treewidth de G est au plus $k-1$ ssi G admet une triangulation H dont la taille maximale des cliques est bornée par k.*

En outre, la treewidth d'un graphe triangulé est égale à la taille maximum de ses cliques moins un et aussi tout graphe triangulé admet une décomposition arborescente dont les clusters sont ses cliques maximales. Dès lors, pour un graphe quelconque G, le calcul d'une triangulation induit par ce procédé, une décomposition arborescente de G. La qualité de cette décomposition

est assujettie à celle de la triangulation.
Il faut noter également qu'un graphe triangulé G a au plus n cliques maximales ([FG65]) qui peuvent être calculées en un temps linéaire ([Gav72]), où n est le nombre de sommets de G. La qualité des triangulations, et donc des décompositions arborescentes induites, a été étudiée surtout du point de vue des critères graphiques : principalement recherche de l'optimum ou d'approximations de l'optimum avec des garanties plus ou moins fortes. Différentes méthodes ont été proposées avec des succès variés. Nous allons présenter dans la section suivante les différentes classes d'algorithmes proposés pour le problème treewidth.

3.3 Les différentes classes de méthodes de triangulation

3.3.1 Les algorithmes exacts

Les algorithmes exacts calculent une triangulation optimale. Ce problème étant NP-difficile, il n'existe pas d'algorithme connu pour le résoudre en un temps polynomial. Les méthodes proposées ont donc une complexité exponentielle. En outre, elles ont souvent un intérêt pratique limité car pour la plupart, elles ne permettent pas de calculer une triangulation dans un temps raisonnable.
Dans [SG97], les auteurs proposent le QuickTree, un algorithme qui selon eux, est le premier à pouvoir calculer une triangulation optimale dans un temps raisonnable. Il est basé sur la notion de séparateurs minimaux. Il se dote d'une valeur k qui est un minorant de la treewidth du graphe et calcule l'ensemble des séparateurs minimaux $S(k)$ dont la taille est au plus k, dit ensemble k-régulier. A chaque séparateur est associé un ensemble de fragments qui sont les unions entre les composantes connexes induites par le séparateur et le séparateur complété (transformé en clique). Si pour un séparateur de $S(k)$, les fragments associés sont minimalement k-triangulés (cliques maximales de taille au plus k) alors la composition de ces derniers donne une triangulation minimale du graphe de départ de largeur au plus k. Sinon, les fragments sont ordonnés de manière croissante suivant leur taille. Pour chaque fragment non triangulé, QuickTree calcule si possible, un ensemble k-régulier de séparateurs minimaux de cet ensemble tel que chaque fragment est soit une clique, soit minimalement k-triangulé. La composition de ces différents fragments triangulés donne une triangulation minimale de treewidth au plus k et de ce fait une triangulation optimale. Si un ensemble vérifiant ces conditions n'existe pas, cela veut dire que la treewidth de G est supérieure à k.
Les expérimentations sont d'un certain intérêt. Les auteurs considèrent des graphes structurés en générant des arbres de cliques partiels avec des clusters de taille égale et bornée par 11. Le nombre de sommets est compris entre 50 et 100 et le pourcentage d'arêtes retirées entre 30 et 50 pour cent. Le temps pour calculer l'ensemble des séparateurs minimaux est assez élevé ($O(Nb_{sepmin}.n^5)$, avec Nb_{sepmin} le nombre de séparateurs minimaux) comparé au temps nécessaire pour calculer une triangulation optimale à partir de cet ensemble de séparateurs ($O(Nb_{sepmin}.n^3.Nb_{cliqmax})$, avec $Nb_{cliqmax}$ le nombre de cliques maximales). Le nombre de cliques maximales est bornée de manière abusive par $2^{Nb_{sepmin}}$ mais en pratique, il est largement plus petit. A partir de 100 sommets et pour une treewidth bornée par 10 et un pourcentage d'arêtes retirées supérieure ou égale à 30 pour cent, calculer les séparateurs devient très long.

Dans [GD04], QuickTree est comparé à la méthode QuickBB. Cette dernière calcule la treewidth par une technique de branch and bound sur l'ensemble des ordres sur les sommets du graphe. On commence par calculer un majorant et un minorant grâce à des techniques heuristiques. Les auteurs proposent une heuristique de calcul de minorant appelée minor-min-width. Le minorant de départ est $lb = 0$. A chaque étape, l'heuristique contracte l'arête entre un sommet de degré minimum v dans le graphe courant et un sommet de degré minimum dans son voisinage. Ensuite, elle met à jour lb qui prend la valeur maximum entre sa valeur actuelle et le degré de v avant la contraction. Ce traitement est poursuivi jusqu'à ce qu'il n'y ait plus aucun sommet dans le graphe. Des expérimentations présentées dans l'article montrent que minor-min-

width obtient de bons résultats.
Si, le majorant et le minorant de départ sont égaux alors la treewidth est égale à cette valeur. Sinon, QuickBB réalise une recherche par branch and bound avec une construction incrémentale de schémas parfaits d'élimination sur les sommets, en maintenant un minorant sur la largeur de la triangulation en cours de construction. Le majorant au début du branch and bound est donné par le majorant fourni par l'heuristique choisie. A chaque étape, un sommet est éliminé. Cela engendre la complétion de son voisinage dans le graphe courant et une mise à jour du minorant. Ce dernier aura pour valeur le maximum entre son ancienne valeur et le degré du sommet éliminé. L'heuristique minor-min-width calcule un minorant de la largeur du graphe courant qui est agrégée au minorant courant. Si le minorant dépasse le majorant, un backtrack est réalisé pour choisir un autre sommet. Sinon, QuickBB choisit un nouveau sommet dans le graphe courant. Si ce graphe est vide, le majorant est mis à jour et sa nouvelle valeur est la largeur de la triangulation construite. La recherche continue jusqu'à ce que la totalité de l'espace des ordres possibles soit visitée. La treewidth est donc la dernière valeur du majorant. La méthode peut être améliorée grâce à des techniques de réduction du graphe. En effet, certains sommets du graphe courant peuvent être éliminés sans affecter la procédure de branch and bound. Les règles du sommet simplicial ou quasi-simplical [BKvdEvdG01], permettent d'éliminer tout sommet (quasi-)simplicial dans le graphe courant. Dans le cas d'un sommet simplicial, la valeur du minorant devient le maximum entre son ancienne valeur et le degré de ce sommet parce que la largeur de cet ordre sera au moins égale au degré du sommet. On peut de même éliminer un sommet quasi-simplicial dont le degré est inférieur à la valeur du minorant car cela n'augmente pas la largeur de la triangulation en cours de construction. En outre, le choix du prochain sommet peut être restreint à ceux qui ne sont pas adjacents au sommet courant. D'autres résultats théoriques permettent aussi de rajouter directement une arête entre deux sommets non encore considérés, de faire de la propagation et un élagage plus important. Grâce à ces techniques, QuickBB s'avère assez efficace, même si sa complexité est exponentielle ($O(n^{n-w})$, w la treewidth du graphe). La comparaison réalisée entre QuickBB et QuickTree donne des résulats nettement en la faveur du premier. Cela est vérifié sur des graphes structurés (treewidth limitée) mais surtout sur ceux sans structure particulière grâce à une complexité qui diminue avec l'augmentation de la treewidth. Or, sur ces graphes non structurés, le nombre de séparateurs minimaux peut être exponentiel et rendre donc QuickTree inopérationnel. QuickBB part également avec un avantage non négligeable grâce au majorant calculé au départ, avec l'heuristique minfill. Nous allons voir dans la suite que minfill est d'une très grande efficacité dans le calcul d'un majorant de la treewidth de qualité.

Dans [BT02], les auteurs donnent une classe de graphes pour laquelle le problème treewidth est polynomial. Ils résolvent la conjecture ESA'93 qui disait que les problèmes treewidth et minimum fill-in sont polynomiaux pour les graphes ayant un nombre polynomial de séparateurs minimaux. Ils proposent dans ce cadre un algorithme de triangulation basé sur la notion de cliques maximales potentielles.

Définition 3.3.1 *Un ensemble de sommets S de G est une clique maximale potentielle si S est une clique maximale dans une triangulation minimale de G.*

Ils avaient déjà montré dans [BT01b] que les problèmes treewidth et minimum fill-in étaient polynomiaux pour les graphes dont l'ensemble des cliques maximales potentielles pouvait être énuméré en un temps polynomial. De même, il est possible d'énumérer polynomialement l'ensembles des cliques maximales potentielles si on dispose de l'ensemble des séparateurs minimaux du graphe. Cela donne le résultat suivant.

Théorème 3.3.1 *La treewidth et le minimum fill-in d'un graphe peuvent être calculés en un temps polynomial en la taille du graphe et le nombre de ses séparateurs minimaux. La complexité de l'algorithme est en $O(n^3 Nb^3_{sepmin} + n^2 m Nb^2_{sepmin})$.*

Ils utilisent la programmation dynamique à l'instar de [SG97] pour résoudre ces problèmes à partir de l'ensemble des séparateurs minimaux et des cliques maximales potentielles.
Pour le cas général, [FKT04] ont proposés un algorithme en $O(n^4.1,9601^n)$. Cette complexité fait tomber la barrière constituée jusque-là par $O(2^n.poly(n))$. Cette méthode est basée sur une légère modification de l'algorithme proposée dans [BT02] et une analyse combinatoire plus fine du nombre de séparateurs minimaux et des cliques potentielles. Cet algorithme n'a pas été implémenté car les attentes au niveau de ces techniques n'intègrent pas l'utilité pratique.

3.3.2 Les algorithmes d'approximation avec garanties

Ces algorithmes garantissent une approximation de l'optimum à un facteur près : la largeur de la triangulation calculée est inférieure à ce facteur fois l'optimum. Pour une approximation à un facteur constant près, il n'existe à ce jour que des algorithmes de complexité exponentielle. Une approximation en un temps polynomial à une constante près reste un problème ouvert. Robertson et Seymour ont défini une méthode d'approximation basée sur la notion de branchwidth ([RS95]). [Ree92] a prouvé que la branchwidth donne une approximation supérieure d'au plus 4 fois à la treewidth, avec une complexité en $O(w^2 3^{3w} n^2)$. L'auteur propose, pour sa part, une approximation d'un facteur 5 constant, avec une complexité en $O(w^2 3^{4w} nlogn)$. La technique de Becker et Geiger ([BG96]) utilise la programmation linéaire pour approcher l'optimum à un facteur constant de 3,66. Elle a une complexité en $O(2^{4,66w} n.poly(n))$, $poly(n)$ étant la complexité de l'algorithme de programmation lineaire. Amir, dans [Ami02], donnent deux algorithmes qui améliorent les complexités en temps des algorithmes de Robertson et Seymour en $O(2^{4,38w} n^2.w)$ et de Becker et Geiger en $O(2^{3,6982w} n^3.w^3.log^4 n)$. Il définit pour cela la notion de α-separateur. L'idée est de choisir un séparateur qui induit des composantes connexes contenant au plus un certain pourcentage du nombre de sommets du graphe de départ. Ce pourcentage est défini par la valeur de α. Ensuite, cette opération est réitérée sur les différentes composantes connexes. On s'assure ainsi que les séparateurs que nous choisissons décomposent le graphe de manière équilibrée.

Définition 3.3.2 *Soit G un graphe, V un sous-ensemble de sommets de G et $0 \leq \alpha \leq 1$, un réel.*
Un α-separateur de V dans G est un ensemble de sommets S tel que chaque composante connexe de $G[X - S]$ contient au plus $\alpha|V|$ sommets de V.
Un α-separateur bidirectionnel admet une condition supplémentaire imposant l'existence d'exactement deux ensembles CC_1 et CC_2 séparés par S tel que $CC_1 \cup CC_2 \cup S = X$ et $|CC_i| \leq \alpha|V|$, pour $i = 1, 2$.

La définition de α-separateur bidirectionnel impose l'existence de deux et seulement deux composantes connexes induite par le séparateur et qui contiennent au plus $\alpha.|V|$ sommets de V. Cette restriction est assouplie dans la définition de γ-separateur tridirectionnel. Dans ce cas, nous avons toujours l'obligation d'avoir au moins deux composantes connexes induites, mais il est possible d'en avoir plus cette fois. Le nombre de sommets contenus dans les composantes n'est plus borné par rapport à un ensemble de sommets du graphe, mais par rapport à sa treewidth.

Définition 3.3.3 *Soit G un graphe, V un sous-ensemble de sommets de G et $\gamma \geq 1$, un réel.*
Un γ-separateur tridirectionnel vérifie la condition suivante : $|(CC_i \cap V) \cup S| \leq (1 + \gamma)w$, pour $i = 1, 2, 3$, avec au moins deux CC_i non vides.

Le premier algorithme utilise les separateurs bidirectionnels et une valeur de α égale à $2/3$ alors que le second est basé sur les séparateurs tridirectionnnels avec une valeur de γ de $4/3$. Il propose également un algorithme d'approximation à un facteur constant de 4,5 près, avec une meilleure complexité en temps en $O(2^{3k} n^2 w^{1,5})$, grâce aux separateurs bidirectionnels et $\alpha = 1/2$. Ces séparateurs sont calculés avec des techniques de calcul de flots maximum.

Les approximations exponentielles se montrent incapables de résoudres les graphes avec une treewidth supérieure à 4 [Röh99]. Selon Amir, ces techniques se révèlent inopérantes pour des graphes de plus de 100 sommets avec une treewidth de plus de 10 : elles sont incapables de résoudre ces problèmes en moins de 24h [BG96, SG97]. Généralement les problèmes intéressants se trouvent au-delà de cette limite.

En ce qui concerne les algorithmes d'approximation polynomiaux, le meilleur facteur d'approximation a longtemps été $O(log(n))$ [Klo94, Ba95]. En 2002, Amir, toujours dans le même article [Ami02], propose un algorithme d'approximation avec un facteur $log(w)$ et une complexité en temps de $O(n^3.log^4(nw^5).log(w))$ qui utilise des séparateurs tridirectionnels. Ensuite les auteurs de [BKMT04] ont défini une stratégie, avec les mêmes garanties (facteur de $O(log(w))$), qui consiste à choisir à chaque étape un séparateur minimal dont les composantes connexes contiennent au plus la moitié des sommets du graphe courant. La complexité en temps est moins bonne que celle de l'algorithme d'Amir. Cependant, sa constante cachée (au sens de la notation "grand O") est légèrement supérieure à 675, là où celle de l'algorithme d'Amir dépasse 850, d'après [BKMT04]. De plus, Amir indique dans ses expérimentations (voir [Ami02]) que sur les benchmarks testés, son algorithme prend de 6 minutes à 6 jours, selon l'instance, alors qu'une heuristique naïve comme *min-degré* donne sur les mêmes benchmarks respectivement de 1 seconde à 2 minutes, avec de plus des résultats dont la qualité est significativement meilleure (l'approximation de la largeur d'arborescence est de l'ordre de 50 % inférieure).
Jusqu'à maintenant, cette approche n'a pas d'intérêt pratique dans la mesure où les algorithmes proposés n'offrent que des garanties très faibles à l'ombre de la notation "grand O" tout en demeurant très coûteux au niveau du temps de calcul.

3.3.3 Les algorithmes heuristiques

Ces approches construisent en général un ordre (dynamiquement) et ajoutent des arêtes dans le graphe, de sorte qu'en fin de traitement, l'ordre obtenu soit un ordre d'élimination parfait pour le graphe résultant G'. La complexité de ces méthodes est en général polynomiale (souvent même linéaire) mais elles n'offrent, en contrepartie, aucune garantie d'optimalité. D'après [Kja90a], cette approche est justifiée en pratique. En effet, Kjærulff a observé entre autres, qu'au contraire des résultats attendus, ces heuristiques produisent des triangulations raisonnablement proches de l'optimum. De plus, elles sont en général très faciles à implémenter. Nous présentons ci-dessous les plus célèbres et souvent les plus efficaces, dont la complexité en temps varie de $O(n + m')$ à $O(n(n + m'))$, m' étant le nombre d'arêtes de G'.

- *LBFS* [RTL76]. Elle a pour objet la reconnaissance des graphes triangulés. Elle ordonne les sommets de n à 1 en étiquetant tous les sommets à vide au départ et en choisissant arbitrairement le premier sommet (numéroté n). Elle rajoute n à l'étiquette des sommets voisins du sommet n. Ensuite, elle choisit comme sommet suivant un sommet ayant la plus grande étiquette suivant un ordre lexicographique et rajoute le numéro du sommet aux étiquettes de ses voisins non encore numérotés (en cas d'égalité, LBFS fait un choix arbitraire). Sa complexité est en $O(n + m')$.
- *MCS* [TY84]. Elle consitue un "cousin simplifié de LBFS" ([Ber98]). C'est un algorithme de reconnaissance des graphes triangulés. Elle ordonne les sommets de n à 1 en choisissant arbitrairement le premier sommet (numéroté n) et ensuite comme sommet suivant un sommet ayant le plus grand nombre de voisins déjà numérotés (en cas d'égalité, MCS fait un choix arbitraire). Sa complexité est en $O(n + m')$.
- *mindeg-interieur*. Elle ordonne les sommets de 1 à n, en sélectionnant comme nouveau sommet, le sommet qui possède le plus petit nombre de voisins déjà numérotés. Sa complexité est généralement en $O(n^3)$.
- *mindeg*. Elle ordonne les sommets de 1 à n, en sélectionnant comme nouveau sommet, le sommet qui possède le plus petit nombre de voisins non numérotés. Sa complexité est généralement en $O(n^3)$.

- *minfill.* Elle ordonne les sommets de 1 à n, en sélectionnant comme nouveau sommet, le sommet qui conduira à ajouter un minimum d'arêtes si l'on complète le sous-graphe induit par ses voisins non encore numérotés. Sa complexité est en $O(n(n+m'))$.
- *Sparse fill-in [Bodlaender].* Cette heuristique est une combinaison de minfill et mindeg. Elle a été définie par Bodlaender dans la bibliothèque TreewidthLIB (http ://people.cs.uu.nl/hansb/treewidthlib/index.php).
  Aucune référence à un article dans lequel elle serait présentée n'est fournie. Les sommets sont ordonnés de 1 à n, en sélectionnant comme nouveau sommet, un dont la différence entre le nombre d'arêtes rajoutées si l'on complète le sous-graphe induit par ses voisins non encore numérotés, et son degré dans le graphe courant, est minimum. Sa complexité est en $O(n(n+m'))$.

[Kos99] a proposé une heuristique de triangulation *MSVS* différente de celles que nous venons de présenter dans le sens où elle n'est pas basée sur la construction d'un ordre d'élimination parfait. Elle calcule récursivement une décomposition arborescente en partant d'un cluster contenant tous les sommets du graphe. En utilisant les techniques de calcul flots maximum, les clusters sont récursivement décomposés à l'aide des séparateurs minimum (minimum separating vertex sets). A chaque étape, MSVS choisit un séparateur minimum contenu dans le cluster courant et le complète, puis recommence la même opération sur les différents blocs ainsi construits (un bloc est une réunion d'un séparateur et d'une composante connexe induite par ce séparateur). On obtient au final une triangulation du graphe de départ représentée dans ce cas par une décomposition arborescente.

3.3.4 Les algorithmes minimaux

A l'image des algorithmes heuristiques, les algorithmes minimaux calculent des triangulations à un coût assez faible. Mais contrairement aux heuristiques, ils nous garantissent que le retrait d'une arête rajoutée pour la triangulation d'un graphe donne un nouveau graphe qui n'est pas triangulé. En effet, une triangulation minimale ne contient pas d'arête redondante.

Définition 3.3.4 *Une triangulation minimale de G est un ensemble d'arêtes T tel que $G' = (X, C \cup T)$ est triangulé et pour tout sous-ensemble $T' \subset T$, le graphe $G" = (X, C \cup T')$ n'est pas triangulé.*

Une triangulation peut être minimale sans être optimale. Mais il évident que, si on rajoute des arêtes à une triangulation d'un graphe, on ne peut que s'éloigner de l'optimum. En outre, l'intérêt de ce type d'approche repose sur l'existence d'algorithmes de complexité polynomiale. Ces derniers proposent une approximation de l'optimum sans garanties autres que toutes les arêtes présentes soient nécessaires. Nous présentons ci-dessous, un certain nombre d'algorithmes de triangulation minimale. Pour plus de détails, [Heg06] propose un survey très complet sur ces méthodes.

- *Lex-M* [RTL76]. C'est une extension de LBFS qui calcule des ordres de triangulation minimale. Elle ordonne les sommets de n à 1 en étiquetant tous les sommets à vide au départ et choisissant arbitrairement le premier sommet v (numéroté n). Elle rajoute n à l'étiquette des voisins de v et de tout sommet non numeroté u pour lequel il existe un chemin entre u et v, composé uniquement de sommets non numérotés et dont les étiquettes sont lexicographiquement plus petites que celles de u et v. Ces chemins sont appelés des *fill-paths*. Ensuite, elle choisit comme sommet suivant un sommet v ayant la plus grande étiquette suivant un ordre lexicographique et rajoute le numéro du sommet à l'étiquette des voisins de v et de tout sommet non numeroté u pour lequel il existe un fill-path entre u et v (les égalités sont cassées arbitrairement). Sa complexité est en $O(nm')$.
- *MCS-M* [BBH02, BBHP04] C'est une extension de MCS qui calcule des ordres de triangulation minimale. Elle consitue donc un "cousin simplifié de Lex-M". Elle ordonne les sommets de n à 1 en donnant un poids nul à tous les sommets au départ et choisissant

arbitrairement le premier sommet v (numéroté n). Elle incrémente le poids des voisins de v et de tout sommet non numeroté u pour lequel il existe un chemin entre u et v, composé uniquement de sommets non numérotés dont les poids sont plus petits que ceux de u et v. Ces chemins sont des fill-paths. Ensuite, elle choisit comme sommet suivant un sommet v ayant le plus grand poids et incrémente le poids des voisins de v et de tout sommet non numeroté u pour lequel il existe un fill-path entre u et v (les égalités sont cassées arbitrairement). Sa complexité est en $O(nm')$.

– *LBTriang* [Ber99] Cet algorithme utilise le théorème 3.2.3 pour calculer une triangulation minimale. Il utilise un ordre quelconque sur les sommets et les rend LB-simpliciaux suivant cet ordre. Il peut calculer toutes les triangulations minimales possibles, suivant l'ordre utilisé. Le prochain sommet x est choisi suivant l'ordre puis les séparateurs minimaux inclus dans le voisinage de x sont calculés puis complétés. Compléter les voisinages des composantes connexes induites par les voisinages des sommets du graphe avait déjà été proposée par Fujisawa et Orino [FO74] et présentée dans [Kja90b] avec une complexité en $O(n(m + m''))$, m'' étant le nombre d'arêtes rajoutées. Là où les méthodes classiques complètent tout le voisinage, cet algorithme complète la partie nécessaire avec une complexité en $O(nm')$.

Ces trois algorithmes construisent des sommets (LB-)simpliciaux suivant un certain ordre. Il existe d'autres méthodes qui pour calculer une triangulation minimale partent d'une triangulation quelconque et retirent des arêtes jusqu'à avoir une triangulation minimale. Cette approche était justifiée par le fait que, jusqu'à la définition de LBTriang, LEX-M était le meilleur algorithme de triangulation avec une complexité en $O(nm')$. Mais, on a pu voir que cet algorithme calcule des triangulations parfois très éloignées de l'optimum en rajoutant un grand nombre d'arêtes. De même, elle permet de calculer un nombre restreint d'ordres minimaux. De ce fait, il pouvait être plus intéressant de partir d'une triangulation non minimale de qualité et de retirer les arêtes en redondantes. Mais cela a un coût non négligeable. Ces techniques sont souvent basées sur le résultat suivant.

Théorème 3.3.2 *[RTL76] Soit $G' = (X, C \cup T)$ une triangulation de G. T est une triangulation minimale ssi chaque arête de T est l'unique corde d'un cycle de taille 4.*

Kjaerulff dans [Kja90b] a proposé des méthodes en $O(d^2|T|^2)$ avec d une constante qui exprime la densité du graphe et en $O((d|T|/d')^2)$ avec d' une constante liée à l'ordre d'ajout des arêtes de T. Lors de la triangulation, il est possible de rajouter un ensemble d'arêtes à chaque étape (complétion d'un voisinage ou d'un séparateur). L'auteur ordonne ces ensembles d'arêtes suivant l'étape à laquelle elles ont été rajoutées. Ainsi, il les examine dans l'ordre inverse de leur ajout pour éviter de les reconsidérer plusieurs fois pour déterminer celles qu'il faut supprimer. En 1996, Blair, Heggernes et Telle [BHT01] ont proposé une méthode avec une complexité en $O(|T|(m + |T|))$ qui prend en entrée un ordre pour trianguler le graphe et utilise la même technique pour enlever les arêtes non nécessaires. En 1997, Dahlhaus [Dah97] propose un algo en $O(nm)$, qui consiste à calculer une triangulation du graphe suivant un ordre donné et un arbre de clique (une décomposition arborescente) de ce graphe. Ensuite, il procède à un redécoupage des nœuds de l'arbre de sorte que les intersections entre nœuds soient des séparateurs minimaux du graphe de départ. Peyton [Pey01] a proposé une technique similaire en 2001 mais qui utilise des techniques issues du domaine des matrices creuses. L'algorithme se comporte très bien en pratique, mais aucune borne de complexité n'est donnée. Berry, Heggernes et Simonet définissent dans [BHS03] un algorithme en $O(nm)$, appelé minimal minimum degree. Il utilise mindeg qui est une heuristique très robuste pour calculer de bonnes approximations de la treewidth. En outre, la plupart des triangulations calculées par mindeg sont minimales. Une fois la triangulation avec mindeg calculée, on retire toutes les arêtes qui ont été rajoutées en dehors d'un ensemble donné de séparateurs minimaux.
Le but est de partir d'une triangulation de qualité et la rendre minimale. Cela doit, évidemment

permettre d'améliorer l'approximation définie par la triangulation. Cependant, il n'y a pas d'assurance à ce niveau.
La dernière méthode a été proposée par Berry, Heggernes et Villander dans [BHV03] avec une complexité en $O(nm)$. C'est une triangulation incrémentale qui part d'un graphe vide et rajoute un à un les sommets du graphe et les arêtes le reliant avec les sommets déjà présents en maintenant le graphe courant minimal, soit en rajoutant des arêtes (pour avoir un surgraphe minimal) ou en ne rajoutant pas toutes les arêtes reliant le sommet courant et ceux déjà présents (un sous-graphe minimal). L'ordre dans lequel on considère les sommets peut être quelconque. C'est une technique intéressante pour mettre à jour des données, car on a la possibilité de rajouter un sommet à n'importe quel moment tout en gardant la minimalité de la triangulation.

3.4 Définition de nouvelles stratégies de triangulation.

3.4.1 Stratégies basées sur les séparateurs minimaux du graphe.

Dans cette section, nous allons définir une approche basée sur la complétion d'un ensemble de séparateurs minimaux du graphe de départ ou d'un ensemble de blocs induits par ces séparateurs. Il existe des liens étroits entre les séparateurs minimaux d'un graphe et les triangulations minimales de ce dernier.

Définition 3.4.1 *[PS97] Soient S et S' deux séparateurs de G. S croise S' si S intersecte au moins deux composantes connexes induites par S'.*
Si S et S' ne se croisent pas, on dit qu'ils sont parallèles.

Ces deux relations sont symétriques.

Théorème 3.4.1 *[PS97] Soit H une triangulation minimale de G. L'ensemble des séparateurs minimaux de H est un ensemble maximal de séparateurs deux à deux parallèles de G. H est obtenu par complétion de cet ensemble de séparateurs.*

Corollaire 3.4.1 *[PS97] La complétion d'un ensemble maximal de séparateurs minimaux de G deux à deux parallèles donnent une triangulation minimale de G.*

Toute triangulation minimale peut être obtenue par la complétion d'une famille maximale de séparateurs minimaux deux à deux parallèles. Calculer la triangulation optimale revient donc à trouver la bonne famille de séparateurs. Dans la section 3.3.2, nous avons présenté des algorithmes de triangulation reposant sur la complétion de séparateurs. Notamment, la méthode d'approximation à un facteur $O(log(w))$ près de [BKMT04], qui consiste à choisir à chaque étape un séparateur minimal dont les composantes connexes contiennent au plus la moitié des sommets du graphe courant. Nous savons que cette méthode donne une approximation assez large avec une constante cachée supérieure à 675. Dans le même article, les auteurs définissent une technique, MCSep (Minimum Cardinality Separator strategy), qui n'offre pas de garantie sur la qualité de la triangulation minimale obtenue. Elle choisit à chaque étape, un séparateur de taille minimum qui décompose le graphe courant en plusieurs composantes connexes et complète ce séparateur. Ce processus est réitéré récursivement sur les différents blocs (un bloc est une réunion du séparateur avec une composante connexe qu'il induit) ainsi générés jusqu'à obtenir des blocs complets (des cliques). En définitive, il y a une complétion d'une famille maximale de séparateurs deux à deux parallèles. Il existe un algorithme avec une complexité en temps de $O(n^{4,5})$ qui calcule un séparateur de taille minimum et limite de ce fait, la complexité de MCSep à $O(n^{5,5})$.

Nous allons dans la suite définir des stratégies de triangulation avec une approche similaire à celle de MCSep. Notre objectif est d'avoir un cadre permettant de calculer une décomposition arborescente de manière très souple tout en maîtrisant un certain nombre de propriétés telles

que la taille des séparateurs, des clusters, et d'autres critères dits sémantiques, liés aux caractéristiques du (V)CSP sous-jacent. Contrairement aux algorithmes ci-dessus, nous allons plutôt essayer d'avoir une résolution du (V)CSP le plus efficace possible en pratique. Nous disposons, dans cette optique de l'algorithme introduit dans [BBC99, BBC00] qui calcule l'ensemble des séparateurs minimaux avec une complexité en $O(Nb_{sepmin}n^3)$ avec Nb_{sepmin} le nombre de séparateurs en question. La complexité de cet algorithme améliore significativement les bornes proposées jusqu'alors. Cependant, on peut penser que le calcul de l'ensemble des séparateurs minimaux d'un graphe, est dénué de sens pratique dans la mesure où leur nombre peut être très important. Mais nous travaillons ici sur une classe de graphes qui possède une structure intéressante. Il n'existe pas de définition claire et précise de la notion de graphes structurés ou de (V)CSP structurés, mais cela fait référence généralement à des graphes qui ont une treewidth très inférieure au nombre de sommets du graphe. On pense que le nombre de séparateurs sur ces graphes est limité, comparé aux graphes quelconques. Nous allons pu voir dans nos expérimentations que cette conjecture est généralement vérifiée.
Dans [BT02], les auteurs résolvent la conjecture ESA'93. Ils définissent un algorithme de complexité $O(n^3 Nbsepmin^3 + n^2 mNbsepmin^2)$, qui calcule la treewidth des graphes ayant un nombre polynomial de séparateurs minimaux. Notre technique est différente en plusieurs points. Premièrement, une fois l'ensemble des séparateurs calculés nous utilisons des heuristiques très simples pour choisir une famille maximalement parallèles. Ces séparateurs, une fois complétés, donnent une triangulation minimale (théorème 3.4.1). Cela permet d'avoir une complexité moindre. Dans un deuxième temps, si le nombre de séparateurs s'avère élevé, même s'il est polynomial, nous avons la possibilité de borner le nombre de séparateurs à calculer pour construire ensuite une triangulation qui n'est pas forcément minimale, mais qui en pratique demeure de qualité pour les graphes structurés. Le choix de la famille de séparateurs minimaux est crucial dans la qualité par rapport à l'optimum de l'approximation calculée. Choisie avec soin, elle peut induire une triangulation optimale.

L'algorithme TSep(G, G', Max) (voir algorithme 21) calcule une triangulation, G', du graphe G qui est minimale si TSep retourne une valeur $minimale = 1$. Max permet de borner le nombre maximum de séparateurs minimaux de G qui sera calculé par AllMinSep (ligne 1), l'algorithme de [BBC99]. Il limite par là même, la durée d'exécution de TSep. Si le nombre total de séparateurs minimaux de G est inférieur ou égal à Max alors $minimale = 1$ et G' sera une triangulation minimale obtenue par complétion d'une famille maximale de séparateurs minimaux deux à deux parallèles. A chaque étape (lignes 4-10), la procédure *Choisir* (ligne 5) opte pour un séparateur, S, de SM, ensemble des séparateurs minimaux de G. S est retiré de SM et rajouté à FSM (lignes 6 et 7), la famille maximale en cours de construction. Ensuite, tous les éléments de SM qui ne sont pas parallèles à S sont retirés de cet ensemble (lignes 8-10). Quand $SM = \emptyset$, FSM est une famille maximale de séparateurs deux à deux parallèles, qu'il suffit de compléter (ligne 11) pour avoir une triangulation minimale. Si $minimale = 0$, cela veut dire que AllMinSep n'a pas calculé tous les séparateurs minimaux car le nombre dépasse Max. De ce fait, il n'existe plus l'assurance de pouvoir construire une famille maximale de séparateurs. TSep calcule, dans ce cas, un ensemble de blocs EB (lignes 13-19) qui seront complétés (ligne 20) afin de construire une triangulation, pas forcément minimale. A l'image de MSVS [Kos99], EB contient un unique bloc égale à G au départ (ligne 12). Un séparateur S est choisi dans SM (ligne 14). S induit plusieurs composantes connexes CC_i, $1 \leq i \leq r$. Dans la procédure Décomposer (ligne 15), le bloc G de EB est remplacé par les blocs $CC_i \cup S$, pour tout i, $1 \leq i \leq r$. Ensuite, tout séparateur dans SM croisant S (intersecte au moins deux composantes connexes induites par S) est retiré de SM (lignes 17-19). A chaque étape (lignes 13-19), un nouveau choix de séparateur S est effectué dans SM. Puisque les séparateurs qui n'étaient pas parallèles aux séparateurs déjà choisis, ont été retirés de SM, S est contenu dans un unique bloc B_S de EB. S décompose le graphe induit par B_S en plusieurs composantes connexes CC_j. La procédure Décomposer remplace donc B_S par les blocs $CC_j \cup S$. Quand $SM = \emptyset$, les éléments de EB sont complétés et donne le graphe triangulé G' (ligne 20).

Algorithme 11 : TSep(G, G', Max)

```
1  SM ← AllMinSep(G, Max, minimale)
2  if minimale = 1 then
3      FSM ← ∅
4      while SM ≠ ∅ do
5          S ← Choisir(SM)
6          FSM ← FSM ∪ {S}
7          SM ← SM\{S}
8          forall S' ∈ SM do
9              if S' croise S then
10                 SM ← SM\{S'}
11     Compléter(G, FSM)
12 EB ← {G}
13 while SM ≠ ∅ do
14     S ← Choisir(SM)
15     Decomposer(EB, S)
16     SM ← SM\{S}
17     forall S' ∈ SM do
18         if S' croise S then
19             SM ← SM\{S'}
20 Compléter(G, EB)
21 return minimale
```

Nous avons défini plusieurs stratégies pour la procédure *Choisir* :

- $Tmin$: choisit un séparateur de taille minimum
- $Dense$: choisit un séparateur de densité maximale
- $Centre$: choisit un séparateur qui minimise la taille maximale des blocs ainsi générés
- $Parall$: choisit un séparateur qui est parallèle au plus grand nombre de séparateurs encore présents dans SM.

Les objectifs sont sensiblement différents avec ces stratégies. La complexité en espace de BTD repose sur la taille des séparateurs minimaux (intersections entre clusters). Pour une limitation de cette borne de complexité, il s'avère absolument nécessaire de maîtriser la taille des séparateurs. Il semble dès lors évident que choisir des séparateurs de petite taille ($Tmin$) peut permettre d'obtenir une décomposition de qualité. De même, une décomposition avec des séparateurs très denses ($Dense$) et donc avec *a priori* peu de solutions, peut conduire à enregistrer moins d'informations et donc requérir moins d'espace mémoire. Cependant une bonne complexité en espace est largement insuffisante pour une résolution efficace en pratique. En effet, la complexité en temps de la méthode dépend de la taille maximale des clusters. De ce point de vue une bonne décomposition est une décomposition assez proche de l'optimum. Il semble ainsi nécessaire de choisir des séparateurs qui minimisent la taille des blocs générés ($Centre$) ou bien qui évincent le moins de séparateurs possibles ($Parall$) laissant ainsi la possibilité de décomposer encore plus le graphe.

3.4.2 Minesp : une stratégie basée sur l'espérance mathématique du nombre de solutions d'un CSP.

Nous allons présenter, ici, une nouvelle heuristique de triangulation appelée Minesp. Elle repose, à travers l'espérance du nombre de solution définie dans [Smi94], sur des critères sémantiques et graphiques du CSP.

Estimation du nombre de solutions partielles L'espérance mathématique du nombre de solutions est une estimation probabiliste du nombre de solutions d'un CSP. Nous allons présenter cette notion avec un CSP binaire, mais elle est facilement applicable aux problèmes d'arité quelconque. Soit $\mathcal{P} = (X, D, C, R)$ un CSP binaire, la dureté t_{ij} d'une contrainte $c_{ij} \in C$ est le rapport du nombre de tuples interdits sur celui des tuples possibles. On considère, également, une contrainte universelle entre chaque paire de variables non contraintes. Soit $\mathcal{A} = (v_1, \ldots, v_n)$ une instanciation totale, $\mathcal{A}$ est une solution de $\mathcal{P}$ si v_i et v_j sont compatibles pour la contrainte c_{ij}, pour tout $1 \leq i < j \leq n$. La probabilité de cet événement (v_i et v_j sont compatibles) est de $(1 - t_{ij})$. Donc la probabilité pour que $\mathcal{A}$ soit solution est $\prod_{1 \leq i < j \leq n} (1 - t_{ij})$. Etant donné que le nombre d'instanciations totales est d^n alors l'espérance du nombre de solutions est $Esp(\mathcal{P}) = d^n \prod_{1 \leq i < j \leq n} (1 - t_{ij})$. Ce critère donne une assez bonne estimation de la difficulté du problème et du temps de résolution. Dans notre étude, l'estimation du nombre de solutions du problème en entier aura un intérêt limité comparé à celle des sous-problèmes définis par les clusters de la décomposition arborescente. Cette estimation de solutions partielles permet de définir des stratégies d'exploitation de décompositions pour BTD d'une grande efficacité. Elle permet également de définir l'heuristique Minesp.

Minesp Cette heuristique ordonne les sommets de 1 à n, en sélectionnant comme nouveau sommet, un dont le sous-problème induit par ses voisins non encore numérotés, a une espérance du nombre de solutions partielles minimum.

3.5 Etude expérimentale

3.5.1 Qualité de l'approximation

Nous avons présenté différentes approches de triangulation qui visent l'optimum ou une approximation de qualité. Dans un premier temps, nous allons nous focaliser sur la qualité de la triangulation en termes de proximité de l'optimum. Certes le temps de calcul est secondaire, toutefois, les algorithmes avec un coût trop prohibitif pour terminer dans des délais raisonnables sont exclus de cette étude comparative. Leur intérêt dans une phase préléminaire de résolution de (V)CSP est presque nul. Dès lors, les algorithmes optimaux et d'approximation avec garanties ne peuvent fonctionner que pour des graphes de très petite taille. Les algorithmes optimaux, quand ils terminent, permettent de voir la distance qui sépare les autres méthodes de l'optimum. Dans les autres cas, on se contente de bornes inférieures de la treewidth comme pour apprécier la qualité de la triangulation. S'il y a égalité entre la meilleure borne inférieure et la largeur de la triangulation alors on en conclut qu'elle est optimale. Cette étude comparative est grandement facilitée par la bibliothèque TreewidthLIB (http ://people.cs.uu.nl/hansb/treewidthlib/index.php). Elle regroupe plus de 700 graphes d'origines variées, la majeure partie des méthodes de calcul ou d'approximation de treewidth et les résultats qu'elles ont obtenus sur ces graphes. Parmi tous ces graphes, nous avons extrait ceux qui ont une structure de qualité. En réalité, nous avons considéré des graphes dont la treewidth ou sa meilleure approximation n'est pas au-delà de 20% des sommets du graphe, même si nous pouvons penser que pour des graphes de

grande taille, cette borne soit trop haute. Les tableaux 8.1 à 8.5 du chapitre Annexes présentent les résultats (disponibles dans la bibliothèque TreewidthLIB) des algorithmes sur les graphes structurés de cette bibliothèque. Il faut noter que ces résultats sont les meilleurs obtenus pour plusieurs exécutions des méthodes avec des stratégies différentes. La première colonne contient le numéro d'identifiant des graphes, la seconde, le nombre de sommets (n) et d'arêtes (m) de chaque graphe. UB et LB sont respectivement le meilleur majorant et le meilleur minorant de la treewidth des graphes. Les algorithmes ayant calculé ses meilleurs majorants et minorants sont donnés dans les colonnes Algorithmes-UB et Algorithmes-LB. Le terme Exact dans ces colonnes, traduit qu'un algorithme exact a pu calculer la treewidth en un temps raisonnable. All (resp. All-X) signifie que tous les algorithmes minimaux et heuristiques, présentés ici (resp. excepté X), donnent le même majorant. De même, X+TM est une triangulation minimale calculée par suppression d'arêtes (TM étant la méthode de [BHT01]) de la triangulation donnée par la méthode X.
Une première observation montre que les méthodes exactes n'arrivent pas à calculer une triangulation d'un graphe de plus de 52 sommets dans cette collection, dans des délais raisonnables. Elles ont fourni des résultats pour 22 graphes de taille réduite sur un total de 215. Cela confirme leur intérêt limité. Pour tous les autres graphes, le meilleur majorant, voire la treewidth, est obtenu grâce aux algorithmes minimaux et heuristiques, à l'exception de deux graphes pour lesquels, ce résultat est fourni par une technique de programmation dynamique et celle de recherche Tabou. Les approximations de la treewidth sans garantie sont les seules solutions valables pour le calcul d'une décomposition arborescente de manière efficace. En effet, elles sont les plus performantes sur 191 graphes parmi 193, inaccessibles aux méthodes exactes. D'autant plus que, concernant les 22 autres graphes, elles calculent avec succès un majorant égale à la treewidth pour 19 d'entre eux. Leur comportement est résumé dans le tableau 3.1.

Sparse-fill+TM (Sparse-fill est noté Sfill) obtient les meilleurs résultats, suivie de près par minfill et minfill+TM. Elle calcule une des meilleures triangulations (dont la largeur est le meilleur majorant de la treewidth) pour plus de la moitié des graphes structurés (115 sur 215) et cette triangulation est optimale pour 53 d'entre eux. Cependant, le coût de TM n'est pas négligeable. En outre, l'amélioration est souvent mineure par rapport à la triangulation de départ. L'intérêt de ce type d'approches réside principalement dans le calcul du meilleur majorant possible de la treewidth. Dans le cadre de la résolution d'un (V)CSP, une réduction de la plus grande clique d'une ou de deux unités n'est pas très important par rapport au coût de la méthode TM. Nous allons préférer donc utiliser les algorithmes minfill et MCS pour calculer une décomposition arborescente. Leurs performances sont très proches de celles de Sfill. Elles calculent un meilleur majorant pour près de la moitié des graphes : 104 sur 215 graphes pour minfill (dont 51 optimaux) et 76 pour MCS (dont 18 optimaux). En ce qui concerne MCS, son comportement peut surprendre dans la mesure où, elle est considéré comme une technique de qualité mineure dans le cas général. Cependant, pour les graphes structurés, il peut être vu comme l'une des toutes meilleures techniques avec un temps d'exécution très faible.

Nous avons également expérimenté $TSep$ avec ses différentes stratégies de choix de séparateurs, de même que la triangulation $LBTriang$ [Ber99], sur ces graphes structurés de TreewidthLIB. Les résultats sont présentés dans les tableaux de 8.6 à 8.10 du chapitre Annexes. UB est le meilleur majorant disponibe dans la bibliothèque TreewidthLIB. La colonne sep/clust informe si la triangulation est obtenue par complétion de séparateurs (sep) ou de blocs (clust), sachant qu'elle est minimale dans le premier cas. Le comportement des stratégies sur les graphes, est résumé dans le tableau 3.2. La colonne Nb-UB donne le nombre de graphes pour lesquels la stratégie a atteint le meilleur majorant de la treewidth disponible dans la bibliothèque. Nb-Exact est le nombre de graphes pour lesquels ce majorant est en fait égale à la treewidth et AM, le nombre de graphes pour lesquels il améliore le meilleur majorant Notre implémentation de l'algorithme $TSep$ n'est pas encore optimisée. Son temps d'exécution peut être réduit avec une implémentation plus fine. Pour limiter le temps nécessaire, le nombre maximum de séparateurs autorisés (Max) est limité à 150000. Pour des graphes de petite taille, cette borne est largement

Algorithmes	Nb-BUB	Nb-Exact
MCS	76	18
MCS+TM	99	27
MCS-M	48	27
LBFS	34	26
LBFS+TM	35	26
Lex-M	51	29
mindeg	54	38
mindeg+TM	54	38
minfill	104	51
minfill+TM	100	49
Sfill	52	29
Sfill+TM	115	53
MSVS	46	27

Tab. 3.1 – Comportement des triangulations heuristiques sur les 215 graphes structurés de la bibliothèque TreewidthLIB : la colonne Algorithmes contient les triangulations, Nb-UB donne le nombre de graphes pour lesquels l'algorithme a atteint le meilleur majorant de la treewidth et Nb-Exact le nombre de graphes pour lesquels ce majorant est en fait égale à la treewidth.

suffisante. Mais pour un graphe contenant 2000 sommets, une valeur plus importante serait nécessaire. Pour ce type de graphes, une amélioration de l'implémentation de $TSep$ permettra une étude plus précise. En effet, si $TSep$ dispose de l'ensemble des séparateurs minimaux, elle calcule une triangulation de grande qualité. Elle fait preuve d'une grande robustesse en fournisant à chaque fois un majorant de la treewidth très proche du meilleur, voire égal. Si par contre, elle est dispose d'une petite partie des séparateurs minimaux, les résultats sont très mauvais. Le mode de calcul des séparateurs minimaux explique ce comportement. En effet, il part d'un séparateur pour calculer tous ceux qui sont dans son voisinage. Au final, cette petite partie des séparateurs permet de décomposer la région explorée. Le reste du graphe n'est pas pris en compte et se retrouve dans un seul bloc qui sera complété pour former une clique maximale de taille considérable. Dès lors, il est important de disposer de la majeure partie des séparateurs pour assurer une décomposition équilibrée. Concernant les graphes structurés de la bibliothèque, tous les séparateurs minimaux sont calculés pour 1/4 d'entre eux. Dans ce cas, les stratégies $Tmin$, $Dense$ et $Centre$ atteignent le meilleur majorant pour plus de la moitié des graphes. $Centre$ est la plus efficace avec 32 graphes sur 55, dont 27 pour lesquels le majorant est égal à la treewidth. $Dense$ et $Tmin$ sont équivalentes sur ces graphes et atteignent le meilleur majorant pour 27 graphes parmi les 55. En outre, elles améliorent le meilleur majorant pour un graphe.

Dans les cas où $TSep$ ne dispose pas de l'ensemble des séparateurs minimaux (160 graphes), les résultats sont moins probants. $Centre$ qui reste la meilleure stratégie atteint le meilleur majorant à 22 reprises. Cependant, pour les graphes de taille raisonnable (on dispose d'une bonne partie des séparateurs), le majorant reste proche du meilleur disponible. Il améliore même ce dernier pour 7 graphes.

La stratégie $Parall$ obtient les moins bons résultats. En essayant de se laisser une grande marge de manœuvre par le choix d'un séparateur parallèle au plus grand nombre, elle "oublie" de faire des choix pertinents pour une bonne décomposition du graphe.

La méthode LBTriang atteint le meilleur majorant pour seulement 43 graphes parmi les 215. Néanmoins, elle améliore le majorant disponible pour 7 graphes.

En conclusion, nous avons observé que les meilleures approximations de l'optimum dans des temps raisonnables, sont l'œuvre de techniques heuristiques. MCS et minfill se sont révélés très efficaces sur cet éventail très larges de graphes structurés issus de domaines multiples. $TSep$, quant à elle, est une technique prometteuse qui obtient déjà d'excellents résultats sur les graphes

Algorithmes	Nb-graphes	Nb-BUB	Nb-Exact	AM
$Tmin/Dense$(sep)	55	27	21	1
$Tmin/Dense$(clust)	160	11	6	0
$Centre$(sep)	55	32	27	0
$Centre$(clust)	160	22	8	4
$Parall$(sep)	55	20	17	0
$Parall$(clust)	160	1	1	0
$LBTriang$	215	43	25	7

Tab. 3.2 – Comportement des stratégies de TSep sur les 215 graphes structurés de la bibliothèque TreewidthLIB : la colonne Algorithmes contient les différentes stratégies de TSep (sep signifie que tous les séparateurs ont pu être calculés et clust que cela n'a pas été le cas) et la triangulation LBTriang, Nb-UB donne le nombre de graphes pour lesquels la stratégie a atteint le meilleur majorant de la treewidth, Nb-Exact le nombre de graphes pour lesquels ce majorant est en fait égale à la treewidth et AM le nombre de graphes pour lesquels il améliore le meilleur majorant.

de taille raisonnable.

3.5.2 Qualité des triangulations par rapport à la résolution pratique de (V)CSP

Nous avons fini de déterminer les approches possibles pour calculer une décomposition arborescente d'un (V)CSP. Maintenant, nous allons voir le comportement de ces différents algorithmes dans la résolution de problèmes. Le critère qui importe dorénavant est le temps. Nous allons générer des (V)CSP aléatoires et les résoudre en utilisant FC-BTD ou FC-BTD-val et les décompositions arborescentes données par les algorithmes : MCS, Lex-M, LBTriang, mindeg, minfill, minesp et $TSep$ associée avec les stratégies $Tmin$, $Dense$, $Centre$ et $Parall$. En plus, nous avons défini une nouvelle stratégie de choix de séparateurs dans $TSep$, $SMinesp$.
$SMinesp$: choisit un séparateur dont l'espérance du nombre de solutions est minimum.

3.5.2.1 Résultats sur les CSP

Le générateur de CSP binaires aléatoires structurés partiels requiert plusieurs paramètres pour une classe de problèmes (n, d, w, t, s, ns, p) : n est le nombre de variables des instances de cette classe qui ont des domaines de taille d. Leurs graphes de contraintes sont des arbres de cliques avec ns sommets de taille au plus $w+1$ et des tailles de séparateurs bornées par s. t donne le nombre de couples de valeurs interdits pour chaque contrainte. A chaque fois qu'une instance est construite de la sorte, un pourcentage p d'arêtes est retiré de son graphe de contraintes de manière aléatoire. Cette modification donne naissance à l'instance structurée définitive.
Les expérimentations ont été menées sur un PC sous linux avec un processeur Pentium 4 3,2 GHz et 1 Go de RAM. Pour chaque classe d'instances, les résultats présentés dans les tableaux sont les moyennes sur un ensemble de 50 instances. Le terme Mem traduit qu'une instance n'a pu être résolue parce qu'elle requiert plus de 1 Go de mémoire. Cela entraîne la non résolution de toutes les instances qui la suivent dans la classe.
Il faut noter que la manière dont BTD explore la décomposition arborescente influe sur les résultats. Dans cette étude, nous avons utilisé la meilleure stratégie pour ces problèmes, basée sur l'espérance du nombre de solutions partielles des clusters. Elle est présentée dans le chapitre suivant dans une étude à grande échelle de ces stratégies.

Ces premiers résultats (présentés dans les tableaux 3.3 à 3.6) montrent que w n'est pas le critère le plus pertinent. La quantité de mémoire disponible est limitée. Dès qu'elle est atteinte,

CSP (n,d,w,t,s,ns,p)	MCS	Lex-M	LBTriang	Mindeg	Minfill	Minesp
(a)(150,25,15,215,5,15,10)	3,96	6,30	5,68	14,40	36,00	4,86
(b)(150,25,15,237,5,15,20)	3,75	Mem	2,90	9,19	16,28	5,49
(c)(150,25,15,257,5,15,30)	2,15	Mem	2,61	37,60	11,01	2,24
(d)(150,25,15,285,5,15,40)	1,62	Mem	3,58	47,67	16,09	2,13
(e)(250,20,20,107,5,20,10)	24,33	Mem	18,87	Mem	110,18	56,22
(f)(250,20,20,117,5,20,20)	16,95	Mem	25,84	Mem	82,60	18,96
(g)(250,20,20,129,5,20,30)	14,51	Mem	Mem	Mem	Mem	49,66
(h)(250,20,20,146,5,20,40)	4,12	Mem	Mem	Mem	Mem	17,64
(i)(250,25,15,211,5,25,10)	8,61	Mem	12,25	Mem	64,21	31,96
(j)(250,25,15,230,5,25,20)	13,59	Mem	6,66	68,14	32,73	25,13
(k)(250,25,15,253,5,25,30)	5,83	Mem	10,27	119,28	41,85	18,33
(l)(250,25,15,280,5,25,40)	18,61	Mem	17,85	31,41	Mem	18,48
(m)(250,20,20,99,10,25,10)	Mem	Mem	Mem	Mem	Mem	86,63
(n)(500,20,15,123,5,50,10)	8,06	Mem	12,16	Mem	58,69	24,49
(o)(500,20,15,136,5,50,20)	14,29	Mem	14,21	Mem	85,10	38,55

Tab. 3.3 – Durées moyennes (en s) de résolution par FC-BTD des instances des classes CSP (n,d,w,t,s,ns,p) de CSP aléatoires structurés pour des décompositions arborescentes calculées à partir différents algorithmes de triangulation : le terme "Mem" signifie que la résolution d'un problème a échoué car elle nécessitait plus de 1Go de mémoire.

CSP	MCS	Lex-M	LBTriang	Mindeg	Minfill	Minesp
(a)	-	-	-	-	-	-
(b)	-	(22,21)	-	-	-	-
(c)	-	(31,30)	-	-	-	-
(d)	-	(20,18)	-	-	-	-
(e)	-	(29,28)	-	(18,16)	-	-
(f)	-	(36,35)	-	(18,16)	-	-
(g)	-	(28,27)	(16,15)	(16,15)	(16,15)	-
(h)	-	(23,21)	(15,14)	(15,14)	(15,13)	-
(i)	-	(23,22)	-	(14,12)	-	-
(j)	-	(31,30)	-	-	-	-
(k)	-	(21,20)	-	-	-	-
(l)	-	(39,38)	-	-	(11,10)	-
(m)	(19,17)	(30,29)	(18,17)	(18,17)	(18,17)	-
(n)	-	(30,29)	-	(14,12)	-	-
(o)	-	(24,23)	-	(13,11)	-	-

Tab. 3.4 – Tailles maximales des clusters et des intersections entre clusters de la décomposition arborescente, calculée à partir des différentes heuristiques de triangulation, de l'instance non résolue par FC-BTD pour raison de quantité mémoire requise (Mem) dans les classes CSP (n,d,w,t,s,ns,p) de CSP aléatoires structurés : elles sont données sous-formes de couples (taille maximale des clusters, taille maximale des intersections), le caractère "-" traduit la résolution de toutes les instances de la classe.

CSP	Centre	Tmin	Dense	SMinesp	Parall
(a)	6,65	6,09	6,04	6,03	Mem
(b)	3,63	3,36	4,03	Mem	Mem
(c)	Mem	1,55	2,92	Mem	Mem
(d)	11,88	4,68	7,52	Mem	Mem
(e)	37,89	18,46	18,66	19,97	59,34
(f)	Mem	31,35	35,43	Mem	Mem
(g)	Mem	Mem	23,05	Mem	Mem
(h)	Mem	55,03	49,38	Mem	Mem
(i)	Mem	15,79	13,23	15,73	Mem
(j)	Mem	10,10	11,04	Mem	Mem
(k)	Mem	11,34	13,89	Mem	Mem
(l)	Mem	20,08	24,00	Mem	Mem
(m)	Mem	Mem	Mem	Mem	Mem
(n)	Mem	11,65	11,82	14,61	Mem
(o)	Mem	16,31	15,79	Mem	Mem

Tab. 3.5 – Durées moyennes (en s) de résolution par FC-BTD des instances des classes CSP (n,d,w,t,s,ns,p) de CSP aléatoires structurés pour des décompositions arborescentes calculées à partir différentes stratégies de TSep : le terme "Mem" signifie que la résolution d'un problème a échoué car elle nécessitait plus de 1Go de mémoire.

CSP	Centre	Tmin	Dense	SMinesp	Parall
(a)	-	-	-	-	(38,37)
(b)	-	-	-	(20,19)	(22,21)
(c)	(16,15)	-	-	(18,17)	(23,21)
(d)	-	-	-	(33,32)	(20,19)
(e)	-	-	-	-	-
(f)	(22,21)	-	-	(36,35)	(17,16)
(g)	(33,32)	(16,15)	-	(37,36)	(25,24)
(h)	(21,20)	-	-	(30,27)	(45,44)
(i)	(18,17)	-	-	-	(34,33)
(j)	(20,19)	-	-	(61,60	(21,20)
(k)	(16,15)	-	-	(33,32)	(32,31)
(l)	(15,13)	-	-	(34,33)	(29,28)
(m)	(18,17)	(18,16)	(18,16)	(18,17)	(30,29)
(n)	(24,23)	-	-	-	(32,31)
(o)	(16,15)	-	-	(36,35)	(28,27)

Tab. 3.6 – Tailles maximales des clusters et des intersections entre clusters de la décomposition arborescente, calculée à partir des différentes stratégies de TSep, de l'instance non résolue par FC-BTD pour raison de quantité mémoire requise (Mem) dans les classes CSP (n,d,w,t,s,ns,p) de CSP aléatoires structurés : elles sont données sous-formes de couples (taille maximale des clusters, taille maximale des intersections), le caractère "-" traduit la résolution de toutes les instances de la classe.

la résolution est bloquée. Il est donc préférable d'avoir une décomposition de largeur importante, mais qui permette de terminer la résolution grâce à un ensemble de séparateurs de taille maîtrisée. Limiter la taille des séparateurs est donc très important, voire même le plus important. En effet, Minesp est l'unique méthode qui fournit des décompositions arborescentes permettant à FC-BTD de résoudre l'ensemble des instances. Toutes les autres méthodes échouent sur au moins une instance. Exceptée l'instance non résolue de la classe (250,20,20,99,10,25,10), MCS semble être la plus robuste avec souvent les meilleurs résultats. Cette classe engendre une explosion mémoire pour toutes les méthodes exceptée minesp. LBTriang rencontre des problèmes de mémoire pour trois instances des classes (250,20,20,129,5,20,30), (250,20,20,146,5,20,40) et (250,20,20,99,10,25,10). Ses performances sur les autres classes ne sont pas très éloignées de celles de MCS. Le comportement de la triangulation Lex-M est catastrophique. Elle ne réussit à résoudre que la première classe d'instances. Elle rencontre des problèmes mémoire dans toutes les autres.

Concernant $TSep$, la stratégie *Dense* réussit à résoudre l'ensemble des classes, exceptée toujours la classe (250,20,20,99,10,25,10). Ces résultats sont moins bons que ceux MCS et LB-Triang, mais ils s'en approchent souvent malgré le coût de $TSep$ qui est plus important que celui des autres techniques. La stratégie $Tmin$ a des performances similaires, hormis la classe (250,20,20,129,5,20,30) qui occasionne une explosion mémoire. $SMinesp$, $Centre$ et $Parall$ échouent dans la résolution de la majeure partie des classes.

Il est nécessaire de réduire la taille des intersections entre les clusters de la décomposition arborescente pour permettre à la résolution d'arriver à son terme. Les propriétés des instances non résolues (taille maximale des clusters de la décomposition arborescente et taille maximale des intersections de ces clusters) nous donnent déjà des indices sur la taille des séparateurs à ne pas dépasser.

On observe une explosion mémoire pour une décomposition arborescente calculée grâce à minfill et dont la taille maximale des intersections entre clusters est limitée à 10. Cette valeur qui peut paraître assez faible au départ, peut donc causer une requête d'espace mémoire considérable. En effet, suivant le problème, la masse d'informations à enregistrer peut varier de manière importante. Cela dépend de la consistance ou non du problème, de la rapidité de FC-BTD à trouver une solutio s'il en existe, du nombre d'affectations consistantes sur les séparateurs, etc. Il faut ainsi borner la taille des séparateurs à une valeur qui permette à la résolution d'aboutir même en cas de mémorisation d'un grand nombre d'informations sur les intersections.

3.5.2.2 Résultats sur les VCSP

Nous poursuivons notre étude sur les VCSP.

Le générateur de VCSP binaires aléatoires structurés partiels est similaire à celui de CSP de la section précédente. Il requiert les mêmes paramètres pour une classe d'instances (n, d, w, t, s, ns, p). L'unique différence est qu'un coût entre 1 et 10 est associé à chaque tuple interdit. Les résultats présentés dans les tableaux sont les moyennes sur un ensemble de 50 instances. Nous utilisons la meilleure stratégie de parcours de décompositions pour ces problèmes, qui est basée sur l'espérance du nombre de solutions partielles des clusters.

Néanmoins, au niveau des méthodes de triangulation, nous avons apporté une modification à la définition de l'espérance du nombre de solutions pour en prendre en compte, pas uniquement les tuples autorisés, mais tous les tuples avec les coûts associés. Cela nécessite donc une implémentation différente qui n'a pas encore été faite. Donc, ces techniques ne figurent pas dans les expérimentations présentées ici.

Les résultats (présentés dans les tableaux 3.7 à 3.10) confirment les observations réalisées dans le cadre CSP. Le problème lié à l'espace mémoire requis et à la taille des séparateurs, est encore plus crucial ici. La nécessité de trouver l'affectation de valuation optimale fait que FC-BTD-val doit explorer tout l'espace de recherche pour faire la preuve d'optimalité. Aussi, la masse d'informations à mémoriser est plus importante. La valeur de w n'est pas non plus le

VCSP (n,d,w,t,s,ns,p)	MCS	Lex-M	LBTriang	Mindeg	Minfill
(75,10,15,30,5,8,10)	Mem	Mem	Mem	Mem	Mem
(75,10,15,30,5,8,20)	22,49	Mem	Mem	Mem	Mem
(75,10,15,33,3,8,10)	38,45	Mem	Mem	31,47	44,35
(75,10,15,34,3,8,20)	6,59	Mem	7,27	Mem	7,93
(75,10,10,40,3,10,10)	7,10	Mem	6,94	45,26	7,11
(75,10,10,42,3,10,20)	1,95	Mem	2,95	5,80	4,09
(75,15,10,102,3,10,10)	Mem	Mem	Mem	Mem	Mem
(100,5,15,13,5,10,10)	Mem	Mem	Mem	Mem	Mem

Tab. 3.7 – Durées moyennes (en s) de résolution par FC-BTD-val des instances des classes VCSP (n,d,w,t,s,ns,p) de VCSP aléatoires structurés pour des décompositions arborescentes calculées à partir différents algorithmes de triangulation : le terme "Mem" signifie que la résolution d'un problème a échoué car elle nécessitait plus de 1Go de mémoire.

VCSP	MCS	Lex-M	LBTriang	Mindeg	Minfill
(75,10,15,30,5,8,10)	(14,12)	(17,16)	(14,12)	(14,12)	(14,12)
(75,10,15,30,5,8,20)	-	(13,12)	(12,11)	(12,11)	(12,11)
(75,10,15,33,3,8,10)	-	(19,18)	(13,12)	-	-
(75,10,15,34,3,8,20)	-	(19,18)	-	(13,11)	-
(75,10,10,40,3,10,10)	-	(14,13)	-	-	-
(75,10,10,42,3,10,20)	-	(15,14)	-	-	-
(75,15,10,102,3,10,10)	(9,8)	(20,19)	(9,8)	(9,8)	(9,8)
(100,5,15,13,5,10,10)	(14,13)	(19,18)	(14,13)	(14,13)	(14,13)

Tab. 3.8 – Tailles maximales des clusters et des intersections entre clusters de la décomposition arborescente, calculée à partir des différentes heuristiques de triangulation, de l'instance non résolue par FC-BTD-val pour raison de quantité mémoire requise (Mem) dans les classes VCSP (n,d,w,t,s,ns,p) de VCSP aléatoires structurés : elles sont données sous-formes de couples (taille maximale des clusters, taille maximale des intersections), le caractère "-" traduit la résolution de toutes les instances de la classe.

VCSP	Centre	Tmin	Dense	Parall
(75,10,15,30,5,8,10)	Mem	Mem	Mem	Mem
(75,10,15,30,5,8,20)	Mem	Mem	Mem	Mem
(75,10,15,33,3,8,10)	28,17	30,50	30,47	Mem
(75,10,15,34,3,8,20)	8,81	7,05	7,22	Mem
(75,10,10,40,3,10,10)	Mem	6,49	6,45	Mem
(75,10,10,42,3,10,20)	4,52	3,12	3,29	Mem
(75,15,10,102,3,10,10)	Mem	Mem	Mem	Mem
(100,5,15,13,5,10,10)	Mem	Mem	Mem	Mem

Tab. 3.9 – Durées moyennes (en s) de résolution par FC-BTD-val des instances des classes VCSP (n,d,w,t,s,ns,p) de VCSP aléatoires structurés pour des décompositions arborescentes calculées à partir différentes stratégies de TSep : le terme "Mem" signifie que la résolution d'un problème a échoué car elle nécessitait plus de 1Go de mémoire.

VCSP	Centre	Tmin	Dense	Parall
(75,10,15,30,5,8,10)	(13,12)	(14,12)	(14,12)	(15,14)
(75,10,15,30,5,8,20)	(18,17)	(12,11)	(12,10)	(12,11)
(75,10,15,33,3,8,10)	-	-	-	(18,17)
(75,10,15,34,3,8,20)	-	-	-	(15,14)
(75,10,10,40,3,10,10)	(16,15)	-	-	(14,13)
(75,10,10,42,3,10,20)	-	-	-	(12,11)
(75,15,10,102,3,10,10)	(9,8)	(9,8)	(9,8)	(14,13)
(100,5,15,13,5,10,10)	(14,13)	(14,13)	(14,13)	(19,18)

Tab. 3.10 – Tailles maximales des clusters et des intersections entre clusters de la décomposition arborescente, calculée à partir des différentes stratégies de TSep, de l'instance non résolue par FC-BTD-val pour raison de quantité mémoire requise (Mem) dans les classes VCSP (n,d,w,t,s,ns,p) de VCSP aléatoires structurés : elles sont données sous-formes de couples (taille maximale des clusters, taille maximale des intersections), le caractère "-" traduit la résolution de toutes les instances de la classe.

critère le plus pertinent dans ce cadre.

MCS parvient à résoudre 5 classes parmi 8. Elle semble être la technique la plus robuste, ici aussi. Minfill et les stratégies *Dense* et $Tmin$ de $TSep$ résolvent à leur tour la moitié des classes. Au niveau temps, les performances sont très proches. Lex-M et la stratégie *Parall* ne résolvent aucune classe. Elles confirment la nécessité de maîtriser la complexité spatiale.

On observe au niveau de la classe (75,15,10,102,3,10,10), une explosion mémoire pour toutes les décompositions arborescentes calculées grâce à toutes les méthodes, cela, malgré une taille maximale des intersections entre clusters limitée à 8.

3.5.3 Limitation de la taille des séparateurs : une nouvelle comparaison

Ici, nous reprenons les expérimentations des sections précédentes avec, cette fois-ci, une limitation de la taille maximale des séparateurs. Cette dernière est bornée à la valeur 5. Nous verrons dans le chapitre suivant que cette valeur permet de faire un excellent compromis expace/temps pour les problèmes traités. En dehors de $TSep$, la méthode consiste à calculer une décomposition arborescente du graphe de contraintes grâce aux techniques de triangulation, pour ensuite fusionner toute paire de clusters dont la taille de l'intersection, qui est un séparateur, dépasse la limite fixée. Pour $TSep$, les choses sont faites durant la triangulation. Tous les séparateurs dont la taille dépasse la limite sont directement retirés de l'ensemble SM, juste après son calcul par AllSepMin. Cette approche semble plus séduisante. En effet, avec une borne sur la taille des séparateurs, il est possible de construire différentes décompositions. A partir de là, celle qui minimise la taille des clusters est fortement souhaitée pour avoir une complexité en temps intéressante. $TSep$ donne des moyens d'agir sur celle qui sera calculée, contrairement aux autres techniques.

3.5.3.1 Résultats sur les CSP

Les deux tableaux réunis dans la table 3.11 présentent les résultats obtenus pour les différentes triangulation. La dernière ligne (EC) des tableaux de résultats sur la durée de résolution donne l'écart moyen avec le meilleur résultat obtenu pour une classe. EC traduit la robustesse des méthodes. En se basant sur cette valeur, on établit un classement des méthodes. Ce classement est important car notre préférence va vers les heuristiques les plus robustes qui obtiennent donc soit les meilleurs résultats ou du moins en reste proches en général. Dans ce cas, une méthode

ayant de très bons résultats sur toutes les classes sauf une instance serait moins préférable pour calculer la décomposition arborescente. Cela peut sembler contestable, mais nous faisons ce choix.

L'heuristique la plus performante est la stratégie $Tmin$ de $TSep$ suivie de très près par $Dense$. La différence de comportement entre elles est quasi nulle. Leur écart moyen est inférieur à 0,45 secondes. Elles obtiennent presque toujours les meilleurs résultats. MCS et LBTriang sont également d'une grande efficacité avec respectivement $EC = 0,58$ et $EC = 0,70$. Ces écarts moyens étant très proches, on pourrait dire que ces quatre techniques sont équivalentes. Cependant, il faut noter que le coût de calcul des triangulations avec $TSep$ est largement supérieur à celui des autres méthodes. Malgré cela, $Tmin$ et $Dense$ sont plus performantes. Certes la différence est faible, mais une amélioration de l'implémentation de $TSep$ pourrait engendrer un écart beaucoup plus conséquent. Elle semble donc très prometteuse. En outre, son comportement est expliqué par sa capacité à calculer des triangulations de largeur réduite tout en respectant la borne imposée pour la taille des séparateurs (tableau 3.12). Les stratégies calculent toujours les meilleures triangulations en termes de largeur. La largeur des triangulations de LBTriang et MCS étant très proche de celle des premières, on note que le fait de pouvoir préserver cette largeur tout en réduisant la taille des séparateurs est primordial pour assurer un bon compromis. La treewidth n'est pas l'unique critère pertinent, mais il reste pertinent.

Ensuite, nous observons un écart important avec les autres heuristiques. Minesp est relativement efficace malgré des triangulations de largeur importante. Cela est compensé par la difficulté relativement réduite des clusters construits grâce à l'espérance du nombre de solutions.

Lex-M ($EC = 8,39$) a un comportement largement meilleur par rapport à la première partie de nos expérimentations (taille de séparateurs non bornée). Cette tendance à l'amélioration des résultats est vérifiée pour toutes les heuristiques. Cela n'est pas uniquement lié à la réduction du nombre d'informations enregistrées et exploitées, mais également à la liberté accrue laissée à l'heuristique de choix de variables qui est dynamique à l'intérieur des clusters dont la taille a augmenté en général. Cependant, il ne suffit d'avoir des clusters de très grande taille pour être performant. La largeur réduite des triangulations des meilleures techniques ($Tmin$, $Dense$, MCS et LBTriang) nous le confirme. Cette réduction de la taille des intersections entraîne la fusion de clusters qui partageaient beaucoup de variables. Ceci augmente légèrement la taille des clusters et donne la possibilité à l'heuristique de choix de variables d'affecter des variables qui étaient inaccessibles avant (car se trouvant dans un autre cluster) pour détecter les inconsistances encore plus tôt.

3.5.3.2 Résultats sur les VCSP

Nous procédons de la même manière que dans la section précédente pour valider nos observations dans le cadre des VCSP. Les deux tableaux de la table 3.13 présentent les résultats obtenus pour les différentes triangulation. Malheureusement, la résolution de VCSP est souvent très difficile. Pour voir le mieux possible les différences de performances, nous n'avons pas imposé de temps limite au bout duquel la résolution d'une instance est abandonnée. Les techniques les plus efficaces ont abouti rapidement, tandis que les autres travaillent encore. Ainsi, le signe "-" traduit que la résolution des instances de cette classe n'est pas terminée.

Nous retrouvons les mêmes heuristiques $Tmin$, $Dense$, MCS et LBTriang comme étant les plus efficaces. Dans le cadre VCSP plus qu'ailleurs, la nécessité d'avoir une triangulation de largeur réduite est très forte (tableau 3.14). Le comportement de minfill à ce niveau lui permet d'obtenir de très bons résultats assez proches des meilleurs.

Ces résultats confirment totalement nos premières observations. $TSep$ est ainsi une méthode très prometteuse. Elle permet de limiter la taille des séparateurs tout en calculant une décomposition de largeur satisfaisante. Elle constitue un moyen très efficace de réaliser le compromis espace/temps.

CSP	MCS	Lex-M	LBTriang	Mindeg	Minfill	Minesp
(a)	3,38	4,88	3,54	22,07	8,63	4,29
(b)	2,38	2,90	2,32	5,95	5,63	4,47
(c)	1,11	12,66	1,21	2,21	6,23	1,05
(d)	0,62	1,54	0,61	6,45	14,29	0,75
(e)	13,73	14,02	13,13	57,04	51,92	30,96
(f)	12,69	11,87	12,72	77,95	77,60	11,89
(g)	9,32	37,62	10,80	65,92	56,25	24,70
(h)	5,12	12,97	5,26	20,82	14,04	15,37
(i)	10,23	11,99	9,87	87,82	24,61	19,33
(j)	5,65	8,29	5,77	77,34	28,73	17,99
(k)	5,34	9,07	5,06	34,68	5,46	5,98
(l)	2,64	19,71	4,77	18,08	28,86	4,38
(m)	89,46	120,68	89,44	99,81	87,36	88,58
(n)	7,87	9,52	7,60	219,70	21,90	12,67
(o)	8,22	17,14	7,49	54,23	76,92	29,21
EC	0,58	8,39	0,70	45,40	22,63	6,84

CSP	Centre	Tmin	Dense	SMinesp	Parall
(a)	36,93	4,20	4,19	3,62	10,43
(b)	20,08	2,30	2,31	15,49	20,49
(c)	4,39	1,10	1,11	3,49	19,37
(d)	5,50	0,62	0,61	4,57	6,39
(e)	26,65	13,56	13,59	23,07	44,86
(f)	38,84	12,53	12,50	29,47	37,79
(g)	14,49	10,97	11,02	8,75	45,65
(h)	6,84	4,42	4,43	16,00	18,09
(i)	21,85	9,77	9,74	30,07	64,91
(j)	31,79	4,82	4,82	37,44	15,19
(k)	68,18	5,01	5,17	36,38	25,56
(l)	13,39	4,63	4,61	14,95	24,29
(m)	89,20	87,59	87,70	88,30	120,13
(n)	32,49	7,35	7,37	38,41	55,82
(o)	41,09	6,66	6,67	43,09	17,13
EC	18,85	0,43	0,45	14,94	23,81

Tab. 3.11 – Durées moyennes (en s) de résolution par FC-BTD des instances des classes CSP (n,d,w,t,s,ns,p) de CSP aléatoires structurés pour des décompositions arborescentes calculées à partir des différentes heuristiques de triangulation, avec une taille maximale d'intersections entre clusters bornée par 5 : la dernière ligne EC, l'écart moyen entre les résultats de ces différentes techniques et le meilleur résultat obtenu pour chaque classe.

CSP	MCS		Lex-M		LBTriang		Mindeg		Minfill		Minesp	
	T	C	T	C	T	C	T	C	T	C	T	C
(a)	0,01	15,04	0,01	18,36	0,01	15,00	0,01	21,18	0,01	18,12	0,09	25,14
(b)	0,01	15,04	0,01	22,68	0,01	15,00	0,01	18,68	0,01	17,80	0,08	23,68
(c)	0,01	15,86	0,01	24,68	0,01	15,02	0,01	18,80	0,01	19,02	0,06	20,50
(d)	0,01	16,48	0,01	26,76	0,02	15,00	0,01	18,34	0,01	17,30	0,05	17,94
(e)	0,02	20,00	0,03	25,38	0,04	20,00	0,02	31,20	0,03	27,94	0,32	31,78
(f)	0,02	20,00	0,03	29,02	0,04	20,00	0,02	31,24	0,03	26,80	0,28	28,88
(g)	0,02	20,82	0,03	32,76	0,04	20,16	0,02	32,38	0,03	27,86	0,25	26,26
(h)	0,03	21,44	0,03	37,48	0,04	20,04	0,03	30,94	0,03	27,36	0,21	25,22
(i)	0,02	15,00	0,03	21,98	0,04	15,00	0,02	27,32	0,02	20,76	0,27	29,24
(j)	0,02	15,40	0,03	27,12	0,04	15,08	0,02	26,48	0,03	21,82	0,21	23,04
(k)	0,03	16,02	0,03	29,58	0,04	15,06	0,03	24,78	0,03	20,86	0,18	21,56
(l)	0,03	18,90	0,03	34,20	0,04	15,22	0,03	21,10	0,03	20,66	0,15	19,86
(m)	0,02	58,28	0,03	62,04	0,04	58,28	0,02	74,64	0,02	60,96	0,61	60,46
(n)	0,10	15,02	0,12	26,12	0,17	15,00	0,08	42,32	0,10	26,12	0,95	26,50
(o)	0,11	15,44	0,13	31,14	0,17	15,02	0,09	38,28	0,11	26,90	0,89	25,58

CSP	Centre		Tmin		Dense		SMinesp		Parall	
	T	C	T	C	T	C	T	C	T	C
(a)	0,04	16,62	0,04	15,00	0,04	15,00	0,04	15,70	0,04	26,18
(b)	0,09	16,74	0,09	15,00	0,09	15,00	0,09	16,56	0,09	25,28
(c)	0,25	16,72	0,25	15,00	0,25	15,00	0,25	16,96	0,24	22,68
(d)	0,51	16,46	0,51	14,98	0,51	14,98	0,50	16,76	0,49	19,78
(e)	0,19	21,38	0,18	20,00	0,18	20,00	0,18	20,84	0,18	33,42
(f)	0,55	21,36	0,54	20,00	0,54	20,00	0,54	21,72	0,54	31,08
(g)	4,35	21,42	4,26	20,00	4,30	20,00	4,33	21,80	4,35	28,58
(h)	1,21	21,52	118,85	20,00	118,98	20,00	119,65	22,04	118,78	27,26
(i)	0,12	17,08	0,12	15,00	0,12	15,00	0,12	17,24	0,12	28,68
(j)	0,29	17,26	0,28	15,00	0,29	15,00	0,28	17,80	0,28	27,30
(k)	0,73	17,28	0,72	15,00	0,72	15,00	0,72	17,98	0,72	24,56
(l)	3,93	17,12	3,92	15,00	3,93	15,00	3,91	17,72	3,87	21,16
(m)	0,29	58,56	0,29	58,28	0,29	58,28	0,29	58,52	0,29	65,82
(n)	0,55	18,30	0,53	15,00	0,53	15,00	0,53	17,24	0,52	28,72
(o)	1,47	18,30	1,45	15,00	1,44	15,00	1,44	18,72	1,44	27,62

Tab. 3.12 – Durées moyennes (en s) de calcul des décompositions arborescentes, calculées à partir des différentes heuristiques, avec une taille maximale des intersections entre clusters limitée à 5 des instances des classes CSP (n,d,w,t,s,ns,p) de CSP aléatoires structurés et tailles maximales moyennes des clusters de ces décompositions : la colonne CSP présente les classes d'instances CSP, les colonnes suivantes donnent dans un premier temps (colonne T) le temps de calcul en moyenne des décompositions obtenues grâce aux différentes stratégies et dans un second temps (colonne C) la taille maximale en moyenne des clusters de ces décompositions.

VCSP	MCS	Lex-M	LBTriang	Mindeg	Minfill
(75,10,15,30,5,8,10)	149,12	187,72	148,60	397,35	189,51
(75,10,15,30,5,8,20)	12,16	14,88	12,51	152,95	11,36
(75,10,15,33,3,8,10)	23,69	55,35	21,18	51,14	28,07
(75,10,15,34,3,8,20)	4,71	9,55	4,23	8,44	5,48
(75,10,10,40,3,10,10)	5,64	-	5,64	-	5,84
(75,10,10,42,3,10,20)	1,64	-	1,63	-	1,75
(75,15,10,102,3,10,10)	44,68	-	44,50	-	49,88

VCSP	Centre	Tmin	Dense	Parall
(75,10,15,30,5,8,10)	420,01	148,81	148,81	-
(75,10,15,30,5,8,20)	56,50	12,55	12,25	-
(75,10,15,33,3,8,10)	155,12	21,07	21,04	-
(75,10,15,34,3,8,20)	20,31	4,35	4,32	-
(75,10,10,40,3,10,10)	-	5,44	5,56	-
(75,10,10,42,3,10,20)	-	1,69	1,69	-
(75,15,10,102,3,10,10)	-	43,84	44,53	-

Tab. 3.13 – Durées moyennes (en s) de résolution par FC-BTD-val des instances des classes VCSP (n,d,w,t,s,ns,p) de VCSP aléatoires structurés pour des décompositions arborescentes calculées à partir des différentes heuristiques avec une taille maximale d'intersections entre clusters bornée par 5 : la colonne VCSP présente les classes d'instances VCSP, les colonnes suivantes donnent le temps obtenu par FC-BTD-val grâce aux différentes techniques.

3.6 Conclusion

Nous avons présenté les différentes techniques de calcul de triangulations qui ont été définies pour approcher le plus possible l'optimum. Les méthodes utilisables pour le calcul d'une décomposition qui est une phase préliminaire dans la résolution de (V)CSP sont les techniques heuristiques et minimales. Les méthodes exactes et d'approximation avec garanties ont un coût trop prohibitif dans ce cadre. Une étude expérimentale à large échelle, grâce à la bibliothèque TreewidthLIB, a révélé que les triangulations heuristiques et minimales donnent de très bonnes approximations en un temps assez faible. Nous nous sommes concentrés sur les graphes structurés qui ont un intérêt dans ce travail. La méthode MCS, de qualité moyenne dans le cas général, se révèle très efficace dans ce cadre. Minfill obtient les meilleurs résultats, tandis que $TSep$ avec les stratégies Tmin, Dense et Centre, est déjà très prometteur dans l'optique d'une meilleure implémentation. En ce qui concerne la résolution de problème, la largeur des décompositions n'est pas le critère le plus important. La taille des séparateurs pouvant mener à la requête d'un espace mémoire indisponible et bloquer la résolution, il faut maîtriser ce paramètre. Cependant, pour une limite fixée, il est possible d'avoir des décompositions de largeur différente. TSep-Tmin, TSep-Dense, MCS et LBTriang arrivent à maîtiser ces deux paramètres. L'ordre d'affectation des variables a un impact très important dans l'efficacité des méthodes énumératives, telles que FC et MAC. Les heuristiques statiques obtiennent des résultats désastreux comparés à ceux des heuristiques dynamiques. L'ordre induit par une décomposition limite les degrés de liberté laissés à BTD pour ordonner les variables. Le chapitre suivant est consacré à la prise en compte de ce critère dans l'appréciation de la qualité des décompositions.

VCSP	MCS		Lex-M		LBTriang		Mindeg		Minfill	
	T	C	T	C	T	C	T	C	T	C
(75,10,15,30,5,8,10)	0,01	15,00	0,01	17,38	0,01	15,00	0,01	16,54	0,01	15,86
(75,10,15,30,5,8,20)	0,01	15,22	0,01	18,48	0,01	15,00	0,01	16,28	0,01	15,84
(75,10,15,33,3,8,10)	0,01	15,06	0,01	15,88	0,01	15,00	0,01	15,04	0,01	15,00
(75,10,15,34,3,8,20)	0,01	15,00	0,01	16,60	0,01	15,00	0,01	15,10	0,01	15,00
(75,10,10,40,3,10,10)	0,01	10,00	-	-	0,01	10,00	-	-	0,01	10,00
(75,10,10,42,3,10,20)	0,01	10,00	-	-	0,01	10,00	-	-	0,01	10,00
(75,15,10,102,3,10,10)	0,01	10,00	-	-	0,01	10,00	-	-	0,01	10,00

VCSP	Centre		Tmin		Dense		Parall	
	T	C	T	C	T	C	T	C
(75,10,15,30,5,8,10)	0,01	15,72	0,01	15,00	0,01	15,00	-	-
(75,10,15,30,5,8,20)	0,03	15,74	0,03	15,00	0,02	15,00	-	-
(75,10,15,33,3,8,10)	0,01	15,52	0,01	15,00	0,01	15,00	-	-
(75,10,15,34,3,8,20)	0,02	15,62	0,02	15,00	0,02	15,00	-	-
(75,10,10,40,3,10,10)	-	-	0,01	10,00	0,01	10,00	-	-
(75,10,10,42,3,10,20)	-	-	0,01	10,00	0,01	10,00	-	-
(75,15,10,102,3,10,10)	-	-	0,01	10,00	0,01	10,00	-	-

Tab. 3.14 – Durées moyennes (en s) de calcul des décompositions arborescentes, calculées à partir des différentes heuristiques de triangulation, avec une taille maximale des intersections entre clusters limitée à 5 des instances des classes VCSP (n,d,w,t,s,ns,p) de VCSP aléatoires structurés et tailles maximales moyennes des clusters de ces décompositions : la colonne VCSP présente les classes d'instances VCSP, les colonnes suivantes donnent dans un premier temps (colonne T) le temps de calcul en moyenne des décompositions obtenues grâce aux différentes techniques et dans un second temps (colonne C) la taille maximale en moyenne des clusters de ces décompositions.

Chapitre 4

Exploitation d'hypergraphes acycliques recouvrants

4.1 Introduction

Le chapitre précédent a montré que la taille des clusters n'est pas l'unique critère pertinent pour une résolution efficace de (V)CSP structuré. Une limitation de la taille des séparateurs est incontournable pour éviter d'enregistrer une masse trop importante d'informations qui conduirait au blocage de la résolution. Un autre aspect d'une résolution efficace se situe au niveau de l'heuristique de choix de variables. Sans heuristique de choix de variables de qualité, MAC et FC se révèlent incapables de résoudre un grand nombre de problèmes. Les heuristiques efficaces sont dynamiques car elles tiennent compte de l'évolution du problème pour faire des choix justifiés contrairement aux heuristiques statiques. Ces dernières ont des résultats souvent catastrophiques. La version de base de BTD utilise un ordre statique induit par la décomposition. Il y a donc une nécessité impérieuse de sortir de cette rigidité. L'exploitation dynamique de la décomposition va autoriser une liberté accrue dans le choix de l'ordre d'affectation des variables. Dans les articles [JNT05a, JNT06, JNT07], nous avons proposé des heuristiques pour une exploitation dynamique des décompositions arborescentes. L'objectif est de trouver un bon compromis entre la qualité des bornes de complexité théorique et la nécessité d'exploiter des heuristiques dynamiques efficaces de choix de variables. Les travaux concernant le "AND/OR Branch-and-Bound" pour l'optimisation sous contraintes [MD06], et le Recursive conditionning de Darwiche [Dar01] rentrent aussi dans ce cadre.
Pour faciliter ce type d'approche, nous proposons un cadre différent de celui basé sur une décomposition arborescente du problème. Ce cadre repose sur le concept de recouvrement par hypergraphe acyclique qui est très proche de celui de décomposition arborescente, mais demeure plus général. Son utilisation conduit à l'exploitation d'une voire plusieurs décompositions arborescentes construites de manière incrémentale durant la résolution. Nous sortons de ce fait du choix arbitraire d'un arbre d'une décomposition qui était opéré jusqu'alors. Cela conduit à de meilleurs résultats théoriques et pratiques. Etant donné l'hypergraphe de contraintes $H = (X, C)$ du problème traité, nous considérons un recouvrement de ce graphe par un hypergraphe acyclique $H_A = (X, E)$: l'ensemble des sommets est le même alors que, pour toute arête $c \in C$, il existe une hyperarête $E_i \in E$ recouvrant c ($c \subset E_i$). Dans la suite, nous allons définir différentes classes d'hypergraphes acycliques qui recouvrent H_A. Ces classes sont définies sur la base de critères relatifs à la nature des recouvrements et des relations existant avec les méthodes de résolution : bornes de complexité issues de paramètres du type de la treewidth, préservation des séparateurs de H, fusion d'hyperarêtes voisines, capacité à l'implémentation d'heuristiques efficaces (essentiellement dynamiques). Dans un premier temps, nous proposons une nouvelle

définition d'acyclicité dans les hypergraphes. Ensuite, les recouvrements acycliques de problèmes sont étudiés théoriquement, de façon à déterminer leurs caractéristiques et leurs propriétés. Après cela, nous montrons qu'il est possible de préserver les résultats de complexité connus, et nous en présentons d'autres. De plus, nous indiquons comment cette notion de recouvrement permet d'offrir un cadre naturel pour la gestion dynamique de la structure. Ensuite, nous présentons un nouvel algorithme défini dans ce cadre et pour lequel il est assez facile d'étendre les heuristiques connues. Il devient dès lors facile d'accroître dynamiquement l'étendue du choix offert à une heuristique dynamique d'ordonnancement des variables. Enfin, une étude pratique montre que son implémentation reste simple, nous permettant ainsi de mieux apprécier l'intérêt pratique de cette approche.

4.2 Une nouvelle définition des hypergraphes acycliques

De manière très surprenante, la notion d'acyclicité sur les hypergraphes en termes de cycle sur les hyperarêtes a été très peu étudiée contrairement au cas des graphes. Il existe des définitions équivalentes multiples qui font référence pour la plupart aux articulations des hypergraphes ou aux cycles dans une représentation en graphe des propriétés de connexion entre les hyperarêtes des hypergraphes. Cet état de fait peut paraître paradoxal vu le lien étroit existant entre les deux termes.

[Wan05] a proposé une définition de cycle dans les hypergraphes qui induit une équivalence entre l'acyclicité d'un hypergraphe et l'absence de cycle dans ce dernier.

Définition 4.2.1 *[Wan05] Soit $H = (X, C)$ un hypergraphe. Un cycle est une séquence d'hyperarêtes $(C_{i_1}, C_{i_2}, \ldots, C_{i_K})$ vérifiant les deux conditions suivantes :*
- $C_{i_K} = C_{i_1}$
- $\forall 2 \leq j \leq K-2$ *et* $\forall c \in C$, $(S_{j-1} \cup S_j \cup S_{j+1}) \backslash c \neq \emptyset$, *avec* $S_j = C_{i_j} \cap C_{i_{j+1}}$, $\forall 1 \leq j \leq K-1$.

Cette définition est très simple, mais englobe des séquences d'hyperarêtes qui sont appelées des pseudo cycles car ils ne sont pas à l'origine de l'acyclicité des hypergraphes. En effet, les pseudo cycles ne prennent pas en compte la propriété de running intersection des hypergraphes qui est équivalente à celle d'acyclicité de ces derniers (théorème 2.2.1). Cependant, il est possible de montrer que l'existence d'un pseudo cycle entraîne celle d'un cycle dit essentiel pour la propriété d'acyclicité.

Aussi, nous proposons ici une définition de cycle dans les hypergraphes, appelé α-cycle, qui offre une nouvelle équivalence avec l'acyclicité. Les α-cycles s'approchent des cycles essentiels définis dans [Wan05]. En outre, ils sont à la base de notre technique de construction incrémentale d'arbres de décompositions arborescente présentée dans la section 4.4.1.

Tout d'abord, nous allons rappeler la notion d'intergraphes minimaux et quelques propriétés la concernant qui jouent un rôle central dans les preuves de nos résultats.

On considère tout au long de cette section un hypergraphe $H = (X, C)$.

Définition 4.2.2 *[Ber70] $R = (C, B)$, avec $B = \{\{C_i, C_j\} | C_i, C_j \in C,\ C_i \neq C_j$ et $C_i \cap C_j \neq \emptyset\}$, est appelé graphe représentatif des hyperarêtes de H.*

Les sommets du graphe représentatif de H sont les hyperarêtes de ce dernier et il existe une arête entre deux sommets de R si leur intersection n'est pas vide. R traduit les intersections dans H.

Définition 4.2.3 *[BG81] Soit $R = (C, B)$ le graphe représentatif des hyperarêtes de H. Un graphe $G = (C, A)$ est un intergraphe de H, si $A \subset B$ et $\forall C_i, C_j \in C$ tel que $C_i \cap C_j \neq \emptyset$, alors il existe une chaîne $(C_i = C_{u_1}, C_{u_2}, \ldots C_{u_R} = C_j)$ telle que $\forall 1 \leq k < R$, $C_i \cap C_j \subset C_{u_k} \cap C_{u_{k+1}}$. On note $\mathcal{C}(H)$ l'ensemble des intergraphes de H.*

Un intergraphe est un graphe partiel de R dans lequel il n'est pas nécessaire d'avoir une arête entre deux sommets d'intersection non vide, s'il existe une chaîne entre eux qui contient cette intersection (c'est-à-dire qu'elle est incluse dans les intersections entre éléments consécutifs de la chaîne).

Définition 4.2.4 *Un intergraphe $G = (C, A)$ de H est minimal si $\forall A' \subsetneq A$, $G = (C, A')$ n'est pas un intergraphe.*
On note $\mathcal{C}_m(H)$ l'ensemble des intergraphes minimaux de H.

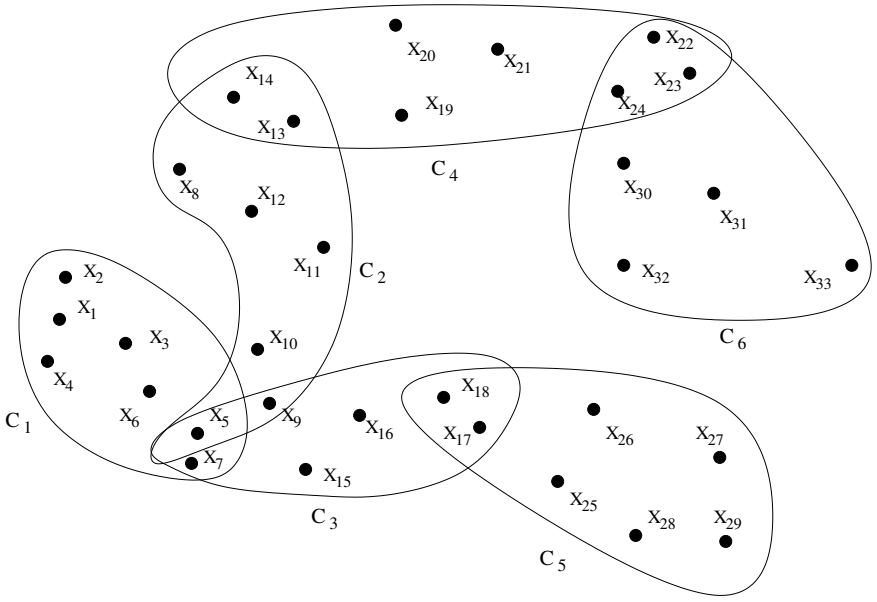

Fig. 4.1 – Un hypergraphe acyclique.

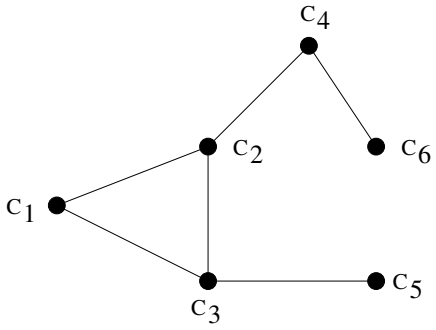

Fig. 4.2 – Graphe représentatif des hyperarêtes de l'hypergraphe de la figure 4.1.

Cette condition de minimalité sur les intergraphes impose qu'il existe une arête entre deux sommets d'intersection non vide ssi il n'existe pas une chaîne entre eux contenant leur intersection. Cette définition d'intergraphe minimal est similaire à celle d'arbre de jointures. Les intergraphes minimaux d'un hypergraphe donné, vérifient une propriété commune.

Théorème 4.2.1 *[JV93] Si $G = (C, A)$ et $G' = (C, A')$ sont deux intergraphes minimaux de H, alors $|A| = |A'|$.*

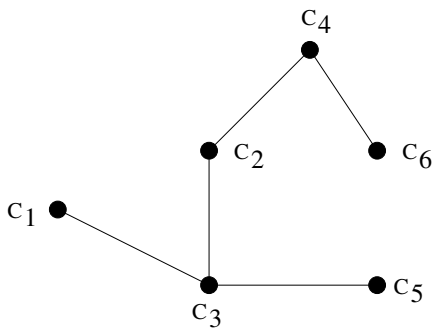

Fig. 4.3 – Un intergraphe minimal de l'hypergraphe de la figure 4.1.

Le nombre d'arêtes dans les intergraphes minimaux d'un hypergraphe est un invariant. On peut noter également que tout intergraphe ayant le même nombre d'arêtes qu'un intergraphe minimal est minimal. Par ailleurs, il existe une équivalence entre l'acyclicité d'un hypergraphe et celle de ses intergraphes minimaux.

Théorème 4.2.2 *[BFMY83] H est acyclique ssi tout intergraphe minimal de H est acyclique.*

En utilisant les propriétés sur les intergraphes minimaux, on pourra aussi noter qu'un intergraphe minimal de H est acyclique ssi tous les autres le sont également. La figure 4.1 présente un hypergraphe acyclique dont le graphe de la figure 4.2 est le graphe de représentation des hyperarêtes. Ce graphe est également un intergraphe de l'hypergraphe. Cependant, il n'est pas minimal car le graphe partiel de la figure 4.3 est un intergraphe de l'hypergraphe. Ce dernier est minimal. En outre, il est acyclique montrant par la même occasion que l'hypergraphe est acyclique.

Nous allons maintenant définir les notions d'hyperarêtes α-voisines et de α-chaîne dans un hypergraphe $H = (X, C)$. Nous nous inspirons pour cela des intergraphes minimaux.

Définition 4.2.5 *Soient deux hyperarêtes C_u et C_v telles que $C_u \cap C_v \neq \emptyset$.*
On appelle séquence de voisinage entre C_u et C_v, une séquence $(C_u = C_{i_1}, C_{i_2}, \ldots, C_{i_R} = C_v)$ telle que $R > 2$ et $C_u \cap C_v \subsetneq C_{i_j} \cap C_{i_{j+1}}$, pour $j = 1, \ldots, R-1$.
On appelle séquence de voisinage élémentaire entre C_u et C_v, une séquence de voisinage entre C_u et C_v, $(C_{i_1}, C_{i_2}, \ldots, C_{i_Q})$ telle que $\nexists C_{i_a}, C_{i_b}$, avec $1 \leq i_a < i_Q$, $1 \leq i_b < i_Q$ et $C_{i_a} \neq C_{i_b}$, tel que $C_{i_a} \cap C_{i_{a+1}} \subset C_{i_b} \cap C_{i_{b+1}}$.

Deux hyperarêtes sont α-voisines s'il n'existe pas un autre chemin (une séquence de voisinage) permettant d'aller de l'une à l'autre.

Dans le cas où, il existe une chaîne de voisinage entre deux hyperarêtes C_u et C_v, il n'existe pas d'intergraphes minimaux de H qui contiennent une arête entre C_u et C_v.

Définition 4.2.6 *Soient C_u et C_v deux hyperarêtes de H. C_u et C_v sont α-voisines s'il n'existe pas une séquence de voisinage entre elles.*

Donc, C_u et C_v sont α-voisines s'il existe un intergraphe minimal de H qui contient une arête entre C_u et C_v.

Par exemple, l'hypergraphe de la figure 4.1 contient deux hyperarêtes C_1 et C_2 d'intersection non vide qui ne sont pas α-voisines car (C_1, C_3, C_2) est une séquence de voisinage reliant C_1 et C_2. L'intergraphe minimal de la figure 4.3 de l'hypergraphe de la figure 4.1 ne contient pas l'arête $\{C_1, C_2\}$.

Théorème 4.2.3 *Si H est connexe, alors toute séquence de voisinage entre deux hyperarêtes C_u et C_v de H contient une séquence de voisinage élémentaire entre C_u et C_v.*

Preuve Soit $Seq_1 = (C_{i_1}, C_{i_2}, \ldots, C_{i_R})$ la séquence de voisinage entre C_u et C_v. Si Seq_1 n'est pas élémentaire alors il existe deux hyperarêtes C_{i_a}, C_{i_b} telles que $i_a, i_b < i_R$, $C_{i_a} \neq C_{i_b}$ et $C_{i_a} \cap C_{i_{a+1}} \subset C_{i_b} \cap C_{i_{b+1}}$. On pose $q = C_{i_a} \cap C_{i_{a+1}}$. $q \subset C_{i_a}$ et $q \subset C_{i_{b+1}}$, donc $q \subset C_{i_a} \cap C_{i_{b+1}}$ et $C_u \cap C_v \subsetneq C_{i_a} \cap C_{i_{b+1}}$. On suppose que $i_a < i_{b+1}$. On peut construire une nouvelle séquence Seq_2 à partir de Seq_1. $Seq_2 = (C_{i_1}, \ldots, C_{i_a}, C_{i_{b+1}}, \ldots, C_{i_R})$ est égale à Seq_1 de C_{i_1} à C_{i_a}. Cette dernière est suivie de $C_{i_{b+1}}$ et le reste de Seq_2 est égale Seq_1 de $C_{i_{b+1}}$ à C_{i_R}. Si $i_{b+1} < i_a$, $C_{i_{b+1}}$ est avant C_{i_a} dans Seq_2. Si Seq_2 est élémentaire, notre but est atteint. Sinon, il existe deux hyperarêtes C_{i_c}, C_{i_d} telles que $i_c, i_d < i_R$, $C_{i_c} \neq C_{i_d}$ et $C_{i_c} \cap C_{i_{c+1}} \subset C_{i_d} \cap C_{i_{d+1}}$. Et, on recommence le même travail effectué pour Seq_1. Sachant qu'à chaque fois, au moins une hyperarête est retirée de Seq_i pour construire Seq_{i+1}, le dernier élément de cette suite est une séquence de voisinage élémentaire entre C_u et C_v.□

Corollaire 4.2.1 *Soient C_u et C_v deux hyperarêtes de H qui ne sont pas α-voisines. Si H est connexe, alors il existe une séquence de voisinage élémentaire entre C_u et C_v.*

Preuve H étant connexe et C_u et C_v n'étant pas α-voisines, il existe une séquence de voisinage entre C_u et C_v. Cette séquence contient une séquence élémentaire de voisinage entre C_u et C_v, d'après le théorème précédent.□

Définition 4.2.7 *Une α-chaîne dans H est une séquence d'hyperarêtes $(C_{i_1}, \ldots, C_{i_R})$ telle que $\forall j, 1 \leq j < R, C_{i_j}$ et $C_{i_{j+1}}$ sont α-voisines.*
Une α-chaîne élémentaire est une α-chaîne $(C_{i_1}, \ldots C_{i_R})$ telle que $\forall j, k, 1 \leq j, k < R$, si $|j-k| \geq 2$ alors C_{i_j} et C_{i_k} ne sont pas α-voisines.

Nous conservons avec cette notion de α-chaîne la propriété de connexion liée à la définition usuelle de chaîne.

Définition 4.2.8 *H est α-connexe s'il existe une α-chaîne reliant toute paire d'hyperarêtes de H.*

Théorème 4.2.4 *Soient C_u et C_v deux hyperarêtes de H qui ne sont pas α-voisines. Si H est connexe, alors il existe une α-chaîne entre C_u et C_v dont les intersections entre hyperarêtes consécutives contiennent strictement $C_u \cap C_v$.*

Preuve H étant connexe et C_u, C_v n'étant pas α-voisines, il existe une séquence de voisinage élémentaire Seq_1 telle que l'intersection entre deux hyperarêtes consécutives contiennent strictement $C_u \cap C_v$ (corollaire 4.2.1).

Nous allons construire cette α-chaîne entre C_u et C_v à partir de la séquence de voisinage. Il y a deux cas possibles.

Premier cas : les éléments consécutifs de Seq_1 sont deux à deux α-voisins donc Seq_1 est une α-chaîne élémentaire entre C_u et C_v. En plus, les intersections entre hyperarêtes consécutives contiennent strictement $C_u \cap C_v$ et aucune intersection de ce type n'est incluse dans une autre.

Deuxième cas : il existe dans Seq_1 au moins une paire d'hyperarêtes consécutives qui ne sont pas α-voisines. Pour chaque paire (C_{a_1}, C_{b_1}) d'hyperarêtes consécutives qui ne sont pas α-voisines, il existe une séquence de voisinage, Seq' dont les intersections entre hyperarêtes consécutives contiennent strictement $q_1 = C_u \cap C_v$.

Soient C_{c_1} et C_{d_1} deux hyperarêtes consécutives de Seq_1.

On suppose qu'il existe deux hyperarêtes consécutives de Seq', $C_{c'_1}$ et $C_{d'_1}$, telles que $C_{c'_1} \cap C_{d'_1} \subset C_{c_1} \cap C_{d_1}$, alors, étant donné que $C_{a_1} \cap C_{b_1} \subsetneq C_{c'_1} \cap C_{d'_1}$, $C_{a_1} \cap C_{b_1} \subsetneq C_{c_1} \cap C_{d_1}$. Or, Seq_1 est une séquence de voisinage élémentaire, donc cela est impossible.

Supposons, cette fois, $C_{c_1} \cap C_{d_1} \subsetneq C_{c'_1} \cap C_{d'_1}$. Nous transformons Seq_1 en passant directement de $C_{c'_1}$ à C_{d_1} car $C_{c_1} \cap C_{d_1} \subset C_{c'_1} \cap C_{d_1}$.

Ensuite, nous poursuivons la transformation la séquence en mettant entre deux hyperarêtes consécutives non α-voisines, la séquence de voisinage qui les relie. On a ainsi, une nouvelle séquence Seq_2 reliant C_u et C_v.

Si les éléments consécutifs de Seq_2 sont deux à deux α-voisins, Seq_2 est une α-chaîne entre C_u et C_v dont les intersections entre hyperarêtes consécutives contiennent strictement $C_u \cap C_v$.

Sinon, il existe au moins une paire d'hyperarêtes consécutives qui ne sont pas α-voisines. Nous réitérons l'opération précédente pour construire une nouvelle séquence Seq_3 et ainsi de suite.

Cette suite de Seq_k est finie. Supposons par l'absurde qu'elle soit infinie. Dans chaque Seq_k, il existe au moins une paire d'hyperarêtes consécutives qui ne sont pas α-voisines. Pour chaque paire (C_{a_k}, C_{b_k}) d'hyperarêtes consécutives qui ne sont pas α-voisines et qui se situe dans $Seq_k \backslash Seq_{k-1}$, il existe une séquence de voisinage dont les intersections d'hyperarêtes consécutives contiennent strictement q_k, $q_{k-1} \subsetneq q_k$. La suite des q_k est aussi infinie. Mais, cette suite étant strictement croissante et le nombre de sommets de H étant fini, il est impossible qu'elle soit infinie.

Donc la suite de Seq_k est finie. Son dernier élément est une α-chaîne entre C_u et C_v dont les intersections entre hyperarêtes consécutives contiennent strictement $C_u \cap C_v$.□

Corollaire 4.2.2 *H est connexe ssi H est α-connexe.*

Preuve

Supposons que H est α-connexe. Soient C_u et C_v deux hyperarêtes de H. Il existe une α-chaîne reliant C_u et C_v. Or, les hyperarêtes consécutives d'une α-chaîne ont une intersection non vide. Donc, cette α-chaîne est une séquence d'hyperarêtes d'intersection non vide qui relie C_u et C_v. H est connexe.

Maintenant, on suppose que H est connexe. Soient C_u et C_v, deux hyperarêtes de H. Si elles sont α-voisines, (C_u, C_v) est une α-chaîne les reliant. Sinon, le théorème 4.2.4 permet de conclure qu'il existe une α-chaîne reliant C_u et C_v. H est donc α-connexe.□

Définition 4.2.9 *Un α-cycle dans H est une α-chaîne $(C_{i_1}, C_{i_2}, \ldots, C_{i_R})$ telle que $R > 3$, $C_{i_1} = C_{i_R}$, $\nexists 1 \leq a \neq b < R$, $C_{i_a} \cap C_{i_{a+1}} \subset C_{i_b} \cap C_{i_{b+1}}$.*

Le théorème suivant montre l'équivalence entre l'acyclicité des hypergraphes et l'existence de α-cycle.

Théorème 4.2.5 *H est acyclique ssi H ne contient pas de α-cycle.*

Preuve

Nous allons commencer par montrer que H est acyclique s'il ne contient pas de α-cycle.

Faisons une preuve par l'absurde. On suppose que H est acyclique. Supposons que H contienne un α-cycle : $(C_{i_1}, C_{i_2}, \ldots, C_{i_R})$. On va démontrer qu'il existe un intergraphe minimal de H dans lequel $(C_{i_1}, C_{i_2}, \ldots C_{i_R})$ forme un cycle. $\forall a,\ 1 \leq a < R$, on considère les hyperarêtes C_{i_a} et $C_{i_{a+1}}$. Dans tout $G = (C, A) \in \mathcal{C}_m(H)$, il existe une chaîne $(C_{i_a} = C_{u_1}, C_{u_2}, \ldots, C_{u_L} = C_{i_{a+1}})$ contenant l'intersection $C_{i_a} \cap C_{i_{a+1}}$. Puisque C_{i_a} et $C_{i_{a+1}}$ sont des éléments consécutifs du α-cycle, ils sont α-voisins. Donc, il existe $b,\ 1 \leq b < L$ tel que $C_{i_a} \cap C_{i_{a+1}} = C_{u_b} \cap C_{u_{b+1}}$ (sinon cette chaîne serait une séquence de voisinage entre C_{i_a} et $C_{i_{a+1}}$).

Considérons le graphe G' tel que $G' = (C, A')$ avec $A' = (A \backslash \{\{C_{u_b}, C_{u_{b+1}}\})\} \cup \{\{C_{i_a}, C_{i_{a+1}}\}\}$. Dans G', les propriétés de connexion sont conservées car C_{u_b} et $C_{u_{b+1}}$ sont reliées par la chaîne $(C_{u_b}, C_{u_{b-1}}, \ldots, C_{u_2}, C_{i_a}, C_{i_{a+1}}, C_{u_{L-1}}, \ldots, C_{u_{b+1}})$. Donc $G' \in \mathcal{C}(H)$. De plus, G' a le même nombre d'arêtes que G, alors $G' \in \mathcal{C}_m(H)$. On vient de construire un intergraphe minimal G' de H dans lequel C_{i_a} et $C_{i_{a+1}}$ sont voisines pour tout $1 \leq a < R$. $(C_{i_1}, C_{i_2}, \ldots C_{i_R})$ est donc un cycle de G'. Or H étant acyclique, il ne peut admettre un intergraphe minimal cyclique. Donc H ne contient pas de α-cycle.

Maintenant, nous allons prouver que si H ne contient pas de α-cycle alors il est acyclique. Pour cela, nous allons montrer la contre-apposée : si H est cyclique alors il contient un α-cycle.

On suppose que H cyclique. On sait que tout intergraphe minimal $G = (C, A)$ de H est cyclique.

Soit $(C_{u_1}, C_{u_2}, \ldots, C_{u_K}, C_{u_{K+1}} = C_{u_1})$ un cycle de G. On a donc $K \geq 3$. De plus, $\forall b, 1 \leq b \leq K,\ C_{u_b} \cap C_{u_{b+1}} \neq \emptyset$. Par ailleurs, puisque G est minimal, $\forall b, 1 \leq b \leq K$, il n'existe pas de chaîne $(C_{u_b} = C_{v_1}, C_{v_2}, \ldots, C_{v_L} = C_{u_{b+1}})$ dans G telle que $\forall c, 1 \leq c \leq L,\ C_{u_b} \cap C_{u_{b+1}} \subsetneq C_{v_c} \cap C_{v_{c+1}}$. Si c'était le cas, le graphe $G' = (C, A')$, avec $A' = A \backslash \{\{C_{u_b}, C_{u_{b+1}}\}\}$, serait un intergraphe de H. Ceci contredirait la minimalité de G. Dès lors, on peut dire qu'il n'existe de séquence de voisinage entre C_{u_b} et $C_{u_{b+1}}$. Elles sont donc α-voisines.

Nous allons montrer que le cycle $(C_{u_1}, C_{u_2}, \ldots, C_{u_K}, C_{u_1})$ induit l'existence d'un α-cycle dans H. Nous avons deux cas de figures.

Tout d'abord, s'il n'existe pas a et b, $1 \leq a \neq b \leq K$ tel que $C_{u_a} \cap C_{u_{a+1}} \subset C_{u_b} \cap C_{u_{b+1}}$ alors $(C_{u_1}, C_{u_2}, \ldots, C_{u_K}, C_{u_1})$ est un α-cycle de H.

Deuxième cas de figure, il existe a et b, $1 \leq a \neq b \leq K$ tel que $C_{u_a} \cap C_{u_{a+1}} \subset C_{u_b} \cap C_{u_{b+1}}$. Pour simplifier, on suppose que $a < b$ dans la suite. Dans ce cas, $K > 3$. En effet, si $K = 3$, le cycle contient uniquement $(C_{u_1}, C_{u_2}, C_{u_3}, C_{u_1})$. Puisqu'une intersection $C_{u_a} \cap C_{u_{a+1}}$ est contenue dans une autre, alors la troisième intersection contient également $C_{u_a} \cap C_{u_{a+1}}$. L'arête $\{C_{u_a}, C_{u_{a+1}}\}$ est donc redondante puisqu'il existe une chaîne reliant C_{u_a} et $C_{u_{a+1}}$ qui contient leur intersection. Ceci est en contradiction avec la minimalité de G. Donc, nécessairement $K \geq 4$.

Par ailleurs, on ne peut pas avoir $u_{a+1} = u_b$ et $u_{b+1} = u_a$ car il s'agirait de la même intersection dans ce cas. Or, nous cherchons à construire un α-cycle dans le cas où il existe une intersection entre les éléments du cycle $(C_{u_1}, C_{u_2}, \ldots, C_{u_K}, C_{u_1})$ qui soit incluse dans une autre. Ainsi, si $u_{a+1} = u_b$ alors $u_{b+1} \neq u_a$. De même, si $u_{b+1} = u_a$ alors $u_{a+1} \neq u_b$.

Supposons dans un premier temps que $u_{b+1} = u_a$, donc $u_{a+1} \neq u_b$.
En outre, l'arête $\{C_{u_{a+1}}, C_{u_b}\}$ n'est pas présente dans G. Dans le cas contraire, l'arête $\{C_{u_a}, C_{u_{a+1}}\}$ serait redondante à cause de l'existence de la chaîne $(C_{u_{a+1}}, C_{u_b}, C_{u_{b+1}} = C_{u_a})$ qui relie C_{u_a} et $C_{u_{a+1}}$ et contient leur intersection $C_{u_a} \cap C_{u_{a+1}}$ car $C_{u_a} \cap C_{u_{a+1}} \subset C_{u_b} \cap C_{u_{b+1}}$. Cela est impossible car G est minimal.
Considérons le graphe $G' = (C, A')$, avec $A' = (A \backslash \{\{C_{u_a}, C_{u_{a+1}}\}\}) \cup \{\{C_{u_{a+1}}, C_{u_b}\}\}$. C_{u_a} et $C_{u_{a+1}}$ sont maintenant reliée par la chaîne $(C_{u_{a+1}}, C_{u_b}, C_{u_{b+1}} = C_{u_a})$ qui contient leur intersection. Nous conservons les propriétés de connexion, donc G' est un intergraphe. En outre, G' contient le même nombre d'arêtes que G, un intergraphe minimal, donc il est minimal. Dans le graphe G', $(C_{u_{a+1}}, C_{u_{a+2}}, \ldots, C_{u_b}, C_{u_{a+1}})$, composée d'un sous-ensemble d'éléments du cycle $(C_{u_1}, C_{u_2}, \ldots, C_{u_K}, C_{u_1})$, est un cycle. En effet, il contient tous les éléments de
$(C_{u_1}, C_{u_2}, \ldots, C_{u_K}, C_{u_1})$, excepté $C_{u_{b+1}} = C_{u_a}$. Or, $K \geq 4$, donc $(C_{u_{a+1}}, C_{u_{a+2}}, \ldots, C_{u_b}, C_{u_{a+1}})$ contient au moins trois éléments distincts. Nous venons de définir un cycle d'un intergraphe minimal G' qui contient moins d'éléments que notre premier cycle de G, ceci dans le cas où $C_{u_{b+1}} = C_{u_a}$.

Supposons maintenant que $u_{b+1} \neq u_a$.
Nous avons deux cas possibles : $u_{a+1} = u_b$ et $u_{a+1} \neq u_b$.

Si $u_{a+1} = u_b$ alors l'arête $\{C_{u_a}, C_{u_{a+1}}\}$ est redondante à cause de l'existence de la chaîne $(C_{u_{a+1}} = C_{u_b}, C_{u_{b+1}}, C_{u_a})$ qui relie C_{u_a} et $C_{u_{a+1}}$ et contient leur intersection $C_{u_a} \cap C_{u_{a+1}}$ car $C_{u_a} \cap C_{u_{a+1}} \subset C_{u_b} \cap C_{u_{b+1}}$. Or, G est minimal. Donc, cela est impossible.

On a montré ainsi que $u_{a+1} \neq u_b$.

Par ailleurs, les deux chaînes $(C_{u_a}, C_{u_{a+1}}, \ldots, C_{u_b}, C_{u_{b+1}})$ et $(C_{u_a}, C_{u_{a-1}}, \ldots, C_{u_{b+1}})$ définies sur les éléments du cycle $(C_{u_1}, C_{u_2}, \ldots, C_{u_K}, C_{u_1})$, relient C_{u_a} et $C_{u_{b+1}}$.

Nous envisageons deux cas :
Premier cas, $C_{u_a} \cap C_{u_{a+1}}$ est inclus tout au long d'au moins une de ces deux chaînes. On suppose que cette chaîne soit $(C_{u_a}, C_{u_{a+1}}, \ldots, C_{u_b}, C_{u_{b+1}})$. Considérons le graphe $G' = (C, A')$, avec $A' = (A \backslash \{\{C_{u_a}, C_{u_{a+1}}\}\}) \cup \{\{C_{u_a}, C_{u_{b+1}}\}\}$. C_{u_a} et $C_{u_{a+1}}$ sont maintenant reliée par la chaîne $(C_{u_{a+1}}, \ldots, C_{u_b}, C_{u_{b+1}}, C_{u_a})$ qui contient $C_{u_a} \cap C_{u_{a+1}}$. Nous conservons les propriétés de

connexion, donc G' est un intergraphe. En outre, l'arête $\{C_{u_a}, C_{u_{b+1}}\}$ n'est pas présente dans G. Dans le cas contraire, l'arête $\{C_{u_a}, C_{u_{a+1}}\}$ serait redondante à cause de l'existence de la chaîne $(C_{u_{a+1}}, \ldots, C_{u_b}, C_{u_{b+1}}, C_{u_a})$ qui contient $C_{u_a} \cap C_{u_{a+1}}$. Ceci contredirait la minimalité de G. Donc G' a le même nombre d'arêtes que G, un intergraphe minimal. On en déduit que G' est minimal. Dans ce graphe minimal, $(C_{u_a}, C_{u_{a-1}}, \ldots, C_{u_{b+1}}, C_{u_a})$ est un cycle qui contient moins d'éléments que le cycle $(C_{u_1}, C_{u_2}, \ldots, C_{u_K}, C_{u_{K+1}} = C_{u_1})$ de G. En effet, il contient au moins trois éléments distincts : C_{u_a}, $C_{u_{b+1}}$ et les éléments de la chaîne $(C_{u_a}, C_{u_{a-1}}, \ldots, C_{u_{b+1}})$ dont au moins un est distinct de C_{u_a} et $C_{u_{b+1}}$ car ces dernières ne sont pas voisines dans G. En plus, $C_{u_{a+1}}$ n'appartient pas à ce nouveau cycle.

Si, $(C_{u_a}, C_{u_{a-1}}, \ldots, C_{u_{b+1}})$ est la chaîne contenant $C_{u_a} \cap C_{u_{a+1}}$, on construit de la même manière un cycle dans un intergraphe minimal G' de telle sorte que ce cycle contienne moins d'éléments que $(C_{u_1}, C_{u_2}, \ldots, C_{u_K}, C_{u_1})$.

Deuxième cas, $C_{u_a} \cap C_{u_{a+1}}$ n'est pas inclus tout au long d'une des deux chaînes $(C_{u_a}, C_{u_{a+1}}, \ldots, C_{u_b}, C_{u_{b+1}})$ et $(C_{u_a}, C_{u_{a-1}}, \ldots, C_{u_{b+1}})$. Cependant, il existe nécessairement une chaîne $(C_{u_a} = C_{v_1}, C_{v_2}, \ldots, C_{v_L} = C_{u_{b+1}})$ qui relie C_{u_a} et $C_{u_{b+1}}$ et contient leur intersection. Il existe à nouveau deux cas de figure.

Première possibilité, cette chaîne contient la séquence $(C_{u_a}, C_{u_{a+1}})$. Considérons le graphe $G' = (C, A')$, avec $A' = (A \backslash \{\{C_{u_a}, C_{u_{a+1}}\}\}) \cup \{\{C_{u_a}, C_{u_{b+1}}\}\}$. Nous conservons les propriétés de connexion car C_{u_a} et $C_{u_{a+1}}$ sont dorénavant reliées par la chaîne $(C_{u_{a+1}} = C_{v_{a'}}, C_{v_{a'+1}}, \ldots, C_{v_L} = C_{u_{b+1}}, C_{u_a})$ qui contient $C_{u_a} \cap C_{u_{a+1}}$. G' est donc un intergraphe. En plus, ayant le même nombre d'arêtes que G, G' est minimal. Comme précédemment, $(C_{u_a}, C_{u_{a-1}}, \ldots, C_{u_{b+1}}, C_{u_a})$ est un cycle de G' qui contient moins d'éléments que le cycle $(C_{u_1}, C_{u_2}, \ldots, C_{u_K}, C_{u_{K+1}} = C_{u_1})$ de G. En effet, $(C_{u_a}, C_{u_{a-1}}, \ldots, C_{u_{b+1}}, C_{u_a})$ contient C_{u_a}, $C_{u_{b+1}}$ et au moins un autre élément. Dans le cas contraire, $C_{u_a} \cap C_{u_{a+1}}$ serait inclus tout au long de la chaîne $(C_{u_a}, C_{u_{b+1}})$: ce qui est impossible.

Deuxième possibilité, la chaîne $(C_{u_a} = C_{v_1}, C_{v_2}, \ldots, C_{v_L} = C_{u_{b+1}})$ ne contient pas la séquence $(C_{u_a}, C_{u_{a+1}})$.
Considérons le graphe $G' = (C, A')$, avec $A' = (A \backslash \{\{C_{u_a}, C_{u_{a+1}}\}\}) \cup \{\{C_{u_{a+1}}, C_{u_{b+1}}\}\}$. Nous avons toujours les propriétés de connexion car C_{u_a} et $C_{u_{a+1}}$ sont reliées par la chaîne $(C_{u_a} = C_{v_1}, C_{v_2}, \ldots, C_{v_L} = C_{u_{b+1}}, C_{u_{a+1}})$ qui contient $C_{u_a} \cap C_{u_{a+1}}$. De ce fait, G' est un intergraphe qui a le même nombre d'arêtes que G. Il est donc minimal.
En plus, $(C_{u_{a+1}}, C_{u_{a+2}}, \ldots, C_{u_b}, C_{u_{b+1}}, C_{u_{a+1}})$ est un cycle de G' qui contient moins d'éléments que le cycle $(C_{u_1}, C_{u_2}, \ldots, C_{u_K}, C_{u_{K+1}} = C_{u_1})$ de G (C_{u_a} n'y figure pas).

Nous avons montré que dans tous les cas, soit le cycle $(C_{u_1}, C_{u_2}, \ldots, C_{u_K}, C_{u_1})$ induit l'existence d'un α-cycle dans H ou d'un cycle strictement plus court dans un autre intergraphe minimal. Dans le cas où il induit un cycle plus court, nous pouvons répéter le procédé sur ce dernier. Le nombre d'éléments du cycle de départ étant fini, cette opération ne pourra être rééditée qu'à un nombre fini de reprises. La dernière étape permet de déduire l'existence d'un α-cycle dans H.□

Le résultat suivant est important dans la définition de notre procédure de construction incrémentale de décompositions arborescentes.

Théorème 4.2.6 *Soit E_u, E_v et E_w trois hyperarêtes d'un hypergraphe acyclique, H_A, qui sont mutuellement α-voisines. Deux parmi les trois intersections entre hyperarêtes sont égales et contenues dans la troisième.*

Preuve S'il n'existe pas d'intersection entre ses éléments qui en contienne une autre, alors ils forment un cycle. Puisque H_A est acyclique, il ne contient pas de cycle. Donc il existe une intersection qui en contient au moins une autre. On pose $q_1 = E_u \cap E_v$, $q_2 = E_v \cap E_w$ et $q_3 = E_w \cap E_u$. On suppose que $q_1 \subset q_2$: $q_1 \subset E_u$, $q_1 \subset E_w$ et donc $q_1 \subset q_3$. Si $q_1 \subsetneq q_2$ et $q_1 \subsetneq q_3$, alors (E_v, E_w, E_u) est une séquence de voisinage entre E_u et E_v. Ceci est impossible car ils sont α-voisins. Donc, q_1 est égale à q_2 et/ou q_3.□

4.3 Recouvrements par hypergraphes acycliques

Les notions de décomposition arborescente de graphes et de recouvrement par un hypergraphe acyclique sont très voisines. Cependant, celle de recouvrement s'avère moins restrictive. En effet, le calcul d'une décomposition arborescente commence en général par une triangulation de la 2-section ([Ber70]) de l'hypergraphe de contraintes. Les cliques maximales de la 2-section triangulée définissent les clusters de la décomposition. Enfin, la construction d'un arbre sur les clusters donne les relations de voisinage entre les clusters. Cet arbre n'est pas unique.

Prenons le cas simple où deux clusters E_i et E_j intersectent un troisième E_k tel qu'une des intersections contienne l'autre : $E_i \cap E_k \subset E_j \cap E_k$. E_i et E_j peuvent être voisins de E_k dans un arbre alors que dans un autre arbre, E_i serait voisin de E_j qui serait lui voisin de E_k. Le choix de l'un d'entre eux est alors *a priori* complètement arbitraire.

Le choix de l'un de ses arbres avant le début de la recherche peut être totalement arbitraire. A l'image des heuristiques de choix de variables dynamiques, il serait sans doute préférable de faire ce choix durant la résolution pour prendre en compte la nature et les évolutions du problème. Nous décidons, ici, d'adopter cette approche. Nous conservons l'ensemble des cliques maximales qui définit en fait un hypergraphe acyclique recouvrant de l'hypergraphe de contraintes. Cet hypergraphe va permettre de construire de manière incrémentale l'ordre de visite des clusters et de ce fait l'arbre de la décomposition arborescente associée. En outre, cet ordre peut être dynamique et mener à l'utilisation de différents arbres de différentes décompositions. Nous bénéficions ainsi d'une liberté accrue dans le choix de l'heuristique d'ordonnancement des variables.

Les deux graphes de la figure 4.5 sont des arbres de deux décompositions arborescentes possibles dont les clusters sont les hyperarêtes du recouvrement acyclique de la figure 4.4.

Définition 4.3.1 *Soit $H = (X, C)$ un hypergraphe. Un recouvrement par un hypergraphe acyclique (CAH) du graphe H est un hypergraphe acyclique $H_A = (X, E)$ tel que pour chaque $c \in C$, il existe $E_i \in E$ tel que $c \subset E_i$.*
La largeur γ d'un CAH (X, E) est égale à $max_{E_i \in E}|E_i|$. La CAH-largeur γ^* de H est la largeur minimale parmi tous les CAH de H. Enfin, $\mathcal{CAH}(H)$ dénotera l'ensemble des CAHs de H.

Pour une décomposition arborescente (E, T) de la 2-section de $H = (X, C)$, le couple (X, E) est un CAH de $H : \forall c \in C, \exists E_i \in E, c \subset E_i$. De plus, $\gamma^* = w + 1$, où w est la treewidth de la 2-section de H. Inversement, pour un hypergraphe donné et un CAH de ce dernier, il peut exister plusieurs décompositions arborescentes définies sur les hyperarêtes de ce CAH.

Etant donné un CSP doté d'un CAH de largeur γ, la complexité en temps des meilleures méthodes structurelles, pour sa résolution, est $O(exp(\gamma))$ alors que la complexité en espace peut être réduite à $O(exp(s))$ où s est la taille de la plus grande intersection $E_i \cap E_j$ (qui correspond à un séparateur) entre hyperarêtes voisines de l'hypergraphe. Dans la suite, étant donné un hypergraphe $H = (X, C)$ et l'un de ses CAH, $H_A = (X, E)$, nous étudierons plusieurs classes de recouvrements acycliques de H_A. Ces recouvrements correspondent aux recouvrements d'hyperarêtes (éléments de E) par d'autres hyperarêtes (plus grandes mais *a priori* moins nombreuses), et qui appartiennent à un hypergraphe défini sur le même ensemble de sommets et qui est acyclique. Dans tous les cas, ces extensions seront définies par rapport à un CAH H_A particulier, appelé *CAH de référence.*

Définition 4.3.2 *L'ensemble des recouvrements d'un CAH $H_A = (X, E)$ d'un hypergraphe $H = (X, C)$ est défini par $\mathcal{CAH}_{H_A} = \{(X, E') \in \mathcal{CAH}(H) : \forall E_i \in E, \exists E'_j \in E' : E_i \subset E'_j\}$.*

Les classes de recouvrements suivantes seront des restrictions successives de cette première classe $\mathcal{CAH}_{H_A}$. Elles utilisent différents critères pour recouvrir les hyperarêtes de H_A. Il n'est pas nécessaire de s'assurer de l'acyclicité au moment du choix de ces critères. Puisque les classes définies sont des restrictions de $\mathcal{CAH}_{H_A}$, leurs éléments sont par définition acycliques. Néanmoins, les critères de recouvrement vont donner naissance à des classes de taille différente,

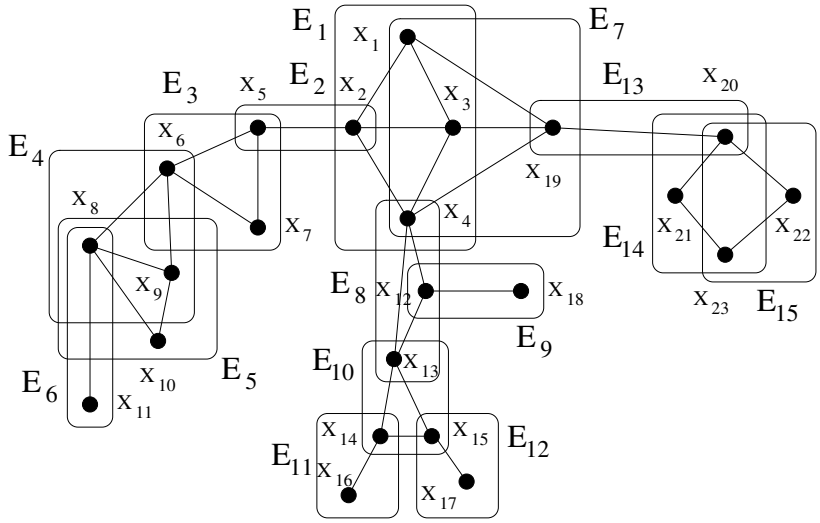

Fig. 4.4 – Un recouvrement par un hypergraphe acyclique d'un graphe.

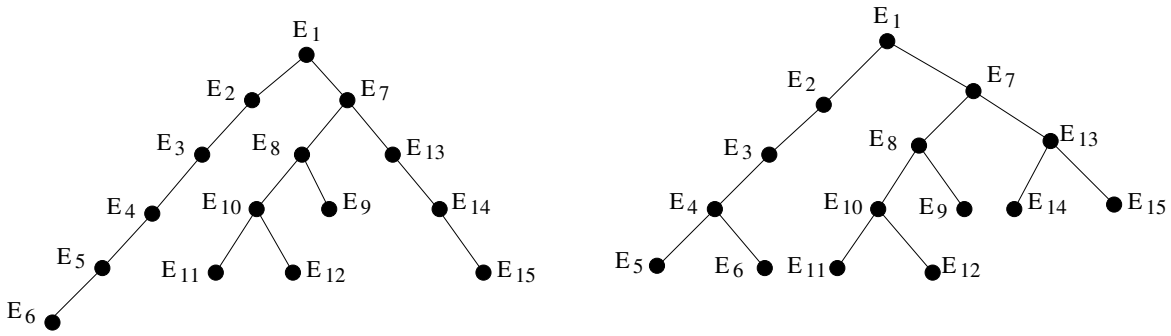

Fig. 4.5 – Deux arbres possibles de décompositions arborescentes définies sur les hyperarêtes de l'hypergraphe acyclique de la figure 4.4.

voire même des classes vides qui n'ont pas d'intérêt. De ce fait, nous allons définir des classes basées sur des critères capables entre autres, de préserver l'acyclicité. La première restriction sur l'ensemble des recouvrements $\mathcal{CAH}_{H_A}$ impose que les hyperarêtes E_i recouvertes (éventuellement partiellement) par une même hyperarête E'_j forme un ensemble α-connexe dans H_A, le but étant d'éviter la création de α-cycles. Cette classe est appelée *ensemble des recouvrements-α-connexes d'un CAH* et sera notée $\mathcal{CAH}_{H_A}[C^+]$.

Définition 4.3.3 $\mathcal{CAH}_{H_A}[C^+] = \{(X, E') \in \mathcal{CAH}_{H_A} : \forall E'_i \in E', E'_i \subset E_{i_1} \cup E_{i_2} \cup \ldots \cup E_{i_R}$ *avec* $E_{i_j} \in E$ *et* $\forall E_{i_u}, E_{i_v}, 1 \leq u < v \leq R$, *il existe une α-chaîne reliant* E_{i_u} *et* E_{i_v} *dans* H_A *définie sur les hyperarêtes appartenant à* $\{E_{i_1}, E_{i_2}, \ldots E_{i_R}\}\}$.

Si $\forall E'_i \in E', E'_i = E_{i_1} \cup E_{i_2} \cup \ldots \cup E_{i_R}$, *la classe est notée* $\mathcal{CAH}_{H_A}[C]$.

Il est possible de la restreindre en conditionnant la nature de l'ensemble $\{E_{i_1}, E_{i_2}, \ldots E_{i_R}\}$. D'une part, on peut se limiter à un ensemble qui définit une α-chaîne : classe des *recouvrements-α-chaînes d'un CAH* notée $\mathcal{CAH}_{H_A}[P^+]$.

Définition 4.3.4 $\mathcal{CAH}_{H_A}[P^+] = \{(X, E') \in \mathcal{CAH}_{H_A} : \forall E'_i \in E', E'_i \subset E_{i_1} \cup E_{i_2} \cup \ldots \cup E_{i_R}$ *avec* $E_{i_j} \in E$ *et* $(E_{i_1}, E_{i_2}, \ldots, E_{i_R})$ *est une α-chaîne dans* $H_A\}$.

Si $\forall E'_i \in E', E'_i = E_{i_1} \cup E_{i_2} \cup \ldots \cup E_{i_R}$, *la classe est notée* $\mathcal{CAH}_{H_A}[P]$.

D'autre part, on peut limiter la distance (le nombre d'hyperarêtes α-voisines) qui sépare les hyperarêtes recouvertes (classe des *recouvrements-famille d'un CAH* notée $\mathcal{CAH}_{H_A}[F^+]$). On autorise ainsi le recouvrement, dans une même hyperarête, d'un ensemble d'hyperarêtes dans lequel un des éléments est α-voisin de tous les autres (une famille).

Définition 4.3.5 $\mathcal{CAH}_{H_A}[F^+] = \{(X, E') \in \mathcal{CAH}_{H_A} : \forall E'_i \in E', E'_i \subset E_{i_1} \cup E_{i_2} \cup \ldots \cup E_{i_R}$ *avec* $E_{i_j} \in E$ *et* $\exists E_{i_u} \in E, \forall E_{i_v}, 1 \leq v \leq R, E_{i_u}$ *et* E_{i_v} *sont* α*-voisines* $\}$.
Si $\forall E'_i \in E', E'_i = E_{i_1} \cup E_{i_2} \cup \ldots \cup E_{i_R}$, *la classe est notée* $\mathcal{CAH}_{H_A}[F]$.

On peut aussi définir une classe (appelée *recouvrements-uniques d'un CAH* et notée $\mathcal{CAH}_{H_A}[U^+]$) qui impose le recouvrement d'une hyperarête E_i par une unique hyperarête de E'_j. L'objectif est d'éviter un recouvrement avec des hyperarêtes qui partagent beaucoup de variables. Cela pourrait accroître considérablement la taille des intersections et donc la complexité spatiale au niveau des méthodes de résolution, mais également la largeur du recouvrement.

Définition 4.3.6 $\mathcal{CAH}_{H_A}[U^+] = \{(X, E') \in \mathcal{CAH}_{H_A} : \forall E_i \in E, \exists! E'_j \in E' : E_i \subset E'_j\}$.
Si $\forall E'_i \in E', E'_i = E_{i_1} \cup E_{i_2} \cup \ldots \cup E_{i_R}$, *la classe est notée* $\mathcal{CAH}_{H_A}[U]$.

Finalement, il est possible d'étendre la classe $\mathcal{CAH}_{H_A}$ dans une autre direction (classe des *recouvrements-proches d'un CAH* notée $\mathcal{CAH}_{H_A}[B^+]$), en n'assurant ni la connexité, ni l'unicité : on peut recouvrir des hyperarêtes dont l'intersection est vide mais qui possèdent une α-voisine commune (des frères). Cette restriction aux recouvrements de frères assure qu'on ne crée pas de α-cycles.

Définition 4.3.7 $\mathcal{CAH}_{H_A}[B^+] = \{(X, E') \in \mathcal{CAH}_{H_A} : \forall E'_i \in E', E'_i \subset E_{i_1} \cup E_{i_2} \cup \ldots \cup E_{i_R}$ *avec* $E_{i_j} \in E$ *et* $\exists E_k \in E$ *tel que* $\forall E_{i_v}, 1 \leq v \leq R$ $E_k \neq E_{i_v}$ *et* E_k *et* E_{i_v} *sont* α*-voisines* $\}$.
Si $\forall E'_i \in E', E'_i = E_{i_1} \cup E_{i_2} \cup \ldots \cup E_{i_R}$, *la classe est notée* $\mathcal{CAH}_{H_A}[B]$.

Les classes $\mathcal{CAH}_{H_A}[X^+]$ autorisent le recouvrement partiel d'hyperarêtes contrairement aux classes $\mathcal{CAH}_{H_A}[X]$.
Nous avons déjà rappelé dans l'introduction, le caractère essentiel des séparateurs dans les méthodes exploitant la structure. Nous tenons compte de cela, en imposant une nouvelle restriction de $\mathcal{CAH}_{H_A}$ (classe des *recouvrements-basés-séparateur d'un CAH* notée $\mathcal{CAH}_{H_A}[S]$). Le but est de rendre possible la limitation des séparateurs à un sous-ensemble de ceux qui existent dans l'hypergraphe de référence.

Définition 4.3.8 $\mathcal{CAH}_{H_A}[S] = \{(X, E') \in \mathcal{CAH}_{H_A} : \forall E'_i, E'_j \in E', i \neq j, \exists E_k, E_l \in E, k \neq l :$
$E'_i \cap E'_j = E_k \cap E_l\}$.

Cette classe impose l'unicité du recouvrement d'une hyperarête et constitue donc une sous-classe $\mathcal{CAH}_{H_A}[U]$

Théorème 4.3.1 $\mathcal{CAH}_{H_A}[S] \subset \mathcal{CAH}_{H_A}[U]$

Preuve Soit H'_A un hypergraphe recouvrant de $\mathcal{CAH}_{H_A}[S]$. On suppose que $H'_A \notin \mathcal{CAH}_{H_A}[U]$ alors $\exists E_i \in E, \exists E'_{j_1} \in E', \exists E'_{j_2} \in E' : E'_{j_1} \neq E'_{j_2}, E_i \subset E'_{j_1}$ et $E_i \subset E'_{j_2}$. Donc $E_i \subset E'_{j_1} \cap E'_{j_2}$. Or, E_i n'est pas une intersection entre hyperarêtes de H_A et donc $H'_A \notin \mathcal{CAH}_{H_A}[S]$. Ceci étant impossible, on a $H'_A \in \mathcal{CAH}_{H_A}[U]$.□

Notons que cette classification n'est pas exhaustive. Nous pourrions considérer d'autres classes, mais dont l'intérêt (et par voie de conséquence l'étude) serait parfois limité. Notons pour finir que le calcul d'un élément de ces différentes classes est facile en termes de complexité. Par exemple, étant donnés H et H_A, nous pouvons calculer $H'_A \in \mathcal{CAH}_{H_A}[S]$ en fusionnant des arêtes α-voisines dans H.

Dans tous les cas, on constate que la valeur de la largeur des hypergraphes s'accroît à l'intérieur de ces classes : $\forall H'_A \in \mathcal{CAH}_{H_A}[X^+]$, la largeur de H'_A est supérieure ou égale à celle

de H_A car $\forall E_i \in E, \exists E'_j \in E'$ tel que $E_i \subset E'_j$. Dès lors, la taille des hyperarêtes ne peut que croître dans les classes que nous avons définies. Pour les classes $\mathcal{CAH}_{H_A}[X^+]$, cet accroissement est assimilable à un accroissement additif :

Théorème 4.3.2 *$\exists \Delta \geq 0$ tel que $\forall H'_A \in \mathcal{CAH}_{H_A}[X^+]$ avec $X = C, P, F, U, B$, $\gamma_{H'_A} \leq \gamma_{H_A} + \Delta$, où $\gamma_{H'_A}$ est la CAH-largeur de H'_A.*

Preuve Le nombre d'hyperarêtes de H_A étant fini, il existe un nombre fini de recouvrements de H_A. Ainsi, $\mathcal{CAH}_{H_A}[X^+]$ contient un nombre fini d'éléments dans tous les cas ($X = C, P, F, U, B$). On pose $\gamma_{max} = Max_{H'_A \in \mathcal{CAH}_{H_A}[X^+]} \gamma_{H'_A}$. γ_{max} existe car $\mathcal{CAH}_{H_A}[X^+]$ contient un nombre fini d'éléments. Soit $\Delta = \gamma_{max} - \gamma_{H_A}$. H'_A étant un recouvrement de H_A, $\gamma_{H_A} \leq \gamma_{H'_A}$ et donc $\Delta \geq 0$. Puisque $\gamma_{max} = \gamma_{H_A} + \Delta$ et que $\forall H'_A \in \mathcal{CAH}_{H_A}[X^+]$, $\gamma_{H'_A} \leq \gamma_{max}$, alors $\gamma_{H'_A} \leq \gamma_{H_A} + \Delta$.□

Concernant les autres classes, l'accroissement est multiplicatif. En effet, dans chaque cas, le recouvrement est relatif à la fusion d'arêtes de (X, E) :

Théorème 4.3.3 *$\exists \delta \geq 1$ tel que $\forall H'_A \in \mathcal{CAH}_{H_A}[C]$ tel que $\gamma_{H'_A} \leq \delta(\gamma - s^-) + s^-$, où s^- est la taille minimum des séparateurs de H_A.*

Preuve $\mathcal{CAH}_{H_A}[C]$ contient un nombre fini d'éléments. On pose :
$\delta = Max_{E'_i \in H'_A, H'_A \in \mathcal{CAH}_{H_A}[C]} R_{E'_i}$, où $R_{E'_i}$ est le nombre d'hyperarêtes de H_A recouvertes par E'_i ($E'_i \subset E_{i_1} \cup E_{i_2} \cup \ldots \cup E_{i_{R_{E'_i}}}$). $\delta \geq 1$ car $R_{E'_i} \geq 1$. En outre, δ existe car $\mathcal{CAH}_{H_A}[C]$ contient un nombre fini d'éléments et $\forall E'_i \in H'_A, H'_A \in \mathcal{CAH}_{H_A}[C]$, $R_{E'_i}$ est fini car le nombre d'hyperarêtes de H_A est fini. La taille d'une hyperarête E'_i d'un élément de $\mathcal{CAH}_{H_A}[C]$ est bornée par la taille d'une hypothétique hyperarête, E'_{max} qui contiendrait δ hyperarêtes de H_A de taille maximale (γ_{H_A}). Les éléments de E'_{max} étant connexes, la valeur $\delta \times \gamma_{H_A}$ comptabilise deux fois la taille des intersections entre ces hyperarêtes. Le nombre d'intersections dans ce cas est $\delta - 1$. La taille de E'_i est bornée ainsi par $\delta(\gamma_{H_A} - s^-) + s^-$ et donc $\gamma_{H'_A} \leq \delta(\gamma_{H_A} - s^-) + s^-$.□

Pour les classes $\mathcal{CAH}_{H_A}[U]$ et $\mathcal{CAH}_{H_A}[B]$, les hyperarêtes peuvent être déconnectées et par conséquent avec des intersections vides. Dans ce cas, nous ne pourrons pas prendre en compte la taille des séparateurs :

Théorème 4.3.4 *$\forall H'_A \in \mathcal{CAH}_{H_A}[U] \cup \mathcal{CAH}_{H_A}[B], \exists \delta \geq 1$ tel que $\gamma_{H'_A} \leq \delta \gamma_{H_A}$.*

Preuve La preuve de ce théorème est similaire au précédent. Le seul changement est qu'il n'est plus possible de soustraire la taille des séparateurs car les hyperarêtes peuvent être déconnectées.□

Ces remarques sont utiles parce qu'elles ont des conséquences sur la complexité des algorithmes qui exploiteront ces recouvrements. Elles illustrent en particulier le fait que les classes $\mathcal{CAH}_{H_A}[C]$ (et donc $[P]$, $[F]$ et $[S]$) doivent être privilégiées, au détriment de classes telles que $\mathcal{CAH}_{H_A}[U]$ ou $\mathcal{CAH}_{H_A}[B]$. Concernant la taille des séparateurs, on peut observer que pour la classe $\mathcal{CAH}_{H_A}[S]$, la valeur s associée à H_A constitue une borne supérieure pour tout hypergraphe H'_A considéré. Formellement :

Théorème 4.3.5 *$\forall H'_A \in \mathcal{CAH}_{H_A}[S], s' \leq s$.*

Preuve $s = Max_{E_i, E_j \in H_A} |E_i \cap E_j|$. s existe car H_A contient un nombre fini d'hyperarêtes. $s' = Max_{E'_k, E'_l \in H'_A} |E'_k \cap E'_l|$, s' existe également car H'_A contient un nombre fini d'hyperarêtes. Puisque $H'_A \in \mathcal{CAH}_{H_A}[S]$, $\forall E'_k, E'_l \in H'_A, k \neq l, E'_k \cap E'_l \neq \emptyset$, $\exists E_i, E_j \in H_A, i \neq j, E'_k \cap E'_l = E_i \cap E_j$. Donc, $s' \leq s$.□

Cette étude nous indique les classes les plus prometteuses. D'un point de vue théorique, il semble que la classe $\mathcal{CAH}_{H_A}[S]$ devrait être la plus utile, à condition de limiter la taille des CAH-largeurs induites.

Dans la suite, nous allons exploiter ces concepts au niveau algorithmique. Aussi, chaque CAH sera maintenant doté d'une arête privilégiée - la racine - à partir de laquelle les différentes

résolutions devront démarrer. Aussi, les connexions entre arêtes de l'hypergraphe seront orientées. Ainsi, certains concepts introduits plus haut seront maintenant exprimés par des mots tels que "hyperarête père", "hyperarête fils" ou "hyperarête frère" comme c'est le cas dans les arborescences.

4.4 Exploitation Algorithmique des CAHs : BDH

Nous proposons ici une extension de BTD, appelée *BDH* (pour "Backtracking sur des recouvrements Dynamiques par Hypergraphes acycliques") et qui est donc basée sur une exploitation dynamique des CAH. Cette approche va rendre possible l'intégration d'heuristiques d'ordonnancement des variables qui seront plus dynamiques. De telles heuristiques sont nécessaires pour s'assurer une résolution pratique efficace. Afin de faciliter l'implémentation et de garantir des bornes de complexité intéressantes, tant pour le temps que pour l'espace, nous considérerons uniquement des hypergraphes recouvrants appartenant à $\mathcal{CAH}_{H_A}[S]$, pour lesquels $H_A = (X, E)$ est l'hypergraphe de référence qui recouvre l'hypergraphe de contraintes H.

Dans les faits, BDH permet d'utiliser tous les arbres de l'ensemble des décompositions arborescentes définies sur les hyperarêtes de H_A. Le choix arbitraire de l'un de ses arbres pour BTD n'étant pas justifié, nous souhaitons construire un arbre pendant la recherche en se basant sur les caractéristiques et les évolutions du problème durant la résolution. Il est également possible de ne pas se restreindre à cet unique arbre mais de calculer un arbre adapté à chaque stade de la résolution. Pour un arbre donné, l'heuristique de fusion va regrouper des clusters voisins dans cet arbre de manière incrémentale. Cela va donner naissance à un hypergraphe de $\mathcal{CAH}_{H_A}[S]$.

4.4.1 Construction incrémentale des arbres

Dans la suite, nous allons faire un amalgame entre les nœuds de l'arbre d'une décomposition arborescente et les clusters de cette décomposition auxquels ils sont associés.

A chaque étape de la résolution, BDH utilise un arbre $T_c = (N_c, A_c)$ (dit courant) d'une décomposition arborescente construite sur les hyperarêtes de H_A. T_c est construit de manière incrémentale. Au départ, le sous-arbre courant T_{c_0} de T_c est vide : il ne contient aucun sommet. En premier lieu, nous considérons une hyperarête dite racine d'où va débuter la construction de T_c, de même que la recherche. Soit E_1 cette hyperarête racine. Le nœud E_1 est ajouté à T_{c_0} et donne un nouvel sous-arbre T_{c_1} de T_c.

A chaque étape, il faut déterminer l'ensemble des voisins d'un nœud E_i de $T_{c_a} = (N_{c_a}, A_{c_a})$ et les rajouter à ce dernier pour construire le sous-arbre suivant $T_{c_{a+1}}$. Ce sera un sous-ensemble des α-voisins de E_i dans H_A. En effet, si un α-voisin de E_i n'est pas α-voisin d'un autre α-voisin de E_i, il est rajouté aux voisins de E_i dans $T_{c_{a+1}}$. Sinon, plusieurs possibilités d'arbres existent. Soient E_j et E_k deux α-voisins de E_i qui sont eux-même α-voisins. Si l'intersection entre deux éléments parmi ces trois hyperarêtes contient strictement les deux autres intersections, l'arête entre ces deux éléments est nécessaire dans $T_{c_{a+1}}$. C'est la seule qui soit nécessaire. Si, elle relie E_j et E_k, alors il y aura une seule arête reliant E_i avec E_j ou E_k. Si les 3 intersections sont égales alors il faut choisir deux arêtes parmi les trois possibles pour relier E_i, E_j et E_k sachant qu'au moins une d'entre elles doit relier E_i avec E_j ou E_k. Par exemple, on peut avoir une arête entre E_i et E_j et une autre entre E_i et E_k ou entre E_j et E_k. Nous allons maintenant définir des ensembles d'hyperarêtes qui vont nous permettre de calculer les voisins de E_i dans $T_{c_{a+1}}$. On pose :

- $N^*(E_i) = \{E_j \in E |\ E_j \notin N_{c_a},\ E_i \textit{ et } E_j \textit{ sont } \alpha\textit{-voisines}\}$ est l'ensemble des α-voisins de E_i, qui contient ses voisins dans $T_{c_{a+1}}$,
- $M^*(E_i) = \{E_j \in N^*(E_i) |\ \exists E_k \in N^*(E_i) \cap N^*(E_j)\}$ est un ensemble dont les éléments du complémentaire dans $N^*(E_i)$ sont des voisins obligatoires de E_i dans $T_{c_{a+1}}$,
- $K^*(E_i) = \{E_j \in N^*(E_i) |\ \exists E_k \in N^*(E_i) \cap N^*(E_j) \textit{ tel que } E_i \cap E_k \subsetneq E_i \cap E_j\}$ est un ensemble de voisins obligatoires de E_i dans $T_{c_{a+1}}$,

- $L^*(E_i) = \{\{E_j, E_k\} \in N^*(E_i) \times N^*(E_i) |\ E_j \in N^*(E_k)$ *et* $E_i \cap E_j \subsetneq E_j \cap E_k\}$ est un ensemble de paires d'hyperarêtes qui ne peuvent pas être simultanément voisines de E_i dans $T_{c_{a+1}}$,
- $I^*(E_i) = \{E_j \in N^*(E_i) |\ \exists \{E_j, E_k\} \in L^*(E_i),\ \{E_i, E_k\} \in A_{c_a}\}$ est un ensemble d'hyperarêtes qui ne peuvent pas être voisines de E_i dans $T_{c_{a+1}}$.

Grâce à ces ensembles, nous pouvons définir un premier ensemble d'hyperarêtes qui seront obligatoirement voisines de E_i : $O^*(E_i) = (N^*(E_i) \backslash M^*(E_i)) \cup K^*(E_i)$. Il faut directement rajouter tous les éléments de $O^*(E_i)$ dans T_{c_a}, de même qu'une arête entre E_i et chacun de ces derniers. On obtient ainsi un nouveau sous-arbre $T_{c_{a+1}}$ de T_c.
L'ensemble $P^*(E_i) = N^*(E_i) \backslash (O^*(E_i) \cup I^*(E_i))$ contient quant à lui des hyperarêtes qui pourraient être voisines de E_i dans $T_{c_{a+1}}$. A ce niveau, un choix heuristique est opéré. En outre, une mise à jour des ensembles $I^*(E_i)$ et $P^*(E_i)$ est nécessaire à chaque ajout d'arête dans $T_{c_{a+1}}$ entre E_i et un élément constitutif d'une paire de $L^*(E_i)$.

Dans l'algorithme BDH, la fonction *Construire* calcule de manière incrémentale l'arbre T_c. A chaque étape, *Construire* rajoute toutes les hyperarêtes de $O^*(E_i)$ dans le sous-arbre courant T_{c_a} et les relie à E_i, puis fait un choix heuristique parmi les éléments de $P^*(E_i)$. On obtient ainsi un nouveau sous-arbre $T_{c_{a+1}}$. Si $O^*(E_i) = \emptyset$, il faut au moins choisir un élément de $P^*(E_i)$ pour sauvegarder la connexité de l'arbre. Une fois que *Construire* a décidé de la sorte de l'ensemble des fils de E_i. L'heuristique de fusion va choisir de manière incrémentale ceux qui seront fusionnés avec E_i. Soit E_j le premier fils choisi pour être fusionné avec E_i. Elle commence par calculer les fils de E_j dans l'arbre comme ce fût le cas pour E_i. Ensuite, E_i et E_j sont fusionnées et l'ensemble des fils de cette nouvelle hyperarête est la réunion des fils de E_i, excepté E_j, et ceux de E_j. L'heuristique de fusion choisit parmi toutes ces hyperarêtes, la prochaine à fusionner avec cette hyperarête. Une fois la nouvelle hyperarête $E'_{i'}$ construite, elle est totalement instanciée. Ensuite, BDH choisit un de ses fils et recommence la même opération.

L'opération de fusion transforme le sous-arbre $T_{c_{a+1}}$ en un autre sous-arbre $T_{cf_{b+1}}$ qui contient les nouvelles hyperarêtes $(E'_{i'})$. On dit que $T_{cf_{b+1}}$ est associé à $T_{c_{a+1}}$ et, de même, T_{cf} est associé à T_c.

L'ordre d'affectation des variables est induit par T_{cf} (plus précisément par les $T_{cf_{b+1}}$), mais la construction de ce dernier est basée sur T_c.

Nous allons montrer que le graphe ainsi calculé par *Construire* est un arbre d'une décomposition arborescente dont les clusters recouvrent les hyperarêtes de H_A.

Proposition 4.4.1 (E, T_c) *est une décomposition arborescente de la 2-section de* H.

Preuve Il faut montrer que les éléments de E recouvrent les hyperarêtes de H, que T_c est un arbre et que l'intersection entre deux clusters est incluse dans tous les clusters situé sur la chaîne entre eux.

Premièrement, E recouvrent les hyperarêtes de H par définition du recouvrement acyclique d'hypergraphe $H_A = (X, E)$.

Deuxièmement, T_c est un arbre parce qu'il est connexe et acyclique.
T_c est connexe car chaque nœud est relié à son père qui est lui-même relié à son père et ainsi de suite jusqu'à la racine. Tous les nœuds sont donc mutuellement accessibles en passant par leur ancêtre commun.
T_c est acyclique parce qu'à chaque étape, l'hyperarête courante est reliée uniquement à ses fils. La seule manière de créer un cycle est de retrouver une même hyperarête dans deux descendances d'hyperarêtes différentes, ce qui est impossible car ces dernières sont disjointes par construction. Donc T_c est un arbre.

Troisièmement, l'intersection entre deux clusters est incluse dans tous les clusters situé sur la chaîne entre eux.
Soient E_i et E_j deux nœuds de l'arbre. Il existe une unique chaîne les reliant :
$(E_i, E_{p_1}, \ldots, E_{p_R}, E_j)$. Il faut montrer que :

$E_i \cap E_j \subset E_i \cap E_{p_1}, E_i \cap E_j \subset E_{p_1} \cap E_{p_2}, \ldots, E_i \cap E_j \subset E_{p_R} \cap E_j$. Si $E_i \cap E_j = \emptyset$, c'est le cas. Si $E_i \cap E_j \neq \emptyset$, on va regarder les différents cas.

Premier cas : E_i et E_j sont α-voisins dans H_A. Supposons que E_i soit le premier à être rajouté dans T_c. E_j est voisin de E_i dans T_c à moins qu'il existe E_{k_1} α-voisin dans H_A de E_i et E_j, voisin de E_i dans T_c et E_j est dans le sous-arbre enraciné en E_{k_1}. En effet, soit E_j est voisin de E_{k_1} dans T_c à moins qu'il existe E_{k_2} α-voisin dans H_A de E_{k_1} et E_j, voisin de E_{k_1} dans T_c et E_j est dans le sous-arbre enraciné en E_{k_2}. La suite étant finie, il existe E_{k_R} qui est voisin de E_j dans T_c. $E_i \cap E_j$ est incluse dans tous les éléments de cette suite (E_{k_u}). En effet, E_i, E_j et E_{k_1} sont tous mutuellement α-voisins. E_i et E_{k_1} étant voisins dans T_c, $E_i \cap E_j \subset E_i \cap E_{k_1}$: $E_i \cap E_j$ est incluse dans E_{k_1}. Ensuite, à l'étape p, E_{k_p}, $E_{k_{p+1}}$ et E_j sont tous mutuellement α-voisins. E_{k_p} et $E_{k_{p+1}}$ étant voisins dans T_c, $E_j \cap E_{k_p} \subset E_{k_p} \cap E_{k_{p+1}}$. Or, $E_i \cap E_j \subset E_j \cap E_{k_p}$, donc $E_i \cap E_j \subset E_{k_p} \cap E_{k_{p+1}}$: $E_i \cap E_j$ est incluse dans $E_{k_{p+1}}$.

Second cas : E_i et E_j ne sont pas α-voisins dans H_A. Il existe alors une α-chaîne qui les relie et dont les intersections contiennent strictement $E_i \cap E_j$ (théorème 4.2.4). Soit E_{k_1} le premier élément de la α-chaîne rajoutée à T_c. L'élément qui suit E_{k_1} dans la sous-α-chaîne qui le relie à E_i est son α-voisin. D'après le premier cas, il existe une chaîne dans l'arbre qui les relie et dont les éléments contiennent leur intersection et donc $E_i \cap E_j$. On peut poursuivre le même raisonnement sur tous les éléments de la α-chaîne jusqu'à E_i. On aura donc construit dans T_c une chaîne qui relie E_{k_1} et E_i et dont les éléments contiennent $E_i \cap E_j$. On procéde de la même manière entre E_{k_1} et E_j. Au final, on a construit un chaîne dans T_c reliant E_i et E_j et dont les éléments contiennent leur intersection.

Par ailleurs, il existe une unique chaîne reliant E_i et E_j dans tous les cas parce que T_c est un arbre. Donc, $E_i \cap E_j$ est incluse dans tous les clusters sur la chaîne qui relie E_i et E_j.□

L'étape de fusion d'hyperarêtes transforme la décomposition arborescente en une nouvelle dont les clusters recouvrent ceux de (E, T_c).

Proposition 4.4.2 *(E', T_{cf}) est une décomposition arborescente de la 2-section de H.*

Preuve T_{cf} est construit à partir de T_c qui est un arbre, en fusionnant une hyperarête E_i avec un de ses fils E_j dans T_c. Ensuite, les fils de E_j dans T_c deviennent ceux de la nouvelle hyperarête $E'_{i'}$. Cette opération est répétée plusieurs fois.

Nous allons montré que cette opération préserve les propriétés essentielles de la décomposition arborescente de départ (E, T_c)

Tout d'abord, les éléments de E' recouvrent les hyperarêtes de H car ils recouvrent les éléments de E qui vérifient déjà cette condition.

Ensuite, l'opération de fusion ne remet pas en cause l'acyclicité, ni la connexité de T_c. Donc, T_{cf} est un arbre.

Pour finir, l'intersection entre deux clusters de T_{cf} est incluse dans tous les clusters situé sur la chaîne entre eux.
Soient $E'_{i'}$ et $E'_{j'}$ deux nœuds de l'arbre. Ils recouvrent des nœuds de T_c. $E'_{i'} \cap E'_{j'}$ est constituée par l'union des intersections entre une hyperarête E_i recouverte par $E'_{i'}$ et une E_j recouverte par $E'_{j'}$. En outre, il existe une unique chaîne CH' les reliant dans T_{cf}. Les éléments de cette chaîne recouvrent d'autres nœuds de T_c. Vu que les hyperarêtes fusionnées forment un ensemble connexe dans T_c, tous les nœuds de ce dernier recouverts dans CH' forment un ensemble connexe dans T_c. De ce fait, il existe une chaîne CH d'hyperarêtes recouvertes par les éléments de CH' qui contient $E_i \cap E_j$. On en déduit que $E_i \cap E_j$ est incluse dans tous les éléments de CH'. Donc, $E'_{i'} \cap E'_{j'}$ est également incluse dans les éléments de CH'.

On montre ainsi que l'intersection entre deux clusters de T_{cf} est incluse dans tous les clusters situé sur la chaîne entre eux.

Donc, (E', T_{cf}) est une décomposition arborescente de la 2-section de H.□

L'hypergraphe (X, E') induit par (E', T_{cf}) est un élément $\mathcal{CAH}_{H_A}[C]$ par construction (on ne recouvre que des ensembles d'hyperarêtes connectées). De même, il appartient à $\mathcal{CAH}_{H_A}[U]$

car par définition *Construire* respecte l'unicité de l'hyperarête recouvrante. Mais, le résultat le plus important est que (X, E') est un élément de $\mathcal{CAH}_{H_A}[S]$.

Proposition 4.4.3 *(X, E') est un élément de $\mathcal{CAH}_{H_A}[S]$.*

Preuve Soient E'_k et E'_l deux hyperarêtes de E'. On suppose que $E'_k \cap E'_l \neq \emptyset$.

Si E'_k et E'_l sont deux nœuds voisins de (E', T_{cf}), alors il existe, par construction de (E', T_{cf}), deux nœuds voisins E_i et E_j de (E, T_c) tels que $E_i \subset E'_k$ et $E_j \subset E'_l$.

$E'_k \cap E'_l = E_i \cap E_j$ car il existe un unique chemin entre tout cluster E_u de (E, T_c) recouvert par E'_k et un autre cluster E_v recouvert par E'_l. Ce chemin passe obligatoirement par E_i et E_j. Donc, $E_u \cap E_v$ est contenu dans E_i et E_j : $E_u \cap E_v \subset E_i \cap E_j$. Ainsi, (X, E') est effectivement un élément de $\mathcal{CAH}_{H_A}[S]$.□

4.4.2 Description de l'algorithme BDH

Pour un nœud E_i d'un arbre de décomposition arborescente, $Père(E_i)$ désignera son nœud père et $Fils(E_i)$ l'ensemble de ses fils. La descendance de E_i ($Desc(E_i)$) est l'ensemble des variables des hyperarêtes contenues dans le sous-hypergraphe acyclique enraciné en E_i. On appelle sous-problème enraciné en E_i le sous-problème induit par les variables de $Desc(E_i)$.

BDH possède 5 entrées : $\mathcal{A}$ l'affectation courante, E'_i l'hyperarête courante, $V_{E'_i}$ l'ensemble des variables non affectées de E'_i, H_A l'hypergraphe de référence et T_{cf_b} le sous-arbre courant. Cet arbre est calculé récursivement par la fonction *Construire* pendant la résolution. La construction de E'_i précède toujours l'affectation de ses variables exceptées celles qui se trouvent également dans $Pere(E_i)$. BDH résout récursivement le sous-problème enraciné en E'_i et fournit en résultat *true* si $\mathcal{A}$ peut être étendue de façon consistante sur ce sous-problème et $false$ sinon. Lors de l'appel initial, l'affectation $\mathcal{A}$ est vide, le sous-problème enraciné en E_1 correspond à l'intégralité du problème et T_c ne contient que E_1. De manière analogue à BTD, l'ordre dans lequel les variables sont affectées est partiellement induit par l'arbre courant T_{cf_b}. Tant que $V_{E'_i}$ n'est pas vide, BDH choisit une variable x dans $V_{E'_i}$ (ligne 17) et une valeur dans son domaine (ligne 19) (s'il n'est pas vide). Si cette extension de l'affectation courante satisfait toutes les contraintes (en considérant aussi celles induites par les éventuels nogoods), alors il y a un appel récursif BDH($\mathcal{A} \cup \{x \leftarrow v\}, E'_i,\ V_{E'_i} \backslash \{x\}, H_A, T_{cf_b}$) sur le reste du cluster (ligne 20). Si par contre, l'extension n'est pas consistante, on essaie une nouvelle valeur (s'il en reste). Dans le cas où toutes les valeurs de x ont été essayées, il y a un backtrack sur la variable précédente. Quand toutes les variables dans E'_i sont affectées, l'algorithme choisit un fils E_j du cluster courant (ligne 4) (s'il en existe un). La fonction *Construire* étend la construction de T_{cf_b} en calculant un cluster $E'_{j'}$ recouvrant E_j et en déterminant les fils de $E'_{j'}$. Si $\mathcal{A}[E'_i \cap E'_{j'}]$ est un good (ligne 7), nous savons que $\mathcal{A}$ peut alors être étendue de façon consistante sur $Desc(E'_{j'})$. De même, si E'_i contient un séparateur $s_k = E_{k_u} \cap E_{k_{u'}}$ de H_A, avec $E_{k_{u'}}$ un fils de E_{k_u}, tel que le sous-problème enraciné en $E'_{j'}$ est inclus dans celui enraciné en $E_{k_{u'}}$ et si $\mathcal{A}[s_k]$ est un good (ligne 9), nous savons que $\mathcal{A}$ peut être étendue de façon consistante sur $Desc(E_{k_{u'}})$. Ainsi, $\mathcal{A}$ peut également être étendue de façon consistante sur $Desc(E'_{j'})$. Par conséquent, un forward-jump sera réalisé et l'algorithme poursuivra la résolution sur le reste du problème. Puisque les nogoods sont exploités comme des contraintes, l'affectation $\mathcal{A}$ sera stoppée dans E'_i si elle contient un nogood (ligne 20). Si E'_i ne contient pas de (no)good, alors la résolution se poursuit sur le sous-problème enraciné en $E'_{j'}$. Si $\mathcal{A}$ admet une extension consistante sur ce sous-problème, $\mathcal{A}[E'_i \cap E'_{j'}]$ est enregistrée comme good (ligne 12) et *true* est renvoyé comme résultat, sinon $\mathcal{A}[E'_i \cap E'_{j'}]$ est enregistré comme nogood (ligne 13) et $false$ est renvoyé. Si BDH échoue lors de l'extension de $\mathcal{A}$ sur E'_i alors, il renvoie $false$.

Cette extension de BTD procure plusieurs avantages. Elle permet d'élargir l'éventail des heuristiques éligibles grâce à la liberté donnée par une exploitation plus souple de la structure H_A de départ. Ainsi, la prochaine variable à instancier ne sera pas forcément choisie dans un unique cluster. Cela n'empêche en rien d'exploiter les nogoods déjà enregistrés, de même qu'une

Algorithme 12 : BDH($\mathcal{A}, E'_i, V_{E'_i}, H_A, T_{cf_b}$)

```
1  if V_{E'_i} = ∅ then
2      Cons ← true ; F ← Fils(E'_i)
3      while F ≠ ∅ and Cons do
4          E_j ← Choix-hyperarete(F)
5          F ← F\{E_j}
6          E'_{j'} ← Construire(T_{cf_b}, H_A, E_j)
7          if A[E'_{j'} ∩ E'_i] est un good then Cons ← true
8          else
9              if ∃ s_k = E_{k_u} ∩ E_{k_u'} dans T_{cf_b} tel que
               E_{k_u'} ∈ Fils(E_{k_u}), s_k ⊂ E'_i, E'_{j'} ⊂ Desc(E_{k_u'}) et A[s_k] est un good then
               Cons ← true
10             else
11                 Cons ← BDH(A, E'_{j'}, E'_{j'}\(E'_{j'} ∩ E'_i), H_A, T_{cf_{b+1}})
12                 if Cons then Enregistrer le good A[E'_{j'} ∩ E'_i]
13                 else Enregistrer le nogood A[E'_{j'} ∩ E'_i]
14     if Cons then ∀E_u ∩ E_v ⊂ E'_i, enregistrer le good A[E_u ∩ E_v]
15     return Cons
16 else
17     x ← Choix-var(V_{E'_i}) ; d_x ← D_x ; Cons ← false
18     while d_x ≠ ∅ and ¬Cons do
19         v ← Choix-val(d_x) ; d_x ← d_x\{v}
20         if A ∪ {x ← v} ne viole aucune contrainte ou nogood then
           Cons ← BDH(A ∪ {x ← v}, E'_i, V_{E'_i}\{x}, H_A, T_{cf_b})
21     return Cons
```

grande partie des goods. La complexité en espace n'est pas modifiée ($O(exp(s))$) du fait que la méthode est basée sur les séparateurs de H_A. Finalement, son implémentation est facilitée par la restriction à $\mathcal{CAH}_{H_A}[S]$. Les hypergraphes de cet ensemble pouvant être obtenus par de simples fusions d'hyperarêtes de H_A.

Théorème 4.4.1 *BDH est correct, complet et termine.*

La preuve de ce théorème est similaire en presque tout point à celle de la correction de BTD. Nous allons donc rappeler quelques résultats préliminaires.

Lemme 4.4.1 *(saut par les goods) [JT03] Soient E_i un cluster, E_j un de ses fils, $Y \subset X$ tel que $Desc(E_j) \cap Y = E_i \cap E_j$. Pour tout good g, de E_i par rapport à E_j, toute instanciation consistante de $\mathcal{A}$ de Y telle que $\mathcal{A}[E_i \cap E_j] = g$ possède une extension consistante sur $Desc(E_j)$.*

Lemme 4.4.2 *(coupe par les nogoods) [JT03] Soient E_i un cluster, E_j un de ses fils, $Y \subset X$ tel que $Desc(E_j) \cap Y = E_i \cap E_j$. Pour tout nogood ng, de E_i par rapport à E_j, il n'existe pas d'affectation $\mathcal{A}$ de Y telle que $\mathcal{A}[E_i \cap E_j] = ng$ et telle que $\mathcal{A}$ possède une extension consistante sur $Desc(E_j)$.*

Nous allons maintenant faire la preuve du théorème.

Preuve On va faire une preuve par induction sur le nombre de variables qu'il faut instancier pour déterminer l'existence ou non d'une extension consistante de $\mathcal{A}$ dans la descendance de E'_i. Notons cet ensemble $AFF(E'_i, V_{E'_i})$: $AFF(E'_i, V_{E'_i}) = V_{E'_i} \cup (\bigcup_{E'_j \in Fils(E'_i)} (Desc(E'_j) - (E'_i \cap E'_j)))$.

Pour démontrer la correction de BDH, nous allons prouver la propriété suivante :
$P(\mathcal{A}, AFF(E'_i, V_{E'_i}))$: BDH$(A, E'_i, V_{E'_i}, H_A, T_{cf_b})$ retourne True si l'affectation consistante $\mathcal{A}$ peut être étendue en une affectation consistante sur $V_{E'_i}$ et la descendance de E'_i, et False sinon.

Si toutes les variables de E'_i sont affectées ($V_{E'_i} = \emptyset$) et que E'_i n'a pas de fils ($Fils(E'_i) = \emptyset$) alors BDH retourne True et $AFF(E'_i, V_{E'_i}) = \emptyset$. Dans ce cas, l'assertion "$\mathcal{A}$ peut être étendue de manière consistante sur $AFF(E'_i, V_{E'_i})$" est vérifiée . $P(\mathcal{A}, \emptyset)$ est vérifiée.

Supposons qu'il existe $S = AFF(E'_i, V_{E'_i}), S \neq \emptyset$ tel que $\forall S' \subsetneq S, P(\mathcal{A}, S')$ soit vérifiée. On va montrer que $P(\mathcal{A}, S)$ est vérifiée.
Si $V_{E'_i} \neq \emptyset$, BDH choisit une nouvelle variable $x \in V_{E'_i}$ à la ligne 17 et lui affecte une valeur, v, de son domaine dans la boucle while des lignes 18-20. Si cette extension $\mathcal{A} \cup \{x \leftarrow v\}$ est consistante (ne viole aucune contrainte et aucun nogood) alors il y a un appel récursif de BDH à la ligne 20 : $BDH(\mathcal{A} \cup \{x \leftarrow v\}, E'_i, V_{E'_i} \backslash \{x\}, H_A, T_{cf_b})$. Or $AFF(E'_i, V_{E'_i} \backslash \{x\}) \subsetneq AFF(E'_i, V_{E'_i})$, donc, en utilisant notre hypothèse d'induction, on peut conclure que $BDH(\mathcal{A} \cup \{x \leftarrow v\}, E'_i, V_{E'_i} \backslash \{x\}, H_A, T_{cf_b})$ retourne True si $\mathcal{A} \cup \{x \leftarrow v\}$ possède une extension consistante sur $V_{E'_i} \backslash \{x\}$ et la descendance de E'_i et False, sinon.
S'il renvoie True alors $BDH(\mathcal{A}, E'_i, V_{E'_i}, H_A, T_{cf_b})$ retourne True car il existe une extension consistante de $\mathcal{A}$ sur $V_{E'_i}$ et la descendance de E'_i. $P(\mathcal{A}, S)$ est vérifiée. Si False est renvoyée, cela veut dire que $\mathcal{A} \cup \{x \leftarrow v\}$ ne possède pas d'extension consistante dans $V_{E'_i}$ et la descendance de E'_i. Alors, BDH choisit une nouvelle valeur dans le domaine de x. Si, par contre, $\mathcal{A} \cup \{x \leftarrow v\}$ viole une contrainte alors cette extension n'est pas consistante. Si, elle viole un nogood alors elle est consistante mais elle ne peut pas conduire à une extension consistante de $\mathcal{A}$ sur S (lemme 4.4.2). Dans les deux cas, BDH choisit une nouvelle valeur. Si toutes les valeurs du domaine courant ont été testées sans succès : aucune ne conduit à une extension consistante de $\mathcal{A}$ sur S alors BDH retourne False. $P(\mathcal{A}, S)$ est vérifiée car il n'existe pas d'extension consistante de $\mathcal{A}$ sur S.

Supposons maintenant que $V_{E'_i} = \emptyset$.

Puisque $AFF(E'_i, V_{E'_i}) \neq \emptyset$ alors $Fils(E'_i) \neq \emptyset$. A la ligne 4, BDH choisit un fils $E_j \in Fils(E'_i)$ et *Construire* calcule un cluster $E'_{j'}$ qui recouvre E_j.

Si $\mathcal{A}[E'_i \cap E'_{j'}]$ est un good (ligne 7) alors il existe une extension consistante de $\mathcal{A}$ sur $Desc(E'_{j'})$ (lemme 4.4.1). Dès lors, la recherche est poursuivie sur $S' = Desc(E'_i) \backslash (E'_i \cup Desc(E'_{j'}))$ (boucle while lignes 3-13).

Si $\mathcal{A}[E'_i \cap E'_{j'}]$ n'est pas un good, mais qu'il existe un séparateur $s_k = E_{k_u} \cap E_{k_{u'}}$ dans E'_i tel que $E'_{j'} \subset Desc(E_{k_{u'}})$ et $\mathcal{A}[s_k]$ est good, alors il existe une extension consistante de $\mathcal{A}$ sur $Desc(E'_{j'})$ (ligne 9). Comme précédemment, la recherche est poursuivie sur $S' = Desc(E'_i) \backslash (E'_i \cup Desc(E'_{j'}))$.

Si $\mathcal{A}[E'_i \cap E'_{j'}]$ n'est pas un good et que $E'_{j'}$ n'est pas contenu dans la descendance d'un good alors il y a un appel récursif de BDH sur $Desc(E'_{j'})$ (ligne 11). Dès lors, il peut y avoir deux cas de figure.

Si $Desc(E'_{j'}) = S$, cela veut dire E'_i n'a qu'un seul fils, $E'_{j'}$.

BDH$(\mathcal{A}, E'_{j'}, E'_{j'} \backslash (E'_{j'} \cap E'_i), H_A, T_{cf_{b+1}})$ va chercher à étendre $\mathcal{A}$ sur $Desc(E'_{j'})$. Puisque, $V_{E'_{j'}} \neq \emptyset$, BDH choisit une variable $y \in V_{E'_{j'}}$ et tente d'étendre $\mathcal{A}$ en affectant y à une valeur de son domaine. Par le même raisonnement que précédemment, on peut montrer que BDH$(\mathcal{A}, E'_{j'}, E'_{j'} \backslash (E'_{j'} \cap E'_i), H_A, T_{cf_{b+1}})$ renvoie True s'il existe une extension consistante de $\mathcal{A}$ sur $Desc(E'_{j'})$ et False, sinon.
On en déduit ainsi que $P(\mathcal{A}, S)$ est vérifiée car BDH$(\mathcal{A}, E'_i, V_{E'_i}, H_A, T_{cf_b})$ renvoie la même valeur.

Si $Desc(E'_{j'}) \subsetneq S$, alors grâce à l'hypothèse d'induction, on sait également que
BDH$(\mathcal{A}, E'_{j'}, E'_{j'} \backslash (E'_{j'} \cap E'_i), H_A, T_{cf_{b+1}})$ renvoie True s'il existe une extension consistante de $\mathcal{A}$ sur $Desc(E'_j)$ et False, sinon. Dans le cas où False est renvoyée, $\mathcal{A}$ ne peut être étendue de manière consistante sur $Desc(E'_j) \subset S$. $BDH(\mathcal{A}, E'_i, V_{E'_i}, H_A, T_{cf_b})$ retourne False car il n'existe pas d'extension consistante de $\mathcal{A}$ sur la descendance de E'_i. $P(\mathcal{A}, S)$ est vérifiée.

Dans le cas où True est renvoyée, la recherche continue sur $S \backslash Desc(E'_j)$. BDH choisit un autre

fils $E_k \in Fils(E'_i)$, calcule un cluster $E'_{k'}$ qui recouvre E_k et tente d'étendre $\mathcal{A}$ sur $Desc(E'_j)$. Cela continue jusqu'à ce que False soit renvoyée pour un fils de E'_i ou que $F = \emptyset$. Dans le premier cas, on montre comme précédemment que $P(\mathcal{A}, S)$ est vérifiée. Dans le second, $\mathcal{A}$ admet une extension consistante sur l'ensemble des descendances des fils de E'_i. Elle peut être étendue de manière consistante sur $Desc(E'_i)$ et BDH($\mathcal{A}, E'_i, V_{E'_i}, H_A, T_{cf_b}$) renvoie True. Donc, $P(\mathcal{A}, S)$ est vérifiée.

On vient de montrer par induction que $\forall S, S = AFF(E'_i, V_{E'_i})$, $P(\mathcal{A}, S)$ est vérifiée. De ce fait, $P(\emptyset, AFF(E'_1, V_{E'_1}))$ est satisfaite. Nous avons prouvé la correction partielle de BDH.
La correction totale est facile à voir. En effet, BDH doit explorer au départ un espace de recherche de taille finie. A chaque étape, cet espace est réduit d'au moins un nœud. BDH est correct, complet et termine.□

BDH utilise un sous-ensemble d'hypergraphes de $\mathcal{CAH}_{H_A}[S]$. D'après le théorème 4.3.2, on sait qu'il existe $\Delta \geq 0$ tel que pour tout H'_A dans ce sous-ensemble, $\gamma' \leq \gamma + \Delta$. La complexité de la méthode dépend évidemment de Δ. En outre, ce facteur est paramétrable : il est possible de fixer la valeur de Δ et considérer uniquement les hypergraphes recouvrants de $\mathcal{CAH}_{H_A}[S]$ dont la largeur est bornée par $\gamma + \Delta$. Dans tous les cas, la complexité de BDH est donnée par le théorème suivant.

Théorème 4.4.2 *La complexité en temps de BDH est $O(nombre_{T_c}.exp(\gamma + \Delta + 1))$, avec $nombre_{T_c}$ le nombre d'arbres T_c utilisés par BDH.*

Preuve Soient $\mathcal{P} = (X, D, C, R)$ un CSP, $H_A = (X, E)$ le CAH de référence de $H = (X, C)$. On considère une décomposition arborescente (E, T_c) de H dont l'arbre T_c a été effectivement construit par BDH.

Nous allons montrer qu'il est possible de recouvrir H_A par des ensembles V_a de $\gamma + \Delta + 1$ variables qui vérifient la condition que toute affectation sur leurs variables ne sera générée par BDH qu'au plus deux fois.

Nous commençons par la définition des ensembles V_a qui contiennent des nœuds de T_c qui forment une chaîne sur une de ses branches.

Soit V_a un ensemble de $\gamma+\Delta+1$ variables tel que $\exists(E_{u_1}, \ldots, E_{u_r})$ une chaîne sur une branche dans T_c, $V_a \subset E_{u_1} \cup \ldots \cup E_{u_r}$ ($r \geq 2$ puisque $|V_a| = \gamma + \Delta + 1$ et γ est la taille maximale des hyperarêtes de H_A) et $E_{u_2} \cup \ldots \cup E_{u_{r-1}} \subsetneq V_a$ (resp. $E_{u_1} \cap E_{u_2} \subset V_a$) si $r \geq 3$ (resp. $r = 2$).

On considère un recouvrement de T_c par un ensemble de V_a. Ce recouvrement peut être construit facilement en recouvrant chaque branche de T_c par des V_a (ils peuvent s'intersecter). Le nombre de V_a est borné par le nombre de nœuds de T_c (le nombre d'hyperarêtes de H_A).

Maintenant, nous allons montrer que toute affectation sur un V_a est générée par BDH au plus deux fois.

Pour chaque T_{cf} associé à T_c, $\exists(E'_{i_1}, \ldots, E'_{i_{r'}})$, une chaîne dans une branche T_{cf}, telle que $V_a \subset E'_{i_1} \cup \ldots \cup E'_{i_{r'}}$ ($r' \geq 2$ puisque la taille maximale des hyperarêtes dans T_{cf} est $\gamma + \Delta$) et $E'_{i_2} \cup \ldots \cup E'_{i_{r'-1}} \subsetneq V_a$ (resp. $E'_{i_1} \cap E'_{i_2} \subsetneq V_a$) si $r' \geq 3$ (resp. $r' = 2$).

Soit $\mathcal{A}$ une affectation à étendre sur V_a. L'ordre dans lequel les variables de V_a seront instanciées est induit par T_{cf}. On supposera dans la suite que les hyperarêtes recouvrant V_a sont ordonnées suivant leur ordre d'affectation : E'_{i_j} est affectée avant $E'_{i_{j'}}$ si $j < j'$. De ce fait E_{u_1} est toujours recouvert par le premier nœud affecté E'_{i_1} parmi celles recouvrant V_a dans T_{cf}. Donc, $s_1 = E_{u_1} \cap E_{u_2}$ est incluse dans E'_{i_1}.

Si $E'_{i_1} \cap E'_{i_2} = s_1$, la résolution du sous-problème enraciné en E'_{i_2} avec l'affectation $\mathcal{A}$ conduit à l'enregistrement d'un (no)good sur le séparateur s_1 : $\mathcal{A}[s_1]$. Soit $\mathcal{B}$ une nouvelle affectation à étendre sur V_a avec les mêmes valeurs que $\mathcal{A}[V_a]$.

On considère T'_{cf} un arbre associé à T_c qui induit l'ordre dans lequel les variables de V_a sont affectées sur $\mathcal{B}$. $\exists(E''_{j_1}, \ldots, E''_{j_{r''}}), r'' \geq 2$, une chaîne dans une branche de T'_{cf} telle que $V_a \subset E''_{j_1} \cup \ldots \cup E''_{j_{r''}}$ et $E''_{j_2} \cup \ldots \cup E''_{j_{r''-1}} \subset V_a$ (resp. $E''_{j_1} \cap E''_{j_2} \subset V_a$) si $r'' \geq 3$ (resp. $r' = 2$). Etant donné que $s_1 \subset E''_{j_1}$, dès que E''_{j_1} est totalement affectée, $\mathcal{A}[s_1]$ arrête l'affectation des

variables de V_a. Alors $\mathcal{A}[V_a]$ est générée une deuxième fois sur uniquement $\gamma + \Delta$ variables de V_a au pire.

On suppose maintenant que $E'_{i_1} \cap E'_{i_2} \neq s_1$.

On suppose tout d'abord que $\mathcal{A}[E'_{i_1} \cap E'_{i_2}]$ peut être étendue de manière consistante sur le sous-problème enraciné en E'_{i_2} alors $\mathcal{A}[E'_{i_1} \cap E'_{i_2}]$ est enregistrée comme un good. Par ailleurs, puisque $s_1 \subset E'_{i_1}$, nous aurons deux cas possibles.

Premier cas, l'affectation courante $\mathcal{A}$ peut être étendue de manière consistante sur les variables non instanciées du sous-problème enraciné en E'_{i_1} alors $\mathcal{A}[s_1]$ est enregistrée comme un good. Dès lors, si on essaie d'étendre $\mathcal{B}$ sur V_a, le good $\mathcal{A}[s_1]$ stoppe l'affectation des variables de V_a dès que E''_{j_1} est totalement affectée. Comme précédemment, $\mathcal{A}[V_a]$ est générée une deuxième fois sur uniquement $\gamma + \Delta$ variables de V_a au pire.

Deuxième cas, $\mathcal{A}$ ne peut être étendue de manière consistante sur au moins un sous-problème enraciné en un fils $E'_{t'}$ de E'_{i_1} différent de E'_{i_2}. Donc $\mathcal{A}[E'_{i_1} \cap E'_{t'}]$ est enregistrée comme un nogood. Néanmoins, il est possible de générer $\mathcal{A}[V_a]$ une deuxième fois si les deux séparateurs $E'_{i_1} \cap E'_{i_2}$ et $E'_{i_1} \cap E'_{t'}$ sont contenus dans le dernier nœud à être affecté à savoir $E''_{j_{r''}}$. Dans ce cas, une information supplémentaire est enregistrée.

Première possibilité, $\mathcal{B}[E''_{j_1} \cap E''_{j_2}]$ peut être étendue de manière consistante sur le sous-problème enraciné en E''_{j_2} alors $\mathcal{B}[E''_{j_1} \cap E''_{j_2}]$ est enregistrée comme un good.

Par ailleurs, puisque $s_1 \subset E''_{j_1}$, soit $\mathcal{B}[s_1]$ est enregistrée comme un good soit $\mathcal{A}[E''_{j_1} \cap E''_{t''}]$ est enregistrée comme un nogood, avec $E''_{t''}$ un fils de E''_{j_1} différent de E''_{j_2}.

Si $\mathcal{B}[s_1]$ est un good alors $\mathcal{B}[V_a]$ est générée à nouveau sur uniquement $\gamma + \Delta$ variables de V_a au pire.

Si $\mathcal{B}[E''_{j_1} \cap E''_{t''}]$ est un nogood : deux nogoods sont enregistrés sur deux séparateurs dans V_a. Dès que le premier de ces deux séparateurs est totalement instancié, le nogood associé à ce séparateur arrête l'affectation sur V_a. De ce fait $\mathcal{A}[V_a]$ n'est pas générée à nouveau.

Seconde possibilité, $\mathcal{B}[E''_{j_1} \cap E''_{j_2}]$ ne peut être étendue de manière consistante sur le sous-problème enraciné en E''_{j_2} alors $\mathcal{B}[E''_{j_1} \cap E''_{j_2}]$ est enregistré comme un nogood. Puisque deux nogoods sont enregistrés, $\mathcal{A}[V_a]$ n'est pas générée à nouveau.

On suppose maintenant que $\mathcal{A}[E'_{i_1} \cap E'_{i_2}]$ ne peut être étendue de manière consistante sur le sous-problème enraciné en E'_{i_2} alors $\mathcal{A}[E'_{i_1} \cap E'_{i_2}]$ est enregistrée comme un nogood.

Si on essaie d'étendre $\mathcal{B}$ sur V_a, il est possible de générer $\mathcal{A}[V_a]$ une deuxième fois si le séparateur $E'_{i_1} \cap E'_{i_2}$ est inclus dans le dernier nœud à être affecté $E''_{j_{r''}}$.

Dans ce cas, nous aurons deux possibilités.

Pour la première, $\mathcal{B}[E''_{j_1} \cap E''_{j_2}]$ peut être étendue de manière consistante sur le sous-problème enraciné en E''_{j_2} alors $\mathcal{B}[E''_{j_1} \cap E''_{j_2}]$ est enregistrée comme un good.

En outre, $\mathcal{B}[s_1]$ est enregistrée soit comme un good, soit il existe un fils $E''_{t''}$ de E''_{j_1} différent de E''_{j_2} tel que $\mathcal{B}[E''_{j_1} \cap E''_{t''}]$ est enregistrée comme un nogood.
Dans les deux cas, $\mathcal{B}[V_a] = \mathcal{A}[V_a]$ n'est pas générée une nouvelle fois.

Deuxième possibilité, $\mathcal{B}[E''_{j_1} \cap E''_{j_2}]$ ne peut être étendue de manière consistante sur le sous-problème enraciné en E''_{j_2} alors $\mathcal{B}[E''_{j_1} \cap E''_{j_2}]$ est enregistrée comme un nogood.
Deux nogoods étant enregistrés, $\mathcal{A}[V_a]$ n'est pas générée une nouvelle fois.

On a prouvé donc que toute affectation sur V_a est générée au pire deux fois.

Ainsi, le nombre d'affectations possibles sur V_a est bornée par $d^{\gamma+\Delta+1}$.
Donc le nombre d'affectations possibles sur l'ensemble des variables du problème est borné par $nombre_{T_c}.nombre_{V_a}.d^{\gamma+\Delta+1}$, avec $nombre_{V_a}$ le nombre maximum d'ensembles V_a recouvrant un T_c. Le nombre de V_a étant borné, la complexité de BDH est en $O(nombre_{T_c}.exp(\gamma + \Delta + 1))$. □

Cette complexité reste bornée par $O((exp(h)))$, avec h le nombre maximum de variables dans une branche d'un arbre T_c, qui est la complexité de la méthode sans apprentissage.

Une amélioration de BDH consiste à utiliser à la place du Backtracking standard, des méthodes plus efficaces telles FC et MAC, à l'instar de BTD. Dans le cadre de nos expérimentations, nous avons privilégié la version FC-BDH.

4.5 Extension de BDH aux VCSP : BDH-val

Nous proposons ici une extension de BDH aux VCSP. A l'image de BDH, cette approche va rendre possible l'intégration d'heuristiques d'ordonnancement des variables plus dynamiques. Le but étant toujours d'assurer un meilleur comportement de la résolution en pratique. De même, nous allons nous restreindre aux hypergraphes recouvrants de la classe $\mathcal{CAH}_{H_A}[S]$, avec H l'hypergraphe de contraintes et H_A l'hypergraphe de référence.

BDH-val possède 7 entrées : $\mathcal{A}$ l'affectation courante, E'_i l'hyperarête courante, $V_{E'_i}$ l'ensemble des variables non affectées de E'_i, $ub_{E'_i}$ le majorant actuel de la valuation optimale du VCSP induit $\mathcal{P}_{\mathcal{A},Pere(E'_i)/E'_i}$, $lb_{E'_i}$ le minorant de la valuation de l'affectation sur $Desc(E'_i)$ en cours de construction, H_A l'hypergraphe de référence et T_{cf_b} le sous-arbre courant.
T_{cf_b} est calculé récursivement de la même manière que dans BDH.
BDH-val résout récursivement les sous-problèmes enracinés en E'_i et fournit en résultat la valuation optimale du CSP induit $\mathcal{P}_{\mathcal{A},Pere(E'_i)/E'_i}$. Lors de l'appel initial, l'affectation $\mathcal{A}$ est vide, le sous-problème enraciné en E_1 correspond à l'intégralité du problème. Tant que $V_{E'_i}$ n'est pas vide et que le minorant est plus petit que le majorant, BDH-val choisit une variable x dans $V_{E'_i}$ (ligne 15) et une valeur dans son domaine (ligne 18) (s'il n'est pas vide), puis met à jour le minorant de l'affectation en cours de construction. Si le minorant dépasse ou égale le majorant, BDH-val change de valeur si possible ou effectue un backtrack. Sinon BDH-val($\mathcal{A} \cup \{x \leftarrow v\}, E'_i, V_{E'_i} \backslash \{x\}, H_A, T_{cf_b}$) est appelé sur le reste de l'hyperarête (ligne 26). Quand toutes les variables dans E'_i sont affectées, l'algorithme choisit un fils E_j de l'hyperarête courante (ligne 5) (s'il en existe un). La fonction *Construire* étend la construction de T_{cf_b} en calculant une hyperarête $E'_{j'}$ recouvrant E_j et en déterminant les fils de $E'_{j'}$. Si $\mathcal{A}[E'_i \cap E'_{j'}]$ est un good valué (ligne 8), nous connaissons la valuation optimale sur $Desc(E'_{j'})$ et celle-ci est directement agrégée au minorant et la recherche se poursuit sur le reste du problème. Si E'_i ne contient pas de good valué, alors la résolution se poursuit sur le sous-problème enraciné en $E'_{j'}$. Une fois la valuation optimale sur $Desc(E'_{j'})$ calculée, elle est enregistrée comme un good valué avec $\mathcal{A}[E'_i \cap E'_{j'}]$ (ligne 12) et renvoyé comme résultat.

La complexité en espace demeure en $O(exp(s))$ puisque nous considérons toujours des hypergraphes recouvrants appartenant à $\mathcal{CAH}_{H_A}[S]$.

Théorème 4.5.1 *BDH-val est correct, complet et termine.*

Lemme 4.5.1 *(coupe par les goods valués) [TJ03] Soient E_i un cluster, E_j un de ses fils, $Y \subset X$ tel que $Desc(E_j) \cap Y = E_i \cap E_j$. Pour tout good valué (g, α), de E_i par rapport à E_j, α est la valuation de l'extension optimale sur $Desc(E_j)$ de toute instanciation $\mathcal{A}$ de Y telle que $\mathcal{A}[E_i \cap E_j] = g$.*

Nous allons maintenant faire la preuve du théorème.

Preuve On va faire une preuve par induction sur le nombre de variables qu'il faut instancier pour déterminer la valuation optimale du VCSP induit $\mathcal{P}_{\mathcal{A},Pere(E'_i)/E'_i}$. Cet ensemble est noté $AFF(E'_i, V_{E'_i})$: $AFF(E'_i, V_{E'_i}) = V_{E'_i} \cup (\bigcup_{E'_j \in Fils(E'_i)} (Desc(E'_j) - (E'_i \cap E'_j)))$. Pour démontrer la correction de BDH-val, nous allons prouver la propriété suivante :
$P(\mathcal{A}, AFF(E'_i, V_{E'_i}), ub_{E'_i}, lb_{E'_i})$: $ub_{E'_i}$ est la valuation de la meilleure affectation $\mathcal{B}'$ connue sur $\mathcal{P}_{\mathcal{A},Pere(E'_i)/E'_i}$ et $lb_{E'_i} = v_{\mathcal{P},E'_i}(\mathcal{A}) \prec ub_{E'_i}$, BDH-val($\mathcal{A}, E'_i, V_{E'_i}, ub_{E'_i}, lb_{E'_i}, H_A, T_{cf_b}$) renvoie :
- la valuation de la meilleure affectation $\mathcal{B}$ sur $Desc(E'_i)$ telle que $\mathcal{B}[E'_i \backslash V_{E'_i}] = \mathcal{A}[E'_i \backslash V_{E'_i}]$ et $v_{\mathcal{P}_{\mathcal{A}[Pere(E'_i) \cap E'_i], Pere(E'_i)/E'_i}}(\mathcal{B}) \prec ub_{E'_i}$, si elle existe
- une valuation supérieure ou égale à $ub_{E'_i}$, sinon.

Si $AFF(E'_i, V_{E'_i}) = \emptyset$ alors $V_{E'_i} = \emptyset$ et $Fils(E'_i) = \emptyset$. Donc, $E'_i \backslash V_{E'_i} = E'_i$ et $lb_{E'_i}$ est la valuation de la meilleure affectation $\mathcal{B}$ sur $Desc(E'_i)$ telle que $\mathcal{B}[E'_i \backslash V_{E'_i}] = \mathcal{A}[E'_i \backslash V_{E'_i}]$. Puisque BDH-val renvoie $lb_{E'_i}$, $P(\mathcal{A}, \emptyset, ub_{E'_i}, lb_{E'_i})$ est vérifiée.

Algorithme 13 : BDH-val($\mathcal{A}, E'_i, V_{E'i}, ub_{E'_i}, lb_{E'_i}, H_A, T_{cf_b}$)

```
1  if V_{E'i} = ∅ then
2      F ← Fils(E'i)
3      lb ← lb_{E'_i}
4      while F ≠ ∅ and lb ≺ ub_{E'_i} do
5          E_j ← Choix-hyperarete(F)
6          F ← F\{E_j}
7          E'_{j'} ← Construire(T_{cf_b}, H_A, E_j)
8          if (A[E'_{j'} ∩ E'_i], α) est un good then lb ← lb ⊕ α
9          else
10             α ←BDH-val(A, E'_{j'}, E'_{j'}\(E'_{j'} ∩ E'_i), ⊤, ⊥, H_A, T_{cf_{b+1}})
11             lb ← lb ⊕ α
12             Enregistrer le good (A[E'_{j'} ∩ E'_i], α)
13     return lb
14 else
15     x ← Choix-var(V_{E'_i})
16     d_x ← D_x
17     while d_x ≠ ∅ and lb_{E'_i} ≺ ub_{E'_i} do
18         v ← Choix-val(d_x)
19         d_x ← d_x\{v}
20         L ← {c ∈ E_{P,E'_i} | X_c = {x, y}, avec y ∉ V_{E'_i}}
21         lb_v ← ⊥
22         while L ≠ ∅ and lb_{E'_i} ⊕ lb_v ≺ ub_{E'_i} do
23             Choisir c ∈ L
24             L ← L\{c}
25             lb_v ← lb_v ⊕ c(A ∪ {x ← v})
26         if lb_{E'_i} ⊕ lb_v ≺ ub_{E'_i} then
           ub_{E'_i} ← min(ub_{E'_i},BDH-val(A ∪ {x ← v}, E'_i, V_{E'_i}\{x}, ub_{E'_i}, lb_{E'_i} ⊕ lb_v, H_A, T_{cf_b})
27     return ub_{E'_i}
```

Supposons qu'il existe $S = AFF(E'_i, V_{E'_i}), S \neq \emptyset$ tel que $\forall S' \subsetneq S, P(\mathcal{A}, S', ub_{E'_i}, lb_{E'_i})$ soit vérifiée. On va montrer que $P(\mathcal{A}, S, \alpha_{E'_i}, l_{E'_i})$ est vérifiée.

Si $V_{E'_i} \neq \emptyset$, BDH-val choisit une variable $x \in V_{E'_i}$ à la ligne 15 et lui affecte une valeur v de son domaine (ligne 18) dans la boucle while des lignes 17-26. Si le minorant de la valuation de cette extension $\mathcal{A} \cup \{x \leftarrow v\}$ $lb_{E'_i} \oplus lb_v$ est inférieur à $ub_{E'_i}$ alors il y a un appel récursif de BDH-val à la ligne 26 : BDH-val($\mathcal{A} \cup \{x \leftarrow v\}, E'_i, V_{E'_i}\backslash\{x\}, ub_{E'_i}, lb_{E'_i} \oplus lb_v, H_A, T_{cf_b}$). Or $AFF(E'_i, V_{E'_i}\backslash\{x\}) \subsetneq S$, donc, en utilisant notre hypothèse d'induction, on peut conclure que BDH-val($\mathcal{A} \cup \{x \leftarrow v\}, E'_i, V_{E'_i}\backslash\{x\}, ub_{E'_i}, lb_{E'_i} \oplus lb_v, H_A, T_{cf_b}$) retourne soit la valuation de la meilleure affectation $\mathcal{B}$ sur $Desc(E'_i)$ telle que $\mathcal{B}[E'_i\backslash(V_{E'_i}\backslash\{x\})] = \mathcal{A} \cup \{x \leftarrow v\}[E'_i\backslash(V_{E'_i}\backslash\{x\})]$ et $v_{\mathcal{P}_{\mathcal{A}[Pere(E'_i)\cap E'_i], Pere(E'_i)/E'_i}}(\mathcal{B}) \prec ub_{E'_i}$ ou bien une valuation supérieure ou égale à $ub_{E'_i}$. Dans le premier cas, la valuation de $\mathcal{B}$ est meilleure que celle de $\mathcal{B}'$ alors $ub_{E'_i}$ prend comme nouvelle valeur la valuation de $\mathcal{B}$. Dans le second cas, il n'y a pas de changement.

Ensuite, BDH-val choisit une nouvelle valeur dans le domaine de x et procède de la même manière que précédemment. Si $lb_{E'_i} \oplus lb_v$ n'est pas inférieure à $ub_{E'_i}$ alors on sait qu'il n'existe pas d'affectation $\mathcal{B}$ sur $Desc(E'_i)$ telle que $\mathcal{B}[E'_i\backslash(V_{E'_i}\backslash\{x\})] = \mathcal{A} \cup \{x \leftarrow v\}[E'_i\backslash(V_{E'_i}\backslash\{x\})]$ et $v_{\mathcal{P}_{\mathcal{A}[Pere(E'_i)\cap E'_i], Pere(E'_i)/E'_i}}(\mathcal{B}) \prec ub_{E'_i}$. Dans ce cas, BDH-val choisit une nouvelle valeur à affecter

à x.

Si toutes les valeurs du domaine courant ont été testées : la valeur courante de $ub_{E'_i}$ est soit la valuation de la meilleure affectation $\mathcal{B}$ sur $Desc(E'_i)$ telle que $\mathcal{B}[E'_i \backslash V_{E'_i}] = \mathcal{A}[E'_i \backslash V_{E'_i}]$ et $v_{\mathcal{P}_{\mathcal{A}[Pere(E'_i)\cap E'_i], Pere(E'_i)/E'_i}}(\mathcal{B}) \prec ub_{E'_i}$, soit elle est restée inchangée ce qui veut dire qu'il n'existe pas d'affectation $\mathcal{B}$ vérifiant les conditions précédentes. $P(\mathcal{A}, AFF(E'_i, V_{E'_i}), ub_{E'_i}, lb_{E'_i})$ est donc vérifiée.

Supposons maintenant que $V_{E'_i} = \emptyset$.

Puisque $AFF(E'_i, V_{E'_i}) \neq \emptyset$ alors $Fils(E'_i) \neq \emptyset$. A la ligne 5, BDH-val choisit un fils $E_j \in Fils(E'_i)$ et $Construire$ calcule un cluster $E'_{j'}$ qui recouvre E_j.

Si $\mathcal{A}[E'_i \cap E'_{j'}]$ est un good valué (ligne 8) alors cette valuation optimale est directement rajoutée au minorant et BDH-val choisit un autre fils de E'_i.

Sinon, il y a un appel récursif de BDH-val : BDH-val$(\mathcal{A}, E'_{j'}, E'_{j'} \backslash (E'_{j'} \cap E'_i), \top, \bot, H_A, T_{cf_b})$ (ligne 10). Dès lors, il peut y avoir deux cas de figure. Premier cas : $Desc(E'_{j'}) = S$. Cela veut dire E'_i n'a qu'un seul fils, $E'_{j'}$. BDH-val$(\mathcal{A}, E'_{j'}, E'_{j'} \backslash (E'_{j'} \cap E'_i), \top, \bot, H_A, T_{cf_b})$ va résoudre le VCSP induit $\mathcal{P}_{\mathcal{A}, Pere(E'_{j'})/E'_{j'}}$. Puisque, $V_{E'_{j'}} \neq \emptyset$, BDH-val choisit une variable $y \in V_{E'_{j'}}$ et lui affecte une valeur de son domaine. Par le même raisonnement que précédemment, on peut montrer que BDH-val$(\mathcal{A}, E'_{j'}, E'_{j'} \backslash (E'j' \cap E'_i), \top, \bot, H_A, T_{cf_b})$ renvoie :
- la valuation de la meilleure affectation $\mathcal{B}$ sur $Desc(E'_i)$ telle que $\mathcal{B}[E'_i \backslash V_{E'_i}] = \mathcal{A}[E'_i \backslash V_{E'_i}]$ et $v_{\mathcal{P}_{\mathcal{A}[Pere(E'_i)\cap E'_i], Pere(E'_i)/E'_i}}(\mathcal{B}) \prec ub_{E'_i}$, si elle existe
- une valuation supérieure ou égale à $ub_{E'_i}$, sinon.

BDH-val renvoie la meilleure valuation. Cette dernière est rajoutée au minorant et BDH-val choisit un autre fils de E'_i tant que le minorant lb est plus petit que $ub_{E'_i}$.

Si lb dépasse ou atteint $ub_{E'_i}$ alors BDH-val retourne lb puisqu'il n'existe d'affectation $\mathcal{B}$ qui vérifie les bonnes conditions. $P(\mathcal{A}, AFF(E'_i, V_{E'_i}), \alpha_{E'_i}, l_{E'_i})$ est donc vérifiée.

Si tous les fils ont été considérés alors, en renvoyant lb BDH-val renvoie la valuation de la meilleure affectation $\mathcal{B}$ sur $Desc(E'_i)$ telle que $\mathcal{B}[E'_i \backslash V_{E'_i}] = \mathcal{A}[E'_i \backslash V_{E'_i}]$ et
$v_{\mathcal{P}_{\mathcal{A}[Pere(E'_i)\cap E'_i], Pere(E'_i)/E'_i}}(\mathcal{B}) \prec ub_{E'_i}$.

$P(\mathcal{A}, AFF(E'_i, V_{E'_i}), ub_{E'_i}, lb_{E'_i})$ est donc vérifiée.

On vient de montrer par induction que $\forall S, S = AFF(E'_i, V_{E'_i})$, $P(\mathcal{A}, S, \alpha_{E'_i}, l_{E'_i})$ est vérifiée. De ce fait, $P(\emptyset, AFF(E'_1, V_{E'_1}), \top, \bot)$ est satisfaite.

Nous avons prouvé la correction partielle de BDH-val.
La correction totale est facile à voir. En effet, BDH-val doit explorer au départ un espace de recherche de taille finie. A chaque étape, cet espace est réduit d'au moins un nœud. Donc, BDH-val est correct, complet et termine.□

La complexité de BDH_{val} dépend du paramètre Δ qui permet de borner la largeur des hypergraphes de $\mathcal{CAH}_{H_A}[S]$ utilisés durant la résolution. La taille maximale des hyperarêtes de ces hypergraphes est inférieure ou égale à $\gamma + \Delta$.

Théorème 4.5.2 *La complexité en temps de BDH_{val} est $O(nombre_{T_c}.(\gamma+\Delta).exp(\gamma+\Delta+1))$, avec $nombre_{T_c}$ le nombre d'arbres T_c utilisés par BDH-val.*

Preuve Soient $\mathcal{P} = (X, D, C, S)$ un VCSP, H_A le CAH de référence de $H = (X, C)$.

On considère une décomposition arborescente (E, T_c) de H dont l'arbre T_c a été effectivement construit par BDH-val.

A l'image de la preuve de complexité de BDH, il est possible de recouvrir H_A (T_c) par des ensembles V_a de $\gamma + \Delta + 1$ variables qui vérifient la condition que toute affectation sur leurs variables ne sera générée par BDH qu'au plus $nombre_{Sep}(V_a)$ fois, avec $nombre_{Sep}(V_a)$ le nombre de séparateurs de H_A contenus dans V_a.

La définition des ensembles V_a est exactement identique à celle de la preuve de BDH.

Soit V_a un ensemble de $\gamma+\Delta+1$ variables tel que $\exists(E_{u_1},\ldots,E_{u_r})$ une chaîne sur une branche dans T_c, $V_a \subset E_{u_1} \cup \ldots \cup E_{u_r}$ ($r \geq 2$ puisque $|V_a| = \gamma + \Delta + 1$ et γ est la taille maximale des hyperarêtes de H_A) et $E_{u_2} \cup \ldots \cup E_{u_{r-1}} \subsetneq V_a$ (resp. $E_{u_1} \cap E_{u_2} \subset V_a$) si $r \geq 3$ (resp. $r = 2$).

V_a contient $r-1$ séparateurs qui sont les intersections entre hyperarêtes qui le recouvrent. En effet, $(E_{u_1},\ldots,E_{u_r})$ étant une chaîne et aucune hyperarête n'étant incluse dans une autre, les séparateurs se situent uniquement entre deux éléments consécutifs de la chaîne.

Lors de la résolution, il est possible de recouvrir V_a de différentes manières avec les arbres T_{cf} associés à T_c. Néanmoins, au moins un séparateur de V_a sera une intersection entre deux clusters dans chaque T_{cf}. Soit T_{cf} un arbre associé à T_c tel que s_1 soit une intersection entre deux clusters. La résolution basée sur cet arbre va générer une affectation sur V_a et enregistrer sur s_1 un good valué. Si s_1 est également une intersection entre deux clusters dans un arbre T'_{cf} associé à T_c, utilisé lors d'une nouvelle tentative d'affectation des variables de V_a avec les mêmes valeurs, le good valué va permettre de stopper l'affectation. Par contre si s_1 n'est pas une intersection, la situation du good peut conduire à reproduire totalement l'affectation mais un autre good valué sera enregistré sur un autre séparateur, s_2 de V. Dorénavant, si s_1 ou s_2 est une intersection entre des clusters d'un arbre associé à T_c utilisé lors de la recherche, l'affectation ne sera pas reproduite. Donc une affectation sur V_a peut être reproduite autant de fois qu'il est possible de le décomposer au travers de ses séparateurs : donc autant de fois qu'il a de séparateurs.

Le nombre maximum de séparateurs $(r-1)$ d'un V_a est borné par $\gamma+\Delta$ car le nombre d'éléments de la chaîne $(E_{u_1},\ldots,E_{u_r})$ est bornée par $\gamma+\Delta+1$.

On a prouvé donc que toute affectation sur V est générée au pire $\gamma+\Delta$ fois.

Sur chacun des V_a recouvrant T_c une affectation est produite au plus $\gamma+\Delta$ fois. Le nombre d'affectations possibles sur V_a est borné par $d^{\gamma+\Delta+1}$. Donc le nombre d'affectations possibles sur l'ensemble des variables du problème est borné par $nombre_{T_c}.nombre_{V_a}.(\gamma+\Delta).d^{\gamma+\Delta+1}$, avec $nombre_{V_a}$ le nombre d'ensembles V_a recouvrant T_c. Le nombre de V_a étant borné par celui d'hyperarêtes de H_A, la complexité de BDH est en $O(nombre_{T_c}.(\gamma+\Delta).exp(\gamma+\Delta+1))$. □

Comme pour BDH classique, cette complexité est bornée par $O(exp(h))$ (la complexité de la méthode sans apprentissage), avec h le nombre maximum de variables dans une branche d'un arbre T_c.

Comme pour BTD-val, BDH-val peut être amélioré par des techniques de consistances locales valuées. Nous pouvons définir LC-BDH-val une extension de BDH-val avec LC une technique de consistance locale valuée. Dans nos expérimentations, nous avons utilisé la méthode FC-BDH-val. De même, il nous faudra poursuivre nos expérimentations avec LC-BDH-val$^+$ (algorithme 16). Cette méthode est similaire à LC-BTD-val$^+$ [dGSV06] que nous avons présenté dans le chapitre 2. Les motivations sont identiques à celles de LC-BTD-val$^+$, c'est-à-dire permettre un élagage plus important en débutant la résolution d'un problème par un meilleur majorant que $\top$. Nous utilisons donc comme majorant la différence entre la valuation de la meilleure solution connue et la valuation locale de l'affectation courante comme c'est la cas dans [dGSV06]. Dans ce cas, les informations enregistrées ne sont pas forcément des goods valués. Il peut arriver que ce majorant soit inférieur à la valuation optimale du sous-problème. Ainsi, LC-BTD-val$^+$ ne dispose pas de la valuation optimale, mais d'un minorant de cette dernière. Cette information est malgré tout enregistrée, modifiant ainsi la définition de good valué structurel. Un good valué est composé d'une affectation $\mathcal{A}[E_i \cap E_j]$ sur une intersection entre un cluster E_i et un de ses fils E_j et d'une valuation qui est soit un minorant ($opt = 0$) de la valuation optimale du problème enraciné en E_j, soit cette valuation optimale ($opt = 1$).

Théorème 4.5.3 *La complexité en temps de LC-BDH-val$^+$ est* $O(\alpha^*.(\gamma+\Delta).nombre_{T_c}.exp(\gamma+\Delta+1))$, *avec* $nombre_{T_c}$ *le nombre d'arbres* T_c *utilisés par LC-BDH-val$^+$ et* α^* *la valuation optimale du VCSP.*

Preuve La preuve est similaire à celle de BDH-val. Mais, on s'appuie par ailleurs sur l'observation de [dGSV06] concernant la complexité de LC-BTD-val^+. En effet, une affectation sur

Algorithme 14 : LC-BDH-val$^+(\mathcal{A}, E'_i, V_{E'_i}, lb_{E'_i}, ub, cub, H_A, T_{cf_b})$

```
1  if V_{E'_i} = ∅ then
2      lb ← lb_{E_i}
3      F ← Fils(E'_i)
4      while F ≠ ∅ and lb ≺ ub do
5          E_j ← Choix-hyperarete(F)
6          F ← F\{E_j}
7          E'_{j'} ← Construire(T_c, H_A, E_j)
8          if (A[E'_{j'} ∩ E'_i], lb_{E'_{j'}}, opt) est un good valué then lb ← lb ⊕ lb_{E'_{j'}}
9          else
10             ub' ← ub − lb_{E'_i} + lb_{E'_{j'}}
11             lb'_{E'_{j'}} ← LC-BDH-val(A, E'_{j'}, V_{E'_{j'}}, lb_{E'_{j'}}, ub', H_A, T'_c)
12             LB_{E'_{j'}} ← lb'_{E'_{j'}} + ΔC(E'_{j'})
13             opt' ← (lb'_{E'_{j'}} ≺ ub')
14             Enregistrer (A[E'_{j'} ∩ E'_i], lb'_{E'_{j'}}, opt')
15     return lb
16 else
17     x ← Choix-var(V_{E'_i})
18     d_x ← D_x
19     while d_x ≠ ∅ and lb_{E'_i} ≺ ub do
20         v ← Choix-val(d_x)
21         d_x ← d_x\{v}
22         lb_v ←LC(E'_i)
23         if lb_{E'_i} ⊕ lb_v ≺ ub then
           ub ← min(ub,LC-BDH-val(A ∪ {x ← v}, E'_i, V_{E'_i}\{x}, lb_{E'_i} ⊕ lb_v, ub, H_A, T_c)
24     return ub
```

un V_a est reproduite au pire $\gamma + \Delta$ fois, si les informations enregistrées sont des goods valués. Cependant, si on ne dispose que d'un minorant, l'affectation peut être reproduite encore. Mais, à chaque nouvelle génération le minorant est augmente strictement. Donc, il faut au pire α^* générations pour atteindre l'optimum.

Ainsi, toute affectation dans un V_a est générée au pire à $\alpha^*(\gamma + \Delta)$ reprises.

Donc la complexité de LC-BDH-val^+ est en $O(\alpha^*.(\gamma + \Delta).nombre_{T_c}.exp(\gamma + \Delta + 1)).\square$

Dans ce cas, les informations enregistrées ne sont pas forcément des goods valués.

4.6 Ordre sur les variables : choix hypergraphes recouvrants

BDH et BDH-val recèlent plusieurs points de choix heuristiques : choix de variables, d'hyperarêtes, de fusion d'hyperarêtes. Suivant ces heuristiques, les bornes de complexité de ces méthodes sont différentes. En réalité, leur impact majeur se situe dans l'ordre d'affectation des variables et la liberté qu'elle accorde dans la construction de ce dernier. Nous commençons par définir un ensemble de classes d'ordres qui offrent de plus en plus de liberté.

Dans cette partie, le terme hyperarête sera utilisée à la fois pour les hyperarêtes d'un hypergraphe et pour les clusters d'une décomposition arborescente, ces deux termes renvoyant à un

même objet.

4.6.1 Les Classes d'ordres

- **Classe 1.** Ordre sur les variables statique et compatible avec un arbre d'une décomposition arborescente T_c qui induit un hypergraphe de $\mathcal{CAH}_{H_A}$.
  Dans cette classe, l'arbre de décomposition arborescente dans BDH ou BDH-val est toujours le même. Il est calculée avant le début de la recherche. Ensuite, un ordre compatible statique est calculé sur les clusters de cette décomposition. Enfin, l'ordre d'affectation des variables dans chaque cluster est également calculé de manière statique. Cette classe regroupe les ordres définis pour les premières versions de BTD et BTD-val. Son principal défaut est l'inefficacité des ordres d'affectation statiques.
- **Classe 2.** Ordre sur les clusters statique et compatible avec un arbre d'une décomposition arborescente T_c qui induit un hypergraphe de $\mathcal{CAH}_{H_A}$, ordre dynamique sur les variables dans chaque cluster.
  Comme dans la classe 1, la décomposition est toujours la même et est calculée de manière statique. L'ordre sur les clusters est aussi statique et compatible avec la décomposition. Par contre, l'ordre d'affectation des variables dans chaque cluster est calculé de manière dynamique, au cours de la recherche. Les premiers résultats de BTD et BTD-val ont été obtenus avec des ordres de cette classe.
- **Classe 3.** Ordre sur les hyperarêtes dynamique et compatible avec un arbre d'une décomposition arborescente T_c qui induit un hypergraphe de $\mathcal{CAH}_{H_A}$, ordre dynamique sur les variables dans chaque cluster de cet arbre.
  Ici, la liberté est accrue comparée à la classe précédente. La décomposition reste inchangée durant la résolution et est toujours calculée de manière statique. Mais, en plus d'un ordre dynamique d'affectation des variables dans chaque cluster, l'ordre sur les clusters est également dynamique tout en restant compatible avec la décomposition.
- **Classe 4.** Ordre de la *Classe 3* sur un unique arbre, T_{cf}, associé à un arbre d'une décomposition arborescente T_c qui induit un hypergraphe de $\mathcal{CAH}_{H_A}$.
  A partir de cette classe, nous allons accroître le caractère dynamique de l'ordre au détriment de la complexité en temps. La décomposition tout en restant inchangée durant la résolution, est calculée de manière statique par fusion de clusters connectés dans l'arbre T_c. Cette opération de fusion donne un nouvel arbre d'une nouvelle décomposition (utilisée par BDH ou BDH-val) qui induit un hypergraphe de $H'_A \in \mathcal{CAH}_{H_A}[S]$. Ainsi, tout ordre de la classe 3 sur cette décomposition est un ordre de cette classe.
- **Classe 5.** Ordre de la *Classe 3* sur un ensemble d'arbres, T_{cf}, associés à un arbre d'une décomposition arborescente T_c qui induit un hypergraphe de $\mathcal{CAH}_{H_A}$.
  Ici, contrairement à la classe 4, la fusion des clusters est dynamique. On part d'un arbre T_c et à chaque étape on choisit dynamiquement le prochain cluster suivant un ordre compatible, on le fusionne si nécessaire avec d'autres clusters du sous-problème dont il est la racine et pour finir les variables de ce nouveau clusters sont instanciées suivant un ordre dynamique. En définitive, les ordres de cette classe sont de classe 3 sur un ensemble d'arbres de décompositions arborescentes associés à un arbre T_c.
- **Classe 6.** Ordre de la *Classe 5* sur un ensemble d'arbres T_c qui induisent des hypergraphes de $\mathcal{CAH}_{H_A}$.
  Cette classe ne se restreint pas à un arbre T_c, mais en utilise plusieurs.
- **Classe ++.** Ordre sur les variables dynamique.
  Les ordres de cette classe sont totalement dynamiques. Aucune restriction venant de H_A n'est utilisée. Cette classe n'offre pas de garantie en terme de complexité en temps.

Ces différentes classes d'ordres forment une hiérarchie : *Classe 1* $\subset$ *Classe 2* $\subset$ *Classe 3* $\subset$ *Classe 4* $\subset$ *Classe 5* $\subset$ *Classe 6* $\subset$ *Classe ++*.

Théorème 4.6.1 *Avec un ordre de la Classe 1, Classe 2 ou Classe 3, BDH et BDH-val ont une*

complexité en temps de $O(exp(\gamma))$.

Théorème 4.6.2 *Avec un ordre de la Classe 4, BDH et BDH-val ont une complexité en temps de $O(exp(\gamma + \Delta))$, avec $\gamma + \Delta$ la taille maximale des clusters de la décomposition arborescente utilisée.*

Théorème 4.6.3 *Avec un ordre de la Classe 5, BDH a une complexité en temps de $O(exp(\gamma + \Delta + 1))$ tandis que BDH-val a une complexité en temps de $O((\gamma + \Delta).exp(\gamma + \Delta + 1))$, avec $\gamma + \Delta$ la taille maximale des clusters de la décomposition arborescente utilisée.*

Théorème 4.6.4 *Avec un ordre de la Classe 6, BDH a une complexité en temps de $O(Nombre_{T_c}.exp(\gamma + \Delta + 1))$ tandis que BDH-val a une complexité en temps de $O(Nombre_{T_c}.(\gamma+\Delta).exp(\gamma+\Delta+1))$, avec $\gamma+\Delta$ la taille maximale des clusters des décompositions arborescentes utilisées et $Nombre_{T_c}$ le nombre d'arbres qui induisent des hypergraphes de $\mathcal{CAH}_{H_A}[S]$ auxquels sont associés les arbres des décompositions arborescentes.*

4.6.2 Heuristiques de construction des arbres.

Dans cette partie, nous donnons quelques stratégies de construction d'un arbre T_c d'une décomposition arborescente dont les clusters sont les hyperarêtes du recouvrement de référence H_A. Ces stratégies permettent de faire des choix sur les arêtes à rajouter dans T_c dans le but d'obtenir un arbre de qualité par rapport aux caractéristiques du problème et de H_A. A chaque étape, les heuristiques déterminent l'ensemble des fils de l'hyperarête E_i dans T_c suivant les règles établies à la section 4.4.1. Elles peuvent être statiques (calculées avant le début de la résolution) ou dynamiques (calculées durant la résolution).

Les stratégies statiques :

- $hauteur_{arb}$: On choisit un arbre de hauteur minimum, le but étant de réduire la complexité de LC-BDH-val.
- $taille_{arb}$: On choisit comme prochaine hyperarête à relier avec la courante, celle de plus grande taille.
- $minesp_{arb}$: On choisit comme prochaine hyperarête à relier avec la courante, celle qui minimise le nombre de solutions espérées.

Les stratégies dynamiques :

- $minesp_{arbdyn}$: On choisit comme prochaine hyperarête à relier avec la courante, celle qui minimise le nombre de solutions espérées.

4.6.3 Heuristiques de fusion d'hyperarêtes.

Ici, nous allons définir plusieurs heuristiques de fusion d'hyperarêtes. Elles interviennent juste après les stratégies précédentes. Une fois que l'ensemble des fils de E_i est calculé, ces heuristiques choisissent parmi ces derniers ceux qui seront fusionnés avec E_i. Cela conduit à un nouvel arbre T_{cf} d'une nouvelle décomposition arborescente qui recouvre les nœuds de T_c. L'hypergraphe induit par les nœuds de T_{cf} est un éléments de $\mathcal{CAH}_{H_A}[S]$.

Les heuristiques statiques :

- *sep* : Cette heuristique prend en paramètre la valeur à ne pas dépasser pour la taille des séparateurs. Elle fusionne chaque paire d'hyperarêtes de type <père, fils> dont la taille du séparateur dépasse cette valeur.
- *vp* : On lui fournit le nombre minimal de variables propres (variables qui n'appartiennent pas à l'hyperarête père) que doit contenir une hyperarête. Elle fusionne toute hyperarête dont le nombre de variables propres est inférieur ou égal cette valeur, avec son père.
- *esp* : Ce critère est basé sur l'espérance du nombre de solutions d'une hyperarête. On fusionne deux hyperarêtes si l'hyperarête résultant possède *a priori* moins de solutions que les deux prises séparément, l'objectif étant de construire des hyperarêtes avec plus de variables et moins de solutions en vue de découvrir les inconsistances le plus tôt possible.

Les heuristiques dynamiques :

- $minesp_{fdyn} - sep$: Cette heuristique prend en paramètre la valeur qu'on souhaite ne pas dépasser pour la taille des séparateurs et la valeur maximale de Δ. Les hyperarêtes sont ordonnées dynamiquement suivant leur espérance du nombre de solutions. L'heuristique considère les hyperarêtes suivant cet ordre et fusionne l'hyperarête courante E_i avec son premier fils E_j pour lequel la taille du séparateur dépasse la valeur souhaitée. Ensuite, cette nouvelle hyperarête est fusionnée avec le premier fils de E_j selon l'ordre donné par l'espérance du nombre de solutions dont la taille du séparateur avec E_j dépasse la valeur souhaitée. Cette opération est réitérée tant que le cardinal de l'hyperarête courante ne dépasse pas $\alpha + \Delta$.
- $minesp_{fdyn} - vp$: Cette heuristique prend en paramètre le nombre de variables propres minimum souhaité et la valeur maximale de Δ. Les hyperarêtes sont ordonnées dynamiquement suivant leur espérance du nombre de solutions. L'heuristique considère les hyperarêtes suivant cet ordre et fusionne l'hyperarête courante E_i avec son premier fils E_j pour lequel le nombre de variables propres est inférieur ou égal à la valeur souhaitée. Ensuite, cette nouvelle hyperarête est fusionnée avec le premier fils de E_j selon l'ordre donné par l'espérance du nombre de solutions dont le le nombre de variables propres est inférieur ou égal à la valeur souhaitée. Cette opération est réitérée tant que le cardinal de l'hyperarête courante ne dépasse pas $\alpha + \Delta$.

Combinaisons des heuristiques statiques et dynamiques :

- sep+$minesp_{fdyn} - sep$: un hypergraphe recouvrant est construit de manière statique avec sep, puis on utilise $minesp_{fdyn} - sep$ sur ce recouvrement
- vp+$minesp_{fdyn} - sep$: un hypergraphe recouvrant est construit de manière statique avec vp, puis on utilise $minesp_{fdyn} - sep$ sur ce recouvrement
- sep+$minesp_{fdyn} - vp$: un hypergraphe recouvrant est construit de manière statique avec sep, puis on utilise $minesp_{fdyn} - vp$ sur ce recouvrement
- vp+$minesp_{fdyn} - vp$: un hypergraphe recouvrant est construit de manière statique avec vp, puis on utilise $minesp_{fdyn} - vp$ sur ce recouvrement

4.6.4 Heuristiques de choix d'hyperarêtes.

Les heuristiques de choix d'hyperarêtes sont utilisées pour bien choisir la prochaine hyperarête qui sera intanciée parmi les fils de l'hyperarête courante.

Ordres statiques :

Choix de racine :

- *alea* : Le choix est aléatoire.
- *minesp* : Cette heuristique est basée sur l'espérance du nombre de solutions partielles d'une hyperarête et de sa taille. Les expériences menées ont montré qu'il était toujours préférable de choisir comme prochaine hyperarête une de taille importante. Cela justifie le choix de l'hyperarête qui minimise le ratio entre l'espérance du nombre de solutions partielles et la taille de l'hyperarête comme racine. Cette heuristique permet d'explorer en premier une hyperarête de grande taille avec peu de solutions.
- *card* : On choisit l'hyperarête de plus grand cardinal (nombre de variables de l'hyperarête).
- *taille* : On choisit l'hyperarête de plus grande taille (produit de la taille des domaines des variables de l'hyperarête).
- *bary* : On choisit l'hyperarête barycentre comme racine. Cette notion de barycentre utilise une autre de distance entre deux hyperarêtes dans l'hypergraphe qui est définie par la plus courte chaîne entre les deux. L'hyperarête barycentre x est donc celle qui minimise $\Sigma_{y \in X} dist(x, y)$.

Choix de fils :

- $alea_f$: Les hyperarêtes fils sont ordonnés aléatoirement.

- $minesp_f$: Cette heuristique est similaire à $minesp$ mais elle ordonne les fils suivant la valeur croissante du ratio de ces derniers.
- $minesp_{fdyn}$: On ordonne les hyperarêtes fils suivant la valeur croissante de l'espérance du nombre de solutions des problèmes enracinés en chacune de ces hyperarêtes.
- $maxesp_{fdyn}$: On ordonne les hyperarêtes fils suivant la valeur décroissante de l'espérance du nombre de solutions des problèmes enracinés en chacune de ces hyperarêtes.
- $mincard_{fdyn}$: On ordonne les hyperarêtes fils suivant la valeur croissante du cardinal des problèmes enracinés en chacun de ces hyperarêtes.
- $mintaille_{fdyn}$: On ordonne les hyperarêtes fils suivant la valeur croissante de la taille des problèmes enracinés en chacun de ces hyperarêtes.
- $maxcard_{fdyn}$: On ordonne les hyperarêtes fils suivant la valeur décroissante du cardinal des problèmes enracinés en chacun de ces hyperarêtes.
- $maxtaille_{fdyn}$: On ordonne les hyperarêtes fils suivant la valeur décroissante de la taille des problèmes enracinés en chacun de ces hyperarêtes.
- $minsep_f$: On ordonne les hyperarêtes fils suivant la valeur croissante de la taille du séparateur avec l'hyperarête parent.

Ordres dynamiques :

Choix racine :

Pour choisir l'hyperarête racine, étant donné qu'il est nécessaire de le faire pour savoir où débuter la recherche, on peut utiliser les heuristiques statiques. Cependant on en propose une nouvelle :
pv : Les heuristiques dynamiques de choix de variables ont permis d'améliorer grandement l'efficacité des algorithmes d'énumération. Pour tirer profit de cela, on choisit une heuristique de choix de variables dynamique et on prend comme hyperarête racine celle dans laquelle se trouve la prochaine variable (les prochaines variables) dans l'ordre.

Choix de fils :

La partie dynamique de l'ordre se situe en fait dans l'ordonnancement des fils.

- $alea_{fdyn}$: On choisit aléatoirement la prochaine hyperarête fils à visiter.
- pv_f : C'est une déclinaison de pv pour ordonner la filiation. On visite en premier l'hyperarête qui contient la prochaine variable parmi celles qui se trouvent dans les hyperarêtes fils non encore instanciés, dans l'ordre de l'heuristique de choix de variables dynamique qu'on a choisie.
- $minesp_{fdyn}$: C'est une version dynamique de $minesp_f$ qui estime le nombre de solutions d'une hyperarête uniquement quand les variables de son père ont toutes été affectées. Elle fait un meilleur choix en tenant compte des modifications éventuelles du problème dues au filtrage des domaines durant la recherche.

4.6.5 Heuristiques de choix de variables.

Dans cette partie, nous définissons les ordres dans lesquels les variables d'une hyperarête seront affectées.

Les ordres statiques :

- $alea_s$: L'ordre d'instanciation des variables est calculé aléatoirement.
- mdd_s : On ordonne les variables suivant la valeur croissante du ratio taille du domaine sur degré.

Les ordres dynamiques :

- $alea_d$: On choisit la prochaine variable à instancier dans le cluster aléatoirement.
- mdd_d : On choisit comme prochaine variable à affecter celle qui minimise le ratio taille du domaine courant (après un éventuel filtrage) sur degré.

Toutes les heuristiques proposées donnent un éventail très large de combinaisons possibles pour définir des ordres totaux sur les variables de toutes les classes d'ordres définies dans la section 4.6.1.

4.7 Etude expérimentale

Dans cette étude, nous ne considérons pas les ordres de la classe 6 car cela requiert une étude supplémentaire pour une implémentation différente qui n'a pas encore été faite. On va dans un premier temps comparer les heuristiques d'ordonnancement des clusters dans les classes 1, 2 et 3. Cela permettra d'avoir une idée assez précise des stratégies efficaces pour l'exploitation d'une décomposition. Une fois ce travail effectué, on peut dès lors comparer les critères d'une décomposition efficace en pratique en utilisant des heuristiques des classes 4 et 5 qui partant d'une bonne décomposition essaie de l'améliorer en fusionnant des clusters pour accroître l'aspect dynamique de l'ordre d'affectation des variables. Cette étude comporte deux parties, une première consacrée aux CSP et une seconde aux VCSP.

Dans les deux cas, un ordre sur les variables est défini par l'utilisation d'une heuristique de fusion de clusters (uniquement pour les classes 4 et 5), suivie d'une heuristique de choix de racine, puis d'ordonnancement des fils d'un cluster et enfin de choix de variables à l'intérieur d'un cluster.

4.7.1 Comparaison des heuristiques sur les CSP

Cette première partie utilise les instances de CSP aléatoires structurés définies dans le chapitre précédent. Pour chaque classe d'instances, les résultats présentés sont les moyennes sur un ensemble de 50 instances. Le temps de résolution est limité à 30 minutes par instance, au-delà le solveur est arrêté et le problème correspondant est considéré comme non résolu. De ce fait, dans les tableaux, le symbole $>$ signifie qu'au moins une instance est non résolue et donc que le temps de résolution réelle est supérieure à la valeur donnée pour laquelle on considère tout de même une durée de 30 minutes pour l'instance. Le terme Mem signifie qu'au moins une instance ne peut pas être résolue parce qu'elle requiert plus de 1 Go de mémoire. La dernière ligne des tableaux (EC) donne, pour chaque heuristique, l'écart moyen entre sa durée de résolution en moyenne sur chaque classe d'instances et la plus petite durée de résolution moyenne obtenue par l'ensemble des heuristiques sur cette classe. Si une classe n'est pas résolue à cause d'un espace mémoire insuffisant, l'écart moyen considère que toutes les instances de la classe ont été résolues en 30 minutes. EC traduit la robustesse de l'heuristique et constitue de ce fait un critère de comparaison de tout premier plan. Dans le chapitre 3, une étude a été menée sur les algorithmes de triangulations qui sont utilisés pour le calcul des décompositions arborescentes. Il en est ressorti que MCS donnait en un temps très limité de très bonnes décompositions au niveau de l'efficacité pratique de la résolution. Cela justifie son utilisation pour le calcul des décompositions dans cette étude.

Les premières observations mettent en lumière les très mauvais résultats des ordres de la classe 1. Cela n'est pas surprenant, car les ordres statiques sur les variables ont un comportement similaire au niveau des méthodes énumératives.

L'heuristique dynamique de choix variables mdd_d est utilisée pour tous les résultats présentés dans cette section. A partir de là, un ordre sur les variables est défini par une heuristique de choix de racine, d'ordonnancement de fils. Cela est associé à une heuristique de fusion de clusters modifiant la structure de la décomposition pour les classes 4 et 5. Les classes 2 et 3 donnent d'assez bons résultats grâce à l'apport de la dynamicité. Le tableau 4.1 montre le temps d'exécution des meilleures heuristiques et d'un ordre aléatoire, ainsi que la largeur des décompositions arborescentes calculées et la taille maximale de leurs séparateurs. Il permet de se rendre compte de l'importance d'un choix de racine judicieux. En effet, le symbole $>$ vient d'une instance non résolue à cause d'un choix de racine désastreux. Ce problème non résolu a pour effet d'augmenter de manière substantielle la moyenne des résultats alors que sur les 49 autres instances l'heuristique se comporte très bien par rapport à un choix aléatoire. Il est à noter aussi que les instances non résolues sont différentes pour les heuristiques *card* et *minesp*. Les heuristiques $card{+}minsep_f$ et $taille{+}pv$ sont les plus robustes parmi celles des

CSP	γ	s	Classe 2				Classe 3		
			alea	*card*	*minesp*	*minesp*	*minesp*	*pv*	*card*
			$alea_f$	$minsep_f$	$minsep_f$	$minesp_f$	$minesp_{fdyn}$	pv_f	pv_f
(a)	14,00	12,22	4,86	3,41	2,50	2,52	2,45	6,85	5,34
(b)	13,54	11,90	16,96	>41,10	2,54	2,69	2,32	>40,07	>41,47
(c)	13,16	11,40	4,76	3,38	4,95	5,06	4,97	5,67	3,55
(d)	12,52	10,64	4,39	1,12	>36,87	>36,87	>37,27	0,95	1,16
(e)	18,82	16,92	50,47	16,03	>56,81	>56,77	>56,44	>99,67	15,26
(f)	18,24	16,56	48,64	23,25	14,06	14,03	13,04	>77,14	24,00
(g)	17,80	15,80	Mem	92,52	64,63	64,87	>78,47	>81,60	>107,10
(h)	16,92	15,24	Mem	26,24	3,92	3,91	4,51	10,61	17,99
(i)	14,04	12,34	16,57	15,16	43,16	43,41	44,66	>53,53	17,89
(j)	13,86	11,98	17,15	8,51	>42,61	>43,12	>50,84	10,93	19,17
(k)	13,38	11,82	8,71	7,01	10,91	10,94	5,06	6,01	6,91
(l)	12,80	11,16	15,46	3,82	16,84	16,88	18,13	>52,97	5,03
(m)	18,92	17,02	Mem	Mem	Mem	Mem	Mem	Mem	Mem
(n)	14,04	12,58	11,67	7,01	7,78	8,08	7,31	8,32	7,54
(o)	13,94	12,10	11,68	25,54	24,19	23,49	27,01	7,26	15,11
EC			366,33	130,51	134,36	134,42	135,74	143,01	131,41

Tab. 4.1 – Valeurs des paramètres γ et s du recouvrement calculé par MCS et durées d'exécution en secondes de FC-BDH avec différentes heuristiques des classes d'ordres 2 et 3 : la première ligne en dessous des classes donne l'heuristique de choix de racines et la seconde celle d'ordonnancement des fils. mdd_d est utilisée pour le choix de variables à l'intérieur des clusters. *EC* donne l'écart moyen des résultats de l'ordre avec les meilleurs résultats obtenus dans les classes 2, 3 et 4.

CSP	γ	s	*alea*	*card*	*minesp*	*minesp*	*minesp*	*pv*	*card*
			$alea_f$	$minsep_f$	$minsep_f$	$minesp_f$	$minesp_{fdyn}$	pv_f	pv_f
(a)	15,04	4,98	7,19	2,75	2,17	2,17	2,08	4,65	2,65
(b)	15,04	5,00	7,31	2,58	1,76	1,76	1,63	2,47	2,97
(c)	15,86	5,00	17,30	1,41	1,03	1,05	1,13	1,23	1,30
(d)	16,48	5,00	6,86	1,67	0,39	0,39	0,63	0,88	1,75
(e)	20,00	4,98	>143,96	10,66	>52,16	>52,14	>51,67	>50,96	10,92
(f)	20,00	5,00	>93,54	10,05	8,87	8,81	8,39	45,18	10,34
(g)	20,82	5,00	>220,65	33,93	4,66	4,61	4,41	41,92	34,20
(h)	21,44	5,00	>231,63	11,38	3,17	3,17	3,17	7,58	10,63
(i)	15,00	5,00	>47,60	5,86	7,75	7,71	6,65	8,86	6,44
(j)	15,40	5,00	>46,72	4,19	3,97	3,94	3,36	4,99	6,81
(k)	16,02	5,00	>44,06	2,80	3,62	3,71	3,52	4,49	3,06
(l)	18,90	5,00	>82,38	4,03	1,38	1,40	1,26	14,78	3,55
(m)	58,28	4,88	>315,24	66,94	63,35	63,15	62,99	74,32	66,33
(n)	15,02	5,00	>44,98	5,48	4,41	4,50	4,41	5,02	5,86
(o)	15,44	5,00	10,56	4,86	4,94	4,92	3,94	5,54	5,24
EC			>80,24	11,24	>3,15	>3,14	>2,86	>10,43	3,71

Tab. 4.2 – Valeurs des paramètres γ et s du recouvrement calculé par MCS après l'utilisation de l'heuristique de fusion *sep* (avec une taille de séparateurs d'au plus 5) et durées d'exécution en secondes de FC-BDH avec différentes heuristiques de la classe 4 : la première ligne donne l'heuristique de choix de racines et la seconde celle d'ordonnancement des fils. mdd_{dyn} est utilisée pour le choix de variables à l'intérieur des clusters. *EC* donne l'écart moyen des résultats de l'ordre avec les meilleurs résultats obtenus dans les classes 2, 3 et 4.

classes d'ordres 2 et 3, avec un écart moyen tout de même très important ($EC > 130$ secondes). Cet écart est calculé sur l'ensembles des résultats des classes 2, 3 et 4. Le passage à cette dernière classe 4 résout les problèmes des classes 2 et 3. L'heuristique de fusion utilisé est *sep* dans le but de réduire l'espace mémoire requis. Le tableau 4.2 montre les temps d'exécution des heuristiques de cette classe avec une taille de séparateur limitée à 5. Avec des clusters de taille plus importante, les mauvais choix ont des conséquences plus visibles. Un ordre aléatoire éprouve plus de difficultés, surtout sur les problèmes inconsistants. Les heuristiques *minesp* et *pv* résolvent toutes les instances sauf une dans deux classes à cause d'un mauvais choix de racine alors que *card* arrive à résoudre une de plus. Hormis ces instances non résolues, *minesp* donne de très bons résultats. Au niveau de la classe (250,20,20,99,10,25,10) il est possible d'observer un apport net d'un choix dynamique des clusters fils. Mais cet apport devrait être plus important sur des problèmes avec un plus grand nombre de clusters fils ce qui augmenterait les choix possibles de même que leur impact. On constate que l'heuristique *minesp*+$minesp_{fdyn}$ donne les meilleurs résultats, mais on peut penser que cela est dû à la nature aléatoire des problèmes considérés. Il est donc possible que pour des problèmes réels une heuristique du type *card*+(*minsep* ou pv_f) se comporte mieux. *card*+pv_f est très robuste avec un écart moyen limité à $3,71$ secondes. Cette valeur peut être très proche de la meilleure qui n'est pas connue avec exactitude (elle est supérieure à $2,86$ secondes). Le choix d'un cluster de grande taille satisfait le principe du first-fail. Il permet de faire des choix très contraints. Ensuite, une fois cette racine totalement instanciée, le fait de laisser une heuristique efficace de choix variables (mdd_{dyn}) guider l'ordre dans lequel les clusters fils seront visités, permet de continuer dans le même sens en continuant de faire des choix très contraints. L'extension de l'ordre avec *sep* comme critère de fusion et $s = 5$, permet d'améliorer les résultats des heuristiques.

Pour trouver la meilleure valeur de s, on a fait varier cette dernière sur les tableaux 4.3 et 4.4, avec l'heuristique *minesp*+$minesp_{fdyn}$. Quand la valeur de s est grande, la modification de la décomposition est assez limitée, avec une faible augmentation de la taille des clusters. Plus s diminue, plus l'ordre des variables est dynamique et plus la résolution devrait être efficace. Néanmoins, à partir d'une certaine valeur, la taille des clusters devient trop grande et leur nombre trop petit pour permettre une utilisation de la structure pour limiter les redondances et une conservation de bornes de complexité acceptables. A partir de là, le nombre d'instances non résolues se multiplient et le comportement de BDH se rapproche de celui d'algorithmes énumératifs comme FC ou MAC (qui sont souvent incapables de résoudre une bonne partie des instances utilisées ici). Cependant, on observe l'existence d'une instance non résolue pour la classe (250,20,20,107,5,20,10) entre $s = 5$ et $s = 10$, et pour la classe (250,20,20,129,5,20,30) entre $s = 7$ et $s = 10$. Cela est dû à un mauvais choix de racine de l'heuristique *minesp*+$minesp_{fdyn}$. Pour les autres instances, le comportement est celui attendu. La valeur $s = 5$ est la plus robuste pour les instances considérées.

On observe un comportement semblable avec l'heuristique de fusion *vp* (voir le tableau 4.5). Le temps de résolution va généralement s'améliorer lorsque le nombre de variables propres dans les clusters augmente. Mais, à partir d'un certain seuil (variable selon les classes d'instances considérées), l'apport de l'exploitation de la structure disparaît du fait de la diminution du nombre de clusters et de l'augmentation significative de leur taille. Notons que cette heuristique permet d'éviter le problème de consommation excessive de mémoire bien qu'elle n'agissent pas directement sur le paramètre s. La valeur 3 comme borne du nombre de variables propres donne les meilleures décompositions au vue de la résolution ($EC = 4,94$ secondes).

Enfin, concernant l'heuristique de fusion *esp*, elle revient généralement à regrouper une grande partie des clusters ensemble. Elle s'avère donc inutilisable en pratique.

On a pu montrer en pratique que la dynamicité de l'ordre des variables permet de faire des choix plus éclairés et ainsi d'améliorer les performances de la méthode. Néanmoins, il est nécessaire de maintenir des bornes de complexités intéressantes pour tirer profit de la structure des problèmes. Alors, la classe 4 avec un Δ maîtrisé, donne les meilleures stratégies de parcours des décompositions arborescentes.

CSP	3		4		5		6	
(a)	4,04	65	2,71	36	2,08	2	2,07	2
(b)	3,02	65	1,54	37	1,63	3	2,38	2
(c)	1,74	66	0,75	36	1,13	11	3,08	3
(d)	0,81	65	0,36	35	0,63	11	0,62	10
(e)	90,76	101	12,71	43	>51,67	2	>51,62	2
(f)	>152,45	102	7,61	44	8,39	2	8,57	2
(g)	>78,87	103	5,57	44	4,41	17	4,40	8
(h)	>87,78	103	19,60	44	3,17	18	3,24	13
(i)	19,26	69	6,91	48	6,65	1	14,76	1
(j)	10,29	70	13,12	49	3,36	6	3,74	3
(k)	4,83	71	3,25	50	3,52	15	3,25	3
(l)	>56,85	69	15,81	48	1,26	20	1,66	10
(m)	>466,65	152	>191,42	152	62,99	110	67,94	91
(n)	>90,71	121	8,17	62	4,41	2	4,37	2
(o)	22,75	122	3,41	63	3,94	5	4,70	3
EC	66,67		13,48		4,57		5,71	

Tab. 4.3 – Durées d'exécution en secondes de FC-BDH et valeurs de Δ pour différentes tailles maximum de séparateurs (entre 3 et 6) de l'heuristique de fusion *sep*. L'ordre sur les variables appartient à la classe 4 et est défini par les heuristiques mdd_{dyn}+*minesp*+$minesp_f$. EC donne l'écart moyen des résultats obtenus grâce aux différentes tailles maximum de séparateurs.

CSP	7		8		9		10	
(a)	2,09	2	2,47	2	2,50	1	2,55	1
(b)	2,43	2	6,65	2	6,31	2	1,79	2
(c)	3,54	3	1,51	2	5,42	2	3,91	2
(d)	0,45	8	0,46	3	1,01	3	0,58	2
(e)	>51,58	2	>51,67	2	>51,68	2	>51,60	2
(f)	8,73	2	9,37	2	10,37	2	15,82	2
(g)	>41,47	4	>41,38	4	>41,51	4	>41,00	4
(h)	3,36	10	3,59	6	6,99	6	8,03	5
(i)	18,81	1	19,02	1	20,07	1	19,99	1
(j)	3,80	3	3,92	3	4,21	3	5,27	2
(k)	3,43	3	3,31	2	3,47	2	4,97	2
(l)	5,99	6	13,01	4	11,33	3	15,38	3
(m)	44,41	72	44,03	62	35,83	30	53,33	6
(n)	4,52	2	4,54	2	5,13	1	5,47	1
(o)	5,80	2	6,40	2	6,63	2	6,45	2
EC	7,31		8,04		8,11		9,69	

Tab. 4.4 – Durées d'exécution en secondes de FC-BDH et valeurs de Δ pour différentes tailles maximum de séparateurs (entre 7 et 10) de l'heuristique de fusion *sep*. L'ordre sur les variables appartient à la classe 4 et est défini par les heuristiques mdd_{dyn}+*minesp*+$minesp_f$. EC donne l'écart moyen des résultats obtenus grâce aux différentes tailles maximum de séparateurs.

CSP	1	2	3	4	5	6	7	8
(a)	2,85	2,08	2,08	2,06	15,58	2,23	2,27	2,52
(b)	6,10	6,00	5,89	1,92	1,84	1,83	10,88	4,39
(c)	5,65	7,44	2,84	2,88	1,63	1,19	1,02	28,29
(d)	0,74	0,98	0,49	0,47	0,43	0,53	0,52	0,79
(e)	>52,18	>52,13	>51,66	11,59	11,66	11,65	11,03	10,19
(f)	>45,51	7,30	10,00	7,86	8,26	8,45	9,46	>47,16
(g)	16,30	17,23	2,85	>43,84	12,03	14,14	2,93	16,26
(h)	8,44	5,37	6,02	4,84	18,84	20,35	20,08	31,05
(i)	39,47	9,37	5,20	5,15	5,14	6,04	8,74	15,12
(j)	5,53	3,24	4,94	4,76	7,01	6,98	5,49	6,19
(k)	4,82	>40,84	3,95	4,22	30,09	>40,44	>42,04	>79,08
(l)	11,75	12,97	5,45	>40,39	>40,81	>73,36	>75,03	43,72
(m)	47,65	44,92	59,90	61,25	66,37	52,31	57,56	>96,71
(n)	4,63	4,66	4,43	6,33	5,44	14,49	11,17	14,11
(o)	6,99	12,80	9,69	9,70	3,64	7,42	>43,61	>48,23
EC	10,49	8,40	4,94	7,06	8,50	10,67	13,37	22,83

Tab. 4.5 – Durées d'exécution en secondes de FC-BDH pour différentes valeurs minimum du nombre de variables propres (entre 1 et 8) de l'heuristique de fusion *vp*. L'ordre sur les variables appartient à la classe 4 et est défini par les heuristiques mdd_{dyn}+$minesp$+$minesp_f$. EC donne l'écart moyen des résultats obtenus grâce aux différentes valeurs minimum du nombre de variables propres.

Le tableau 4.6 montre les durées d'exécution de FC-BDH avec les heuristiques de fusion *sep* (avec la valeur 5), $minesp_{fdyn} - sep$ (avec 5 comme borne maximum de la taille des séparateurs et $\Delta = \frac{\gamma}{2}$ pour la limitation de la taille maximal des clusters) et
vp+$minesp_{fdyn} - sep$ (avec 3 comme nombre minimum de variables propres autorisé, 5 comme borne maximum de la taille des séparateurs et $\Delta = \frac{\gamma}{2}$ pour la limitation de la taille maximal des clusters). Ces heuristiques sont associées aux heuristiques
$minesp$+$minesp_{fdyn}$+mdd_{dyn} pour définir des ordres de classe 4 pour *sep* et de classe 5 pour $minesp_{fdyn} - sep$ et vp+$minesp_{fdyn} - sep$.

On a pu voir précédemment que la classe 4 obtient de meilleurs résultats que les classes 1, 2 et 3. Ici, nous nous contentons de comparer la classe 5 à cette classe. Les heuristiques $minesp_{fdyn} - sep$ et vp+$minesp_{fdyn} - sep$ ont de très bons résultats au-dessus de ceux de la classe 4. Ils réussissent à résoudre toutes les instances de toutes les classes tandis qu'avec la classe 4 un problème de la classe (250,20,20,107,5,20,10) est non résolu en moins de 1800 s. Un meilleur hypergraphe de référence permet à l'heuristique vp+$minesp_{fdyn} - sep$ d'avoir les meilleurs résultats. Cette étude met en lumière la nécessité d'avoir un bon hypergraphe de référence. En outre, une exploitation dynamique de ce dernier conduit à une amélioration significative du comportement.

4.7.2 Comparaison des heuristiques sur les VCSP

La comparaison a été menée en réalité sur les WCSP aléatoires structurés définies dans le chapitre précédent. Les expérimentations ont été menées sur les mêmes machines que dans le cas CSP. Les résultats sont les moyennes sur 30 instances. MCS est utilisé pour calculer l'hypergraphe acyclique de référence.

Le tableau 4.7 donne les durées d'exécution de FC-BDH-val avec différentes heuristiques des classes 1, 2 and 3 sur des instances aléatoires structurées. Nous observons de manière très nette que le choix de la racine est primordiale comparée à l'ordonnancement des fils. En effet, les résultats sont très similaires avec un même choix de racine. Les différences majeures entre les

CSP	Classe 4	Classe 5	Classe 5
	sep	$minesp_{fdyn} - sep$	$vp{+}minesp_{fdyn} - sep$
(a)	2,08	1,78	1,93
(b)	1,63	1,23	1,12
(c)	1,13	1,40	1,30
(d)	0,63	1,17	0,27
(e)	>51,67	10,49	12,89
(f)	8,39	6,97	13,18
(g)	4,41	3,62	2,80
(h)	3,17	6,99	3,97
(i)	6,65	4,19	3,79
(j)	3,36	2,76	1,69
(k)	3,52	3,73	2,61
(l)	1,26	13,87	11,88
(m)	62,99	82,07	78,16
(n)	4,41	2,23	2,00
(o)	3,94	3,01	3,23
EC	>3,61	2,70	2,38

Tab. 4.6 – Durées d'exécution en secondes de FC-BDH sur des CSP aléatoires structurés pour différentes heuristiques de fusion associées aux heuristiques $mdd_{dyn}{+}minesp{+}minesp_{fdyn}$: *sep* (avec la valeur 5) définit un ordre de la classe 4, $minesp_{fdyn} - sep$ (*sep* avec la valeur 5 et $\Delta = 0.5\alpha$) un ordre de la classe 5 et $vp{+}minesp_{fdyn} - sep$ (*vp* avec la valeur 3, *sep* avec la valeur 5 et $\Delta = 0.5\alpha$) un ordre de la classe 5. EC donne l'écart moyen des résultats de l'ordre avec les meilleurs résultats obtenus dans les classes 4 et 5.

heuristiques sont observées pour des choix de racine différentes. L'effet limité des heuristiques de choix de fils est sans doute lié au fait que les instances générées ont un nombre de fils réduit. Les heuristiques *card* et *minesp* obtiennent souvent les meilleurs résultats, même s'il leur arrive de faire de mauvais choix. L'heuristique *pv* donne de bons résultats, excepté une instance non résolue pour cause de dépassement de la mémoire disponible. L'heuristique *bary* est la plus robuste ($EC = 226,67$) : elle obtient de bons résultats et constitue l'unique heuristique qui soit parvenue à résoudre toutes les instances de la classe $(75, 10, 15, 30, 5, 8, 10)$, les autres nécessitant trop d'espace mémoire.

L'espace mémoire requis peut être réduit grâce aux ordres de la classe 4 avec l'heuristique de fusion *sep*. Le tableau 4.8 présente les durées d'exécution de FC-BDH-val pour des ordres de cette classe avec une taille de séparateur bornée par 5. Une analyse fine montre que la valeur du paramètre Δ est assez limitée (entre 1 et 3). Les résultats de cette classe améliore significativement ceux des classes 2 et 3. Comme précédemment, le choix de la racine demeure primordiale. Les heuristiques de cette classes sont très proches. $minesp + minesp_{fdyn}$ est la plus robuste avec $EC = 0,64s$. Les résultats de $minesp + minesp_f$ sont similaires à ceux de $minesp + minesp_{fdyn}$. L'heuristique $pv{+}pv_{fdyn}$ a des performances d'une grande proximité avec les meilleures.

4.8 Conclusion

Nous avons construit une méthode structurelle BDH qui prend en compte la nécessité impérieuse d'exploiter des heuristiques efficaces de choix de variables. Dans cette optique, elle exploite la notion de recouvrement acyclique de CSP qui est légèrement plus générale que celle de décomposition arborescente. Cette notion nous a permis de sortir du choix arbitraire d'un arbre d'une décomposition arborescente. Dorénavant, il est possible de construire de manière incrémentale un arbre durant la résolution en tenant compte de l'évolution du problème pour faire des choix pertinents. Mais également, BDH rend possible l'exploitation de plusieurs arbres à différents moments de la résolution.

VCSP	Classe 1	Classe 2			Classe 3		
	card $minsep_f$	*bary* $minsep_f$	*minesp* $minesp_f$	*card* $minsep_f$	*minesp* $minesp_{fdyn}$	*card* pv_{fdyn}	*pv* pv_{fdyn}
(75,10,15,30,5,8,10)	Mem	22,31	Mem	Mem	Mem	Mem	Mem
(75,10,15,30,5,8,20)	3,27	4,77	6,13	3,34	6,24	2,88	Mem
(75,10,15,33,3,8,10)	8,30	6,16	7,90	8,67	7,87	8,82	5,36
(75,10,15,34,3,8,20)	2,75	2,29	3,42	2,82	3,52	2,84	2,14
(75,10,10,40,3,10,10)	11,81	1,33	3,02	11,89	4,73	11,87	1,43
(75,10,10,42,3,10,20)	1,02	0,67	0,76	1,02	0,83	1,03	0,79
(75,15,10,102,3,10,10)	11,76	3,74	12,10	12,07	12,09	11,70	4,93
(100,5,15,13,5,10,10)	Mem	Mem	Mem	Mem	Mem	Mem	Mem
EC	451,37	226,67	451,05	451,48	450,92	451,41	673,34

Tab. 4.7 – Durées d'exécution en secondes de FC-BDH-val sur des VCSP aléatoires structurés avec différentes heuristiques des classes d'ordres 1, 2 et 3 : la première ligne en dessous des classes donne l'heuristique de choix de racines et la seconde celle d'ordonnancement des fils. *mdd* est utilisée pour le choix de variables à l'intérieur des clusters pour la classe 1 et mdd_{dyn} pour les classes 2 et 3. *EC* donne l'écart moyen des résultats de l'ordre avec les meilleurs résultats obtenus dans les classes 1, 2, 3 et 4.

Cette construcion incrémentale de décompositions arborescentes repose sur une nouvelle définition d'acyclicité dans les hypergraphes. Nous avons proposé la notion de α-cycle dont l'absence d'un hypergraphe équivaut à son acyclicité. Ce résultat joue un rôle prépondérant dans la méthode BDH. Par ailleurs, il contribue à combler un vide surprenant dans la mesure où les définitions antérieures d'acyclicité des hypergraphes font référence pour la plupart aux articulations des hypergraphes ou à des propriétés sur des représentations en graphe des interconnections entre hyperarêtes.

Contrairement à BTD qui restreint le choix de la prochaine variable à instancier au sein de l'hyperarête courant, BDH permet d'accroître la liberté laissée à l'heuristique de choix variables par la fusion d'hyperarêtes. Pour réaliser un compromis entre cette liberté accordée à l'heuristique de choix de variables et la nécessité d'exploiter la structure du problèmes pour éviter un certain nombre de redondances, un paramètre Δ est introduit pour une limitation de la taille des nouvelles hyperarêtes obtenues par fusion. Par ailleurs, il faut observer la possibilité de procéder à une fusion d'hyperarêtes pour en même temps accroître la dynamicité de l'heuristique de choix de variables et réduire la taille des séparateurs exploités qui demeure sans nul doute le critère le plus pertinent pour une résolution efficace. Par ailleurs, la fusion d'hyperarêtes rentre dans un cadre plus général donné par le recouvrement d'hypergraphes acycliques. Nous avons proposé plusieurs classes de recouvrements d'un hypergraphe acyclique et étudié leurs propriétés. Il s'est avéré que les recouvrements qui conservent l'ensemble des séparateurs de l'hypergraphe de départ étaient les mieux indiqués pour une résolution efficace en pratique car ils entraînent souvent la réduction de l'espace mémoire requis.

Ainsi, BDH est restreint à ce type de recouvrements.

Suivant la valeur de Δ et les arbres construits, la complexité en temps de BDH balaie un large spectre allant de $O(exp(\gamma))$ (la complexité de BTD) à $O(nombre_{T_c}.exp(\gamma + \Delta + 1))$, avec $nombre_{T_c}$ le nombre d'arbres utilisés par BDH. Nous avons réalisé une étude expérimentale sur des CSP aléatoires structurés avec différentes heuristiques de fusion, de choix d'hyperarêtes et de variables. Les multiples combinaisons de ces heuristiques définissent des ordres appartenant à différentes classes qui induisent des complexités différentes. Nous avons pu noter l'énorme gouffre qui sépare les performances moyennes des ordres statiques et des ordres plus dynamiques basés sur l'utilisation d'un unique arbre, des heuristiques dynamiques de fusion, de choix d'hyperarêtes et de choix de variables au sein de ces dernières.

Nous avons également défini une extension BDH-val de BDH aux VCSP. Nous retrou-

VCSP	Classe 4				
	minesp $minesp_f$	*card* $minsep_f$	*minesp* $minesp_{fdyn}$	*card* nv_{fdyn}	*pv* pv_{fdyn}
(75,10,15,30,5,8,10)	9,42	18,99	8,69	18,30	16,77
(75,10,15,30,5,8,20)	1,65	1,67	1,56	1,53	2,32
(75,10,15,33,3,8,10)	5,22	4,31	5,26	4,18	3,24
(75,10,15,34,3,8,20)	1,50	1,61	1,48	1,58	1,40
(75,10,10,40,3,10,10)	0,58	0,81	0,58	0,85	0,52
(75,10,10,42,3,10,20)	0,41	0,42	0,51	0,41	0,38
(75,15,10,102,3,10,10)	5,50	4,73	5,41	4,63	3,13
(100,5,15,13,5,10,10)	9,40	9,05	9,60	9,10	11,71
EC	0,72	1,71	0,64	1,58	1,44

Tab. 4.8 – Durées d'exécution en secondes de FC-BDH-val sur des VCSP aléatoires structurés avec différentes heuristiques de la classe d'ordres 4 et une décomposition modifiée par l'heuristique de fusion *sep* (avec une taille de séparateurs limitée à au plus 5) : la première ligne donne l'heuristique de choix de racine et la seconde celle d'ordonnancement des fils. mdd_{dyn} est utilisée pour le choix de variables à l'intérieur des clusters. *EC* donne l'écart moyen des résultats de l'ordre avec les meilleurs résultats obtenus dans les classes 1, 2, 3 et 4.

vons là également un large spectre de bornes de complexité temporelle théorique. Notre étude expérimentale sur des VCSP aléatoires structurés a confirmé les observations qui ont été faites dans le cadre CSP.

Une large étude de validation est entamée sur des problèmes réels tels que ceux d'allocation de fréquences (radio-link) de l'archive FullRLFAP ([CdGL$^+$99]). Elle doit permettre la confirmation des observations faites sur les premiers résultats présentés.

Il faudra également mener une étude sur une implémentation efficace de la méthode de construction incrémentale des arbres de décompositions arborescentes. Elle doit intégrer le coût de calcul des relations de α-voisinage entre hyperarêtes sur lesquelles repose la notion de α-cycle.

Chapitre 5

SBBT : un cadre générique pour la résolution de CSP

5.1 Introduction

Le travail accompli dans les chapitres précédents a été validé expérimentalement avec la méthode BDH. Mais, l'objectif de cette étude qui est de rendre opérationnelles les techniques structurelles de résolution de (V)CSP, va au-delà de cette méthode. Malgré tout, cette validation à travers BDH ne nous fait pas perdre en généralité. En effet, à travers ce chapitre, nous allons définir un schéma générique d'algorithmes qui permet de retrouver dans un même cadre une bonne partie des méthodes structurelles présentées dans l'état de l'art, de même qu'une bonne partie des méthodes énumératives. Ce schéma est donc basé sur un ensemble quelconque de séparateurs du graphe de contraintes des CSP binaires auxquels nous allons nous restreindre ici. Il utilise également les notions de goods et nogoods structurels dont les définitions seront étendues. En plus de démontrer les grandes similitudes existant entre les méthodes structurelles, ce cadre met en lumière les différences entre les méthodes structurelles énumératives telles que BDH et les méthodes purement énumératives telles que FC. Ces différences résident essentiellement dans l'ordre d'affectation des variables qui, dans le cas des méthodes structurelles, est induit par la décomposition du graphe de contraintes du (V)CSP et qui, dans les cas des méthodes purement énumératives, peut être défini de manière totalement libre. Une autre différence se situe dans la possibilité, au sein des approches structurelles, de mémoriser des informations basées sur des propriétés topologiques du problème (principalement sur les séparateurs du graphe de contraintes). Notre schéma permet donc la définition de nouvelles méthodes structurelles visant un bon compromis entre les bornes de complexité en temps, la limitation de l'espace mémoire disponible et la liberté accordée aux heuristiques d'ordonnancement de variables. Cela grâce à sa capacité à utiliser des heuristiques totalement dynamiques ou totalement statiques, mais aussi toutes les possibilités comprises entre ces deux extrêmes et basées sur un ensemble de séparateurs qui vérifie des propriétés topologiques plus ou moins intéressantes. Il capture ainsi des algorithmes purement énumératifs tels que BT et des méthodes structurelles telles que BTD, BDH, And/Or Search Graph. La version standard du cadre est définie sur la technique du backtracking standard augmentée d'un certain nombre d'améliorations telles que le retour en arrière non chronologique, la décomposition du CSP à travers un ensemble de séparateurs, l'apprentissage de goods et nogoods structurels. Cependant, des techniques de filtrage en prétraitement et de consistance avant de type FC ou MAC peuvent être très facilement intégrées à ce cadre. Dans un premier temps, nous allons présenter le schéma, pour ensuite voir de quelle manière, il capture de nombreuses méthodes.

5.2 SBBT : un schéma générique d'algorithmes énumératifs

5.2.1 Formalisation et justifications

Dans cette section, nous allons introduire des généralisations des définitions de goods et nogoods structurels et donner les résultats théoriques qui en découlent. Le schéma générique que nous proposons, appelé SBBT (pour Separator Based BackTracking), exploite les séparateurs du graphe de contraintes du CSP pour mémoriser des (no)goods structurels. La définition de ces derniers dans le cadre de la méthode BTD repose sur les intersections entre les clusters d'une décomposition arborescente tandis que pour la méthode BDH il s'agit d'intersections entre hyperarêtes d'un recouvrement acyclique. Ces intersections sont des séparateurs du graphe de contraintes du CSP considéré. Notre objectif est d'utiliser les séparateurs à l'image de ces intersections entre clusters. Notre généralisation des définitions de goods et nogoods utilise directement les séparateurs du graphe de contraintes du CSP considéré et les indépendances entre les composantes connexes induites par ces séparateurs. Nous conservons une propriété essentielle résidant sur le fait que certaines parties de l'espace de recherche ne seront pas revisitées puisque leur (in)consistance sera connue. Dans la suite, nous considérerons un CSP $\mathcal{P} = (X, D, C, R)$ et son graphe de contraintes $G = (X, C)$. Etant donné un séparateur S_i de G, CC_{k,S_i} représentera une composante connexe de $G[X \backslash S_i]$.

Définition 5.2.1 *Une surcomposante connexe induite par un séparateur S_i est un ensemble $SP_{k,S_i} = CC_{k,S_i} \cup S_i$, où CC_{k,S_i} est la k-ième composante connexe de $G[X \backslash S_i]$.*

Les composantes connexes CC_{k,S_i} induites par un séparateur S_i, définissent des sous-problèmes indépendants. Autrement dit, il n'existe aucune contrainte reliant deux variables de deux composantes connexes différentes. Par exemple, le séparateur $S_1 = \{x_3\}$ déconnecte le graphe de la figure 5.1 en deux composantes connexes $CC_{1,S_1} = \{x_1, x_2, x_4, x_{10}, \ldots, x_{14}\}$ et $CC_{2,S_1} = \{x_5, \ldots, x_9\}$. Les surcomposantes connexes induites par S_1 sont $SP_{1,S_1} = \{x_1, x_2, x_4, x_{10}, \ldots, x_{14}, x_3\}$ et $SP_{2,S_1} = \{x_5, \ldots, x_9, x_3\}$.

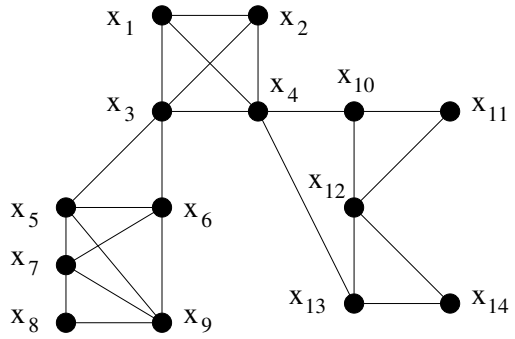

Fig. 5.1 – Un graphe.

Nous allons définir un good (resp. nogood) structurel comme une affectation consistante sur un séparateur, qui peut (resp. ne peut pas) être prolongée de manière consistante sur une composante connexe induite par ce séparateur. De ce fait, un (no)good structurel est associé à une paire formée par un séparateur et une de ses composantes connexes qui constitue une surcomposante connexe. Cela donne une liberté accrue par rapport à la définition originale dans la mesure où nous avons la possibilité d'enregistrer des (no)goods associés à toutes les surcomposantes connexes induites par un séparateur. Ceci n'est pas le cas des (no)goods usuels. Pour permettre à SBBT de capturer une méthode de type BDH, il est important de pouvoir

limiter les surcomposantes connexes induites par un séparateur auxquels seront associés les (no)goods structurels de ce séparateur. Dans cette optique, nous allons définir la notion de séparateur orienté.

Définition 5.2.2 *Un couple (S_i, SP_{l,S_i}) formé d'un séparateur S_i et d'une surcomposante connexe SP_{l,S_i} induite par ce séparateur, est un séparateur orienté.*

Dans le cas d'un séparateur orienté (S_i, SP_{l,S_i}), nous allons interdire l'enregistrement de (no)goods associés à la surcomposante SP_{l,S_i}.

Définition 5.2.3 *Un ensemble de séparateurs est orienté si tous ses séparateurs sont orientés et s'il contient un séparateur orienté (S_r, SP_{r',S_r}) tel que pour tout autre élément (S_i, SP_{l,S_i}) de l'ensemble, SP_{l,S_i} contient S_r.*
(S_r, SP_{r',S_r}) est appelé séparateur racine.

Définition 5.2.4 *Soit (S_i, SP_{l,S_i}) un séparateur orienté. Une surcomposante connexe orientée induite par S_i est une surcomposante connexe SP_{t,S_i} différente de SP_{l,S_i}.*

Pour le graphe de la figure 5.1, $S_1 = \{x_3\}$, $S_2 = \{x_4\}$, $S_3 = \{x_5, x_6\}$ et $S_4 = \{x_7, x_9\}$ sont des séparateurs. Les surcomposantes induites par ces séparateurs sont :
$SP_{1,S_1} = \{x_3, x_5, x_6, x_7, x_8, x_9\}$, $SP_{2,S_1} = \{x_3, x_1, x_2, x_4, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}\}$
$SP_{1,S_2} = \{x_4, x_1, x_2, x_3, x_5, x_6, x_7, x_8, x_9\}$, $SP_{2,S_2} = \{x_4, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}\}$
$SP_{1,S_3} = \{x_5, x_6, x_7, x_8, x_9\}$, $SP_{2,S_3} = \{x_5, x_6, x_3, x_1, x_2, x_4, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}\}$
$SP_{1,S_4} = \{x_7, x_9, x_8\}$, $SP_{2,S_4} = \{x_7, x_9, x_5, x_6, x_3, x_1, x_2, x_4, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}\}$.

Pour construire un ensemble orienté à partir de ces séparateurs, il suffit de choisir un séparateur racine et de décider de son orientation. Prenons S_3 comme racine orientée depuis la surcomposante SP_{1,S_3}. Cette racine impose l'orientation des autres séparateurs qui seront associés à leur surcomposante connexe contenant S_3.
$\{(S_1, SP_{1,S_1}), (S_2, SP_{1,S_2}), (S_3, SP_{1,S_3}), (S_4, SP_{2,S_4})\}$ est donc un ensemble orienté de séparateurs de la figure 5.1.

Les surcomposantes orientées induites par cet ensemble orienté de séparateurs sont : SP_{2,S_1}, SP_{2,S_2}, SP_{2,S_3} et SP_{1,S_4}

Dans un ensemble de séparateurs orientés, les (no)goods structurels mémorisés sont associés à une surcomposante connexe orientée.

Avant de présenter notre généralisation des définitions de goods et nogoods structurels, nous allons donner les justifications de cette généralisation. Le théorème 5.2.1 établit que les interactions entre les sous-problèmes induits par les surcomposantes connexes s'effectuent via le séparateur. Ainsi, les affectations sur ces sous-problèmes sont compatibles à partir du moment où elles sont égales au niveau du séparateur.

Théorème 5.2.1 *Soient S_i un séparateur, SP_{k_1,S_i} et SP_{k_2,S_i} deux surcomposantes connexes induites par S_i. Une affectation $\mathcal{A}_1$ sur SP_{k_1,S_i} et une affectation $\mathcal{A}_2$ sur SP_{k_2,S_i} sont compatibles ssi $\mathcal{A}_1[S_i] = \mathcal{A}_2[S_i]$.*

Preuve : Puisque CC_{k_1,S_i} et CC_{k_2,S_i} induisent deux sous-problèmes indépendants, la compatibilité de deux affectations sur ces sous-problèmes passent forcément par les variables de S_i. Donc, ces affectations sont compatibles ssi elles sont égales sur S_i. □

Considérons une affectation consistante $\mathcal{A}$ sur un séparateur S_i. Soit SP_{k,S_i} une surcomposante induite par S_i. Deux cas peuvent se produire. Si $\mathcal{A}$ ne possède aucune extension consistante sur CC_{k,S_i}, cette inconsistance provient uniquement de la violation de contraintes liant deux variables de CC_{k,S_i} ou une variable de S_i et une de CC_{k,S_i} car seul S_i connecte CC_{k,S_i} au reste du problème. Aussi, cette affectation sur S_i peut être considérée comme un nogood structurel puisque qu'aucune affectation partielle $\mathcal{B}$ avec $\mathcal{B}[S_i] = \mathcal{A}$ ne peut être étendue en une affectation

consistante sur CC_{k,S_i}. De même, si $\mathcal{A}$ possède une extension consistante sur CC_{k,S_i}, cette affectation sur S_i peut être considérée comme un good structurel car n'importe quelle affectation $\mathcal{B}$ avec $\mathcal{B}[S_i] = \mathcal{A}$ peut être étendue de façon consistante sur CC_{k,S_i}. Ces notions de (no)goods structurels relatifs à une surcomposante connexe sont formellement définies ainsi :

Définition 5.2.5 *Soit S_i un séparateur. Un good (resp. nogood) structurel relatif à une surcomposante connexe SP_{k,S_i} est une affectation consistante sur S_i qui peut (resp. ne peut pas) être étendue de façon consistante sur le sous-problème induit par CC_{k,S_i}.*

Une variable x est dite *instanciable par le good* $\mathcal{A}$ relatif à SP_{k,S_i} si $x \in CC_{k,S_i}$. Le théorème 5.2.2 montre que certaines parties de l'espace de recherche peuvent être coupées grâce aux (no)goods structurels.

Théorème 5.2.2 *Soient S_i un séparateur, $\mathcal{A}$ une affectation sur S_i et $\mathcal{B}$ une affectation partielle consistante sur $X - CC_{k,S_i}$. Si $\mathcal{A}$ est un good (resp. un nogood) relatif à SP_{k,S_i} et $\mathcal{B}[S_i] = \mathcal{A}$, alors $\mathcal{B}$ peut (resp. ne peut pas) être étendue de façon consistante sur CC_{k,S_i}.*

Preuve : Si $\mathcal{A}$ est un good, alors $\mathcal{A}$ peut s'étendre de façon consistante sur CC_{k,S_i}. Soit $Sol_{\mathcal{A},SP_{k,S_i}}$ l'affectation consistante sur SP_{k,S_i} relative à ce good. Comme $\mathcal{B}[S_i] = \mathcal{A}$, $Sol_{\mathcal{A},SP_{k,S_i}}$ et $\mathcal{B}$ sont compatibles (d'après le théorème 5.2.1). Ainsi, $\mathcal{B}$ peut s'étendre de façon consistante sur CC_{k,S_i}.

Si $\mathcal{A}$ est un nogood, $\mathcal{A}$ ne peut s'étendre de façon consistante sur CC_{k,S_i}. Puisque $\mathcal{B}[S_i] = \mathcal{A}$, si $\mathcal{B}$ possédait une extension consistante sur SP_{kS_i}, ce serait également une extension consistante du nogood (d'après le théorème 5.2.1), ce qui est impossible. Ainsi, il n'existe aucune extension consistante de $\mathcal{B}$ sur CC_{k,S_i}. □

5.2.2 Algorithme SBBT

Dans le schéma SBBT (15), $\mathcal{A}$ représente l'affectation partielle courante (qui est consistante), V l'ensemble des variables non instanciées, V_g l'ensemble des variables instanciables par des goods, x la variable courante, D_x le domaine initial de x, d son domaine courant, v la valeur courante de x, J l'ensemble des variables ayant causé les échecs précédents lors des tentatives d'extension de l'affectation courante. SBBT inclut plusieurs fonctions ou procédures.

- $Heuristic_{var}$ est l'heuristique d'ordonnancement des variables. Elle peut être définie de différentes manières afin d'exploiter plus ou moins la structure du problème. $Heuristic_{val}$ est l'heuristique d'ordonnancement des valeurs.
- *Check_Good_Nogood*$(\mathcal{A}', x, V, V'_g, J)$ vérifie pour chaque séparateur S_j devenant complètement instancié dans la nouvelle affectation courante $\mathcal{A}'$, si $\mathcal{A}'[S_j]$ est un good ou un nogood relatif au sous-problème SP_{k,S_j}. Dans le cas où $\mathcal{A}'[S_j]$ est un nogood relatif à SP_{k,S_j}, les variables de SP_{k,S_j} sont ajoutées à J car SP_{k,S_j} contient forcément les variables causant l'échec actuel. Ensuite, $false$ est retourné signifiant que, l'affectation $\mathcal{A}'$ ne peut pas conduire à une solution puisqu'elle contient un nogood.
  Dans le cas où $\mathcal{A}'[S_j]$ est un good relatif à SP_{k,S_j}, les variables de SP_{k,S_j} sont ajoutées à V'_g. Cet ensemble est retourné à SBBT si $\mathcal{A}'$ ne contient aucun nogood et ainsi ces variables deviennent instanciables grâce à des goods.
- *Good_Recording*$(\mathcal{A}', x, V, V_g, J)$ enregistre $\mathcal{A}'[S_j]$ comme un good relatif à SP_{k,S_j} pour chaque SP_{k,S_j} devenant complètement instancié dans l'affectation courante.
- *Good_Cancel*(x, V, V_g) retire de V_g toutes les variables instanciables grâce à des goods contenant la variable x dont la valeur est sur le point d'être changée dans SBBT.
- La procédure *Failure*$(\mathcal{A}', x)$ retourne un ensemble de variables contenant celles qui sont actuellement en cause dans l'échec. Elle utilise la technique CBJ [Pro93] pour déterminer cet ensemble de variables.

– *Nogood_Recording*$(\mathcal{A}, x, V, J)$ enregistre $\mathcal{A}[S_j]$ comme un nogood relatif à SP_{k,S_j} pour chaque séparateur S_j complètement instancié dans $\mathcal{A}$ tel que $x \in CC_{k,S_j}$ et CC_{k,S_j} est complètement instancié et est impliqué dans les raisons de l'échec (dans J).

A titre d'exemple, les algorithmes 16 à 20 proposent une implémentation possible des différentes fonctions ou procédures respectant les spécifications énoncées ci-dessus. Elles permettent de définir une nouvelle méthode de résolution.

Le schéma générique d'algorithmes énumératifs SBBT (algorithme 15) résout récursivement le problème avec comme entrées $\mathcal{A}$, V et V_g. Il repose sur un ensemble de séparateurs et les surcomposantes connexes associées. Dans le cas où les séparateurs sont orientés, seules les surcomposantes connexes orientées sont considérées. SBBT renvoie $\emptyset$ si l'affectation $\mathcal{A}$ admet une extension consistante sur V, un ensemble J de variables causant les échecs sinon. $Heuristic_{var}$ choisit dans $V - V_g$ la prochaine variable x à instancier (ligne 3). Si le domaine courant d de x n'est pas vide, $Heuristic_{val}$ choisit une valeur v dans d. Dans le cas où l'extension $\mathcal{A}'$ de $\mathcal{A}$ n'est pas consistante, *Failure* ajoute à J l'ensemble (ou un sur-ensemble) des variables impliquées dans l'échec (ligne 16) et une nouvelle valeur (s'il en reste) est choisie grâce à $Heuristic_{val}$. Si $\mathcal{A}'$ est consistante, *Check_Good_Nogood*$(\mathcal{A}', x, V, V'_g, J)$ renvoie $false$ si $\mathcal{A}'$ contient un nogood impliquant l'affectation courante de x. $Heuristic_{val}$ choisit à nouveau une nouvelle valeur si le domaine de x n'est pas vide. Si aucun nogood n'est trouvé, *Check_Good_Nogood* retourne *true* ainsi que l'ensemble V'_g contenant les variables instanciables grâce à des goods impliquant l'affectation courante de x. Ces variables sont alors ajoutées dans V_g. A la ligne 11, *Good_Recording* mémorise les éventuels nouveaux goods contenant x. Ensuite, SBBT est appelé récursivement par l'appel SBBT$(\mathcal{A}', V - \{x\}, V_g)$. Si $\mathcal{A}'$ ne possède aucune extension consistante, l'ensemble J' des variables impliquées dans l'échec est renvoyé et la valeur courante de x doit être changée. Puis, on retire de V_g les variables instanciables par des goods impliquant x. Si x est une des causes de l'échec, SBBT ajoute J' à J et une nouvelle valeur est choisie pour x (s'il en reste). Sinon, $J = J'$ et on effectue un saut en arrière vers une des variables causant l'échec (d'après J). Enfin, si le domaine courant de x est vide ou qu'un saut en arrière se produit, *Nogood_Recording*$(\mathcal{A}, x, V, J)$ mémorise d'éventuels nouveaux nogoods contenant x puis SBBT renvoie J.

Théorème 5.2.3 *SBBT est correct, complet et termine.*

Preuve : SBBT repose sur BT qui est correct, complet et termine. Aussi, nous devons prouver que ces propriétés de BT ne sont pas altérées par les coupes réalisées grâce aux (no)goods et au backjumping de SBBT. Un good est mémorisé lorsqu'un sous-problème induit par un SP_{k,S_i} est complètement instancié dans l'affectation courante $\mathcal{A}$. Par conséquent, $\mathcal{A}[S_i]$ possède une extension consistante sur CC_{k,S_i}. Ainsi $\mathcal{A}[S_i]$ est un good structurel par rapport au sous-problème SP_{k,S_i}. Pour n'importe quelle affectation $\mathcal{B}$ telle que $\mathcal{B}[S_i] = \mathcal{A}[S_i]$, nous savons que $\mathcal{B}$ peut s'étendre de façon consistante sur CC_{k,S_i} (d'après le théorème 5.2.2). Donc, continuer la recherche sur $V \backslash SP_{k,S_i}$ est correcte.

Concernant l'enregistrement de nogoods, nous savons que si certaines variables de CC_{k,S_i} sont instanciées avant toutes les variables de S_i, nous ne pouvons pas mémoriser l'affectation sur S_i comme un nogood lorsqu'elle ne peut pas s'étendre de façon consistante sur CC_{k,S_i}. En effet, SBBT n'a pas essayé toutes les affectations possibles sur CC_{k,S_i} quand il revient en arrière sur S_i. Par conséquent, un nogood n'est mémorisé que lorsque S_i est complètement instancié avant tout autre variable d'un sous-problème induit par un CC_{k,S_i} dans l'affectation courante $\mathcal{A}$ et que les raisons de l'échec de l'extension de $\mathcal{A}$ sur CC_{k,S_i} sont incluses dans le sous-problème induit par SP_{k,S_i}. Aussi, puisque $\mathcal{A}[S_i]$ ne peut être étendue de façon consistante sur CC_{k,S_i}, $\mathcal{A}[S_i]$ est un nogood structurel. Pour n'importe quelle autre affectation $\mathcal{B}$ avec $\mathcal{B}[S_i] = \mathcal{A}[S_i]$, nous savons que $\mathcal{B}$ ne peut s'étendre de façon consistante sur CC_{k,S_i} (d'après le théorème 2). Aussi, nous pouvons revenir en arrière car l'affectation courante ne peut s'étendre en une solution.

Enfin, lorsque SBBT échoue dans l'extension de l'affectation courante avec la variable x, il backjumpe sur la dernière variable instanciée dans J, l'ensemble (ou le sur-ensemble) de variables en cause dans l'échec. Les raisons de l'échec étant dans J, revenir ailleurs que sur cette variable

Algorithme 15 : SBBT(in : $\mathcal{A}, V$, in/out : V_g)

```
1  if V − V_g = ∅ then return ∅
2  else
3      x ← Heuristic_var(V − V_g)
4      d ← D_x ; J ← ∅ ; Backjump ← false
5      while d ≠ ∅ and Backjump = false do
6          v ← Heuristic_val(d)
7          d ← d − {v} ; A' ← A ∪ {x ← v}
8          if A' satisfait toutes les contraintes then
9              if Check_Good_Nogood (A', x, V, V'_g, J) then
10                 V_g ← V_g ∪ V'_g
11                 Good_Recording(A', x, V, V_g)
12                 J' ←SBBT(A', V − {x}, V_g)
13                 Good_Cancel(x, V, V_g)
14                 if x ∈ J' then  J ← J ∪ J'
15                 else  J ← J' ; Backjump ← true
16         else J ← J ∪ Failure(A', x)
17     Nogood_Recording (A, x, V)
18     return J
```

Algorithme 16 : Failure(in :$\mathcal{A}', x$)

```
1  return {x} ∪ {y ∉ V | ∃c ∈ C, X_c = {x, y} et A' viole c}
```

Algorithme 17 : Check_Good_Nogood(in :$\mathcal{A}', x, V$,in/out :V'_g, J)

```
1  V'_g ← ∅
2  forall S_j ∈ Sep t.q. S_j ∩ V = {x} do
3      forall SP_{k,S_j} do
4          switch A'[S_j] do
5              case good relatif à SP_{k,S_j}
6                  V'_g ← V'_g ∪ CC_{k,S_j}
7              case  nogood relatif à SP_{k,S_j}
8                  J ← J ∪ SP_{k,S_j} ; return false
9  return true
```

Algorithme 18 : Nogood_Recording (in :$\mathcal{A}, x, V$)

```
1  forall S_j ∈ Sep t.q. S_j ∩ V = ∅ do
2      forall CC_{k,S_j} t.q. x ∈ CC_{k,S_j} do
3          if J ∩ CC_{k,S_j} ≠ ∅ and CC_{k,S_j} ⊂ V then
4              Enregistrer A[S_j] comme un nogood relatif à SP_{k,S_j}
```

conduira au même échec. Les propriétés de BT n'étant pas altérées par les coupes ajoutées, SBBT est correct, complet et termine. □

Les algorithmes 16 à 20 donnent une implémentation possible (dite usuelle) des différentes fonctions ou procédures. Il est possible de les définir de manière différente tout en respectant les spécifications données.

La fonction *Failure* peut être définie de manière naïve (22), en retournant l'ensemble des variables déjà instanciées ($X_{\mathcal{A}'}$), ou en utilisant des techniques plus évoluées telles que GBJ [Dec90] (algorithme 21) ou l'algorithme 23. Ce dernier permet un retour en arrière non chronologique qui se limite à éviter les sous-problèmes totalement affectés indépendants de celui qui contient la variable courante.

De manière analogue, le $Pack(Good)$ formé par les procédures *Good_Recording*, *Good_Cancel* et la partie de la procédure *Check_Good_Nogood* consacrée à la recherche de goods, peut être défini de manière analogue aux algorithmes présentés ici, mais également vide pour empêcher la mémorisation de goods.

On peut procéder de la même manière au niveau du $Pack(Nogood)$ formé par la procédure *Nogood_Recording* et la partie de la procédure *Check_Good_Nogood* consacrée à la recherche de nogoods.

En outre, la version de SBBT présentée, est basée sur la technique du backtracking standard augmentée d'un certain nombre d'améliorations telles que le retour en arrière non chronologique, la décomposition du problème à travers un ensemble de séparateurs et l'apprentissage de goods et nogoods structurels. Cependant, des techniques de filtrage peuvent lui être associées tant qu'elles ne modifient pas la structure du CSP. On peut donc rajouter au cadre très simplement, un filtrage du type de la consistance d'arc, en prétraitement, mais également en consistance avant à l'image des méthodes FC et MAC.

5.3 Analyse de complexité

La complexité de notre schéma générique SBBT dépend de l'ensemble de séparateurs et des fonctions et procédures employées. Par exemple, l'algorithme BT peut être obtenu dans SBBT en employant un ensemble vide de séparateurs, des $Pack(Good)$ et $Pack(Nogood)$ vides et une fonction *Failure* naïve retournant simplement $X_{\mathcal{A}'}$. Utiliser un retour en arrière chronologique conduit généralement à rencontrer plusieurs fois les mêmes échecs. Dans SBBT, ces redondances peuvent être évitées en définissant et en revenant en arrière dans un ensemble de variables causant réellement les échecs (structure de backjump (lignes 14-15) et fonction *Failure*). Cet ensemble peut être calculé de différentes façons (par exemple en exploitant la formule de CBJ [Pro93] ou de GBJ [Dec90]). De plus, l'ensemble de séparateurs et les fonctions ou procédures *Check_Good_Nogood*, *Good_Cancel*, *Good_Recording* et *Nogood_Recording* rendent également possible la réduction de l'espace de recherche à explorer en exploitant des propriétés sémantiques et topologiques du problème. Certaines parties de l'espace de recherche seront élaguées aussitôt leur (in)consistance connue. Par dessus tout, l'heuristique d'ordonnancement des variables (fonction $Heuristic_{var}$) s'avère extrêmement importante pour l'efficacité des algorithmes. Son degré de liberté peut être plus ou moins limité afin de tirer profit de l'exploitation de la structure du problème ou de l'efficacité des heuristiques dynamiques. Il est possible de combiner de multiples façons ces techniques afin de définir de nouveaux algorithmes ou de capturer très facilement des algorithmes existants comme BTD [JT03], BCC [BT01a, Fre85], RC [Dar01], pseudo-tree search [FQ85], Tree-solve et Learning Tree-solve [BM96], ou AND/OR Search Tree et AND/OR Search Graph [DM07].

Par ailleurs, suivant ces combinaisons, la complexité de SBBT est différente. Pour déterminer ces bornes de complexité, nous allons observer que l'ensemble des séparateurs contient plusieurs familles maximales de séparateurs parallèles. Chaque famille induit un hypergraphe acyclique de largeur γ. Ainsi, un ordre d'affectation induit par un de ces hypergraphes à l'images des ordres

définis dans le cadre de la méthode BDH du chapitre précédant, donne à SBBT une complexité exponentielle en γ, la largeur de cet hypergraphe. La preuve de ce résultat est similaire à celle de la complexité de BDH.

5.3.1 SBBT avec un ordre d'affectation des variables et un ensemble de séparateurs induits par une décomposition arborescente

La méthode BTD [JT03] utilise un ordre d'affectation induit par une décomposition arborescente du CSP. SBBT peut facilement capturer BTD, en utilisant un ensemble orienté de séparateurs donnés par les intersections entre clusters de la décomposition arborescente. L'orientation du séparateur racine est fournie par le cluster racine qui est contenu dans la surcomposante connexe associée à ce séparateur. L'ordre d'affectation des variables de la fonction $Heuristic_{var}$ est le même que celui de BTD. On utilise la fonction $Failure_{struct}$ tandis que les autres procédures sont définies de manière usuelle (avec les algorithmes présentés). Par conséquent, SBBT mémorise au moins les mêmes (no)goods structurels que BTD, ce qui permet de garantir la même borne de complexité en temps.

Théorème 5.3.1 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en* $O(exp(w+1))$.

Preuve L'ensemble des séparateurs définit la décomposition arborescente du problème et induit un hypergraphe acyclique de largeur $\gamma = w+1$, où w est la largeur de cette décomposition. L'ordre d'affectation des variables est un ordre de classe 1 de cet hypergraphe. Alors la complexité de SBBT est en $O(exp(\gamma)) = O(exp(w+1))$.□

Il existe une seconde version de BTD légèrement différente, dans laquelle l'ordre d'affectation des variables est modifié pour permettre de choisir la prochaine variable dans toute une branche de la décomposition arborescente. Nous pouvons alors considérer qu'ainsi les clusters d'une même branche sont regroupés dans un même cluster. Cela revient donc à utiliser la première heuristique sur cette nouvelle décomposition arborescente dont la largeur est $h-1$ où h est le nombre maximum de variables dans une branche de la décomposition initiale. Cette seconde version est également capturée par SBBT de la même manière que la première, mais avec un ordre sur les variables identique à celle de cette seconde version.

Théorème 5.3.2 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en* $O(exp(h))$.

Preuve L'ensemble des séparateurs induit un hypergraphe acyclique de largeur $\gamma = w+1$, où w est la largeur de cette décomposition. L'ordre d'affectation des variables est un ordre statique de classe 4 de cet hypergraphe avec une taille maximale des hyperarêtes recouvrantes bornées par h. Alors la complexité de SBBT est en $O(exp(\gamma)) = O(exp(h))$.□

5.3.2 SBBT avec un ordre d'affectation des variables et un ensemble de séparateurs induits par un recouvrement acyclique

BDH repose sur un recouvrement par un hypergraphe acyclique du graphe de contraintes. SBBT capture BDH, en utilisant comme ensemble orienté de séparateurs l'ensemble des intersections entre hyperarêtes de l'hypergraphe de référence dont l'orientation est donnée par l'hyperarête racine. La procédure $Heuristic_{var}$ est définie pour choisir les variables dans le même ordre que BDH. On utilise la fonction $Failure_{struct}$ et une définition usuelle des autres procédures. Suivant la classe à laquelle appartient l'ordre de $Heuristic_{var}$, la complexité en temps est différente. Cependant, SBBT enregistre et exploite le même ensemble de goods et nogoods.

Théorème 5.3.3 *La complexité en temps de SBBT avec la configuration décrite ci-dessus et un ordre de la classe 3 est en $O(exp(\gamma))$, avec γ la taille maximale des hyperarêtes recouvrantes utilisées.*

Preuve L'ensemble des séparateurs définit un hypergraphe acyclique de largeur γ. L'ordre d'affectation des variables est un ordre de classe 3 de cet hypergraphe. Alors la complexité de SBBT est en $O(exp(\gamma))$.□

Théorème 5.3.4 *La complexité en temps de SBBT avec la configuration décrite ci-dessus et un ordre de la classe 4 est en $O(exp(\gamma + \Delta))$, avec $\gamma + \Delta$ la taille maximale des hyperarêtes recouvrantes utilisées.*

Preuve La preuve est similaire à la preuve précédente. L'ordre d'affectation des variables étant un ordre de classe 4 de l'hypergraphe induit par l'ensemble de séparateurs, alors la complexité de SBBT est en $O(exp(\gamma + \Delta))$.□

Théorème 5.3.5 *La complexité en temps de SBBT avec la configuration décrite ci-dessus et un ordre de la classe 5 est en $O(exp(\gamma + \Delta + 1))$, avec $\gamma + \Delta$ la taille maximale des hyperarêtes recouvrantes utilisées.*

Preuve L'ordre d'affectation des variables étant un ordre de classe 5 de l'hypergraphe induit par l'ensemble de séparateurs, alors la complexité de SBBT est en $O(exp(\gamma + \Delta))$.□

Théorème 5.3.6 *La complexité en temps de SBBT avec la configuration décrite ci-dessus et un ordre de la classe 6 est en $O(nb_{arbres}.exp(\gamma + \Delta + 1))$, avec $\gamma + \Delta$ la taille maximale des hyperarêtes recouvrantes utilisées et nb_{arbres} le nombre d'arbres utilisés lors de la résolution.*

Preuve L'ordre d'affectation des variables étant un ordre de classe 6 de l'hypergraphe induit par l'ensemble de séparateurs, alors la complexité de SBBT est en $O(nb_{arbres}.exp(\gamma + \Delta + 1))$.□

5.3.3 SBBT avec un ordre d'affectation des variables et un ensemble de séparateurs induits par une décomposition en bicomposantes

La méthode BCC [BT01a, Fre85] repose sur les composantes biconnexes du graphe de contraintes. Elle instancie les variables selon un ordre statique *BCC-compatible*. SBBT capture la méthode BCC en utilisant comme ensemble orienté de séparateurs l'ensemble des intersections entre bicomposantes de l'arbre BCC considéré dont l'orientation est fournie par la bicomposante qui contient la première variable dans l'ordre d'affectation *BCC-compatible*. L'ordre d'affectation des variables dans la procédure $Heuristic_{var}$ est choisie *BCC-compatible*. On utilise la fonction $Failure_{struct}$ et une définition usuelle des autres procédures. Par conséquent, SBBT mémorise au moins les valeurs marquées (resp. supprimées) par BCC comme des goods (resp. nogoods) structurels. Par ailleurs, SBBT effectue les mêmes sauts en arrière que BCC. D'où le théorème suivant :

Théorème 5.3.7 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en $O(exp(k))$ avec k la taille de la plus grande bicomposante.*

Preuve L'ordre d'affectation des variables est un ordre de classe 1 de l'hypergraphe induit par l'ensemble de séparateurs. Donc, la complexité de SBBT est en $O(exp(k))$. □

5.3.4 SBBT avecun ordre d'affectation des variables et un ensemble de séparateurs induits par un dtree

La méthode Recursive Conditioning [Dar01] est basée sur un dtree construit sur un ensemble de cutsets. Cet ensemble de cutsets est en fait un ensemble de séparateurs. La première version de RC (RC.v1) ne fait pas d'apprentissage. SBBT capture RC.v1 en utilisant l'ensemble des cutsets du dtree comme ensemble orienté de séparateurs dont l'orientation est donné le premier cutset instancié. L'heuristique de choix de variables de la procédure $Heuristic_{var}$ ordonne les variables de la même manière que le RC.v1. En outre, on utilise la fonction $Failure_{struct}$ alors les autres procédures ou fonctions sont définies vides.

Théorème 5.3.8 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en $O(nexp(w.logn))$, où w est la largeur du dtree et n le nombre de variables.*

Preuve L'ordre d'affectation des variables est un ordre statique de classe 4 de l'hypergraphe induit par l'ensemble des séparateurs donné par le dtree. Donc, la complexité de SBBT est en $O(n.exp(w.logn))$, où $w.logn$ est la taille maximale des hyperarêtes recouvrantes.□

Cependant, une seconde version avec mémorisation évite les résolutions multiples d'un même sous-problème, en enregistrant leur consistance (inconsistance). SBBT capture cette seconde version du RC (RC.v2) en utilisant un ensemble orienté de séparateurs et un ordre d'affectation des variables identiques à la première version. On utilise toujours la fonction $Failure_{struct}$. Mais pour cette nouvelle version, les $Pack(Good)$ et $Pack(Nogood)$ sont définis de manière usuelle. Les (no)goods enregistrés au niveau des séparateurs sont identiques à ceux de la seconde version de RC.

Théorème 5.3.9 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en $O(n.exp(w+1))$.*

Preuve L'ordre d'affectation des variables est un ordre de classe 1 de l'hypergraphe induit par l'ensemble de séparateurs. Donc, la complexité de SBBT est en $O(n.exp(w+1))$.□

RC propose de faire un compromis espace-temps qui fait que certains séparateurs ne seront pas utilisés pour enregistrer des (no)goods. Les complexités en temps et en espace seront ainsi comprises entre les deux extrêmes des deux versions précédentes. Ce compromis est également permis par SBBT par réduction de l'ensemble des séparateurs de départ.

5.3.5 SBBT avec un ordre d'affectation des variables et un ensemble de séparateurs induits par une décomposition Hinge

La notion de décomposition hinge est présentée dans [GJC94]. Dans le cadre des CSP binaires, une décomposition hinge peut être vue comme une décomposition arborescente en remplaçant chaque nœud C' de l'arbre par $var(C')$. Ainsi les intersections entre nœuds de l'arbre forment des séparateurs du graphe de contraintes. SBBT peut ainsi utiliser la structure tirée d'une décomposition hinge du graphe de contraintes de la même manière qu'une décomposition arborescente. Les intersections entre nœuds de l'arbre forment l'ensemble orienté de séparateurs comme précédemment avec la méthode BTD. On peut de même utiliser l'heuristique $Heuristic_{var}$ définie pour BTD, la fonction $Failure_{struct}$ et des $Pack(Good)$ et $Pack(Nogood)$ définis de manière usuelle. La complexité de SBBT est donc donnée par le théorème suivant.

Théorème 5.3.10 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en $O(exp(w_H))$, où w_H est le nombre maximum de variables liées par les contraites d'un hinge minimal.*

Preuve L'ordre d'affectation des variables est un ordre de classe 1 de l'hypergraphe induit par l'ensemble de séparateurs. Donc, la complexité de SBBT est en $O(exp(w_H))$, où w_H est la taille maximale des hyperarêtes.□

5.3.6 SBBT avec un ordre d'affectation des variables et un ensemble de séparateurs induits par un pseudo-tree ou un arrangement en arbre orienté

La méthode Pseudo-Tree Search (PTS [FQ85]) utilise la notion de pseudo-arbre. La méthode Tree-Solve [BM96] est très voisine de PTS et repose sur la notion d'arrangement en arbre orienté [Gav77]. La méthode AND/OR Search Tree [DM07] repose sur le calcul d'un espace de recherche AND/OR défini suivant un pseudo-arbre du graphe de contraintes. SBBT capture PTS, Tree-solve et AND/OR Search Tree en utilisant une heuristique de choix de variables induite par un pseudo-arbre (PTS et AND/OR Search Tree) ou un arrangement en arbre orienté (Tree-Solve) du graphe de contraintes du CSP. En outre, les $Pack(Good)$ et $Pack(Nogood)$ sont définis vides et associés à la fonction $Failure_{struct}$. L'ensemble orienté de séparateurs est donné par l'ensemble des ensembles de définition des sous-problèmes donnés par l'arrangement en arbre orienté (Tree-Solve) ou l'ensemble des séparateurs-parents donnés par le pseudo-arbre (PTS et AND/OR Search Tree).

Théorème 5.3.11 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en $O(exp(h))$ avec h la hauteur du pseudo-arbre ou de l'arrangement en arbre orienté.*

Preuve L'ordre d'affectation des variables est un ordre statique de classe 4 de l'hypergraphe induit par l'ensemble de séparateurs. Donc, la complexité de SBBT est en $O(nexp(h))$, où h est la taille maximale des hyperarêtes recouvrantes.□

Les méthodes Tree-Solve et AND/OR Search Tree peuvent être améliorées en enregistrant des informations qui permettent d'éviter un grand nombre de redondances et réduire ainsi la taille de l'espace de recherche. SBBT capture les méthodes Learning Tree-Solve [BM96] et AND/OR Search Graph [DM07] en utilisant le même ensemble orienté de séparateurs que précédemment et un ordre sur les variables identique (donné par une numérotation préfixe de l'arbre). Mais cette fois-ci les procédures *Good_Recording* et *Nogood_Recording* seront définies de manière usuelle. Les (no)goods enregistrés au niveau des séparateurs sont identiques à ceux de la méthode Learning Tree-Solve. Ils permettent, en outre, de retrouver l'ensemble des nœuds fusionnés dans le graphe de contexte minimal de la méthode AND/OR Search Graph.

Théorème 5.3.12 *La complexité en temps de SBBT avec la configuration décrite ci-dessus est en $O(exp(w^*))$.*

Preuve L'ordre d'affectation des variables est un ordre de classe 1 de l'hypergraphe induit par l'ensemble de séparateurs donné par le pseudo-arbre ou l'arrangement. Donc, la complexité de SBBT est en $O(exp(w^*))$, où w^* est la taille maximale des hyperarêtes.□

5.3.7 Complexités de SBBT avec un ordre d'affectation des variables et un ensemble de séparateurs quelconques

Nous voyons que SBBT peut facilement capturer plusieurs méthodes existantes. De plus, il est possible d'en définir de nouvelles comme celle présenté en exemple dans la section 5.2.2. Cette nouvelle méthode permet de calculer directement un ensemble de séparateurs et ainsi de s'assurer de bonnes propriétés (par exemple la taille ou le nombre de séparateurs ou la taille des composantes connexes). Dans la mesure où un ensemble de séparateurs définit une famille de décompositions arborescentes (comme nous l'avons vu dans le chapitre précédent consacré à BDH), il s'agit d'une structure plus générale. Il est également plus aisé de calculer une telle structure avec des propriétés adéquates plutot que de calculer des décompositions arborescentes pour lesquelles un travail supplémentaire doit être accompli pour obtenir les mêmes propriétés. Cette méthode utilise aussi des techniques de backjumping et les notions de (no)goods structurels pour

réduire la taille de l'espace de recherche en évitant certaines redondances. Par ailleurs, la fonction $Heuristic_{var}$ a un impact significatif sur le nombre de (no)goods enregistrés. Contrairement à des méthodes comme BTD, RC ou BCC qui imposent certaines contraintes sur la fonction $Heuristic_{var}$, SBBT laisse une totale liberté à l'heuristique tout en continuant à exploiter des (no)goods. Cependant, nous ne pouvons fournir aucune garantie sur le nombre de (no)goods structurels mémorisés par SBBT. Aussi, il n'est plus possible de garantir de bonnes bornes de complexité car ces bornes dépendent des redondances évitées grâce aux (no)goods. En pratique, le nombre de (no)goods mémorisés peut toujours être considérable, mais, théoriquement, nous ne pouvons garantir que la même complexité en temps que BT.

Théorème 5.3.13 *Dans le cas général, la complexité en temps de SBBT est en* $O(exp(n))$.

La complexité en espace de SBBT ne dépend que de l'ensemble de séparateurs considérés puisque toutes les informations mémoriséessont des affectations sur les séparateurs. Le nombre de (no)goods enregistrés sur un séparateur S_i est donc majoré par $d^{|S_i|}$. Aussi, l'espace-mémoire requis est borné par le nombre maximal de (no)goods qu'on peut enregistrer sur les séparateurs.

Théorème 5.3.14 *Soit s la taille du plus grand séparateur. La complexité en espace de SBBT est en* $O(n.s.exp(s))$.

Nous avons montré, dans cette section, que la complexité en temps du schéma générique SBBT dépend de l'ensemble de séparateurs, de l'heuristique d'ordonnancement des variables et des fonctions ou procédures employées. De plus, selon les choix effectués, nous avons vu que SBBT était en mesure de capturer plusieurs méthodes existantes qui exploitent la structure du problème de différentes façons.

Le tableau 5.1 présente une récapitulation des définitions des procédures et fonctions de SBBT permettant de capturer différentes méthodes. Il faut préciser que les heuristiques de choix de variables de la seconde colonne correspondent à celles utilisées dans ces méthodes et présentées tout au long de cette section.

5.4 Discussion

Le schéma générique que nous proposons dans ce papier nous permet de couvrir un large spectre d'algorithmes selon les choix effectués au niveau de l'ensemble de séparateurs et des procédures et fonctions. Ce spectre inclut des algorithmes allant des méthodes structurelles (par exemple BTD, BDH, BCC, RC, PTS, Tree-Solve, Learning Tree-Solve, AND/OR Search Tree, AND/OR Search Graph) aux méthodes purement énumératives comme BT. De plus, bien que la présentation de SBBT repose sur BT, l'emploi de techniques de filtrage ne remet pas en cause la correction, la terminaison ou la complétude de SBBT à partir du moment où le filtrage ne modifie pas le graphe de contraintes. Par exemple, nous pouvons définir une version de SBBT basée sur FC ou MAC. Cependant, un filtrage comme la consistance de chemin ne pourra être utilisé que dans le cas où SBBT n'exploite pas la structure à travers un ensemble de séparateurs. Ce type de filtrage est susceptible d'ajouter des contraintes et certains séparateurs pourraient ne plus l'être dans le nouveau graphe de contraintes.

Nous pouvons également montrer que SBBT peut facilement capturer GBJ [Dec90] et CBJ [Pro93] en définissant la fonction *Failure* de la bonne façon. Concernant les algorithmes d'apprentissage, SBBT se révèle proche de l'algorithme Nogood Recording (NR [SV94]). En fait, les nogoods structurels de SBBT sont un cas particulier des nogoods classiques exploités dans NR. Ils diffèrent principalement au niveau des justifications. Pour les nogoods structurels, les justifications reposent simplement sur les séparateurs et les sous-problèmes induits (c.-à-d. sur la structure du graphe de contraintes) au lieu des conflits rencontrés pour les nogoods classiques.

Enfin, le spectre d'algorithmes couvert par SBBT comprend les méthodes structurelles. Par exemple, il capture les méthodes PTS et AND/OR Search Tree si l'ordre d'affectation des variables est induit par un pseudo-arbre du graphe de contraintes, Tree-Solve s'il est induit par un arrangement en arbre orienté. Lorsque l'ensemble de séparateurs est calculé à partir d'une décomposition arborescente du graphe de contraintes, SBBT est équivalent à BTD et à BDH s'il s'agit d'un hypergraphe acyclique recouvrant le graphe de contraintes. Si cet ensemble est basé sur les composantes biconnexes du graphe de contraintes, il est équivalent à BCC. Dans le cas où il est basé sur un dtree, SBBT équivaut à RC. De même, notre schéma générique capture la méthode Learning Tree-solve si l'ensemble des séparateurs est calculé à partir d'un arrangement en arbre orienté, et la méthode AND/OR Search Graph dans le cas où les séparateurs sont calculés à partir d'un pseudo-arbre. En revanche, tandis que la plupart de ces méthodes structurelles exploite des ordres statiques sur les variables, SBBT ne souffre pas de cet inconvénient. Il s'ensuit que la complexité en temps et la capacité à mémoriser des nogoods dépendent directement du degré de liberté accordé à l'heurisitique de choix de variables. En effet, les nogoods ne sont mémorisés que lorsque cet enregistrement est correct, ce qui peut conduire à une diminution du nombre de nogoods mémorisés par rapport aux méthodes structurelles évoquées précédemment. Notons, que l'enregistrement de goods, lui, est totalement indépendant de l'ordre sur les variables.

Dans [JL02], les auteurs proposent une unification des algorithmes de résolution de CSP. Cette unification est en fait une caractérisation des algorithmes par leurs composantes P (propagation), L (learning : apprentissage), M (move : déplacement). Elle ne donne pas de définitions précises de ces composantes par soucis de généralité alors que SBBT en intègre plusieurs qui lui permettent de capturer un certain nombre de méthodes. Par ailleurs, dans [JL02], l'apprentissage se limite seulement aux nogoods.

[KDLD05] propose une unification des méthodes de résolution de (V)CSP basées sur une décomposition arborescente du graphe de contraintes. Les auteurs ont défini un cadre qui capture la plupart de ces méthodes en utilisant le concept d'échanges de messages entre les différents clusters de la décomposition. La notion de message recouvre les goods et nogoods structurels, l'existence dans un cluster d'une extension consistante d'une instanciation dans un cluster voisin, la compatibilité entre les solutions de différents clusters... On a ainsi une vision plus globale du fonctionnement de ces méthodes. Ils définissent à travers ce cadre une extension de la Bucket Elimination basée sur une décomposition arborescente du graphe de contraintes du (V)CSP. Par ailleurs, les aspects concernant les compromis espace/temps sont appréhendés et des versions différentes de ce cadre permettant ce compromis sont définies.

Par rapport à SBBT, ce cadre est une restriction aux décompositions arborescentes alors que SBBT utilise un ensemble quelconque de séparateurs induit ou non par une décomposition arborescente. L'échange de message de SBBT est plus encadré dans la mesure où il se limite à la mémorisation de goods et nogoods au niveau des séparateurs.

[HD03] a proposé dans le cadre SAT, une heuristique de choix de variables basée sur des critères structurels. Les auteurs utilisent un dtree qui permet de déceler les indépendances entre des parties du problème. L'ordre d'affectation des variables est induit par ce dtree comme dans le cas de la méthode RC : les cutsets (séparateurs) sont instanciés dans un ordre donné par le dtree tandis que l'ordre d'affectation des variables dans un cutset est totalement libre. Dès qu'un cutset est totalement affecté, cela conduit à plusieurs sous-problèmes indépendants qui peuvent être résolus séparément et de manière récursive. Cette heuristique peut être intégrée à toute méthode énumérative pour guider la recherche. Cependant, cela ne procure aucune garantie au niveau du nombre de redondances évitées. En effet, si la méthode découvre une inconsistance dans une partie du problème, elle peut revenir sur une autre partie indépendante qui ne contient pas les causes de l'échec.

De ce fait, l'heuristique est intégrée à l'excellent solveur SAT ZChaff (www.ee.princeton.edu/ chaff/zchaff.php) qui est une méthode énumérative utilisant une technique de retour en arrière non chronologique de type CBJ. On obtient ainsi une décomposition

de l'espace de recherche et une complexité temporelle en $O(exp(w.log(n)))$. Il est également possible de définir une extension de cette approche par un apprentissage similaire à celui de RC. Dans ce cas, la complexité en temps est en $O(exp(w))$.

Cette combinaison de ZChaff avec une heuristique de choix de variables basée sur des critères structurels, une technique de retour en arrière non chronologique et, éventuellement, une technique d'apprentissage, rentre dans le cadre définit par SBBT. Notre schéma peut être vu comme une extension de cette approche qui a montré dans [HD03] son efficacité à travers une amélioration significative des résultats de ZChaff.

[LvB04] ont proposé une amélioration de cette méthode, toujours dans SAT, par une décomposition de l'hypergraphe représentatif du problème SAT grâce à un ensemble de séparateurs. Les séparateurs utilisés dans ce cas sont des sous-ensembles d'hyperarêtes. L'ordre d'affectation des variables est induit par ces séparateurs. Au début, on calcule un séparateur qui décompose l'hypergraphe en plusieurs composantes connexes. L'affectation préalable de ce séparateur induit donc des sous-problèmes indépendants qui sont résolus récursivement et indépendamment en commençant par le calcul d'un séparateur de l'hypergraphe représentatif de chaque sous-problème.

Dans un premier temps, les séparateurs sont calculés de manière statique avant le début de la résolution. Les variables situées dans un séparateur sont affectées totalement avant de passer au suivant avec la possibilité de rajouter dynamiquement d'autres variables suggérées par une heuristique dynamique d'ordonnancement des variables. Comme dans le cas du calcul dynamique de recouvrements acycliques que nous avons défini dans le cadre de la méthode BDH dans le chapitre précédent, l'objectif est de laisser une plus grande liberté à l'heuristique de choix de variables.

Dans un second temps, les séparateurs sont calculés de manière dynamique durant la résolution. La résolution d'un problème SAT pouvant mener à l'élimination de certaines variables et occasionner ainsi une modification de sa structure, il est possible de calculer un meilleur séparateur dans l'hypergraphe courant.

Comparée à l'heuristique de [HD03], il n'est pas nécessaire de disposer de la totalité de la décomposition au départ dans cette méthode. Elle est construite de manière incrémentale et peut être stoppée à tout moment. Cette technique de décomposition a été intégrée dans ZChaff. Cela a conduit à une amélioration nette des performances de ZChaff, mais également de ZChaff associée à l'heuristique de [HD03] dans le cas du calcul statique des séparateurs. Par ailleurs, un calcul dynamique de ces derniers semble être une voie prometteuse pour une résolution encore plus efficace.

Dans le cas des CSP, les méthodes de type FC ou MAC n'induisent pas d'élimination de variables lors de la résolution. De ce fait, une mise à jour des séparateurs n'est pas justifiée.

SBBT peut être vu comme une extension de cette approche avec l'utilisation de techniques d'apprentissage et d'un ensemble de séparateurs plus général car quelconque. Cependant, nous avons présenté une version de ce cadre basé sur un calcul statique de l'ensemble de séparateurs. Il sera important dans l'avenir de pouvoir faire ce calcul de manière dynamique.

5.5 Conclusion

Nous avons décrit dans ce chapitre un schéma générique d'algorithmes énumératifs appelé SBBT. Ce schéma exploite des propriétés sémantiques et topologiques du graphe de contraintes pour produire des (no)goods. En particulier, SBBT utilise un ensemble de séparateurs du graphe de contraintes. SBBT peut être modulé via l'exploitation d'heuristiques, de méthodes de filtrages, de mémorisation de nogoods classiques ou de (no)goods structurels, et d'un ensemble de séparateurs qui traduit une décomposition du graphe de contraintes du CSP et donne une borne théorique de complexité en temps héritée des méthodes de décomposition comme les décompositions arborescentes. Ainsi, le spectre d'algorithmes décrit par SBBT s'étend

des méthodes structurelles (comme BTD, BCC, RC, PTS, Tree-Solve, Learning Tree-Solve, AND/OR Search Tree ou AND/OR Search Graph) aux méthodes purement énumératives comme BT, GBJ ou CBJ. Aussi, la complexité en temps varie entre $O(exp(w+1))$ et $O(exp(n))$ pour un graphe de contraintes de n variables et dont la tree-width est w. La complexité en espace est $O(n.s.exp(s))$ avec s la taille du plus grand séparateur.

Même si la complexité en temps de SBBT dépend entre autres de l'ensemble de séparateurs utilisé et de l'heuristique de choix de variable, SBBT ne requiert aucune propriété particulière au niveau des séparateurs. Autrement dit, n'importe quel ensemble de séparateurs peut être exploité dans SBBT. Cependant, si cet ensemble de séparateurs repose sur quelques propriétés topologiques du graphe de contraintes (comme une décomposition arborescente ou une décomposition en composantes biconnexes), nous pouvons obtenir un algorithme plus puissant avec une meilleure borne de complexité en temps. Comme aucune condition n'est imposée sur l'ensemble de séparateurs, nous pouvons facilement produire des algorithmes hybrides qui exploiteraient différentes propriétés topologiques selon la partie du graphe de contraintes considérée. Par exemple,les séparateurs pourraient être calculés à partir d'une décomposition arborescente sur une partie du problème et des bicomposantes sur une autre.

De plus, l'heuristique d'ordonnancement des variables possède également une grande influence sur la capacité à mémoriser des nogoods. Plus l'heuristique est libre, moins on enregistrera de nogoods structurels. Comme ces nogoods structurels permettent d'éviter certaines redondances, leur mémorisation et leur utilisation ont un impact significatif sur l'efficacité de la résolution. De même, il est bien connu que les heuristiques de choix de variables jouent un rôle central dans l'efficacité des méthodes de résolution. Aussi, en pratique, il pourrait être intéressant d'exploiter certains compromis entre la liberté donnée à l'heuristique et la capacité à mémoriser des nogoods structurels. Le schéma SBBT est suffisamment puissant pour permettre une mise en œuvre aisée de tels compromis.

Concernant la suite de ce travail, il faudra dans un premier temps essayer de réduire l'influence de l'heuristique de choix de variables sur la capacité à enregistrer des nogoods. Une solution pourrait passer par l'exploitation de techniques proches de celles de l'algorithme Dynamic Backtracking [Gin93]. Ensuite, nous devrons comparer SBBT avec d'autres méthodes structurelles ou purement énumératives afin de déterminer plus précisément quels sont les algorithmes existants que nous pouvons capturer avec SBBT. Concernant l'ensemble de séparateurs, SBBT est présenté avec un ensemble de séparateurs calculé statiquement. Aussi, une extension prometteuse serait de calculer cet ensemble de façon dynamique. Il faudra également mener une étude expérimentale afin de comparer les nouvelles méthodes données par SBBT aux méthodes de résolution déjà existantes, car pour celles qui sont capturées par ce cadre, des résultats sont disponibles dans la littérature. Enfin, il pourrait être utile d'étendre ce travail aux VCSP.

Algorithme 19 : Good_Recording (in :$\mathcal{A}', x, V$, in/out :V_g)

forall SP_{k,S_j} *t.q.* $SP_{k,S_j} \cap (V - V_g) = \{x\}$ **do**
  Enregistrer $\mathcal{A}'[S_j]$ comme un good relatif à SP_{k,S_j}
  $V_g \leftarrow V_g \cup CC_{k,S_j}$

Algorithme 20 : Good_Cancel (in :x, V, in/out :V_g)

forall $S_j \in Sep$ *t.q.* $S_j \cap V = \{x\}$ **do**
  $V_g \leftarrow V_g - \bigcup_k CC_{k,S_j}$

Algorithme 21 : $Failure_{GBJ}$(in :$\mathcal{A}', x$)

return $\{x\} \cup \{y \notin V | \exists c \in C,\ X_c = \{x, y\}\}$

Algorithme 22 : $Failure_{naive}$(in :$\mathcal{A}', x$)

return $\{x\} \cup \{y \notin V\}$

Algorithme 23 : $Failure_{struct}$(in :$\mathcal{A}', x, V$)

return $\{x\} \cup \{y \notin (V \cup \bigcup_{SP_{k,S_j} \text{ t.q. } x \in SP_{k,S_j} \text{ et } SP_{k,S_j} \subset (X \setminus V)} SP_{k,S_j})\}$

Méthodes	$Heuristic_{var}$	$Failure$	$Pack(Good)$	$Pack(Nogood)$
BTD	$Heuristic_{BTD,var}$	$Failure_{struct}$	usuel	usuel
BDH	$Heuristic_{BDH,var}$	$Failure_{struct}$	usuel	usuel
BCC	$Heuristic_{BCC,var}$	usuel	usuel	usuel
RC.v1	$Heuristic_{RC,var}$	$Failure_{struct}$	vide	vide
RC.v2	$Heuristic_{RC,var}$	$Failure_{struct}$	usuel	usuel
PTS	$Heuristic_{PTS,var}$	$Failure_{struct}$	vide	vide
TS	$Heuristic_{TS,var}$	$Failure_{struct}$	vide	vide
LTS	$Heuristic_{LTS,var}$	$Failure_{struct}$	usuel	usuel
AOST	$Heuristic_{AOST,var}$	$Failure_{struct}$	vide	vide
AOSG	$Heuristic_{AOSG,var}$	$Failure_{struct}$	usuel	usuel
BT	$Heuristic_{var}$	$Failure_{naive}$	vide	vide
GBJ	$Heuristic_{var}$	$Failure_{GBJ}$	vide	vide
CBJ	$Heuristic_{var}$	$Failure_{CBJ}$	vide	vide

TAB. 5.1 – Définitions des différentes fonctions et procédures de SBBT permettant de capturer des méthodes présentées : la colonne Méthodes présente les méthodes capturées (BTD, BDH, BCC, RC, PTS, TS (Tree Solve), LTS (Learning Tree Solve), AOST (And/Or Search Tree), AOSG (And/Or Search Graph), BT, GBJ, CBJ), les colonnes suivantes donnent la composition des fonctions et procédures de SBBT permettant de capturer les méthodes en question, Pack(Good) (resp. Pack(Nogood) fait référence aux algorithmes d'enregistrement et d'exploitation de goods (resp. nogoods). La mention "vide" traduit que la fonction ou procédure est définie vide et "usuelle" qu'elle est définie avec l'algorithme présenté dans ce chapitre.

Chapitre 6

Conclusion

Cette thèse avait pour objectif de rendre opérationnelles les méthodes structurelles de résolution de (V)CSP. En effet, ces techniques offrent d'excellentes garanties théoriques sur la résolution. Malheureusement, il existe une inadéquation entre leurs bornes de complexité et les résultats médiocres qu'ils obtiennent en pratique. La plupart de ces garanties théoriques sont obtenues au détriment de l'efficacité en pratique ([GLS00, CJG05]).

Nous avons présenté dans le chapitre 2 un état de l'art sur les formalismes CSP (Problème de Satisfaction de Contraintes) et VCSP (Problème de Satisfaction de Contraintes valuées). Nous avons mis un accent particulier sur les méthodes de résolution structurelles qui offrent les meilleures garanties théoriques concernant l'efficacité de la résolution. Elles sont basées sur différentes techniques de décomposition de l'hypergraphe de contraintes du problème qui calculent des recouvrements acycliques de cet hypergraphe. La première partie de cet état de l'art est ainsi consacrée au rappel de plusieurs résultats sur les graphes et hypergraphes, de même qu'un certain nombre de décompositions de ces derniers.

Le chapitre suivant présente une large étude sur les techniques de triangulation, le but étant de déterminer les meilleurs approches permettant de calculer un recouvrement acyclique de qualité pour une résolution efficace en pratique. Une première comparaison, basée sur des critères graphiques (la treewidth), est menée à l'aide de la bibliothèque TreewidthLIB consacrée aux méthodes de calcul de la treewidth ou d'une approximation de cette dernière sur un grand nombre de graphes d'origines diverses. Ensuite, une seconde étude, basée cette fois-ci sur des critères concernant l'efficacité pratique de la résolution de (V)CSP par une méthode structurelle, est présentée.

Dans le chapitre 4, nous avons défini de nouvelles stratégies d'exploitation de recouvrements acycliques de (V)CSP. La méthode BDH réalise un compromis entre la réduction de l'espace mémoire requis et celle des bornes de complexité temporelles. Nous avons proposé plusieurs classes d'ordres d'affectation de variables de plus en plus dynamiques et mené une étude expérimentale comparative sur ces différentes classes.

Le chapitre 5 présente un cadre générique d'algorithmes énumératifs basé sur un ensemble de séparateurs. Il capture plusieurs méthodes connues allant des techniques énumératives pures à celles qui offrent les meilleures bornes de complexité théorique grâce à l'exploitation de la structure du problème à travers un ensemble de séparateurs.

Notre étude sur les stratégies de calcul et d'exploitation des recouvrements acycliques a pour but d'améliorer les performances de ce type d'approche. Même si les algorithmes de calcul de décompositions arborescentes (et donc de recouvrements acycliques) par triangulation de la 2-section de l'hypergraphe de contraintes, ont été intensivement étudiés, les objectifs étaient purement graphiques. Un recouvrement acyclique du problème de largeur proche de l'optimum donne une excellente borne de complexité théorique. Mais, nous avons pu observé qu'en pratique cette largeur n'était pas le seul critère pertinent. Les méthodes structurelles de type BTD

[JT03] évitent un grand nombre de redondances par un apprentissage sur les séparateurs du recouvrement. Aussi, leur complexité spatiale est exponentielle en la taille de ces séparateurs. Il devient dès lors incontournable de maîtriser la taille de ces derniers, sous peine de voir échouer la résolution à cause d'un espace mémoire requis trop important. Nous avons ainsi défini l'algorithme TSep qui permet une grande maîtrise sur l'ensemble des séparateurs du recouvrement. Il permet de calculer un recouvrement avec un ensemble de séparateurs de taille réduite tout en évitant de faire exploser la largeur du recouvrement. Il améliore ainsi la qualité des recouvrements comparativement à la majeure partie des techniques existantes pour lesquelles un travail supplémentaire de fusion d'hyperarêtes doit être effectué pour réduire la complexité en espace. Cela peut engendrer une explosion de la largeur des recouvrements. Cependant, une étude doit être menée pour trouver une meilleure implémentation de TSep qui pour le moment consomme beaucoup de temps par rapport aux autres algorithmes de triangulation. Cet algorithme obtient, malgré tout, les meilleurs résultats, suivi d'assez près par MCS ([TY84]) qui, sur les graphes structurés, se révèle donc d'une grande efficacité.

Concernant l'exploitation des recouvrements, nous avons observé que l'ordre d'affectation des variables était un critère très important dans l'efficacité pratique de la résolution. La nécessité de disposer d'heuristiques de choix de variables efficaces a été largement démontrée au niveau des méthodes énumératives telles que FC et MAC. Par ailleurs, les heuristiques statiques obtiennent généralement des résultats très médiocres comparés à ceux des heuristiques dynamiques. Or, les méthodes structurelles sont généralement enfermées dans une grande rigidité qui leur impose des heuristiques statiques. Nous avons proposé des stratégies qui permettent d'avoir une plus grande liberté dans le choix des heuristiques. Une exploitation dynamique de la structure, associée à des heuristiques dynamiques de choix de variables de moins en moins bornées, offre à travers les méthodes BDH et BDH-val une amélioration très nette des performances.

Un compromis entre la liberté laissée aux heuristiques de choix de variables et l'exploitation de la structure du problème est primordial pour obtenir d'excellentes performances tout en gardant des garanties théoriques intéressantes. Nous avons ainsi défini des classes d'ordres de plus en plus dynamiques avec des bornes de complexité différentes. La classe 5 qui modifie de manière dynamique la structure d'une décomposition arborescente construite sur le recouvrement et laisse ainsi une grande liberté aux heuristiques de choix de variables, obtient les meilleurs résultats. La classe 6 qui remédie au choix arbitraire d'une décomposition arborescente par une construction incrémentale d'un ensemble de décompositions, devra faire l'objet d'une étude approfondie pour une implémentation efficace.

Le cadre générique d'algorithmes SBBT est basé sur un ensemble quelconque de séparateurs du CSP et plusieurs procédures et fonctions paramétrables. Il offre la possibilité de combiner l'énumération avec des techniques de retour en arrière non chronologique, de filtrages des domaines, de consistance avant, d'apprentissage et de définition d'heuristiques efficaces, mais plus particulièrement avec une exploitation de la structure à travers un ensemble de séparateurs du graphe de contraintes du CSP. Le spectre des algorithmes ainsi capturés s'étend des méthodes purement énumératives telles que BT, FC, MAC à des techniques structurelles telles que BTD, BDH, BCC, PTS, RC,etc. Il rend aisé le compromis nécessaire entre la dynamicité des ordres d'affectation des variables et l'exploitation de la structure du problème à travers l'ensemble des séparateurs. Il permet de manière simple de définir de nouveaux algorithmes qui vont réaliser ce compromis durant la recherche. Il montre également, par le nombre de méthodes qu'il capture, que l'étude menée sur le calcul et l'exploitation des recouvrements acycliques et dont l'importance est validée expérimentalement à travers BDH et BDH-val, a une portée encore plus grande. En effet, les stratégies définies sont intégrées dans SBBT ou peuvent l'être facilement. Nous pouvons, par ce biais, les répercuter dans les différentes méthodes capturées par SBBT.

Des travaux futurs devraient permettrent une amélioration de l'implémentation de TSep et son utilisation pour le calcul des recouvrements acylciques utilisés dans BDH et BDH-val ou celui des ensembles de séparateurs pour SBBT. Cela devrait conduire à une amélioration significative des performances de ces méthodes. De même, une nouvelle implémentation de BDH et BDH-val

est nécessaire pour une étude pratique des ordres de toutes les classes (classe 6 pour BDH, classe 5 et 6 pour BDH-val). Il faudra de même une définition effective de nouvelles méthodes à travers SBBT pour une étude sur les citères permettant d'accomplir un bon compromis entre les heuristiques dynamiques et l'exploitation de la structure du problème.

Bibliographie

[ACP87] S. Arnborg, D. Corneil, and A. Proskuroswki. Complexity of finding embeddings in a k-tree. *SIAM Journal of Discrete Mathematics*, 8 :277–284, 1987.

[Ami02] E. Amir. Approximation algorithms for treewidth, 2002. http ://reason.cs.uiuc.edu/eyal/paper.html.

[Ba95] H. Bodlaender and al. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of algorithms*, 18(2) :238–255, 1995.

[BBC99] A. Berry, J. Bordat, and O. Cogis. Generating all the minimal separators of a graph. In *25th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'99)*, 1999.

[BBC00] A. Berry, J. Bordat, and O. Cogis. Generating all the minimal separators of a graph. *International Journal of Foundations of Computer Science (IJFCS)*, 11 :397–404, 2000.

[BBH02] A. Berry, J. Blair, and P. Heggernes. Maximum Cardinality Search for Computing Minimal Triangulations. In *WG 2002 - 28th Workshop on Graph Theoretical Concepts in Computer Science, Cesky Krumlov, Czech Rebublic, June 2002. Springer Verlag, Lecture Notes in Computer Science*, volume 2573, pages 1–12, 2002.

[BBHP04] A. Berry, J. Blair, P. Heggernes, and B. Peyton. Maximum Cardinality Search for Computing Minimal Triangulations of Graphs. *Algorithmica*, 39-4 :287–298, 2004.

[BCS01] C. Bessière, A. Chmeiss, and L. Saïs. Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of CP'01*, pages 565–569, Paphos, Cyprus, 2001.

[Ber60] C. Berge. Les problèmes de colorations en théorie des graphes. *Publ. Inst. Statist. Univ. Paris 9*, pages 123–160, 1960.

[Ber70] C. Berge. *Graphes et Hypergraphes.* Dunod-France, 1970.

[Ber98] A. Berry. *Désarticulation d'un graphe.* PhD thesis, Université Montpellier II, 1998.

[Ber99] A. Berry. A Wide-Range Efficient Algorithm for Minimal Triangulation. In *Proceedings of SODA'99 SIAM Conference*, january 1999.

[Bes94] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65 :179–190, 1994.

[BFM+96] S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and valued CSPs : Basic properties and comparison. *LNCS*, 1106, 1996.

[BFMY83] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30 :479–513, 1983.

[BG81] P.A. Bernstein and N. Goodman. The power of natural semijoins. *SIAM J. Comput.*, 10-4 :751–771, 1981.

[BG96] A. Becker and D. Geiger. A sufficiently fast algorithm for finding close to optimal junction trees. In *UAI'96*, pages 81–89, 1996.

[BHS03] A. Berry, P. Heggernes, and G. Simonet. The minimum degree heuristic and the minimal triangulation process. In *Proceedings of the* 29^{th} *Workshop on Graph Theoretic Concepts in Computer Science*, pages 58–70, 2003.

[BHT01] J.R.S. Blair, P. Heggernes, and J.A. Telle. A practical algorithm for making filled graphs minimal. *Theor. Comput.*, 250 :125–141, 2001.

[BHV03] A. Berry, P. Heggernes, and Y. Villander. A vertex- incremental approach for dynamically maintaining chordal graphs. In *LNCS Proceedings ISAAC 2003*, pages 47–57, 2003.

[BKMT04] V. Bouchitté, D. Kratsch, H. Muller, and I. Todinca. On treewidth approximations. *Discrete Appl. Math.*, 136(2-3) :183–196, 2004.

[BKvdEvdG01] H. Bodlaender, A. Koster, F. van den Eijkhof, and L. van der Gaag. Prepocessing for triangulation of probabilistic networks. *Uncertainty in Artificial Intelligence*, pages 32–39, 2001.

[BM96] R. J. Bayardo and D. P. Miranker. A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraints Satisfaction Problem. In *Proc. of AAAI*, pages 298–304, 1996.

[BMFL02] C. Bessière, P. Meseguer, E. C. Freuder, and J. Larrosa. On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141 :205–224, 2002.

[BMR95] S. Bistarelli, U. Montanari, and F. Rossi. Constraint solving over semirings. In *Proceedings of the fourteenth International Joint Conference on Artificial Intelligence*, pages 624–630, Montréal, Canada, 1995.

[BR96] C. Bessière and J.-C. Régin. MAC and Combined Heuristics : Two Reasons to Forsake FC (and CBJ ?) on Hard Problems. In *Proceedings of CP'96*, pages 61–75, 1996.

[BR01] C. Bessière and J.-C. Régin. Refining the Basic Constraint Propagation Algorithm. In *Actes des Journées Francophones de Programmation en Logique et par Contraintes*, pages 13–25, 2001.

[BRYZ05] C. Bessière, J.-C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2) :165–185, 2005.

[BT01a] J.-F. Baget and Y. Tognetti. Backtracking Throught Biconnected Components of a Constraint Graph. In *Proc. of IJCAI*, pages 291–296, 2001.

[BT01b] V. Bouchitté and I. Todinca. Treewidth and minimum fill-in : grouping the minimal separators. *SIAM J. Comput.*, 31(1) :212–232, 2001.

[BT02] V. Bouchitté and I. Todinca. Listing all potential maximal cliques of a graph. *Theoret. Comput. Sci.*, 276(1-2) :17–32, 2002.

[CdGL+99] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio Link Frequency Assignment. *Constraints*, 4 :79–89, 1999.

[CdGS07] M. Cooper, S. de Givry, and T. Schiex. Optimal soft arc consistency. In *Proceedings of IJCAI-07*, 2007.

[Chm96] A. Chmeiss. Sur la consistance de chemin et ses formes partielles. In *Actes du Congrès AFCET-RFIA 96*, pages 212–219, Rennes, France, 1996.

[CJ98] A. Chmeiss and P. Jégou. Décomposition : Vers une érosion du pic de difficulté ? In $4^{ème}$ *Journées Nationales Résolution Pratique des Problèmes NP-Complets*, pages 21–29, Nantes, France, 1998. JNPC 98.

[CJG05] D. Cohen, P. Jeavons, and M. Gyssens. A Unified Theory of Structural Tractability for Constraint Satisfaction and Spread Cut Decomposition. In *IJCAI'05*, pages 72–77, 2005.

[Coo89] M. Cooper. An Optimal k-Consistency Algorithm. *Artificial Intelligence*, 41, 1989.

[Coo03] M. Cooper. Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134-3, 2003.

[Coo05] M. Cooper. High-order consistency in Valued Constraint Satisfaction. *Constraints*, 10, 2005.

[CS04] M. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154, 2004.

[Dah97] E. Dahlhaus. Minimal elimination ordering inside a given chordal graph. In *Workshop on Graph Theoretical Concepts in Computer Science*, pages 132–143, 1997.

[Dar01] A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126 :5–41, 2001.

[Dec90] R. Dechter. Enhancement Schemes for Constraint Processing : Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41 :273–312, 1990.

[Dec99] R. Dechter. Bucket Elimination : A Unifying Framework for Reasoning. *Artificial Intelligence*, 113(1-2) :41–85, 1999.

[Dec03] R. Dechter. *Constraint processing*. Morgan Kaufmann Publishers, 2003.

[DF01] R. Dechter and Y. El Fattah. Topological Parameters for Time-Space Tradeoff. *Artificial Intelligence*, 125 :93–118, 2001.

[DF02] R. Dechter and D. Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136 :147–188, 2002.

[DFP93] D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proceedings of the IEEE International Conference on Fuzzy Systems*, pages 1131–1136, 1993.

[dGHZL05] S. de Givry, F. Heras, M. Zytnicki, and J. Larrosa. Existential arc consistency : Getting closer to full arc consistency in weighted CSP. In *Proceedings of IJCAI'05*, 2005.

[dGSV06] S. de Givry, T. Schiex, and G. Verfaillie. Décomposition arborescente et cohérence locale souple dans les CSP pondérés. In *Proceedings of JFPC'06*, 2006.

[DH01] A. Darwiche and M. Hopkins. Using recursive decomposition to construct elimination orders, jointrees and dtrees. In *Trends in Artificial Intelligence, Lecture notes in AI, 2143*, pages 180–191. Springer-Verlag, 2001.

[Dir61] G.A. Dirac. On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburg 25*, MR24 -57 :71–76, 1961.

[DM89] R. Dechter and I. Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. In *Proceedings of the tenth International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 271–277, Detroit, Michigan, August 1989.

[DM07] R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 2007.

[DP87a] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34 :1–38, 1987.

[DP87b] R. Dechter and J. Pearl. The Cycle-cutset method for Improving Search Performance in AI Applications. In *Proceedings of the third IEEE on Artificial Intelligence Applications*, pages 224–230, 1987.

[DP89] R. Dechter and J. Pearl. Tree-Clustering for Constraint Networks. *Artificial Intelligence*, 38 :353–366, 1989.

[Eve79] S. Evens. Graph algorithms. *Computer Science Press, Rockville, MD*, 1979.

[FD94] D. Frost and R. Dechter. Dead-end driven learning. In *Proceedings of AAAI'94*, pages 294–300, 1994.

[FD95] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 572–578, Montréal, Canada, 1995.

[FG65] D.R. Fulkerson and O. Gross. Incidence matrices and interval graphs. *Pacific J. Math.*, 15 :835–855, 1965.

[FKT04] F. Fomin, D. Kratsch, and I. Todinca. Exact (exponential) algorithms for treewidth and minimum fill-in. In *Proceedings of ICALP*, pages 568–580, 2004.

[FL93] H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems : a probabilistic approach. In *Proceedings of ECSQARU'93*, volume 747 of *LNCS*, pages 97–104. 1993.

[FLMCS95] H. Fargier, J. Lang, R. Martin-Clouaire, and T. Schiex. A constraint satisfaction framework for decision under uncertainty. In *Proceedings of the 11^{th} International Conference on Uncertainty in Artificial Intelligence*, 1995.

[FLS93] H. Fargier, J. Lang, and T. Schiex. Selecting preferred solutions in Fuzzy Constraint Satisfaction Problems. In *Proceedings of the first European Congress on Fuzzy and Intelligent Technologies (EUFIT'93)*, 1993.

[FO74] T. Fujisawa and H. Orino. An efficient algorithm of finding a minimal triangulation of a graph. In *IEEE International Symposium on Circuits and Systems*, pages 172–175, 1974.

[FQ85] E. Freuder and M. Quinn. Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In *Proceedings of the ninth International Joint Conference on Artificial Intelligence*, pages 1076–1078, 1985.

[Fre78] E. Freuder. Synthesizing constraint expressions. *CACM*, 21(11) :958–966, 1978.

[Fre82] E. Freuder. A Sufficient Condition for Backtrack-Free Search. *JACM*, 29 (1) :24–32, 1982.

[Fre85] E. Freuder. A Sufficient Condition for Backtrack-Bounded Search. *JACM*, 32 :755–761, 1985.

[FW92] E. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58 :21–70, 1992.

[Gas79] J. Gaschnig. Performance Measurement and Analysis of Certain Search Algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, 1979.

[Gav72] F. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1 (2) :180–187, 1972.

[Gav77] F. Gavril. Some NP-complete Problems on Graphs. In *Proceedings of the 11th Conf. on Information Sciences and Systems*, pages 91–95, 1977.

[GD04] V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *Proceedings of UAI*, pages 201–208, 2004.

[Gin93] M. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46, 1993.

[GJ79] M.R. Garey and D.S. Johnson. Computer and Intractability. In *Freeman*, 1979.

[GJC94] M. Gyssens, P. Jeavons, and D. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66 :57–89, 1994.

[GLS00] G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124 :343–282, 2000.

[Gol80] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[GP84] M. Gyssens and J. Paredaens. A decomposition methodology for cyclic databases. *Advances in Database Theory*, 2 :85–122, 1984.

[HD03] J. Huang and A. Darwiche. A Structure-Based Variable Ordering Heuristic for SAT. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1167–1172, 2003.

[HE80] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[Heg06] P. Heggernes. Minimal triangulations of graphs : A survey. *Discrete Mathematics*, 306-3 :297–317, 2006.

[HL88] C. Han and C. Lee. Comments on Mohr and Henderson's path consistency algorithm. *Artificial Intelligence*, 36 :125–130, 1988.

[HS58] A. Hajnal and J. Suranyi. Über die Auflösung von Graphen in vollständige Teilgraphen. *Ann. Univ. Sci. Budapest Eötvös. Sect. Math. 1*, MR21 -1944 :113–121, 1958.

[JL02] N. Jussien and O. Lhomme. Vers une unification des algorithmes de résolution de CSP. In *Proc. of JNPC*, pages 155–168, 2002.

[JNT05a] P. Jégou, S. N. Ndiaye, and C. Terrioux. Computing and exploiting tree-decompositions for solving constraint networks. In *Proceedings of CP*, pages 777–781, 2005.

[JNT05b] P. Jégou, S. N. Ndiaye, and C. Terrioux. Sur la génération et l'exploitation de décompositions pour la résolution de réseaux de contraintes. In *Actes des JFPC'2005*, pages 149–158, 2005.

[JNT06] P. Jégou, S. N. Ndiaye, and C. Terrioux. Heuristiques pour la recherche énumérative bornée : Vers une libération de l'ordre. Technical Report LSIS.RR.2006.004, Laboratoire des Sciences de l'Information et des Systèmes (LSIS), 2006.

[JNT07] P. Jégou, S.N. Ndiaye, and C. Terrioux. 'Dynamic Heuristics for Backtrack Search on Tree-Decomposition of CSPs. In *Proc. of IJCAI*, pages 112–117, 2007.

[JT03] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146 :43–75, 2003.

[JT04a] P. Jégou and C. Terrioux. A Time-space Trade-off for Constraint Networks Decomposition. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 234–239, 2004.

[JT04b] P. Jégou and C. Terrioux. Decomposition and good recording for solving Max-CSPs. In *Proc. of ECAI*, pages 196–200, 2004.

[JV93] P. Jégou and M. Vilarem. On some partial line graphs of a hypergraph and the associated matroid. *Discrete Mathematics*, 111 :333–344, 1993.

[KDLD05] K. Kask, R. Dechter, J. Larrosa, and A. Dechter. Unifying tree decompositions for reasoning in graphical models. *Artificial Intelligence*, 166 :165–193, 2005.

[Kja90a] U. Kjaerulff. Triangulation of Graphs - Algorithms Giving Small Total State Space. Technical report, Judex R.R. Aalborg., Denmark, 1990.

[Kja90b] U. Kjaerulff. Triangulation of graphs - algorithms giving small total state space. Technical report, Judex R.R. Aalborg, Denmark, 1990.

[Klo94] T. Kloks. Treewith : computations and approximations. *LNCS*, 842, 1994.

[Kos99] A. Koster. *Frequency Assignment - Models and Algorithms*. PhD thesis, University of Maastricht, Novembre 1999.

[Lar00] J. Larrosa. Boosting Search with Variable Elimination. In *Proceedings of the 6^{th} CP*, pages 291–305, 2000.

[Lar02] J. Larrosa. On arc and node consistency. In *Proceedings of AAAI'2002*, pages 48–53, Edmonton (Canada), 2002.

[LB62] C.G. Lekkerkerker and J.Ch. Boland. Representation of a finite graph by a set of intervals on the real line. *Fund. Math. 51*, MR25 -2596 :45–64, 1962.

[LD03] J. Larrosa and R. Dechter. Boosting Search with Variable Elimination in Constraint Optimization and Constraint Satisfaction Problems. *Constraints*, 8(3) :303–326, 2003.

[LM96] J. Larrosa and P. Meseguer. Exploiting the use of DAC in Max-CSP. In *Proceedings of the 2^{nd} CP*, pages 308–322, 1996.

[LMS99] J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107(1) :149–163, 1999.

[LMS02] J. Larrosa, P. Meseguer, and M. Sánchez. Pseudo-Tree Search with Soft Constraints. In *Proceedings of the 15^{th} European Conference on Artificial Intelligence*, pages 131–135, Lyon (France), 2002.

[LS03] J. Larrosa and T. Schiex. In the quest of the best form of local consistency for Weighted CSP. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 239–244, 2003.

[Lue74] G. Lueker. Structured breadth first search and chordal graphs. Technical Report 158, Princeton Univ, 1974.

[LvB04] W. Li and P. van Beek. Guiding Real-World SAT Solving with Dynamic Hypergraph Separator Decomposition. In *Proceedings of ICTAI*, pages 542–548, 2004.

[Mac77] A. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8 :99–118, 1977.

[MD06] R. Marinescu and R. Dechter. Dynamic Orderings for AND/OR Branch-and-Bound Search in Graphical Models. In *Proc. of ECAI*, pages 138–142, 2006.

[MH86] R. Mohr and T. C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28, 1986.

[Mon74] U. Montanari. Networks of Constraints : Fundamental Properties and Applications to Picture Processing. *Artificial Intelligence*, 7 :95–132, 1974.

[MS00] P. Meseguer and M. Sánchez. Tree-based Russian Doll Search. In *Proceedings of Workshop on soft constraint*. CP'2000, 2000.

[MS01] P. Meseguer and M. Sánchez. Specializing Russian Doll Search. In *Proceedings of the 7th CP*, volume LNCS 2239, pages 464–478, 2001.

[MSV02] P. Meseguer, M. Sánchez, and G. Verfaillie. Opportunistic Specialization in Russian Doll Search. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, volume LNCS 2470, pages 264–279, 2002.

[Pey01] B.W. Peyton. Minimal orderings revisited. *SIAM J. Matrix Anal. Appl.*, 23(1) :271–294, 2001.

[Pro93] P. Prosser. Hybrid Algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9 :268–299, 1993.

[PS97] A. Parra and P. Scheffler. Characterizations and algorithmic applications of chordal graph embeddings. *Discrete Applied Math.*, 79(1-3) :171–188, 1997.

[Ree92] B. Reed. Finding approximate separators and computing tree width quickly. In *24th STOC*, pages 221–228, 1992.

[RHZ76] A. Rosenfeld, R. Hummel, and S. Zucker. Scene Labelling by Relaxation Operations. *IEEE Transaction on System, Man, and Cybernetics*, 6(6) :420–433, 1976.

[RS86] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, 7 :309–322, 1986.

[RS95] N. Robertson and P.D. Seymour. Graph minors XIII : The disjoint paths problem. *Comb. Theory, Series B*, 63 :65–110, 1995.

[RTL76] D. Rose, R. Tarjan, and G. Lueker. Algorithmic Aspects of Vertex Elimination on Graphs. *SIAM Journal on computing*, 5 :266–283, 1976.

[Rut94] Z. Ruttkay. Fuzzy constraint satisfaction. In *Proceedings of the 3^{rd} International Conference on Fuzzy Systems*, 1994.

[Röh99] H. Röhrig. *Tree-decomposition : A feasible study.* PhD thesis, University of Saarbrücken, Germany, 1999.

[Sch92] Thomas Schiex. Possibilistic Constraint Satisfaction Problems or "How to handle soft constraints ?". In *Proceedings of the Eight International Conference on Uncertainty in Artificial Intelligence*, pages 268–275, Stanford, CA, 1992.

[SF94] D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proc. of ECAI*, pages 125–129, 1994.

[SFV95] T. Schiex, H. Fargier, and G. Verfaillie. Valued Constraint Satisfaction Problems : hard and easy problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 631–637, 1995.

[SFV97] T. Schiex, H. Fargier, and G. Verfaillie. Problèmes de satisfaction de contraintes valués. *Revue d'Intelligence Artificielle*, 11(3), 1997.

[SG97] K. Shoikhet and D. Geiger. A practical algorithm for finding optimal triangulation. In *Proceedings of AAAI*, pages 185–190, 1997.

[Sin95] M. Singh. Path Consistency Revisited. In *Proceedings of the 5th International Conference on Tools for Artificial Intelligence*, pages 318–325, 1995.

[Sin96] M. Singh. Path Consistency Revisited. *IJAIT*, 5 :127–141, 1996.

[Smi94] B. Smith. The Phase Transition and the Mushy Region in Constraint Satisfaction Problems. In *Proceedings of ECAI*, pages 100–104, 1994.

[SV93] T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. In *Proceedings of the 5^{th} IEEE International Conference on Tools with Artificial Intelligence*, Nov. 1993.

[SV94] T. Schiex and G. Verfaillie. Nogood Recording for static and dynamic constraint satisfaction problems. Notes de recherche, 1994. Extension de [SV93].

[TJ03] C. Terrioux and P. Jégou. Bounded backtracking for the valued constraint satisfaction problems. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP-2003)*, pages 709–723, 2003.

[TY84] R. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13 (3) :566–579, 1984.

[VLS96] G. Verfaillie, M. Lemaître, and T. Schiex. Russian Doll Search for Solving Constraint Optimization Problems. In *Proceedings of the 13th AAAI*, pages 181–187, 1996.

[Wal72] J.R. Walter. *Representations of rigid cycle graphs.* PhD thesis, Wayne State University, 1972.

[Wal75] D. Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, pages 19–91. P. H. Winston, McGraw–Hill, New York, 1975.

[Wal94] R. Wallace. Directed arc consistency preprocessing. In *Proceedings of the ECAI-94 Workshop on Constraint Processing*, volume LNCS 923, pages 121–137, 1994.

[Wan05] J. Wang. On Axioms Constituting the Foundation of Hypergraph Theory. *Acta Mathematicae Applicatae Sinica, English Series*, 21(3) :495–498, 2005.

[ZY01] Y. Zhang and R. Yap. Making AC-3 an Optimal Algorithm. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 316–321, 2001.

Chapitre 7

Notations

- $X = \{x_1, x_2, \ldots, x_n\}$: ensemble de sommets d'un (hyper)graphe ou de variables d'un (V)CSP, pages 9, 22, 38
- n est le nombre d'éléments dans X 9, 22, page 38
- $C = \{c_1, c_2, \ldots, c_m\}$: ensemble des arêtes d'un (hyper)graphe ou de contraines d'un (V)CSP pages 9, 22, 38
- m est le nombre d'éléments dans C 9, 22, page 38
- $G = (X, C)$ est un graphe, avec X l'ensemble et C l'ensemble des arêtes, page 9
- $\omega(G)$ est la taille de la plus grande clique maximale dans un graphe G, page 57
- $\chi(G)$ est le plus petit nombre de couleurs suffisantes pour colorier un graphe G, page 57
- $\sigma = [x_1, \ldots, x_n]$ est un ordre sur les sommets du graphe G , page57
- $\sigma = [c_1, \ldots, c_m]$ est un ordre sur les sommets du graphe G, page 15
- $N(x_i) = \{x_j \in X,\ x_j \textit{ et } x_i \textit{ sont voisins dans } G\}$ est le voisinage de x_i dans le graphe G, page 10
- $N[x_i] = N(x_i) \cup \{x_i\}$ est le voisinage fermé de x_i dans le graphe G, page 10
- $N^+_\sigma(x_i) = \{x_j \in N(x_i)/\sigma(j) > \sigma(i)\}$, page 57
- $S(k)$ est l'ensemble des séparateurs minimaux dont la taille est au plus k, page 59
- Nb_{sepmin} est le nombre de séparateurs minimaux 59, page 66
- Soit V un ensemble de sommets de G,
  - $N(V) = \bigcup_{x \in V} N(x) - V$ est le voisinage de V, page 10
  - $N[V] = \bigcup_{x \in V} N[x]$ est le voisinage fermé de V, page 10
- $G[V]$ est le graphe G restreint aux sommets de $V \subset X$, page 10
- $H = (X, C)$ est un hypergraphe, avec X l'ensemble et C l'ensemble des hyperarêtes, page 12
- H_A est un hypergrpahe acylcique, pages 91, 83, 90
- Un CAH est recouvrement par un hypergraphe acyclique de l'hypergraphe H, page 91
- $\gamma = max_{E_i \in E}|E_i|$ est la largeur d'un CAH (X, E), page 91
- γ^* est la CAH-largeur d'un hypergraphe H, la largeur minimale parmi tous les CAHs de H, page 91
- $\mathcal{CAH}(H)$ est l'ensemble des CAHs d'un hypergraphe H, page 91
- $\mathcal{CAH}_{H_A} = \{(X, E') \in \mathcal{CAH}(H) : \forall E_i \in E, \exists E'_j \in E' : E_i \subset E'_j\}$ est l'ensemble des recouvrements d'un CAH $H_A = (X, E)$ d'un hypergraphe H, page 91
- $\mathcal{CAH}_{H_A}[C^+]$ est la restriction de $\mathcal{CAH}_{H_A}$ aux éléments de cet ensemble qui respecte la condition de "connexité", page 92
- $\mathcal{CAH}_{H_A}[P^+]$ est la restriction de $\mathcal{CAH}_{H_A}$ aux éléments de cet ensemble qui respecte la condition de "chemin", page 92
- $\mathcal{CAH}_{H_A}[F^+]$ est la restriction de $\mathcal{CAH}_{H_A}$ aux éléments de cet ensemble qui respecte la condition de "famille", page 93

- $\mathcal{CAH}_{H_A}[U^+]$ est la restriction de $\mathcal{CAH}_{H_A}$ aux éléments de cet ensemble qui respecte la condition d"unicité", page 93
- $\mathcal{CAH}_{H_A}[B^+]$ est la restriction de $\mathcal{CAH}_{H_A}$ aux éléments de cet ensemble qui respecte la condition de "fraternité", page 93
- $\mathcal{CAH}_{H_A}[X]$ est la restriction de $\mathcal{CAH}_{H_A}$ aux éléments de cet ensemble qui respecte la condition de "$X = C, P, F, U$ ou B" et celle d"égalité", page 93
- $\mathcal{CAH}_{H_A}[S]$ est la restriction de $\mathcal{CAH}_{H_A}$ aux éléments de cet ensemble qui respecte la condition de "maintien des séparateurs", page 93
- CC_i est une composante connexe d'un (hyper)graphe, pages 11, 13
- $(x_{u_1}, x_{u_2}, \ldots, x_{u_p})$ est une séquence de sommets de X , page 10
- Soit G un graphe,
  - $(E, \mathcal{T})$ est une décomposition arborescente d'un graphe G, page 15
  - $\mathcal{T} = (I, F)$ est un arbre, page 15
  - $E = \{E_i : i \in I\}$ une famille de sous-ensembles de X appelés clusters, page 15
- Soient $(E, \mathcal{T})$ est une décomposition arborescente, E_1 la racine de $\mathcal{T}$, $Desc(E_j)$ est l'ensemble des sommets contenues dans le sous-arbre enraciné en E_j, page 31
- s est la taille maximale des intersections entre les clusters d'une décomposition arborescente $(E, \mathcal{T})$, page 32
- s^- est la taille minimale des intersections non vides entre les clusters d'une décomposition arborescente $(E, \mathcal{T})$, page 94
- w^* est la largeur induite d'un pseudo-arbre, page 94
- h est la hauteur d'un arbre, page 34
- (T, χ, λ) est un hyperarbre d'un hypergraphe H, page 19
- HD est une décomposition en hyperarbre d'un hypergraphe H, page 19
- (λ, χ) est un bloc encadré d'un hypergraphe H, page 19
- GB est un ensemble de blocs encadrés d'un hypergraphe H, page 19
- $\mathcal{P}$ est un (V)CSP, pages 22, 38
- D_{x_i} est un domaine associé à une variable x_i d'un (V)CSP, pages 22, 38
- D est l'ensemble des domaines associés aux variables d'un (V)CSP, pages 22, 38
- d est la taille maximale des domaines de D, page 25
- C_i est une contrainte d'un (V)CSP, pages 22, 38
- C est l'ensemble des contraintes d'un (V)CSP, pages 22, 38
- X_c, où $c \in C$, est l'ensemble des variables sur lesquelles porte la contrainte c d'un VCSP, page 38
- R_i est la relation associée à la contrainte C_i d'un CSP, pages 22, 38
- R est l'ensemble des relations d'un CSP, page 22
- $\mathcal{A}$ est une affectation d'un (V)CSP, page 24
- $X_{\mathcal{A}}$ est l'ensemble des variables d'un (V)CSP intanciée dans l'affectation $\mathcal{A}$, page 24
- $\mathcal{A}[Y]$, où $Y \subset X_{\mathcal{A}}$, est la restriction de l'affectation $\mathcal{A}$ aux variables de Y, page 24
- $Sol_{\mathcal{P}}$ est l'ensemble des solutions du CSP $\mathcal{P}$, page 24
- w est la treewidth d'un graphe, page 16
- $S = (E, \preceq, \oplus)$ est une structure de valuation d'un VCSP, page 38
- $C_\emptyset$ est une contrainte d'un VCSP qui ne porte sur aucune variable et dont la valeur est un minorant de la valuation optimale, page 38
- $\mathcal{V}_{\mathcal{P}}(\mathcal{A})$ est la valuation de l'affectation totale $\mathcal{A}$ dans le VCSP $\mathcal{P}$, page 39
- $v_{\mathcal{P}}(\mathcal{A})$ est la valuation de l'affectation paritelle $\mathcal{A}$ dans le VCSP $\mathcal{P}$, page 39
- ub est un majorant de la valuation optimale d'un VCSP, page 41
- lb est un minorant de la valuation optimale d'un VCSP, page 41
- $L(x) = \{c \in C | x \in X_c \subset (X_{\mathcal{A}} \cup x)\}$, est l'ensemble des contraintes contenant x dont toutes les autres variables sont déjà affectées dans $\mathcal{A}$, page 42
- $ic(x, v) = \bigoplus_{c \in L(x)} c(\mathcal{A} \cup \{x \leftarrow v\}[X_c])$ est le coût induit par l'extension de l'affectation $\mathcal{A}$ par l'instanciation de x à la valeur v, page 43

- $dac(x, v)$) est un minorant du coût induit sur les variables non encore instanciées par une affectation de la valeur v à la variable x, page 44
- $\mathcal{P}_{\mathcal{A},E_i/E_j} = (X_{\mathcal{P}_{\mathcal{A},E_i/E_j}}, D_{\mathcal{P}_{\mathcal{A},E_i/E_j}}, C_{\mathcal{P}_{\mathcal{A},E_i/E_j}}, S)$ est le CSP induit par $\mathcal{A}$ sur $Desc(E_j)$, page 48
- $\mathcal{C}(H)$ est l'ensemble des intergraphes d'un hypergraphe H, page 84
- $\mathcal{C}_m(H)$ est l'ensemble des intergraphes minimaux d'un hypergraphe H, page 85
- Soit H_A un hypergraphe acyclique et T_c un arbre d'une décomposition arborescente construite sur les hyperarêtes de H_A
  - $N^*(E_i) = \{E_j \in E,\ E_j \notin T_c,\ E_i$ et E_j sont α-voisines $\}$ est l'ensemble des α-voisins de E_i, qui contient ses voisins dans T_c, page 95
  - $M^*(E_i) = \{E_j \in N^*(E_i), \exists E_k \in N^*(E_i)$ et $E_j \in N^*(E_k)\}$ est un ensemble dont les éléments du complémentaire dans $N^*(E_i)$ sont des voisins obligatoires de E_i dans T_c, page 95
  - $K^*(E_i) = \{E_j \in N^*(E_i), \exists E_k \in N^*(E_i)$ et $E_j \in N^*(E_k),\ E_i \cap E_j \not\supseteq E_i \cap E_k\}$ est un ensemble de voisins obligatoires de E_i dans T_c, page 95
  - $L^*(E_i) = \{\{E_j, E_k\}, E_j \in N^*(E_i), E_k \in N^*(E_i),\ E_j \in N^*(E_k)$ et $E_i \cap E_j \subsetneq E_j \cap E_k\}$ est un ensemble de paires d'hyperarêtes qui ne peuvent pas être simultanément voisines de E_i dans T_c, page 95
  - $I^*(E_i) = \{E_j \in N^*(E_i), \exists (E_j, E_k) \in L^*(E_i),\ (E_i, E_k)$ est une arête de $T_c\}$ est un ensemble d'hyperarêtes qui ne peuvent pas être voisines de E_i, page 95
  - $O^*(E_i) = (N^*(E_i) \backslash M^*(E_i)) \cup P^*(E_i)$ est l'ensemble d'hyperarêtes qui seront obligatoirement voisines de E_i, page 95
  - $P^*(E_i) = (N^*(E_i) \backslash (O^*(E_i) \cup I^*(E_i)$ est l'ensemble des hyperarêtes qui pourraient être voisines de E_i dans T_c, page 95
- Soit une décomposition arborescente ou un recouvrement acyclique orienté par le choix d'un cluster ou hyperarête racine E_1
  - $Père(E_i)$ est le noeud père de E_i, pages 48, 98
  - $Fils(E_i)$ est l'ensemble des noeuds fils de E_i, pages 31, 98

Chapitre 8

Annexes

Les résultats présentés dans les tableaux de ce chapitre sont discutés dans le chapitre 3 consacré au calcul de décompositions arborescentes par triangulation de graphes.

Graphes	n : m	UB	Algorithmes-UB	LB	Algorithmes-LB
314	11 : 20	5	All + Exact	5	Exact
262	15 : 24	7	All + Exact	7	Exact
254	15 : 31	4	All + Exact	4	Exact
229	16 : 36	5	All + Exact	5	Exact
115	16 : 43	6	All-MCS	6	Exact
236	17 : 47	7	All-MCS,MCS+ + Exact	7	Exact
237	17 : 47	7	MCS-M,MinDeg,Mindeg+,Sfill+, + Exact	7	Exact
238	18 : 47	6	Exact	6	Exact
296	19 : 37	4	All-LEX-M,MSVS + Exact	4	Exact
246	23 : 66	8	All-MCS,MCS+ + Exact	8	Exact
315	23 : 71	10	MCS+,Sfill+, + Exact	10	Exact
297	26 : 52	5	LEX-M,Sfill+, + Exact	5	Exact
142	26 : 78	7	All + Exact	7	Exact
143	28 : 135	11	All-MCS + Exact	11	Exact
289	29 : 113	13	All-MCS + Exact	13	Exact
78	30 : 144	10	All + Exact	10	Exact
215	32 : 123	9	SFill,Sfill+, + Exact	9	Exact
66	34 : 156	11	All + Exact	11	Exact
253	35 : 80	4	All + Exact	4	Exact
1	37 : 65	4	All	4	Exact
132	38 : 238	14	Exact	14	Exact
138	39 : 67	3	All-SFill	3	Exact
402	41 : 195	11	LEX-M,MCS-M,MinFill,Minfill+,SFill,Sfill+	10	LBN,LBP
139	42 : 72	3	All-SFill	3	Exact
127	45 : 320	16	MinFill,Minfill+,Sfill+	15	LBN,LBP
636	45 : 320	16	MinFill,Minfill+,SFill,Sfill+	16	TWDP
320	47 : 170	12	MinDeg,Mindeg+,MinFill,Minfill+,MSVS	11	TWDP
316	47 : 236	19	MinDeg,Mindeg+, + Exact	19	Exact
128	47 : 271	15	All-LEX-M,MCS-M,MCS,MCS+	15	TWDP
135	47 : 275	15	LEX-M,MinFill,Minfill+,SFill,Sfill+	15	TWDP
2	48 : 126	7	All-Minfill+	6	LBN,LBP,MMD
116	48 : 264	15	LEX-M,MinFill,Minfill+,Sfill+	15	TWDP
397	48 : 412	22	Exact	19	Exact
261	48 : 84	7	All-Sfill	7	TLB
504	49 : 412	22	Dynprog	18	LBP
342	51 : 140	9	MCS,MCS+	7	LBP
447	51 : 362	15	LEX-M,MCS-M,MCS+	15	TWDP
457	52 : 405	19	LEX-M + Exact	19	Exact
427	52 : 426	20	LEX-M,MCS-M	18	LBP
248	55 : 156	8	MinDeg,Mindeg+,MinFill,Minfill+,Sfill+	8	TWDP
509	57 : 543	28	All-MCS,MCS+	21	LBP
133	58 : 311	15	SFill,Sfill+	15	TWDP
448	59 : 397	15	MCS-M,MCS+	14	LBP

Tab. 8.1 – Résultats sur les graphes structurés de la bibliothèque TreewidthLIB (1/5) : la colonne (n :m) donne donne respectivement le nombre de sommets et d'arêtes du graphe, (UB) le meilleur majorant de la treewidth disponible dans la bibliothèque, (Algorithmes) les techniques qui ont obtenu ce majorant, (LB) le meilleur minorant de la treewidth disponible dans la bibliothèque et (Algorithmes) les techniques qui ont obtenu ce minorant. La méthode X+ est la triangulation minimale X+TM.

Graphes	n : m	UB	Algorithmes	LB	Algorithmes
130	62 : 316	16	DynProg,LEX-M,MCS-M,Sfill+	13	LBN,LBP
224	63 : 180	11	MinDeg,Mindeg+,MinFill,Minfill+,MSVS,Sfill+	10	LBP
508	63 : 658	27	MinFill,Minfill+,SFill,Sfill+	23	LBP
223	66 : 188	11	All-MCS-M,MCS,MCS+	11	TWDP
140	67 : 208	11	All-MCS	9	LBP
519	67 : 555	16	Dynprog,SFill,Sfill+	16	Dynprog
510	68 : 559	16	SFill,Sfill+	15	LBP
479	69 : 463	17	MCS,MCS+	15	LBP
634	75 : 413	16	MinFill,Minfill+,Sfill+	16	TWDP
567	76 : 218	10	MCS,MCS+	7	Bramble,LBP
119	76 : 421	16	MinFill,Minfill+,Sfill+	16	TWDP
121	76 : 421	16	MinFill,Minfill+,Sfill+	16	TWDP
123	76 : 421	16	MinFill,Minfill+,Sfill+	16	TWDP
640	76 : 421	16	MinFill,Minfill+,Sfill+	15	LBP
243	79 : 265	7	MinDeg,Mindeg+,MinFill,Minfill+	7	LBP
70	80 : 426	15	All-LEX-M,MCS,MCS+	13	LBN,LBP
76	81 : 413	15	LEX-M,MinFill,Minfill+,SFill,Sfill+	14	LBP
84	82 : 327	11	All	11	LBP,MMD
4	87 : 406	13	All-MCS,MCS+	12	LBP,TLB
406	87 : 788	25	LEX-M,LBFS,LBFS+,MCS-M	22	LBP
75	89 : 438	15	LEX-M,MCS,MCS+,MinFill,Minfill+,Sfill+	14	TWDP
69	89 : 448	15	MCS-M,MinDeg,Mindeg+,MSVS,Sfill+	15	TWDP
225	90 : 220	11	All-MCS-M,MCS,MCS+	11	TWDP
36	91 : 394	25	Sfill+	15	LBP
602	92 : 263	10	MCS,MCS+,SFill,Sfill+	7	Bramble,LBP
64	92 : 521	16	MinFill,Minfill+,Sfill+	15	LBP
615	95 : 272	10	MCS,MCS+	7	Bramble,LBP,TWDP
52	95 : 467	15	LEX-M,MinFill,Minfill+,Sfill+	13	LBP
600	96 : 272	9	MCS,MCS+	7	Bramble,LBP
240	96 : 313	7	MinDeg,Mindeg+,MinFill,Minfill+	7	LBP
604	97 : 277	10	MCS,MCS+	7	LBP
63	97 : 534	16	MinFill,Minfill+,Sfill+	16	TWDP
560	99 : 279	11	MinFill,Minfill+,Sfill+	7	Bramble,LBP
51	99 : 475	15	MinFill,Minfill+,Sfill+	15	TWDP
603	100 : 283	9	MCS+	7	Bramble,LBP
610	100 : 284	11	MCS,MCS+	7	Bramble,LBP
616	100 : 285	11	MCS,MCS+	7	Bramble,LBP
556	100 : 286	13	MinFill,Minfill+,Sfill+	7	Bramble,LBP
605	100 : 286	11	MCS,MCS+	7	LBP
626	100 : 288	13	MCS,MCS+	7	Bramble,LBP
80	100 : 311	10	All	10	LBP,MMD
50	100 : 358	24	Sfill+	15	LBN,LBP
627	101 : 290	13	MCS,MCS+	7	Bramble,LBP

Tab. 8.2 – Résultats sur les graphes structurés de la bibliothèque TreewidthLIB (2/5) : la colonne (n :m) donne donne respectivement le nombre de sommets et d'arêtes du graphe, (UB) le meilleur majorant de la treewidth disponible dans la bibliothèque, (Algorithmes) les techniques qui ont obtenu ce majorant, (LB) le meilleur minorant de la treewidth disponible dans la bibliothèque et (Algorithmes) les techniques qui ont obtenu ce minorant. La méthode X+ est la triangulation minimale X+TM.

Graphes	n : m	UB	Algorithmes	LB	Algorithmes
435	101 : 841	25	LEX-M,MCS-M,MCS+	21	LBP
601	105 : 292	10	Sfill+	7	LBP
590	107 : 283	7	Sfill+	6	Bramble,LBP
136	109 : 211	6	MCS-M,MinFill,Sfill+	6	LBP,MMD
73	110 : 512	16	LEX-M,MCS-M,Sfill+	13	LBP
153	116 : 276	4	MCS+,MinFill,Minfill+	4	LBP
72	116 : 494	16	LEX-M, MCS-M,Sfill+	14	LBP
319	118 : 636	34	MCS+	19	LBN,LBP
89	119 : 348	17	DynProg,MinFill,Minfill+,SFill,Sfill+	14	TWDP
588	124 : 318	10	MCS,MCS+,MinFill,Minfill+,Sfill+	7	LBP
376	124 : 359	16	MCS,MCS+	8	LBP
55	124 : 622	16	MinFill,Minfill+,Sfill+	15	LBP
58	124 : 622	16	MinFill,Minfill+,Sfill+,MSVS	15	LBP
378	125 : 363	13	All-LEX-M,LBFS,LBFS+,MCS-M	8	LBP,TWDP
375	127 : 368	15	MCS,MCS+	8	LBP
167	128 : 1170	22	MCS-M,Sfill+	22	LBN,LBP
488	128 : 1356	32	LEX-M,MCS-M	27	LBP
377	130 : 377	12	MCS,MCS+	8	LBP
507	131 : 1485	31	LEX-M,MCS-M	25	LBP
586	136 : 377	12	MCS,MCS+	8	LBP
54	138 : 599	16	MinFill,Minfill+,Sfill+	16	TWDP
57	138 : 599	16	MinFill,Minfill+,Sfill+	16	TWDP
613	142 : 410	12	MCS,MCS+	8	Bramble,LBP
583	144 : 393	11	MinDeg,Mindeg+,MinFill,Minfill+,Sfill+	8	LBP
380	144 : 416	15	MCS,MCS+	8	Bramble,LBP
608	147 : 427	13	MCS,MCS+,SFill,Sfill+	8	Bramble,LBP
379	150 : 432	14	MCS,MCS+	9	Bramble
614	150 : 432	12	MCS,MCS+	8	Bramble,LBP
609	150 : 436	12	MCS,MCS+	8	Bramble,LBP
581	152 : 428	10	MinFill,Minfill+,Sfill+	8	LBP
540	159 : 431	12	MCS,MCS+	8	Bramble
85	162 : 764	16	MinFill,Minfill+,Sfill+	15	LBP
226	167 : 455	7	MinDeg,Mindeg+,MinFill,Minfill+	6	LBP
468	169 : 1837	33	LEX-M,MCS-M	29	TWDP
469	170 : 1840	33	LEX-M,MCS-M	26	LBP
409	173 : 1843	38	MCS-M	33	TWDP
244	174 : 449	7	MinDeg,Mindeg+,MinFill,Minfill+	7	LBP
410	180 : 1875	38	MCS-M	28	LBP
458	181 : 2203	47	LEX-M,LBFS+,MCS-M	34	TWDP
459	185 : 2221	47	LEX-M,MCS-M	32	LBP
38	187 : 690	52	Sfill+	30	TWDP
219	189 : 366	11	All-MCS-M,MCS,MCS+	10	LBN,LBP
387	189 : 552	14	MCS,MCS+	8	Bramble,LBP

Tab. 8.3 – Résultats sur les graphes structurés de la bibliothèque TreewidthLIB (3/5) : la colonne (n :m) donne donne respectivement le nombre de sommets et d'arêtes du graphe, (UB) le meilleur majorant de la treewidth disponible dans la bibliothèque, (Algorithmes) les techniques qui ont obtenu ce majorant, (LB) le meilleur minorant de la treewidth disponible dans la bibliothèque et (Algorithmes) les techniques qui ont obtenu ce minorant. La méthode X+ est la triangulation minimale X+TM.

Graphes	n : m	UB	Algorithmes	LB	Algorithmes
564	195 : 562	16	MinFill,Minfill+	10	Bramble
606	197 : 572	15	MCS,MCS+	9	Bramble
611	197 : 578	15	MCS,MCS+,MSVS	9	Bramble
386	198 : 571	14	MCS,MCS+	8	Bramble,LBP
607	200 : 580	15	MCS,MCS+	9	Bramble
612	200 : 586	15	MCS,MCS+,MSVS	9	Bramble
77	200 : 721	15	MCS+,MinFill,Minfill+,SFill,Sfill+	13	LBN,LBP
65	200 : 817	16	MinFill,Minfill+,Sfill+	15	LBP
298	215 : 497	9	MCS,MCS+	6	LBP
245	217 : 646	8	MinDeg,Mindeg+,MinFill,Minfill+,Sfill+	8	LBP
299	219 : 509	9	MCS,MCS+	6	LBP
300	220 : 502	9	MCS,MCS+	6	LBP
546	225 : 622	16	MCS,MCS+,SFill,Sfill+	10	Bramble
579	226 : 586	9	MinDeg,Mindeg+,MinFill,Minfill+,Sfill+	7	LBP
361	233 : 10783	66	All	66	LBN,LBP
617	258 : 761	18	MCS,MCS+	9	Bramble,LBP
618	262 : 773	19	MCS,MCS+	9	Bramble,LBP
575	264 : 772	16	MCS,MCS+	8	Bramble
252	271 : 724	8	MinDeg,Mindeg+,MinFill,Minfill+,Sfill+	8	LBN,LBP
371	280 : 788	15	MCS,MCS+	8	LBP
573	299 : 864	14	MCS,MCS+	8	Bramble
147	308 : 1158	15	MinFill,Minfill+,MSVS,SFill,Sfill+	13	TWDP
331	317 : 12720	56	All-Sfill	56	LBN,LBP
239	317 : 674	7	MinDeg,Mindeg+	6	LBP
152	332 : 662	4	MinFill,Minfill+,MSVS,SFill,Sfill+	4	LBP
364	333 : 7907	31	All-MCS-M,MCS,MCS+	31	LBN
362	333 : 7910	31	All-MCS-M,MCS,MCS+	66	LBP
146	339 : 1194	15	MinFill,Minfill+,MSVS,SFill,Sfill+	13	TWDP
56	340 : 1130	16	MinFill,Minfill+,Sfill+	15	LBP
59	340 : 1130	16	MinFill,Minfill+,Sfill+	15	LBP
333	363 : 8897	31	MinFill,Minfill+,Sfill+	31	LBN,LBP
334	364 : 8887	31	MinFill,Minfill+,SFill,Sfill+	31	LBN
554	400 : 1183	22	MCS+	11	Bramble
391	404 : 1148	11	MSVS	8	LBP
151	413 : 819	4	MinDeg,Mindeg+,MinFill,Minfill+	4	LBP,TLB
390	417 : 1179	11	MinFill,Minfill+,MSVS	7	Bramble
569	439 : 1297	21	MCS,MCS+	11	Bramble
156	441 : 806	10	MinDeg,Mindeg+,MinFill,Minfill+,SFill,Sfill+	8	LBP
594	442 : 1286	12	MCS,MCS+	12	Bramble
340	450 : 8169	270	Tabu search treewidth	78	LBP
630	488 : 1452	23	MCS,MCS+	14	Bramble
631	493 : 1467	23	MCS,MCS+	14	Bramble
534	574 : 1708	25	MCS,MCS+	14	Bramble

Tab. 8.4 – Résultats sur les graphes structurés de la bibliothèque TreewidthLIB (4/5) : la colonne (n :m) donne donne respectivement le nombre de sommets et d'arêtes du graphe, (UB) le meilleur majorant de la treewidth disponible dans la bibliothèque, (Algorithmes) les techniques qui ont obtenu ce majorant, (LB) le meilleur minorant de la treewidth disponible dans la bibliothèque et (Algorithmes) les techniques qui ont obtenu ce minorant. La méthode X+ est la triangulation minimale X+TM.

Graphes	n : m	UB	Algorithmes	LB	Algorithmes
562	575 : 1699	29	MinFill,Minfill+	14	Bramble
301	620 : 1277	11	Sfill+	7	LBP
596	644 : 1785	13	MinFill,Minfill+,Sfill+	7	Bramble
628	650 : 1938	27	MCS+	13	Bramble
597	654 : 1806	13	MinFill,Minfill+,Sfill+	7	Bramble
629	657 : 1958	27	MCS+	13	Bramble
530	724 : 2117	26	MCS+	16	Bramble
561	783 : 2322	32	MinFill,Minfill+	16	Bramble
592	1002 : 2972	31	Sfill+	16	Bramble
220	1003 : 1662	7	MinDeg,Mindeg+,MinFill	6	LBN,LBP
222	1041 : 1843	8	MinDeg,Mindeg+,MinFill,Minfill+,Sfill+	8	LBP
221	1044 : 1745	7	MinDeg,Mindeg+,MinFill	7	LBN,LBP
544	1060 : 3153	31	MCS,MCS+	15	Bramble
528	1084 : 2869	24	MinFill,Minfill+	12	Bramble
585	1164 : 3475	38	MCS,MCS+	18	Bramble
595	1173 : 3501	35	MCS+	18	Bramble
381	1291 : 3845	37	MCS,MCS+	19	Bramble
552	1304 : 3879	30	Sfill+	17	Bramble
550	1323 : 3950	32	Sfill+	17	Bramble
598	1367 : 4081	38	MCS+	20	Bramble
599	1379 : 4115	40	MCS,MCS+	21	Bramble
622	1390 : 4108	18	MinFill,Minfill+,SFill,Sfill+	14	Bramble
623	1400 : 4138	18	MinFill,Minfill+,SFill,Sfill+	10	Bramble
542	1432 : 4204	43	MCS,MCS+	28	Bramble
98	1497 : 3055	30	MinFill,Minfill+,MSVS,Sfill+	14	Bramble
302	1505 : 3071	28	MinFill,Minfill+,Sfill+	14	Bramble
620	1571 : 4619	23	MCS+	10	Bramble
621	1577 : 4637	24	MCS,MCS+	10	Bramble
385	1654 : 4888	47	MSVS	20	Bramble
384	1655 : 4890	45	SFill,Sfill+	20	Bramble
388	1748 : 4784	33	MinFill,Minfill+	17	Bramble
538	1817 : 5386	42	MCS,MCS+	-	-
548	1889 : 5631	33	Sfill+	18	Bramble
95	1939 : 2935	19	Sfill+	6	LBP
97	1985 : 3109	21	MinFill,Minfill+,Sfill+	5	LBN
93	2048 : 3087	21	MinFill,Minfill+	12	Bramble
632	2099 : 6278	36	MCS,MCS+	23	Bramble
101	2103 : 3973	11	Sfill+	6	LBN
633	2103 : 6290	37	MCS,MCS+	23	Bramble
537	2152 : 6312	49	MCS,MCS+	23	Bramble
535	2319 : 6869	57	MCS,MCS+	41	Bramble
577	2392 : 7125	50	MinFill,Minfill+	21	Bramble
99	3326 : 5147	29	MinFill,Minfill+,Sfill+	14	Bramble

Tab. 8.5 – Résultats sur les graphes structurés de la bibliothèque TreewidthLIB (5/5) : la colonne (n :m) donne donne respectivement le nombre de sommets et d'arêtes du graphe, (UB) le meilleur majorant de la treewidth disponible dans la bibliothèque, (Algorithmes) les techniques qui ont obtenu ce majorant, (LB) le meilleur minorant de la treewidth disponible dans la bibliothèque et (Algorithmes) les techniques qui ont obtenu ce minorant. La méthode X+ est la triangulation minimale X+TM.

Graphes	UB	taillemin-densite	centre	mininter	sep/clust	LBTriang
314	5	5	5	5	sep	5
262	7	7	7	7	sep	7
254	4	4	5	6	sep	4
229	5	5	5	5	sep	5
115	6	6	6	7	sep	6
236	7	8	7	7	sep	7
237	7	8	7	7	sep	7
238	6	8	7	7	sep	7
296	4	5	5	5	sep	4
246	8	8	8	9	sep	9
315	10	11	10	10	sep	10
297	5	6	6	7	sep	5
142	7	7	7	7	sep	7
143	11	11	11	11	sep	11
289	13	13	13	13	sep	13
78	10	10	10	12	sep	10
215	9	10	11	11	sep	11
66	11	11	11	11	sep	11
253	4	4	4	4	sep	4
1	4	4	4	4	sep	4
132	14	16	15	17	sep	18
138	3	3	3	3	sep	3
402	11	13	11	13	sep	11
139	3	3	3	3	sep	4
127	16	18	18	19	sep	19
636	16	18	18	19	sep	18
320	12	12	13	17	clust	14
316	19	20	20	27	sep	20
128	15	15	15	22	sep	16
135	15	17	16	20	sep	19
2	7	7	9	7	sep	7
116	15	16	15	20	sep	17
397	22	27	23	32	sep	24
261	7	7	9	7	sep	7
504	22	26	22	27	sep	26
342	9	10	9	13	clust	10
447	15	17	15	25	sep	15
457	19	24	20	23	sep	24
427	20	24	21	24	sep	21
248	8	8	9	18	clust	12
509	28	28	26	34	clust	28
133	15	16	16	21	clust	18
448	15	17	15	25	sep	17

Tab. 8.6 – Résultats des stratégies de TSep sur les graphes structurés de la librairie Treewidth-LIB (1/5) : la colonne (UB) donne le meilleur majorant de la treewidth disponible dans la bibliothèque et les colonnes suivantes le majorant obtenu par les différentes stratégies de TSep et par la méthode LBTriang.

Graphes	UB	taillemin-densite	centre	mininter	sep/clust	LBTriang
130	16	18	17	20	clust	21
224	11	12	13	20	clust	15
508	27	29	27	32	clust	29
223	11	12	11	20	clust	15
140	11	11	12	11	sep	12
519	16	18	17	20	sep	17
510	16	19	16	25	sep	19
479	17	21	17	24	clust	18
634	16	18	16	19	sep	18
567	10	23	14	17	clust	10
119	16	18	18	23	clust	24
121	16	18	18	23	clust	24
123	16	18	18	23	clust	24
640	16	18	17	22	clust	18
243	7	7	8	44	clust	7
70	15	15	15	16	clust	15
76	15	16	15	17	sep	15
84	11	11	11	11	sep	11
4	13	13	14	13	sep	13
406	25	33	26	35	clust	31
75	15	16	15	21	sep	15
69	15	15	15	15	clust	15
225	11	11	12	31	clust	15
36	25	37	29	44	clust	28
602	10	13	12	27	clust	20
64	16	18	18	19	sep	17
615	10	27	15	24	clust	14
52	15	16	15	21	clust	16
600	9	22	14	17	clust	10
240	7	7	7	38	clust	7
604	10	24	13	16	clust	12
63	16	18	16	19	clust	18
560	11	12	12	25	clust	11
51	15	16	15	21	clust	16
603	9	19	12	14	clust	15
610	11	24	13	30	clust	11
616	11	25	16	22	clust	13
556	13	13	14	25	clust	19
605	11	15	14	23	clust	14
626	13	24	21	34	clust	16
80	10	10	10	12	sep	10
50	24	41	36	46	clust	29
627	13	27	21	34	clust	16

Tab. 8.7 – Résultats des stratégies de TSep sur les graphes structurés de la librairie Treewidth-LIB (2/5) : la colonne (UB) donne le meilleur majorant de la treewidth disponible dans la bibliothèque et les colonnes suivantes le majorant obtenu par les différentes stratégies de TSep et par la méthode LBTriang.

Graphes	UB	taillemin-densite	centre	mininter	sep/clust	LBTriang
435	25	28	25	35	clust	29
601	10	14	13	16	clust	9
590	7	14	11	19	clust	8
136	6	6	6	6	sep	7
73	16	18	16	19	clust	19
153	4	4	4	50	clust	5
72	16	21	16	19	clust	20
319	34	69	51	60	clust	41
89	17	33	25	37	clust	23
588	10	20	17	14	clust	13
376	16	45	37	57	clust	17
55	16	18	17	23	clust	23
58	16	18	17	23	clust	23
378	13	36	38	37	clust	19
375	15	45	37	45	clust	17
167	22	25	23	25	clust	29
488	32	37	33	48	clust	34
377	12	33	29	42	clust	20
507	31	38	30	52	clust	33
586	12	47	28	40	clust	14
54	16	18	16	23	clust	21
57	16	18	16	23	clust	21
613	12	42	27	42	clust	19
583	11	28	15	32	clust	14
380	15	45	43	47	clust	14
608	13	41	28	49	clust	21
379	14	49	41	53	clust	15
614	12	32	34	54	clust	15
609	12	42	29	49	clust	19
581	10	18	35	23	clust	14
540	12	38	32	44	clust	13
85	16	18	18	23	clust	18
226	7	8	8	25	clust	8
468	33	64	34	74	clust	39
469	33	64	32	44	clust	39
409	38	84	48	67	clust	52
244	7	7	27	57	clust	9
410	38	60	45	66	clust	53
458	47	84	56	88	clust	58
459	47	110	59	83	clust	58
38	52	91	101	104	clust	55
219	11	13	13	13	clust	18
387	14	74	52	77	clust	21

Tab. 8.8 – Résultats des stratégies de TSep sur les graphes structurés de la librairie Treewidth-LIB (3/5) : la colonne (UB) donne le meilleur majorant de la treewidth disponible dans la bibliothèque et les colonnes suivantes le majorant obtenu par les différentes stratégies de TSep et par la méthode LBTriang.

Graphes	UB	taillemin-densite	centre	mininter	sep/clust	LBTriang
564	16	71	40	85	clust	14
606	15	68	47	76	clust	22
611	15	61	58	67	clust	24
386	14	67	59	88	clust	17
607	15	73	58	76	clust	17
612	15	65	51	67	clust	26
77	15	16	14	17	clust	18
65	16	18	17	18	clust	20
298	9	72	62	67	clust	15
245	8	10	10	20	clust	11
299	9	83	59	73	clust	18
300	9	83	58	96	clust	16
546	16	78	63	77	clust	17
579	9	14	12	20	clust	12
361	66	66	66	216	sep	66
617	18	102	83	112	clust	26
618	19	96	89	102	clust	22
575	16	111	85	106	clust	17
252	8	10	9	22	clust	13
371	15	56	55	62	clust	20
573	14	70	58	84	clust	25
147	15	52	52	99	clust	36
331	56	56	56	295	sep	56
239	7	10	9	30	clust	10
152	4	21	23	36	clust	5
364	31	31	31	310	sep	34
362	31	31	31	310	sep	34
146	15	60	81	61	clust	36
56	16	18	16	22	clust	21
59	16	18	16	22	clust	21
333	31	35	34	337	sep	38
334	31	35	33	338	sep	38
554	22	156	149	188	clust	38
391	11	69	48	115	clust	24
151	4	20	15	47	clust	5
390	11	81	38	79	clust	24
569	21	181	162	207	clust	34
156	10	14	14	39	clust	12
594	12	191	180	203	clust	35
340	270	353	361	360	clust	297
630	23	230	203	249	clust	25
631	23	236	199	249	clust	25
534	25	266	236	283	clust	33

Tab. 8.9 – Résultats des stratégies de TSep sur les graphes structurés de la librairie Treewidth-LIB (4/5) : la colonne (UB) donne le meilleur majorant de la treewidth disponible dans la bibliothèque et les colonnes suivantes le majorant obtenu par les différentes stratégies de TSep et par la méthode LBTriang.

Graphes	UB	taillemin-densite	centre	mininter	sep/clust	LBTriang
562	29	293	251	307	clust	20
301	11	184	155	181	clust	25
596	13	226	188	274	clust	43
628	27	295	273	297	clust	42
597	13	193	185	241	clust	16
629	27	295	275	306	clust	42
530	26	314	287	345	clust	33
561	32	379	347	417	clust	23
592	31	476	411	526	clust	34
220	7	11	12	59	clust	8
222	8	10	11	21	clust	17
221	7	9	13	81	clust	43
544	31	497	464	516	clust	68
528	24	337	351	411	clust	21
585	38	591	523	631	clust	68
595	35	579	529	613	clust	64
381	37	710	607	793	clust	44
552	30	534	525	578	clust	97
550	32	555	547	637	clust	61
598	38	761	670	807	clust	40
599	40	753	672	809	clust	39
622	18	711	507	703	clust	58
623	18	711	508	721	clust	58
542	43	807	673	805	clust	66
98	30	626	526	640	clust	47
302	28	637	554	649	clust	47
620	23	370	368	432	clust	64
621	24	370	365	433	clust	64
385	47	855	772	917	clust	67
384	45	858	780	917	clust	62
388	33	603	611	700	clust	77
538	42	1021	-	1146	clust	93
548	33	747	-	849	clust	71
95	19	547	-	592	clust	53
97	21	627	-	624	clust	32
93	21	583	-	637	clust	44
632	36	1294	-	1315	clust	154
101	11	677	-	743	clust	36
633	37	1294	-	1315	clust	154
537	49	1259	-	1301	clust	80
535	57	1460	-	1497	clust	89
577	50	1221	-	1308	clust	92
99	29	1144	-	1195	clust	49

Tab. 8.10 – Résultats des stratégies de TSep sur les graphes structurés de la librairie TreewidthLIB (5/5) : la colonne (UB) donne le meilleur majorant de la treewidth disponible dans la bibliothèque et les colonnes suivantes le majorant obtenu par les différentes stratégies de TSep et par la méthode LBTriang.

Calcul et exploitation de recouvrements acycliques pour la résolution de (V)CSP

Résumé. L'objectif de ce travail est de rendre opérationnelles les méthodes structurelles de résolution de problèmes représentés dans le formalisme CSP (problème de satisfaction de contraintes) ou une de ses extensions dans le cas de l'optimisation sous-contraintes, le formalisme VCSP. Ces techniques en tirant profit des propriétés topologiques du problème procurent d'excellentes bornes de complexité en temps, bien meilleures que celles des méthodes énumératives standards. Cependant, elles se révèlent souvent totalement inefficaces en pratique à cause de leur coût prohibitif en espace mémoire, mais également de la rigidité qu'elles imposent dans l'ordre d'affectation des variables. Nous avons mené une étude assez large sur les méthodes de calcul de décompositions arborescentes et la qualité de ces dernières pour une résolution efficace en pratique. Cela a conduit à la définition de techniques qui permettent de faire un bon compromis entre l'espace mémoire requis et la borne de complexité temporelle. Ce compromis est ensuite élargi à un autre critère, la liberté accordée aux heuristiques de choix de variables. Ce point crucial dans l'efficacité pratique des méthodes énumératives, constitue un des points faibles des méthodes structurelles qui imposent en général des ordres statiques dont les performances sont souvent médiocres comparées à celles des ordres dynamiques. Nous avons proposé deux méthodes, BDH et BDH-val, qui permettent de définir des heuristiques efficaces plus dynamiques grâce à l'utilisation d'un recouvrement acyclique du problème qui est une notion plus générale que celle de décomposition arborescente. Elles obtiennent ainsi de bien meilleurs résultats que BTD et BTD-val dont elles constituent des généralisations. Enfin, nous définissons un schéma générique d'algorithmes énumératifs de résolution de CSP basé sur un ensemble quelconque de séparateurs du problème et des fonctions paramétrables qui recouvrent des techniques d'exploitation de la structure, d'apprentissage, de retour en arrière non chronologique et de définition d'heuristiques d'affectation de variables efficaces. Ce cadre permet de capturer BDH, mais également, un large éventail des méthodes structurelles connues, montrant ainsi l'applicabilité de ces stratégies à ces dernières.

Mots clés : Problème de Satisfaction de Contraintes, Optimisation sous-contraintes, méthodes structurelles, décomposition arborescente, triangulation, recouvrement acyclique, séparateur, heuristiques.

Computing and exploiting acyclic coverings for (V)CSP solving

Abstract. This work aims to make efficient structural methods for solving problems represented within the Constraint Satisfaction Problem (CSP) formalism or the VCSP formalism which are an extension of the CSP one to the constraints optimization problem. These techniques derive benefit from the problem topological properties to offer much better theoretical time complexity bounds than thoses given by the standard enumerative approach. Nevertheless, they are often totally inefficient in practice because of the large amount of memory space required and their rigidity in the variable ordering. We run a large study on techniques computing tree decompositions and their quality with respect to an efficient solving in practice. The study led to the definition of new techniques which allow doing a good tradeoff between the space memory required and the time complexity bound. This tradeoff is extended to another citerion : the freedom granted to the variable ordering heuristic. This heuristic is crucial in the efficiency of the enumerative methods. However, it is one of the major drawbacks for structural methods which generally impose static orders which are known to obtain much poor results compared to dynamic ones. We propose two new methods, BDH and BDH-val, that allow to define new efficient dynamic heuristics thanks to the notion of acyclic coverings which are slightly more general than the tree-decomposition. Their results enhance significantly those of BTD and BTD-val whose generalizations they are. Finally, we define a generic enumerative algorithms for solving CSP framework based on a set of separators of the problem and parametrizable functions which covers techniques like exploiting the problem structure, learning, backjumping and defining efficient variable ordering heuristics. This scheme captures BDH and many well known structural methods and shows the applicability of these strategies in these methods.

Key words : Constraint Satisfaction Problem, Constraint optimization, structural methods, tree-decomposition, triangulation, acyclic covering, separator, heuristics.

Laboratoire des Sciences de l'Information et des Systèmes

Domaine Universitaire Saint-Jérôme
Avenue Escadrille Normandie-Niemen 13397 Marseille Cedex 20