Résolution des problèmes (W)CSP et #CSP par approches structurelles : Calcul et exploitation dynamique de décompositions arborescentes

Hélène Kanso

AIX-MARSEILLE UNIVERSITÉ

FACULTÉ DES SCIENCES

LABORATOIRE DES SCIENCES DE L'INFORMATION
ET DES SYSTÈMES - UMR CNRS 7296

Thèse présentée pour obtenir le grade universitaire de docteur

Discipline : Informatique

Hélène KANSO

Résolution des problèmes (W)CSP et #CSP par approches structurelles : Calcul et exploitation dynamique de décompositions arborescentes

Sous réserve de l'avis des rapporteurs,
soutenance prévue le 20 Décembre 2017 devant le jury composé de :

M. Simon DE GIVRY	INRA	Rapporteur
M. Philippe JÉGOU	Aix-Marseille Université	Directeur de thèse
M. Christophe LECOUTRE	Université d'Artois	Rapporteur
M. Thomas SCHIEX	INRA	Examinateur
Mme Christine SOLNON	INSA Lyon	Examinatrice
M. Cyril TERRIOUX	Aix-Marseille Université	Directeur de thèse

Remerciements

Le travail de recherche de cette thèse est loin d'être un travail solitaire. Ce manuscrit est le résultat d'un travail de longue haleine qui a demandé l'aide et la collaboration de plusieurs personnes.

Je tiens tout d'abord à remercier les rapporteurs de cette thèse M. Simon De Givry et M. Christophe Lecoutre pour leur lecture minutieuse du manuscrit et l'intérêt qu'ils ont porté à mon travail. Je remercie également Mme Christine Solnon et M. Thomas Schiex pour me faire l'honneur de participer au jury. Les membres du jury ont contribué par leurs multiples suggestions et remarques à améliorer la qualité de la mémoire. Je leur remercie profondément.

L'aboutissement de ce travail n'aurait pas été possible sans la présence de mes directeurs de thèse M. Philippe Jégou et M. Cyril Terrioux. Malgré mes connaissances très limitées dans le domaine de la programmation par contraintes au début, ils m'ont fait confiance. Ils étaient toujours à l'écoute et répondaient à mes questions même pendant les week-ends ! Je n'ai pas la place pour citer toutes leurs qualités, c'est pourquoi je retiens la sagesse de Philippe et la rigourosité de Cyril qui ont le plus influencé mon travail. Ils m'ont vraiment initié au métier d'un chercheur et pour ça je leur en suis très reconnaissante.

J'adresse aussi mes remerciements aux membres du laboratoire qui ont rendu le quotidien agréable. Je remercie tout particulièrement Djamal, Estelle, Marianna et Chantal qui m'ont supporté pendant les moments difficiles. Je n'oublie pas les personnels administratifs et techniques du laboratoire, les soldats inconnus, sans qui tout aurait été plus compliqué.

Un grand merci à mes parents, à mes frères et à ma petite Reine, à qui je dois tout ce que je suis aujourd'hui. Merci à ma famille en France : Hussein, Manar, Armand et la joie du petit Ralph. Merci à tous ceux et celles que je n'ai pas mentionnés, proches et amis, mais qui se reconnaîtront sûrement. Enfin, merci à mon chéri mon compagnon de chemin qui me ressourçait de patience et d'enthousiasme pour donner le meilleur de moi-même.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Notations

A

est le plus grand nombre de tuples associés à une relation 80

$\mathcal{A}$

est une affectation portant sur un sous-ensemble de variables de X 69

$B(\varphi)$

est l'ensemble des sommets de X de H bloqués par la fonction φ 50

β

est la branch-width du graphe G 51

β_{hbfs}

est la borne supérieure utilisée dans $HBFS$ pour calculer Z_{hbfs} 114

$C = \{c_1, c_2, \ldots, c_m\}$

est l'ensemble des arêtes d'un (hyper)graphe ou de contraintes d'une instance CSP 36, 68

c_{ij}

est une contrainte binaire portant sur les variables x_i et x_j 68

clb

est la borne inférieure courante d'un algorithme de résolution d'une instance WCSP 113

cub

est la borne supérieure courante d'un algorithme de résolution d'une instance WCSP 113

CM

est l'ensemble des cliques maximales de G'_i, résultant de la triangulation de G_i 139

cwd

est la clique-width du graphe G 51

d

est la taille maximale des domaines de D, soit $d = max_{x_i \in X}|D_{x_i}|$ 68

$Desc(E_i)$

est l'ensemble de clusters correspondants aux nœuds situés dans le sous-arbre enraciné en i dans T, c'est-à-dire $\{E_i\} \cup_{E_j \in Fils(E_i)} Desc(E_j)$ 48

$D = \{D_{x_1}, D_{x_2}, \ldots, D_{x_n}\}$

est l'ensemble des domaines finis à raison d'un domaine par variable pour les instances (W)CSP 67, 108

$D_{x_i}^{curr}$

est le domaine courant de la variable x_i 78

e

est le nombre initial d'arêtes du graphe G considéré pour la calcul de la décomposition arborescente 48

E_{bt}

représente le cluster vers lequel $\#EBTD$ fait un retour-arrière 214

E_i

est le cluster suivant à construire par $H\text{-}TD$ 133

$E_{p(i)}$

est le cluster parent du cluster E_i 48

E_r

est le cluster racine de la décomposition arborescente 48

(E, T)

est une décomposition arborescente de G 47

(E', T')

est la décomposition résultante de l'application d'une ou de plusieurs opérations de fusion à (E, T) 175

E_0

est le premier cluster calculé de la décomposition par $H\text{-}TD$ 133

$E = \{E_0, \ldots, E_l\}$

est l'ensemble de clusters de la décomposition calculée par $H\text{-}TD$ 132

$\mathcal{E}$

est la taille de l'espace de recherche 164

$\mathcal{E}_{avant}$

est la taille de l'espace de recherche avant la réalisation d'une affectation donnée 164

$\mathcal{E}_{après}$

est la taille de l'espace de recherche après la réalisation d'une affectation donnée 164

e'

est le nombre d'arêtes du graphe résultant de la triangulation de G 53

E_j^*

est le cluster résultant de la fusion du cluster E_j avec ses descendants, c'est-à-dire $E_j^* = \cup_{E_p \in Desc(E_j)} E_p$ 191

F

est une file d'attente contenant les composantes connexes à traiter par $H\text{-}TD$. 133

fhw

est l'hypertree-width fractionnaire de l'hypergraphe H 51

$Fils(E_i)$

est l'ensemble de clusters fils de E_i pour un enracinement donné 48

$Fut(c_i)$

est l'ensemble de variables de $S(c_i)$ non encore assignées 163

G^d

est l'ensemble de goods enregistrés pour $BTD\text{-}MAC(+Fusion)$, des #goods pour $\#BTD$ et des goods pour $\#EBTD$ 92, 173, 209, 213

$G[V_i \cup X_i] = G_i$

est le sous-graphe de G induit par V_i et X_i pour $H\text{-}TD$ 139

$G_{\subseteq}$

est soit un sous-graphe de G, soit un graphe partiel 36, 37

$G' = (X, C \cup C')$

est un graphe triangulé, résultant de la triangulation de G, avec C', l'ensemble des arêtes ajoutées à G pendant la triangulation 45

G'_O

est le graphe triangulé construit à partir du graphe G et de l'ordre d'élimination O 53

ghw

est la generalized hypertree-width de l'hypergraphe H 49

$H = (X, C)$

est un hypergraphe, avec X l'ensemble des sommets et C l'ensemble des hyperarêtes 40

h

est la hauteur de l'arrangement en arbre de G 95

hw

est l'hypertree-width de l'hypergraphe H 49

k

est le coût maximum dans une instance WCSP 108

$lb(P_j|\mathcal{A})$

est le minorant du sous-problème $P_j|\mathcal{A}$ 116

$LB_{P_j|\mathcal{A}}$

représente la borne inférieure enregistrée pour $P_i|\mathcal{A}$ 116

m

est le nombre d'éléments de C ou de W 36, 40, 68, 108

n

est le nombre d'éléments dans X 36, 40, 67, 108

N^d

est l'ensemble de nogoods enregistrés pour $BTD\text{-}MAC$(+Fusion) ou de $\#EBTD$ 92, 173, 213

N_{hbfs}

est la limite de backtracks utilisée dans $HBFS$ 114

N_r

est l'ensemble des nld-nogoods réduits enregistrés 94

$N(x_i)$

est le voisinage de x_i 37

$N[x_i]$

est le voisinage fermé de x_i 37

$N(X_\subseteq)$

est le voisinage de l'ensemble $X_\subseteq$ 37

$N[X_\subseteq]$

est le voisinage fermé de l'ensemble $X_\subseteq$ 37

$N(x_i, X_\subseteq)$

est le voisinage du sommet x_i circonscrit à $X_\subseteq$ 37

$N(X'_\subseteq, X_\subseteq)$

est le voisinage de $X'_\subseteq$ circonscrit à $X_\subseteq$ 37

$N[X'_\subseteq, X_\subseteq]$

est le voisinage fermé de $X'_\subseteq$ circonscrit à $X_\subseteq$ 37

$N_u(x_i)$

est le voisinage ultérieur de x_i étant donné un ordre d'élimination sur les sommets de G 42

O

est un ordre d'élimination pour les sommets de G 42

$P = (X, D, C)$

est une instance CSP donnée par le triplet (X, D, C) 67

P_j

est le sous-problème enraciné en cluster E_j 90

P_j^{RDS}

est le sous-problème enraciné en E_j pour RDS qui porte sur $V_{Desc(E_j)} \backslash (E_{p(j)} \cap E_j)$ 116

$P_j|\mathcal{A}$

est le sous-problème de P enraciné en E_j et induit par l'affectation $\mathcal{A}$ 90

$Pos(\Sigma)$

est l'ensemble des décisions positives dans Σ 77

$\mathcal{P}(X)$

est l'ensemble des parties de X 40

$\mathcal{P}_2(X)$

est l'ensemble des paires non orientées de sommets de X 36

Q_{E_i}

est une file d'attente contenant les clusters à traiter par les algorithmes à base de BTD 92

r

est l'arité maximale des hyperarêtes de C ou l'arité maximale des contraintes de C, c'est-à-dire $r = max_{c_i \in C}|S(c_i)|$ 40, 68

R

est le nombre de redémarrages réalisés par $BTD\text{-}MAC{+}RST(+Fusion)$ 94

$R(c_i)$

est la relation de compatibilité associée à c_i 68

rwd

est la rank-width du graphe G 52

s

est la taille du plus grand séparateur d'une décomposition 48

s^*

est la taille du plus grand séparateur exploité par $BTD\text{-}DFS{+}DYN$ 196

S

est une borne supérieure sur la taille maximale s permise pour un séparateur de la décomposition pour $H\text{-}TD$ 138

$S(c_i)$

est la portée de l'hyperarête ou de la contrainte c_i 40, 68

$\mathcal{S}(k)$

est la structure de valuation associée à une instance WCSP 108

$\mathcal{S}_T$

est l'arbre de recherche AND/OR 95

sol_{E_j}

est le nombre de solutions du problème enraciné en E_j, soit le problème $P_j|\mathcal{A}[E_j \cap E_{p(j)}]$, s'il est évoqué dans le cadre d'un good structurel exact ou uniquement une borne inférieure sur le nombre de solutions s'il est évoqué dans le cadre d'un good structurel partiel 208, 212

Sol_P

est l'ensemble de solutions d'une instance CSP P 71

$T = (I, F)$

est un arbre avec I l'ensemble de nœuds et F l'ensemble d'arêtes de T 44

(T, χ, λ)

est un hyperarbre (fractionnaire) d'un hypergraphe H avec χ et λ deux fonctions portant sur les sommets de T 49

t_Y

est un tuple portant sur les variables d'indice dans Y 109

V_{desc_i}

est l'ensemble des variables non instanciées de $V_{Desc(E_i)}$ 192

$V_{Desc(E_j)}$

est l'ensemble des variables des clusters de $Desc(E_j)$ 90

V_{E_i}

est l'ensemble de variables à assigner dans E_i 92

V_i

est le voisinage de la composante connexe en cours de traitement par $H\text{-}TD$ .. 132

V'

est l'ensemble de variables représentant un choix entre V_{E_i} et entre V_{desc_i} 193

w

est la largeur arborescente du graphe G 47

w'^{+}

est la largeur de la décomposition (E', T') 177

w_i

est la fonction de coût portant sur la variable x_i 108

$w^i_\emptyset$

désigne la borne inférieure localisée relative au cluster E_i 116

w_Y

est la fonction de coût portant sur l'ensemble de variables d'indice appartenant à Y 110

w^o

est la largeur associée à l'ordre d'élimination dans le cadre de l'élimination de variables 97

w^t

est la largeur de l'ordre d'élimination associé au dtree 96

w^+

est la largeur de la décomposition, c'est-à-dire la taille du plus grand cluster - 1 47

w^{*+}

est la taille du plus grand cluster exploité par $BTD\text{-}DFS{+}DYN$ 196

$\omega(G)$

est la taille de la plus grande clique de G 38

W

est l'ensemble de fonctions de coût d'une instance WCSP 108

$w_{i_1,\ldots,i_p}$

est la fonction de coût portant sur les variables $x_{i_1}, \ldots, x_{i_p}$ d'une instance WCSP 108

$W_{unaires}$

est l'ensemble de fonctions de coût unaires d'une instance WCSP 108

W^+

est l'ensemble de fonctions de coût de W qui ne sont pas unaires ou d'arité nulle d'une instance WCSP 108

$\overline{\Delta W}_{P_j|\mathcal{A}[E_i \cap E_j]}$

est le coût qui quitte le sous-problème P_j une fois le séparateur $E_i \cap E_j$ affecté 116

$\overline{W}^j_\emptyset$

est un minorant relatif au sous-problème P_j 116

$X = \{x_1, x_2, \ldots, x_n\}$

est l'ensemble des sommets d'un (hyper)graphe ou des variables d'une instance (W)CSP 36, 40, 67, 108

$X_{\mathcal{A}}$

est l'ensemble de variables sur lequel porte l'affectation $\mathcal{A}$ 70

X_f

est le sous-ensemble de variables de X dont le domaine est réduit suite à une affectation 164

X_i

est la composante connexe en cours de traitement par $H\text{-}TD$ 132

$X_{\subseteq}$

est un sous-ensemble de sommets de X 36

$\{X_{i_1}, X_{i_2}, \ldots X_{k_i}\}$

est l'ensemble des composantes connexes de $G[X_i \backslash E_i]$ calculé par $H\text{-}TD$, c'est-à-dire les composantes connexes induites par la suppression dans $G[X_i]$ des sommets de E_i 132

X'

est l'ensemble des sommets déjà considérés du graphe par $H\text{-}TD$ 132

X''_i

est l'ensemble des sommets choisi par $H\text{-}TD$ dans X_i pour former E_i 134

Z

est la pile de clusters contenant les clusters à traiter par $\#EBTD$ 214

Z_{hbfs}

est le nombre de backtracks permis pour DFS dans $HBFS$ 114

$Z_{inconnu}$

est une pile locale au cluster courant E_i qui contient chaque cluster fils E_j de E_i pour lequel $\mathcal{A}[E_i \cap E_j]$ ne correspond pas à un good de E_i par rapport à E_j ni à un nogood 216

$Z_{good_{\geq}}$

est une pile locale au cluster courant E_i qui contient chaque cluster fils E_j de E_i pour lequel $\mathcal{A}[E_i \cap E_j]$ correspond à un good partiel de E_i par rapport à E_j .. 216

$2Sec(H)$

est le graphe représentant la 2-section ou le graphe primal de l'hypergraphe H . 41

α_{hbfs}

est la borne inférieure utilisée dans $HBFS$ pour calculer Z_{hbfs} 114

Δ

est un (no)good structurel, un nld-nogood réduit ou un nogood 86

σ

est un ordre d'élimination parfait pour les sommets de G 42

Σ

est une suite de décisions réalisées 76

φ

est une fonction de poids associant à une hyperarête un poids dans le cadre de la décomposition en hyperarbre fractionnaire 50

ρ^*

est le nombre couverture d'arêtes fractionnaire d'un hypergraphe H 50

Λ

est l'ordre de choix de variables partiel ou total suivi pendant la résolution 166

Introduction

Motivation de la thèse

La notion de *contrainte* nous accompagne dans notre vie de tous les jours : de l'organisation de notre emploi du temps au quotidien, au sudoku auquel nous jouons dans le métro, à la gestion de notre budget ou au respect des restrictions caloriques imposées par le diététicien. Au-delà, les exemples abondent dans le monde réel à plus grande échelle, à travers les applications industrielles par exemple, comme la vision, les problèmes de conception, les problèmes de planification, d'ordonnancement ou d'allocation de ressources et la configuration d'équipements. En outre, de nombreux exemples académiques peuvent être cités comme le problème des n-dames, le problème de coloration de graphe, la résolution de puzzles ou de graphes cryptarithmétiques. Une contrainte limite le nombre de possibilités dans un certain espace. Par exemple, le but du problème de coloration de graphe est d'attribuer une couleur à chaque sommet d'un graphe de façon à ce que deux sommets liés par une arête n'aient pas la même couleur. Ainsi, si le graphe est un triangle, nous pouvons facilement déduire que la possibilité d'utiliser seulement une ou deux couleurs est à rejeter.

La première discipline à s'intéresser aux problèmes sous contraintes est sans doute la *recherche opérationnelle* (RO). La RO moderne débute durant la deuxième guerre mondiale afin de servir les opérations militaires comme l'implantation de radars. Malheureusement, les problèmes réels rencontrent des difficultés à être formalisés et résolus du fait notamment de la non-prise en compte des spécificités de chaque problème. C'est ainsi que d'autres approches ont vu le jour. La *programmation par contraintes* (CP) est une approche qui offre un cadre général permettant d'exprimer ces problèmes sous forme d'un ensemble de variables et d'un ensemble de contraintes portant sur ces variables. Elle rassemble sous sa houlette des problèmes issus de l'*intelligence artificielle*, mais aussi de la recherche opérationnelle comme l'ordonnancement ou le problème du voyageur de commerce. Une instance du *problème de satisfaction de contraintes* (CSP) est constituée d'un ensemble de *variables*, d'un *domaine* pour chacune des variables, dans lequel elle puise ses *valeurs*, et d'un ensemble de *contraintes* liant un certain nombre de variables et indiquant les combinaisons de valeurs *autorisées*. La question la plus simple qui se pose est : est-ce que cette instance admet une *solution* ? Ou autrement dit, est ce qu'il existe une *affectation* de variables, c'est-à-dire une association d'une valeur à chaque variable, qui satisfait simultanément toutes les contraintes ? Ce problème constitue un problème de *décision* qui est NP-complet. C'est le problème de satisfaction de contraintes CSP. Des questions plus complexes peuvent se poser comme, par exemple, déterminer le nombre de solutions de l'instance. Il s'agit alors du problème de *comptage* CSP appelé le problème #CSP. Naturellement, ce problème est plus difficile en théorie (il appartient à la classe #P-complet) comme en pratique. Il a de nombreuses applications en intelligence artificielle, comme dans le raisonnement approximatif ou le diagnostic, et dans d'autres domaines plus

éloignés de l'informatique comme la physique statistique ou la chimie. Malgré l'expressivité et la richesse de sa littérature, le cadre CSP présente malheureusement des limitations de formalisme. En effet, il ne permet de modéliser que des contraintes dites *dures* qui doivent absolument être satisfaites. C'est pourquoi de nombreux travaux ont proposé des enrichissements de ce cadre permettant de prendre en compte des *préférences* ou des souhaits. Un exemple d'un tel formalisme est le cadre WCSP [Freuder and Wallace, 1992; Schiex, 2000; Larrosa, 2002] ou le *problème de satisfaction de contraintes pondérées*. Ces pondérations permettent d'associer à chaque solution un *coût*. Plus il est élevé, moins la solution est désirée. Pour le problème WCSP, le but est de trouver la *solution ayant le coût minimum*. Ce problème est NP-difficile. À titre d'exemple, nous citons les problèmes d'allocation de ressources et de routage de véhicules.

Le cadre (W)CSP a suscité l'engouement de la communauté, non seulement du fait de la puissance du formalisme, mais aussi grâce à l'existence de *solveurs* efficaces. Une propriété des *solveurs* a été pointée dans [Puget, 2004; Gent et al., 2006] et dans [Lecoutre, 2013] qui est la *transparence totale* du solveur par rapport à l'utilisateur. En d'autres termes, idéalement, l'utilisateur ne doit pas être conscient des techniques sophistiquées utilisées par le solveur, ou avoir la responsabilité de guider la recherche et de sélectionner le meilleur algorithme filtrant l'espace de recherche. Un défi majeur de la programmation par contraintes est alors de permettre aux utilisateurs des solveurs de les exploiter comme étant des *boîtes noires* et de se focaliser uniquement sur les entrées et les sorties. Pour y parvenir, ce solveur doit être configuré opportunément de façon à avoir le meilleur comportement qu'il aurait pu avoir en le mettant au point finement. Selon Lecoutre, un tel solveur doit être *robuste* et *efficace*. Ces deux notions sont intimement liées. La robustesse relève de la capacité du solveur à compenser les failles de la modélisation du problème et d'homogénéiser son efficacité quel que soit le modèle utilisé pour le même problème. Le solveur compte ainsi sur les algorithmes de renforcement de cohérence locale, les heuristiques adaptatives, les enregistrements et d'autres techniques avancées. L'emploi de telles techniques augmente l'efficacité du solveur et facilite son usage notamment par des utilisateurs non experts. Par conséquent, l'influence de la programmation par contraintes augmenterait dans le monde académique et industriel. Dans cette thèse, nous tentons de faire un pas dans cette direction à savoir augmenter l'efficacité de la résolution et renforcer le pouvoir d'adaptation du solveur à l'instance en question sans configuration préalable de la part de l'utilisateur.

Un solveur est constitué de plusieurs briques : le type de décisions prises, l'emploi de cohérence locale, le type de retour en arrière, l'enregistrement de nouvelles informations, le choix de la prochaine variable ou valeur à instancier et l'exploitation des redémarrages. Un solveur se définit par les choix qu'il fait au niveau de chaque brique. Ces choix sont tous aussi stratégiques les uns que les autres et doivent être combinés savamment pour définir un solveur efficace. Notons que nous n'avons pas toujours accès à ces informations. En effet, les solveurs modernes sont vus comme une seule et même entité. Un solveur classique explore l'espace de recherche de façon exhaustive en se basant sur du *backtracking* et en maintenant éventuellement une propriété de cohérence locale, c'est-à-dire une propriété dont l'application permet de supprimer des valeurs ne pouvant pas participer à une solution. Le type de décisions réalisées peut changer d'un solveur à l'autre ainsi que la façon dont il choisit la prochaine variable ou valeur à instancier. Il peut faire des retours en arrière non chronologiques, procéder à des enregistrements pour éviter de visiter le même sous-espace de recherche plusieurs fois et exploiter des redémarrages. Les méthodes de résolution énumératives les plus efficaces sont basées sur des algorithmes comme MAC [Sabin and Freuder, 1994] et RFL [Nadel, 1988]. Les solveurs modernes sont connus par

leur efficacité qui est mise en avant à travers les compétitions organisées [CP0, 2008; XC1, 2017]. En résolvant le problème du comptage, les solveurs perdent une grande partie de leur efficacité. Ceci est dû à la nécessité d'explorer tout l'espace de recherche et de développer toutes les solutions dont le nombre peut être gigantesque (certaines instances ont un nombre de solutions astronomique dépassant les 10^{300} solutions !). En passant au cadre de l'optimisation, des méthodes de résolution efficaces existent également comme la méthode $HBFS$ [Allouche et al., 2015] qui explore l'espace de recherche en hybridant un parcours en profondeur et un parcours selon le meilleur d'abord. Les solveurs implémentant de telles méthodes sont dits *classiques* puisqu'ils explorent l'espace de recherche comme un seul et unique bloc. Malheureusement, malgré leur efficacité pour la résolution du problème de décision et celui d'optimisation, ces méthodes sont parfois désavantagées à cause de leur complexité en temps qui est en $O(exp(n))$, où n le nombre de variables du problème. Ainsi, le problème du passage à l'échelle pourrait se poser pour de telles méthodes.

Une approche différente, sur laquelle nous allons nous focaliser dans cette thèse, consiste à exploiter la *structure* de l'instance à résoudre. En effet, vu la difficulté du problème de satisfaction de contraintes et de ses extensions, des efforts ont été déployés afin d'identifier des *classes polynomiales*, c'est-à-dire des classes d'instances qui peuvent être résolues en *temps polynomial*. Une façon pour assurer la traitabilité est d'imposer une restriction sur la structure des instances. Le plus souvent, cette structure est représentée par un hypergraphe dit *(hyper)graphe de contraintes* où chaque variable correspond à un *sommet* et chaque contrainte est associée à une arête contenant tous les sommets correspondant aux variables incluses dans sa portée. La notion principale menant à la traitabilité est *l'acyclicité* de l'hypergraphe de contraintes. Toutes les méthodes structurelles sont généralement basées sur un *recouvrement acyclique* de l'hypergraphe de contraintes. Ce recouvrement contribue essentiellement à détecter les *parties indépendantes* et à décomposer le problème initial. Une des meilleures bornes théoriques obtenues est en $O(exp(w))$ où w est la *largeur arborescente* ou *tree-width* du graphe. Cette notion fait référence à la notion de *décomposition arborescente* [Robertson and Seymour, 1986] qui permet de recouvrir acycliquement un graphe par un ensemble de « sacs de sommets » appelés *clusters*. Graphiquement, w est la taille maximale des clusters moins un d'une décomposition *optimale*, c'est-à-dire dont son plus grand cluster est le plus petit en taille parmi toutes les décompositions possibles. w peut être vu comme une mesure de l'acyclicité. En effet, plus la structure de l'hypergraphe s'éloigne d'une structure acyclique, plus w sera élevé. Par exemple, un graphe dont tous les sommets sont liés deux à deux a un w égal à $n-1$ tandis qu'un graphe acyclique a un w égal à 1. L'intérêt à la tree-width va bien au-delà du problème de satisfaction de contraintes. En effet, pour tous les modèles graphiques [Darwiche, 2009; Koller and Friedman, 2009; Dechter, 2013; Pearl, 2014], la classe ayant une *tree-width* bornée est traitable. D'autres méthodes structurelles ont des bornes de complexités encore meilleures (par exemple [Gottlob et al., 1999]) en théorie. Or, elles sont souvent inexploitables en pratique. L'intérêt des méthodes basées sur une décomposition arborescente tient surtout au fait qu'il existe de nombreux problèmes réels pour lesquels $w \ll n$ [Givry et al., 2006]. Afin d'exploiter cette nouvelle borne théorique, la décomposition arborescente est calculée en amont de la résolution. Or, calculer une décomposition optimale, c'est-à-dire ayant une largeur w, est un problème NP-difficile [Arnborg et al., 1987]. L'intérêt des *méthodes exactes* est limité en pratique notamment lorsque le nombre de sommets du graphe augmente. Dès lors, les méthodes heuristiques sont souvent utilisées. Par exemple, $Min\text{-}Fill$ fait fréquemment office de référence dans la communauté CP en calculant une bonne approximation de w, notée w^+, en temps raisonnable. La décomposition arborescente a été exploitée initialement pour résoudre des instances CSP avec la méthode du *tree-clustering*

[Dechter and Pearl, 1989]. Il s'agit d'une approche de type *programmation dynamique* qui vu son coût en mémoire prohibitif est pratiquement inexploitable. Une méthode exploitant la décomposition arborescente qui a montré son efficacité tout en conservant la borne de complexité théorique est BTD [Jégou and Terrioux, 2003]. Cette méthode est un compromis entre les méthodes énumératives classiques et l'approche tree-clustering. Elle suit un schéma énumératif en se laissant guider par la décomposition.

L'objectif de cette thèse est d'améliorer l'efficacité des méthodes structurelles pour la résolution du problème de décision CSP, pour le problème de dénombrement de solutions et pour la résolution du problème WCSP. Si nous avons choisi de nous focaliser sur BTD, cela n'empêche pas que les idées proposées peuvent être appliquées à d'autres méthodes structurelles qui partagent de nombreux points communs avec BTD. L'excellente borne théorique offerte par BTD offre une nouvelle voie et donne un nouvel élan à la résolution de ces problèmes. BTD puise son intérêt dans la souplesse de l'approche énumérative sur laquelle elle est basée, guidée par la décomposition. Elle exploite l'indépendance détectée sur différentes parties pour résoudre les sous-problèmes correspondants *séparément*. En outre, elle procède à l'*enregistrement* du résultat de la résolution dans le but d'élaguer les sous-arbres correspondants à des sous-problèmes déjà visités. Ce faisant, elle résout parfois des instances difficiles pour lesquelles les méthodes classiques auraient plus de difficulté. Cependant, un fossé existe entre la théorie et la pratique que nous souhaitons appréhender dans cette thèse et tenter de réduire. Nous nous intéressons à plusieurs aspects en particulier. Tout d'abord, le comportement de BTD est fortement influencé par la qualité de la décomposition calculée en amont de la résolution. La qualité a été jusque-là liée uniquement à la largeur w^+ de la décomposition. Or, des travaux récents [Jégou et al., 2005; Jégou and Terrioux, 2014b, 2017] ont montré que d'autres paramètres jouent un rôle primordial dans l'efficacité pratique de BTD. En outre, nous constatons que le temps de calcul de la décomposition est parfois trop important dépassant largement le temps de résolution d'une méthode classique, ce qui peut être pénalisant pour ce type d'approches. Au-delà, BTD sacrifie la liberté du choix de la prochaine variable à instancier en faveur de la complexité théorique. En d'autre termes, afin d'obtenir une borne de complexité en temps exponentiel en w^+, BTD doit suivre un ordre partiel de choix de variables imposé par la décomposition. Cependant, nous savons déjà qu'un tel ordre est désavantagé par rapport à un ordre totalement dynamique comme dans le cas des méthodes non structurelles. Aussi, comme la décomposition est calculée avant la résolution sur la base de critères purement structurels, l'ordre induit n'est *a priori* pas lié à un ordre souhaitable vis-à-vis de la résolution. Finalement, nous avons remarqué que la façon dont est exploitée la décomposition n'est pas toujours en adéquation avec la nature du problème que nous essayons de résoudre comme c'est le cas du problème du comptage.

Nous inscrivons cette thèse parmi les travaux qui visent à rendre les solveurs modernes indissociables aux yeux des utilisateurs [Blet, 2015]. Nous contribuons à l'augmentation de l'autonomie des solveurs qui va leur permettre de s'adapter à la nature de l'instance à résoudre sans intervention requise de la part de l'utilisateur. L'objectif est de mettre en valeur la « brique structure » des solveurs qui n'est exploitée que rarement par ces derniers. L'ambition de cette thèse est que la brique structure soit une brique primordiale dans les solveurs les plus efficaces et qu'elle y soit intégrée par défaut.

Organisation et contributions de la thèse

Cette thèse est organisée en six chapitres. Les deux premiers chapitres présentent le bagage nécessaire pour la compréhension de cette thèse, à savoir un rappel des notions de

décompositions existantes, des méthodes de calcul d'une décomposition arborescente et des notions incontournables des problèmes CSP, #CSP et WCSP. Les chapitres suivants sont consacrés aux contributions. Dans cette thèse, nous présentons quatre contributions principales pour la résolution des problèmes CSP, #CSP et WCSP. Les grandes lignes sont présentées ci-dessous.

Calcul de la décomposition arborescente (chapitre 3) Les contributions de ce chapitre ont été présentées dans [Jégou et al., 2015b,a]. Deux paramètres sont au cœur de ce travail : le *temps* de calcul de la décomposition et sa *qualité*. Habituellement, le calcul d'une décomposition arborescente se fait par le biais des *méthodes de triangulation* qui peuvent être coûteuses notamment lorsqu'il s'agit de « gros » graphes ayant un grand nombre de sommets. Ce fait est d'autant plus handicapant que les méthodes de résolution classiques attaquent directement l'instance en question et peuvent même la résoudre avant que le calcul de la décomposition ne soit terminé. Au-delà du temps de calcul, les méthodes de calcul de décomposition visent à minimiser la taille des clusters. Pourtant, d'autres critères sont entrés en jeu récemment comme la taille des séparateurs (i.e. la taille des intersections entre clusters) et la connexité des clusters.

À travers ce travail, nous proposons un cadre de calcul de décomposition général qui a la vertu d'améliorer le temps de calcul de décomposition en permettant un calcul qui ne soit pas forcément basé sur la triangulation. Il permet aussi le paramétrage de la décomposition à calculer afin de capturer des critères voulus par l'utilisateur comme la largeur arborescente w ou d'autres critères plus pertinents à l'égard de la résolution.

Fusion dynamique de la décomposition dans le cas du problème CSP (chapitre 4) Les contributions de ce chapitre ont été présentées dans [Jégou et al., 2016c,b]. Le calcul de la décomposition en amont de la résolution est dissocié de la résolution elle-même. Or, cette décomposition pèse énormément sur la suite de la résolution étant donnée qu'elle imposera un ordre partiel sur les variables tout au long de la résolution. En outre, il est fort probable que cet ordre soit inopportun vis-à-vis du souhait d'une heuristique de choix de variables ayant toute la liberté quant au choix de la prochaine variable.

Nous proposons alors via ce travail de changer la décomposition pendant la résolution par la *fusion dynamique* des clusters, i.e. la mise en commun des variables de deux clusters pour former un seul cluster. Ce cadre permet de fusionner des clusters de sorte que l'heuristique de choix de variables acquiert davantage de liberté sur le choix de la prochaine variable. La fusion des clusters se fait grâce aux recommandations de l'heuristique de choix de variables. La première décomposition calculée se voit ainsi attribuer moins d'importance en permettant sa correction pendant la résolution. Finalement, il assure l'aspect adaptatif de l'algorithme à la nature de l'instance à résoudre et se rapprocher d'un solveur « boîte noire ».

Changement dynamique de la décomposition dans le cas du problème WCSP (chapitre 5) Les contributions de ce chapitre ont été présentées dans [Jégou et al., 2017b,a]. La résolution des instances WCSP par le biais des méthodes classiques est actuellement très efficace. En particulier, l'introduction de l'algorithme $HBFS$ a significativement amélioré leur résolution. Nous pouvons alors légitimement nous poser la question de l'intérêt d'exploiter une décomposition.

Nous proposons ici un cadre qui exploite la décomposition d'une manière plus flexible que son exploitation classique. Il rejoint le cadre précédent pour son pouvoir d'adaptation aux particularités de l'instance en question. Il permet en plus d'utiliser la décomposition

pour un sous-problème uniquement lorsque la résolution sans décomposition semble inefficace. En outre, il permet de répondre aux retours d'information de l'algorithme en cours d'exécution afin de décider l'exploitation ou la non-exploitation de la décomposition.

Amélioration de #BTD dans le cas du problème #CSP (chapitre 6) Les contributions de ce chapitre ont été présentées dans [Jégou et al., 2016a]. L'adaptation de BTD ($\#BTD$ [Favier et al., 2009]) à la résolution des instances #CSP a amélioré significativement les performances des méthodes exactes. En effet, l'exploitation des enregistrements (le nombre de solutions d'un sous-problème dans ce cas) a permis d'éviter de nombreuses redondances et ainsi de réussir à résoudre des instances ayant un très grand nombre de solutions. Néanmoins, la façon dont $\#BTD$ parcourt la décomposition n'est pas adaptée au problème du comptage et pourrait induire de coûteux calculs inutiles. Plus précisément, le nombre d'extensions d'une affectation pour un sous-problème peut être calculé sans forcément être utilisé si l'affectation en question n'admet pas au moins une extension cohérente pour les autres sous-problèmes.

Nous proposons alors une amélioration de cet algorithme qui vise à modifier la façon dont la décomposition est parcourue. Son but est de s'assurer de l'existence d'au moins une extension cohérente de l'affectation courante de chaque sous-problème avant de compter toutes les extensions cohérentes possibles.

Première partie

État de l'art

Chapitre 1

(Hyper)graphes et décompositions

Sommaire

1.1 Introduction

Ce chapitre ainsi que le chapitre suivant fourniront le bagage nécessaire pour la compréhension des objectifs de cette thèse et des contributions proposées. Il présente également des informations complémentaires afin de permettre au lecteur d'avoir une meilleure vue d'ensemble et de mieux situer nos travaux. Ce chapitre ne présente pas un état de l'art exhaustif. Cependant, il permet de faire la lumière sur les travaux les plus influents dans les domaines concernés. Dans ce premier chapitre, nous nous intéressons aux notions de décomposition les plus connues, permettant de capturer la structure d'un (hyper)graphe dans la section 1.3.1, après avoir rappelé d'abord des notions incontournables des (hyper)graphes. Nous nous focalisons après, dans la section 1.3.2, sur les méthodes de calcul d'une décomposition arborescente qui constitue la brique de base de tous nos travaux.

1.2 (Hyper)Graphes

Dans cette partie, nous rappelons les notions préliminaires découlant de la théorie des graphes. Nous abordons les concepts relatifs à la définition d'un graphe et nous passons au cas plus général des hypergraphes (du fait qu'un graphe est un cas particulier d'hypergraphes). Nous nous focalisons après sur le cas des graphes triangulés vu leur importance dans la compréhension des objectifs de cette thèse. Pour des notions plus poussées, le lecteur intéressé pourra consulter les ouvrages de Berge [Berge, 1973], de Golumbic [Golumbic, 2004], de Bondy et Murty [Bondy and Murty, 1976] ou de Brandstädt, Le et Spinrad [Brandstädt et al., 1999].

1.2.1 Notations et définitions basiques

Nous énonçons d'abord les notations et les définitions qui concernent les graphes.

Graphes

Soit $X = \{x_1, x_2, \ldots, x_n\}$ un ensemble fini d'éléments appelés *sommets*. n désigne le nombre de sommets de X, c'est-à-dire $|X| = n$. Pour pouvoir définir un graphe, nous utilisons la notion de l'ensemble des paires de sommets de X noté $\mathcal{P}_2(X)$. D'où,

Définition 1 *Un* graphe G *est une paire* (X, C) *si* $C \subseteq \mathcal{P}_2(X)$ *avec* X *l'ensemble des sommets de* G *et* $C = \{c_1, c_2, \ldots, c_m\}$ *l'ensemble des* m arêtes *de* G.

Comme il n'existe aucune orientation dans une arête, elle est définie comme étant un ensemble de deux sommets $\{x_i, x_j\}$ avec $1 \leq i, j \leq n$. La figure 1.1 montre, par exemple, un graphe $G = (X, C)$ ayant 17 sommets et 21 arêtes avec :
$X = \{x_1, x_2, \ldots, x_{17}\}$ et
$C = \{\{x_1, x_2\}, \{x_1, x_3\}, \{x_2, x_5\}, \{x_3, x_6\}, \{x_5, x_9\}, \{x_6, x_9\}, \{x_9, x_{12}\}, \{x_4, x_7\},$
$\{x_4, x_8\}, \{x_7, x_8\}, \{x_7, x_{13}\}, \{x_8, x_{14}\}, \{x_{13}, x_{14}\}, \{x_7, x_{14}\}, \{x_8, x_{13}\}, \{x_{10}, x_{11}\},$
$\{x_{15}, x_{16}\}, \{x_{10}, x_{15}\}, \{x_{11}, x_{16}\}, \{x_{11}, x_{17}\}, \{x_{16}, x_{17}\}\}$.

Deux notions rattachées aux graphes ont un intérêt particulier : le sous-graphe de G induit par un ensemble de sommets et le graphe partiel de G induit par un ensemble d'arêtes.

Définition 2 *Soit* $X_\subseteq$ *un sous-ensemble de sommets de* X. *Le* sous-graphe induit *par* $X_\subseteq$ *est le graphe* $G_\subseteq = (X_\subseteq, C_\subseteq)$ *avec* $C_\subseteq = \{\{x_i, x_j\} : \{x_i, x_i\} \in C$ *et* $x_i, x_j \in X_\subseteq\}$.

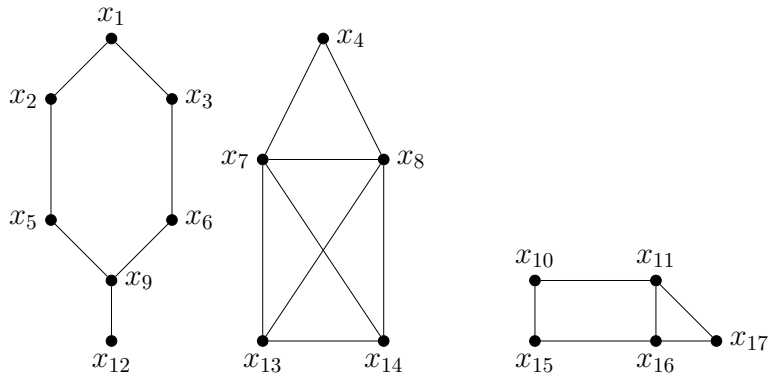


FIGURE 1.1 – Un graphe non orienté.

En d'autres termes, $C_{\subseteq}$ contient toutes les arêtes de C ayant leurs deux extrémités dans $X_{\subseteq}$. Le graphe $G_{\subseteq}$ est dit induit par $X_{\subseteq}$ et est noté $G[X_{\subseteq}]$. Sur l'exemple de la figure 1.1, soit $X_{\subseteq} = \{x_4, x_7, x_8, x_{13}, x_{14}\}$. Alors, $G[X_{\subseteq}]$ désigne un graphe de 5 sommets (soit x_4, x_7, x_8 x_{13} et x_{14}) et 8 arêtes avec $C_{\subseteq} = \{\{x_4, x_7\}, \{x_4, x_8\}, \{x_7, x_8\}, \{x_7, x_{13}\}, \{x_8, x_{14}\}, \{x_{13}, x_{14}\}, \{x_7, x_{14}\}, \{x_8, x_{13}\}\}$.

Définition 3 *Soit $C_{\subseteq}$ un sous-ensemble d'arêtes de C. Le* graphe partiel induit par $C_{\subseteq}$ *d'un graphe G est le graphe $G_{\subseteq}$ ayant les mêmes sommets que G et contenant les arêtes de $C_{\subseteq}$.*

Sur l'exemple de la figure 1.1, soit $C_{\subseteq} = \{\{x_{10}, x_{11}\}, \{x_{15}, x_{16}\}, \{x_{16}, x_{17}\}\}$. Ainsi, le graphe partiel induit par $C_{\subseteq}$ contient tous les sommets de X mais ne considère que les arêtes de $C_{\subseteq}$. En outre, nous pouvons aussi parler du sous-graphe d'un graphe partiel de G. Par conséquent, le graphe $G_{\subseteq} = (\{x_{15}, x_{16}, x_{17}\}, \{\{x_{15}, x_{16}\}, \{x_{16}, x_{17}\}\})$ est un sous-graphe du graphe partiel induit par $C_{\subseteq}$.

Nous nous intéressons maintenant à la notion de *voisinage*.

Définition 4 *Deux sommets x_i et x_j sont dits* voisins ou adjacents *dans G si $\{x_i, x_j\} \in C$.*

Sur l'exemple de la figure 1.1, x_7 et x_{14} sont des voisins tandis que x_4 et x_{14} ne le sont pas.

Définition 5 *Le voisinage d'un sommet x_i dans G est l'ensemble des sommets voisins $N(x_i) = \{x_j \in X : \{x_i, x_j\} \in C\}$. Le voisinage fermé de x_i est $N[x_i] = \{x_i\} \cup N(x_i)$. Au niveau d'un sous-ensemble de sommets $X_{\subseteq}$, son voisinage est défini comme l'ensemble $N(X_{\subseteq}) = \cup_{x_i \in X_{\subseteq}} N(x_i) \backslash X_{\subseteq}$ (l'ensemble $\cup_{x_i \in X_{\subseteq}} N(x_i)$ privé de $X_{\subseteq}$) et son voisinage fermé $N[X_{\subseteq}] = \cup_{x_i \in X_{\subseteq}} N[x_i]$.*

Par exemple, dans la figure 1.1, $N(x_4) = \{x_7, x_8\}$ et $N[x_4] = \{x_4, x_7, x_8\}$. En outre, $N(\{x_1, x_2\}) = \{x_3, x_5\}$ et $N[\{x_1, x_2\}] = \{x_1, x_2, x_3, x_5\}$. Nous définissons aussi la notion de *voisinage circonscrit à un sous-ensemble* qui nous sera particulièrement utile dans la suite de la thèse.

Définition 6 *Soit $X_{\subseteq}$ et $X'_{\subseteq}$ deux sous-ensembles disjoints de X. Le voisinage du sommet x_i circonscrit à $X_{\subseteq}$ est $N(x_i, X_{\subseteq}) = \{x_j \in X_{\subseteq} : \{x_i, x_j\} \in C\}$. Le voisinage de l'ensemble $X'_{\subseteq}$ circonscrit à $X_{\subseteq}$ est $N(X'_{\subseteq}, X_{\subseteq}) = \cup_{x_i \in X'_{\subseteq}} N(x_i, X_{\subseteq})$. Son voisinage fermé est $N[X'_{\subseteq}, X_{\subseteq}] = X'_{\subseteq} \cup N(X'_{\subseteq}, X_{\subseteq})$.*

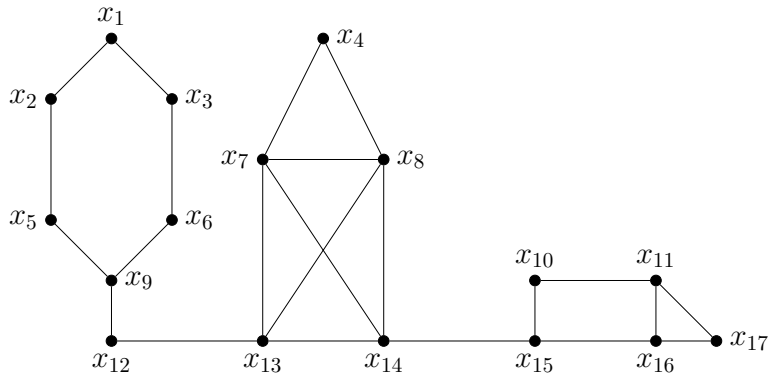


FIGURE 1.2 – Un graphe connexe.

Sur l'exemple de la figure 1.1, si $X'_{\subseteq} = \{x_{10}, x_{15}\}$ et $X_{\subseteq} = \{x_{11}, x_{16}, x_{17}\}$ alors $N(X'_{\subseteq}, X_{\subseteq}) = \{x_{11}, x_{16}\}$ et $N[X'_{\subseteq}, X_{\subseteq}] = \{x_{10}, x_{11}, x_{15}, x_{16}\}$.

Définition 7 *Un graphe G est dit* complet *si pour tout x_i, $x_j \in X$ avec $x_i \neq x_j$, x_i et x_j sont voisins. Ainsi pour chaque sommet x_i de G, $N[x_i] = X$.*

Le graphe de la figure 1.1 n'est évidemment pas complet. Cependant, le sous-graphe de G induit par l'ensemble de sommets $X_{\subseteq} = \{x_7, x_8, x_{13}, x_{14}\}$, $G[X_{\subseteq}]$, est un graphe complet. L'exemple ci-dessus nous mène à la définition d'une *clique*.

Définition 8 *Un sous-ensemble $X_{\subseteq}$ de X de a sommets induit une* a-clique *d'un graphe G si $G[X_{\subseteq}]$ est complet. Une clique $X_{\subseteq}$ est dite* maximale *si G ne contient aucune clique contenant strictement $X_{\subseteq}$ comme un sous-ensemble. Une clique est* maximum *si G ne contient aucune clique de cardinalité supérieure.*

En particulier, un sommet est une 1-clique et une arête est une 2-clique. La clique $X_{\subseteq} = \{x_7, x_8, x_{13}, x_{14}\}$ est, selon la définition, maximale et maximum à la fois. Une attention particulière est portée sur la taille de la plus grande clique de G.

Définition 9 *$\omega(G)$ est le nombre de sommets dans une clique maximum de G, c'est-à-dire la taille de la plus grande clique de G.*

Concernant le graphe de la figure 1.1, $\omega(G) = 4$.

Définition 10 *Une chaîne de x_{k_1} à x_{k_r} dans G est une suite de sommets $[x_{k_1}, x_{k_2}, \ldots, x_{k_{r-1}}, x_{k_r}]$ tel que deux sommets consécutifs sont voisins, c'est-à-dire $\forall i \in \{1, \ldots, r-1\}\{x_{k_i}, x_{k_{i+1}}\} \in C$. Nous disons que x_{k_1} et x_{k_r} sont* mutuellement accessibles.

Par exemple, $[x_4, x_7, x_8, x_{13}, x_{14}]$ est une chaîne de x_4 à x_{14} dans l'exemple de la figure 1.1. Cette notion de chemin va nous permettre de définir une notion essentielle : la *connexité*.

Définition 11 *Un graphe G est* connexe *s'il existe une chaîne entre chaque paire de sommets de G, c'est-à-dire tous ses sommets sont mutuellement accessibles.*

Le graphe de la figure 1.1 n'est pas connexe puisqu'il n'existe aucun chemin entre les sommets x_4 et x_{16} par exemple. Au contraire, le graphe de la figure 1.2 est connexe. Dans un graphe non connexe nous pouvons distinguer plusieurs composantes indépendantes appelées les *composantes connexes* du graphe.

Définition 12 *$X_\subseteq \subseteq X$ est une composante connexe de G si et seulement si $G[X_\subseteq]$ est un graphe connexe et s'il n'existe aucun ensemble $X'_\subseteq$ tel que $X_\subseteq \subset X'_\subseteq$ et $G[X'_\subseteq]$ est un graphe connexe.*

Sur le graphe de la figure 1.1, nous distinguons 3 composantes connexes :
$X_1 = \{x_1, x_2, x_3, x_5, x_6, x_9, x_{12}\}$, $X_2 = \{x_4, x_7, x_8, x_{13}, x_{14}\}$ et $X_3 = \{x_{10}, x_{11}, x_{15}, x_{16}, x_{17}\}$.
Il est à noter que comme la relation d'accessibilité mutuelle est une relation d'équivalence, les composantes connexes constituent les composantes maximales de cette relation.

Par la suite, nous ne considérons que des graphes connexes. En effet, dans le cadre de cette thèse, les graphes non connexes sont traités en considérant chaque composante connexe séparément. Nous définissons maintenant un élément primordial dans cette thèse : le *séparateur*.

Définition 13 *Soient x_i et x_j deux sommets de G. Un ensemble de sommets $Y \subseteq X$ est appelé x_i,x_j-séparateur si x_i et x_j se trouvent dans des composantes connexes différentes de $G\backslash Y$ qui représente le graphe G privé des sommets de Y. Un x_i,x_j-séparateur est dit x_i,x_j-séparateur* minimal *s'il ne contient pas un autre x_i,x_j-séparateur. L'ensemble de sommets Y est appelé séparateur minimal de G s'il existe une paire de sommets x_i et x_j tel que Y est un x_i,x_j-séparateur minimal.*

Sur l'exemple de la figure 1.2, $\{x_{15}\}$ est un séparateur minimal des sommets x_{14} et x_{16}. En plus, $\{x_{11}, x_{15}\}$ est un séparateur minimal des sommets x_{10} et x_{16}. Nous remarquons ainsi qu'un séparateur minimal peut être strictement inclus dans un autre séparateur minimal ($\{x_{15}\} \subset \{x_{11}, x_{15}\}$), mais pour des paires de sommets différentes. Nous avons donc besoin d'une caractérisation plus précise de la minimalité. Nous définissons d'abord la notion de la *composante connexe pleine*.

Définition 14 *Soit Y un ensemble de sommets de G et soit X_i une composante connexe parmi les composantes connexes de $G\backslash Y$. Si tout sommet de Y admet au moins un voisin dans X_i, nous dirons que X_i est une* composante connexe pleine associée à Y *dans G. Ainsi, si $N(X_i) = Y$ alors X_i est une composante connexe pleine associée à Y.*

L'idée consiste à ne considérer dans le séparateur que les sommets qui s'avèrent nécessaires pour déconnecter X_i du reste du graphe. Sur la figure 1.2, soit $Y = \{x_8, x_{13}, x_{14}\}$. La figure 1.3 montre le graphe résultant de la suppression de Y de G. Les composantes connexes de $G\backslash Y$ sont : $X_1 = \{x_4, x_7\}$, $X_2 = \{x_1, x_2, x_3, x_5, x_6, x_9, x_{12}\}$ et $X_3 = \{x_{10}, x_{11}, x_{15}, x_{16}, x_{17}\}$. Sur la figure 1.2, on constate que : $N(X_1) = \{x_8, x_{13}, x_{14}\}$, $N(X_2) = \{x_{13}\}$ et $N(X_3) = \{x_{14}\}$. Ainsi, la seule composante connexe pleine associée à Y est X_1. En se basant sur la notion de composante connexe pleine, nous définissons la notion du *séparateur minimal* non rattachée à une paire de sommets.

Définition 15 *Y est un* séparateur minimal *de G si et seulement si $G\backslash Y$ a au moins deux composantes connexes pleines associées à Y.*

D'après cette définition, $Y = \{x_{14}\}$ est, par exemple, un séparateur minimal du graphe de la figure 1.2 tandis que $Y = \{x_8, x_{13}, x_{14}\}$ ne l'est pas.

Une autre notion essentielle, qui est effectivement au cœur de l'intérêt cette thèse, est la notion d'*acyclicité*.

Définition 16 *Un* cycle de longueur k_r *dans G est une chaîne $[x_{k_1}, x_{k_2}, \ldots, x_{k_{r-1}}, x_{k_r}, x_{k_1}]$ avec $r \geq 3$ ayant au moins 3 sommets distincts.*

Dans la figure 1.2, $[x_4, x_7, x_{13}, x_{14}, x_8, x_4]$ est un cycle de longueur 5.

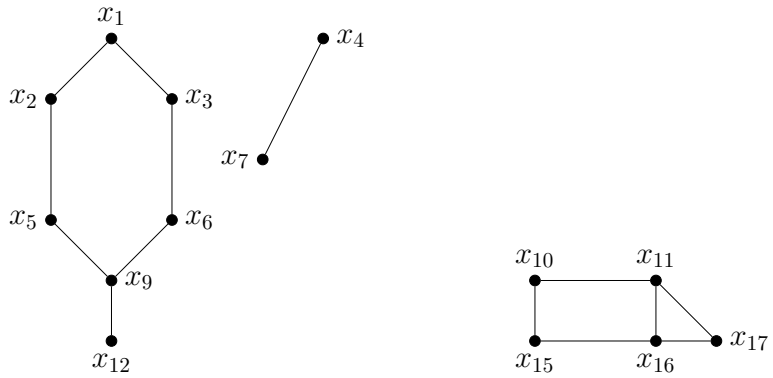


FIGURE 1.3 – Le graphe résultant de la suppression de $Y = \{x_8, x_{13}, x_{14}\}$ du graphe de la figure 1.2.

Définition 17 *Un graphe est* acyclique *si et seulement s'il ne contient aucun cycle.*

Ainsi, le graphe de la figure 1.2 n'est sûrement pas acyclique. En particulier, un arbre est un graphe connexe acyclique.

Hypergraphes :

Nous nous focalisons dans cette partie sur les hypergraphes. Un hypergraphe est basé sur un ensemble de sommets $X = \{x_1, x_2, \ldots, x_n\}$ et un ensemble d'hyperarêtes. Une hyperarête se distingue d'une arête par le nombre de sommets sur lequel elle porte. Si une arête est un sous-ensemble d'exactement deux sommets de X (dite d'arité 2), une hyperarête peut porter sur un nombre quelconque de sommets (dite d'arité quelconque). De ce fait, un graphe est un cas particulier d'un hypergraphe.

Soit $\mathcal{P}(X)$ l'ensemble des parties de X.

Définition 18 *Un* hypergraphe $H = (X, C)$ *est tel que* $C \subseteq \mathcal{P}(X)$ *avec* X *l'ensemble des* n sommets *et* C *l'ensemble des* m hyperarêtes. $S(c_i)$ *est l'ensemble de sommets sur lesquels portent une hyperarête* c_i. *La taille maximale d'une hyperarête, c'est-à-dire le nombre de sommets de l'hyperarête portant sur le plus de sommets est appelée l'*arité *de* C *et est notée* r.

La figure 1.4 montre un hypergraphe de 17 sommets et 6 hyperarêtes avec :

- $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{17}\}$ et

- $C = \{\{x_1, x_2, x_3, x_4, x_5\}, \{x_3, x_5, x_6, x_7, x_8\}, \{x_8, x_9, x_{10}, x_{11}\}, \{x_2, x_{10}, x_{12}, x_{13}\}, \{x_1, x_{14}, x_{15}, x_{16}\}, \{x_1, x_2, x_{17}\}\}$.

Les hypergraphes sont des objets plus complexes à manipuler en théorie et en pratique. Par exemple, certaines notions simples au niveau des graphes comme l'acyclicité possède plusieurs généralisations comme l'α-acyclicité [Beeri et al., 1983], la β-acyclicité [Graham, 1980], ... La littérature sur les hypergraphes est ainsi moins riche que celle sur les graphes. Aussi, il est souvent nécessaire de se ramener à un graphe, notamment via la notion de 2-section (aussi appelée graphe primal [Berge, 1973]). Ce faisant, nous pouvons ainsi exploiter toutes les notions et les concepts relatifs à celui-ci. Nous remarquons aussi que sa

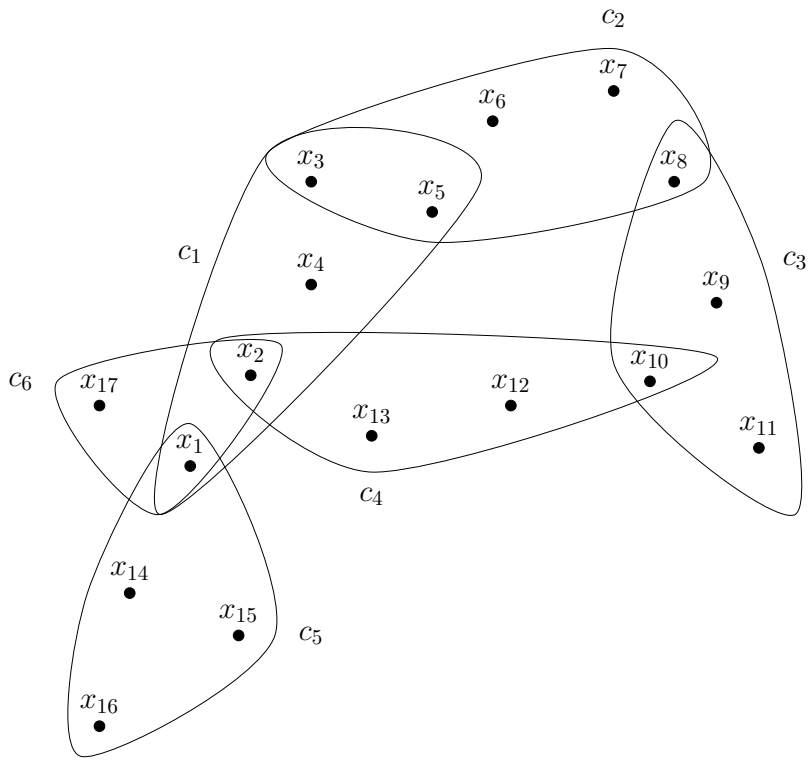


FIGURE 1.4 – Un hypergraphe.

construction est particulièrement facile. C'est ainsi que tout au long de cette thèse, nous exploitons le graphe primal pour construire un graphe à partir d'un hypergraphe.

Définition 19 *Soit $H = (X, C)$ un hypergraphe. La 2-section de H est le graphe $2Sec(H) = (X, C')$ avec $\{x_i, x_j\} \in C'$ si et seulement s'il existe une hyperarête $c_k \in C$ telle que $\{x_i, x_j\} \in c_k$.*

La figure 1.5 est une représentation graphique de la 2-section de l'hypergraphe de la figure 1.4.

Nous évoquons maintenant la définition de l'α-acyclicité [Beeri et al., 1983] qui va nous intéresser dans cette thèse :

Définition 20 *Un hypergraphe (X, C) est α-acyclique, s'il existe un ordre $(c_{i_1}, c_{i_2}, \ldots, c_{i_m})$ tel que $\forall q, 1 < q \leq m, \exists p < q$, $(S(c_{i_q}) \cap \bigcup_{j=1}^{q-1} S(c_{i_j})) \subseteq S(c_{i_p})$.*

En particulier, l'hypergraphe de la figure 1.4 n'est pas α-acyclique. Dans cette thèse, lorsque nous parlons d'un hypergraphe acyclique, nous visons implicitement la notion de l'α-acyclicité.

1.2.2 Graphes triangulés

Nous rappelons maintenant la notion des graphes triangulés [Hajnal and Surányi, 1958] qui joue un rôle important dans la compréhension de la suite de cette thèse.

Définition 21 *Un graphe G est* triangulé *si tout cycle de G de longueur strictement supérieure à 3 possède une corde, c'est-à-dire une arête reliant deux sommets non consécutifs du cycle.*

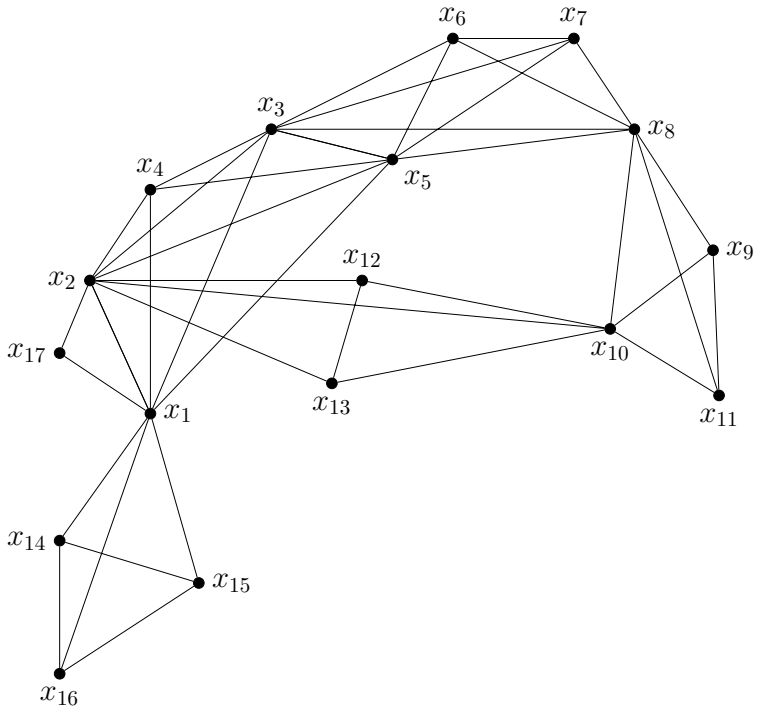


FIGURE 1.5 – La 2-section d'un hypergraphe.

Ainsi, le graphe de la figure 1.6(a) n'est pas triangulé vu qu'il contient le cycle $[x_2, x_3, x_5, x_4, x_2]$ n'ayant pas de cordes. Le graphe de la figure 1.6(b) est de son côté triangulé, l'arête $\{x_3, x_4\}$ étant une corde pour le cycle $[x_2, x_3, x_5, x_4, x_2]$. Il est à noter que la propriété d'être triangulé est une propriété héréditaire dans le sens où tout sous-graphe d'un graphe triangulé l'est aussi.

Afin de caractériser les graphes triangulés, on définit la notion de sommet *simplicial*.

Définition 22 *Un sommet x_i de G est simplicial si son voisinage $N(x_i)$ induit un sous-graphe complet de G, c'est-à-dire une clique (pas nécessairement maximale).*

Dans les deux graphes de la figure 1.6, x_1 est clairement un sommet simplicial vu que $N(x_1) = \{x_2, x_3\}$ et que $N(x_1)$ forme une clique.

Une notion primordiale des graphes triangulés est la notion d'*ordre d'élimination parfait*. Avant de la présenter formellement, nous définissons d'abord le *voisinage ultérieur d'un sommet* $N_u(x_i)$.

Définition 23 *Soit $G = (X, C)$ un graphe et soit O un ordre sur les sommets de X. Le voisinage ultérieur de x_i est $N_u(x_i) = \{x_j \in N(x_i) : x_j$ est situé après x_i dans $O\}$*

Soit $O = [x_2, x_3, x_1, x_4, x_5, x_6, x_7, x_8]$ un ordre sur les sommets du graphe de la figure 1.7(a). Pour cet ordre, $N_u(x_3) = \{x_1, x_4, x_5\}$.

Définition 24 *Soit $G = (X, C)$ un graphe et soit O un ordre sur les variables de X. O est appelé un ordre d'élimination parfait et si chaque sommet x_i est un sommet simplicial dans le sous-graphe induit $G[N_u(x_i)]$, c'est-à-dire si $G[N_u(x_i)]$ est complet.*

Par la suite, nous noterons σ un ordre d'élimination parfait. Par exemple, le graphe de la figure 1.7(a) admet un ordre d'élimination parfait $\sigma = [x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]$. Il est à

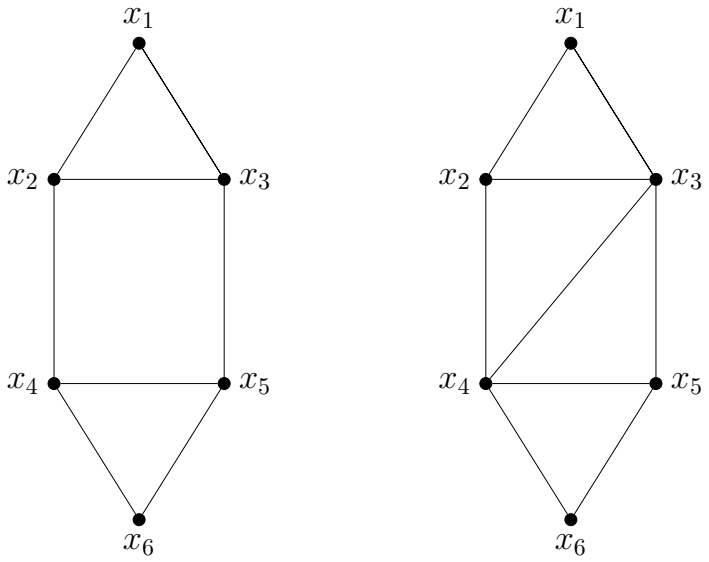


FIGURE 1.6 – Un graphe quelconque (a) un graphe triangulé (b).

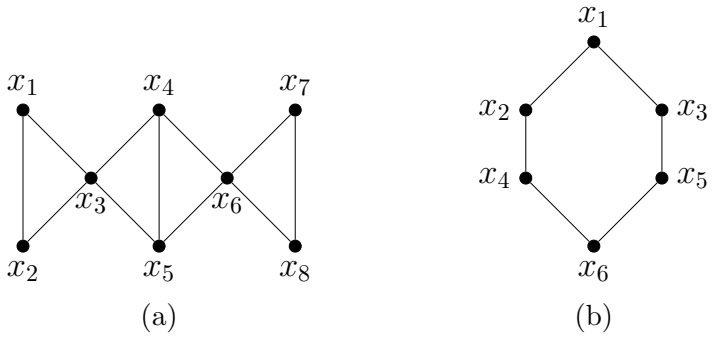


FIGURE 1.7 – Deux graphes : le premier ayant un ordre d'élimination parfait (a) et l'autre non (b).

noter que cet ordre n'est pas unique et il peut y avoir d'autres ordres d'élimination parfaits possibles. En revanche, le graphe de la figure 1.7(b) n'admet aucun ordre d'élimination parfait. En effet, il ne contient aucun sommet simplicial, empêchant ainsi de pouvoir commencer la construction d'un ordre parfait.

Nous rappelons maintenant plusieurs caractérisations des graphes triangulés fournies par Fulkerson et Gross [Fulkerson and Gross, 1965], par Dirac [Dirac, 1961] et par Lekkerkerker et Boland [Lekkeikerker and Boland, 1962].

Théorème 1 *Soit G un graphe non orienté. Les 4 propositions suivantes sont équivalentes :*

(i) G est un graphe triangulé.

(ii) G possède un ordre d'élimination parfait. En plus, tout sommet simplicial peut débuter un ordre d'élimination parfait.

(iii) Tout séparateur minimal induit un sous-graphe complet de G.

(iv) Tout sommet x_i a la propriété suivante : chaque séparateur minimal $Y \subseteq N(x_i)$ de G est une clique.

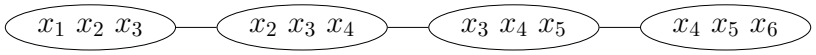


FIGURE 1.8 – Un arbre des cliques.

Ainsi, comme le graphe de la figure 1.7(a) admet un ordre d'élimination parfait, il est triangulé. Ceci n'est évidemment pas le cas du graphe de la figure 1.7(b). Nous remarquons aussi au niveau du graphe de la figure 1.7(a) que les séparateurs minimaux induisent un sous-graphe complet. Par exemple, $\{x_3\}$ est un séparateur minimal induisant trivialement un sous-graphe complet. Idem pour $\{x_4, x_5\}$. Finalement, dans le voisinage de x_4 par exemple, les séparateurs minimaux sont $\{x_3\}$ et $\{x_6\}$.

Nous évoquons aussi un lemme de [Dirac, 1961] qui nous servira plus tard dans cette thèse.

Lemme 1 *Tout graphe triangulé G possède un sommet simplicial. En plus, si G n'est pas une clique, il possède alors deux sommets simpliciaux qui ne sont pas voisins.*

Par exemple, deux sommets x_1 et x_8 sont deux sommets simpliciaux non voisins du graphe de la figure 1.7(a). En se basant sur ce résultat, Fulkerson et Gross [Fulkerson and Gross, 1965] ont proposé un algorithme pour reconnaître les graphes triangulés en utilisant conjointement la propriété d'hérédité. Il consiste à repérer à chaque étape un sommet simplicial et à l'éliminer du graphe jusqu'à ce qu'il n'y ait plus de sommets, ce qui prouve que le graphe est triangulé. Si, à une étape donnée, aucun sommet simplicial n'est trouvé, le graphe n'est alors pas triangulé. Le graphe de la figure 1.6(b) est bien triangulé vu que les sommets peuvent être éliminés les uns après les autres jusqu'à ce que il n'y ait plus de sommets. Ainsi, un ordre d'élimination parfait peut être $\sigma = [x_1, x_2, x_3, x_4, x_5, x_6]$. Ce n'est pas le cas du graphe de la figure 1.6(a), qui après l'élimination des sommets x_1 et x_6 ne possède plus de sommets simpliciaux à éliminer.

Nous évoquons maintenant un théorème essentiel des graphes triangulés démontré par Waltz [Waltz, 1972], puis par Gavril [Gavril, 1974] et par Buneman [Buneman, 1974].

Théorème 2 *Soit G un graphe quelconque. G est triangulé si et seulement s'il existe un arbre $T = (I, F)$ avec I l'ensemble de ses sommets et F l'ensemble de ses arêtes tel que chaque élément de I (appelé nœud) correspond à une clique maximale de G de sorte que chaque sous-graphe induit $T_{I_{x_i}}$ est connexe (soit un sous-arbre) où I_{x_i} désigne les cliques maximales contenant le sommet x_i de G.*

Ce théorème est d'une importance particulière parce qu'étant donné un graphe G triangulé, il nous permet de structurer le graphe G en un arbre. Nous verrons après que cette structuration, appelée *décomposition arborescente*, a permis la proposition de méthodes efficaces pour résoudre des problèmes combinatoires comme le problème de satisfaction de contraintes sur lequel nous allons nous focaliser dans la suite de cette thèse. Ces méthodes ont notamment permis de résoudre des instances relativement difficiles pour d'autres types de méthodes.

Dans [Fulkerson and Gross, 1965], Fulkerson et Gross ont constaté qu'étant donné un graphe triangulé G et un ordre d'élimination parfait σ, une clique maximale est de la forme $\{x_i\} \cup N_u(x_i)$ avec x_i le premier sommet de la clique. Par conséquent, ils ont établi le résultat suivant :

Proposition 1 *Un graphe triangulé ayant n sommets a au maximum n cliques maximales, avec égalité si et seulement si le graphe ne contient aucune arête.*

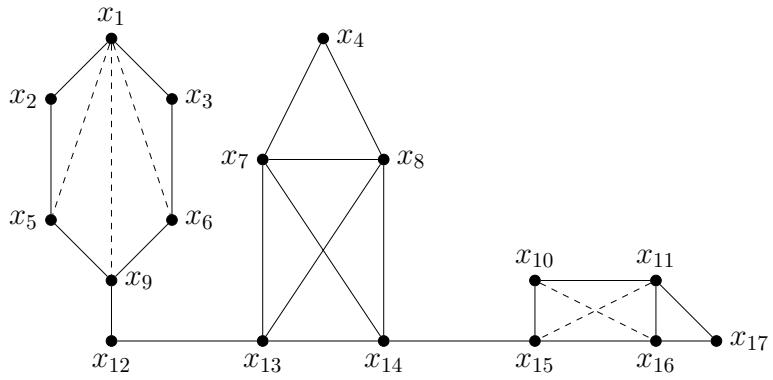


FIGURE 1.9 – Un graphe triangulé correspondant au graphe de la figure 1.2.

En se basant sur ces résultats, un algorithme de détection des cliques maximales a été proposé qui étant donné un graphe triangulé et un ordre d'élimination parfait renvoie l'ensemble de cliques maximales de G en veillant à filtrer les cliques qui ne sont pas maximales. Sa complexité est en $O(|X| + |C|)$. En utilisant un tel algorithme, l'ensemble des cliques maximales du graphe de la figure 1.6(b) peut être, par exemple, calculé. Il est composé des cliques $\{x_1, x_2, x_3\}$, $\{x_2, x_3, x_4\}$, $\{x_3, x_4, x_5\}$ et de $\{x_4, x_5, x_6\}$. En se référant au théorème 2, nous savons que l'ensemble de ces cliques maximales peut être représenté sous forme d'un arbre de sorte que pour un sommet x_i le sous-graphe de l'arbre induit par l'ensemble des cliques contenant x_i est un sous-arbre. La figure 1.8 montre l'arbre formé à partir de ces cliques.

Nous nous intéressons maintenant à la construction d'une graphe triangulé à partir d'un graphe non triangulé.

Définition 25 *Soit $G = (X, C)$ un graphe. Une triangulation de G consiste à ajouter un ensemble d'arêtes C' à G tel que le graphe obtenu $G' = (X, C \cup C')$ est triangulé.*

Par exemple, la figure 1.6(b) montre une triangulation du graphe de la figure 1.6(a) avec $C' = \{\{x_3, x_4\}\}$. Le graphe de la figure 1.9 est également le graphe résultant de la triangulation du graphe de la figure 1.2. Les arêtes ajoutées sont représentées en tirets et sont : $C' = \{\{x_1, x_5\}, \{x_1, x_6\}, \{x_1, x_9\}, \{x_{10}, x_{16}\}, \{x_{11}, x_{15}\}\}$.

Nous définissons maintenant la notion de *triangulation minimale*.

Définition 26 *Soit $G = (X, C)$ un graphe. La triangulation ajoutant à G l'ensemble d'arêtes C' est minimale si le graphe $G' = (X, C \cup C')$ est non triangulé pour tout sous-ensemble $C'_{\subset}$ strictement inclus dans C'.*

Par exemple, le graphe de la figure 1.9 ne présente pas une triangulation minimale vu que le graphe $(X, C \cup C'_{\subset})$ tel que $C'_{\subset} = \{\{x_1, x_5\}, \{x_1, x_6\}, \{x_1, x_9\}, \{x_{10}, x_{16}\}\}$ est également une triangulation de G tout en ayant $C'_{\subset} \subset C'$ (graphe de la figure 1.10).

Nous définissons également la notion de *triangulation minimum*.

Définition 27 *Soit $G = (X, C)$ un graphe. La triangulation ajoutant à G l'ensemble d'arêtes C' est minimum s'il n'existe aucun ensemble C'' tel que $|C''| < |C'|$ et $G'' = (X, C \cup C'')$ est un graphe triangulé.*

Le graphe de la figure 1.10 est également une triangulation minimum vu que nous ne pouvons pas avoir une triangulation du graphe en ajoutant moins d'arêtes. Notons que si la triangulation n'est pas minimale, elle n'est évidemment pas minimum. Cependant, une triangulation peut être minimale sans être minimum.

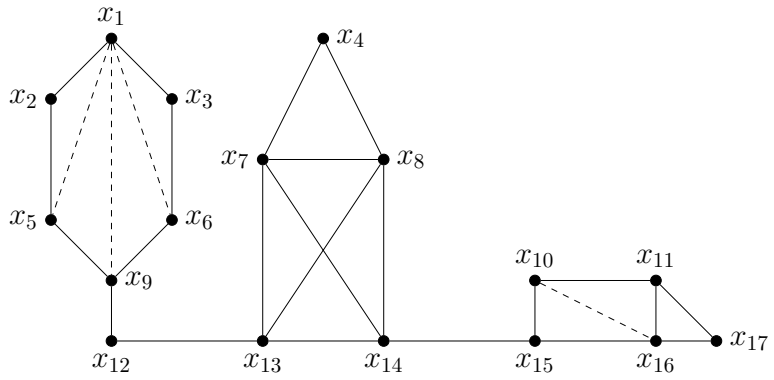


FIGURE 1.10 – Une triangulation minimale et minimum du graphe de la figure 1.2.

1.3 Décompositions des (hyper)graphes

Dans cette partie, nous nous intéressons aux différentes notions de décomposition des (hyper)graphes et aux méthodes permettant de les calculer.

La connaissance peut être représentée par un modèle graphique [Lauritzen, 1996; Ruppert, 2001; Jordan, 2004; Nielsen and Jensen, 2009; Darwiche, 2009; Koller and Friedman, 2009; Dechter, 2013; Pearl, 2014]. Dans un modèle graphique, un sommet correspond à une variable et une arête représente des dépendances entre différentes variables. Un graphe est une façon intuitive de représenter les relations entre différentes variables. Il permet également de relever les indépendances pouvant exister entre les variables. Les classes les plus connues des modèles graphiques sont les réseaux Bayésiens et les réseaux de Markov. Ils permettent de représenter d'une façon compacte des problèmes réels et complexes. Leur notoriété découle de leur flexibilité, de leur puissance et aussi de l'augmentation de la capacité à apprendre efficacement à travers ces modèles et à réaliser des inférences pour de grands graphes. Selon la question à laquelle l'utilisation des modèles graphiques tente de répondre, la complexité varie de manière conséquente.

Heureusement, plusieurs types d'inférence peuvent être réalisés efficacement pour une classe très importante des modèles graphiques. En revanche, pour un très grand nombre de modèles, l'inférence est non traitable. Les modèles graphiques ayant une *tree-width* [Robertson and Seymour, 1986] bornée par une constante permettent de répondre à certaines questions en temps polynomial. La tree-width est un paramètre structurel rattaché au graphe. En particulier, un graphe acyclique a une tree-width de 1. Lorsque le graphe n'est pas acyclique, il se peut qu'il contient peu de cycles, des cycles de longueur faible ou qui peuvent être transformés en structures acycliques par des opérations simples, c'est-à-dire une structure qui n'est pas très éloignée d'une structure acyclique. Dans ces cas, la tree-width serait ainsi relativement petite.

Au-delà, les notions de décomposition qui font l'objet de cette partie définissent toutes une notion de *largeur* qui peut être considérée comme une *mesure de la cyclicité*. Ainsi nous nous intéressons aux modèles ayant un graphe de largeur bornée par k, pour un k fixé, vu leur rapprochement d'une structure acyclique. L'intérêt d'une décomposition se définit alors par rapport à la borne de complexité théorique qui en découle, exprimée en fonction de sa largeur. Nous disposons actuellement d'un grand nombre de décompositions possibles définissant différemment la notion de *largeur*. Nous détaillons dans cette section certaines de ces décompositions.

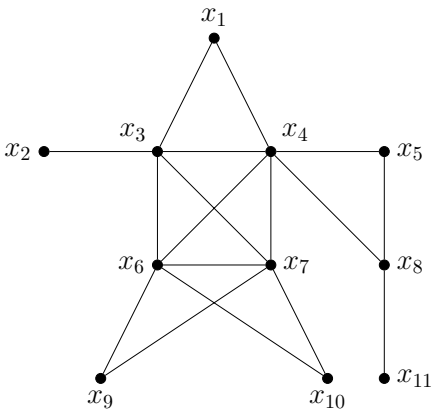


FIGURE 1.11 – Un graphe.

1.3.1 Quelques décompositions existantes

1.3.1.1 Décomposition arborescente

La *décomposition arborescente* [Robertson and Seymour, 1986] consiste en un recouvrement acyclique du graphe par des ensembles de sommets appelés *clusters* (ou *sacs* dans la communauté des mathématiques). Les sommets du graphe correspondant sont alors regroupés en clusters de sorte que ces clusters forment un arbre. Nous présentons maintenant la définition formelle d'une décomposition arborescente.

Définition 28 *Une* décomposition arborescente *d'un graphe $G = (X, C)$ est un couple (E, T) où $T = (I, F)$ est un arbre (I est un ensemble de nœuds et F un ensemble d'arêtes) et $E = \{E_i : i \in I\}$ une famille de sous-ensembles de X, telle que chaque sous-ensemble E_i correspond à un nœud i de T et vérifie :*

(i) $\cup_{i \in I} E_i = X$,

(ii) pour chaque arête $\{x, y\} \in C$, il existe $i \in I$ avec $\{x, y\} \subseteq E_i$, et

(iii) pour tout $i, j, k \in I$, si k est sur le chemin entre i et j dans T, alors $E_i \cap E_j \subseteq E_k$

La largeur d'une décomposition arborescente w^+ est égale à $max_{i \in I}|E_i| - 1$. La largeur arborescente dite tree-width *w de G est la largeur minimale pour toutes les décompositions arborescentes de G. Une décomposition arborescente est dite* optimale *si sa largeur w^+ est égale à la largeur arborescente w.*

Un graphe possédant au moins une arête est acyclique si sa tree-width est égale à 1.

La figure 1.11 présente un graphe et la figure 1.12 montre une de ses décompositions arborescentes optimales. Elle est composée de 7 clusters avec $E = \{E_0, E_1, \ldots, E_6\}$. Nous pouvons vérifier que chaque sommet est présent dans au moins un cluster ainsi que toutes les arêtes. La troisième condition peut être facilement vérifiée en constatant que pour tout sommet x_i, les nœuds de T contenant x_i forment un sous-arbre. Ainsi, il est évident que si deux clusters partagent un sous-ensemble non vide de sommets en commun, ces derniers sont présents dans tout cluster situé sur le chemin menant l'un à l'autre. Le plus grand cluster de la décomposition contient 4 sommets, c'est-à-dire $w^+ = 3$. La décomposition de la figure 1.12 est dite optimale puisqu'il n'existe aucune autre décomposition dont la largeur est strictement inférieure à 3. Ainsi, $w = 3$.

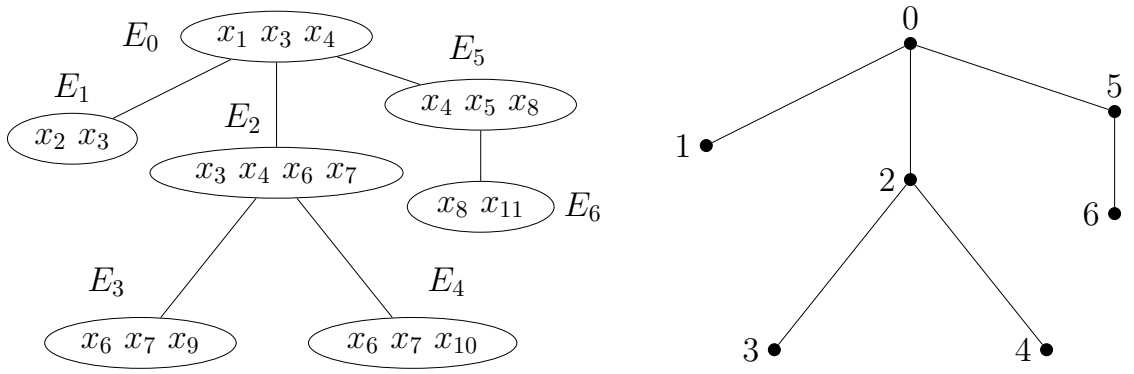


FIGURE 1.12 – Une décomposition arborescente optimale du graphe de la figure 1.11.

Nous pouvons associer à une décomposition arborescente un *cluster racine* noté E_r. Sur l'exemple de la figure 1.12, supposons que $E_r = E_0$. L'enracinement de la décomposition lui offre une orientation permettant de déduire les *fils d'un cluster*. L'ensemble des fils d'un cluster E_i de E (noté $Fils(E_i)$) est l'ensemble de clusters correspondants aux nœuds fils de i dans T. Ainsi, $Fils(E_2) = \{E_3, E_4\}$. E_2 est aussi appelé le cluster parent de E_3 et de E_4. Nous notons $E_{p(i)}$ le *cluster parent* du cluster E_i. Un sommet appartenant à E_i et n'appartenant pas à $E_{p(i)}$ est appelé *sommet propre* de E_i. La *descendance* d'un cluster E_i peut être définie comme $Desc(E_i) = \{E_i\} \cup_{E_j \in Fils(E_i)} Desc(E_j)$. Tout cluster $E_j \in Desc(E_i)$ est un descendant de E_i. Sur l'exemple de la figure 1.12, $Desc(E_2) = \{E_2, E_3, E_4\}$. L'intersection entre deux clusters E_i et E_j sera appelée ici un *séparateur*. Par exemple, $E_2 \cap E_3 = \{x_6, x_7\}$. La taille du plus grand séparateur de la décomposition est notée s. D'où, $s = 2$ sur notre exemple. Nous rappelons que la taille maximum des clusters donne une indication sur le niveau de cyclicité du graphe. Ainsi si w est « petit » par rapport à n, le graphe a une structure proche d'une structure acyclique. En particulier, w est égale à 1 si le graphe est acyclique et il possède au moins une arête. Au contraire, plus w est proche de n, plus le graphe a une structure éloignée d'une structure acyclique. Nous citons à l'appui l'exemple du graphe complet dont la décomposition n'admet qu'un seul cluster et est dont la largeur w est égale à $n - 1$.

La notion de la décomposition arborescente est uniquement définie pour des graphes, mais elle peut être considérée pour des hypergraphes en exploitant leur graphe primal. Par souci de clarté, nous notons e le nombre d'arêtes du graphe considéré pour le calcul de la décomposition arborescente. Ainsi, $e = m$ si le graphe considéré est à l'origine un graphe et $e > m$ s'il est à l'origine un hypergraphe possédant au moins une hyperarête d'arité strictement supérieure à 2.

Finalement, lorsque l'arbre T n'est constitué que d'une chaîne, on parle de décomposition linéaire (path-decomposition) et de largeur linéaire (path-width) [Robertson and Seymour, 1983].

Un lemme important pour cette thèse est énoncé par Gavril dans [Gavril, 1974] est :

Lemme 2 *Étant donnés un graphe G et une de ses décompositions arborescentes $(E, T = (I, F))$. Pour toute clique $X_\subseteq$ de G, il existe $p \in I$ tel que $X_\subseteq \subseteq E_p$*

En d'autres termes, une clique de G est forcément incluse dans au moins un cluster de E. Par exemple, nous pouvons facilement voir que toutes les cliques du graphe de la figure 1.11 sont présentes dans un cluster de la décomposition de la figure 1.12 comme $\{x_3, x_4, x_6, x_7\}$ et $\{x_4, x_5, x_8\}$. Ce lemme sera utilisé à plusieurs reprises tout au long de cette thèse pour démontrer certains théorèmes.

1.3.1.2 Décomposition en hyperarbre (fractionnaire)

Dans cette partie, nous nous focalisons sur la décomposition en hyperarbre [Gottlob et al., 1999] et sur la décomposition en hyperarbre fractionnaire [Grohe and Marx, 2006, 2014]. La décomposition en hyperarbre généralise la décomposition arborescente alors que la décomposition en hyperarbre fractionnaire généralise la décomposition en hyperarbre.

Décomposition en hyperarbre [Gottlob et al., 1999] La notion d'hyperarbre a été définie dans [Gottlob et al., 1999] dans le contexte des bases de données. Cette notion est adaptée aux hypergraphes dans [Gottlob et al., 2000]. Un *hyperarbre* d'un hypergraphe $H = (X, C)$ est un triplet (T, χ, λ) avec $T = (I, F)$ un arbre enraciné, et χ et λ deux fonctions qui associent à chaque sommet p de I deux ensembles $\chi(p) \subseteq X$ et $\lambda(p) \subseteq C$. Si $T' = (I', F')$ est un sous-arbre de T, nous définissons $\chi(T') = \cup_{v \in I'} \chi(v)$. En plus, pour tout $p \in I$, T_p représente le sous-arbre de T enraciné en p. En outre, si $C_\subseteq \subseteq C$, $sommets(C_\subseteq) = \cup_{c_i \in C_\subseteq} c_i$.

Définition 29 *Une décomposition en hyperarbre d'un hypergraphe H est un hyperarbre (T, χ, λ) de H qui satisfait les conditions suivantes :*

(i) pour chaque hyperarête $c_i \in C$, il existe $p \in I$ tel que $c_i \subseteq \chi(p)$ (autrement dit p couvre c_i),

(ii) pour chaque sommet $x_i \in X$, l'ensemble $\{p \in I : x_i \in \chi(p)\}$ induit un sous-arbre de T,

(iii) pour tout $p \in I$, $\chi(p) \subseteq sommets(\lambda(p))$,

(iv) pour tout $p \in I, sommets(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$.

Notons que la condition (iv) est en réalité une égalité, vu que la condition (iii) implique l'inclusion inverse. Une hyperarête $c_i \in C$ est *fortement recouverte* par la décomposition en hyperarbre s'il existe $p \in I$ tel que $c_i \subseteq \chi(p)$ et $c_i \in \lambda(p)$. Une décomposition en hyperarbre est dite *complète* si chaque hyperarête est fortement recouverte. La largeur d'une décomposition en hyperarbre (T, χ, λ) est $max_{p \in I}|\lambda(p)|$. La *hypertree-width* de H, notée hw, est la largeur minimale pour toutes les décompositions en hyperarbre de H. Une décomposition en hyperarbre de H est optimale si sa largeur est égale à hw. Les hypergraphes acycliques sont exactement ceux dont la largeur est 1. La figure 1.13 montre une décomposition en hyperarbre du graphe de la figure 1.2. Chaque cluster de la décomposition correspond à un nœud p de l'arbre T et montre les deux ensembles $\chi(p)$ et $\lambda(p)$. La largeur de la décomposition est 2 du fait que $max_{p \in I}|\lambda(p)| = 2$. En plus, nous pouvons facilement vérifier que cette décomposition est complète. Nous pouvons constater que nous sommes ainsi capables d'exploiter une décomposition qui a une largeur plus intéressante que celle de la décomposition arborescente qui d'ailleurs peut avoir au mieux une largeur 3 vu que le graphe contient la clique $\{x_7, x_8, x_{13}, x_{14}\}$. Il est à noter qu'une décomposition en hyperarbre est une décomposition en hyperarbre généralisée qui satisfait en plus la condition (iv). La largeur hyperarborescente généralisée de H, notée ghw, est la largeur minimale pour toutes les décompositions généralisées en hyperarbre de H. En outre, il a été prouvé dans [Adler et al., 2007] que $ghw \leq hw \leq 3ghw + 1$.

Ultérieurement une nouvelle décomposition, appelée la *décomposition en hyperarbre fractionnaire*, a été proposée [Grohe and Marx, 2006].

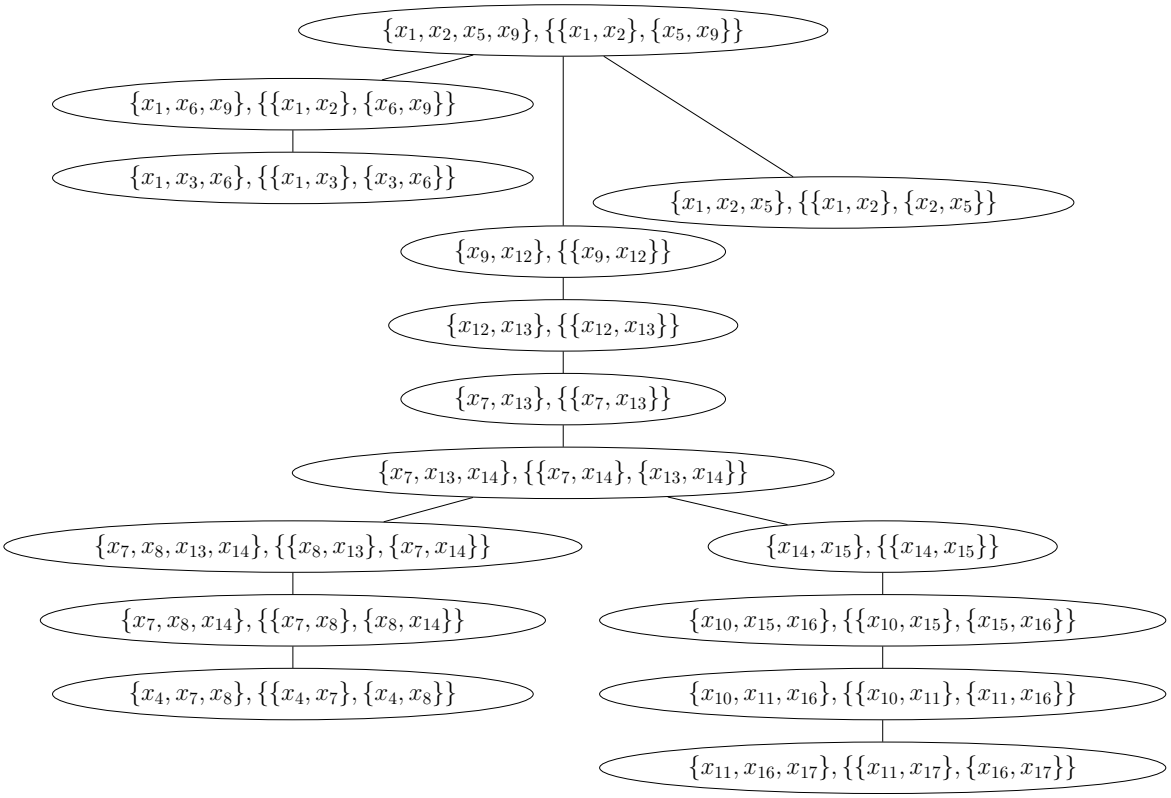


FIGURE 1.13 – Une décomposition en hyperarbre.

Décomposition en hyperarbre fractionnaire [Grohe and Marx, 2006, 2014] La définition de la *couverture fractionnaire d'arêtes* combinée avec la définition de la décomposition en hyperarbre a permis de définir la décomposition en hyperarbre fractionnaire.

Définition 30 *La couverture d'arêtes fractionnaire d'un hypergraphe H est une fonction $\varphi : C \to [0, \infty[$ qui associe à chaque hyperarête un nombre dit* poids de l'hyperarête *tel que pour tout sommet x_i, la somme des poids des hyperarêtes dans lesquelles il apparaît est supérieure ou égale à 1. Le nombre $\Sigma_{c \in C} \varphi(c)$ est le poids de φ, noté poids(φ). Le* nombre couverture d'arêtes fractionnaire $\rho^*(H)$ *de H est le minimum des poids de toutes les couvertures d'arêtes fractionnaires.*

Il a été constaté dans [Grohe and Marx, 2006] qu'il est facile de construire des classes d'hypergraphes ayant un nombre couverture d'arêtes fractionnaire borné sans que la largeur hyperarborescente ne le soit. C'est ainsi qu'il a été proposé de généraliser ces deux concepts et de se baser sur la notion de la *largeur hyperarborescente fractionnaire*. Il est à noter que le nombre couverture d'arêtes fractionnaire et la hypertree-width sont incomparables du fait que certains hypergraphes peuvent avoir un nombre couverture d'arêtes fractionnaire borné et une hypertree-width non bornée, tandis que d'autres peuvent présenter le cas inverse. Nous définissons d'abord l'ensemble $B(\varphi)$ pour une fonction de poids φ : $B(\varphi) = \{x \in X : \Sigma_{c \in C : x \in C} \varphi(c) \geq 1\}$. $B(\varphi)$ est également appelé l'ensemble des sommets de X bloqués par φ.

Définition 31 *Soit H un hypergraphe. Une* décomposition en hyperarbre fractionnaire *est un triplet (T, χ, λ) avec $T = (I, F)$ un arbre enraciné, et χ et λ deux fonctions*

qui associent à chaque sommet p de I, $\chi(p) \subseteq X$ et $\lambda(p)$ une fonction de poids. La décomposition satisfait les conditions suivantes :

(i) (χ, T) correspond à une décomposition arborescente de H,

(ii) pour tout $p \in I$, nous avons $\chi(p) \subseteq B(\lambda(p))$.

La largeur de (T, χ, λ) est max$\{poids(\lambda(p)) : p \in I\}$. La largeur hyperarborescente fractionnaire de H, notée fhw, est le minimum parmi les largeurs des décompositions en hyperarbre fractionnaire.

Clairement, pour tout hypergraphe H, nous avons les inéquations $fhw \leq \rho^*(H)$ et $fhw \leq hw$. Il est à noter que la décomposition en hyperarbre généralisée et celle non généralisée sont équivalentes vis-à-vis de la définition de la décomposition en hyperarbre fractionnaire. Une décomposition en hyperarbre fractionnaire du graphe de la figure 1.2 peut être déduite de la décomposition en hyperarbre de la figure 1.13 en créant pour chaque sommet $p \in T$ une fonction de poids qui associe à chaque hyperarête de $\lambda(p)$ de la décomposition en hyperarbre un poids égal à 1. La largeur de la décomposition est 2 du fait que $\max\{poids(\lambda(p)) : p \in I\} = 2$.

1.3.1.3 Autres décompositions

L'état de l'art sur les méthodes de décomposition contient de nombreuses autres notions de décompositions, chacune définissant une notion de largeur différente. Sans prétendre à l'exhaustivité, nous pouvons citer les *composantes biconnexes* [Even, 1979], la *décomposition hinge* [Gyssens and Paredaens, 1982; Gyssens et al., 1994], l'*ensemble et hyper-ensemble coupe-cycle* [Dechter, 1992] ou les *décompositions encadrées* [Cohen et al., 2008].

Certaines décompositions définissent une largeur qui a été comparée à la tree-width. Nous mentionnons par la suite quelques-unes.

La notion de la *branch-width* a été introduite dans [Robertson and Seymour, 1986] et est notée $\beta(G)$. Il s'agit d'une notion qui est fortement liée à celle de la tree-width. D'ailleurs, ces deux notions sont comparées dans [Robertson and Seymour, 1986] permettant d'établir l'inéquation : $\max(\beta(G), r) \leq w+1 \leq \max(\lfloor (3/2)\beta(G) \rfloor, r, 1)$. Son calcul a été abordé dans plusieurs papiers. Il est NP-difficile pour des graphes en général. Cependant, dans [Seymour and Thomas, 1994], il a été montré qu'il peut être réalisé en temps polynomial pour des graphes planaires (graphes pouvant être représentés sur un plan sans qu'aucune arête n'en croise une autre).

Une deuxième mesure relative au graphe est la notion de *clique-width* (cwd(G)) introduite dans [Courcelle and Olariu, 2000]. La décomposition d'un graphe G peut être perçue ici comme un terme fini composé d'opérations définies pour les graphes. La complexité du graphe est alors mesurée par rapport à un paramètre entier k relatif à ces opérations. Ainsi, un graphe a une complexité qui augmente avec l'augmentation de k. Une motivation mentionnée pour l'introduction de ce paramètre est, comme pour la tree-width, l'existence de plusieurs problèmes NP-complets ayant des algorithmes linéaires pour des graphes de clique-width bornée par k. Cette notion est comparable à la notion de la tree-width. En effet, dans [Courcelle and Olariu, 2000], il a été démontré que $cwd(G) \leq 2^{w+1} + 1$. Cependant, il est impossible d'avoir une relation telle que : $w \leq f(cwd(G))$ valide pour tous les graphes. Par exemple, les graphes complets ont une largeur arborescente non bornée $n-1$ tandis que leur clique-width est au plus 2. Il est à noter que ce genre d'inégalité peut être obtenu pour des graphes appartenant à des classes particulières comme les graphes

planaires. On en déduit de cette inégalité qu'avoir une clique-width bornée est plus général que le fait d'avoir une tree-width bornée. Dans [Oum and Seymour, 2006], un algorithme en $O(n^9.\log n)$ a été proposé qui, pour un entier k fixé, et pour un graphe G donné en entrée, calcule une décomposition de largeur au plus $2^{3k+2}-1$ ou met en évidence que le graphe a une clique-width d'au moins $k+1$. Si avant le fait de résoudre un problème polynomialement était conditionné par l'existence d'une décomposition de clique-width au plus k et que cette dernière soit donnée en entrée, la deuxième condition n'est désormais plus nécessaire. Pour démontrer l'algorithme en question, les auteurs dans [Oum and Seymour, 2006] se servent de la définition d'un nouveau paramètre, appelé la *rank-width* et noté $rwd(G)$.

La *rank-width* est un paramètre associé au graphe et lié à la notion de clique-width tout en étant plus traitable. Les auteurs de [Oum and Seymour, 2006] ont montré que la rank-width et la clique-width sont dans un sens approximativement égales. Plus précisément, les deux notions sont liées par l'inégalité suivante : $\log_2(cwd(G)+1)-1 \leq rwd(G) \leq cwd(G)$. Ainsi, l'ensemble des graphes de rank-width bornée a également une clique-width bornée et vice versa.

La liste des décompositions existantes est bien loin d'être terminée. Nous pouvons ainsi citer d'autres décompositions plus récentes données par le paramètre graphique auquel elles sont associées comme *boolean-width* [Bui-Xuan et al., 2011], *matching-width* [Jeong et al., 2015], *modular-width* [Gajarskỳ et al., 2013], *treecut-width* [Wollan, 2015], *tree-depth* [Nešetřil and Mendez, 2012] ou *connected tree-width* [Müller, 2012; Diestel and Müller, 2017; Hamann and Weißauer, 2016].

1.3.1.4 Bilan

Parmi les différentes décompositions évoquées, nous nous focalisons dans cette thèse sur la décomposition arborescente. Nous exploitons la décomposition d'un graphe G ou du graphe primal d'un hypergraphe H. La décomposition arborescente a fait l'objet de nombreuses études vu que plusieurs problèmes de graphe qui sont NP-difficiles peuvent être résolus d'une façon polynomiale si le graphe en question admet une tree-width bornée. Nous devons ce résultat à Courcelle qui en 1990, a montré que tout problème exprimable en logique monadique du second ordre (MSO) peut être résolu en temps linéaire pour des graphes de tree-width bornée par une constante [Courcelle, 1990]. La logique MSO est une extension de la logique du premier ordre qui permet la quantification des ensembles. Parmi ces problèmes nous citons quelques-uns des plus célèbres comme le problème du transversal minimum (VERTEX COVER), problème du stable maximum (INDEPENDENT SET) ou la k-coloration (k-COLORABILITY) pour un entier k fixé [Cook, 1971; Karp, 1972]. Le théorème de Courcelle a été généralisé ultérieurement par Arnborg, Lagergren et Seese dans [Arnborg et al., 1991] aux EMSO (extension des MSO). L'intérêt de la tree-width est la raison essentielle pour laquelle elle admet de nombreuses applications en théorie de *complexité paramétrée*. La complexité paramétrée est une approche qui explore si un problème difficile peut être résolu dans un temps proche d'un temps polynomial pour certaines instances. Ces instances peuvent être celles ayant une tree-width relativement « petite » si le problème peut être efficacement résolu pour des graphes dont la structure se rapproche d'un arbre. Cela nous mène à la notion d'algorithme FPT en un paramètre k (pour *traitable avec paramètre fixé*). Un algorithme est FPT en k s'il permet de résoudre un problème paramétré par k en temps proportionnel à $f(k).poly(|I|)$ avec $poly(|I|)$ une fonction polynomiale en fonction de la taille des données du problème et f une fonction quelconque. Pour un problème exprimable dans la logique MSO du théorème de Courcelle,

il existe un algorithme FPT paramétré par la tree-width. Lorsque la tree-width est « petite », un tel algorithme peut être considérablement efficace en dépit de la NP-difficulté du problème. Pour plus d'informations concernant la complexité paramétrée, nous renvoyons le lecteur aux travaux comme [Fellows and Downey, 1999; Flum and Grohe, 2006; Niedermeier, 2006]. Finalement, dans le cadre de cette thèse, nous exploitons l'existence des algorithmes FPT paramétrés par la tree-width pour la résolution du problème de satisfaction de contraintes (CSP) [Montanari, 1974], du problème du comptage (#CSP) et du problème de satisfaction de contraintes pondérées (WCSP) [Freuder and Wallace, 1992; Schiex, 2000; Larrosa, 2002] pour lesquelles nous donnerons des détails plus amples dans le chapitre suivant.

1.3.2 Calcul de la largeur et de la décomposition arborescente

Notre intérêt pour la tree-width met en avant le problème de son calcul. Cette partie se concentre alors sur son calcul ainsi que celui de la décomposition arborescente. Le problème TREE-WIDTH consiste à dire si la largeur arborescente d'un graphe est au plus k pour un entier k fixé. Ce problème est NP-complet [Arnborg et al., 1987]. La version optimisation de ce problème, c'est-à-dire pouvoir déterminer le k minimum, est NP-difficile. Le fait que ce problème soit NP-difficile, laisse peu d'espoir pour pouvoir trouver un algorithme qui détermine la tree-width d'un graphe quelconque en temps polynomial. C'est pourquoi les algorithmes exacts connus dans la littérature sont exponentiels en temps. Par conséquent, d'autres classes d'algorithmes de calcul de tree-width sont apparues et ont fait l'objet de beaucoup de travaux de recherche. Aussi, nous détaillerons ces classes dans cette partie.

La partie 1.2.2 rappelant les notions préliminaires sur les graphes triangulés est d'une grande importance pour la compréhension de cette partie. En effet, il existe une relation indissociable entre les graphes triangulés et la notion de décomposition arborescente. D'ailleurs, la manière habituelle pour calculer une décomposition du graphe G consiste à procéder d'abord à une triangulation de G.

Nous évoquons tout d'abord le théorème essentiel de [Robertson and Seymour, 1986] justifiant ce propos.

Théorème 3 *Soit G un graphe et k un entier strictement positif. La* tree-width *de G est au plus $k-1$ si et seulement si la triangulation de G permet d'obtenir G' telle que la taille maximale des cliques maximales de G' est au plus k.*

Selon ce théorème, une possibilité pour calculer la largeur arborescente de G, consiste à calculer une triangulation G' telle que la taille maximale des cliques maximales est minimum. La tree-width est alors la taille maximale des cliques moins 1.

En général, un graphe peut admettre plusieurs triangulations. Le nombre d'arêtes ainsi que l'ensemble d'arêtes ajoutées peut varier d'une triangulation à l'autre. Une manière simple de construire un graphe triangulé G' à partir de G consiste à prendre un ordre O sur les sommets de X et à le rendre un ordre parfait en ajoutant les arêtes nécessaires. Le graphe triangulé résultant est ainsi appelé G'_O. Lorsqu'il n'y a pas d'ambiguïté, il sera simplement noté G'. Pour y parvenir, il suffit que chaque sommet x de O devienne un sommet simplicial en complétant le sous-graphe correspondant à son voisinage ultérieur, c'est-à-dire $G[N_u(x)]$. Cela est illustré dans l'algorithme 1.1. Les lignes 4-6 illustrent la complétion du voisinage ultérieur d'un sommet x. Nous notons e' le nombre d'arêtes de G' si le graphe G fait l'objet d'une triangulation. Sa complexité est en $O(n + e')$. D'ailleurs, avant que le lien entre l'ordre d'élimination et les graphes triangulés n'ait été établi, une analogie entre les graphes et l'élimination gaussienne a été remarquée par Parter en 1961 à travers « le jeu d'élimination » [Parter, 1961]. Cet algorithme en plus de ce qui est

Algorithme 1.1 : Trianguler (G, O)

Entrées : Un graphe $G = (X, C)$, un ordre O sur X
Sorties : Le graphe triangulé G'_O

```
1 G'_O ← G
2 pour i ← 1 à n − 1 faire
      /* Soit x le i-ème sommet de l'ordre O                    */
3     x ← O^{-1}(i)
      /* Compléter le sous-graphe induit par G'_O[N_u(x)]       */
4     pour chaque paire de voisins y,z de x de G'_O avec y ≠ z et O(y) > O(x) et
       O(z) > O(x) faire
5         si y et z ne sont pas voisins dans G'_O alors
6             Ajouter {y, z} dans G'_O
7 retourner G'_O
```

décrit dans l'algorithme 1.1 élimine le sommet x une fois son voisinage ultérieur complété. Il travaille ainsi sur un graphe temporaire et ajoute à la fin toutes les arêtes ajoutées au graphe temporaire au graphe G. L'élimination est valide puisque nous savons que le sommet x ne sera plus lié à d'autres sommets après la complétion de son voisinage ultérieur. Certes, la triangulation est différente selon l'ordre O choisi au départ. Le choix entre les ordres possibles O dépend des caractéristiques voulues par la triangulation. Un des premiers objectifs consistait à ajouter le plus petit nombre d'arêtes (problème *minimum fill-in*) [Yannakakis, 1981]. Ce problème est, à l'instar du problème TREE-WIDTH, un problème NP-difficile [Arnborg et al., 1987]. La qualité de la triangulation dépend de l'usage ultérieur du graphe triangulé.

Au-delà, une fois le graphe G' calculé, les cliques maximales de ce graphe sont identifiées en examinant le voisinage ultérieur de chaque sommet. Nous pouvons ainsi noter que la largeur de la décomposition finalement calculée est le nombre maximum de voisins ultérieurs d'un sommet par rapport à l'ordre O exploité.

Calcul de la décomposition arborescente Nous pouvons également construire une décomposition arborescente de G. Une fois les cliques du graphe identifiées, chacune correspondra à un cluster de la décomposition. Ces clusters sont finalement liés de façon à respecter les conditions d'une décomposition arborescente comme dans [Jarnìk, 1930] grâce à l'algorithme de Jarnìk. Chaque sommet est censé être représenté dans au moins un cluster. Une arête doit être également présente dans au moins un cluster. Finalement, si le même sommet est inclus dans deux clusters différents, il devra l'être dans tous les clusters situés sur le chemin entre ces deux clusters. Un nœud est alors associé à chaque cluster. L'ensemble des nœuds obtenus est I. Deux nœuds sont liés s'ils correspondent à deux clusters dont l'intersection est non vide. L'algorithme de Jarnìk calcule ensuite un arbre recouvrant T.

Vu la NP-difficulté du problème du calcul de la tree-width et du minimum fill-in, les algorithmes peuvent être classés en différentes catégories selon la qualité de la borne fournie pour la largeur arborescente. On distingue alors 4 catégories : les algorithmes exacts, les algorithmes d'approximation avec garanties, les algorithmes de triangulation minimaux et les algorithmes heuristiques.

1.3.2.1 Algorithmes exacts

Ces algorithmes visent à calculer la tree-width d'un graphe G (sa valeur exacte). En raison de la NP-difficulté du problème, nous pouvons se poser des questions sur la performance des algorithmes existants. En effet, il n'existe pas d'algorithme connu pour pouvoir résoudre ce problème en temps polynomial pour des graphes quelconques.

Dans [Arnborg et al., 1987], les auteurs proposent un premier algorithme qui étant donné un entier k fixé permet de tester en $O(n^{k+2})$ si le graphe donné en entrée a une largeur d'au plus k. L'algorithme a alors une complexité en $O^*(2^n)$ [1]. Dans [Bodlaender, 1996], Bodlaender propose un algorithme linéaire qui étant donnés une constante k et un graphe G détermine si la largeur arborescente de G est au plus k et trouve dans ce cas une décomposition arborescente de G d'une largeur au plus k. Cependant, il est à noter que la constante cachée par la notation O est doublement exponentielle en k^2. Dans [Bouchitté and Todinca, 2001], les auteurs ont établi qu'étant donnés un graphe G et une liste des cliques potentielles maximales (cliques maximales dans une triangulation minimale du graphe), la tree-width et le minimum fill-in d'un graphe G peuvent être déterminés en un temps polynomial en fonction de la taille de G et du nombre de cliques potentielles maximales. En plus, dans [Bouchitté and Todinca, 2002], les auteurs démontrent que la liste des cliques potentielles maximales peut être calculée en temps polynomial en fonction de la taille de G et du nombre minimal des séparateurs. Ainsi, la tree-width et le minimum fill-in d'un graphe sont polynomialement traitables en fonction de la taille de G et du nombre de séparateurs minimaux. En se basant sur ces résultats, les auteurs dans [Fomin et al., 2004] ont démontré que le calcul de la tree-width et du minimum fill-in peut être fait en $O(1,9601^n)$. Les résultats ont été ultérieurement améliorés dans [Villanger, 2006] pour atteindre une borne de complexité en $O(1,8899^n)$. Bien que ces algorithmes offrent une meilleure borne de complexité, ils n'ont pas montré leur intérêt pratique. En effet, ils sont basés sur la notion des cliques maximales potentielles qui sont difficilement calculables en pratique. Dans [Bodlaender et al., 2006], les auteurs proposent une implémentation optimisée d'un ancien algorithme de programmation dynamique introduit dans [Held and Karp, 1962] pour le problème du voyageur du commerce. Ils exploitent implicitement la triangulation et obtiennent une complexité temporelle en $O^*(2^n)$ et une complexité spatiale en $O^*(2^n)$. En pratique, cet algorithme (appelé DP) peut être utilisé pour de petits graphes (au plus 50 sommets). Au-delà, les expérimentations ont montré que l'espace utilisé par l'algorithme est trop important vis-à-vis de la taille des entrées. Les auteurs ont ainsi proposé deux algorithmes ayant des complexités en temps en $O^*(4^n)$ et en $O^*(2,9512^n)$ tout en ayant des complexités polynomiales en espace. Les expérimentations conduites ultérieurement dans [Bodlaender et al., 2012] affirment que les algorithmes en espace polynomial sont très lents même pour des petits graphes. Une amélioration théorique majeure de ces deux algorithmes est présentée dans [Fomin and Villanger, 2012] qui propose un algorithme qui calcule la tree-width de G en $O(1,7549^n)$ en utilisant un espace exponentiel et un autre algorithme qui a une complexité en temps en $O(2,6151^n)$ en utilisant un espace polynomial.

La tree-width peut être calculée en pratique en utilisant des algorithmes de type *séparation et évaluation* (*brand and bound*). Dans ce contexte, la méthode *QuickBB* a été proposée dans [Gogate and Dechter, 2004]. Elle construit un ordre d'élimination parfait sur les sommets d'un graphe G. L'algorithme utilise des techniques d'élagage de branches et de propagation différentes. Entre autres, cette méthode a été comparée à la méthode *QuickTree* introduite dans [Shoikhet and Geiger, 1997]. Selon ses auteurs, QuickTree est

1. La notation O^* supprime les facteurs polynomiaux en n.

le premier algorithme à calculer une triangulation optimale dans un temps raisonnable. Elle est basée sur la notion des séparateurs minimaux. Notons que QuickTree est une modification de l'algorithme de [Arnborg et al., 1987] puisqu'ils sont tous les deux basés sur l'idée de l'utilisation de la programmation dynamique pour construire des triangulations des « grandes parties » partant de la triangulation de « petites parties ». La comparaison entre QuickBB et QuickTree révèle que QuickBB est meilleure que QuickTree en termes de temps de calcul et de passage à l'échelle. Toutefois, QuickBB n'est efficace que pour de petits graphes (autour de 200 sommets) et favorise les problèmes ayant une tree-width élévée vu sa complexité en $O(n^{n-w})$. Dans [Bachoore and Bodlaender, 2006], Bodlaender et Bachoore ont proposé un autre algorithme basé sur le principe du branch and bound. Cependant, son efficacité en pratique dépend de propriétés très restrictives du graphe. Par exemple, le graphe doit avoir au plus 20 sommets ou doit être triangulé ou encore avoir une tree-width relativement petite (inférieure à 7). En pratique, cet algorithme est surclassé par QuickBB.

En outre, des méthodes exactes basées sur SAT (problème de satisfiabilité booléenne) [Boole, 1854] et sa variante MaxSAT (problème de satisfiabilité maximum) [Johnson, 1973] ont été proposées pour le calcul de la tree-width et de la décomposition arborescente. En effet, dans [Berg and Järvisalo, 2014], les auteurs proposent d'exprimer le problème de détermination de la tree-width sous la forme d'une instance SAT (ou MaxSAT) en passant par la notion d'ordre d'élimination parfait. Plus précisément, la construction d'une décomposition arborescente n'est pas faite explicitement mais ils codent le fait que si la tree-width de G est w, alors il existe un ordre d'élimination parfait pour les sommets de X tel que le maximum de voisins ultérieurs d'un sommet dans le graphe triangulé est au plus w. Il est à noter que ce travail est basé sur celui de [Samer and Veith, 2009] avec comme différence le codage des règles qui est plus compact et plus efficace. Ces méthodes sont comparées entre elles et vis-à-vis des méthodes dédiées au calcul de la tree-width comme l'algorithme DP de [Bodlaender et al., 2006] et QuickBB de [Gogate and Dechter, 2004]. Les résultats montrent que les méthodes SAT (incrémentale et itérative) et MaxSAT de [Berg and Järvisalo, 2014] ont une meilleure performance que celles de [Samer and Veith, 2009]. En comparant DP aux méthodes SAT, DP n'est compétitive que pour les plus petits graphes (au plus 10 sommets) tandis qu'au-delà elle explose en espace. QuickBB surclasse DP et est légèrement surpassée par les méthodes SAT. Malgré cela, les méthodes proposées restent considérablement limitées au regard des tailles des graphes utilisés dans les expérimentations (128 sommets au maximum).

Récemment, les compétitions PACE (pour *Parameterized Algorithms and Computational Experiments*) 2016 et 2017 ont été organisées (voir `https://pacechallenge.wordpress.com/`). Un de leurs objectifs est d'évaluer les nouvelles implémentations disponibles pour le calcul de la tree-width. Cette compétition a notamment montré que le calcul de la tree-width commence à atteindre une efficacité intéressante. Parmi les instances utilisées, nous mentionnons les graphes nommés [NG1, 2016], les graphes de contrôle de flux [CF1, 2016] et des instances de coloration de graphes DIMACS [CI1, 2016]. Le nombre de sommets atteint 3 282 sommets. Parmi les soumissions, nous trouvons une implémentation basée sur l'approche de Arnborg et al. [Arnborg et al., 1987] [MU1, 2016], une deuxième comptant sur des séparateurs équilibrés et sur la programmation dynamique [UU1, 2016] et une basée sur un SAT-solveur [LU1, 2016]. La plupart des instances ont un temps limite de 100 s tandis que les autres ont un temps limite de 1 000 s ou de 3 600 s. L'implémentation basée sur l'approche de [MU1, 2016] réussit à trouver la tree-width pour la totalité des instances sauf une. La compétition PACE 2017 a montré que malgré l'utilisation des instances plus difficiles que celles de l'année 2016, le calcul de la tree-width

est significativement amélioré. La plupart des instances sont résolues en quelques secondes. Les deux meilleures soumissions ont réussi à résoudre toutes les instances disponibles en un temps maximum de 30 minutes.

Nous retenons qu'en pratique, les algorithmes exacts ont un intérêt très limité. Ils sont quasi-inopérants hormis pour des « petits » graphes même si de nouvelles implémentations plus efficaces sont apparues pour le calcul de la tree-width.

1.3.2.2 Algorithmes d'approximation avec garanties

Ces algorithmes visent à calculer une approximation de la tree-width à un facteur près. Plus précisément, la largeur calculée est inférieure ou égale à ce facteur multiplié par la tree-width. Il existe de nombreux algorithmes qui approximent la largeur arborescente et qui diffèrent selon leur complexité en temps et selon la garantie qu'il offrent quant à la qualité de la largeur qu'ils estiment. Nous présentons dans ce qui suit quelques-uns parmi les plus importants.

Algorithmes polynomiaux En ce qui concerne les algorithmes polynomiaux, le premier à avoir donné une garantie est celui proposé dans [Bodlaender et al., 1995]. Il calcule une décomposition arborescente dont la largeur n'est pas supérieure à $O(\log n)$ fois la largeur arborescente. Indépendamment, [Amir, 2001] et [Bouchitté et al., 2004] améliorent cet algorithme pour atteindre un facteur de $O(\log w)$. Amir exploite la notion des séparateurs tridirectionnels, très similaire à la notion des ρ-séparateurs utilisée par Bouchitté et al. La complexité en temps de l'algorithme de Amir est meilleure que celle de Bouchitté et al. tandis que sa constante cachée (au sens de la notation grand O) est de 850, plus grande que celle de Bouchité et al. qui est d'environ 675. Selon les auteurs de [Bouchitté et al., 2004], ces algorithmes ont uniquement un intérêt théorique puisque la recherche des ρ-séparateurs et des objets similaires dans un graphe est d'une réelle difficulté en pratique. L'algorithme polynomial qui de nos jours occupe le premier rang est celui présenté dans [Feige et al., 2008]. Il permet de calculer des décompositions dont la largeur est en $O(w.\sqrt{\log w})$. Il est à noter que le problème de trouver un algorithme polynomial d'approximation avec un facteur constant est un problème ouvert et très difficile. Les auteurs de [Austrin et al., 2012] contribuent à cette question et suggèrent, en se basant sur la conjecture SSE (small set expansion) [Raghavendra and Steurer, 2010], qu'il n'existe aucun algorithme polynomial dont la garantie est un ratio constant.

Algorithmes exponentiels Nous disposons également d'une grande variété d'algorithmes exponentiels qui, étant donné un entier k en entrée, affirment que la tree-width est plus grande que k ou bien calcule une décomposition arborescente de largeur au plus $c.k$ (pour une constante c donnée). Ces algorithmes partagent plus ou moins le même schéma algorithmique. Ils trouvent un séparateur qui sépare le graphe en plusieurs composantes connexes contenant « peu » de sommets. Si le séparateur est de « très grande taille », la tree-width est plus grande que k, sinon ce principe est réappliqué sur les sous-graphes induits par les composantes connexes et le séparateur en question. Par ailleurs, même les algorithmes polynomiaux de [Bouchitté et al., 2004; Amir, 2001; Bodlaender et al., 1995] sont structurés ainsi. Nous citons parmi ces algorithmes exponentiels les algorithmes présentés dans :

- [Reed, 1992] ayant une complexité en $2^{O(w.\log w)}.n.\log n$ et donnant une largeur d'au plus $8w + O(1)$,

Algorithme 1.2 : Heuristique-H (G)

Entrées : Un graphe $G = (X, C)$
Sorties : Un ordre d'élimination parfait σ, le graphe triangulé G'_σ

```
1 G'_σ ← G
2 pour i ← 1 à n faire
3     Choisir un sommet x de G'_σ selon le critère heuristique H
4     σ(x) ← i
5     Compléter le sous-graphe induit par G'_σ[N_u(x)]
6 retourner (σ, G'_σ)
```

- [Robertson and Seymour, 1995] ayant une complexité en $O(3^{3w}).n^2$ et donnant une largeur d'au plus $4w + 3$,
- [Lagergren, 1996] ayant une complexité en $2^{O(w.\log w)}.n.\log^2 n$ et donnant une largeur d'au plus $8w + 7$,
- [Amir, 2010] ayant des complexités en $O(2^{3,6982w}.w^3).n^2$ et en $O(2^{3w}.w^{3/2}).n^2$ avec respectivement des largeurs d'au plus $(3 + 2/3)w$ et de $4,5w$ et
- [Bodlaender et al., 2016] ayant des complexités en $2^{O(w)}.n$ et $2^{O(w)}.n.\log n$ avec des largeurs d'au plus $5w + 4$ et $3w + 4$.

L'algorithme proposé dans [Bodlaender et al., 2016] ayant le meilleur ratio (3) parmi les algorithmes polynomiaux en n et exponentiel en w a, selon les auteurs, principalement un intérêt théorique du fait de la grande constante à la base de l'exponentiel. Dans [Amir, 2010], Amir évalue les algorithmes proposés et les compare à un algorithme heuristique (minimum-degree). Les résultats reportés concernent des graphes d'au maximum 570 sommets et 3 820 arêtes. Les temps d'exécution des algorithmes proposés varient d'une minute à 6 jours tandis que l'algorithme heuristique a besoin d'au maximum 2 minutes. Un point important à préciser est que les largeurs calculées par l'algorithme heuristique sont toujours meilleures sur le benchmark utilisé que celles des algorithmes d'approximation. Il est à noter que les algorithmes heuristiques constituent une classe très importante parmi les classes des algorithmes de calcul de décompositions en raison de la difficulté du problème du calcul de la tree-width. C'est ainsi qu'ils feront l'objet de la partie suivante.

En résumé, du côté des algorithmes d'approximation, nous avons le choix entre des algorithmes polynomiaux dont le facteur est au mieux en $O(\sqrt{\log w})$ et des algorithmes exponentiels lents en pratique mais qui disposent d'un facteur constant. Malheureusement, il y a peu d'espoir d'obtenir un algorithme polynomial exploitable en pratique, dont la garantie serait un ratio constant.

1.3.2.3 Algorithmes heuristiques

Ces algorithmes calculent une borne supérieure de la tree-width en visant la meilleure estimation possible. Ils sont généralement de complexité polynomiale.

Approches par triangulation La plupart sont basés sur la notion de triangulation et partagent le schéma général de l'algorithme 1.2. L'algorithme *Heuristique-H* peut être instancié par l'heuristique H. Il vise à établir un ordre d'élimination parfait σ de G'_σ le graphe résultant du processus de triangulation. Le choix du prochain sommet x dépend

de l'heuristique H. Une fois le sommet x choisi, $N_u(x)$ sera transformé en clique à l'instar de l'algorithme 1.1. Les heuristiques qui peuvent être utilisées sont par exemple :

- *Minimum-Degree* [Markowitz, 1957] : L'heuristique Minimum-Degree ordonne les sommets de 1 à n et choisit comme prochain sommet celui ayant le plus petit degré dans G'_σ. La motivation de Minimum-Degree est la création de clusters de taille égale au degré du sommet sélectionné plus 1. Sa complexité est en $O(n^3)$.

- *Min-Fill* [Rose, 1972] : L'heuristique Min-Fill ordonne les sommets de 1 à n et choisit comme prochain sommet celui qui induit le nombre minimum d'arêtes à ajouter pour compléter son voisinage ultérieur. En d'autres termes, il choisit le sommet ayant le plus petit nombre de paires de voisins qui ne sont pas voisins entre eux. Les égalités sont généralement cassées arbitrairement. Min-Fill vise à ajouter le plus petit nombre d'arêtes et ainsi diminuer la possibilité d'augmenter les degrés des autres sommets lors de leur élimination. Cette heuristique s'avère légèrement plus lente en pratique que Minimum-degree. Or, les largeurs reportées par Min-Fill sont en moyenne plus intéressantes en pratique que celles de Minimum-degree [Bodlaender and Koster, 2010; Koster et al., 2001]. Dans le même sens, selon [Gogate and Dechter, 2004], Min-Fill a une meilleure performance que MCS [Tarjan and Yannakakis, 1984] et Minimum-Degree quant aux largeurs calculées. L'analyse de sa complexité en temps permet de déduire une complexité en $O(n(n + e'))$. Cette heuristique est notamment célèbre dans la communauté CP (tout comme MCS qui va être présenté ultérieurement) et y est considérée comme étant l'heuristique de l'état de l'art. Ces principaux points forts sont la qualité de la largeur calculée et son efficacité pratique en général.

- Des heuristiques alternatives ont été proposées dans [Bodlaender and Koster, 2010]. Elles considèrent deux paramètres $d_{G'_\sigma}(x_i)$ le degré de x_i dans G'_σ et $f_{G'_\sigma}(x_i)$ le nombre d'arêtes nécessaires pour la complétion du voisinage ultérieur de x_i. Ces heuristiques appelées, *Degree+FillIn*, *Sparsest-Subgraph*, *FillInDegree*, *DegreeFillIn*, choisissent comme sommet suivant le sommet qui minimise des combinaisons différentes des deux paramètres.

- D'autres heuristiques ont été proposées dans [Clautiaux et al., 2004; Bachoore and Bodlaender, 2005]. Par le biais d'une de ces heuristiques, le sommet suivant choisi est celui qui a la plus petite somme de deux fois la borne inférieure de la tree-width relative à l'élimination de ce sommet et son degré dans G'_σ. Notons que ces bornes inférieures sont calculées en phase de prétraitement. Une vue d'ensemble sur les méthodes calculant des bornes inférieures peut être retrouvée dans [Bodlaender and Koster, 2011].

Les deux heuristiques suivantes reposent sur deux algorithmes définis pour des graphes triangulés.

- *MCS (pour Maximum Cardinality Search)* [Tarjan and Yannakakis, 1984] : MCS ordonne les variables de n à 1 en choisissant comme prochain sommet celui ayant le plus grand nombre de sommets voisins parmi les sommets déjà numérotés. Le premier sommet choisi est sélectionné d'une façon arbitraire ainsi que le cassage des égalités. Sa complexité est en $O(n + e)$.

- *Lex* [Rose et al., 1976] : Il construit un ordre d'élimination en numérotant les sommets de n à 1. Chaque sommet est doté d'une étiquette qui est une séquence d'entiers

distincts entre 2 et n, initialisée à vide. Le prochain sommet x_i choisi est le sommet ayant l'étiquette la plus grande selon l'ordre lexicographique (les égalités sont cassées arbitrairement). Ensuite, le sommet x_i désigné ajoute $\sigma(x_i)$ à l'étiquette de tout sommet x_j non numéroté voisin de x_i. Le premier sommet est choisi arbitrairement. Sa complexité est en $O(n + e)$.

Pour calculer une triangulation, il suffit d'appliquer l'algorithme 1.1. Leur complexité est alors en $O(n + e')$.

Approche sans triangulation D'autres heuristiques construisent des décompositions arborescentes en recherchant un certain nombre de séparateurs dans le graphe. L'idée principale est de séparer le graphe en plusieurs parties grâce à un séparateur, de calculer récursivement une décomposition arborescente de chacune de ces parties et de rassembler finalement ces différentes décompositions. Nous avons déjà vu des algorithmes basés sur ce principe dans la partie dédiée aux algorithmes d'approximation avec garanties. Nous regardons maintenant de plus près l'heuristique MSVS (Minimum Separating Vertex Sets) de Koster [Koster, 1999] qui s'inspire du même principe sans donner explicitement une garantie quant à la largeur de la décomposition calculée. Cette heuristique peut être utilisée pour calculer une décomposition mais aussi pour raffiner une décomposition quelconque. Elle démarre généralement par une décomposition triviale, c'est-à-dire un cluster contenant tous les sommets de X. Son but consiste à examiner les plus grands clusters de la décomposition et d'essayer de les éclater en plusieurs clusters de taille plus petite. Cette opération vise à diminuer la largeur de la décomposition calculée. Pour raffiner un cluster E_i, elle construit tout d'abord le graphe $G[E_i]$. Ensuite, pour tout $x_j, x_k \in E_i$ et $j \neq k$ tels que $x_j, x_k \in E_l$ avec $i \neq l$, $\{x_j, x_k\}$ sera ajoutée à E_i. Après, elle recherche un séparateur minimum Y dans $G[E_i]$ grâce aux techniques de réseaux de flots (voir par exemple [Even, 1979]). Ainsi, le graphe $G[E_i \backslash Y]$ a au moins deux composantes connexes. Soit X_1, X_2, ..., X_p l'ensemble des p composantes connexes induites par $G[E_i \backslash Y]$. Le raffinement de E_i consiste alors à remplacer E_i par $p + 1$ clusters tels que $E_{i_0} = Y$ et pour $1 \leq q \leq p$, $E_{i_q} = Y \cup X_q$. Tous les autres nœuds de l'arbre restent intacts ainsi que les arêtes entre eux. D'autre part, i_0 est désormais voisin de chaque i_q avec $1 \leq q \leq p$. Finalement, chaque nœud $j \in I$ initialement voisin du nœud i sera voisin d'un des nouveaux nœuds i_q. Pour y parvenir, considérons $Y_j = E_j \cap E_i$. Notons que Y_j est une clique dans $G[E_i]$ en raison des arêtes ajoutées lors de la construction de $G[E_i]$. Par conséquent, il existe certainement une composante connexe X_q de $G[E_i \backslash Y]$ qui contient Y_j (voir lemme 2). Alors, j sera désormais un voisin de i_q. La décomposition résultante est une décomposition arborescente. Cette opération peut être répétée autant de fois que nécessaire jusqu'à ce que le plus grand cluster de la décomposition soit une clique. Dans ce cas, aucun raffinement n'est possible.

En pratique, les expérimentations ont montré que les heuristiques sont avantageuses du fait de leur bonne approximation de la tree-width et de leur temps de calcul souvent raisonnable [Bodlaender and Koster, 2010]. C'est ainsi qu'elles sont principalement exploitées vis-à-vis des autres méthodes de calcul de décompositions. Nous retenons notamment l'heuristique *Min-Fill* [Rose, 1972] qui assure un bon compromis entre le temps de calcul et la qualité de la largeur trouvée. Notons que la dernière compétition PACE a montré que de nouvelles implémentations améliorent d'une façon remarquable *Min-Fill*. En effet, la méthode qui a été classée première calcule la largeur en utilisant des améliorations locales, un algorithme de calcul exact de tree-width, un algorithme de calcul exact de path-width et un algorithme heuristique.

1.3.2.4 Algorithmes de triangulation minimaux

Le problème du calcul d'une triangulation minimale d'un graphe a été premièrement abordé dans les années 70. Le champ large d'applications du problème de la triangulation minimum a guidé la recherche vers le problème de la triangulation minimale du fait de leur rapprochement et que le premier est NP-difficile tandis que le deuxième est traitable. Deux caractérisations ont été données aux triangulations minimales dans [Rose et al., 1976] et sont la base de plusieurs algorithmes de calcul de triangulation minimale.

Caractérisation 1 *Une triangulation permettant d'obtenir G' à partir de G est minimale ssi le retrait de n'importe quelle arête de G' qui ne figure pas dans G produit un graphe non triangulé.*

Caractérisation 2 *Une triangulation permettant d'obtenir G' à partir de G est minimale ssi chaque arête ajoutée est l'unique arête reliant deux sommets non consécutifs d'un cycle de longueur* 4.

Nous regardons, tout d'abord, les algorithmes qui construisent un ordre d'élimination parfait minimal. Ces algorithmes se basent sur la caractérisation suivante de [Ohtsuki et al., 1976] :

Caractérisation 3 *Une triangulation permettant d'obtenir G' à partir de G est minimale ssi G' résulte de l'application d'un ordre d'élimination parfait minimal σ, c'est-à-dire qu'il n'existe aucun autre ordre σ' tel que son application à G produit un G'' de sorte que G'' est un graphe partiel de G'.*

- *Lex-M* [Rose et al., 1976] : Lex-M est une extension de Lex basée sur la caractérisation 2. Il construit un ordre d'élimination parfait σ en numérotant les sommets de n à 1. Chaque sommet est doté d'une étiquette initialisée à vide. Le prochain sommet x_i choisi est le sommet ayant l'étiquette la plus grande selon l'ordre lexicographique (les égalités sont cassées arbitrairement). Ensuite, le sommet x_i désigné ajoute $\sigma(x_i)$ à l'étiquette de tout sommet x_j non numéroté voisin de x_i ou tel qu'il existe une chaîne de x_j à x_i $[x_j, x_{k_1}, \ldots, x_{k_q}, x_i]$ de sorte que l'étiquette de chaque x_{k_p} soit inférieure à x_j et que x_{k_p} ne soit pas encore numéroté. Cette chaîne est appelé un *fill path*. Le premier sommet est choisi arbitrairement. Sa complexité est en $O(n.e)$. Notons que les auteurs de [Kratsch and Spinrad, 2006] proposent un algorithme d'une complexité en $O(n^{2,69})$.

- *MCS-M* [Berry et al., 2004] : À l'instar de Lex-M, MCS-M modifie l'étiquette de tout sommet x_j non numéroté auquel il est lié via un *fill path*. Cependant, au lieu d'utiliser des étiquettes lexicographiques, il utilise des poids. Alors, le prochain sommet choisi est celui qui a le plus grand poids. Sa complexité est aussi en $O(n.e)$.

- *Ohtsuki* [Ohtsuki, 1976] : Ohtsuki présente un algorithme qui exploite des principes similaires à ceux de Lex-M. Cependant, au lieu d'ordonner des sommets, il ordonne des sous-ensembles de sommets. Il partitionne X en une série de sous-ensembles qui satisfont des propriétés ensemblistes qu'il définit. Il démontre que les sommets du premier sous-ensemble peuvent être numérotés par les plus grands nombres, le deuxième sous-ensemble juste après, et ainsi de suite. Sa complexité est en $O(n.e)$. Toutefois, l'inconvénient réside dans la difficulté du calcul de la partition.

Nous nous focalisons maintenant sur les algorithmes exploitant la caractérisation des graphes triangulés par leurs séparateurs minimaux (cf. théorème 1). Un algorithme déduit directement de cette caractérisation et appelé SMS (pour *Saturate minimal separators*) consiste à trouver un séparateur minimal, le compléter et continuer ainsi jusqu'à ce qu'il n'y ait plus de séparateurs minimaux à compléter. La difficulté de trouver ces séparateurs constitue un obstacle pour son implémentation directe. D'autres algorithmes ont été proposés :

- *LB-Triang* [Berry et al., 2006] : Cet algorithme traite les n sommets les uns après les autres selon un ordre d'élimination quelconque O et construit progressivement le graphe G'. Pour chaque sommet x_i, les séparateurs minimaux de $N(x_i)$ dans G' sont complétés. L'ensemble d'arêtes ajoutées pour chaque sommet est inclus ou égal à l'ensemble d'arêtes ajoutées durant une triangulation classique. Une implémentation intelligente de cet algorithme a une complexité en $O(n.e)$ [Heggernes and Villanger, 2002]

- *Vertex incremental minimal triangulation* [Berry et al., 2003b] : Cet algorithme traite les sommets de G selon un ordre quelconque. Il part d'un ensemble $X_{\subseteq} = \emptyset$ qui représente l'ensemble des sommets déjà traités. Il reconstruit le graphe G en ajoutant ses sommets un à un à $X_{\subseteq}$ tout en rajoutant les arêtes liant ce sommet à des sommets déjà présents et celles nécessaires afin de maintenir une triangulation minimale de $G[X_{\subseteq}]$. L'algorithme exploite le théorème énoncé par les auteurs : un graphe G est triangulé ssi toute arête $\{x_i, x_j\}$ de G est telle que $N(x_i) \cap N(x_j)$ est un x_i,x_j-séparateur minimal dans $(X, C\backslash\{x_i, x_j\})$. Sa complexité temporelle est en $O(n.e)$.

- *FMT (Fast minimal triangulation)* [Heggernes et al., 2005] : Cet algorithme calcule une triangulation minimale de G d'une façon récursive. Étant donné un sous-ensemble $X_{\subseteq}$ de X, il sépare le graphe en plusieurs composantes connexes X_i induites par la suppression de $X_{\subseteq}$ de X. Ensuite, il complète les ensembles $N(X_i)$ pour obtenir le graphe G'. Finalement, il réapplique l'algorithme récursivement sur $G'[X_{\subseteq}]$ et les sous-graphes induits par chaque X_i, $G'[N[X_i]]$. Sa complexité en temps est en $O(n^{2,376})$.

Enfin, nous évoquons des algorithmes partant de la caractérisation 1.

- *MinimalChordal* [Blair et al., 2001] : Cet algorithme prend en entrée un graphe G et un ordre quelconque O. Il construit d'abord le graphe triangulé G'_O. Pendant ce calcul, il stocke à chaque étape i les arêtes ajoutées dans des ensembles séparés F^i. Le constat principal de l'algorithme est que le parcours de ces ensembles dans le sens inverse (F^n jusqu'à F^1) évite le fait de devoir vérifier plusieurs fois si une arête peut être supprimée. Ainsi, des arêtes candidates sont supprimées en suivant ce modèle. À chaque étape, dans le sens inverse, les sous-graphes non triangulés sont passés à Lex-M qui calcule une triangulation minimale. Finalement, il a été remarqué que l'application de Lex-M sur des petits graphes plusieurs fois consomme beaucoup moins de temps vis-à-vis d'une seule application sur tout le graphe. Sa complexité est en $O((e'-e)e')$ (e' est le nombre d'arêtes ajoutées lors du premier calcul de G'_O).

- *Dahlhaus* [Dahlhaus, 1997] : Cet algorithme calcule aussi un graphe triangulé à partir d'un ordre O suivi par un calcul de la décomposition arborescente (E, T). Il procède ensuite à un redécoupage des nœuds de T de sorte que les intersections des nœuds soient des séparateurs minimaux de G. Ce découpage est fait jusqu'à ce qu'aucun autre découpage ne soit possible. Sa complexité est en $O(n.e)$.

- *Peyton* [Peyton, 2001] : Cet algorithme est similaire à celui de Dahlhaus mais utilise des outils relatifs aux matrices creuses. Il se comporte bien en pratique mais aucune borne de complexité n'a été donnée.

- *Minimal minimum degree* [Berry et al., 2003a] : L'algorithme exploite l'heuristique minimum-degree qui est connue par sa bonne approximation de la largeur arborescente. En plus, il a été observé dans [Blair et al., 2001] et dans [Peyton, 2001] que cette heuristique calcule souvent une triangulation minimale. Ainsi, la première étape consiste à calculer une triangulation grâce à cette heuristique. Le but est de partir d'une triangulation de qualité et d'augmenter les chances de pouvoir calculer des triangulations ajoutant peu d'arêtes. À chaque étape de l'algorithme, les arêtes ajoutées, n'apparaîssant pas dans un séparateur minimal dans le voisinage d'un sommet, sont supprimées. La validité de l'algorithme ne dépend pas de l'ordre donné en entrée mais l'ordre donné par minimum-degree est fortement souhaitable pour une bonne performance en pratique.

Notons finalement que le constat commun à tous ces algorithmes est que la triangulation minimale peut être très éloignée d'une triangulation minimum. Notons aussi que bien que les algorithmes heuristiques comme Min-Fill ne garantissent pas une triangulation minimale, il s'avère en pratique qu'ils calculent très souvent une triangulation minimale [Kjaerulff, 1990]. Un état de l'art plus détaillé et plus complet est disponible dans [Heggernes, 2006].

Des algorithmes métaheuristiques ont été également proposées comme les algorithmes génétiques, la recherche tabu ou la recherche itérative locale (voir [Hammerl et al., 2015] pour une vue d'ensemble). Encore une fois, même si les méthodes métaheuristiques calculent de très bonnes bornes supérieures pour la tree-width, elles ne sont pas efficaces pour de gros graphes où les méthodes heuristiques calculent de bornes légèrement moins bonnes mais beaucoup plus efficacement.

1.3.2.5 Bilan

Parmi les différents types d'algorithmes présentés, il semble que les algorithmes heuristiques sont les plus exploitables en pratique. Ils ont la vertu de pouvoir passer à l'échelle contrairement aux autres algorithmes comme les algorithmes exacts. Ils ont souvent des temps d'exécution raisonnables tout en réussissant à calculer une estimation de la tree-width d'une bonne qualité. En effet, les largeurs calculées sont dans beaucoup de cas relativement proches de la largeur arborescente. Parmi les heuristiques les plus connues, nous citons *Min-Fill* et *MCS*.

Tous ces algorithmes se sont focalisés sur la minimisation de w^+, sans se soucier du problème qui sera traité une fois le graphe décomposé. En effet, selon le problème considéré, les décompositions visant à minimiser la taille des clusters peuvent être plus ou moins avantageuses. Cette question va particulièrement nous intéresser dans cette thèse.

1.4 Conclusion

Dans ce chapitre, nous avons rappelé tout d'abord des notions préliminaires sur les (hyper)graphes. Nous nous sommes particulièrement intéressés au cas des graphes triangulés qui occupent une place importante dans la compréhension des objectifs de cette thèse. En effet, nous avons vu que calculer une triangulation du graphe G est souvent utile pour trouver une décomposition arborescente de ce graphe. Ensuite, nous nous sommes attardés

sur cette notion de décomposition arborescente qui est la brique de base de cette thèse. La notoriété de cette décomposition provient du paramètre associé qui est la tree-width. En effet, pour certains problèmes combinatoires, le fait que la tree-width du graphe soit bornée garantit qu'il existe un algorithme de résolution polynomial. Après avoir rappelé d'autres décompositions existantes, nous nous sommes alors finalement focalisés sur les méthodes de calcul de la tree-width. En particulier, nous avons vu que les méthodes exactes de calcul de la tree-width sont incriminées à cause de leur temps de calcul élevé et de leur incapacité à passer à l'échelle. Les méthodes de calcul les plus exploitables semblent être les méthodes heuristiques. Entre autres, l'heuristique *Min-Fill* semble réaliser le meilleur compromis entre temps de calcul et qualité de largeur trouvée. C'est ainsi que nous comptons sur cette heuristique comme méthode de référence pour le calcul de décomposition arborescente. La question de l'efficacité de cette heuristique qui vise à minimiser w^+ (ainsi que celle des autres) vis-à-vis de son utilisation ultérieure sera traitée dans cette thèse.

Dans le chapitre suivant, nous nous focalisons sur les problèmes dont la résolution d'une manière efficace est l'objectif principal de cette thèse. Ces problèmes sont : le problème de satisfaction de contraintes (CSP), le problème du comptage (#CSP) et le problème de satisfaction de contraintes pondérées (WCSP).

Chapitre 2

Les problèmes CSP, #CSP et WCSP

Sommaire

2.1 Introduction

Ce chapitre s'intéresse au problème de satisfaction de contraintes (CSP) (section 2.2), au problème de dénombrement de solutions (#CSP) (section 2.3) ainsi qu'au problème de satisfaction de contraintes pondérées (WCSP) (section 2.4). Il vise à rappeler les principales notions sans pour autant vouloir être exhaustif. Pour des informations plus approfondies et plus complètes, nous invitons le lecteur à se référer à [Apt, 2003; Dechter, 2003; Rossi et al., 2006; Lecoutre, 2013] par exemple. Dans le cadre de cette thèse, nous nous intéressons à la résolution de ces problèmes. Nous visons à élaborer des méthodes de résolution efficaces exploitant en particulier la structure de l'instance en question. Cette structure peut être capturée par l'intermédiaire des décompositions d'hypergraphe que nous avons évoquées dans le chapitre précédent. Dans ce chapitre, nous allons donc fournir tous les éléments nécessaires à la compréhension des contributions correspondantes. Pour chaque problème, nous nous focalisons sur sa définition, sa sémantique et les principales méthodes de résolution connues.

2.2 Problème de satisfaction de contraintes : CSP

Le problème de satisfaction de contraintes (CSP) est suffisamment expressif et général pour pouvoir avoir des domaines d'application très vastes comme l'ordonnancement de tâches avec ou sans ressources, l'élaboration d'emplois du temps, la planification, la configuration, le raisonnement, la résolution des jeux et de nombreux autres problèmes réels ou académiques. La brique de base à l'origine de sa puissance est la notion de *contrainte*. Une contrainte est un critère ou une propriété portant sur certains objets appelés *variables* et les mettant en relation. Cette contrainte limite les *valeurs* que peuvent prendre simultanément ces variables. Ces valeurs sont sélectionnées parmi l'ensemble des valeurs possibles d'une variable appelé, son *domaine*. Le but du problème est d'attribuer à chaque variable du problème une valeur de son domaine de sorte que toutes les contraintes soient satisfaites, c'est-à-dire que, pour chaque contrainte, les valeurs de ses variables sont compatibles vis-à-vis de la propriété qu'elle définit. La place que ce problème occupe à la fois en intelligence artificielle et en recherche opérationnelle est dûe à la grande variété de types de contraintes existantes algébriques, temporelles, géométriques... Cela lui offre un pouvoir de modélisation important. Le problème le plus connu parmi les problèmes associés à une instance CSP consiste à dire si une telle attribution de valeurs est possible. Il s'agit d'un problème de décision qui est NP-complet. C'est ainsi le prix à payer pour avoir ce niveau d'expressivité. Heureusement, ce problème admet, de nos jours, des méthodes de résolution efficaces grâce à l'essor qu'il a connu ces 40 dernières années. Nous regardons ainsi, de plus près, ce problème dans cette partie.

2.2.1 Formalisme

Nous donnons maintenant la définition formelle d'une instance du problème de satisfaction de contraintes (CSP) [Montanari, 1974].

Définition 32 *Une* instance du problème de satisfaction de contraintes CSP *est définie par la donnée d'un triplet* $P = (X, D, C)$ *où :*

- $X = \{x_1, x_2, \ldots, x_n\}$ *est un ensemble de* n *variables,*
- $D = \{D_{x_1}, D_{x_2}, \ldots, D_{x_n}\}$ *est un ensemble de domaines finis tel que chaque* D_{x_i} *correspond à une variable* x_i*,*

- *$C = \{c_1, c_2, \ldots, c_m\}$ est un ensemble de m contraintes. Chaque contrainte c_i est un couple $(S(c_i), R(c_i))$ où :*
  - *$S(c_i) = \{x_{i_1}, x_{i_2}, \ldots, x_{i_q}\} \subseteq X$ est la portée de c_i avec $|S(c_i)|$ l'arité de la contrainte c_i,*
  - *$R(c_i) \subseteq D_{x_{i_1}} \times D_{x_{i_2}} \times \cdots \times D_{x_{i_q}}$ est sa relation de compatibilité.*

L'arité maximale des contraintes r est égale à $max_{c_i \in C}|S(c_i)|$. La taille maximum des domaines d est $max_{x_i \in X}|D_{x_i}|$.

Les instances CSP considérées dans cette thèse sont dites *normalisées*.

Définition 33 *[Apt, 2003; Bessiere, 2006] Une instance CSP P est dite* normalisée *ssi il n'existe pas deux contraintes différentes c_i et c_j dans C telles que $S(c_i) = S(c_j)$.*

Une instance CSP peut être binaire ou n-aire.

Définition 34 *Une instance CSP est dite* binaire *si l'arité de chaque contrainte $c_i \in C$ est égale au plus à 2. Si l'arité d'une contrainte est supérieure à 2, l'instance CSP est dite* n-aire.

Notons qu'une contrainte portant uniquement sur deux variables est dite binaire. Une contrainte binaire portant sur les variables x_i et x_j sera notée c_{ij}. Au contraire, une contrainte dont la portée contient plus que deux variables est dite n-aire.

Les relations associées à ces contraintes peuvent être représentées *en extension ou en intention*. Pour représenter la relation d'une contrainte en extension, les tuples des valeurs autorisés (supports) ou interdits (conflits) sont énumérés. Par exemple, (X, D, C) avec $X = \{x_1, x_2\}$, $D = \{\{1, 2\}, \{1, 2\}\}$ et $C = \{c_1 = (\{x_1, x_2\}, \{(1, 1), (2, 1), (2, 2)\})\}$ est une instance CSP contenant une seule contrainte c_1 dont la relation est exprimée en extension. Il s'agit de la contrainte $\ll x_1$ est plus grand ou égal à $x_2 \gg$. La même relation peut être représentée en intention en utilisant des propriétés mathématiques connues comme le prédicat $x_1 \geq x_2$. Une contrainte exprimée initialement en intention peut être exprimée en extension. En pratique, l'utilisation d'une représentation ou l'autre n'est pas sans conséquences. La représentation des contraintes en extension peut induire des coûts en espace mémoire très importants du fait du nombre de tuples énumérés qui peut être très élevé. Une représentation en intention peut réduire significativement l'espace mémoire requis. Le test de satisfaction d'une contrainte par un tuple a, à son tour, un coût en temps différent selon que les contraintes soient représentées en extension ou en intention. Par exemple, Si les contraintes sont représentées en extension, le test de satisfaction se ramène à la recherche du tuple dans la table des tuples correspondante. Ce test peut ainsi être fait en temps constant grâce à une implémentation judicieuse. Sinon, ce test consiste à vérifier la formule ou le prédicat en question. Par conséquent, ce coût peut ne pas être constant.

Une *contrainte globale* est l'une des meilleures représentations d'une contrainte ayant un nombre trop important de tuples. Cette contrainte a une signification implicite. Sa définition telle qu'évoquée dans [Lecoutre, 2013] est :

Définition 35 *Une* contrainte globale *est un modèle de contrainte qui capture des relations ayant une sémantique précise et qui peut impliquer un nombre quelconque de variables.*

Nous pouvons citer par exemple la contrainte *allDifferent* qui signifie que toutes les variables entrant en jeu doivent impérativement avoir des valeurs différentes. Il est facile de

voir que cette contrainte peut porter sur un nombre quelconque de variables. Pour plus d'informations sur les contraintes globales, le lecteur peut se référer à [Beldiceanu et al., 2005; van Hoeve and Katriel, 2006] par exemple.

La structure d'une instance CSP $P = (X, D, C)$ est donnée par un hypergraphe appelé *hypergraphe de contraintes*, tel que :

- chaque sommet représente une variable de X,
- chaque hyperarête correspond à la portée $S(c_i)$ d'une contrainte c_i de C.

Notons que, dans le cas d'une instance CSP binaire, l'hypergraphe est un graphe.

Exemple 1 *Considérons maintenant l'exemple suivant. Robert est un père de famille qui a* 6 *enfants. Il dispose des billets de* 5, 10, 20 *et* 50 *euros et des pièces de* 1 *ou* 2 *euros. Il veut donner de l'argent à ses enfants à raison d'un billet ou d'une pièce par enfant. Comme il est un peu maniaque des mathématiques, il élabore un ensemble de contraintes algébriques pour la distribution de l'argent. Les contraintes expriment des relations algébriques relativement simples comme le fait qu'un fils recevra plus d'argent qu'un autre ou que la somme de l'argent de* 3 *fils devra être inférieure à une somme donnée. Ce problème peut être représenté sous forme d'une instance CSP P comme suit :*

- $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$,
- $D = \{D_{x_1}, D_{x_2}, D_{x_3}, D_{x_4}, D_{x_5}, D_{x_6}\}$ *avec* $D_{x_i} = \{1, 2, 5, 10, 20, 50\}, \forall 1 \leq i \leq 6$,
- $C = \{c_1, c_2, c_3, c_4\}$ *avec :*
  - $c_1 = (\{x_1, x_2\}, x_2 \geq x_1)$,
  - $c_2 = (\{x_1, x_5\}, 3x_1 + x_5 > 50)$,
  - $c_3 = (\{x_3, x_4, x_5\}, x_3 + x_4 + x_5 < 8)$,
  - $c_4 = (\{x_4, x_5, x_6\}, 2x_4 + x_5 = x_6)$.

Dans cet exemple, X est l'ensemble des 6 *enfants avec une variable par enfant. Chaque variable* x_i *peut prendre une valeur parmi les valeurs de* D_{x_i} *représentant les valeurs des billets et des pièces dont dispose Robert. d est alors égal à* 6*. Les relations associées aux contraintes sont représentées en intention par des équations et des inéquations. Elles peuvent aussi être représentées par des tables de compatibilité.* $R(c_3)$ *est alors représentée par la table 2.1. Les contraintes* c_3 *et* c_4 *portent sur le plus grand nombre de variables (*3 *variables). D'où, r est égal à* 3*. Ce problème peut être représenté par l'hypergraphe de contraintes de la figure 2.2 contenant* 6 *sommets et* 4 *hyperarêtes.*

2.2.2 Sémantique

Maintenant que nous avons défini formellement le problème de satisfaction de contraintes (CSP), nous allons nous intéresser à sa sémantique.

Une notion préliminaire est la notion d'*affectation*. On emploie également les termes d'*instanciation* ou d'*assignation*.

Définition 36 *Soit* $P = (X, D, C)$ *une instance CSP. L'*affectation *d'une variable* x_i *de X est l'attribution d'une valeur* v_i *à* x_i *telle que* $v_i \in D_{x_i}$*. Elle est notée* $x_i \leftarrow v_i$*. Elle est également définie pour* $X_\subseteq$ *sous-ensemble de X avec* $X_\subseteq = \{x_{i_1}, x_{i_2}, \ldots, x_{i_q}\}$*.* $\mathcal{A}$ *est appelée affectation de* $X_\subseteq$ *si elle associe à chaque variable* x_{i_p} *de* $X_\subseteq$ *avec* $1 \leq p \leq q$*, une valeur* $v_{i_p} \in D_{x_{i_p}}$*.*

$R(c_3) : x_3 + x_4 + x_5 < 8$		
x_3	x_4	x_5
1	1	1
1	1	2
1	1	5
1	2	1
1	2	2
1	5	1
2	1	1
2	1	2
2	2	1
2	2	2
5	1	1

FIGURE 2.1 – La relation associée à la contrainte c_3 donnée en extension (supports).

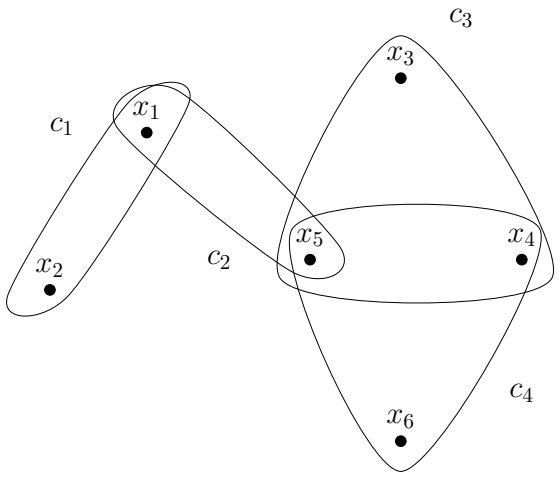


FIGURE 2.2 – L'hypergraphe de contraintes correspondant à l'instance P.

Une affectation $\mathcal{A}$ peut être représentée sous forme d'une association variable/valeur, c'est-à-dire : $\mathcal{A} = \{x_{i_1} \leftarrow v_{i_1}, x_{i_2} \leftarrow v_{i_2}, \ldots, x_{i_q} \leftarrow v_{i_q}\}$ ou simplement comme une séquence de valeurs $(v_{i_1}, v_{i_2}, \ldots, v_{i_q})$. L'ordre des variables de $X_{\subseteq}$ est implicite. Une affectation qui porte sur toutes les variables de X est dite *complète*, elle est dite *partielle* sinon. Ainsi, si $X_{\mathcal{A}}$ note l'ensemble de variables sur lesquelles porte l'affectation $\mathcal{A}$, $\mathcal{A}$ est dite complète si $X_{\mathcal{A}} = X$ et partielle sinon.

Nous pouvons aussi restreindre l'affectation $\mathcal{A}$ à un sous-ensemble de $X_{\mathcal{A}}$.

Définition 37 *Soit $\mathcal{A}$ une affectation et $X_{\subseteq} \subseteq X_{\mathcal{A}}$. La projection de A sur $X_{\subseteq}$, notée $\mathcal{A}[X_{\subseteq}]$, est la restriction de $\mathcal{A}$ aux variables de $X_{\subseteq}$.*

Dans l'exemple 1, une affectation $\mathcal{A}$ complète de X peut être : $\mathcal{A} = \{x_1 \leftarrow 1, x_2 \leftarrow 5, x_3 \leftarrow 50, x_4 \leftarrow 10, x_5 \leftarrow 20, x_6 \leftarrow 50\}$. $A[\{x_1, x_2\}] = \{x_1 \leftarrow 1, x_2 \leftarrow 5\}$.

Une contrainte peut être *satisfaite, violée ou ni l'un, ni l'autre* par une affectation.

Définition 38 *Soient P une instance CSP et $\mathcal{A}$ une affectation donnée. $\mathcal{A}$ satisfait la contrainte $c_i = (S(c_i), R(c_i))$ de C si $S(c_i) \subseteq X_{\mathcal{A}}$ et $\mathcal{A}[S(c_i)] \in R(c_i)$. Au contraire, $\mathcal{A}$ viole c_i si $S(c_i) \subseteq X_{\mathcal{A}}$ et $\mathcal{A}[S(c_i)] \notin R(c_i)$.*

Dans l'exemple 1, l'affectation $\mathcal{A}$ satisfait la contrainte c_1 vu que $\mathcal{A}[\{x_1, x_2\}] = (1, 5) \in R(c_1)$ $(5 \geq 1)$ mais viole c_2 puisque $\mathcal{A}[\{x_1, x_5\}] = (1, 20) \notin R(c_2)$ $(3.1 + 20 = 23 < 50)$.

Nous définissons maintenant la notion d'*affectation cohérente*.

Définition 39 *Étant donnée une instance CSP $P = (X, D, C)$, une affectation $\mathcal{A}$ d'un sous-ensemble de variables de X est cohérente ssi :*

$$\forall c_i \in C \text{ telle que } S(c_i) \subseteq X_{\mathcal{A}}, \mathcal{A} \text{ satisfait } c_i.$$

En d'autres termes, une affectation est cohérente si elle ne viole aucune contrainte. La vérification de la cohérence d'une affectation s'effectue en temps polynomial.

Cette définition nous mène à la définition d'une *solution* d'une instance CSP.

Définition 40 *Une solution d'une instance CSP $P = (X, D, C)$ est une affectation complète cohérente. L'affectation satisfait alors toutes les contraintes de P. L'ensemble de solutions de P est noté Sol_P.*

Il est à noter que vérifier si *une affectation complète est une solution* est un problème traitable en temps polynomial. Dans l'exemple 1, l'affectation complète $\mathcal{A}$ spécifiée n'est pas une solution puisqu'elle viole c_2. Nous pouvons facilement vérifier dans l'exemple 1 que l'affectation $\mathcal{A} = \{x_1 \leftarrow 20, x_2 \leftarrow 50, x_3 \leftarrow 2, x_4 \leftarrow 2, x_5 \leftarrow 1, x_6 \leftarrow 5\}$ est une solution du problème.

Finalement, nous définissons une instance CSP *cohérente*.

Définition 41 *Une instance CSP $P = (X, D, C)$ est dite cohérente ssi $Sol_P \neq \emptyset$*

Autrement dit, une instance CSP P est cohérente si elle admet au moins une solution et incohérente sinon. L'instance de l'exemple 1 est ainsi cohérente car elle admet au moins une solution.

Une propriété importante sur les affectations est la *cohérence globale*.

Définition 42 *Étant donnée une instance CSP $P = (X, D, C)$, une affectation $\mathcal{A}$ sur un sous-ensemble de variables de X est dite globalement cohérente ssi il existe une solution $\mathcal{S}$ de Sol_P telle que $\mathcal{A} \subseteq \mathcal{S}$.*

Dans l'exemple 1, $\mathcal{A} = \{x_1 \leftarrow 20, x_2 \leftarrow 50\}$ est une affectation globalement cohérente car $\mathcal{S} = \{x_1 \leftarrow 20, x_2 \leftarrow 50, x_3 \leftarrow 2, x_4 \leftarrow 2, x_5 \leftarrow 1, x_6 \leftarrow 5\}$ est une solution de cette instance.

Nous fournissons également la définition de l'équivalence des instances CSP.

Définition 43 *Deux instances CSP $P = (X, D, C)$ et $P' = (X', D', C')$ sont dites équivalentes ssi $Sol_P = Sol'_P$.*

D'après cette définition, résoudre P est équivalent à résoudre P' puisqu'elles ont le même ensemble de solutions. Cette équivalence est d'une grande importance. Grâce à l'équivalence des instances CSP, nous pouvons, pour la résolution de P, considérer le problème P' dont la résolution est éventuellement plus simple. Dans l'exemple 1, en constatant que la somme des variables x_3, x_4 et x_5 doit être inférieure à 8, nous pouvons facilement déduire que ces variables ne peuvent pas avoir les valeurs 10, 20 ou 50. Ainsi, la suppression de ces 3 valeurs des domaines D_{x_3}, D_{x_4} et D_{x_5} résultent en une nouvelle instance CSP équivalente à la première parce qu'elle ne modifie pas l'ensemble des solutions de P. Or, comme la taille des domaines est désormais plus petite, la résolution du nouveau problème est *a priori* plus simple vu que le nombre d'affectations complètes possibles est plus faible.

Étant donnée une instance CSP, de nombreux problèmes peuvent nous intéresser :

- Une instance CSP est elle cohérente ? : il s'agit d'un problème de décision qui est NP-complet,
- Calculer le nombre de solutions,
- Rechercher une ou toutes les solutions,
- Trouver un ensemble des valeurs qui figure dans l'ensemble de solutions,
- ...

La difficulté théorique et pratique varie d'un problème à l'autre. Ainsi, dire si une instance CSP possède une solution est sûrement moins difficile que le fait de compter toutes les solutions de cette dernière. Ce fait est constaté en pratique, mais est aussi prouvé en théorie vu que le premier est NP-complet tandis que le deuxième est #P-complet. Dans cette thèse, nous nous intéressons aux questions de l'existence d'une solution et du dénombrement des solutions d'une instance.

2.2.3 Solveurs modernes

La résolution du problème CSP a considérablement évolué durant la dernière décennie. Si nous parlions avant d'une méthode de résolution, aujourd'hui la méthode de résolution employée n'est plus qu'une instanciation possible d'un solveur. Notons d'ailleurs que ce fait peut être facilement constaté dans les compétitions CSP organisées. Les solveurs modernes rassemblent des techniques et des mécanismes diversifiés plus ou moins sophistiqués. Ils témoignent d'une efficacité remarquable, ce qui a permis de mettre en valeur davantage le cadre CSP. Le point de vue nouvellement adopté par certains travaux comme [Puget, 2004; Gent et al., 2006] au sein de la communauté consiste à considérer le solveur comme une *boîte noire*. Selon les partisans de ce point de vue, le principal défi qui se pose à la programmation par contraintes est la *simplicité de l'utilisation*. L'enjeu ne se limite pas à la résolution de l'instance en question mais s'étend à la modélisation de l'instance elle-même. D'une part, la prise en compte de contraintes hétérogènes a enrichi le cadre CSP et lui a fait gagné en pouvoir de modélisation et en intérêt pratique depuis les années 90. D'autre part, la modélisation d'un problème est devenue plus difficile et pourrait nécessiter une expertise afin de profiter pleinement des algorithmes liés à chaque type de contrainte (les contraintes globales par exemple) [Lecoutre, 2013]. Au-delà de la modélisation, idéalement, l'utilisateur ne doit pas être conscient des techniques et des algorithmes employés pour la résolution de l'instance. Au contraire, seules les entrées et les sorties du solveur lui seront visibles. Il n'est pas ainsi responsable de modifier, d'étendre ou d'adapter le solveur à l'instance à résoudre. En effet, un bon solveur est capable de s'adapter à l'instance à résoudre par le biais des techniques sophistiquées qui y sont implémentées. Ce faisant, le solveur permet de compenser les défauts de la modélisation tout en gagnant en robustesse. Par conséquent, l'efficacité du solveur est améliorée. Se rapprocher davantage d'un solveur boîte noire, faciliterait son emploi par des non experts ce qui augmenterait potentiellement l'impact de la programmation par contraintes dans le monde industriel et académique.

Dans la suite de cette section, nous examinerons les différents éléments constitutifs d'un solveur montrés dans la figure 2.3. Les principales briques de base sont :

- le type de branchement,
- le filtrage,
- les retours-arrière chronologiques ou non chronologiques,

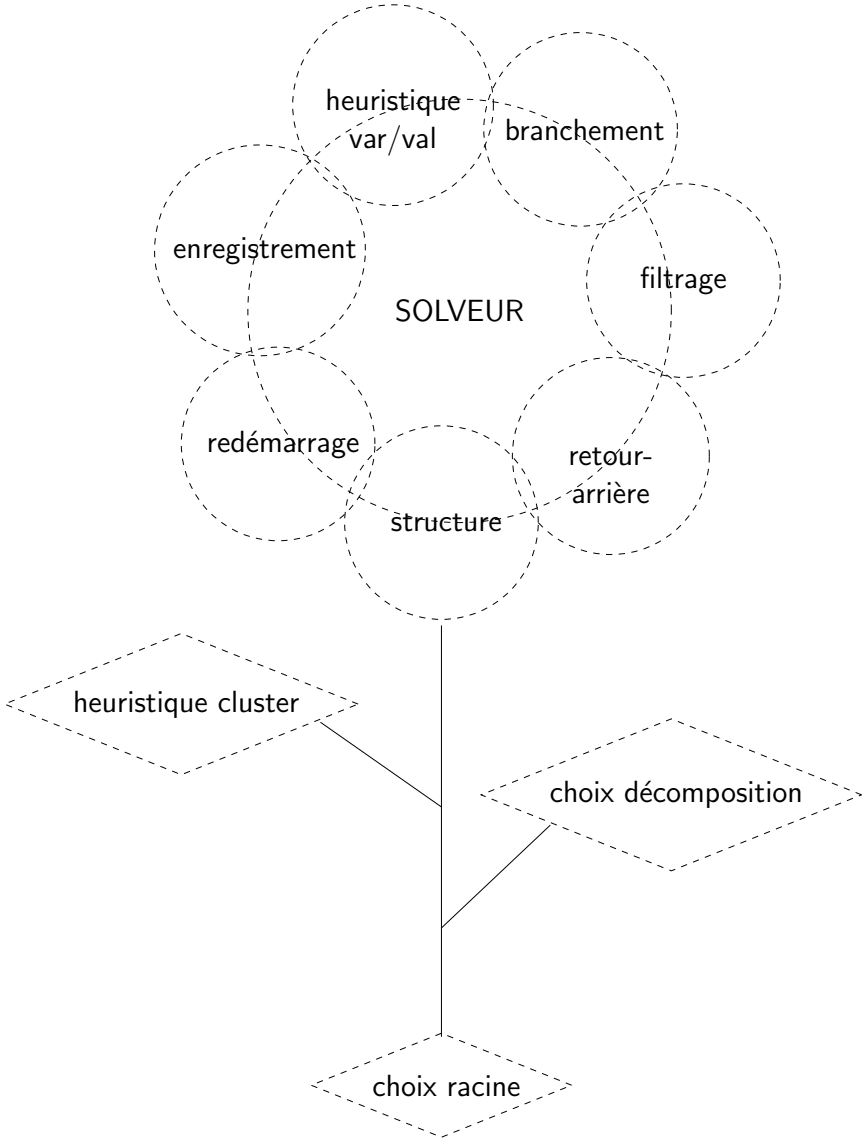


FIGURE 2.3 – Les principales techniques intégrées dans un solveur.

- les enregistrements,
- les heuristiques de choix de variables/valeurs,
- les redémarrages,
- **l'exploitation de la structure**.

Nous nous intéressons donc au type de branchement exploité (binaire ou d-aire), aux techniques de filtrage éventuellement utilisées en prétraitement et pendant la résolution, et aux retours-arrière chronologiques ou non chronologiques employés. Nous nous focalisons également sur les enregistrements pouvant être réalisés, sur les heuristiques de choix de variables et de valeurs employées et sur les redémarrages qui seront probablement exploités. Finalement, nous nous concentrons sur l'exploitation de la structure à laquelle nous accorderons un intérêt particulier vu que l'objectif principal de cette thèse consiste à améliorer les méthodes structurelles. Malheureusement, les solveurs actuels exploitent

rarement cette brique qui est le plus souvent absente. Sa présence dans un solveur est l'une des ambitions de ce travail. Un algorithme de résolution est une configuration précise de ces différents paramètres.

2.2.4 Résolution dans le cas général

Différentes techniques de résolution du problème CSP ont été développées. Elles peuvent être classées en méthodes *complètes* ou *incomplètes*. Les méthodes complètes garantissent de vérifier l'existence d'une solution ou à en détecter l'absence sinon. Si la méthode n'est pas complète, elle est dite incomplète. Les méthodes incomplètes sont connues par leur efficacité vu qu'elles sacrifient la complétude. Les algorithmes de *recherche locale* [Hoos and Stützle, 2004; Hoos and Tsang, 2006] sont qualifiés d'incomplets. Ils permettent généralement de trouver une solution dans un « temps raisonnable », mais ne permettent pas d'en déduire l'absence. Dans cette thèse, nous nous intéressons uniquement aux méthodes complètes.

Les méthodes complètes mettent en avant deux catégories d'algorithmes :

- les *algorithmes de recherche énumératifs*[1] classiques [Beek, 2006],
- les *algorithmes basés sur la programmation dynamique* [Bertele and Brioschi, 1972; Dechter, 2006].

Les algorithmes énumératifs entrelacent *recherche arborescente* et *simplification du problème* par le biais des méthodes de filtrage. Le problème CSP est connu pour être NP-complet. C'est ainsi que les méthodes de résolution énumératives existantes sont exponentielles en n en temps. Nous nous intéressons dans un premier temps aux algorithmes énumératifs classiques qui n'exploitent pas la structure du problème, du moins explicitement.

Dans un second temps, nous nous focalisons sur les méthodes dites *structurelles*. Ces méthodes ont suscité l'engouement de la communauté en raison de leur complexité théorique temporelle avantageuse par rapport aux autres méthodes énumératives classiques. Pour y parvenir, ces méthodes exploitent la structure de l'(hyper)graphe de contraintes représentant le problème CSP en question. Nous nous concentrons sur les méthodes à base de la *décomposition arborescente* dont nous avons déjà vu la définition dans la partie 1.3.1 et les méthodes de calcul dans la partie 1.3.2. En effet, l'objectif de cette thèse est de faire évoluer les méthodes de résolution à base d'une décomposition arborescente que ce soit pour le problème CSP, #CSP ou le problème WCSP. Derrière cet intérêt se cache la notion des *classes polynomiales*. Comme tout problème NP-complet, la NP-complétude du problème CSP ne signifie pas forcément qu'il n'existe pas de classes d'instances pouvant être résolues en temps polynomial. Une classe polynomiale est un ensemble d'instances qui admettent un algorithme capable de les résoudre en temps polynomial. Nous expliquons le lien entre les classes polynomiales et les décompositions arborescentes dans la partie dédiée aux méthodes structurelles.

Dans ce qui suit, nous détaillons d'abord les différentes briques d'un solveur avant de nous focaliser sur l'exploitation de la structure. Tous les algorithmes de résolution se basent sur l'algorithme naïf *Generate and test*. Il consiste à générer toutes les affectations complètes possibles en visitant la totalité de l'espace de recherche. Il procède ensuite à la vérification de la cohérence de ces affectations. Évidemment, cet algorithme explose combinatoirement. Sa complexité est en $O(exp(n))$ et il s'avère donc inopérant en pratique. C'est pourquoi d'autres algorithmes ont été proposés.

1. au sens de l'énumération de l'ensemble des affectations possibles jusqu'à l'obtention d'une solution et non pas de l'ensemble de solutions du problème

Algorithme 2.1 : BT $(P, \mathcal{A}, V)$

Entrées : Une affectation $\mathcal{A}$, un ensemble de variables V
Entrées-Sorties : $P = (X, D, C)$: une instance CSP
Sorties : *vrai* si l'affectation $\mathcal{A}$ s'étend d'une façon cohérente sur les variables de V, *faux* sinon

```
1  si V = ∅ alors
2  |  retourner vrai
3  sinon
4  |  Choisir x ∈ V
5  |  D ← D_x
6  |  tant que D ≠ ∅ et ¬cohérent faire
7  |  |  Choisir v ∈ D
8  |  |  D ← D − {v}
9  |  |  si A ∪ {x ← v} ne viole aucune contrainte de C alors
10 |  |  |  cohérent ← BT(P, A ∪ {x ← v}, V\{x})
11 |  retourner cohérent
```

2.2.4.1 Algorithme Backtracking (BT)

Nous nous servons de l'algorithme de backtracking BT comme algorithme de base. BT prend en entrée un ensemble de variables V qu'il vise à instancier de façon cohérente en partant d'une affectation $\mathcal{A}$ vide. Pour y parvenir, il se base sur un *parcours en profondeur d'abord* des variables et construit un arbre de recherche. Un arbre de recherche est un arbre orienté dont les nœuds sont les variables du problème et les arcs sont les différentes valeurs des domaines. L'arc partant du nœud de la variable x portant sur la valeur v correspond à l'affectation $x \leftarrow v$. Ces affectations apparaîssent dans l'ordre dont lequel elles sont réalisées. Une branche de l'arbre correspond alors à une séquence d'affectations faite. Le but de l'algorithme backtracking est d'éviter de visiter systématiquement tout l'espace de recherche, c'est-à-dire de tester toutes les combinaisons possibles des valeurs des variables à l'image de *generate and test*. En effet, si l'affectation $\mathcal{A}[V_\subseteq]$ de $V_\subseteq \subseteq V$ s'est avérée incohérente, il est évident que toute affectation la contenant est également incohérente. Ainsi, l'algorithme backtracking rejette cette affectation partielle incohérente ainsi que toute affectation pouvant la contenir. Cette idée permet d'économiser la génération et le test d'affectations infructueuses. L'algorithme backtracking est présenté par l'algorithme 2.1. Si l'ensemble de variables V est vide, alors il ne reste aucune variable à instancier et ainsi l'affectation $\mathcal{A}$ est cohérente (lignes 1 et 2). Si initialement $V = X$ alors l'instance CSP est cohérente et $\mathcal{A}$ est une solution. Si $V \neq \emptyset$, une variable x est sélectionnée parmi les variables de V selon un ordre défini sur ces variables (ligne 4). Les valeurs attribuées à x sont choisies parmi les valeurs de son domaine D_x (ligne 5). Lorsqu'une valeur v de D_x est choisie (ligne 7), BT vérifie la cohérence de l'affectation $\mathcal{A} \cup \{x \leftarrow v\}$ (ligne 9). Si cette dernière ne viole aucune contrainte, BT poursuit en sélectionnant une autre variable et en essayant de l'instancier de façon cohérente (ligne 10). Sinon, l'affectation $\mathcal{A} \cup \{x \leftarrow v\}$ est rejettée et une autre valeur de D_x est testée. Si en essayant toutes les valeurs possibles v, $\mathcal{A} \cup \{x \leftarrow v\}$ est incohérente, BT revient d'une façon chronologique sur la dernière variable instanciée et tente de l'instancier différemment. Si BT fait un retour en arrière (backtrack) tandis que $\mathcal{A}$ est vide, alors aucune affectation cohérente des variables de V ne peut être trouvée. Si, en plus, $V = X$ alors l'instance CSP en question est incohérente. Dans le

pire des cas, cette méthode explore tout l'espace de recherche et a ainsi une complexité en $O(m.r.d^n)$ (ou $O(exp(n))$). BT s'avère totalement inefficace en pratique à cause de la taille de l'espace de recherche exploré. Parmi ses inconvénients nous citons :

- la découverte tardive des incohérences,
- le retour en arrière chronologique,
- la redondance des incohérences locales découvertes,
- l'ordre arbitraire défini pour le choix de la prochaine variable à instancier (le choix des instanciations au début de la recherche a un impact important sur la suite de la résolution).

Les améliorations de BT traitent ces différents aspects. Le *filtrage* assure qu'à chaque nouvelle décision, nous testons si l'affectation obtenue respecte un certain niveau de cohérence. Il permet de supprimer des valeurs des variables non encore instanciées qui ne sont pas compatibles avec l'affectation courante. Cela évite essentiellement l'exploration de certains sous-espaces de recherche inutiles, c'est-à-dire ne contenant pas de solutions. Le retour en arrière est aussi amélioré en permettant un *retour en arrière non chronologique* Cela évite de remettre en cause des affectations qui n'ont aucun impact sur la résolution. L'*enregistrement* des informations peut être utilisé. Il permet d'éviter certaines redondances et le gaspillage du temps. D'autres travaux se sont focalisés sur l'amélioration du *choix de la prochaine variable* qui s'est montré essentiel pour permettre une résolution efficace et faire des choix judicieux. Au-delà, les techniques de *redémarrage* ont considérablement amélioré la résolution parce qu'elles remettent en cause en particulier les affectations réalisées au plus tôt. Finalement, le *type de décisions prises* réalisé par BT est d-aire (un nœud peut avoir au maximum d fils, un fils par valeur du domaine). Un branchement binaire (un nœud peut avoir deux fils) pourrait être toutefois particulièrement bénéfique pour la résolution de certaines instances. Nous pouvons ainsi constater que l'efficacité des solveurs actuels dépend d'un grand nombre de paramètres qui sont tous d'une grande importance. Notons qu'une configuration qui permet de résoudre une instance efficacement peut avoir une performance médiocre pour une autre instance.

Nous détaillons tout d'abord les types de branchement pouvant être réalisés.

2.2.4.2 Type de branchement réalisé

Un algorithme réalise une série de décisions que nous notons Σ. Nous pouvons distinguer deux types de décisions :

- une *décision positive* de la forme $x = v$ qui affecte la valeur v à la variable x,
- une *décision négative* de la forme $x \neq v$ qui garantit que x ne peut pas être affectée à v.

Une affectation $x \leftarrow v$ correspond alors simplement à une décision positive.

Certains algorithmes réalisent uniquement des décisions positives comme l'algorithme BT. Lorsque BT affecte la valeur v à x et qu'une incohérence est détectée, une nouvelle valeur v' sera choisie dans D_x et affectée à x. Tant qu'un échec est rencontré, une nouvelle valeur de x sera choisie. Lorsque x est affectée d'une façon cohérente et qu'une nouvelle variable est choisie, une nouvelle décision positive portant sur cette dernière est effectuée. En procédant ainsi, l'arbre de recherche construit est alors *d-aire*. Le branchement est dit *branchement non binaire*.

D'autres, au contraire, exploitent à la fois des décisions positives et des décisions négatives [Sabin and Freuder, 1994; Lecoutre et al., 2007e]. Dans ce cas, lorsqu'un échec est rencontré une fois une décision réalisée, la décision opposée sera effectuée. Ainsi, si x est initialement affectée à v, la décision opposée consiste à réaliser $x \neq v$ et vice-versa. Ensuite, une nouvelle décision est réalisée qui peut impliquer la même variable ou une variable différente. L'arbre de recherche construit dans ce cas est alors *binaire*. C'est ainsi que nous parlons de *branchement binaire*. Souvent, un ordre sur le type de décisions est spécifié au préalable. En particulier, dans [Sabin and Freuder, 1994], les décisions positives sont réalisées avant les décisions négatives. L'ensemble des décisions positives de Σ est noté $Pos(\Sigma)$. La réalisation d'une nouvelle décision est notée $\Sigma \cup \langle x = v \rangle$ s'il s'agit d'une décision positive et $\Sigma \cup \langle x \neq v \rangle$ sinon.

Nous pouvons constater que les algorithmes effectuant un branchement binaire sont plus généraux que ceux effectuant un branchement non binaire dans le sens où un algorithme effectuant un branchement binaire pourrait imiter le comportement de celui effectuant un branchement non binaire. Le contraire ne serait pas possible. Cependant, pour une variable x donnée et pour un ordre de choix de valeurs identique pour les deux algorithmes, l'algorithme suivant un branchement binaire ferait en général quasiment deux fois plus de décisions que celui opérant avec un branchement non binaire avant d'effectuer une affectation positive précise. Certains travaux [Mitchell, 2003; Hwang and Mitchell, 2005] ont montré qu'en théorie, les algorithmes exploitant un branchement binaire sont plus efficaces que leurs équivalents effectuant un branchement non binaire pour la résolution d'instances incohérentes. Toutefois, en pratique, la comparaison est plus délicate et dépend fortement de l'instance à résoudre.

Dans la partie suivante, nous nous intéressons à la notion de cohérence locale, aux algorithmes de filtrage chargés de maintenir un certain niveau de cohérence et à certains algorithmes employant la cohérence locale durant la résolution.

2.2.4.3 Cohérences locales et filtrage

La notion d'équivalence en CSP est souvent exploitée dans le but de transformer une instance CSP P en une instance CSP P' qui lui est équivalente tout en étant plus facile à résoudre. L'instance P' peut être, par exemple, obtenue à partir de P par réduction des domaines des variables en supprimant des valeurs ne pouvant pas participer à une solution. La taille de l'espace de recherche à explorer est alors réduite. Les techniques qui sont capables de réaliser de telles réductions sont appelées *techniques de filtrage*. Elles peuvent être utilisées en amont de la résolution - en phase de prétraitement - comme pendant la résolution. L'utilisation de telles techniques semble être cruciale vu la taille des instances à résoudre et, par voie de conséquence, la taille de l'espace de recherche à explorer qui est potentiellement exponentielle. Concrètement, les techniques de filtrage visent à maintenir une notion de *cohérence locale*. Une instance CSP P ne respectant pas cette notion est transformée via le filtrage en une instance CSP P' qui la respecte. Elle est dite locale par opposition au terme globale puisqu'elle ne concerne pas tout le CSP mais uniquement des sous-parties du CSP.

Suite à la définition de nombreuses propriétés de cohérence locale, Freuder donne une définition permettant d'en généraliser une partie d'entre elles dans [Freuder, 1978].

Définition 44 *Une instance CSP $P = (X, D, C)$ est k-cohérente ssi pour tout k-uplet de variables $(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$, pour toute affectation $\mathcal{A}$ cohérente des premières $k-1$ variables, il existe une valeur $v_{i_k} \in D_{x_{i_k}}$ telle que $\mathcal{A} \cup \{x_{i_k} \leftarrow v_{i_k}\}$ est également cohérente. Elle est fortement k-cohérente ssi, $\forall i, 1 \leq i \leq k$, P est i-cohérente.*

Notons que si une instance CSP est fortement n-cohérente, elle est *globalement cohérente* ce qui n'est pas le cas si nous nous contentons de la propriété de k-cohérence. La cohérence globale puise son intérêt dans le fait qu'une instance globalement cohérente peut être résolue en temps polynomial [Freuder, 1982]. Notons que la k-cohérence (forte) peut être généralisée par l'hyper k-cohérence (forte) [Jégou, 1993]. Au niveau du filtrage qui l'accompagne, dans [Cooper, 1989], l'auteur propose un algorithme général afin de maintenir la k-cohérence forte. Sa complexité temporelle et spatiale est en $O(n^k.d^k)$. Le filtrage par k-cohérence forte est capable de supprimer des valeurs et/ou des tuples ne participant pas à des solutions. Elle n'est cependant pas la seule à pouvoir supprimer des tuples [Janssen et al., 1989]. Un point important à clarifier est que le filtrage par k-cohérence ($k > 2$) est susceptible d'altérer la structure du problème en lui rajoutant explicitement de nouvelles contraintes. Non seulement la structure est modifiée, mais l'ajout de contraintes a des conséquences néfastes sur le temps d'exécution ainsi que sur l'espace mémoire requis. Il en découle, en plus de la complexité en $O(n^k.d^k)$, que le filtrage peut être délicat à mettre en œuvre. Ceci explique la limitation habituelle à de faibles niveaux de cohérence. En effet, souvent nous parlons principalement de cohérence d'arc (2-cohérence pour les CSP binaires) [Ullmann, 1966; Waltz, 1972; Montanari, 1974; Mackworth, 1977] ou de cohérence de chemin (3-cohérence pour les CSP binaires) [Montanari, 1974]. De nombreux algorithmes de maintien de cohérence de chemin ont été proposés [Mackworth, 1977; Han and Lee, 1988; Chmeiss and Jégou, 1998; Bessière et al., 2005; Lecoutre et al., 2007b,a, 2011]. Cependant, elle reste utilisée principalement en prétraitement. Par la suite, nous nous focalisons sur la cohérence d'arc.

Cohérence d'arc La cohérence d'arc est la plus ancienne des cohérences locales et celle qui reste l'une des plus utilisées de nos jours. Elle a été introduite initialement dans [Ullmann, 1966; Waltz, 1972; Montanari, 1974; Mackworth, 1977]. Avant de définir la cohérence d'arc, nous nous focalisons sur la distinction entre tuple *valide*, *support* et *conflit* Cette distinction est essentielle pour la compréhension des aspects dynamiques de certains algorithmes. Soit t_Y un tuple de $|Y|$ valeurs correspondant aux variables appartenant à un sous-ensemble $Y \subseteq X$ de variables. Si $x_i \in Y$, $t_Y[x_i]$ désigne la valeur associée à x_i dans t_Y. Nous notons $D_{x_i}^{curr}$ le domaine courant de la variable x_i. Contrairement à D_{x_i} qui contient toutes les valeurs initialement présentes dans le domaine de x_i, $D_{x_i}^{curr}$ contient uniquement les valeurs restantes suite à la suppression des autres valeurs par filtrage.

Définition 45 *Soit $c_i \in C$. Un tuple $t_{S(c_i)}$ est* valide *pour c_i ssi $\forall x \in S(c_i), t_{S(c_i)}[x] \in D_x^{curr}$.*

Un tuple valide peut être un tuple support ou conflit.

Définition 46 *Un tuple $t_{S(c_i)}$* support *est un tuple valide pour c_i qui appartient à $R(c_i)$. Un tuple valide pour c_i n'appartenant pas à $R(c_i)$ est un tuple* conflit.

Nous définissons maintenant la notion de cohérence d'arc.

Définition 47 *Une instance CSP $P = (X, D, C)$ est arc-cohérente ssi pour toute variable $x_i \in X$, nous avons $D_{x_i} \neq \emptyset$ et $\forall v_i \in D_{x_i}$, $\forall c_j$ avec $x_i \in S(c_j)$, il existe un tuple t support de c_j tel que $t[x_i] = v_i$.*

En d'autres termes, si une valeur v_i de x_i n'apparaît dans aucun tuple support d'une contrainte c_j à laquelle elle est liée, cela signifie que v_i ne peut participer à aucune solution de P. Ainsi, v_i n'admet pas de *supports* dans c_j. Notons qu'une instance CSP arc-cohérente n'est pas forcément cohérente.

Filtrage par cohérence d'arc Une instance CSP P qui n'est pas arc-cohérente peut être transformée en une instance CSP P' arc-cohérente grâce au filtrage en supprimant toute valeur qui n'ait pas de supports pour une contrainte à laquelle la variable en question est liée (sauf si P contient une variable dont le domaine est vide). Il est à noter que l'instance CSP P' résultant est unique et s'appelle la *fermeture d'arc-cohérence* du problème. Lorsqu'une valeur v_i de x_i est supprimée, cette suppression doit être *propagée* sur d'autres variables liées via une contrainte à x_i. En effet, des valeurs de ces variables peuvent perdre leurs supports suite à la suppression de v_i. Lorsque toutes les valeurs du problème possèdent un tuple support pour chaque contrainte, l'algorithme se termine. La version primitive AC-1 est donnée dans [Rosenfeld et al., 1976]. Depuis les algorithmes de filtrage par cohérence d'arc n'ont pas cessé d'être améliorés conduisant à une multitude d'algorithmes comme : AC-2 et AC-3 [Mackworth, 1977], AC-4 [Mohr and Henderson, 1986], AC-5 [Hentenryck et al., 1992], AC-6 [Bessiere, 1994], AC-8 [Chmeiss and Jégou, 1998], AC-2001 [Bessière and Régin, 2001], AC-3.1 [Zhang and Yap, 2001], AC-3.2, AC-3.3 [Lecoutre et al., 2003], AC-3^r [Lecoutre et al., 2007c] (AC-3 avec des résidus unidirectionnels), AC-3_{dl} et AC-3_{ds} [Mehta and van Dongen, 2004] ... En pratique, AC-3^{rm} [Lecoutre et al., 2007c] (AC-3 avec des résidus multi-directionnels) et AC-3^{rm+bit} [Lecoutre and Vion, 2008] semblent être parmi les plus compétitives [Lecoutre, 2013]. Les différences entre les algorithmes se situent principalement au niveau des structures de données et des mécanismes employées pour réaliser un tel filtrage. De telles différences sont répercutées au niveau des complexités temporelles et spatiales ainsi qu'au niveau de l'efficacité pratique. Par exemple, dans le cas des instances binaires, la complexité en temps de AC-4 [Mohr and Henderson, 1986] est en $O(m.d^2)$. Notons que AC-4 constitue la première version de complexité optimale dans le pire des cas. AC-2001 a la même complexité en temps avec une complexité en espace en $O(m.d)$. Le coût généralement raisonnable du filtrage par cohérence d'arc lui permet d'être appliqué en prétraitement, mais aussi durant la résolution pour maintenir la cohérence entre l'affectation courante et la partie restante non affectée du problème. Notons finalement qu'il existe d'autres notions de cohérences locales dont le maintien permet souvent de supprimer plus de valeurs que la cohérence d'arc au détriment de la complexité et de l'efficacité en temps comme SAC [Debruyne and Bessiere, 1997b], PIC [Freuder and Elfe, 1996] ou Max-RPC [Debruyne and Bessiere, 1997a].

Les derniers travaux de recherche autour du filtrage se focalisent essentiellement sur les algorithmes de filtrages spécialisés pour un type de contrainte précis comme les contraintes globales et les contraintes Table (contraintes exprimant explicitement les supports ou les conflits). Par exemple, des algorithmes de filtrage spécifiques ont été fournis pour plusieurs contraintes globales permettant de considérer ses différentes variables et de supprimer les valeurs incohérentes plus efficacement que les algorithmes de filtrage généraux [Bessiere et al., 2007; Samer and Szeider, 2011]. Un tel algorithme de filtrage est appelé *propagateur*. La contrainte globale *AllDifferent* [Régin, 1996] admet comme d'autres contraintes globales des propagateurs efficaces. Une description détaillée de plusieurs contraintes globales et de leurs propagateurs peut être trouvée dans le catalogue des contraintes globales disponible en ligne [Beldiceanu et al., 2007]. Des propagateurs pour les contraintes Table peuvent être trouvés dans [Bessiere and Régin, 1997; Lhomme and Régin, 2005; Lecoutre and Szymanek, 2006; Gent et al., 2007; Ullmann, 2007; Lecoutre, 2011; Lecoutre et al., 2015; Mairy et al., 2014; Perez and Régin, 2014; Wang et al., 2016; Demeulenaere et al., 2016].

Utilisation de la cohérence d'arc Le principal inconvénient de BT est l'exploration inutile de sous-espaces de recherche infructueux. L'utilisation du filtrage en prétraitement

et pendant la résolution contribue souvent à réduire significativement la taille de l'espace de recherche. Ce faisant, une incohérence peut être détectée plus tôt, et par voie de conséquence, la taille de l'arbre de recherche est réduite, le temps de résolution diminue et l'efficacité du solveur en question augmente. Pendant la résolution, le filtrage est réalisé à chaque nouvelle décision. Une décision possible $x_i = v_i$ réduit le domaine de la variable correspondante x_i à la valeur affectée v_i tandis que la décision $x_i \neq v_i$ supprime la valeur v_i du domaine de x_i. Certaines valeurs des variables futures (variables non encore instanciées) peuvent ainsi perdre leurs supports et doivent alors être supprimées. La suppression d'une valeur peut engendrer la suppression de nouvelles valeurs. La propagation garantit que les valeurs ne pouvant pas participer, d'après AC, à une solution sont supprimées. Si le domaine d'une variable devient vide alors l'affectation partielle courante n'admet aucune extension possible et un nouveau nœud fils est ainsi développé. Si aucun nœud fils ne peut être développé, l'algorithme effectue un backtrack. Notons que le filtrage réalisé est uniquement valable pour l'affectation partielle courante. Remettre en cause certaines affectations nécessite alors de restaurer les valeurs supprimées en fonction de celles-ci. Or, ceci peut impliquer la mise en place des mécanismes complexes et l'utilisation des structures de données spécialisées. L'association d'un algorithme de filtrage à BT possède alors à la fois des avantages et des inconvénients :

- ✓ les incohérences sont détectées au plus tôt,
- ✓ une affectation réalisée satisfait forcément toutes les contraintes du problème,
- ✓ la taille de l'arbre de recherche construit ainsi que le temps de résolution sont diminués en général,
- × le coût du maintien de la cohérence locale à chaque nœud est ajouté,
- × le coût du restauration des domaines de variables lors d'un backtrack est ajouté.

Malgré ce bilan mitigé, en pratique, l'apport du filtrage est incontestable. La configuration du solveur doit veiller à trouver un bon compromis entre puissance et coût de filtrage (en se limitant à la cohérence d'arc par exemple). Par la suite, nous nous intéressons à certains algorithmes de résolution exploitant le filtrage.

Nous commençons d'abord par l'algorithme Forward-Checking (FC) [Haralick and Elliott, 1980]. Lors de l'instanciation d'une variable x_i, l'application de FC engendre la suppression des valeurs qui sont incohérentes avec l'instanciation courante, des domaines des variables futures situées dans le voisinage de x_i. C'est ainsi que FC ne considère qu'un sous-ensemble des contraintes habituellement impliquées dans le maintien d'une cohérence d'arc et ce, à cause de la complexité [Bacchus and Grove, 1995]. Il s'agit ainsi d'une forme allégée de AC. Il a été initialement défini pour les instances CSP binaires et étendu ultérieurement au cas n-aire où il admet plusieurs versions nFC_i avec $i \in \{0, \ldots, 5\}$ [Bessière et al., 2002]. Ces différentes versions se distinguent par le sous-ensemble de contraintes qu'elles considèrent et la façon dont le filtrage par cohérence d'arc est appliqué. Dans le cas binaire, sa complexité est la même que celle de BT, à savoir $O(m.d^n)$, bien que son efficacité soit meilleure dans la plupart des cas en termes de nombre de nœuds de l'arbre de recherche, en nombre de tests de contraintes ou en temps. Dans le cas n-aire, la complexité dépend du nFC_i utilisé ; le plus puissant est nFC_5 qui a une complexité en $O(m.A.r.d^n)$ avec A le nombre maximal de tuples de toutes les contraintes. En raison de son efficacité irréprochable à l'époque, plusieurs variantes de FC ont été définies comme $FC\text{-}CBJ$ [Prosser, 1993], Minimal FC [Dent and Mercer, 1994], Minimal FC with backmarking and CBJ [Kwan and Tsang, 1996]... D'autres algorithmes ont été

Algorithme 2.2 : MAC (P, Σ, V)

Entrées : Une suite de décisions Σ, un ensemble de variables V
Entrées-Sorties : $P = (X, D, C)$: une instance CSP
Sorties : $vrai$ si Σ s'étend d'une façon cohérente aux variables de V, $faux$ sinon

```
1  si V = ∅ alors
2      retourner vrai
3  sinon
4      Choisir x ∈ V
5      Choisir v ∈ D_x
6      D_x ← D_x\{v}
7      si AC(P, Σ ∪ ⟨x = v⟩) ∧ MAC(P, Σ ∪ ⟨x = v⟩, V\{x}) alors
8          retourner vrai
9      sinon
10         si AC(P, Σ ∪ ⟨x ≠ v⟩ alors
11             retourner MAC(P, Σ ∪ ⟨x ≠ v⟩, V)
12         sinon
13             retourner faux
```

proposés comme MAC et RFL. Aujourd'hui, MAC et RFL sont les plus utilisés tandis qu'au contraire FC est beaucoup moins employé. Ceci est essentiellement dû aux progrès techniques qui ont permis d'introduire des structures de données pouvant être traitées moins lourdement qu'à l'époque de l'introduction de MAC et RFL. Ainsi, MAC et RFL appliquent un filtrage plus puissant que FC à un coût raisonnable.

Si le filtrage réalisé par FC se limite au voisinage de la variable x_i dernièrement instanciée, MAC [Sabin and Freuder, 1994] (algorithme 2.2) et RFL [Nadel, 1988] (algorithme 2.3) vont bien au-delà. Ils maintiennent durant la résolution une cohérence d'arc établie par la suppression des valeurs incohérentes et la propagation de l'effet de chaque suppression sans se limiter au voisinage de x_i. Notons que ce travail supplémentaire par rapport à FC peut être parfois coûteux, mais il permet cependant d'éviter l'exploration des sous-espaces de recherche infructueux et de réduire le nombre de nœuds de l'arbre de recherche développé. La seule différence entre MAC et RFL réside dans le type de branchement choisi [Sabin and Freuder, 1997]. En effet, MAC réalise un branchement binaire (lignes 7 et 10 de l'algorithme 2.2) tandis que RFL réalise un branchement non binaire (ligne 9 de l'algorithme 2.3). Dans les deux cas, AC est ensuite appliquée à l'instance P. Si le filtrage n'engendre pas de domaine vide, une nouvelle décision est faite. Lorsque tous les nœuds fils ont été explorés sans succès, un backtrack est réalisé (lignes 13 de l'algorithme 2.2 et ligne 11 de l'algorithme 2.3). Leur efficacité a remis en cause celle de FC et sont considérés aujourd'hui comme les méthode de référence des méthodes énumératives. Malgré cela, leur complexité reste en $O(r.m.d^n)$. En pratique, le choix entre RFL et MAC est toujours discutable. Selon l'instance traitée, l'un ou l'autre pourrait être plus efficace [Mitchell, 2003; Hwang and Mitchell, 2005]. La discussion rejoint celle faite dans la partie 2.2.4.2.

À ce point, un solveur peut employer un branchement binaire ou non binaire et choisit si un filtrage sera exploité et, dans le cas échéant, le niveau de cohérence locale souhaitée. Dans la partie suivante, nous détaillons la brique dédiée aux retours en arrière.

Algorithme 2.3 : RFL (P, Σ, V)

Entrées : Une suite de décisions positives Σ, un ensemble de variables V
Entrées-Sorties : $P = (X, D, C)$: une instance CSP
Sorties : $vrai$ si Σ s'étend d'une façon cohérente aux variables de V, $faux$ sinon

```
1  si V = ∅ alors
2      retourner vrai
3  sinon
4      Choisir x ∈ V
5      D'_x ← D_x
6      tant que D'_x ≠ ∅ faire
7          Choisir v ∈ D'_x
8          D'_x ← D'_x \ {v}
9          si AC(P, Σ ∪ ⟨x = v⟩) ∧ RFL(P, Σ ∪ ⟨x = v⟩, V \ {x}) alors
10             retourner vrai
11     retourner faux
```

2.2.4.4 Retour en arrière

Un algorithme énumératif combine exploration de nouveaux nœuds et retour sur des décisions déjà réalisées. Habituellement, lorsqu'il conclut que la branche courante ne peut pas mener à une solution, il modifie la dernière décision prise. Selon le type de branchement réalisé, il tente la décision opposée si le branchement est binaire ou une nouvelle décision qui correspond à une valeur différente du domaine de la variable courante si le branchement est d-aire. Si malgré cela, l'algorithme ne réussit toujours pas à étendre l'affectation courante, il remet en cause naturellement la décision qui précède la décision courante. Il s'agit du *retour en arrière chronologique*. Toutefois, revenir sur cette variable peut ne pas être toujours pertinent. Un exemple simple relève de la structure de l'instance en question représentée par l'(hyper)graphe de contraintes correspondant. Supposons que les sommets correspondants à la variable courante et à la variable précédente se trouvent dans deux composantes connexes différentes. Il est évident dans ce cas que changer la décision précédente n'aura aucun effet sur l'incohérence rencontrée au niveau de la variable courante. C'est ainsi qu'un nouveau type de retour-arrière est apparu appelé le *retour en arrière non chronologique*. Il vise à revenir sur une variable qui pourrait être en cause dans l'échec rencontré et à éviter l'exploration des sous-espaces de recherche non prometteurs. La décision portant sur cette variable est censée contribuer à l'explication de l'échec en question. Nous citons par exemple les travaux dans [Stallman and Sussman, 1977; Gaschnig, 1979; Dechter, 1990; Minton et al., 1992; Verfaillie, 1993; Schiex and Verfaillie, 1993; Prosser, 1993; Ginsberg, 1993; Chen, 2000; Jussien et al., 2000; Jlifi and Ghédira, 2003, 2004]. Toutefois, les retours en arrière non chronologiques engendrent un surcoût qui n'est pas toujours compensé par les économies faites. Notons aussi que l'identification des causes de l'échec devient plus difficile lorsque le filtrage est utilisé. Ainsi, l'utilisation de telles approches semble discutable et peut dépendre de l'instance à résoudre.

2.2.4.5 Enregistrements d'informations

Dans [Dechter, 1986] le mécanisme d'apprentissage apparaît pour la première fois sous la dénotation *Learning While Searching*. Les informations apprises visent à éviter certaines redondances dans la recherche et l'exploration de sous-espaces de recherche infructueux.

Ces informations sont donc produites, mémorisées et finalement réutilisées lors de l'exploration d'une autre région de l'espace de recherche [Dechter, 1986, 1990]. Dans la plupart des travaux, les informations enregistrées correspondent à des instanciations ne pouvant pas aboutir à une solution. Cette information s'appelle un *nogood*.

Définition 48 *[Lecoutre et al., 2007e] Soient $P = (X, D, C)$ une instance CSP et Δ un ensemble de décisions. Soit $P|\Delta$ l'instance CSP (X, D', C) induite par Δ telle que $D' = \{D'_{x_1}, \ldots, D'_{x_n}\}$ avec $D'_{x_i} = \{v_i\}$ si la décision positive $x_i = v_i$ est incluse dans Δ, $D'_{x_i} = D_{x_i} \backslash \{v_i\}$ si la décision négative $x_i \neq v_i$ apparaît dans Δ et $D'_{x_i} = D_{x_i}$ si aucune décision de Δ ne porte sur x_i. Δ est dit un* nogood *si l'instance $P|\Delta$ est incohérente.*

En particulier, une affectation correspondant à un tuple conflit est un nogood trivial. Un nogood peut être produit et mémorisé à chaque fois qu'une incohérence est détectée. Les différents nogoods se distinguent par leur pouvoir à élaguer des sous-arbres inutiles de l'arbre de recherche d'une taille plus ou moins importante. Les premières expérimentations ayant montré l'intérêt de ces enregistrements datent des années 90 [Dechter, 1990; Frost and Dechter, 1994; Schiex and Verfaillie, 1994a]. De nombreux travaux portant sur les enregistrements des nogoods peuvent être retrouvés notamment dans [Stallman and Sussman, 1977; Dechter, 1986; Ginsberg, 1993; Schiex and Verfaillie, 1994a,b; Frost and Dechter, 1994, 1995; Freuder and Wallace, 1995; Richards and Richards, 1996; Bayardo and Miranker, 1996; Jussien et al., 2000; Katsirelos and Bacchus, 2003, 2005; Lecoutre et al., 2007e]. Ces méthodes font face à l'enjeu de l'enregistrement massif d'informations (le nombre de nogoods étant potentiellement exponentiel) risquant ainsi de détériorer l'efficacité des méthodes de résolution. La tendance consiste alors à ne conserver que les informations les plus pertinentes [Schiex and Verfaillie, 1994a; Frost and Dechter, 1994; Bayardo and Miranker, 1996]. À titre d'exemple, les travaux de [Dechter, 1990] limitent la cardinalité des nogoods appris à une constante k arbitraire. C'est ainsi que le nombre de nogoods enregistrés devient plus faible résultant en une diminution de place mémoire occupée. Au final, ces techniques tentent de trouver le meilleur compromis entre coût et apport des enregistrements. La gestion du coût des enregistrements nécessite aussi de bien choisir les structures de données utilisées. C'est ainsi que dans [Lecoutre et al., 2007e] l'ensemble de nogoods est représenté par une contrainte globale exploitant la notion des « watched literals » [Moskewicz et al., 2001]. Cette notion a été introduite dans le cadre du problème de satisfiabilité booléenne (SAT) [Marques Silva et al., 2009]. Les enregistrements peuvent aussi servir à réaliser des retours en arrière non chronologiques judicieux. Par exemple, dans [Ginsberg, 1993], des explications sont mémorisées pour guider la recherche vers une affectation susceptible d'être à l'origine d'une incohérence rencontrée. Par ailleurs, le progrès le plus impressionnant en SAT est en partie dû à l'enregistrement des nogoods [Moskewicz et al., 2001] importé du cadre CSP au cadre SAT. Il a permis essentiellement de résoudre des instances de grande taille inaccessibles jusqu'alors. Notons finalement que les informations enregistrées ne se limitent pas à des nogoods. Par exemple, une affectation pouvant être étendue d'une façon cohérente sur une partie du problème peut être aussi enregistrée [Bayardo and Miranker, 1996; Baget and Tognetti, 2001; Jégou and Terrioux, 2003].

2.2.4.6 Heuristiques de choix de la prochaine variable / valeur

La brique qui a une influence prépondérante sur la recherche est le choix de la prochaine variable à instancier. En effet, nous comptons actuellement sur les heuristiques de choix de variables et/ou de valeurs afin de compenser une partie des lacunes des algorithmes de recherche. Cependant, trouver un ordre optimal est au moins aussi difficile que résoudre

le problème de satisfaction d'une instance [Liberatore, 2000]. C'est ainsi que des heuristiques sont essentiellement utilisées. Employer une heuristique d'ordonnancement efficace s'avère, de nos jours, primordial du fait de sa grande influence sur la performance des algorithmes en termes de nombre de nœuds visités et par conséquent en termes de temps d'exécution. Idéalement, un ordre de choix de variables sélectionne en premier les variables dont l'affectation rend le reste du problème plus facile à résoudre. Cette tendance se base sur le principe *first-fail* qui consiste à faire les choix les plus contraints en premier pour tenter de rencontrer l'échec le plus tôt possible [Haralick and Elliott, 1980]. Dans cette partie, nous regardons celles parmi les plus répandues.

Heuristiques de choix de variables statiques Ces heuristiques emploient le même ordre de choix de variables pendant la résolution, calculé en amont de celle-ci. Elles se basent uniquement sur l'état initial de la recherche. Le plus simple est d'ordonner les variables *lexicographiquement*. L'heuristique *max-deg* ordonne les variables par ordre de degré décroissant sachant que le degré d'une variable est le nombre de contraintes où elle apparaît [Ullmann, 1976; Dechter and Meiri, 1989]. Sa variante *ddeg* considère que le degré d'une variable est le nombre de variables voisines non encore choisies. En outre, l'heuristique de *dureté maximale (MT)* ordonne les variables par ordre décroissant de la somme des duretés des contraintes qui impliquent une variable. Notons que la dureté d'une contrainte est mesurée par le rapport $\frac{\#tuples\ interdits}{\#tuples\ possibles}$. L'inconvénient est qu'il n'est pas toujours facile d'obtenir une telle mesure lorsqu'il s'agit d'une contrainte exprimée en intention. Nous citons aussi l'heuristique *maximum cardinalité (MC)* [Dechter and Meiri, 1989] qui choisit la première variable aléatoirement et ensuite choisit celle liée au plus grand nombre de variables déjà ordonnées. Finalement, l'heuristique *min-width (MW)* [Freuder, 1982] ordonne les variables selon l'ordre inverse d'un ordre de largeur minimale. Ce dernier est calculé en choisissant à chaque étape la variable de degré minimum et en éliminant toutes les contraintes où elle est impliquée. Ces heuristiques sont généralement très peu efficaces parce qu'elles ne tiennent pas compte de l'état courant de la recherche. En particulier, si la cohérence locale est maintenue pendant la résolution, les changements induits par le filtrage ne sont pas pris en compte par une heuristique statique contrairement à une heuristique dynamique.

Heuristiques de choix de variables dynamiques Ces heuristiques sont dites dynamiques parce qu'elles changent généralement d'ordre de choix de variables pendant la résolution. Généralement, les heuristiques dynamiques sont notoirement plus efficaces que les heuristiques statiques [Dechter and Meiri, 1994]. L'heuristique *dom* [Golomb and Baumert, 1965; Bitner and Reingold, 1975; Haralick and Elliott, 1980] choisit d'abord les variables ayant le plus petit domaine. Les heuristiques *dom/deg* [Bessiere and Régin, 1996] et *dom/ddeg* [Bessiere and Régin, 1996] combinent les avantages de *dom* et de *(d)deg*. Elles choisissent la variable suivante ayant le plus petit ratio *dom/(d)deg* afin de de minimiser *dom* et maximiser *(d)deg*. Ces deux heuristiques sont caractérisées par leur simplicité et leur efficacité. Souvent, les heuristiques d'ordonnancement rencontrent des égalités de score de variables notamment au début de la recherche. Les heuristiques peuvent casser les égalités aléatoirement ou selon l'ordre lexicographique. Dans ce cas le nom de l'heuristique est suivi de *+ critère choisi pour casser les égalités*. D'autres heuristiques sont *dom + deg* [Frost and Dechter, 1995] ou *dom + ddeg (bz)* [Brélaz, 1979; Smith, 1999]. L'efficacité de ces heuristiques dépend de l'instance à résoudre. Généralement, les heuristiques *dom*, *bz* et *dom/ddeg* sont considérés comme les plus efficaces. [Bessière et al., 2001] proposent de généraliser les heuristiques en prenant en compte le voisinage de chaque variable. Cela

permet de rendre parfois les solveurs plus robustes. Cependant, ces heuristiques sont surpassées par les heuristiques dites *adaptatives*. D'ailleurs, les solveurs les mieux classés de la compétition 2008 [CP0, 2008] et de la compétition 2017 [XC1, 2017] utilisent tous une ou une combinaison de plusieurs heuristiques adaptatives.

Heuristiques de choix de variables adaptatives Ces heuristiques font des choix qui ne dépendent pas uniquement de l'état courant de la recherche mais aussi de ses états précédents. Ce faisant, ils font des choix plus éclairés et plus adaptés à la nature de l'instance. Parmi les plus connues, nous citons l'heuristique *dom/wdeg* [Boussemart et al., 2004], celle basée sur l'*impact* [Geelen, 1992; Refalo, 2004] ou celle basée sur l'activité d'une variable [Michel and Hentenryck, 2012]. Notons que l'heuristique *dom/wdeg* est la plus utilisée dans les solveurs qui ont participé à la compétition CSP 2017. Vu leur importance au niveau expérimental, elles seront abordées en détails dans le chapitre 4.

Finalement, dans [Lecoutre et al., 2009], les auteurs proposent une heuristique de choix de variables LC (pour *Last Conflict*) qui peut s'associer à d'autres heuristiques et qui prend en compte le dernier conflit. En effet, à chaque conflit, la dernière variable instanciée est mémorisée. Cette variable devient alors prioritaire. Lorsqu'elle est instanciée et qu'aucune incohérence n'est rencontrée, LC redonne la main à l'heuristique de choix de variables de base. Ce faisant, LC imite en quelque sorte les techniques de backjump.

Heuristiques de choix de valeurs L'impact des heuristiques de choix de valeurs a longtemps été considéré comme marginal vu que lorsqu'une instance est incohérente ou lorsque le but est de chercher toutes les solutions, toutes les valeurs doivent être considérées. Dans [Smith and Sturdy, 2005], les auteurs montrent que cet argument peut être valable pour un branchement d-aire mais pas pour un branchement binaire. En tous cas, la plupart des heuristiques de choix de valeurs visent à choisir d'abord les valeurs les plus prometteuses. Par exemple, la plus connue est l'heuristique *min-conflicts* [Minton et al., 1992; Frost and Dechter, 1995] qui choisit la valeur v qui a le plus petit nombre de conflits avec les valeurs des variables non encore instanciées.

Nous nous concentrons dans la partie suivante sur les techniques de redémarrage.

2.2.4.7 Redémarrage

Le redémarrage est une stratégie qui a été initialement proposée dans [Harvey, 1995; Gomes et al., 1998, 2000]. Il ne s'agit pas ici d'une technique découlant d'une idée complexe, bien au contraire, elle relève d'une idée naturelle tout en ayant un apport incontestable sur les performances des méthodes qui l'exploitent. Le redémarrage revient à faire un retour-arrière au niveau de départ de la décision. En d'autres termes, il consiste à désaffecter toutes les affectations réalisées dans le but d'éviter de passer beaucoup de temps dans des « mauvaises » branches. Ainsi, il permet de « corriger » le comportement des algorithmes de recherche. Par exemple, un redémarrage permet de choisir d'abord les variables les plus pertinentes selon l'heuristique *dom/wdeg* qui sera mieux renseignée au fil des redémarrages. Toutefois, il est crucial de ne pas s'engager, lors d'un redémarrage, dans des sous-espaces de recherche déjà explorés. C'est pourquoi, la *diversification* a été introduite. Elle est souvent assurée par la *randomisation* et/ou l'*apprentissage*. L'existence du phénomène *heavy-tailed* (décroissance exponentielle de la probabilité d'excéder le nombre de backtracks x en fonction de x) a permis de donner des éléments pour tenter d'expliquer le succès des redémarrages accompagnés de la randomisation ou de l'apprentissage. En

effet, ces deux techniques permettent de modifier la recherche et ainsi de tenter d'obtenir un « nouveau solveur » plus rapide que le dernier. C'est ainsi que la variabilité dans le comportement d'un solveur d'une instance à l'autre mais aussi pour une même instance qui permet d'obtenir une telle amélioration. En effet, il suffit parfois d'un changement minime au niveau du choix de la prochaine variable pour rendre le solveur efficace ou médiocre.

La randomisation peut être assurée via l'heuristique de choix de variables/valeurs par exemple. Quant à l'apprentissage, il peut être réalisé par le biais des *nogoods* [Lecoutre et al., 2007e]. L'enregistrement des nogoods a été pour la première fois utilisée pour améliorer la résolution des instances CSP dans [Dechter, 1990]. Les premières expérimentations ont été reportées dans [Dechter, 1990; Frost and Dechter, 1994; Schiex and Verfaillie, 1994a,b]. Les techniques de redémarrage ont été également utilisées dans le cadre de la résolution du problème SAT. Elles ont contribué fortement à l'essor des solveurs CDCL (pour *Conflict-Driven Clause Learning*) [Marques Silva et al., 2009]. L'idée est d'enregistrer un nogood lorsqu'une incohérence est rencontrée afin d'éviter ultérieurement de visiter des sous-espaces de recherche futiles. Les nogoods peuvent être aussi exploités autrement, pour identifier la partie de l'espace de recherche déjà exploitée. Ainsi, à chaque redémarrage, des nogoods correspondants à la dernière exécution sont enregistrées et garantissent que la partie de l'espace de recherche qui vient d'être visitée ne sera plus jamais explorée. Au fil des redémarrages, l'espace de recherche est ainsi de plus en plus réduit, ce qui garantit la *complétude* et la *terminaison*. Afin de réduire le nombre de nogoods enregistrés, il a été proposé d'enregistrer uniquement ceux qui correspondent à la dernière branche (la plus à droite) de la recherche pour MAC [Lecoutre et al., 2007e]. Cette idée s'appelle *search signature* [Baptista et al., 2001], *path recording* [Fukunaga, 2003] et *restart nogoods* [Lecoutre et al., 2007d,e]. Nous allons maintenant nous focaliser sur la notion des *nld-nogoods réduits* introduits dans [Lecoutre et al., 2007e].

Nld-nogoods réduits L'algorithme considéré est MAC [Sabin and Freuder, 1994] avec l'hypothèse qu'une décision positive précède une décision négative. Les auteurs définissent tout d'abord la notion de *nld-subsequence* pouvant être extrait de chaque décision négative.

Définition 49 *Soit $\Sigma =< \delta_1, \ldots, \delta_k >$ une suite de décisions et δ_j une décision négative ($1 \leq j \leq k$). Alors, la suite $< \delta_1, \ldots, \delta_j >$ ayant le même préfixe est une nld-subsequence de Σ.*

En se basant sur cette définition, ils définissent la notion de *nld-nogood* et de *nld-nogood réduit*.

Définition 50 *Soit Σ une suite de décisions prises sur une branche de l'espace de recherche (partant de la racine). Pour chaque nld-subsequence $\Sigma' =< \delta_1, \ldots, \delta_j >$ de Σ, l'ensemble $\Delta = \{\delta_i \in \Sigma : 1 \leq i < j\} \cup \{\neg\delta_j\}$ est un nogood généralisé de P, appelé un nld-nogood. L'ensemble $\Delta' = Pos(\Sigma') \cup \{\neg\delta_j\}$ est un nogood, appelé un nld-nogood réduit.*

Δ est un nogood puisque $\neg\delta_j$ (décision positive) a été testée auparavant sans succès. Il est dit généralisé puisqu'il contient des décision négatives. Le fait que Δ' soit un nogood a été démontré dans [Lecoutre et al., 2007e]. Par exemple, soit $\Sigma =< x_1 \neq v_1, x_3 = v_3, x_2 \neq v_2 >$ la suite de décisions la plus à droite. Les nld-subsequences, nld-nogoods et les nld-nogoods réduits sont comme montrés ci-dessous :

nld-subsequences	nld-nogoods	nld-nogoods réduits
$< x_1 \neq v_1 >$	$\{x_1 = v_1\}$	$\{x_1 = v_1\}$
$< x_1 \neq v_1, x_3 = v_3, x_2 \neq v_2 >$	$\{x_1 \neq v_1, x_3 = v_3, x_2 = v_2\}$	$\{x_3 = v_3, x_2 = v_2\}$

L'exploitation des redémarrages et des enregistrements des nld-nogoods par MAC permet d'obtenir l'algorithme $MAC{+}RST{+}NG$ [Lecoutre et al., 2007e]. En pratique, $MAC{+}RST{+}NG$ s'avère généralement être plus efficace que MAC. En ce qui concerne la gestion des enregistrements, la représentation des nld-nogoods réduits sous la forme d'une contrainte globale et l'utilisation de la notion de *watched literals* [Moskewicz et al., 2001] permet de les exploiter efficacement et à un coût raisonnable. En effet, les similitudes qui existent entre les différents nld-nogoods réduits extraits lorsqu'un redémarrage survient ont permis de définir la notion des *increasing-nogoods* [Lee and Zhu, 2014; Lee et al., 2016]. Ces derniers permettent ainsi de représenter les nld-nogoods réduits extraits à chaque lancer sous forme d'une seule contrainte globale. L'objectif d'une telle représentation est la compacité et l'augmentation de leur pouvoir de filtrage par rapport à leur capacité de filtrage lorsqu'ils sont traités indépendamment. Dans [Glorian et al., 2017], les auteurs proposent plusieurs techniques de simplification et de combinaison d'increasing-nogoods afin de permettre un meilleur élagage de l'arbre de recherche. Ces techniques permettent essentiellement de traiter les nogoods en question tout en considérant d'autres informations disponibles comme celles correspondant aux domaines des variables. En pratique, les auteurs [Glorian et al., 2017] montrent que ces techniques ont permis pour une grande sélection d'instances (provenant des compétitions CSP 2006 et 2008) d'augmenter le nombre de valeurs supprimées par les algorithmes de filtrage utilisés moyennant une augmentation au niveau de la complexité et une diminution de l'efficacité de l'heuristique de choix de variables dom/wdeg basée sur les conflits.

Stratégies de redémarrage La question principale à laquelle tente de répondre les stratégies de redémarrage est : quand interrompre la recherche ? Les *stratégies de redémarrage* sont caractérisés par le *critère* sur lequel elles se basent ainsi que sur le *seuil* à partir duquel elles déclenchent le redémarrage. L'algorithme peut compter sur l'augmentation progressive de la limite du temps à partir de laquelle un redémarrage survient pour garantir la complétude et la terminaison. Le critère souvent utilisé est le nombre de backtracks. D'autres critères sont utilisés en SAT comme le nombre de contraintes apprises (clauses), leur taille ou l'agilité des variables (mesure de diversité des affectations des variables) (voir [Pipatsrisawat and Darwiche, 2009; Biere, 2008] par exemple). Nous mentionnons ci-dessous quelques types de redémarrages parmi les plus connus :

- Arithmétique (ou fixé) : un redémarrage est déclenché dès que le *critère* dépasse le seuil x. x est incrémenté de y à chaque redémarrage. Si $y = 0$, la politique est à seuil fixé. Lorsque la distribution du temps d'exécution est connue, les auteurs de [Luby et al., 1993] démontrent qu'une politique à seuil fixé est optimale.

- Géométrique [Walsh, 1999] : un redémarrage est déclenché dès que le *critère* dépasse le seuil x. x est multiplié par y ($y > 1$) à chaque redémarrage. En augmentant x géométriquement, cette politique tente de s'approcher de la valeur optimale du seuil après un nombre limité de redémarrages.

- Luby et al. [Luby et al., 1993] : Les redémarrages sont déclenchés selon la série des nombres : $1, 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1 \ldots$ multipliés par une constante x. Formellement, soit t_i le i-ème nombre de la série. Alors t_i est défini par :

$$t_i = \begin{cases} 2^{k-1} & si\ \exists k \in \mathbb{N} : i = 2^k - 1 \\ t_{i-2^{k-1}+1} & si\ \exists k \in \mathbb{N} : 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

  Cette politique a montré un intérêt théorique pour les algorithmes de type Las Vegas.

Il est à préciser que ce n'était pas dans le cadre CSP. Au-delà, elle s'est avérée efficace en pratique sans explication théorique de sa bonne performance.

Il existe d'autres stratégies de redémarrages qui n'impliquent pas forcément un backtrack jusqu'au niveau initial de décision comme celle de [Lynce et al., 2001]. En outre, il a été aussi proposé de considérer des critères locaux, c'est-à-dire relatifs à une branche par exemple comme dans [Ryvchin and Strichman, 2008].

Nous nous focalisons maintenant sur les méthodes structurelles.

2.2.5 Résolution avec exploitation de la structure

La NP-complétude du problème CSP a conduit à identifier des classes traitables en temps polynomial appelées les *classes polynomiales*. Ces travaux proviennent notamment de la communauté de l'intelligence artificielle et celle des bases de données (vu l'équivalence entre le problème CSP et certains problèmes de base de données [Bibel, 1988; Janssen et al., 1989; Dechter, 1992; Gyssens et al., 1994; Kolaitis and Vardi, 1998]) .

Définition 51 *Une* classe polynomiale *du problème CSP est un ensemble d'instances pour lequel il existe un algorithme capable de résoudre ses instances en temps polynomial.*

Notons que dans certains travaux [Gottlob et al., 2000], décider si une instance appartient à un ensemble d'instances donné en temps polynomial, est requis pour définir une classe polynomiale. Les différentes approches qui ont réussi à mettre en évidence des classes traitables peuvent être divisées en plusieurs catégories [Pearson and Jeavons, 1997] selon la nature des restrictions appliquées aux instances. Dans ce manuscrit, nous nous intéressons uniquement aux restrictions appliquées à la structure. Dans ce cas, les différentes classes polynomiales sont identifiées uniquement sur la base de la structure de la portée des contraintes comme dans [Freuder, 1982]. Il existe de nombreux travaux qui proposent des classes polynomiales en se basant sur différentes propriétés structurelles relatives à l'(hyper)graphe de contraintes correspondant à l'instance CSP. L'une des premières classes polynomiales structurelles proposée est celle des instances CSP binaires dont le graphe de contraintes est acyclique appelée la *classe TREE* [Freuder, 1982].

Définition 52 *La* classe TREE *est l'ensemble des instances CSP binaires ayant un graphe de contraintes acyclique.*

Notons que pour les instances appartenant à cette classe, maintenir la cohérence d'arc suffit pour résoudre le problème [Freuder, 1982]. La généralisation de la notion de l'acyclité des graphes au cas des hypergraphes a conduit à introduire de nouvelles classes polynomiales relatives aux instances CSP n-aires dont celle définie sur la base de l'α-acyclicité [Beeri et al., 1983] dans [Janssen et al., 1989] :

Définition 53 *La* classe α-ACYCLIC *est l'ensemble des instances CSP ayant un hypergraphe de contraintes α-acyclique.*

Malheureusement, toutes les instances CSP ne peuvent pas être représentées par un (hyper)graphe de contraintes acyclique. C'est ainsi qu'une généralisation s'intéresse à mesurer le degré de cyclicité de leur (hyper)graphe de contraintes en termes de largeur comme cela a été détaillé dans le chapitre 1. L'objectif est alors de se ramener à une instance CSP acyclique en exploitant ses propriétés structurelles. Plusieurs notions de largeurs ont été données accompagnant chacune une notion de décomposition différente comme dans [Even, 1979; Gyssens and Paredaens, 1982; Robertson and Seymour, 1986; Dechter, 1992; Gottlob et al., 1999; Courcelle and Olariu, 2000; Oum and Seymour, 2006; Cohen et al., 2008;

Grohe and Marx, 2014]. Nous nous intéressons particulièrement ici à la décomposition arborescente [Robertson and Seymour, 1986].

Selon [Aschinger et al., 2011], la décomposition arborescente est principalement utilisée pour prouver qu'un problème est traitable en temps polynomial pour des instances de tree-width bornée. Elle peut être également utilisée comme outil afin de prouver la traitabilité en temps polynomial en général d'une instance quelconque. Formellement, la décomposition arborescente a permis de définir une nouvelle classe polynomiale : la *classe* BTW_k.

Définition 54 *Soit un entier k. La* classe BTW_k *est l'ensemble des instances dont l'hypergraphe de contraintes a une largeur arborescente bornée par k.*

Les instances appartenant à cette classe peuvent être résolues en $O(n.d^{k+1})$ [Freuder, 1990] en utilisant des algorithmes comme BTD [Jégou and Terrioux, 2003]. De telles méthodes ont clairement un intérêt théorique majeur par rapport aux méthodes énumératives classiques notamment lorsque $k \ll n$. Dans [Freuder, 1982], il a été montré qu'étant donnée une instance CSP de largeur arborescente w, maintenir la cohérence forte $(w+1)$-cohérence (directionnelle qui suppose un ordre sur les variables) permet de résoudre le problème sans retour en arrière.

Dans cette partie, nous nous intéressons aux méthodes structurelles de résolution d'instances CSP. En particulier, nous nous focalisons sur la méthode énumérative BTD qui exploite la notion de la décomposition arborescente. Au-delà, nous évoquons les limitations de telles méthodes.

2.2.5.1 Backtracking on tree-decomposition (BTD)

BTD détaillé BTD [Jégou and Terrioux, 2003] se base sur une décomposition arborescente de l'hypergraphe de contraintes. BTD exploite un algorithme énumératif allant d'un simple BT à des algorithmes plus sophistiqués comme MAC ou RFL, qui sera guidée par cette décomposition. Soient P une instance CSP à résoudre et (E, T) une décomposition arborescente de son (hyper)graphe de contraintes. Un ordre $<$ valide sur les clusters est un ordre compatible avec un parcours en profondeur d'abord de la décomposition à partir d'un cluster racine E_r. Le parcours de la décomposition se fait alors depuis la racine vers les feuilles contrairement à d'autres approches de la programmation dynamique qui suivent une approche *bottom-up* (depuis les feuilles vers la racine) comme dans [Dechter and Pearl, 1989]. Par exemple, un ordre correspondant à la numérotation des clusters choisie sur l'exemple de la figure 1.12 est un ordre valide. Le cluster racine est alors le cluster E_0. Si l'ordre de choix de variables peut être quelconque pour des méthodes énumératives, ce n'est pas le cas pour BTD. En effet, la décomposition induit un ordre partiel d'instanciation de variables.

Définition 55 *Un ordre $\preceq_X$ sur les variables de X tel que $\forall x, y$ avec $x \in E_i$ et $y \in E_j$ tel que $E_i < E_j$, $x \preceq_X y$ est un ordre d'instanciation compatible.*

Sur l'exemple de la figure 1.12, l'ordre $x_1 \preceq_X x_3 \preceq_X x_4 \preceq_X x_2 \preceq_X x_6 \preceq_X x_7 \preceq_X x_9 \preceq_X x_{10} \preceq_X x_5 \preceq_X x_8 \preceq_X x_{11}$ est un ordre d'instanciation compatible. L'ordre d'instanciation induit par la décomposition est partiel puisque :

- le choix du cluster racine E_r peut changer,
- le choix du prochain cluster fils E_j à visiter parmi les clusters fils d'un cluster E_i peut aussi changer,

- le choix de la prochaine variable au sein d'un cluster est libre.

Ces choix s'avèrent être stratégiques pour garantir une résolution efficace vu l'influence de l'ordre de choix de variables sur la résolution. La résolution d'un cluster se fait par le biais de n'importe quel algorithme énumératif ne mettant pas en danger la validité de la décomposition. En effet, si une cohérence locale ajoutant de nouvelles contraintes est maintenue à chaque nœud, ces dernières risquent d'avoir des portées non incluses dans aucun cluster de la décomposition. Bien sûr, cela est particulièrement problématique pour la décomposition considérée. Ainsi, BTD peut tirer profit de MAC [Sabin and Freuder, 1994] ou RFL [Nadel, 1988] qui exploite la liberté laissée par BTD à l'heuristique de choix de variables au sein d'un cluster.

Selon la définition 55, si $E_j \in Fils(E_i)$ alors l'ensemble des variables des clusters de $Desc(E_j)$ (notées $V_{Desc(E_j)}$), exceptées celles de $E_i \cap E_j$, devront être instanciées après celles de E_i. La propriété essentielle relative à la décomposition arborescente est que l'affectation de $E_i \cap E_j$ sépare le problème initial en au moins deux sous-problèmes pouvant être résolus indépendamment. Le premier est constitué des variables de $V_{Desc(E_j)}$ et des contraintes impliquant au moins une variable de $V_{Desc(E_j)} \backslash (E_i \cap E_j)$. Les autres problèmes portent sur le reste de variables et des contraintes les impliquant. Le premier problème est ensuite résolu. À ce niveau deux cas se présentent :

- Dans le premier cas, aucune extension cohérente de l'affectation courante n'est trouvée sur $V_{Desc(E_j)}$. L'incohérence ne peut être expliquée que par une contrainte dont les variables sont toutes dans $V_{Desc(E_j)} \backslash (E_i \cap E_j)$ ou dont une partie est présente dans $V_{Desc(E_j)} \backslash (E_i \cap E_j)$ et l'autre dans $E_i \cap E_j$. Il convient alors de changer l'affectation des variables de $E_i \cap E_j$ puisque toutes les affectations de $V_{Desc(E_j)} \backslash (E_i \cap E_j)$ ont mené à un échec. D'ailleurs, la compatibilité des instanciations de $V_{Desc(E_j)}$ et des autres variables du problème passe toujours via ce séparateur (le théorème 1 dans [Jégou and Terrioux, 2003] garantit qu'il n'y a aucune contrainte contenant à la fois une variable de $V_{Desc(E_j)} \backslash (E_i \cap E_j)$ et une variable de $X \backslash V_{Desc(E_j)}$). Ce constat important permet de déduire que le changement de l'affectation courante qui ne modifie pas l'affectation de $E_i \cap E_j$ ne permettra jamais d'avoir une extension cohérente sur $V_{Desc(E_j)}$. Ainsi, $\mathcal{A}[E_i \cap E_j]$ peut être enregistré comme un *nogood structurel*.

- Dans le deuxième cas et dans le même esprit, si l'affectation courante admet une extension sur $V_{Desc(E_j)}$ alors une nouvelle affectation $\mathcal{A}'$ telle que $\mathcal{A}'[E_i \cap E_j] = \mathcal{A}[E_i \cap E_j]$ admettra également une extension sur $V_{Desc(E_j)}$. Par conséquent, l'affectation $\mathcal{A}[E_i \cap E_j]$ est enregistrée comme *good structurel*.

Formellement, nous rappelons la définition suivante [Jégou and Terrioux, 2003] :

Définition 56 *Soit E_i un cluster et E_j un de ses clusters fils. Un good structurel (resp. nogood) de E_i vis-à-vis de E_j est une affectation cohérente $\mathcal{A}[E_i \cap E_j]$ telle qu'il existe (resp. n'existe pas) une extension cohérente de $\mathcal{A}$ sur $V_{Desc(E_j)}$.*

Ainsi, lorsque BTD instancie les variables de E_i, pour chaque cluster fils E_j de E_i nous obtenons un sous-problème indépendant dont la racine est le cluster fils E_j et qui dépend uniquement de l'affectation $\mathcal{A}[E_i \cap E_j]$. Ce sous-problème sera noté $P_j|\mathcal{A}[E_i \cap E_j]$ ou simplement P_j lorsqu'il n'y a pas d'ambiguïté. Les enregistrements réalisés par rapport à ces séparateurs permettent à BTD d'éviter de résoudre plusieurs fois le même sous-espace de recherche. Lorsqu'un cluster fils E_j est désigné, si $\mathcal{A}[E_i \cap E_j]$ est un good de E_i vis-à-vis de E_j, la recherche effectuera un *saut* et passera à un nouveau sous-problème car nous

savons que $\mathcal{A}[E_i \cap E_j]$ admet une extension cohérente sur P_j. S'il s'agit au contraire d'un nogood, un *élaguage* du sous-arbre de recherche est réalisé et l'affectation de $E_i \cap E_j$ est modifiée.

L'algorithme 2.4 présente une extension de BTD basée sur MAC en raison de son efficacité par rapport à BT. Nous ne nous intéressons pas à ce stade aux lignes 25-28 ni au booléen *inconnu* retourné par $BTD\text{-}MAC$. Cet algorithme prend en entrée une suite de décisions Σ, le cluster courant E_i de E et l'ensemble des variables non encore affectées V_{E_i} de E_i. Il mémorise l'ensemble des goods et nogoods respectivement dans G^d et N^d. Si $V_{E_i} \neq \emptyset$, le comportement de $BTD\text{-}MAC$ est identique à MAC au sein du cluster E_i. Il choisit librement la prochaine variable à instancier parmi les variables de V_{E_i} jusqu'à ce qu'il n'y ait plus de variables à affecter. Notons qu'initialement $V_{E_i} = E_i \backslash (E_i \cap E_{p(i)})$. En effet, nous ne considérons que les variables propres de E_i i.e. les variables appartenant à E_i mais pas à $E_{p(i)}$ vu que les variables de $E_i \cap E_{p(i)}$ sont déjà affectées. Lorsque $V_{E_i} = \emptyset$, $BTD\text{-}MAC$ traite les fils de E_i les uns après les autres (lignes 19-33). Trois cas se présentent selon la nature de $\mathcal{A}[E_i \cap E_j]$:

- si elle est un nogood de E_i vis-à-vis de E_j, $BTD\text{-}MAC$ ne visite pas le problème $P_j|\mathcal{A}[E_i \cap E_j]$ et arrête la recherche sur ce sous-problème incohérent et retourne *faux* (lignes 7-8).
- si elle est un good de E_i vis-à-vis de E_j, $BTD\text{-}MAC$ ne visite pas le problème $P_j|\mathcal{A}[E_i \cap E_j]$ et passe au sous-problème suivant.
- si elle n'est ni l'un, ni l'autre, $BTD\text{-}MAC$ explore le sous-problème en question (ligne 11) et enregistre un nogood s'il est incohérent en retournant *faux* (lignes 15-16) ou un good s'il est cohérent en retournant *vrai* (ligne 12-13).

Il a été démontré que $BTD\text{-}MAC$ est complet, correct et termine [Jégou and Terrioux, 2003] et que sa complexité est la suivante :

Théorème 4 *$BTD\text{-}MAC$ a une complexité en temps en $O(n.s^2.m.log(d).d^{w^+ + 2})$ et en espace en $O(n.s.d^s)$.*

Par conséquent, BTD a une complexité en temps bien meilleure que celle des méthodes énumératives notamment lorsque $w^+ \ll n$ au détriment d'une complexité spatiale plus élevée.

BTD : avantages et inconvénients La comparaison de BTD (ou de ses extensions) à des méthodes énumératives classiques met en avant à la fois ses avantages et ses inconvénients. Tout d'abord, sur le plan théorique, nous constatons que la complexité temporelle est clairement améliorée notamment lorsque w^+ est significativement plus petit que n. Cela dépend principalement de la tree-width de l'instance elle-même (car $w \leq w^+$) et de la qualité de l'estimation w^+ faite pour w. En pratique, grâce aux enregistrements réalisés par BTD, nous évitons de visiter inutilement le même sous-espace de recherche plusieurs fois. Cependant, les méthodes énumératives s'avèrent plus libres au niveau du choix de la prochaine variable tandis que le choix de variables est partiellement induit par la décomposition dans le cas de BTD. Cet ordre de choix de variables peut être trop restrictif allant jusqu'à un ordre totalement statique. Vu l'apport considérable des heuristiques de choix de variables dynamiques par rapport à celles qui sont statiques, les restrictions imposées à l'heuristique de choix de variables par la décomposition peuvent être malheureusement très pénalisantes pour BTD. En outre, la complexité spatiale peut être handicapante. En effet, une taille maximale de séparateurs s trop importante pourrait engendrer

Algorithme 2.4 : BTD-MAC $(P, \Sigma, E_i, V_{E_i}, G^d, N^d)$

Entrées : Σ : suite de décisions ; E_i : cluster ; V_{E_i} : ensemble de variables
Entrées-Sorties : $P = (X, D, C)$: instance CSP, G^d : ensemble de goods ; N^d : ensemble de nogoods
Sorties : *vrai* si une solution au sous-problème de P enraciné en E_i et induit par Σ a été trouvée, $faux$ s'il est prouvé qu'il n'en possède pas, *inconnu* sinon

```
1  si V_Ei = ∅ alors
2      résultat ← vrai
3      Q_Ei ← Fils(E_i)
4      tant que résultat ∉ {faux, inconnu} et Q_Ei ≠ ∅ faire
5          Choisir un cluster E_j ∈ Q_Ei
6          Q_Ei ← Q_Ei \ {E_j}
7          si Pos(Σ)[E_i ∩ E_j] est un nogood dans N^d alors
8              résultat ← faux
9          sinon
10             si Pos(Σ)[E_i ∩ E_j] n'est pas un good de E_i par rapport à E_j dans G^d
                alors
11                 résultat ← BTD-MAC(P,Σ,E_j, E_j\(E_i ∩ E_j),G^d,N^d)
12                 si résultat= vrai alors
13                     Enregistrer Pos(Σ)[E_i ∩ E_j] comme good de E_i par rapport à E_j
                        dans G^d
14                 sinon
15                     si résultat= faux alors
16                         Enregistrer Pos(Σ)[E_i ∩ E_j] comme nogood de E_i par rapport
                            à E_j dans N^d
17     retourner résultat
18 sinon
19     Choisir une variable x ∈ V_Ei
20     Choisir une valeur v ∈ D_x
21     D_x ← D_x \ {v}
22     si AC (P,Σ ∪ ⟨x = v⟩) alors
23         résultat ← BTD-MAC(P,Σ∪ ⟨x = v⟩,E_i, V_Ei\{x},G^d,N^d)
24     sinon résultat ← faux;
25     si résultat = faux alors
26         si redémarrage alors
27             Enregistrer nld-nogoods par rapport à la suite de décisions Σ[E_i]
28             retourner inconnu
29         sinon
30             si AC (P,Σ ∪ ⟨x ≠ v⟩) alors
31                 retourner BTD-MAC(P,Σ ∪ ⟨x ≠ v⟩,E_i, V_Ei,G^d,N^d)
32             sinon retourner faux;
33     sinon retourner résultat;
```

une croissance rapide en besoin d'espace mémoire jusqu'à rendre certaines décompositions inexploitables. Un compromis temps/espace est incontournable. Pour réduire, le besoin en espace mémoire [Dechter, 1996; Dechter and Fattah, 2001] propose de fusionner ensemble

Algorithme 2.5 : BTD-MAC+RST (P)

Entrées : $P = (X, D, C)$: CSP
Sorties : $vrai$ si P possède une solution, $faux$ sinon

```
1 G^d ← ∅ ; N^d ← ∅
2 répéter
3 |   Choisir un cluster racine E_r
4 |   résultat ← BTD-MAC (P,∅,E_r,E_r,G^d,N^d)
5 jusqu'à résultat ≠ inconnu;
6 retourner résultat
```

deux clusters ayant une intersection de grande taille. Nous diminuons ainsi la taille du plus grand séparateur au détriment d'obtenir de plus grand clusters et par conséquent augmenter la complexité en temps. Bien que cette opération ne semble pas bénéfique du point de vue théorique, elle est souvent désirable en pratique [Jégou et al., 2005]. Elle permet non seulement de se débarrasser des séparateurs de grande taille mais aussi d'offrir davantage de liberté quant au choix de la prochaine variable à instancier. Au-delà des valeurs de w^+ et s, d'autres caractéristiques de la décomposition semblent jouer un rôle clé dans l'efficacité de la résolution. Par exemple, dans [Jégou and Terrioux, 2014b], les auteurs ont souligné que les décompositions qui assurent la connexité des clusters permettent une meilleure résolution. Ainsi, le calcul de la décomposition doit aussi être fait avec précaution. Selon ce qui précède, nous constatons que l'exploitation de la décomposition ajoute un certain nombre de paramétrages au solveur comme montré dans la figure 2.3 qui sont les suivants :

- le calcul de la décomposition,
- le choix du cluster racine,
- le choix du prochain cluster à traiter parmi les clusters fils du cluster courant.

À ces paramètres, s'ajoutent bien sûr les heuristiques de choix de variables utilisées au sein de chaque cluster.

Exploitation des redémarrages et de l'enregistrement des nogoods sous BTD Un bon choix pour ces différents paramètres s'avère être difficile quoique primordial pour la performance de BTD. En particulier, BTD est condamné, tout au long de la résolution, à suivre des choix faits en amont de la résolution comme le choix de la décomposition et de sa racine. Dans [Jégou and Terrioux, 2003; Jégou et al., 2006a,b, 2007], les auteurs proposent des améliorations qui visent à donner davantage de liberté à BTD tout en conservant certaines garanties théoriques. Néanmoins, aucune de ses propositions ne permet de corriger un mauvais comportement de BTD dû à un choix de racine non judicieux ou à une décomposition mal calculée.

C'est ainsi que, dans [Jégou and Terrioux, 2014a, 2017], les auteurs proposent d'exploiter les redémarrages et l'enregistrement des nogoods dans BTD. Les redémarrages permettent à BTD de s'affranchir de certaines limitations imposées par la décomposition. Au-delà des bénéfices du redémarrage connues en général, il permet également dans le cadre de BTD, de changer le cluster racine. BTD possède alors l'opportunité de choisir, lors d'un redémarrage, un cluster racine plus adéquat que celui choisi au début de la résolution. Les auteurs dans [Jégou and Terrioux, 2014a, 2017] proposent l'algorithme 2.5 $BTD\text{-}MAC{+}RST$ qui fait appel à l'algorithme 2.4 $BTD\text{-}MAC$ ($BTD\text{-}MAC{+}NG$ dans

[Jégou and Terrioux, 2014a, 2017]) où nous considérons maintenant les lignes 26-28 et la valeur *inconnu*. Pendant la résolution, les redémarrages peuvent être déclenchées comme pour les méthodes standards telles que MAC+RST+NG [Lecoutre et al., 2007e]. Les stratégies de redémarrage basées sur une suite géométrique [Walsh, 1999] ou sur une suite Luby [Luby et al., 1993] sont exploitées dans [Jégou and Terrioux, 2014a, 2017]. Lorsqu'un redémarrage a lieu (ligne 26), BTD-MAC retourne *inconnu* vu que la résolution n'a pas encore abouti (ligne 28). À chaque redémarrage, un nouveau appel à BTD-MAC est réalisé par BTD-MAC+RST (ligne 4). Avant cet appel, BTD-MAC+RST est capable de changer de cluster racine (ligne 3). Dans [Jégou and Terrioux, 2014a, 2017], les auteurs proposent deux heuristiques de choix de racine efficaces. Par exemple, la première choisit le cluster qui maximise la somme des poids (au sens des poids wdeg [Lecoutre et al., 2007e]) de contraintes qui l'intersectent

À chaque redémarrage, BTD-MAC doit enregistrer des nld-nogoods réduits relatifs à chaque cluster. Ces enregistrements sont faisables selon la propriété 1 énoncée et démontrée par les auteurs [Jégou and Terrioux, 2014a, 2017].

Propriété 1 *Soit $\Sigma =< \delta_1, \ldots, \delta_k >$ une suite de décisions réalisée en exploitant la décomposition (E, T) selon un ordre de variables compatible. Soit $\Sigma[E_i]$ la suite construite en considérant uniquement les décisions de Σ impliquant les variables de E_i. Pour toute suite préfixe $\Sigma'_{E_i} =< \delta_{i_1}, \ldots, \delta_{i_j} >$ de $\Sigma[E_i]$ telle que δ_{i_j} est une décision négative et telle que les variables de $E_i \cap E_{p(i)}$ apparaissent dans $Pos(\Sigma'_{E_i})$, l'ensemble $Pos(\Sigma'_{E_i}) \cup \{\neg\delta_{i_j}\}$ est un nld-nogood réduit de P.*

La validité de BTD-MAC n'est pas menaçée sachant que la portée d'un nld-nogood réduit est toujours incluse ou égale à l'ensemble de variables du cluster auquel il appartient. En outre, les auteurs précisent qu'ils considèrent une contrainte globale par cluster pour gérer les nld-nogoods réduits relatifs à ce cluster comme dans [Lecoutre et al., 2007e]. Leur exploitation est assurée via un propagateur spécifique lorsque la cohérence d'arc est renforcée (lignes 23 et 31).

Le changement du cluster racine, à chaque redémarrage, ne peut pas remettre en cause la validité des (no)goods structurels [Jégou and Terrioux, 2014a, 2017] :

Propriété 2 *. Un good structurel de E_i par rapport à E_j peut être uniquement utilisé si le choix du nouveau cluster racine ne change pas le fait que E_j est un fils de E_i. Un nogood structurel reste utilisable quelque soit le choix du nouveau cluster.*

Les auteurs démontrent que BTD-MAC+RST est correct, complet et termine. Sa complexité en temps est $O(R.((n.s^2.m.log(d) + w^+.|N_r|).d^{w^+ +2} + n.(w^+)^2.d))$ avec R le nombre de redémarrages et N_r l'ensemble des nld-nogoods réduits enregistrés. Sa complexité en espace est $O(n.s.d^s + w^+.(d + |N_r|))$. Finalement, les auteurs énoncent que si la stratégie de redémarrage exploitée est le redémarrage géométrique basée sur le nombre de backtracks, que r_0 est le nombre initial de backtracks et que r_q est le ratio alors le nombre R de redémarrages est borné par $\lceil \frac{log(n)+(w^+ +1).log(d)-log(r_0)}{log(r_q)} \rceil$.

2.2.5.2 Autres méthodes structurelles

D'autres méthodes structurelles ont été définies et quelques-unes d'entre elles seront rappelées par la suite.

Tree-Clustering (TC)[Dechter and Pearl, 1989] Cette méthode de programmation dynamique se base également sur une décomposition arborescente. Les différents clusters

sont résolus indépendamment et l'ensemble de leurs solutions mémorisé. TC résout ensuite le nouveau CSP acyclique en temps polynomial. Ses complexités en temps et en espace dépendent alors de la taille du plus grand cluster, $w^+ + 1$, et sont en $O(m.d^{w^+ + 1})$. Cette méthode requiert un espace mémoire important en raison du stockage des solutions relatives à chaque cluster. Ultérieurement, cette complexité a été améliorée en restreignant la mémorisation aux intersections des clusters pour devenir en $O(n.s.d^s)$ [Jensen et al., 1990; Shafer and Shenoy, 1990] comme dans le cas de BTD. Le fait de chercher toutes les solutions pour un cluster si le but est uniquement de savoir si l'instance est cohérente peut être particulièrement lourd et est susceptible de fortement détériorer l'efficacité de la résolution sans compter le coût en espace souvent prohibitif. D'ailleurs, rien ne garantit qu'une solution du cluster est compatible avec au moins une solution des autres clusters. Une approche comme BTD basée sur un algorithme énumératif évite un tel inconvénient.

AND/OR Search Space [Dechter and Mateescu, 2007] Cette méthode se base sur un arbre de recherche AND/OR qui peut être exponentiellement plus petit qu'un arbre de recherche standard tout en permettant de détecter les indépendances du graphe. Un arbre de recherche AND/OR correspondant à un graphe G, noté $\mathcal{S}_T$, est une alternance de niveaux de nœuds AND et OR. Un nœud OR correspond à une variable x_i tandis qu'un nœud AND correspond à $\{x_i \leftarrow v_i\}$ ou simplement à la valeur v_i attribuée à x_i. Cet arbre peut être déduit à partir d'un arrangement en arbre T du graphe G. La racine de l'arbre AND/OR est un nœud OR ayant le même label que celui de T. Un nœud OR, x_i, a un nœud fils v_i si l'affectation $\{x_i \leftarrow v_i\}$ est cohérente avec toutes les affectations déjà réalisées depuis la racine. Un nœud AND, v_i, a un nœud fils OR, x_j, ssi x_j est fils de x_i dans T. Une solution est un sous-arbre contenant le nœud racine, un nœud fils parmi les fils de chaque nœud OR, tous les nœuds fils de chaque nœud AND et des nœuds feuille cohérents. Afin de rendre l'arbre de recherche plus compact, des règles de *fusion* et d'*unification* ont été définies pour confondre des nœuds qui engendrent l'exploration du même sous-espace de recherche. Dans le but de pouvoir détecter ces nœuds, les auteurs [Dechter and Mateescu, 2007] associent à chaque nœud OR et AND respectivement la notion de *parent* et de *parent-separator*. Deux nœuds de même type peuvent être ainsi fusionnés s'ils ont la même affectation sur les variables de leur parent (ou leur parent-separator). Notons que la fusion transforme l'arbre de recherche en un graphe de recherche. Pour trouver une solution, l'algorithme parcourt l'arbre de recherche AND/OR suivant un parcours en profondeur d'abord. Il attribue la valeur 0 à un nœud OR et la valeur 1 à un nœud AND et propage ces valeurs par le biais de la somme booléenne pour un nœud OR et du produit booléen pour un nœud AND en remontant. Sa complexité en temps est en $O(exp(h))$ avec h la hauteur de l'arrangement en arbre de G et est linéaire en espace. Cependant, si la recherche se base sur un graphe de recherche l'algorithme doit gérer les enregistrements pour rendre la fusion de nœuds possible. La complexité en temps et en espace est en $O(exp(w))$. À l'instar des méthodes basées sur une décomposition, la complexité en espace peut être améliorée pour atteindre une complexité en $O(exp(s))$. Notons que w et s notent respectivement la largeur et la taille du plus grand séparateur d'une décomposition optimale du graphe G. Cette méthode obtient parmi les meilleurs résultats des méthodes structurelles en pratique notamment après l'introduction du concept de l'ordre dynamique de variables. Finalement, les auteurs [Dechter and Mateescu, 2007] ont souligné les ressemblances entre cette méthode et d'autres comme BTD [Jégou and Terrioux, 2003], *variable elimination* [Dechter and Pearl, 1987; Dechter, 1999], *recursive conditioning* [Darwiche, 2001c], *value elimination* [Bacchus et al., 2002] ... Par exemple, BTD peut être vu comme explorant un graphe AND/OR où l'arrangement en arbre de G est construit à partir de l'arbre

T relatif à la décomposition arborescente sur laquelle se base BTD. Les goods structurels de BTD correspondent exactement aux niveaux AND de l'espace AND/OR (via les parents-separators). Ainsi, sans surprise, BTD et AND/OR search ont les mêmes bornes de complexité théorique.

Recursive Conditioning (RC)[Darwiche, 2001c] Cette méthode découle du principe *diviser pour régner*. Elle décompose le réseau de contraintes en des sous-problèmes plus petits non connexes qui peuvent être résolus indépendamment. Pour y parvenir, elle se base sur une affectation donnée d'un cutset (un séparateur) qui est à l'origine de cette séparation. Ce principe est appliqué récursivement sur chacun des sous-problèmes. Pour pouvoir capturer cette structure, elle exploite la notion de *dtree* qui est un arbre binaire complet dont les feuilles correspondent aux CPTs (pour *conditional probability table*) du réseau Bayésien. Lorsque les variables d'un nœud interne de l'arbre sont affectées, les deux sous-arbres gauche et droite deviennent totalement indépendants formant à leur tour un dtree. L'algorithme a une complexité en temps $O(n.exp(w^t.\log n))$ et en espace en $O(n)$ avec w^t la largeur de l'ordre d'élimination associé au dtree. Malgré son intérêt au niveau d'espace requis, cette version engendre beaucoup de calculs redondants en raison de l'absence totale d'enregistrements. Une deuxième version de cet algorithme, exploite la notion de *contexte* identique à celle utilisée dans le cadre de AND/OR Search Space. En offrant l'opportunité d'enregistrer la réponse déjà calculée pour un sous-problème induit par une affectation donnée, l'algorithme voit sa complexité en temps chuter à $O(n.exp(w^t))$ au détriment d'une complexité en espace en $O(n.exp(w^t))$. Cette approche est aussi similaire à celle suivie par BTD ou AND/OR search avec tous les inconvénients engendrés par le choix de variables induit par la dtree comme cela était le cas avec les deux autres méthodes. À partir de ces deux versions, Darwiche [Darwiche, 2001c] élabore un algorithme *anyspace* qui s'adapte graduellement à l'espace mémoire disponible en n'enregistrant des informations que si cet espace le permet. Par conséquent, il parie sur un compromis espace/temps. Il évoque également la possibilité d'éliminer des séparateurs de grande taille comme dans [Dechter, 1996] au prix d'avoir des clusters plus grands vu que la complexité spatiale peut être finalement exprimée en fonction de la taille de ces séparateurs. Darwiche [Darwiche, 2001c] souligne aussi la relation étroite entre la tree-width et la largeur d'un ordre d'élimination associé à un dtree qui n'est que la meilleure largeur d'un tel ordre. De là, n'importe quelle méthode efficace pour le calcul d'une décomposition arborescente peut l'être pour le calcul d'un dtree. Cette méthode a obtenu de bons résultats en pratique. Finalement, les auteurs de [Dechter and Mateescu, 2007] précisent que RC explore un espace AND/OR en montrant que lorsque RC décompose le problème en sous-problèmes indépendants, il en est de même pour un espace AND/OR.

Bucket elimination (BE)[Dechter and Pearl, 1987; Dechter, 1999] Bucket elimination est un cadre algorithmique dont la simplicité et la généralité lui ont permis d'être omniprésent dans plusieurs domaines. Son étape basique est l'élimination de variables VE. Dans le cadre de la résolution d'instances CSP, il est connu comme l'algorithme de la *cohérence adaptative* (CA) [Dechter and Pearl, 1987]. Étant donné un ordre sur les variables, il élimine les variables une à une dans le sens inverse, en répercutant l'effet de cette élimination sur le reste du problème et en préservant l'équivalence de l'instance CSP. La première étape consiste à partitionner les contraintes. Pour cela on associe, à chaque variable, les contraintes qui la contiennent dans leur portée parmi les contraintes restantes du traitement de la variable qui la suit dans l'ordre. La première variable traitée est la dernière variable de l'ordre en considérant toutes les contraintes de l'instance CSP.

Ces parties sont appelées les *buckets* et sont traités dans le sens inverse de l'ordre. Le résultat d'un traitement est une nouvelle contrainte calculée via des jointures de relations et ajoutée convenablement aux buckets inférieurs. Notons ainsi que les contraintes doivent être impérativement représentées en extension. Lorsque tous les buckets sont résolus et qu'aucune incohérence n'est inférée, une solution peut être donnée en parcourant les buckets sans retour en arrière selon l'ordre sur les variables. Sa complexité en temps et en espace est en $O(n.exp(w^o))$ avec w^o est la largeur associée à l'ordre d'élimination. Sa complexité en espace vient des contraintes ajoutées dont l'arité peut atteindre w^o. Afin de surmonter l'obstacle de la mémoire, Larrosa [Larrosa, 2000] suggère de combiner la recherche et l'élimination de variables en proposant l'algorithme VES (pour *Variable Elimination Search*). VES prend en entrée un algorithme de recherche comme FC [Haralick and Elliott, 1980] par exemple, et un entier p qui constitue la borne supérieure sur l'arité des contraintes à inférer. Si l'élimination de la variable x_i produit une contrainte dont l'arité est inférieure ou égale à p, l'élimination est permise, sinon elle est interdite. Dans ce dernier cas, VES effectue un branchement sur x_i en considérant ses différentes valeurs. Le comportement de VES peut aller de la CA [Dechter and Pearl, 1987] à un FC ou une hybridation des deux algorithmes qui dépend de p. La complexité en espace de VES est en $O(exp(p))$ et en temps en $O(exp(p+q))$ avec q le nombre de variables où il effectue un branchement. En pratique, VES améliore CA pour les deux algorithmes FC et RFL avec $p = 2$ [Larrosa, 2000] pour les instances binaires aléatoires creux de [Smith, 1994]. VES permet finalement de réaliser un compris temps/espace toutefois sans contrôle de la valeur de q.

Backtracking on biconnected Components (BCC)[Baget and Tognetti, 2001] Cette méthode peut être considérée comme un cas particulier de BTD. En effet, elle opère sur une décomposition en composantes biconnexes qui n'est qu'une décomposition arborescente dont la taille des séparateurs est limitée à 1 et dont les clusters ne contiennent aucun séparateur de taille 1 (appelé *point d'articulation*). À l'image de BTD, l'ordre de choix de variables est dicté par la décomposition (il est même total pour BCC). La méthode de recherche basique utilisée est BT mais la possibilité d'utiliser des algorithmes plus sophistiqué comme MAC est évoquée. Un enregistrement est aussi réalisé d'une façon plus légère que BTD en raison de la taille des séparateurs (suppression de valeurs si incohérence, marquage des valeurs si cohérence). Sans surprise, sa complexité en temps est identique à celle de BTD. Néanmoins, l'exploitation de la structure par BCC est assez limitée vis-à-vis celle de BTD. La taille de ses clusters peut être extrêmement grande voire égale à n si le graphe n'admet aucun point d'articulation.

Cycle cutset(coupe-cycle)(CCM)[Dechter and Pearl, 1986] Cette méthode se base sur le calcul d'un ensemble coupe-cycle. Il s'agit d'un sous-ensemble de sommets du graphe primal de l'(hyper)graphe de contraintes dont son retrait rend ce dernier acyclique. Étant donné un CSP acyclique, la résolution se réduit à une recherche énumérative sur les variable du coupe-cycle. Cette méthode est exponentielle en fonction de la taille de ce coupe-cycle [Pearl, 1988; Dechter, 1990; Díez, 1996]. En pratique, elle n'est efficace que lorsque ce dernier le coupe-cycle est de petite taille. Malheureusement, dans [Bertele and Brioschi, 1972; Shachter et al., 1994], les auteurs montrent que la taille minimale d'un coupe-cycle de n'importe quel graphe peut être significativement supérieure (et jamais inférieure) à sa tree-width. Par conséquent, les garanties temporelles peuvent être significativement moins bonnes que celles de BTD ou TC.

Cyclic-clustering(CC)[Jégou, 1990; Jégou and Terrioux, 2004a; Pinto and Terrioux, 2009] Cet algorithme vise à améliorer TC et CCM en les combinant judicieusement. La première étape consiste à repérer un sous-graphe triangulé du graphe primal de G induit par $X_{\subseteq}$ dont les cliques maximales formeront chacune une arête (et correspond donc à une décomposition arborescente). Les sommets n'appartenant pas au sous-graphe constituent ainsi un ensemble coupe-cycle. Le point intéressant est que la taille de ce dernier est réduite par rapport à une application directe sur l'instance CSP originale. Ensuite, le CSP acyclique induit par $X_{\subseteq}$ est généré laissant place à l'application de CCM. La partie du problème associée à l'ensemble coupe-cycle peut être résolue avec un algorithme énumératif classique alors que la partie associée à $X_{\subseteq}$ est résolue avec une méthode telle que TC. CC offre un compromis espace/temps, à savoir une meilleure complexité en temps que celle de CCM et une meilleure complexité en espace que celle de TC. Des expérimentations conduites avec BTD qui s'avère plus efficace que TC (appelée CC-BTD) ont montré l'intérêt pratique de CC-BTD. Dans [Pinto and Terrioux, 2009], une généralisation de CC-BTD est présentée qui permet de calculer l'ensemble coupe-cycle et la décomposition plus librement et d'alterner les méthodes CC et BTD plus convenablement. En outre, elle exploite une version de BTD dédiée qui évite beaucoup de redondances de calcul. Cette nouvelle méthode est significativement meilleure que CC-BTD. Néanmoins, les auteurs [Pinto and Terrioux, 2009] soulignent l'absence de méthodes pertinentes pour calculer un coupe-cycle de taille raisonnable. Ce problème va alors s'ajouter au problème du calcul d'une décomposition arborescente de qualité vis-à-vis de la résolution. Ils appuient leur propos en donnant l'exemple des problèmes d'allocation de fréquence (instances *fapp* de la compétition CSP 2008) où la grande majorité des variables se trouvaient dans le coupe-cycle.

Méthodes basées sur des décompositions hyperarborescentes (fractionnaires) [Gottlob et al., 2000; Cohen et al., 2005] La résolution consiste à calculer l'ensemble des solutions sur les différents clusters par une jointure des hyperarêtes. Cela donne naissance à un CSP acyclique qui peut être résolu en temps polynomial. Les complexités en temps et en espace de ces méthodes sont en $O(A^{hw^+})$ avec hw^+ la largeur de la décomposition en question et A le nombre le nombre maximal de tuples de toutes les contraintes. En pratique, ces méthodes souffrent de plusieurs faiblesses. Une première est la nécessité d'avoir des contraintes exprimées en extension pour calculer les jointures. Cela semble impossible pour certaines contraintes ayant un nombre de tuples trop grand pour être représentées en extension (contraintes en intention, contraintes globales). En outre, comme dans les cas de TC l'espace requis est trop important. Le calcul des jointures des hyperarêtes des clusters donne de nouvelles contraintes qui peuvent avoir un nombre important de tuples.

Bilan Nous avons rappelé dans cette partie les plus connues des méthodes structurelles pour la résolution d'instances CSP. Ces méthodes exploitent la structure de l'instance et fournissent des bornes de complexité théorique en temps très intéressantes vis-à-vis des méthodes énumératifs classiques. La plupart d'entre elles se laissent guider par une arborescence représentant l'(hyper)graphe de contraintes et misent sur une détection des parties indépendantes du problème et sur des enregistrements permettant d'éviter certaines redondances pendant la recherche (comme BTD, AND/OR, RC ou BCC). Ce faisant, des instances difficiles pour des méthodes classiques mais présentant une « jolie » structure peuvent être efficacement résolues par telles méthodes. Cependant, les enregistrements peuvent conduire à un coût en mémoire plus ou moins élevé. C'est ainsi qu'un compromis

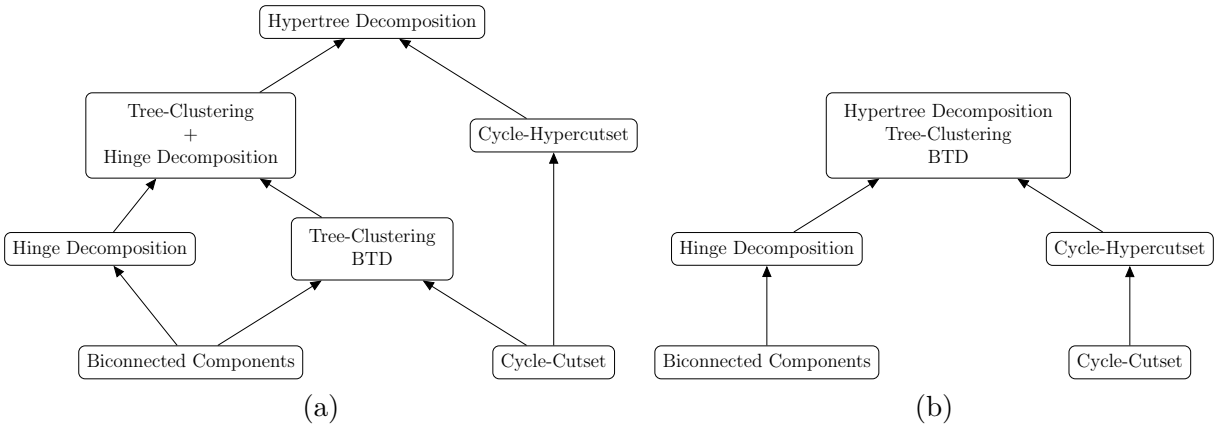


FIGURE 2.4 – La hiérarchie des méthodes de décomposition structurelles [Gottlob et al., 2000] (a) la hiérarchie revisitée grâce à l'évaluation de la complexité de nFC_{2m} [Jégou et al., 2009] (b).

temps/espace doit être considéré. Le principal obstacle à ces approches réside dans les restrictions imposées par l'arborescence de base sur l'heuristique de choix de variables qui se trouve emprisonnée par cette dernière. Il semble que ce problème est directement responsable de la détérioration de l'efficacité de la résolution lorsque la décomposition utilisée est de mauvaise qualité.

2.2.5.3 Comparaison en termes de bornes de complexité et de performances

Dans [Gottlob et al., 2000], plusieurs méthodes de résolution par décompositions sont comparées selon des critères de traitabilité. Cette comparaison permet d'établir une hiérarchie présentée dans la figure 2.4(a). Pour chaque méthode mentionnée, la hiérarchie fait allusion à une méthode de calcul de décomposition et une méthode de résolution l'accompagnant. La puissance d'une méthode de décomposition de la hiérarchie est définie selon la qualité de la borne de complexité théorique qui en découle dans le cadre de la résolution. La relation sur laquelle se base cette hiérarchie est une relation de *généralisation forte*. Le fait qu'une méthode de décomposition D_2 généralise fortement une méthode D_1 (un arc relie alors D_1 à D_2) signifie que D_2 est strictement plus puissante que D_1 du fait que là où D_1 garantit une résolution polynomiale, il en est de même pour D_2, tandis qu'il existe des classes d'instances qui peuvent être résolues d'une façon polynomiale par D_2 mais pas par D_1. Nous pouvons noter que les méthodes basées sur une décomposition hyperarborescente [Gottlob et al., 1999] sont plus générales que toutes les autres méthodes dont celles basées sur une décomposition arborescente.

Cette hiérarchie est revisitée dans [Jégou et al., 2009]. Malgré son intérêt théorique comme outil de comparaison des méthodes de résolution basées sur une décomposition, il existe un fossé entre la théorie et la performance pratique de chaque méthode. D'ailleurs, on suppose dans la hiérarchie que les décompositions en question sont optimales. Cela n'est pas raisonnable en pratique vu que le calcul d'une décomposition optimale peut s'avérer très coûteux comme nous avons déjà vu dans la partie 1.3.2 pour la décomposition arborescente. En particulier, le calcul de l'(hyper)tree-width est NP-difficile [Arnborg et al., 1987; Gottlob et al., 1999]. En outre, les auteurs [Jégou et al., 2009] montrent qu'il est possible d'exploiter des algorithmes comme $nFC_i(i \geq 2)$ [Bessière et al., 2002] ou RFL pour mettre à jour cette hiérarchie. La complexité de la variante de nFC utilisée est évaluée dans le cas des instances n-aires [Jégou et al., 2008] est en $O(|P|.A^m)$. Les auteurs proposent un

raffinement de cette complexité en considérant la notion du *recouvrement minimum* notée $k_{(X,C)}$. Ainsi, sa nouvelle complexité en temps est en $O(|P|.A^{k_{(X,C)}})$ pour une instance CSP $P = (X, D, C)$. Selon les auteurs, ce résultat s'applique sans difficulté à RFL et aux différentes variantes de $nFC_i (i \geq 2)$. En exploitant cette complexité, les conversions possibles entre la décomposition arborescente et la décomposition hyperarborescente et une nouvelle association des contraintes aux clusters, les auteurs démontrent que les méthodes BTD et TC peuvent être placées en tête de la hiérarchie [Jégou et al., 2009]. En effet, à partir d'une décomposition hyperarborescente de largeur hw, une décomposition arborescente peut être construite. Ensuite, l'algorithme RFL ou nFC_i est appliqué à chaque cluster E_i auquel est associée toute contrainte c_j telle que $S(c_j) \cap E_i \neq \emptyset$. La résolution de E_i peut alors s'effectuer en $O(|P_{E_i}|.A^{hw})$ vu qu'un recouvrement minimum est majoré par hw. Nous obtenons ainsi la même complexité qu'une méthode basée sur une décomposition hyperarborescente. Par conséquence, les méthodes TC et BTD basées sur une décomposition arborescente partagent la tête de la hiérarchie avec celle basée sur la décomposition hyperarborescente comme le montre la figure 2.4(b).

Par ailleurs, les méthodes structurelles de la hiérarchie souffrent de multiples défauts en pratique. Tout d'abord, l'approche par programmation dynamique consiste à réaliser une jointure des relations de contraintes de chaque cluster conduisant à une relation unique par cluster. Cette étape permet de créer une instance CSP acyclique pouvant être résolue par maintien de cohérence d'arc. Cependant réaliser la jointure des relations nécessite de représenter les contraintes en extension. Or, certaines contraintes exprimées en intention ou certaines contraintes globales sont difficilement représentables en extension en raison du nombre de tuples supports qui peut être significativement élevé. C'est le cas par exemple des méthodes basées sur la décomposition hyperarborescente ou TC. Ensuite, dans ce cas, toutes les solutions d'un cluster doivent être mémorisées. Du fait de cette masse trop importante d'informations à stocker, ces méthodes sont quasi-inopérantes en pratique. En outre, le fait de calculer toutes les solutions nuit naturellement à la performance de ces méthodes en comparaison à une méthode telle que BTD qui s'assure de la compatibilité d'une solution du cluster avec des solutions des autres sous-problèmes. Pour toutes ces raisons l'utilisation de TC ou des méthodes basées sur une décomposition hyperarborescente est extrêmement limitée en pratique [Gottlob et al., 2002; Habbas et al., 2016]. En ce qui concerne les méthodes basées sur un (hyper)coupe-cycle, un coupe-cycle a en théorie une taille toujours supérieure à la tree-width [Bertele and Brioschi, 1972; Shachter et al., 1994]. En plus, trouver un coupe-cycle pertinent de taille raisonnable semble être difficile en pratique [Pinto and Terrioux, 2009]. Enfin, les méthodes basées sur des composantes biconnexes conduisent à des tailles de clusters trop importantes et à un nombre de séparateurs généralement faible limitant l'intérêt des enregistrements pouvant être réalisés. Cela rend ces méthodes inefficaces en pratique. Au vu de ces différentes faiblesses, la méthode BTD semble la plus exploitable en pratique notamment du fait qu'il existe des heuristiques efficaces en général pour le calcul de la décomposition.

2.2.6 Bilan

La quantité de travaux réalisés autour du problème CSP est basée principalement à la fois sur son pouvoir de modélisation et sur l'efficacité des méthodes de résolution existantes. Une méthode de résolution se définit en fonction des multiples paramètres d'un solveur. Nous distinguons les méthodes énumératives classiques et les méthodes structurelles qui contrairement aux autres méthodes exploitent la structure de l'instance CSP. En raison de la complexité théorique des méthodes classiques, exponentielle en n, ces méthodes peuvent parfois rencontrer quelques difficultés lors du passage à l'échelle malgré

leur efficacité en pratique. Les méthodes structurelles, pour lesquelles BTD fait office de référence, ont une complexité théorique exponentielle seulement en fonction de la tree-width de l'(hyper)graphe de contraintes. En choisissant d'exploiter une décomposition arborescente, plusieurs paramètres s'ajoutent aux paramètres usuels d'un solveur qui sont : le calcul de la décomposition, le choix du cluster racine et les heuristiques de choix du prochain cluster. BTD, comme de nombreuses méthodes structurelles, peut souffrir du manque de liberté au niveau du choix de la prochaine variable qui est en partie induit par l'arborescence sur laquelle il se base. Dans le cas de BTD, calculer une décomposition de qualité, choisir judicieusement le cluster racine et le prochain cluster permettent sans doute d'améliorer la performance de BTD. Un premier pas vers une décomposition plus dynamique a été présenté dans [Jégou and Terrioux, 2014a, 2017] en intégrant les redémarrages à BTD et le choix d'un nouveau cluster racine à chaque redémarrage. Toutefois, cette solution ne remet pas en cause le calcul d'une décomposition peu pertinente en amont de la résolution. Ainsi, nous nous focalisons dans cette thèse sur l'amélioration des méthodes structurelles de résolution d'instances CSP notamment BTD.

2.3 Problème du comptage : #CSP

Ce problème vise à répondre à la question suivante : étant donnée une instance CSP P, combien de solutions admet-elle ? Elle peut en admettre aucune si elle est incohérente ou une ou plusieurs solutions sinon. Nous pouvons, par exemple, vérifier que l'exemple 1 admet 6 solutions. Le problème #CSP a de nombreuses applications en intelligence artificielle comme dans le raisonnement approximatif [Roth, 1996], le diagnostic [Kumar, 2002], la révision des croyances [Darwiche, 2001b], l'inférence probabiliste [Sang et al., 2005; Chavira and Darwiche, 2008; Apsel and Brafman, 2012; Choi et al., 2013], la planification [Palacios et al., 2005; Domshlak and Hoffmann, 2006] et dans d'autres domaines plus éloignés de l'informatique comme la physique statistique [Burton and Steif, 1994] ou la chimie pour la prédiction de la structure d'une protéine [Mann et al., 2007]. L'évaluation du nombre de solutions peut également servir pour guider la recherche en orientant l'heuristique du choix de variables et/ou de valeurs comme dans [Kask et al., 2004; Pesant, 2005; Pesant et al., 2012].

Le problème du comptage est un problème extrêmement difficile d'un point de vue théorique en raison de sa complexité puisqu'il appartient à la classe #P-complet [Valiant, 1979]. Sa difficulté est telle qu'il reste #P-complet même si nous nous restreignons au cas des CSP binaires. Cette difficulté est confirmée par le théorème de Toda qui énonce que $PH \subseteq P^{\#P}$ [Toda, 1991]. Des études théoriques ont été réalisées dans le but d'analyser ce problème du point de vue de la complexité théorique en exhibant des classes traitables comme dans [Slivovsky and Szeider, 2013]. D'autres travaux ont analysé leur difficulté par le biais des théorèmes de dichotomie comme dans [Bulatov and Dalmau, 2003; Bulatov, 2008; Dyer and Richerby, 2013]. En pratique, sa résolution est également très difficile. Dans cette partie, nous nous focalisons sur ce problème en insistant sur les méthodes les plus connues pour sa résolution.

2.3.1 Méthodes de résolution

Plusieurs méthodes de résolution ont été proposées. On va surtout se focaliser sur les méthodes structurelles qui nous intéressent le plus dans le cadre de cette thèse.

L'approche naturelle consiste à étendre les algorithmes standards définis pour le problème de décision comme BT à $\#BT$ ou MAC et RFL à $\#MAC$ et à $\#RFL$. Leur

complexité est en $O(r.m.d^n)$. En pratique, ces méthodes peuvent être inefficaces pour le dénombrement des solutions dans certains cas, notamment lorsque le nombre de solutions est très élevé (certaines instances du benchmark que nous utilisons dans la partie contributions possèdent plus de 6.10^{303} solutions). En effet, l'espace de recherche qui doit être visité implique beaucoup de redondances et de recalculs inutiles du nombre d'extensions cohérentes d'une affectation partielle. D'autres algorithmes exacts ont été proposés sans être évalués en pratique comme celui fourni pour les instances CSP binaires dans [Angelsmark and Jonsson, 2003].

Nous nous intéressons d'abord aux méthodes proposées pour le problème #CSP puis aux méthodes proposées pour le problème #SAT. Nous nous focalisons ensuite sur l'approche du comptage après compilation.

2.3.1.1 Méthodes pour #CSP

Les méthodes structurelles sont pertinentes pour le dénombrement de solutions du fait de la difficulté de ce problème et de la borne de complexité théorique en temps séduisante offerte par ces méthodes.

AND/OR Search Space [Dechter and Mateescu, 2004] La méthode AND/OR Search Space évoquée dans le cadre de la décision peut être facilement adaptée au comptage. En effet, il suffit de remplacer les opérations booléennes par des opérations de somme et de produit. Elle a été comparée avec une méthode de backtrack simple adaptée au comptage. Selon les auteurs, AND/OR Search Space surclasse cette dernière ; elle est bien meilleure en termes de nombre de nœuds développés et en temps d'exécution notamment sur des instances ayant un grand nombre de solutions. D'ailleurs, elles ne sont comparables que pour les problèmes incohérents. Finalement, les auteurs ont montré que leur méthode est capable de s'adapter à des réseaux de plus grande taille (au plus 100 variables) que dans le cas de l'autre méthode.

#BTD [Favier et al., 2009] Les auteurs adaptent l'algorithme BTD au dénombrement exact de solutions. Il se base sur la décomposition arborescente pour identifier les parties indépendantes du graphe. Ainsi, étant donnée une affectation $\mathcal{A}$ qui sépare le graphe en plusieurs composantes connexes, le nombre de solutions de chaque composante est calculé indépendamment des autres. Le nombre d'extensions cohérentes de $\mathcal{A}$ est le résultat de la multiplication du nombre de solutions de ces composantes. À l'instar de BTD, $\#BTD$ exploite des enregistrements afin d'éviter de visiter le même sous-espace de recherche plusieurs fois. Si BTD enregistre des (no)goods, $\#BTD$ enregistre pour un sous-problème induit par une affectation donnée le nombre de solutions trouvées. Il s'agit de la notion de #good dont nous rappelons maintenant la définition :

Définition 57 *Soient (E, T) une décomposition arborescente, E_i un cluster et E_j un de ses clusters fils. Un #good de E_i vis-à-vis de E_j est une paire $(\mathcal{A}[E_i \cap E_j], nb)$ avec $\mathcal{A}[E_i \cap E_j]$ une affectation cohérente de $E_i \cap E_j$ et nb le nombre de solutions du sous-problème $P_j|\mathcal{A}[E_i \cap E_j]$.*

Vu son importance dans le cadre de cette thèse, cette méthode sera détaillée dans le chapitre 6.

2.3.1.2 Méthodes pour #SAT

Beaucoup de méthodes ont été proposées dans le cadre du formalisme SAT [Biere et al., 2009]. Un problème SAT peut être vu comme un problème CSP (X, D, C) tel que toutes les variables sont *booléennes* ayant deux valeurs possibles 0 ($faux$) ou 1 ($vrai$). À chaque variable x_i sont associés deux *littéraux*, un positif (x_i) et sa négation $\neg x_i$. x_i et $\neg x_i$ ont des valeurs booléennes différentes. Les contraintes sont des *clauses*, c'est-à-dire des disjonctions de littéraux. Par exemple, $x_1 \vee \neg x_2 \vee x_3$ est une clause qui est satisfaite ssi $x_1 \leftarrow 1$ ou $x_2 \leftarrow 0$ ou $x_3 \leftarrow 1$. Cette représentation est appelée CNF (*conjonctive normal form*). Le but du problème SAT consiste à satisfaire simultanément toutes les clauses. Une solution du problème SAT est appelé un *modèle*. L'algorithme de base de résolution du problème SAT est l'algorithme Davis-Putnam-Logemann-Loveland (DPLL) [Davis et al., 1962]. Le problème de comptage en SAT est noté #SAT. Afin d'exploiter les méthodes de comptage de modèles en SAT pour le comptage en CSP, le moyen le plus simple consiste à encoder l'instance CSP en instance SAT. Parmi les encodages les plus connus, nous citons l'encodage direct, l'encodage logarithmique [Walsh, 2000] et l'encodage tuple [Hurley et al., 2016].

CDP Comme pour le problème #CSP, la première approche pour la résolution du problème #SAT consiste à étendre DPLL. Cette extension s'appelle CDP (pour *counting Davis-Putnam*) et a été proposée dans [Birnbaum and Lozinskii, 1999]. Étant donnée une affectation $\mathcal{A}$ (appelée *interprétation*) de taille k, elle réalise une *propagation unitaire* qui consiste à supprimer chaque clause de taille 1 en affectant convenablement la variable en question. Elle vérifie ensuite s'il existe une clause vide auquel cas l'affectation courante n'admet aucune extension. Si toutes les clauses sont satisfaites le nombre d'extensions de $\mathcal{A}$ est 2^{n-k} vu que les variables restantes peuvent être instanciées d'une façon quelconque. Sinon une k+1-ème variable x_i est choisie. Le nombre d'extensions de $\mathcal{A}$ est alors le nombre d'extensions de $\mathcal{A} \cup \{x_i \leftarrow 1\}$+ le nombre d'extensions de $\mathcal{A} \cup \{x_i \leftarrow 0\}$. L'appel initial à CDP part d'une affectation vide. Si la limite de temps est dépassée, CDP retourne une borne inférieure sur le nombre de solutions trouvé pour le sous-espace déjà visité.

relsat (DDP) Une nouvelle méthode DDP (pour *decomposing Davis-Putnam*) a été ensuite proposée dans [Bayardo and Pehoushek, 2000]. Elle rajoute à CDP une couche d'identification de composantes connexes après la réalisation de la propagation unitaire. Cette étape ressemble à BTD qui sépare les sous-problèmes enracinés en clusters fils E_j d'un cluster E_i suite à l'affectation de ce dernier. CDP peut ainsi calculer le nombre de solutions de chaque composante. En raison de leur indépendance, le nombre d'extensions de $\mathcal{A}$ est la *multiplication* du nombre de solutions de chaque composante. Bien que la détection des composantes connexes ait été jugée assez coûteuse pour le problème SAT, cet effort peut être rentable pour des problèmes plus difficiles comme #SAT. Cet algorithme est implémenté dans *relsat*. $relsat$ exploite plusieurs heuristiques pour la gestion des composantes connexes comme considérer la composante la plus contrainte d'abord ou s'assurer de la satisfiabilité de chaque composante avant de compter toutes les solutions. Ces améliorations ont permis à $relsat$ d'avoir une meilleure performance que CDP [Bayardo and Pehoushek, 2000].

cachet Si l'exploitation de l'indépendance permet d'éviter certains calculs redondants, en revanche, DDP n'exploite pas le fait que la même composante connexe peut être induite par une autre affectation $\mathcal{A}'$. Dans ce cas, DDP effectuera le même calcul autant de fois que cette composante apparaîtra. L'algorithme implémenté dans *cachet* [Sang et al., 2004]

améliore DDP en enregistrant le résultat trouvé pour une composante connexe afin de l'utiliser ultérieurement si besoin.

sharpsat L'inconvénient majeur de l'implémentation de l'enregistrement des solutions pour une composante connexe est le coût en espace. Ce problème est abordé par *sharpsat* [Thurley, 2006] qui propose plusieurs idées afin de réaliser un enregistrement plus compact. Ces techniques ont permis de réduire massivement l'espace mémoire requis par rapport à *cachet* et d'augmenter ainsi l'efficacité. En outre, *sharpsat* utilise des techniques « look-ahead » plus sophistiquées qui semblent accroître l'efficacité du comptage du nombre de modèles d'une instance. D'ailleurs, dans [Davies and Bacchus, 2007], Davies et Bacchus ont montré qu'effectuer une analyse plus poussée à chaque nœud de l'arbre de recherche peut rendre le comptage plus rapide. En effet, cela simplifie davantage la formule, permet de détecter les composantes connexes plus efficacement et peut même augmenter la décomposabilité du problème. Plus récemment, dans [Lagniez and Marquis, 2017b], les auteurs ont montré expérimentalement que l'emploi des techniques de prétraitement plus puissantes et consommant plus de temps pour #SAT que pour SAT est tolérable. En contrepartie, ils permettent de diminuer le temps nécessaire pour le comptage des modèles.

2.3.1.3 Compilation

Une approche différente consiste à compiler la formule CNF ou l'instance CSP en une autre forme logique à partir de laquelle le nombre de modèles pourrait être déduit plus facilement. Dans ce type d'approches, la complexité temporelle est polynomiale par rapport à la taille de la nouvelle formule. L'intérêt de la compilation ne se limite pas au comptage. Plus généralement, la formule obtenue doit permettre de répondre plus efficacement à certaines demandes [Darwiche and Marquis, 2001, 2002] initialement NP-difficiles. Comme signalé dans [Freuder and O'Sullivan, 2014], ce problème est particulièrement critique pour les applications en ligne comme dans la configuration des logiciels [Junker, 2006] ou les systèmes de recommandation [Cambazard et al., 2010] où les demandes envoyées à la volée par les utilisateurs doivent être satisfaites en temps réel. La plupart des travaux se sont alors focalisés sur la recherche de nouveaux langages cibles permettant d'offrir de telles garanties comme dans [Subbarayan et al., 2007; Fargier and Marquis, 2009; Darwiche, 2011]. Par la suite, nous présentons quelques langages parmi les plus connus.

(O)BDD Une formule peut être convertie en un BDD (*binary decision diagram*) [Bryant, 1986]. Il s'agit d'un graphe acyclique orienté et enraciné qui permet de représenter une fonction booléenne. Il comporte deux types de nœuds : deux nœuds terminaux 0 et 1 et des nœuds de décision étiquetés par une variable booléenne. Chaque nœud de décision possède deux fils, un correspond à son affectation par 0 et l'autre correspond à son affectation par 1. Le comptage du nombre de modèles revient à parcourir le BDD à partir du nœud terminal 1. Un BDD est dit ordonné (on parle alors d'OBDD) si les variables apparaîssent dans le même ordre dans tous les chemins partant de la racine. Un compilateur visant le langage OBDD est proposé dans [Huang and Darwiche, 2004] où des garanties sur les complexités théoriques sont fournies en plus d'une borne supérieure sur la taille de l'OBDD. En pratique, il a montré son efficacité vis-à-vis des compilateurs traditionnels.

d-DNNF (compilateurs *c2d*, *Dsharp* **et** *D4***)** Dans [Darwiche, 2001a], Darwiche présente le compilateur *c2d* qui transforme la formule CNF en une formule NNF (*negative normal form*) déterministe et décomposable d-DNNF [Darwiche, 2004]. Une formule NNF est une disjonction de conjonctions de littéraux. Elle peut être représentée sous

forme d'un graphe acyclique où l'étiquette de chaque puits (sommet n'ayant pas de fils) est un littéral et les étiquettes des autres sommets sont les opérateurs AND et OR. Elle est dite *décomposable* si les sous-arbres correspondants aux fils d'un sommet AND sont disjoints au sens de l'ensemble de ses littéraux. Elle est dite *déterministe* si les sous-arbres correspondants aux fils d'un sommet OR induisent des formules qui ne peuvent pas être satisfaites simultanément. Ces deux propriétés permettent le calcul du nombre de modèles d'une formule en parcourant l'arbre des puits vers la source. Le nombre de modèles associé au sommet source est le nombre exact de modèles de la formule. Pour y parvenir, à chaque littéral est associé le nombre 1. Pour chaque sommet AND (resp. OR), le nombre de modèles correspondant est la multiplication (resp. la somme) de nombre de modèles de chacun de ses fils. Dans [Muise et al., 2012], un autre compilateur *Dsharp* visant le langage d-DNNF est présenté. Selon les auteurs, il est généralement plus rapide que c2d tandis que les formules compilées sont souvent de taille comparable. Finalement, un nouveau compilateur CNF vers d-DNNF appelé *D4* est proposé dans [Lagniez and Marquis, 2017a]. Les expérimentations conduites montrent que son temps de compilation est généralement inférieur à celui de *Dsharp* et *c2d* aussi bien que la taille des formes d-DNNF compilées. Au niveau des instances résolues vis-à-vis du comptage, *D*4 résout le plus grand nombre d'instances suivi par *c2d*.

SDD Dans [Darwiche, 2011], Darwiche propose un nouveau compilateur visant le langage (pour *Sentential Decision Diagram*). L'objectif de ce langage est de conserver des propriétés intéressantes du langage OBDD comme la canonicité tout en étant plus général en termes de traitabilité. Il permet également de générer une forme compilée de taille en $O(exp(w))$ avec w la tree-width de la CNF en entrée qui est plus petite que la path-width sur laquelle se base un OBDD. Notons que OBDD est un sous-ensemble de SDD qui est un sous-ensemble de d-DNNF. En pratique, les expérimentations ont montré que la représentation en SDD est généralement plus compacte que celle en OBDD. À l'instar de OBDD, SDD permet également le comptage de modèles. C'est aussi le cas d'un autre langage sous-ensemble de d-DNNF qui est FBDD [Gergov and Meinel, 1994]. Comme d'autres langages qui sont valides pour le comptage des modèles sans être sous-ensemble de d-DNNF, nous citons le langage EADT (*extended affine decision trees*) [Koriche et al., 2013].

MDDG (compilateur *cn2mddg*) Au-delà des formules CNF, pour manipuler des instances CSP et les compiler en langage Decision-DNNF [Oztok and Darwiche, 2014] (langage strictement inclus dans d-DNNF) par exemple, le moyen le plus connu est de suivre un schéma de traduction-compilation. En effet, le réseau de contraintes donné en entrée est d'abord encodé en formule CNF. Ensuite, en exploitant un compilateur comme *c2d* [Darwiche, 2004] ou *Dsharp* [Muise et al., 2012] nous obtenons une représentation Decision-DNNF. Le premier inconvénient de cette approche réside dans le grand nombre de variables booléennes de la formule CNF générée. En outre, la première étape, i.e. l'encodage, induit une perte de la structure initialement présente dans le réseau de contraintes dûe au format CNF. Une approche différente a été alors proposée dans [Koriche et al., 2015]. Dans ce papier, les auteurs proposent le compilateur *cn2mddg* qui compile directement une instance CSP en langage MDDG (*Multivalued decomposable decision graph*). MDDG est une extension du langage Decision-DNNF aux domaines non booléens. Notons que le format donné en entrée à *cn2mddg* est le format XCSP 2.1 [Roussel and Lecoutre, 2009]. Malgré l'augmentation du niveau de généralité obtenu en acceptant des domaines non booléens, les algorithmes polynomiaux utilisés pour l'énumération de solutions, le comp-

tage de solutions ou d'autres problèmes peuvent être trivialement étendus au cas MDDG. Les expérimentations évaluant *cn2mddg* montrent qu'il est plus robuste du fait qu'il arrive à compiler des instances CSP que le schéma traduction-compilation n'y parvient pas. En outre, selon les auteurs, le temps requis pour la compilation de l'instance CSP en MDDG est plus petit que celui nécessaire pour la compilation de CNF en Decision-DNNF. Non seulement du temps est économisé, mais encore, la taille de la représentation MDDG est souvent plus compacte que celle de la représentation d-DNNF. Les auteurs précisent aussi que les techniques implémentées dans le compilateur comme l'heuristique de choix de variables ou les techniques d'enregistrement qui visent à éviter la duplication des sous-parties identiques de la représentation compilée, tirent profit de la structure de l'instance CSP. Cette structure est au contraire beaucoup moins présente dans la formule CNF. Les heuristiques de branchement font l'objet d'une étude [Lagniez et al., 2017] qui montre en particulier que les heuristiques de choix de variables favorisant la décomposabilité du graphe sont prioritaires lorsque le langage MDDG est visé.

2.3.1.4 Méthodes d'approximation

Les méthodes de comptage exactes attaquent ce problème en explorant exhaustivement tout l'espace de recherche. Le fait qu'il soit #P-complet laisse peu d'espoir ainsi pour un passage à grande échelle. En plus, de nombreuses applications du comptage de solutions ne requièrent pas de connaître le nombre exact de solutions et peuvent se contenter d'une approximation de ce nombre exact. C'est ainsi qu'une approche différente est apparue qui consiste à estimer le nombre de solutions plus rapidement. Deux aspects sont considérés : la qualité de l'estimation et la confiance associée à l'estimation reportée. Une méthode approximative associée à *#BTD*, *Approx#BTD*, est donnée dans [Favier et al., 2009]. Elle fournit un majorant du nombre de solutions. Les auteurs de [Gogate and Dechter, 2008] proposent une méthode qui se base sur une décomposition AND/OR et sur l'échantillonnage de l'espace de recherche. Cette méthode fournit une borne inférieure sur le nombre de solutions avec un intervalle de confiance à pourcentage élevé. Ce type de méthode peut malheureusement fournir une borne inférieure nulle pour de gros problèmes et peut nécessiter un réglage de paramètres très coûteux en temps. Une méthode différente se base sur l'ajout des contraintes XOR en SAT dans [Gomes et al., 2006] et en CSP dans [Gomes et al., 2007b]. Cependant, le problème résultant n'est pas forcément plus simple à résoudre. Une autre approche est présentée dans [Pesant, 2005] et consiste à donner une borne supérieure du nombre de solutions en se basant sur une approximation du nombre de solutions de chaque contrainte. Dans [Bailleux and Chabrier, 1996], les auteurs proposent une méthode testée pour les instances SAT mais qui peut s'appliquer facilement aux instances CSP. Elle permet d'obtenir une borne inférieure sans garantie quant à la borne supérieure. D'autres méthodes ont été proposées dans le cadre de #SAT comme la méthode de Karp et Luby [Karp and Luby, 1985], *ApproxCount* [Wei and Selman, 2005], *SampleMinisat* [Gogate and Dechter, 2007], *SampleCount* [Gomes et al., 2007a], *BPCount* et *MiniCount* [Kroc et al., 2008] et la méthode de Lerman et Rouat [Lerman and Rouat, 1999].

2.3.2 Bilan

Le premier constat à faire est la rareté des travaux visant #CSP. La plupart des travaux réalisés dans le cadre du comptage exact concerne le problème #SAT ou la compilation. L'exploitation des compteurs SAT pour la résolution d'instances #CSP n'est pas directe

vu que ces dernières doivent être converties en format CNF. Exprimer l'instance CSP en CNF fait perdre à l'instance sa structure présente initialement. En outre, la conversion peut être particulièrement coûteuse en temps et/ou en espace. Au niveau des compilateurs, à part *cn*2*mddg*, tous les autres compilateurs nécessitent ainsi de convertir d'abord l'instance CSP en format CNF avant d'être compilée en langage cible. Quant aux méthodes d'approximation, malgré la simplification de l'objectif des méthodes exactes, plusieurs problèmes apparaissent notamment en termes de qualité de borne fournie et de niveau de confiance correspondant. Ainsi, il semble judicieux de tenter d'améliorer les méthodes exactes pour #CSP. Vu l'intérêt majeur des méthodes structurelles vis-à-vis du problème du comptage, nous nous intéressons dans le cadre de cette thèse à améliorer les méthodes exactes structurelles notamment $\#BTD$.

2.4 Problème de satisfaction de contraintes pondérées : WCSP

Malgré le très large éventail de problèmes qu'il couvre, le cadre CSP se trouve dans l'impuissance de modéliser certaines notions en pratique. Par exemple, il se peut qu'il soit impossible de satisfaire simultanément toutes les contraintes ou qu'il existe des contraintes contradictoires. Dans ce cas, la modélisation peut être raffinée en choisissant d'ignorer certaines contraintes. En outre, même si, au contraire, le problème admet plusieurs solutions, ces différentes solutions n'ont pas forcément le même poids dans le sens où l'utilisateur pourrait avoir une préférence pour l'une par rapport à l'autre. Dans ces cas, les contraintes exprimeraient plutôt une propriété désirée et non pas une nécessité absolue. Le but consiste alors à éviter le plus possible de violer ces contraintes. Des exemples appuyant ces cas de figure existent dans plusieurs domaines comme la planification, l'allocation de ressources [Cabon et al., 1999], le routage des véhicules, les enchères combinatoires [Sandholm, 2002], la bioinformatique et le raisonnement probabiliste [Pearl, 1988] ... C'est ainsi que le cadre CSP a été étendu afin de prendre en compte ces contraintes dites *souples* [Schiex et al., 1995; Gale and Sotomayor, 1985]. Au contraire, une contrainte devant être nécessairement satisfaite est dite une contrainte *dure*. Nous parlons ainsi des CSP valués (VCSP) [Schiex et al., 1995] qui constituent un cadre générique englobant de multiples formalismes [Schiex, 1992; Rosenfeld et al., 1976; Dubois et al., 1993; Ruttkay, 1994; Freuder and Wallace, 1992; Fargier and Lang, 1993; Fargier et al., 1995, 1993]. La cadre CSP a ainsi gagné considérablement en expressivité.

En particulier, le problème de satisfaction de contraintes pondérées (WCSP) [Freuder and Wallace, 1992; Schiex, 2000; Larrosa, 2002] va faire l'objet de cette partie. Il s'agit d'une sous-classe des CSP valués. À travers la notion de pondération des contraintes, elle intègre la préférence et le souhait. Elle est basée sur la notion de *coût*. Ainsi, une affectation donnée est caractérisée par son coût. Plus le coût est élevé, moins l'affectation est préférable. Le fait que les coûts des différentes contraintes sont combinés grâce à un opérateur qui n'est pas *idempotent* (dont le nombre d'applications a un effet sur le résultat), n'est pas sans conséquences sur les méthodes de résolution ou sur le filtrage pouvant être réalisé. En plus, la recherche n'est pas interrompue, dans le cas du problème WCSP, au premier échec comme dans le cas du problème CSP. L'extension des méthodes initialement proposées dans le cadre CSP n'est pas alors directe. Nous choisissons de travailler sur ce problème puisqu'il est simple, représentatif et puisque à partir de ce problème nous pouvons déjà rencontrer les difficultés des autres CSP valués mentionnées ci-dessus. Nous nous intéressons alors dans cette partie aux méthodes de filtrage et de résolution conçues dans le cadre WCSP. En particulier, nous nous focalisons notamment sur les méthodes structurelles que nous visons à améliorer dans le cadre de cette thèse.

2.4.1 Formalisme

Nous débutons par une définition formelle du problème de satisfaction de contraintes pondérées (WCSP) [Freuder and Wallace, 1992; Schiex, 2000; Larrosa, 2002]. Une instance WCSP se base sur une structure de valuation $\mathcal{S}(k)$ qui est un triplet $(\{0, 1, \ldots, k\}, \oplus, \geq)$ où :

- $k \in \{1, \ldots, \infty\}$ tel que k représente le coût maximal pouvant être attribué,
- $\oplus$ est définie comme $a \oplus b = min\{k, a + b\}$,
- $\geq$ est l'ordre standard sur les entiers naturels.

Le premier élément de ce triplet est l'ensemble de coûts pouvant être ordonnés par $\geq$. Les coûts maximum et minimum sont respectivement k et 0. L'opération $\oplus$ est utilisée pour combiner les coûts. Un opérateur $\ominus$ est aussi définie pour compenser les valeurs additionnées. $\ominus$ est définie comme $a \ominus b = a - b$ si $a < k$ et k sinon [Schiex, 2000].

Définition 58 *Une instance WCSP est un triplet (X, D, W) avec :*

- $X = \{x_1, x_2, \ldots, x_n\}$ *l'ensemble de n variables*
- $D = \{D_{x_1}, D_{x_2}, \ldots, D_{x_n}\}$ *l'ensemble des domaines de variables comme dans le cas d'un CSP*
- $W = w_\emptyset \cup W_{unaires} \cup W^+$ *l'ensemble de fonctions de coût de taille m qui est l'union de :*
  - $w_\emptyset$ *le coût devant être payé par toute affectation,*
  - $W_{unaires} = \{w_1, \ldots, w_i, \ldots, w_n\}$ *avec w_i une fonction unaire associée à la variable x_i telle que $i \in \{1, \ldots, n\}$ et $w_i : D_{x_i} \rightarrow \{0, 1, \ldots, k\}$ associe à chaque valeur $v \in D_{x_i}$ un coût entre 0 et k,*
  - W^+ *représente le reste des fonctions de la forme $w_{i_1,\ldots,i_p}$ définie sur les variables $\{x_{i_1}, x_{i_2}, \ldots, x_{i_p}\}(p \geq 2)$ (toutes les fonctions de cette forme ne sont pas forcément définies) telle que $w_{i_1,\ldots,i_p} : D_{x_{i_1}} \times D_{x_{i_1}} \times \ldots D_{x_{i_p}} \rightarrow \{0, 1, \ldots, k\}$ associe à chaque tuple un coût entre 0 et k.*

  Le coût k correspond à une affectation complètement interdite (exprimant une contrainte dure).

Notons que si $k = 1$ une instance WCSP est réduite au cas classique d'une instance CSP.

2.4.2 Sémantique

Nous regardons dans cette partie la sémantique associée à une instance WCSP. En outre, nous remarquerons l'augmentation du niveau d'expressivité par rapport au cadre CSP.

Définition 59 *Le coût d'une affectation $\mathcal{A}$ telle que $X_\mathcal{A} \subseteq X$ est définie par :*
$w_\emptyset \oplus \sum_{x_i \in X_\mathcal{A}} w_i(\mathcal{A}[x_i]) \oplus \sum_{\{x_{i_1},\ldots,x_{i_p}\} \subseteq X_\mathcal{A}} w_{i_1,\ldots,i_p}(\mathcal{A}[x_{i_1} \ldots x_{i_p}])$

Si le coût d'une affectation complète est nulle, l'instance est cohérente au sens CSP du terme.

Nous définissons maintenant la notion de cohérence dans la cadre de l'optimisation.

Définition 60 *Étant donnée une instance WCSP $P = (X, D, W)$, une affectation $\mathcal{A}$ est cohérente ssi le coût de $\mathcal{A}$ est strictement inférieur à k.*

Le but du problème WCSP consiste à *trouver une affectation complète cohérente ayant le coût minimum*. Ce problème est NP-difficile.

Revenons maintenant sur l'exemple 1.

Exemple 2 *Robert ajoute une nouvelle contrainte c_5 à son instance P : $c_5 = (\{x_2, x_3\}, x_2 = x_3)$. Malheureusement, son instance devient ainsi incohérente. Il ne peut donc pas satisfaire simultanément toutes les contraintes. Il établit alors des priorités : il préfère ne pas satisfaire la contrainte c_i plutôt que de ne pas satisfaire c_{i+1} avec $i \in \{1, \ldots, 4\}$. Il associe donc un coût de i à chaque tuple interdit de c_i. En revanche, il n'accorde pas d'importance à la somme d'argent qu'il donne à chaque enfant. Une solution optimale a un coût de* 1 *et consiste à donner au premier fils* 20 *euros, au deuxième fils* 1 *euro, au 3ème fils* 1 *euro, au 4ème fils* 2 *euros, au 5ème* 1 *euro et au dernier* 5 *euros. En exploitant le cadre WCSP, Robert pourrait exprimer d'autres souhaits comme le fait de réduire l'écart entre les sommes d'argent perçues par ses enfants par exemple.*

Nous définissons maintenant l'équivalence entre deux instances WCSP.

Définition 61 *Deux instances WCSP P et P' définies sur le même ensemble de variables sont dites équivalentes si elles possèdent la même distribution de coût sur les affectations complètes. Ainsi, quelle que soit l'affectation totale $\mathcal{A}$ le coût associé à $\mathcal{A}$ dans P est égal à celui de $\mathcal{A}$ dans P'.*

Cette notion d'équivalence est essentielle pour les notions de cohérence locale et les méthodes de filtrage définies dans le cadre du problème WCSP.

2.4.3 Cohérences locales et filtrage

Dans cette partie, nous regardons de plus près les cohérence souples, les techniques de filtrage et les algorithmes de maintien de cohérence qui ont été étendus à partir du cadre CSP pour s'adapter au cadre WCSP. La valeur de la fonction de coût d'arité nulle $w_\emptyset$ est très importante pour la recherche de solutions. Elle permet d'élaguer des sous-problèmes ayant une borne inférieure plus grande que la meilleure solution trouvée jusque-là.

2.4.3.1 Opérations préservant l'équivalence

Tout d'abord, nous allons regarder les opérations qui transforme une instance WCSP en une autre instance WCSP qui lui est équivalente. Une telle opération est appelée *EPT* (pour *Equivalent Preserving Transformation*) [Cooper and Schiex, 2004]. L'opération de base est l'opération *shift* (algorithme 2.6) qui consiste à déplacer un coût α entre une fonction de coût $w_{Y'}$ et un tuple t_Y tel que $Y \subset Y'$. Le coût α peut être positif ou négatif. En revanche, ces déplacements de coût ne doivent pas créer des coûts négatifs dans le problème. C'est pourquoi, toute addition et soustraction réalisée doit garantir que le résultat est positif ou nul. Le coût associé au tuple t_Y est alors incrémenté de α (ligne 1). Cette incrémentation est compensée pour tout tuple $t_{Y'}$ dont sa projection sur Y est égal à t_Y en retranchant α de son coût associé. Si α est positif, ce déplacement constitue une opération de *projection*, sinon il s'agit d'une opération d'*extension*. Notons que si le résultat d'une opération atteint k cela permet de détecter des tuples incohérents. Dans [Cooper and Schiex, 2004], les auteurs proposent trois opérations de cohérence d'arc souple à la base de *shift* :

Algorithme 2.6 : Shift $(t_Y, w_{Y'}, \alpha)$

Entrées : Un tuple t_Y, une fonction de coût $w_{Y'}$, un coût α

1 $w_Y(t_Y) \leftarrow w_Y(t_Y) \oplus \alpha$
2 **pour chaque** $t_{Y'}$ *tel que* $t_{Y'}[Y] = t_Y$ **faire**
3 $\quad w_{Y'}(t_{Y'}) \leftarrow w_{Y'}(t_{Y'}) \ominus \alpha$

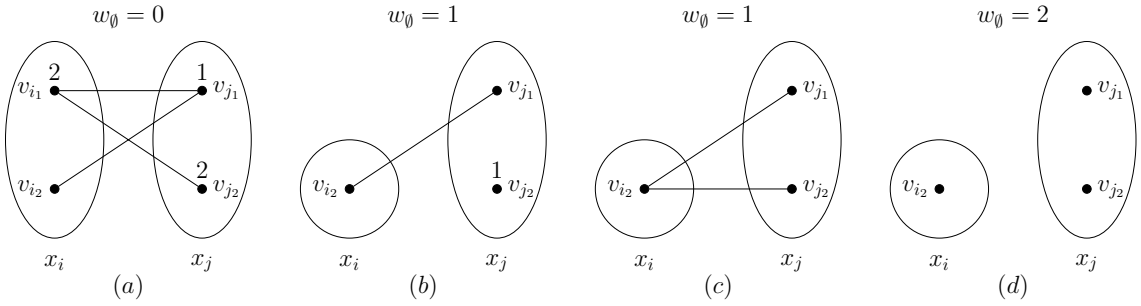


FIGURE 2.5 – Quatre instances WCSP équivalentes.

- $Project(w_Y, x_i, v_i, \alpha) = shift((x_i, v_i), w_Y, \alpha)$ avec $\alpha > 0$ qui projette la valeur α de la fonction de coût w_Y au couple (x_i, v_i).
- $Extend(x_i, v_i, w_Y, \alpha) = shift((x_i, v_i), w_Y, -\alpha)$ avec $\alpha > 0$ qui étend la valeur α du couple (x_i, v_i) à la fonction de coût w_Y.
- $UnaryProject(x_i, \alpha) = shift(\emptyset, w_i, \alpha)$ avec $\alpha > 0$ qui envoie un coût de w_i à $w_\emptyset$.

L'application de l'opération *shift* à une instance WCSP P produit une instance WCSP P' équivalente à P [Schiex, 2000].

Exemple 3 *Considérons l'exemple de la figure 2.5. La figure 2.5(a) montre deux variables x_i et x_j possédant chacune deux valeurs dans son domaine : v_{i_1} et v_{i_2} pour x_i et v_{j_1} et v_{j_2} pour x_j. Une fonction de coût binaire w_{ij} lie x_i et x_j. Les couples de valeurs associées à x_i et x_j ont soit un coût nul, soit un coût de 1 et, dans ce cas, les valeurs correspondantes sont liées par une arête. Par exemple dans la figure 2.5(a), le couple (v_{i_1}, v_{j_2}) a un coût de 1 tandis que (v_{i_2}, v_{j_2}) a un coût nul. À chaque valeur est associée un coût indiqué en haut de la valeur. L'absence d'annotation signifie que le coût unaire associé à la valeur en question est nul. Le coût associé par exemple à v_{j_2} est 2. Finalement la valeur de $w_\emptyset$ est aussi montrée. Nous supposons que $k = 3$. Les figures 2.5(b), (c), (d) montrent trois autres instances WCSP équivalentes. En effet, nous pouvons déduire celle de la figure 2.5(b) en remarquant que nous pouvons appliquer $UnaryProject(x_j, 1)$ ce qui fait augmenter la valeur de $w_\emptyset$ à 1. En outre, nous pouvons constater que nous pouvons ensuite appliquer $Project(w_{ij}, x_i, v_{i_1}, 1)$ ce qui fait augmenter le coût associé à v_{i_1} à 3. En atteignant la valeur maximale, nous pouvons déduire que cette valeur est incohérente. Elle ne sera alors plus représentée dans les instances WCSP suivantes. Ensuite, nous remarquons que nous pouvons appliquer $Extend(x_j, v_{j_2}, w_{ij}, 1)$ à l'instance WCSP de la figure 2.5(b) pour obtenir celle de la figure 2.5(c). Finalement, nous pouvons appliquer $Project(w_{ij}, x_i, v_{i_2}, 1)$ suivie de $UnaryProject(x_i, 1)$ et déduire l'instance WCSP de la figure 2.5(d). En appliquant alors une série de transformations préservant l'équivalence des instances WCSP la fonction $w_\emptyset$ a désormais une valeur de 2. Cela va permettre essentiellement de renforcer la*

capacité de la recherche à élaguer des sous-problèmes inutiles. Notons que la valeur v_{i_1} non montrée dans les trois dernières WCSP peut être effectivement supprimée en maintenant une cohérence locale qui est la cohérence de nœud *que nous allons rappeler ci-dessous.*

2.4.3.2 Cohérences souples

À l'instar des cohérences dures, les cohérences souples visent à produire une instance WCSP qui est équivalente à l'instance d'origine mais ayant un meilleur $w_\emptyset$. Le but est alors d'améliorer la performance des méthodes de résolution en raffinant $w_\emptyset$ et en augmentant la capacité à élaguer des valeurs incohérentes. Étant donnée une notion de cohérence locale souple, une valeur ou un tuple sont dits cohérents s'ils satisfont cette cohérence et incohérents sinon.

Cohérence de nœud(NC) La plus simple des cohérences locales est la cohérence de nœuds qui est définie pour les coûts unaires.

Définition 62 *[Larrosa, 2002] Une variable x_i satisfait la cohérence de nœud (NC) ssi $\forall v_i \in D_{x_i}$, $w_i(v_i) + w_\emptyset < k$ et il existe une valeur $v_i \in D_{x_i}$ telle que $w_i(v_i) = 0$. v_i est alors le nœud support de x_i. Une instance WCSP est nœud cohérente ssi toutes ses variables satisfont la cohérence de nœud.*

Par exemple, l'instance WCSP de la figure 2.5(a) n'est pas nœud cohérente puisque x_j ne satisfait pas cette propriété. En revanche, l'instance WCSP de la figure 2.5(b) est nœud cohérente. Cette propriété permet de supprimer des valeurs comme v_{i_1} lorsque son coût atteint la valeur de 3 ou d'augmenter $w_\emptyset$ en projetant le coût 1 de x_j à $w_\emptyset$ pour l'instance WCSP de la figure 2.5(a). Nous considérons alors par la suite que la valeur v_{i_1} est supprimée de D_{x_i}. NC peut être maintenu en $O(n.d)$.

Cohérences d'arc souples : les supports La notion de cohérence d'arc souple implique un ensemble de variables et une fonction de coût les liant. Elle se base sur la notion de *support simple* et de *support complet.*

Définition 63 *Étant données une variable x_i, une valeur $v_i \in D_{x_i}$ et une fonction de coût w_Y telle que $x_i \in Y$:*

- *un support d'arc simple de (x_i, v_i) pour w_Y est un tuple t_Y tel que $t_Y[x_i] = v_i$ et $w_Y(t_Y) = 0$.*
- *un support d'arc complet de (x_i, v_i) pour w_Y est un tuple t_Y tel que $t_Y[x_i] = v_i$ et $w_Y(t_Y) + \Sigma_{x_j \in Y, j \neq i} w_j(t_Y[x_j]) = 0$.*

Cohérence d'arc (AC) La plus simple des cohérences d'arc souples est la cohérence d'arc basée sur la notion de supports simples. D'après Larrosa et Schiex [Larrosa and Schiex, 2004], une valeur est arc cohérente si elle possède un support AC simple dans chaque fonction de coût. Une instance WCSP P satisfait la cohérence d'arc généralisée (GAC) ssi pour chaque variable x_i et chaque valeur $v_i \in D_{x_i}$ et pour chaque fonction de coût w_Y telle que $|Y| > 1$ et $x_i \in Y$, il existe un support simple pour (x_i, v_i) dans w_Y. P est GAC^* si P est GAC et NC. Par exemple, l'instance WCSP de la figure 2.5(a) n'est pas AC puisque (x_i, v_{i_1}) n'admet pas un support simple tandis que l'instance WCSP de la figure 2.5(d) est AC. Maintenir AC peut casser NC. Par exemple, l'instance WCSP de la figure 2.5(c) est NC mais pas AC. En lui appliquant $Project(w_{ij}, x_i, v_{i_2}, 1)$ elle devient

AC mais n'est plus NC, puisque la variable x_i possède une seule valeur v_{i_2} ayant un coût de 1. En lui appliquant $UnaryProject(x_i, 1)$ en plus, nous obtenons l'instance WCSP de la figure 2.5(d) qui est AC et NC donc AC^*. L'instance WCSP de la figure 2.5(d) est une fermeture AC de l'instance WCSP de la figure 2.5(a). AC^* peut être renforcé pour une instance WCSP binaire en $O(m.d^3)$ [Larrosa and Schiex, 2004].

Fermeture et terminaison du maintien de AC Le fait que l'opérateur combinant les coûts n'est pas idempotent induit des changements au niveau du maintien de la cohérence d'arc. Dans le cas d'une instance CSP, l'instance admet une fermeture AC unique. Ce n'est malheureusement pas le cas en général d'une instance WCSP. En effet, l'application non restreinte d'EPT ne converge généralement pas vers un point fixe unique [Schiex, 2000] et peut même ne pas terminer. Plusieurs fermetures possibles sont obtenues selon l'ordre dans lequel les opérations EPT sont réalisées. Ces fermetures peuvent être en plus de qualité différente. Dans ce cas, nous mesurons la qualité de la fermeture par la valeur de $w_\emptyset$ et nous espérons que cette valeur soit autant élevée que possible. En effet, plus cette valeur est grande, plus le pouvoir d'élaguage des sous-espaces inutiles sera important. Or, trouver le meilleur ordre d'application des EPT est un problème NP-complet [Cooper and Schiex, 2004]. Les travaux qui se sont intéressés à ce problème, ont visé à garantir la terminaison tout en cherchant à trouver la meilleure fermeture possible. Ainsi, ils ont proposé différentes restrictions heuristiques de AC dont le maintien termine toujours découlant en plusieurs variantes telles que AC^*, DAC^* [Cooper, 2003], $FDAC^*$ [Larrosa and Schiex, 2003], EAC^* et $EDAC^*$ [Givry et al., 2005] ... Ces heuristiques se sont avérées efficaces et ont permis de s'approcher d'une fermeture de cohérence d'arc *optimale* (qui maximise $w_\emptyset$) [Cooper et al., 2008]. Ultérieurement, la cohérence d'arc optimale ($OSAC$) a été définie dans [Cooper et al., 2007]. Elle se base sur la programmation linéaire et permet de précalculer en temps polynomial un ensemble d'EPT maximisant la valeur de $w_\emptyset$. Cependant, le maintien de $OSAC$ s'est avéré très coûteux en pratique. D'où, son utilisation a été limitée au prétraitement. Ainsi, la cohérence VAC [Cooper et al., 2008, 2010] a été proposée. Elle est plus exploitable en pratique qu'$OSAC$ au détriment de la puissance. Dans le cadre de cette thèse, la résolution d'instances WCSP exploitera la cohérence VAC en prétraitement et la cohérence $EDAC$ pendant la résolution.

2.4.4 Méthodes de résolution

Nous nous intéressons dans cette partie aux méthodes les plus connues pour la résolution des instances WCSP. Vu que le problème WCSP est une extension du problème CSP, la résolution des instances WCSP partage avec celle des instances CSP certaines similitudes comme les heuristiques de choix de la prochaine variable. Elles se basent également sur le principe *first-fail*. Les plus efficaces parmi les heuristiques définies sont également celles qui sont adaptatives comme l'heuristique $dom/wdeg$ [Boussemart et al., 2004].

Comme pour les autres problèmes, les contributions de cette thèse se focalisent sur les méthodes structurelles en particulier celles reposant sur une décomposition arborescente comme BTD. Nous balayons dans cette partie ensuite un spectre plus large qui ne serait pas limité aux méthodes structurelles.

2.4.4.1 Méthodes non structurelles

Dans cette partie, nous nous focalisons sur les méthodes non structurelles en partant de la méthode de base BB.

Branch and Bound (BB) Les instances WCSP ainsi que les problèmes d'optimisation peuvent être résolus par le biais du *Branch and Bound* (séparation et évaluation) [Land and Doig, 1960; Lawler and Wood, 1966]. Il s'agit d'une variante du backtracking (BT) adaptée à ce type de problèmes. Tout au long de la recherche, elle maintient deux valeurs primordiales : une borne inférieure *clb* (pour *current lower bound*) du coût d'une affectation complète contenant l'affectation courante et une borne supérieure *cub* (pour *current upper bound*) exprimant le coût de la meilleure solution trouvée jusque-là. Contrairement à BT qui effectue un retour en arrière lorsqu'une incohérence est rencontrée, BB le réalise lorsque $clb \geq cub$. Dans ce cas, le sous-arbre correspondant à l'espace de recherche restant à explorer afin d'étendre l'affectation courante est élagué. En effet, toute extension de cette affectation aura un coût supérieur au coût de la meilleure solution trouvée jusqu'alors et ne peut pas ainsi améliorer cette dernière. Une nouvelle valeur de la variable courante est alors essayée. Si la valeur courante est la dernière valeur, BB revient sur la dernière variable affectée. Lorsque l'affectation courante devient complète, *cub* est alors mise à jour convenablement. Plus précisément, *cub* correspond désormais au coût de l'affectation courante. L'efficacité de BB dépend notamment de la qualité de *clb* et *cub*. La qualité de *clb* dépend grandement des mécanismes mis en œuvre pour calculer cette borne. Dans ce contexte, les techniques de filtrage sont habituellement utilisées et ont permis d'améliorer considérablement la performance de l'approche BB. Elles peuvent se baser sur n'importe quelle propriété de cohérence locale. BB doit cependant accepter un coût de maintien de cohérence plus élevé proportionnel au niveau de cohérence désiré. Selon le type de parcours de l'arbre de recherche, nous obtenons divers algorithmes de résolution.

Depth First Search (DFS) et Best-First Search (BFS) DFS se base sur un parcours en profondeur d'abord. Il développe le nœud le plus profond parmi les nœuds de l'arbre de recherche non encore explorés. Son avantage réside dans sa complexité spatiale polynomiale. Il tire profit de l'incrémentalité de la cohérence locale lors du développement d'un nouveau nœud. Il est capable de donner une borne supérieure à chaque instant. En pratique, il a une bonne performance qui peut être considérablement améliorée par les heuristiques de choix de variables, par exemple. Associer DFS à des techniques de redémarrages comme la stratégie de Luby [Luby et al., 1993] peut améliorer significativement la performance de ce dernier et notamment son comportement *anytime* associé à la borne supérieure fournie [Luby et al., 1993; Allouche et al., 2015].

BFS développe au contraire le nœud ayant la borne inférieure la plus petite. Ce faisant, BFS est capable de fournir une borne inférieure à tout instant. Il a été aussi démontré que BFS ne développe pas plus de nœuds que DFS pour la même borne inférieure [Pearl, 1984]. Cependant, il a une complexité spatiale exponentielle et la solution optimale est l'unique solution fournie. Sa complexité spatiale vient du fait qu'il doit maintenir une liste de nœuds correspondants aux sous-problèmes non encore explorés. Cette liste est initialisée avec le nœud racine. Lorsque le nœud de borne inférieure minimale est choisie, il est remplacé par ses fils gauche et droite. Cette liste peut atteindre une taille en $O(d^n)$. L'incrémentalité du maintien de cohérence d'arc n'est pas offerte gratuitement comme dans le cas de DFS. En effet, chaque nœud devrait renfermer les structures de données permettant de réaliser cette incrémentalité. Un coût en $O(m.d)$ est alors nécessaire en plus par nœud. Concernant la complexité en temps, DFS et BFS sont tous les deux exponentiels en fonction du nombre de variables du problème.

Hybrid Best First Search (HBFS) [Allouche et al., 2015] L'idée de cet algorithme est d'élaborer un algorithme hybridant ces deux types de parcours afin d'obtenir les avantages

Algorithme 2.7 : HBFS (clb, cub)

```
   Entrées-Sorties : clb, cub : borne inférieure et supérieure courante
 1 open ← node(Σ = ∅, lb = clb)
 2 tant que open ≠ ∅ et clb < cub faire
 3 |   node ← dépiler(open)
 4 |   Restaurer l'état node.Σ, menant à la suite de décisions Σ et en appliquant la
   |    cohérence locale
 5 |   NodesRecompute ← NodesRecompute + node.profondeur
 6 |   cub ← DFS(Σ, cub, Z_hbfs)
 7 |   clb ← max{clb, lb(open)}
 8 |   si NodesRecompute > 0 alors
 9 |   |   si NodesRecompute/Nodes > β_hbfs et Z_hbfs ≤ N_hbfs alors
10 |   |   |   Z_hbfs ← 2.Z_hbfs
11 |   |   sinon
12 |   |   |   si NodesRecompute/Nodes < α_hbfs et Z_hbfs ≥ 2 alors
13 |   |   |   |   Z_hbfs ← Z/2
14 retourner (clb, cub)
```

de l'un et de l'autre. Plus précisément, il peut se contenter d'une complexité en espace plus raisonnable que celle que de BFS, profiter davantage de l'incrémentalité des cohérences locales comme DFS et être capable de donner à tout instant une borne inférieure et une borne supérieure de la solution optimale. Ainsi, son comportement *anytime* lui permet de refléter l'avancement de la recherche en permettant de suivre l'évolution de l'écart entre ces deux bornes. Le pseudo-code de $HBFS$ est montré par l'algorithme 2.7. La liste notée *open* est censée contenir les nœuds correspondants aux sous-problèmes qui ne sont pas encore explorés. Un nœud *node* contient l'ensemble de décisions Σ menant à ce nœud ainsi que la borne inférieure correspondante. La liste est initialisée, à la ligne 1, au nœud vide. La boucle des lignes 2-13 n'est stoppée que si la liste *open* devient vide ou que $clb \geq cub$. Dans ce cas, les bornes inférieures et supérieures courantes *clb* et *cub* sont retournées (ligne 14). Sinon, un nœud de borne inférieure minimale (comme pour BFS) est extrait de la liste (ligne 3). Pour diminuer le besoin en espace, le coût en $O(m.d)$ est évité au prix de refaire les décisions de $node.\Sigma$ en appliquant la cohérence locale (ligne 4). Certes, cela induit des propagations redondantes dont les auteurs proposent d'éviter une partie. DFS est ensuite lancé. La seule différence avec un DFS classique est la limitation du nombre de backtracks que peut effectuer ce dernier (ligne 6). Une fois la limite Z_{hbfs} atteinte, DFS redonne la main à BFS après avoir inséré les nœuds restants à explorer dans *open*. La borne inférieure *clb* est ensuite mise à jour convenablement (ligne 7). La question qui se pose est comment choisir la valeur de Z_{hbfs}. D'une part, cette valeur doit être suffisamment « grande » pour diminuer la redondance de propagation. D'autre part, elle doit être suffisamment « petite » afin d'offrir l'opportunité de choisir de meilleurs nœuds et de remettre en cause des décisions réalisées plus haut dans l'arbre de recherche. C'est ainsi qu'un compromis est proposé afin de maintenir le ratio des nœuds redondants ($NodesRecompute$) et des nœuds développés ($Nodes$) entre deux bornes α_{hbfs} et β_{hbfs}. Il utilise le nombre limite de backtracks N_{hbfs}. Dans leurs expérimentations, les auteurs proposent les valeurs suivantes : $\alpha_{hbfs} = 5\%, \beta_{hbfs} = 10\%, N_{hbfs} = 10\ 000$. Notons que $HBFS$ peut basculer en simple DFS en permettant un nombre illimité de backtracks afin de diminuer le besoin en espace mémoire. Son efficacité est telle qu'elle est considérée

maintenant comme la méthode de référence des méthodes non structurelles.

Limited Discrepancy Search (LDS) [Harvey and Ginsberg, 1995] Cette méthode a été initialement proposée pour résoudre le simple problème de décision CSP. Cet algorithme se base sur l'idée qu'une bonne heuristique ne pouvant pas résoudre rapidement un problème est susceptible de pouvoir aboutir plus facilement en modifiant un « petit » nombre de ses décisions. Il s'agit d'un algorithme basé sur du backtracking qui peut à un certain point de la décision ignorer la recommandation de l'heuristique et développer un nœud fils différent. C'est ce qu'on appelle une *divergence*. Si l'arbre de recherche est par exemple binaire et l'heuristique développe d'abord le nœud fils gauche, une divergence consiste à developper celui de droite. Le nombre de divergences est limité sur un chemin allant de la racine jusqu'une feuille. Au début, aucune divergence du chemin de l'heuristique n'est permise, ensuite au maximum une, puis deux et ainsi de suite... Si le nombre de divergences atteint la profondeur maximum de l'arbre de recherche, LDS explore exhaustivement l'arbre en entier. Les nœuds les plus hauts dans l'arbre de recherche sont prioritaires pour subir un changement de décision vu leur importance pour l'efficacité de la recherche. Comparée à d'autres méthodes de résolution de WCSP dans [Allouche et al., 2015] comme DFS et $HBFS$, LDS témoigne d'une très bonne performance en nombre d'instances résolues à l'optimum (un peu moins que $HBFS$). En plus, il s'avère qu'elle fournit la meilleure borne supérieure le plus souvent.

Poupées russes (RDS) [Verfaillie et al., 1996] Cette méthode se base sur la résolution de n problèmes emboîtés les uns dans les autres, à l'image de poupées russes. Étant donné un ordre total sur les n variables, la i-ème étape consiste à résoudre le problème constitué des i dernières variables. Pour résoudre, un problème RDS compte sur une méthode de type DFS. L'optimum ainsi que l'affectation accompagnante seront enregistrés. Lorsque le problème $i+1$ est résolu, l'optimum trouvé auparavant pour le problème i est exploité pour générer un minorant de bonne qualité. En effet, un minorant plus faible établi par un filtrage similaire au filtrage par cohérence de nœud est combiné à cet optimum pour fournir un minorant plus fort. Sa complexité est en $O(\frac{d}{d-1}.exp(n))$. Plusieurs améliorations de RDS sont proposées comme $SRDS$ [Meseguer and Sánchez, 2001], $OSRDS$ [Meseguer et al., 2002] et $TRDS$ [Meseguer and Sánchez, 2000].

2.4.4.2 Méthodes structurelles

Ces méthodes sont au centre d'intérêt de cette thèse pour le problème CSP, #CSP et aussi pour le problème WCSP. En particulier, nous nous intéressons à la méthode BTD qui sera évoquée dans cette partie. D'autres méthodes structurelles utilisées seront également rappelées. Leur principal intérêt réside dans leur complexité en temps qui est meilleure que celles des méthodes non structurelles.

BTD, Lc-BTD$^+$ [Terrioux and Jégou, 2003; Jégou and Terrioux, 2004b; Givry et al., 2006] BTD a été adapté au problème d'optimisation dans [Terrioux and Jégou, 2003; Jégou and Terrioux, 2004b]. Lorsqu'un cluster E_i ayant un cluster fils E_j est affecté, le sous-problème $P_j|\mathcal{A}[E_i \cap E_j]$ peut être résolu comme un sous-problème indépendant avec un majorant initial de k. L'optimum est alors enregistré afin d'éviter de visiter le même sous-problème une autre fois. Cependant, un tel majorant ne permet pas un élaguage efficace. En effet, l'optimum de ce sous-problème combiné avec le coût des clusters déjà affectés et des minorants déjà calculés pour les clusters non encore explorés peut largement

dépasser le coût de la meilleure solution trouvée jusque-là. Cela induit un effort inutile et un surcoût en temps et en espace. Dans [Givry et al., 2006], les auteurs proposent alors l'algorithme $Lc\text{-}BTD^+$ qui exploite une borne supérieure non triviale et qui combine BTD avec le maintien d'une propriété de cohérence locale souple notée Lc. Le nouveau majorant calculé garantit que l'optimum, une fois trouvée, sera suffisamment bon pour pouvoir s'intégrer dans une solution globale. Certes, cela n'est pas toujours faisable. Ainsi, soit un tel optimum est trouvé, soit l'algorithme montre que ce sous-problème est incohérent. Dans ce cas, $Lc\text{-}BTD^+$ déduit que le majorant initialement utilisé est un minorant qui sera enregistré et noté $LB_{P_j|\mathcal{A}}$. Bien qu'un sous-problème peut être visité plusieurs fois, cela ne modifie pas les complexités temporelles et spatiales de BTD. En effet, même si le problème $P_j|\mathcal{A}$ n'est pas résolu à l'optimum, à chaque visite le minorant est forcément amélioré. Le nombre de visites est ainsi borné par l'optimum du sous-problème lui même. L'intégration de la cohérence locale à BTD impose une gestion plus sophistiquée que lorsqu'elle est intégrée à un algorithme n'exploitant pas une décomposition. Tout d'abord, les projections et les extensions binaires entre deux fonctions de w_{ij} et w_i modifie la distribution de coûts et les optimums calculés si w_{ij} et w_i sont associés à deux sous-problèmes différents. Ainsi, une structure additionnelle est alors utilisée pour préciser le coût qui a quitté le sous-problème P_j une fois le séparateur $E_i \cap E_j$ affecté. Ce coût est noté $\overline{\Delta W}_{P_j|\mathcal{A}[E_i \cap E_j]}$. Cette structure entraîne un surcoût spatial en $O(m.n.d)$. De même, la projection unaire pourrait déplacer un coût entre w_i et $w_\emptyset$. Afin de résoudre ce problème, les auteurs proposent de localiser la fonction d'arité nulle et d'associer, à chaque cluster E_i, une fonction de coût locale $w^i_\emptyset$. Pour un sous-problème P_j, son minorant est obtenu en additionnant toutes les fonctions $w^p_\emptyset$ des clusters E_p de $Desc(E_j)$ et est noté $\overline{W}^j_\emptyset$. Ayant deux minorants, le minorant choisi pour un sous-problème P_j est le meilleur des deux, à savoir $lb(P_j|\mathcal{A}) = max\{LB_{P_j|\mathcal{A}} - \overline{\Delta W}_{P_j|\mathcal{A}[E_i \cap E_j]}, \overline{W}^j_\emptyset\}$. Ensuite, des règles sont définies pour le filtrage de valeurs. La coupe locale consiste à filtrer une valeur v_p de x_p dans P_j si $\overline{W}^j_\emptyset + w_p(v_p) \geq cub$ avec cub la borne supérieure courante de P_j en cours de résolution. Une règle meilleure exploite $lb(P_j|\mathcal{A}[E_i \cap E_j])$ pour le filtrage sous certaines conditions. Une coupe globale peut être utilisée conjointement. Cependant, certaines garanties sont perdues comme la complexité théorique. Les expérimentations ont montré que la méthode la plus robuste est obtenue en combinant l'utilisation des majorants améliorés, une cohérence locale forte (comme $FDAC$), une mémorisation des minorants et de leur correction et un filtrage basé sur une coupe locale.

RDS-BTD [Sanchez et al., 2009] En intégrant RDS à BTD, les auteurs visent à rendre la résolution d'un sous-problème encore plus informée du contexte global que dans $Lc\text{-}BTD^+$ afin d'éviter le plus possible la résolution des sous-problèmes ne contribuant pas à une solution globale. L'utilisation de RDS fournit des minorants plus forts pour chaque sous-problème et ainsi des majorants initiaux de meilleure qualité lorsque ce sous-problème est résolu par BTD. Au lieu de résoudre n sous-problèmes emboîtés, RDS résout $|E|$ sous-problèmes en parcourant l'arbre T en profondeur en commençant par les feuilles tout en remontant vers la racine. Un sous-problème P^{RDS}_j, au contraire de P_j, ne contient pas les variables de $E_i \cap E_j$ (E_j est un fils de E_i) et est constitué des fonctions de coût qui portent uniquement sur des variables $V_{Desc(E_j)} \backslash (E_i \cap E_j)$. L'optimum de P^{RDS}_j est alors un minorant de P_j qui ne dépend pas de l'affectation de $E_i \cap E_j$. En effet, P^{RDS}_j est une relaxation de P_j ce qui permet d'extraire un minorant puissant. Contrairement à RDS qui résout un sous-problème par DFS, $RDS\text{-}BTD$ compte sur BTD (la version $Lc\text{-}BTD^+$) pour les résoudre. L'exploitation de BTD permet la ré-utilisation des minorants enregistrés dans les itérations précédentes de $RDS\text{-}BTD$. Les complexités de BTD restent inchangés.

RDS-BTD a permis de résoudre une instance d'allocation de fréquence qui demeurait ouverte depuis 10 ans. Il s'agit de l'instance *scen07* du CELAR [Cabon et al., 1999].

BTD-HBFS [Allouche et al., 2015] La combinaison de $HBFS$ avec BTD dote BTD du comportement *anytime* implémenté par $HBFS$. $HBFS$ met à jour constamment la borne inférieure et la borne supérieure connues pour un problème. Ainsi, à un sous-problème est désormais associé non seulement la borne inférieure $LB_{P_j|\mathcal{A}}$ mais aussi la borne supérieure $UB_{P_j|\mathcal{A}}$. BTD est *anytime* puisque, pour tout j, $UB_{P_j|\mathcal{A}} < k$, il est capable de donner une solution globale en combinant les solutions disponibles de tous les sous-problèmes. En outre, l'amélioration permanente des bornes inférieures $LB_{P_j|\mathcal{A}}$ est exploitée dans les autres clusters afin de renforcer la capacité d'élaguage. À l'instar de $HBFS$ qui est basée sur DFS, BTD-$HBFS$ utilise BTD-DFS (DFS intégré à BTD [Allouche et al., 2015]). Lorsque le nombre de backtracks accordé à DFS est épuisé, BTD-DFS redonne la main à BTD-$HBFS$. Chaque cluster de la décomposition, ainsi que chaque sous-problème possède à son tour un nombre limite de backtracks. Supposons, par exemple, que le problème P_j est en cours de résolution et que BTD-DFS vient de passer la main à BTD-$HBFS$. Si la limite du nombre de backtracks accordée n'est pas dépassée, BTD-$HBFS$ rechoisit un nœud de la liste *open* associée à $P_j|\mathcal{A}[E_i \cap E_j]$ et redonne la main à BTD-DFS. Sinon la résolution de P_j est arrêtée. La résolution de P_j est également arrêtée si l'une des bornes inférieure ou supérieure est améliorée. Le fait d'interrompre la résolution de P_j rend la recherche plus dynamique et les bornes exploitables au plus tôt. BTD-$HBFS$ surclasse BTD-DFS au point d'être considérée comme la méthode référence des méthodes structurelles notamment lorsque la décomposition initialement calculée est modifiée en supprimant les séparateurs de grande taille. Sa complexité en temps est en $O(k.n.d^{w^+})$ et en espace en $O(k.n.d^{2w^+})$.

AND/OR search space [Marinescu and Dechter, 2005b,a, 2007] Cette méthode a été adaptée à la résolution des WCSP dans [Marinescu and Dechter, 2005b]. L'avantage recherché dans les espaces AND/OR est leur capacité à capturer les indépendances entre les différents sous-problèmes à travers un pseudo-arbre. À chaque arc est associé désormais la notion de *label* et, à chaque nœud, sa *valeur* qui peut être calculée récursivement depuis les feuilles jusqu'à la racine. La solution de coût minimum correspond à la valeur du nœud racine. À chaque nœud est également associée une fonction heuristique de calcul de borne inférieure permettant d'élaguer des sous-arbres inutiles. La complexité en temps de cet algorithme appelé *AOBB* (pour *AND/OR Branch and Bound*) est en $O(n.exp(m))$ et est linéaire en espace. En pratique, il améliore les algorithmes basés sur un espace OR classique. Dans [Marinescu and Dechter, 2005a], les auteurs étendent cet algorithme afin d'intégrer les enregistrements basés sur les contextes. Ainsi, ils explorent un graphe AND/OR et non pas un arbre AND/OR. Ces enregistrements sont très similaires de ceux faits dans le cadre de BTD. Dans [Marinescu and Dechter, 2007], AND/OR Branch and Bound a été combiné avec BFS, vu ses bénéfices par rapport à DFS. Les expérimentations ont montré que BFS AND/OR améliore significativement DFS AND/OR sur certains benchmarks. Malgré sa complexité en temps similaire à celle de BTD-DFS, l'espace requis par BFS AND/OR est un frein. En outre, l'algorithme est loin d'avoir un comportement *anytime* vu que BFS ne fournit que la solution optimale. Otten et Dechter évoquent aussi dans [Otten and Dechter, 2012] le manque de caractère anytime du comportement de DFS lorsqu'il exploite une décomposition du problème via un espace de recherche AND/OR. L'inconvénient principal réside dans la nécessité de résoudre à l'optimum tous les sous-problèmes indépendants induits par une décomposition sauf un avant de pouvoir construire

une solution globale. Ainsi, ils proposent l'algorithme $BRAOBB$ (pour *Breadth-Rotating AND/OR Branch and Bound*) qui combine à la fois la recherche en profondeur d'abord et la recherche en largeur. Ils introduisent la notion de rotation sur les différents sous-problèmes selon un style round-robin. L'algorithme traite chaque sous-problème sans le résoudre forcément à l'optimum avant de passer au prochain sous-problème. Cette approche améliore l'aspect anytime. Cependant, la résolution qui s'appuie toujours sur DFS ne peut produire de meilleures bornes inférieures. Cet algorithme est comparé avec BTD et $BTD\text{-}HBFS$ dans [Allouche et al., 2015]. Les expérimentations montrent que $BRAOBB$ est largement surpassé par les algorithmes BTD notamment par $BTD\text{-}HBFS$.

Pseudo-tree search [Larrosa et al., 2002] Dans [Larrosa et al., 2002], les auteurs étendent PTS [Freuder and Quinn, 1985] au problème WCSP. PTS hérite des méthodes de recherche leur complexité spatiale polynomiale et des méthodes de décomposition une complexité en temps exponentielle en la hauteur de l'arbre. L'adaptation de PTS au WCSP est faisable sans aucune difficulté. Cependant, selon les auteurs, une implémentation directe n'est pas compétitive. Le nouvel algorithme $PTS\text{-}BB$ exploite les indépendances créées une fois une variable x_i instanciée. Chaque sous-problème enraciné en un fils de x_i constitue un problème indépendant qui peut être résolu séparément. Les optimums des différents sous-problèmes peuvent être combinés afin de déduire l'optimum du sous-problème enraciné en x_i. Si x_i correspond au nœud racine, l'optimum est l'optimum global. Afin de résoudre chaque sous-problème, il est primordial de se doter d'une borne supérieure de bonne qualité. Pour y parvenir, $PTS\text{-}BB$ utilise le minimum d'une borne locale et d'une borne calculée en fonction de bornes inférieures des autres sous-problèmes. Une autre approche consiste à intégrer RDS [Verfaillie et al., 1996] à PTS [Larrosa et al., 2002]. Cette combinaison permet à RDS de tirer profit de la structuration en sous-problèmes indépendants offerte par le pseudo-arbre. De son côté, PTS bénéficie de la bonne qualité des minorants fournis par RDS.

Or-decomposition [Kitching and Bacchus, 2008] Dans [Kitching and Bacchus, 2008], les auteurs soulignent de nouveau l'inconvénient de traiter les différents sous-problèmes indépendamment. Bien que leur algorithme se basent sur une décomposition arborescente, plus de flexibilité est donnée à l'heuristique de choix de variables en lui permettant d'entrelacer l'affectation des variables appartenant à plusieurs sous-problèmes. Les informations récoltées auprès d'un sous-problème peuvent être exploités afin d'améliorer les bornes des autres sous-problèmes et réfuter éventuellement un sous-ensemble de ces derniers sans qu'ils soient forcément résolus à l'optimum. Certaines vertus de la décomposition demeurent exploitables comme la mise à l'écart des toutes les variables d'une composante résolue à l'optimum. Cependant, une composante connexe peut être visitée à plusieurs reprises avec la même affectation partielle sur ses variables. Si l'affectation partielle a un coût élevé, cela pourrait nuire au comportement anytime de l'algorithme dans la mesure où les autres composantes verront difficilement une solution de bonne qualité capturée. En outre, la borne inférieure d'une composante ne peut être mise à jour que lorsque la branche droite correspondante au nœud racine est explorée.

Bucket Elimination - BB [Larrosa and Dechter, 2003] Dans [Larrosa and Dechter, 2003], la méthode VES [Larrosa, 2000] combinant Bucket Elimination [Dechter and Pearl, 1987; Dechter, 1999] et une méthode de recherche telle que FC est adaptée au problème d'optimisation et appelée $BE\text{-}BB$. À l'instar de VES, la motivation principale d'une telle hybridation est la grande arité des contraintes pouvant être inférées par l'élimination

de certaines variables. Ainsi, *BE-BB* pourra n'éliminer que les variables pour lesquelles cette opération n'est pas très coûteuse. Un paramètre p associé à *BE-BB* spécifie la taille maximale de contraintes pouvant être générées. Il contrôle le compromis entre *BE* et *BB*. *BE-BB* est exponentielle en temps en p et dépend de la topologie du graphe de contraintes. Les expérimentations ont montré que cette association est efficace dans certains cas notamment pour des problèmes aléatoires et creux.

Méthode de Koster [Koster, 1999] Dans [Koster, 1999], Koster propose un nouvel algorithme de programmation dynamique pour la résolution des instances WCSP. Il se base sur une décomposition arborescente du graphe de contraintes du problème. Il exploite le fait que lorsque la tree-width du graphe est bornée par une constante, l'algorithme est polynomial. La brique essentielle de cet algorithme est qu'étant donnés un séparateur Y et une affectation sur ses variables, le problème initial est décomposé en plusieurs sous-problèmes indépendants. L'algorithme parcourt la décomposition depuis les feuilles vers la racine. Il trouve d'abord toutes les affectations d'un cluster feuille tout en les mémorisant. Une fois tous les clusters fils de E_i résolus, les affectations sur $V_{Desc(E_i)}$ peuvent être calculées. Pour y parvenir, les affectations sur les problèmes enracinés en chaque cluster fils de E_i ayant la même affectation sur les variables en commun sont combinées avec l'affectation des variables restantes de E_i. Koster précise que vu que E_i est considéré comme séparateur, toutes les affectations de $V_{Desc(E_i)}$ ne doivent pas être enregistrées. En effet, uniquement les affectations dont l'affectation sur E_i est différente seront enregistrées. En plus, pour chaque affectation de E_i, seule la meilleure affectation sur $V_{Desc(E_i)}$ est mémorisée. Une fois le cluster racine atteint, une solution optimale sera trouvée. Sa complexité en temps est en $O(n.d^{3w})$ et celle en espace est en $O(n.d^{w+1})$. Koster propose ensuite plusieurs techniques de réduction permettant de supprimer des sommets ou des arêtes du graphe de contraintes ou même certaines valeurs des domaines des variables en phase de prétraitement. L'exploitation des techniques de réduction a permis de résoudre certaines instances du problème d'allocation de fréquence [Cabon et al., 1999] inaccessibles jusqu'alors. Malheureusement, malgré l'utilisation des techniques de réduction, les ressources temporelles et spatiales restent insuffisantes pour pouvoir résoudre des instances d'une taille plus importante. D'une part, la largeur w^+ des décompositions calculées peut être très grande. D'autre part, le nombre de valeurs dans chaque domaine peut être très élevé. C'est ainsi que Koster [Koster, 1999] s'est focalisé sur la deuxième raison et a proposé de partitionner les domaines des variables et assigner un ensemble de valeurs à une variable plutôt qu'une seule valeur. Cette méthode s'est avérée meilleure sur des instances qui explosaient en mémoire sans toutefois réussir à surmonter l'obstacle de la mémoire requise pour toutes les instances.

2.4.5 Bilan

Nous nous sommes intéressés dans cette partie au cadre WCSP qui apporte notamment plus d'*expressivité* au cadre CSP. En raison des ressemblances qui existent entre ces deux cadres, certaines techniques utilisées pour la résolution des instances CSP sont directement exploitables pour la résolution des instances WCSP. Cependant, le fait que l'opérateur utilisé pour la définition d'une instance WCSP n'est pas idempotent, induit des changements significatifs sur d'autres éléments importants du solveur, comme le filtrage. Nous nous sommes focalisés, dans cette partie, sur les méthodes exploitant la structure du graphe de contraintes. Ces méthodes, comme dans le cas CSP, offrent des garanties théoriques intéressantes en temps. Les méthodes visitées tentent généralement de trouver des minorants de bonne qualité pour un sous-problème afin de renforcer la capacité

d'élaguage pendant la résolution. Différentes méthodes ont évoqué la nécessité de tenir la résolution d'un sous-problème informée de la résolution des autres sous-problèmes. En effet, la résolution d'un sous-problème à l'optimum sans tenir compte des informations récoltées auprès des autres sous-problèmes diminuerait la chance de participation de l'optimum trouvé à une solution globale. Il existe actuellement des méthodes de résolution efficaces en pratique comme $BTD\text{-}HBFS$ [Allouche et al., 2015] qui est considérée comme la référence des méthodes structurelles. Toutefois, comme dans le cas des autres méthodes structurelles, $BTD\text{-}HBFS$ reste emprisonné par la décomposition chargée de capturer cette structure. Dans cette thèse, nous visons alors à améliorer les méthodes structurelles de résolution d'instances WCSP notamment celles basées sur une décomposition arborescente comme BTD.

2.5 Conclusion

Le cadre CSP est suffisamment expressif pour pouvoir couvrir un champ large de problèmes réels, aléatoires ou académiques. Le formalisme WCSP est encore plus riche du fait de la notion de préférence qu'il permet de modéliser. La littérature abonde de travaux visant à résoudre ces deux problèmes ainsi que le problème du comptage découlant du formalisme CSP. Des méthodes de résolution classiques comme MAC [Sabin and Freuder, 1994] pour la résolution d'instances CSP et $HBFS$ [Allouche et al., 2015] pour la résolution d'instances WCSP ont prouvé leur efficacité en pratique. En revanche, leur complexité théorique en temps exponentielle en n est parfois un frein en vue du passage à l'échelle de ces méthodes. Les méthodes structurelles offrent de meilleures garanties théoriques temporelles, mais sont généralement moins compétitives vis-à-vis des méthodes énumératives. En termes de complexité spatiale, elles requièrent un espace mémoire plus important qui peut rendre certaines méthodes inopérantes. BTD [Jégou and Terrioux, 2003] est une méthode structurelle dont l'intérêt est tel qu'il peut résoudre des instances difficiles. Ces instances présentent souvent une structure intéressante capturée par la décomposition arborescente sur laquelle BTD est basé. L'efficacité de BTD dépend sûrement de la qualité de cette décomposition. Cette dernière est souvent calculée par le biais d'heuristiques dont $Min\text{-}Fill$ [Rose, 1972] constitue l'état de l'art, du moins dans la communauté CP. Elle vise essentiellement à minimiser la taille des clusters de la décomposition, le paramètre clé de la complexité théorique. Certains travaux ont mis l'accent sur d'autres critères comme la taille des intersections entre les clusters ou la connexité des clusters. Ce point nous intéressera particulièrement dans cette thèse. En effet, nous cherchons à calculer des décompositions qui capturent désormais des paramètres susceptibles d'être plus propices à une résolution efficace. BTD initialement proposé pour la résolution du problème CSP a été adaptée pour le dénombrement des solutions et l'optimisation tout en gardant ses propriétés principales. Il a permis de résoudre avec succès des instances de ces problèmes. BTD souffre néanmoins de certains problèmes en pratique, comme le fait d'être emprisonné par la décomposition sur laquelle il est basé. Cette thèse traite alors ce problème et tente de relâcher les contraintes imposées à BTD.

Dans le prochain chapitre, nous nous focalisons sur les méthodes de calcul de décomposition arborescente et nous présentons les principaux inconvénients des méthodes existantes. Nous proposons ainsi un nouveau cadre de calcul heuristique de décompositions qui vise essentiellement à permettre de capturer des caractéristiques qui ne se limitent pas à la taille des clusters.

Deuxième partie

Contributions

Chapitre 3

Calcul de décompositions arborescentes

Sommaire

3.1 Introduction

Nous avons déjà vu que les méthodes de résolution classiques de (W)CSP ont une complexité en temps théorique en $O(exp(n))$ avec n le nombre de variables du problème. Une deuxième approche consiste à exploiter la structure du problème en s'appuyant sur la notion de *décomposition arborescente*, par exemple. L'exploitation d'une décomposition permet de décomposer le problème original en plusieurs sous-problèmes indépendants et d'éviter beaucoup de redondances grâce à l'enregistrement de certaines informations pendant la résolution. D'un point de vue théorique, l'intérêt d'une telle approche réside dans sa complexité en temps qui est de l'ordre de $O(exp(w))$ avec w la largeur arborescente du réseau de contraintes. En pratique, ces approches sont particulièrement justifiées car il existe de nombreux problèmes réels pour lesquels w est significativement plus petit que n comme, par exemple, les problèmes d'allocation de fréquence radio [Cabon et al., 1999]. Nous avons aussi souligné que malheureusement, calculer une décomposition optimale est un problème NP-difficile. Ainsi, en pratique, la complexité en temps théorique est en $O(exp(w^+))$ avec $w^+ \geq w$ qui est une approximation de w qui correspond à la largeur de la décomposition employée. Alors, calculer efficacement une décomposition d'une largeur la plus proche possible de la largeur optimale est aujourd'hui un défi majeur. Par conséquent, les décompositions sont généralement calculées par le biais d'approches heuristiques. Ces approches ont clairement montré leur intérêt pratique par rapport aux méthodes exactes qui sont généralement inopérantes en pratique. Toutefois, notamment du point de vue de la résolution d'instances (W)CSP, ces heuristiques souffrent de multiples défauts que nous détaillons dans la section suivante. Ensuite, nous proposons, dans la section 3.3, un cadre général de calcul de décompositions, appelé *H-TD*, qui vise à minimiser, voire éviter ces défauts. Nous évaluons son intérêt pratique dans la section 3.4 avant de conclure.

3.2 Défauts des décompositions existantes

Les décompositions existantes, dont *Min-Fill* constitue l'heuristique de l'état de l'art pour la communauté CP et au-delà, visent notamment à minimiser la taille des clusters de la décomposition et ainsi la largeur w^+ de celles-ci. *Min-Fill* est surtout connue pour sa bonne approximation de la largeur arborescente w et par son temps de calcul qui reste raisonnable par rapport aux méthodes exactes de calcul de décompositions. Pour rappel, la décomposition issue de *Min-Fill* est calculée en deux étapes :

- La première étape consiste à trianguler le graphe G, c'est-à-dire à rajouter les arêtes nécessaires dans le but d'établir un ordre d'élimination parfait σ et d'obtenir un graphe triangulé G'_σ à partir de G. Pour construire l'ordre d'élimination parfait σ, *Min-Fill* ordonne les sommets de G de 1 à n en choisissant comme sommet suivant le sommet qui nécessite d'ajouter un minimum d'arêtes pour compléter son voisinage ultérieur. Généralement, *Min-Fill* casse les égalités d'une façon arbitraire. Après avoir rajouté les arêtes en question, *Min-Fill* continue à procéder de la même façon jusqu'à ce que tous les sommets de G soient numérotés. Cet algorithme est une instanciation spécifique de l'algorithme *Heuristique-H* de la partie 1.3.2.3. Il est montré par l'algorithme 3.1.

- La deuxième étape consiste, une fois le graphe G triangulé, à identifier les cliques maximales de G'_σ, grâce à σ, qui formeront chacune un cluster de la décomposition.

En ajoutant *a priori* moins d'arêtes, nous pouvons espérer produire des cliques maximales plus petites et par voie de conséquence une décomposition de largeur plus proche de

Algorithme 3.1 : Min-Fill (G)

Entrées : Un graphe $G = (X, C)$
Sorties : Un ordre d'élimination parfait σ, le graphe triangulé G'_σ

```
G'_σ ← G
Calcul du nombre d'arêtes nécessaires pour la complétion du voisinage de chaque sommet
pour i ← 1 à n faire
    Choisir un sommet non numéroté x de G'_σ qui nécessite d'ajouter un minimum d'arêtes pour compléter son voisinage non numéroté
    σ(x) ← i
    Compléter le sous-graphe induit par G'_σ[N_u(x)]
    Mettre à jour le nombre d'ajouts d'arêtes nécessaires pour chaque sommet non numéroté
retourner (σ, G'_σ)
```

l'optimum. Cependant, *Min-Fill*, comme toutes les méthodes de calcul de décompositions basées sur la triangulation, présente plusieurs inconvénients liés d'une part à la façon dont elle procède pour calculer une décomposition et d'autre part à l'intérêt de l'utilisation de cette dernière vis-à-vis de la résolution. Nous utilisons, par la suite, la notation G' au lieu de G'_σ lorsqu'il n'y a pas d'ambiguïté.

3.2.1 Effet boule de neige

Nous rappelons tout d'abord que la triangulation obtenue par les heuristiques de triangulation n'est généralement pas minimum, ni même minimale [Kjaerulff, 1990]. Autrement dit, il est possible de pouvoir supprimer des arêtes ajoutées à G' tout en gardant un graphe G' triangulé. L'ajout des arêtes additionnelles non nécessaires vis-à-vis de la triangulation entraîne forcément un surcoût au niveau du temps de triangulation en pratique. Plus important, cela peut entraîner une augmentation de la taille des clusters de la décomposition et de sa largeur w^+. En outre, ces heuristiques présentent un phénomène qui peut être qualifié de *l'effet boule de neige*. Il est caractérisé par un ajout considérable d'arêtes au graphe triangulé (potentiellement en $O(n'^3)$ pour une grille ou *grid graph* de $n' \times n'$ sommets, par exemple). Informellement, *l'effet boule de neige* consiste en une augmentation du nombre d'arêtes ajoutées au graphe entraînant à leur tour l'ajout d'un plus grand nombre d'arêtes. Souvent, cet aspect est dû à un ajout d'arêtes ne respectant pas ce qui serait souhaitable au regard de la topologie du graphe. En effet, la démarche suivie par ces heuristiques ignore la topologie du graphe. Par exemple, *Min-Fill* ne tient compte que du nombre d'arêtes nécessaires pour compléter le voisinage ultérieur d'un sommet. Au niveau de la topologie du graphe, nous nous intéressons notamment à l'indépendance entre des sous-graphes du graphe G, c'est-à-dire à l'absence d'arêtes entre deux sous-graphes de G. La plupart des heuristiques de triangulation ne prennent pas en compte, du moins explicitement, ce paramètre pendant la triangulation et peuvent potentiellement ajouter des arêtes entre deux sous-graphes indépendants, ce qui serait inutile du point de vue de la triangulation. Ainsi, la survenue de *l'effet boule de neige* est tout à fait possible avec ces heuristiques.

Nous illustrons ce phénomène par la figure 3.1 en utilisant *Min-Fill*. Toutefois, nous pouvons trouver des exemples similaires pour les autres heuristiques comme MCS [Tarjan and Yannakakis, 1984], *Lex* [Rose et al., 1976], *Minimum-Degree*[*Markowitz*, 1957]...

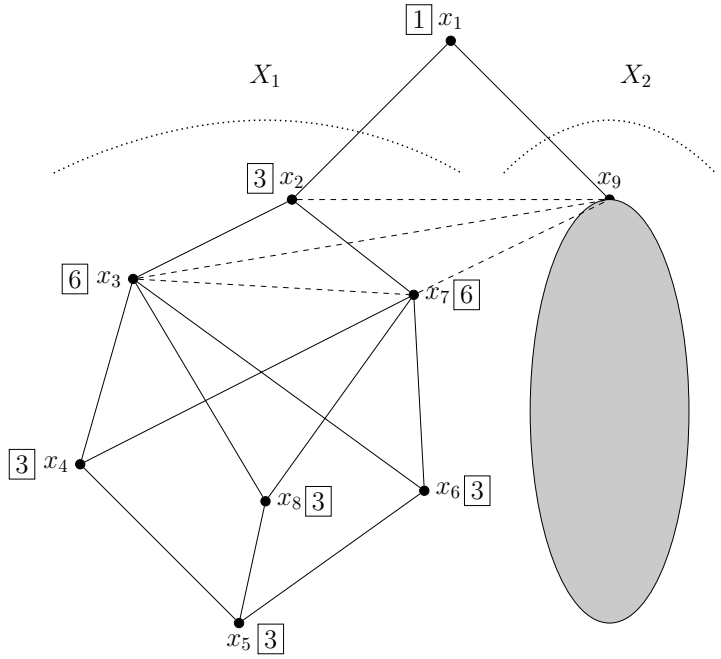


FIGURE 3.1 – Illustration de *l'effet boule de neige*.

Seuls 9 sommets du graphe sont représentés par la figure 3.1. Les arêtes pleines sont les arêtes du graphe original G et les arêtes en tirets sont les arêtes ajoutées par la triangulation. Chaque sommet est annoté d'un nombre encadré qui désigne le nombre initial (avant tout ajout d'arêtes) d'arêtes à ajouter entre les voisins de ce sommet afin de compléter le sous-graphe induit par ce voisinage. Par exemple, le sommet x_1 nécessite l'ajout d'une seule arête qui est $\{x_2, x_9\}$ et le sommet x_2 nécessite l'ajout de trois arêtes $\{x_1, x_3\}$, $\{x_1, x_7\}$ et $\{x_3, x_7\}$. Si nous supprimons x_1, ce graphe contient deux composantes connexes indépendantes X_1 et X_2 qui sont délimitées par les deux arcs en pointillés. Plus précisément, $X_1 = \{x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$ et X_2 est représentée par la partie grisée par souci de simplification. Ces deux composantes connexes sont dites indépendantes puisqu'il n'existe aucune arête de G liant un sommet de l'une à un sommet de l'autre. Elles sont cependant liées grâce au sommet x_1 qui est considéré comme étant leur séparateur. Autrement dit, X_1 et X_2 sont les deux composantes connexes induites par la suppression du sommet x_1 du graphe. Nous supposons que les sommets de X_2 nécessitent tous un nombre d'ajout d'arêtes supérieur ou égal à 3. Compte tenu de l'heuristique suivie par *Min-Fill* pour ordonner les sommets, *Min-Fill* choisit le sommet nécessitant le minimum d'ajout d'arêtes, c'est-à-dire x_1 dans ce cas. En choisissant le sommet x_1, nous ajoutons l'arête $\{x_2, x_9\}$. Une fois mis à jour, les sommets du graphe ont toujours besoin du même nombre d'arêtes à ajouter. Désormais, X_1 et X_2 forment une seule composante connexe même après avoir supprimé le sommet x_1. Ce phénomène est susceptible de se reproduire si nous choisissons, à l'étape suivante, le sommet x_2, ce qui conduira à rajouter les arêtes $\{x_3, x_7\}$, $\{x_3, x_9\}$ en plus de $\{x_7, x_9\}$. Toutes les arêtes ajoutées jusqu'à ce niveau ne sont pas nécessaires au regard de la triangulation parce qu'il ne s'agit pas d'arêtes reliant des sommets non-adjacents d'un cycle (vu l'indépendance de X_1 et X_2 une fois x_1 supprimée). Une solution à ce problème consiste à traiter indépendamment d'abord les sommets de X_1 ou de X_2. Pour généraliser, l'ajout d'une arête inutile du point de vue de la triangulation entre deux composantes indépendantes pourrait engendrer de plus en plus d'ajouts non nécessaires. En effet, la rétroaction permettrait de renforcer *l'effet boule de neige*.

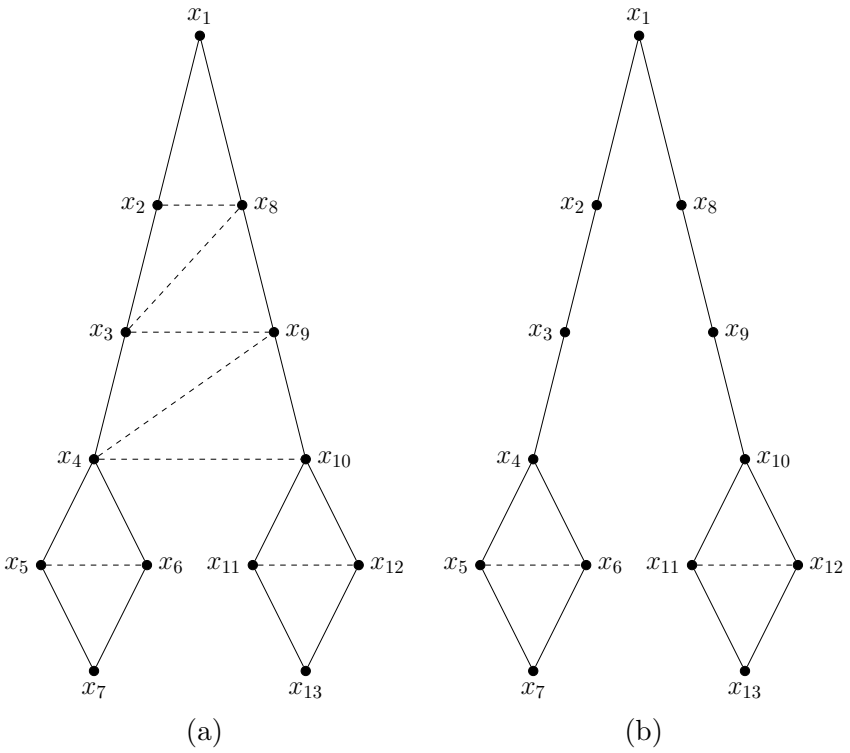


FIGURE 3.2 – Triangulation ne respectant pas la topologie (a) et une triangulation respectant la topologie (b).

Une solution possible à ce problème consiste à trianguler chaque composante connexe indépendamment des autres.

La figure 3.2 montre deux triangulations possibles d'un graphe. Dans la figure 3.2, les deux composantes connexes induites par la suppression de x_1 sont $X_1 = \{x_2, x_3, x_4, x_5, x_6, x_7\}$ et $X_2 = \{x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}\}$. Cette figure montre que la triangulation qui respecte la topologie du graphe ajoute moins d'arêtes que la version qui n'en tient pas compte. En effet, *Min-Fill* établit dans la figure 3.2(a) un ordre alternant des sommets de X_1 et de X_2, c'est-à-dire un ordre $O = [x_1, x_2, x_8, x_3, x_9, x_7, x_5, x_6, x_4, x_{10}, x_{11}, x_{12}, x_{13}]$. Cependant dans la figure 3.2(b), *Min-Fill* réussit à ajouter moins d'arêtes en traitant tout d'abord tous les sommets de X_1 avant ceux de X_2 avec un ordre $O = [x_7, x_5, x_6, x_4, x_3, x_2, x_1, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}]$.

Les inconvénients de *l'effet boule de neige* ne se limitent pas à l'ajout excessif d'arêtes qui augmente le temps de calcul de la triangulation et plus généralement le temps d'obtention de la décomposition arborescente. Il peut y avoir aussi un impact sur la largeur arborescente w^+ de la décomposition calculée. En effet, l'ajout excessif d'arêtes augmente potentiellement la taille des cliques du graphe en cours de triangulation, et par voie de conséquence, la taille des clusters de la décomposition, donc sa largeur. En conclusion, malgré son intérêt au niveau de la minimisation du w^+, la triangulation menée par des heuristiques comme *Min-Fill* peut augmenter considérablement le temps de décomposition mais aussi sa largeur w^+.

3.2.2 Efficacité des décompositions vis-à-vis de la résolution

Grâce aux heuristiques de calcul de décompositions, les décompositions arborescentes ont déjà été exploitées avec succès pour résoudre des instances (W)CSP et pour le comp-

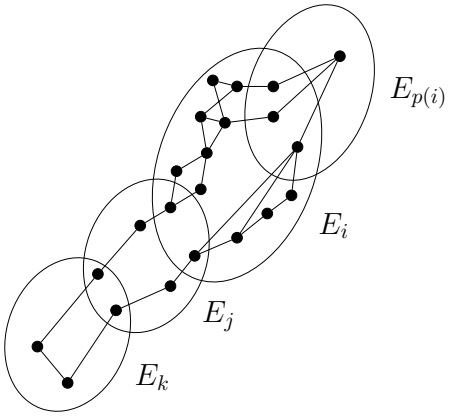


FIGURE 3.3 – Une décomposition avec le cluster E_i non connexe.

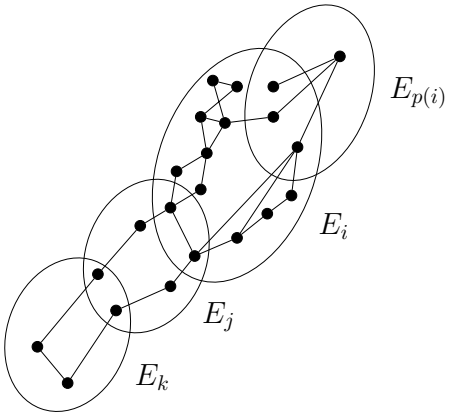


FIGURE 3.4 – Une décomposition avec le cluster E_i non connexe.

tage [Jégou and Terrioux, 2003, 2004b; Givry et al., 2006; Favier et al., 2009; Sanchez et al., 2009; Allouche et al., 2015]. Cependant, les décompositions existantes ne sont pas nécessairement les plus adaptées du point de vue de la résolution [Jégou et al., 2005; Jégou and Terrioux, 2014b]. D'ailleurs, le temps de calcul de décomposition peut effectivement largement dépasser le temps de résolution des méthodes non basées sur une décomposition dans certains cas. En plus, même si la largeur w^+ des décompositions calculées demeure très intéressante par rapport à la largeur optimale w, il ne semble pas que ce critère soit le plus pertinent au regard de l'efficacité de la résolution. En effet, les décompositions existantes, qu'elles soient ou non basées sur la triangulation, présentent les problèmes listés ci-dessous.

Limitation de la liberté de l'heuristique de choix de variables Afin de garantir une complexité en temps en $O(exp(w^+))$, les méthodes structurelles efficaces comme BTD utilisent un ordre d'affectation des variables qui est partiellement induit par la décomposition considérée. Quand des heuristiques comme $Min\text{-}Fill$ sont utilisées, cette liberté est même encore plus restreinte du fait du nombre limité de sommets propres dans les clusters. Le nombre de sommets propres peut d'ailleurs atteindre un seul sommet pour certains clusters. Dans ce cas, la liberté du choix de la prochaine variable se limite au choix du prochain cluster fils. Si, en plus, un cluster n'a qu'un seul cluster fils, cela implique une résolution exploitant un ordre de choix de variables total, donc statique. Or, il est bien

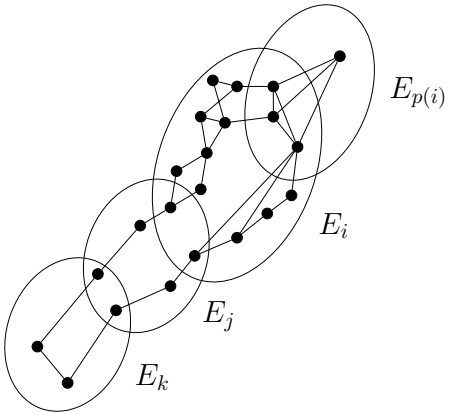


FIGURE 3.5 – Une décomposition avec le cluster E_i connexe.

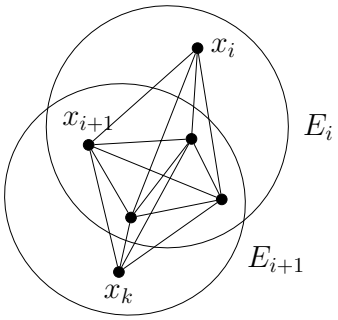


FIGURE 3.6 – Deux clusters d'une décomposition (cliques issues d'une triangulation) ayant $s = w^+ = |E_i| - 1$.

connu que pour avoir une résolution efficace, il est souhaitable d'avoir une liberté maximale pour le choix des prochaines variables à affecter et le plus de dynamicité possible à ce niveau.

Non-connexité des clusters La décomposition calculée peut éventuellement contenir des clusters non connexes (c'est-à-dire des clusters E_i tels que $G[E_i]$ possède plusieurs composantes connexes), ce qui peut conduire à une forte dégradation de l'efficacité de la résolution [Jégou and Terrioux, 2014b]. Les figures 3.3, 3.4 et 3.5 montrent trois décompositions formées de 4 clusters, E_i ayant comme père le cluster $E_{p(i)}$ et un fils noté E_j. Le cluster E_k est à son tour le fils de E_j. Nous nous focalisons sur le cluster E_i et nous distinguons trois cas possibles :

- Cas de la figure 3.3 : E_i est non connexe et le graphe $G[E_i \backslash (E_i \cap E_{p(i)})]$ contient deux composantes connexes indépendantes. Tout d'abord, la présence de plusieurs composantes connexes dans un cluster E_i peut diminuer l'efficacité de la résolution localement. En effet, si l'une des composantes connexes est insatisfaisable et que ce n'est pas le cas des autres composantes connexes, la recherche peut potentiellement explorer toutes les solutions des autres composantes connexes avant de prouver l'insatisfaisabilité de celle-ci. Dans le cadre du problème CSP ou #CSP, en utilisant BTD, les variables de E_i distribuées dans plusieurs composantes connexes doivent être assignées d'une façon cohérente. Si le sous-problème enraciné en E_i (contenant

les variables de E_i, E_j et E_k) est facile, c'est-à-dire possède beaucoup de solutions, l'affectation du cluster E_i est *a priori* rapide et s'étend plus probablement à une solution du sous-problème en question. Dans ce cas, la non-connexité du cluster E_i ne pose pas un véritable problème. Si au contraire le sous-problème n'admet que très peu de solutions, voire aucune, la résolution assigne les variables de E_i plus difficilement en passant potentiellement d'une composante connexe à l'autre. En effet, si des techniques de filtrage sont utilisées, l'affectation d'une variable x_i d'une composante connexe n'a que très peu d'influence sur les domaines des variables des autres composantes connexes. Donc, ces variables seront assignées sans vraiment tenir compte de l'affectation de x_i. C'est ainsi qu'une incohérence pourrait être détectée plus tard dans la recherche lors du traitement de l'un des fils de E_i. Cela diminue l'efficacité de la résolution en gaspillant du temps, mais aussi, en consommant de l'espace, en augmentant essentiellement le nombre de nogoods enregistrés. De même, dans le cadre du problème WCSP, la non-connexité peut avoir un impact sur la qualité de l'affectation de E_i (en termes de coût) et peut ainsi demander plusieurs retours à E_i avant de pouvoir trouver la solution optimale.

- Cas de la figure 3.4 : Le fait que le cluster E_i soit non connexe et que $G[E_i \backslash (E_i \cap E_{p(i)})]$ soit connexe signifie que le cluster E_i est non connexe à cause de la non-connexité du séparateur. Vu que les variables du séparateur sont assignées avant l'affectation des variables propres de E_i, la non-connexité n'induit alors aucun problème dans ce cas au niveau de la résolution.

- Cas de la figure 3.5 : Le fait que le cluster E_i soit connexe et que $G[E_i \backslash (E_i \cap E_{p(i)})]$ ne le soit pas nous refait revenir au cas de la figure 3.3. En effet, comme les variables du séparateur sont instanciées avant celles des variables propres de E_i cela produit un cluster E_i non connexe lors de l'instanciation de ses variables propres.

En guise de conclusion, un cluster E_i doit avoir le sous-graphe $G[E_i \backslash (E_i \cap E_{p(i)})]$ connexe. Notons que le choix du cluster racine indique l'orientation de la décomposition et ainsi les liens de parenté et filiation qui existent entre les clusters. Donc, la présence non souhaitable des cas des figures 3.3 et 3.5 dépend du choix de ce cluster. Or, il n'y a pas de garantie de pouvoir trouver un cluster racine convenable, c'est-à-dire qui garantit l'absence des cas des figures 3.3 et 3.5.

Séparateurs de grande taille Les décompositions calculées par les heuristiques de triangulation comme *Min-Fill* peuvent avoir des séparateurs de très grande taille. Souvent, la taille du plus grand séparateur s est très proche de la largeur w^+, voire même égale à w^+ parfois. La figure 3.6 montre deux clusters E_i et E_{i+1} d'une décomposition ayant la taille du plus grand séparateur $s = w^+$. Considérons que le sommet x_i est le premier sommet de E_i au regard de l'ordre d'élimination établi par *Min-Fill*. Notons x_{i+1} le premier voisin de x_i par rapport à cet ordre. Supposons que x_{i+1} est voisin du sommet x_k qui n'est pas voisin de x_i. Pendant la triangulation le voisinage de x_i est complété et formera avec x_i le cluster E_i. Lorsque x_{i+1} est traité, son voisinage est complété. En particulier, x_k est lié à tous les sommets de E_i sauf x_i (éliminé à ce stade). Un nouveau cluster E_{i+1} est alors formé partageant avec E_i tous les sommets sauf un. Ainsi, la taille du plus grand séparateur s est égal à la taille de $E_i - 1$ soit à w^+. L'augmentation de s peut mener à un coût qui s'avère prohibitif en termes d'espace mémoire. En effet, les méthodes de résolution basées sur une décomposition ont une complexité spatiale en $O(exp(s))$. En plus, l'augmentation de la taille des séparateurs peut avoir un impact sur le taux d'utilisation des informations enregistrées. En effet, l'augmentation de la taille du séparateur augmente le nombre de ses

affectations possibles ce qui rend moins probable l'utilisation des informations enregistrées pour une affectation donnée. Cela peut être problématique pour l'efficacité de la résolution et peut la détériorer significativement. D'ailleurs, la limitation de la taille des séparateurs a été démontrée cruciale pour l'efficacité en pratique dans [Jégou et al., 2005].

Contraintes à portée étirée sur plusieurs clusters En ce qui concerne la résolution, *BTD* essaye d'exploiter les contraintes *dès que possible*. Nous distinguons deux cas possibles :

- Pour une contrainte binaire, dès qu'une variable est assignée et l'algorithme de filtrage appliqué, les valeurs restantes dans le domaine de l'autre variable sont garanties d'être cohérentes puisque le filtrage répercute les conséquences de l'affectation de la variable sur la deuxième variable.
- Dans le cas de contraintes non binaires, certaines peuvent se retrouver partagées dans plusieurs clusters. En d'autres termes, les variables sur lesquelles porte cette contrainte peuvent se retrouver distribuées dans plusieurs clusters. Ceci est dû à l'emploi de la 2-section de l'hypergraphe de contraintes qui transforme toute contrainte d'arité b en une b-clique. Par construction, nous savons qu'il existe forcément un cluster qui recouvre entièrement cette b-clique. Par contre, rien n'empêche que certaines arêtes de cette b-clique soient présentes dans d'autres clusters. Selon le choix du cluster racine, la résolution pourrait exploiter en premier un cluster E_i contenant au plus $b-2$ variables d'une contrainte d'arité b. Dans ce cas, cette contrainte pourrait n'être que très partiellement exploitée lors des propagations. De plus, l'exploiter pleinement pourrait requérir d'instancier une partie des variables manquantes, ces dernières se trouvant dans un ou plusieurs autres clusters plus profonds par rapport à la décomposition. Soit E_k le cluster contenant l'avant-dernière variable de la portée, par exemple. Cela signifie que toutes les variables présentes dans les clusters situés sur le chemin entre E_i et E_k seront instanciées au préalable. Il apparaît clairement que ceci peut être fortement préjudiciable en termes d'efficacité, en particulier, si la recherche n'aboutit pas à cause des affectations des premières variables de cette contrainte.

3.3 Nouveau cadre de calcul de décompositions : H-TD

Le but de ce chapitre est de proposer une méthode de calcul de décompositions qui évite d'une part l'effet boule de neige, soit en empêchant l'utilisation de la triangulation, soit en triangulant d'une manière mieux adaptée à la structure. D'autre part, elle permet d'améliorer l'efficacité de la décomposition vis-à-vis de la résolution en tenant compte des différents critères mentionnés ci-dessus. Nous proposons donc un nouveau cadre de calcul de décompositions, appelé *H-TD* (pour *Heuristic Tree-Decomposition*), qui calcule une décomposition arborescente d'un graphe $G = (X, C)$. Il est à préciser que l'algorithme *H-TD* construit l'ensemble des clusters E de la décomposition sans calculer explicitement l'arbre T. D'ailleurs, l'arbre T peut être calculé en post-traitement grâce à l'algorithme de Jarnìk ou mieux encore en liant au fur et à mesure de la création des clusters le nœud correspondant à un cluster à celui correspondant à son cluster père. Comme les autres heuristiques de calcul de décompositions, aucune garantie n'est offerte quant à l'optimalité de la largeur obtenue. *H-TD* vise à :

- Permettre de calculer des décompositions sans forcément se baser sur la triangulation,

Algorithme 3.2 : H-TD (G)

Entrées : Un graphe $G = (X, C)$
Sorties : Un ensemble de clusters $E_0, \ldots, E_l$ d'une décomposition arborescente de G

1 $X_0 \leftarrow X$
2 Choix d'un premier cluster E_0 dans G
3 $X' \leftarrow E_0$
4 Soient $X_{0_1}, \ldots, X_{0_{k_0}}$ les composantes connexes de $G[X_0 \backslash E_0]$
5 $F \leftarrow \{X_{0_1}, \ldots, X_{0_{k_0}}\}$
6 **tant que** $F \neq \emptyset$ **faire** `/* calcul d'un nouveau cluster` E_i `*/`
7 Enlever X_i la composante connexe courante de F
8 Soit $V_i \subseteq X'$ le voisinage de X_i dans G
9 Déterminer un sous-ensemble $X''_i \subseteq X_i$ tel que $X''_i \cup V_i$ constitue un cluster définitif de la décomposition
10 $E_i \leftarrow X''_i \cup V_i$
11 $X' \leftarrow X' \cup X''_i$
12 Soient $X_{i_1}, X_{i_2}, \ldots, X_{i_{k_i}}$ les composantes connexes de $G[X_i \backslash E_i]$
13 $F \leftarrow F \cup \{X_{i_1}, X_{i_2}, \ldots X_{k_i}\}$

- Améliorer le temps de décomposition en pratique et la complexité théorique temporelle,

- Éviter les inconvénients de la triangulation en exploitant les propriétés topologiques du graphe,

- Permettre la paramétrisation de la décomposition dans le but de pouvoir prendre en compte plusieurs critères comme la largeur de la décomposition w^+ ou s la taille maximale des séparateurs.

H-TD s'inspire de l'algorithme *BC-TD* (pour *Bag Connected Tree-Decomposition*) [Jégou and Terrioux, 2014b] qui vise à calculer des décompositions sans triangulation formées de clusters connexes. D'ailleurs, *BC-TD* peut être considéré comme une des heuristiques de *H-TD* ou une paramétrisation possible de ce cadre avec en plus l'avantage d'empêcher de calculer des clusters inclus dans d'autres ce qui constitue un gain en temps non négligeable.

3.3.1 Schéma général

Le calcul des décompositions arborescentes par *H-TD* s'effectue en général grâce à un parcours du graphe G en se basant sur les propriétés liées aux séparateurs et à leurs composantes connexes associées. Ce calcul est constitué de deux éléments principaux, le calcul du premier cluster E_0 et le calcul du cluster suivant E_i à partir d'une composante connexe X_i et de son voisinage noté V_i. *H-TD* est décrit dans l'algorithme 3.2. Il prend en entrée un graphe $G = (X, C)$. Notons que nous partons de l'hypothèse que G est un graphe connexe. Si ce n'est pas le cas, il est traité comme plusieurs graphes connexes, un par composante connexe de G. L'ensemble de sommets X de G est considéré comme étant la composante connexe initiale X_0 (ligne 1 de l'algorithme 3.2). *H-TD* retourne l'ensemble de clusters $E = \{E_0, \ldots, E_l\}$ de la décomposition obtenue.

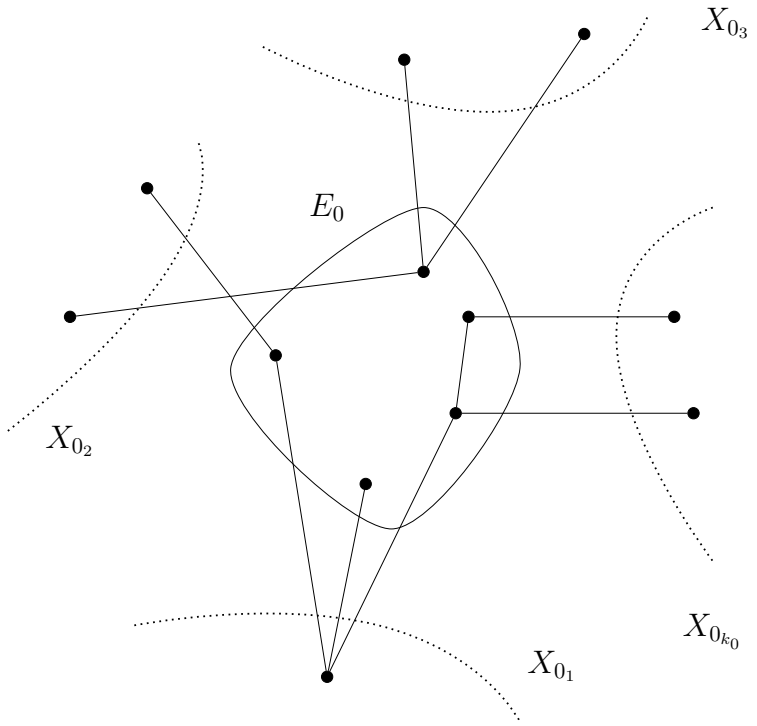


FIGURE 3.7 – Calcul du premier cluster E_0.

Calcul du premier cluster E_0 et traitement postérieur La figure 3.7 illustre le calcul du premier cluster E_0. Elle montre uniquement les arêtes liant des sommets de E_0 à une composante induite par la suppression de ce dernier de G. Le calcul du premier cluster E_0 se fait grâce à des techniques heuristiques (ligne 2 de l'algorithme 3.2). Ce premier cluster peut consister en une clique de grande ou de petite taille mais peut aussi être un séparateur de taille bornée. Le choix de E_0 dépend notamment des critères souhaités à l'égard de la décomposition calculée. Ainsi, si les clusters de la décomposition obtenue doivent être connexes, alors il est impératif que E_0 soit connexe. Notons aussi que l'heuristique de choix du premier cluster doit avoir un coût raisonnable afin de ne pas augmenter le temps de calcul de *H-TD*. X' représente l'ensemble des sommets déjà considérés de G et est initialisé à E_0 (ligne 3). Chaque sommet de X' est alors présent dans au moins un cluster parmi les clusters déjà formés de la décomposition. $X_{0_1}, X_{0_2}, \ldots, X_{0_{k_0}}$ représentent les composantes connexes relatives à $G[X_0 \backslash E_0]$ (ligne 4), induites par la suppression dans G des sommets de E_0. Chacune de ces composantes connexes est insérée dans une file d'attente F (ligne 5).

Calcul du cluster suivant E_i et traitement postérieur Le deuxième élément est le calcul du cluster suivant E_i. Il est calculé à partir de la composante connexe courante X_i extraite de la file d'attente F. Pour chaque élément X_i supprimé de F (ligne 7), V_i représente l'ensemble de sommets de X' adjacents à au moins un sommet de X_i (ligne 8). Nous pouvons constater que V_i est un séparateur du graphe G vu que la suppression des sommets de V_i de G déconnecte G (X_i étant déconnecté du reste de G). V_i constituera ainsi l'intersection entre le cluster E_i à calculer et son cluster parent $E_{p(i)}$. D'ailleurs, la construction de l'algorithme garantit qu'il y a un cluster $E_{p(i)}$ contenant l'ensemble V_i. Nous considérons maintenant le sous-graphe de G induit par V_i et X_i, c'est-à-dire $G[V_i \cup X_i]$. L'étape suivante (ligne 9) peut être paramétrée. Elle recherche un sous-ensemble

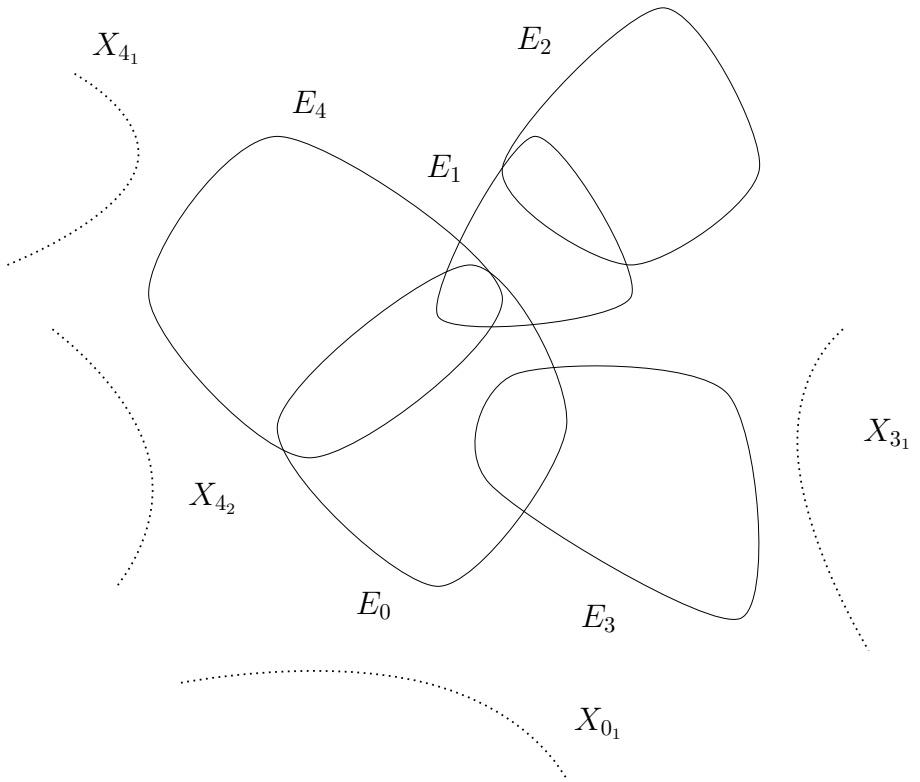


FIGURE 3.8 – La fin du calcul du cluster E_4.

de sommets $X''_i \subseteq X_i$ tel que $X''_i \cup V_i$ sera un nouveau cluster E_i de la décomposition. Pour que $X''_i \cup V_i$ soit un cluster de la décomposition, il faut garantir que l'ensemble des sommets de $X''_i \cup V_i$ ne sera pas inclus dans de futurs clusters de la décomposition. Certes, le fait d'avoir un cluster inclus dans un autre n'a pas d'impact sur la validité de la décomposition. Cependant, cette propriété permet d'empêcher de calculer un cluster contenant un cluster déjà calculé et ayant une taille plus grande. Ainsi, ceci permet d'économiser le temps de calcul des clusters inclus dans d'autres clusters ainsi que le temps de leur suppression notamment que leur présence peut éventuellement être contre-productive vis-à-vis de la résolution. Cela peut être garanti s'il existe au moins un sommet v de V_i tel que tous ses voisins dans X_i figurent dans X''_i. Plus précisément, si $N(v, X_i) = \{x \in X_i : \{v, x\} \in C\}$, nous devons garantir l'existence d'un sommet v de V_i avec $N(v, X_i) \subseteq X''_i$. En d'autres termes, nous garantissons par cette condition que le cluster E_i n'est pas inclus dans d'autres clusters de la décomposition comme c'est le cas de la figure 3.9. En effet, si $V_{i-1} = \{x_1\}$, la construction du cluster E_{i-1} ne respecte pas la condition vu que E_{i-1} ne contient pas tous les voisins de x_1, soit x_2, x_3 et x_4. Ainsi, à l'étape suivante, $X_i = \{x_4\}$ et $V_i = \{x_1, x_2, x_3\}$ d'où $E_i = \{x_1, x_2, x_3, x_4\}$. De ce fait, $E_{i-1} \subset E_i$. Notons que cette condition est suffisante mais pas nécessaire. D'ailleurs, il se peut qu'il existe $X'''_i \subset X''_i$ tel que le cluster $E_i = X'''_i \cup V_i$ obtenu est garanti non inclus dans de futurs clusters. En outre, d'autres conditions peuvent garantir que le cluster E_i ne sera pas inclus dans de futurs clusters. C'est le cas de l'heuristique *Min-Fill-MG* que nous allons décrire plus tard. Nous définissons alors un nouveau cluster $E_i = X''_i \cup V_i$ (ligne 10). Puis, nous rajoutons à X' les sommets de X''_i (ligne 11) avant de calculer les composantes connexes du sous-graphe issu de la suppression des sommets de E_i dans $G[X_i]$, notées $X_{i_1}, X_{i_2}, \ldots X_{i_{k_i}}$ (ligne 12). Le calcul de ces composantes connexes permet la prise en compte de la topologie du graphe et empêche la création de clusters contenant des sommets de différentes composantes

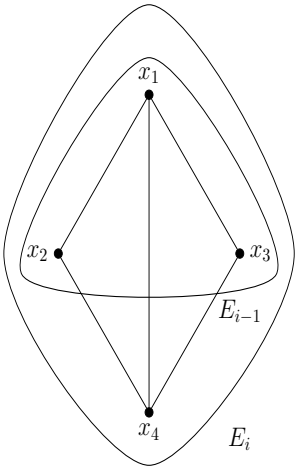


FIGURE 3.9 – Deux clusters d'une décomposition ayant $E_{i-1} \subset E_i$.

connexes. Chacune de ces composantes connexes est insérée à son tour dans F pour être traitée ultérieurement (ligne 13). Ce processus est répété jusqu'à ce que la file F soit vide. La figure 3.8 montre la progression du calcul des clusters. Après le choix du cluster E_0, les clusters E_1, E_2, E_3 et E_4 ont été calculés. Le prochain cluster à calculer est le cluster E_5. *H-TD* fera alors un choix entre les composantes connexes disponibles dans F, à savoir X_{0_1}, X_{3_1}, X_{4_1} ou X_{4_2}.

La figure 3.10 récapitule finalement les étapes principales de *H-TD*.

Selon les heuristiques choisies aux lignes 2 et 9, *H-TD* peut posséder la propriété suivante :

Propriété 3 *Soit $E_0, \dots E_l$ l'ensemble de clusters de la décomposition calculée par H-TD selon l'heuristique employée. $\forall E_i, \nexists E_j$ tel que $i \neq j$ et $E_i \subseteq E_j$.*

En effet, les heuristiques choisies à la ligne 2 et 9 doivent garantir cette propriété. Si l'heuristique du calcul du cluster E_0 ne tient pas compte de cette propriété, le cluster E_0 calculé peut être éventuellement inclus dans d'autres clusters. Cependant, comme l'heuristique de la ligne 9 garantit que le cluster calculé ne sera pas inclus dans de futurs clusters, seulement le cluster E_0 serait inclus dans d'autres clusters et peut éventuellement être supprimé.

H-TD possède la propriété suivante :

Propriété 4 *Pour tout séparateur V_i de la décomposition calculée par H-TD, G admet au moins une composante connexe pleine associée à V_i.*

En effet, chaque sommet v de V_i est lié à au moins un sommet de X_i (par construction). X_i est alors une composante connexe pleine associée à V_i. Si G admet en plus une deuxième composante connexe pleine, alors V_i est minimal.

3.3.2 Heuristiques proposées non basées sur une triangulation

Nous déclinons tout d'abord ce schéma selon cinq heuristiques non basées sur la triangulation : H_1-*TD*, H_2-*TD*, H_3-*TD*, H_4-*TD* et H_5-*TD* [Jégou et al., 2015a, 2016b]. Pour chacune de ces heuristiques le sous-ensemble X_i'' choisi à la ligne 9 est tel qu'il existe au moins un sommet $v \in V_i$ avec $N(v, X_i) \subseteq X_i''$. Cette condition garantit que le cluster E_i nouvellement construit ne sera pas inclus dans de futurs clusters (comme illustré dans la figure 3.9). L'objectif des différentes heuristiques est de :

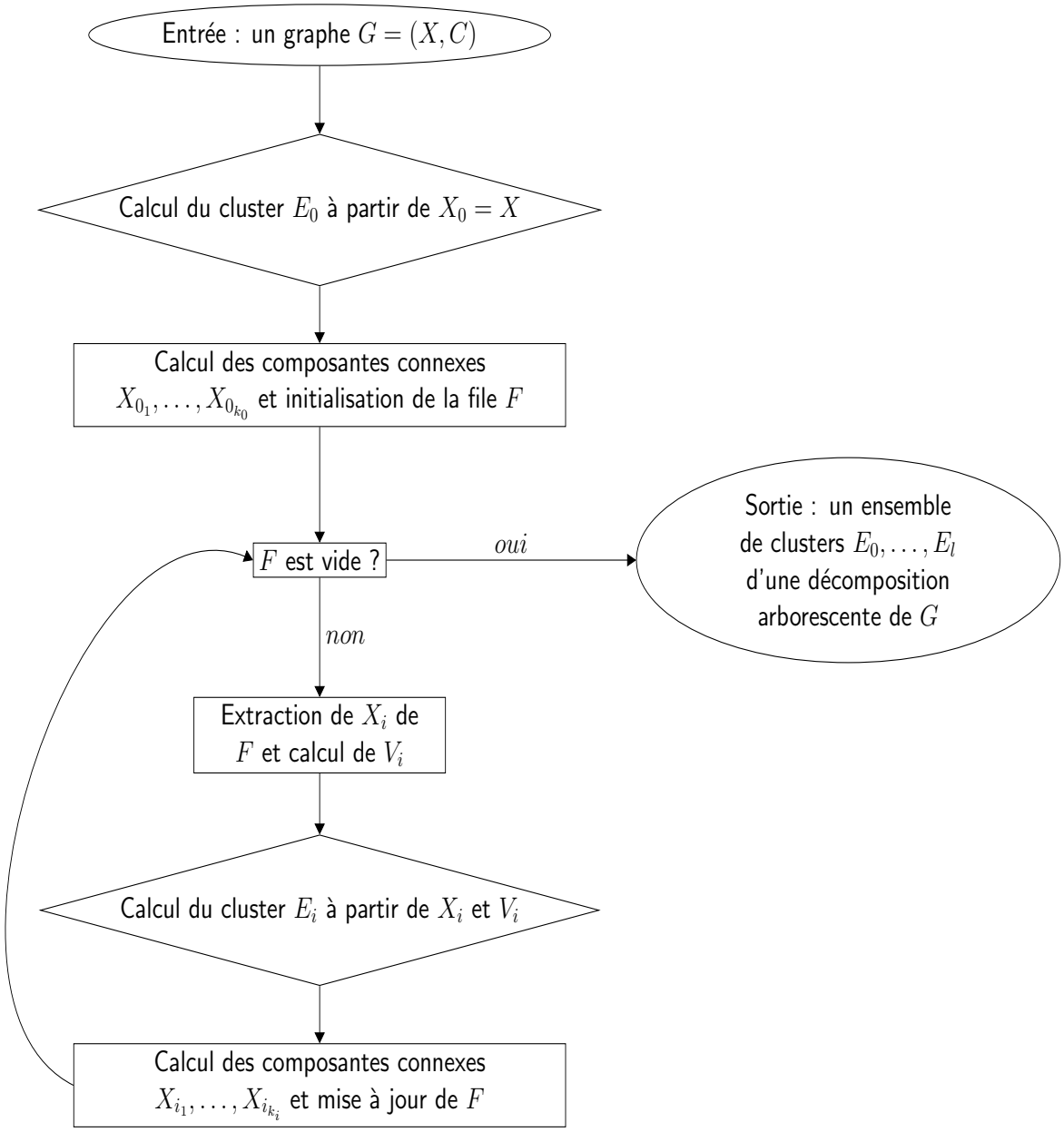


FIGURE 3.10 – Le schéma général de H-TD.

- Minimiser la largeur de la décomposition : H_1-TD
- Calculer des décompositions plus adaptées à la résolution d'instances (W)CSP
  - H_2-TD : calculer une décomposition avec des clusters connexes,
  - H_3-TD : calculer une décomposition telle que les clusters ont plusieurs fils,
  - H_4-TD : limiter la taille des séparateurs,
  - H_5-TD : limiter la taille des séparateurs tout en essayant d'augmenter leur nombre.

Par la suite, nous illustrons la façon dont chaque heuristique calcule le prochain cluster E_i grâce au graphe de la figure 3.11. Le premier cluster choisi E_0 est l'ensemble $\{x_1, x_2, x_3, x_4\}$.

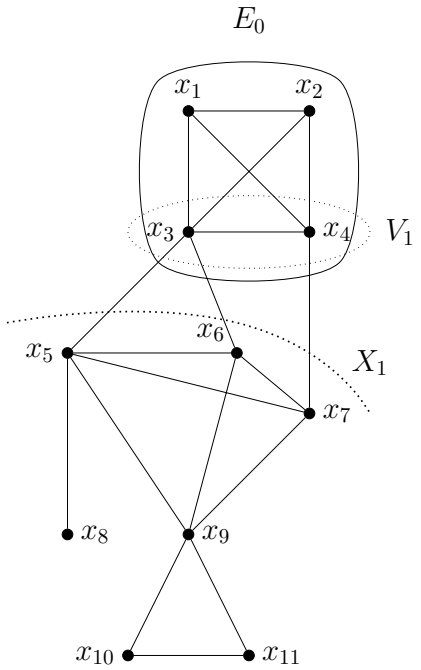


FIGURE 3.11 – Un graphe à décomposer par H-TD.

La composante connexe courante induite par la suppression des sommets de E_0 de G est $X_{0_1} = X_1 = \{x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}\}$ (représentée par un arc en pointillés). Son voisinage correspondant est $V_1 = \{x_3, x_4\}$ (représenté par une ellipse en pointillés). Nous expliquons alors comment est calculé le prochain cluster E_1 à partir de V_1 et de X_1 selon l'heuristique considérée.

- H_1-TD : Comme indiqué ci-dessus, cette heuristique vise à minimiser localement la taille du prochain cluster E_i. Cela permet essentiellement de minimiser la largeur de la décomposition obtenue, c'est-à-dire le paramètre central en termes de complexité théorique. C'est pourquoi nous recherchons le plus petit sous-ensemble X_i'' pour qu'il forme avec V_i le prochain cluster E_i. Cela est possible en cherchant un sommet v de V_i ayant le minimum de voisins dans X_i. Ce sommet v choisi, $X_i'' = N(v, X_i)$ et ainsi $E_i = X_i'' \cup V_i$.

  Dans la figure 3.11, le séparateur V_1 contient les deux sommets x_3 et x_4. Dans X_1, x_3 a deux voisins (x_5 et x_6) et x_4 en a un seul (x_7). Ainsi, le sommet v de V_1 qui a le moins de voisins dans X_1 est x_4. Nous construisons alors le cluster $E_1 = \{x_3, x_4, x_7\}$. Nous calculons ensuite la composante connexe $X_{1_1} = \{x_5, x_6, x_8, x_9, x_{10}, x_{11}\}$ et nous ajoutons X_{1_1} à F.

- H_2-TD : Cette heuristique calcule le prochain cluster E_i qui doit être connexe vu l'importance montrée de ce paramètre au regard de la résolution. Elle se base sur la notion de *connected tree-width* récemment introduite en théorie des graphes dans [Müller, 2012; Diestel and Müller, 2017; Hamann and Weißauer, 2016]. Elle s'inspire de l'algorithme BC-TD présenté en détails dans [Jégou and Terrioux, 2014b] tout en empêchant le calcul de nouveaux clusters qui incluent des clusters déjà calculés. Notons que garantir la connexité de E_i lors de sa résolution nécessite que $G[E_i \backslash (E_i \cap E_{p(i)})]$ soit connexe. Or, si les redémarrages pendant la résolution sont exploitées, ils peuvent éventuellement s'accompagner d'un changement du cluster racine. Ainsi,

$E_{p(i)}$ est susceptible de changer. Garantir la connexité de $G[E_i \backslash (E_i \cap E_{p(i)})]$ quel que soit le cluster $E_{p(i)}$ peut augmenter considérablement la taille des clusters. C'est pourquoi nous nous contentons de la connexité du cluster E_i.

Nous montrons le calcul du prochain cluster E_1 du graphe représenté dans 3.11. Plusieurs possibilités sont envisageables. E_1 peut par exemple être $E_1 = \{x_3, x_4, x_5, x_6\}$ qui induit un sous-graphe connexe avec $N(v, X_1) = N(x_3, X_1) = \{x_5, x_6\} = X_i'' = \{x_5, x_6\}$. Les deux composantes connexes induites sont ainsi : $X_{1_1} = \{x_8\}$ et $X_{1_2} = \{x_7, x_9, x_{10}, x_{11}\}$.

- H_3-TD : Cette heuristique construit le prochain cluster E_i grâce aux propriétés topologiques du graphe. Elle vise à identifier plusieurs composantes indépendantes du graphe et à les séparer. Pour y parvenir, H_3-TD ajoute des sommets au cluster suivant E_i grâce à un parcours en largeur du graphe en commençant par les sommets de V_i. C'est ainsi que les voisins de V_i dans X_i constituent le premier niveau de X_i, les voisins des voisins de V_i dans X_i constituent le deuxième niveau de X_i et ainsi de suite. Alors, au niveau $u = 1$, $E_{i_1} = N[V_i, X_i]$ et au niveau u ($u > 1$), $E_{i_u} = N[E_{i_{u-1}}, X_i]$. Pour calculer E_i, cette heuristique continue à avancer à travers les niveaux et à ajouter des sommets jusqu'à ce que X_i soit séparée en plusieurs composantes connexes. En d'autres termes, H_3-TD continue à avancer jusqu'à un niveau $u = U$ tel que $G[X_i \backslash E_{i_U}]$ contient strictement plus qu'une composante connexe ou bien jusqu'à ce que $G[X_i \backslash E_{i_U}]$ soit vide.

  Si nous considérons l'exemple de la figure 3.11, $E_{1_1} = \{x_3, x_4, x_5, x_6, x_7\}$, $E_{1_2} = \{x_3, x_4, x_5, x_6, x_7, x_8, x_9\}$ et $E_{1_3} = \{x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}\}$ (dans cet exemple, il y a 3 niveaux au maximum). Dans cet exemple, le parcours en largeur est stoppé au niveau $U = 1$ parce que $G[X_1 \backslash E_{1_1}]$ contient deux composantes connexes $X_{1_1} = \{x_8\}$ et $X_{1_2} = \{x_9, x_{10}, x_{11}\}$ qui seront ajoutées à F. D'où, $E_1 = \{x_3, x_4, x_5, x_6, x_7\}$.

- H_4-TD : Cette heuristique vise à limiter la taille des séparateurs de la décomposition qui est un paramètre crucial à prendre en compte pour permettre une résolution plus efficace. Pour y parvenir, elle considère le paramètre S qui représente une borne supérieure sur la taille maximale permise pour un séparateur de la décomposition ($s \leq S$). Cette heuristique ajoute de nouveaux sommets à E_i de la même manière que H_3-TD. Néanmoins, H_4-TD s'arrête au niveau $u = U$ tel que $G[X_i \backslash E_{i_U}]$ ne contient pas des composantes connexes ayant des séparateurs de taille supérieure à S.

  Avec l'exemple de la figure 3.11, supposons que $S = 2$. $E_{1_1} = \{x_3, x_4, x_5, x_6, x_7\}$ et induit deux composantes connexes $X_{1_1} = \{x_8\}$ ayant un séparateur de taille 1 (contenant uniquement le sommet x_5) et $X_{1_2} = \{x_9, x_{10}, x_{11}\}$ ayant un séparateur de taille 3 (contenant x_5, x_6 et x_7). Ainsi, nous ne pouvons pas nous arrêter au niveau $u = 1$ puisque tous les séparateurs n'ont pas une taille au maximum égale à 2. Cependant, $E_{1_2} = \{x_3, x_4, x_5, x_6, x_7, x_8, x_9\}$ induit une seule composante connexe $X_{1_1} = \{x_{10}, x_{11}\}$ ayant un séparateur contenant un seul sommet x_9. Ainsi, le parcours peut s'arrêter au niveau $U = 2$ et ainsi $E_1 = \{x_3, x_4, x_5, x_6, x_7, x_8, x_9\}$.

- H_5-TD : Comme H_4-TD, cette heuristique vise aussi à limiter la taille des séparateurs de la décomposition tout en raffinant cette décomposition. En effet, elle permet de détecter plus de séparateurs de taille au plus S. Là où H_4-TD doit atteindre un niveau de la recherche en largeur auquel tous les séparateurs sont de taille au plus S, H_5-TD va être plus opportuniste. Si à un niveau donné, une nouvelle composante connexe apparaît avec un séparateur de taille au plus S, ce séparateur va être pris

en compte. Sa composante connexe sera rajoutée à la file d'attente, et la recherche va se poursuivre sur le reste du sous-graphe en cours d'exploitation, exceptée bien sûr, la partie associée à cette nouvelle composante connexe.

Nous illustrons le calcul du prochain cluster E_1 par H_5-TD sur l'exemple de la figure 3.11. Nous supposons aussi que $S = 2$. $E_{1_1} = \{x_3, x_4, x_5, x_6, x_7\}$ et induit deux composantes connexes $X_{1_1} = \{x_8\}$ ayant un séparateur de taille 1 (contenant uniquement le sommet x_5) et $X_{1_2} = \{x_9, x_{10}, x_{11}\}$ ayant un séparateur de taille 3 (contenant x_5, x_6 et x_7). X_{1_1} induit alors un séparateur d'une taille inférieure à 2. La recherche va certes se poursuivre mais l'existence du séparateur $\{x_5\}$ est exploitée. Cela va permettre, non pas d'identifier un nouveau cluster, mais de circonscrire la poursuite de la recherche en largeur à un sous-graphe duquel a été enlevé le sommet x_8 car il n'est plus connecté au reste des autres sommets. La recherche en largeur se poursuit donc mais sur la partie de X_1 qui n'a pas été parcourue, soit $\{x_9, x_{10}, x_{11}\}$ (dont $\{x_8\}$ a été supprimé). Nous constatons alors que le niveau suivant, uniquement constitué du sommet x_9, forme un séparateur de taille 1, donc inférieure à la borne fixée $S = 2$. Le parcours en largeur est alors stoppé et le nouveau cluster est maintenant constitué : $E_1 = \{x_3, x_4, x_5, x_6, x_7, x_9\}$. Les composantes connexes induites par la suppression de E_1 de X_1 sont : $X_{1_1} = \{x_8\}$ et $X_{1_2} = \{x_{10}, x_{11}\}$. Elles sont ajoutées à F.

3.3.3 Heuristique à base de triangulation

Une heuristique qui a recours à la triangulation est aussi proposée. Elle exploite et définit une nouvelle version de Min-$Fill$ que nous appelons Min-$Fill$-MG (pour Min-$Fill$ Mieux Guidé). À l'instar de Min-$Fill$, elle vise à minimiser la largeur de la décomposition w^+. Elle permet d'exploiter la topologie du graphe à décomposer contrairement à Min-$Fill$. Dans ce cas, la ligne 9 de l'algorithme 3.2 consiste à :

- Considérer le graphe $G[V_i \cup X_i] = G_i$,
- Compléter V_i dans G_i,
- Calculer le graphe G'_i qui représente l'application de Min-$Fill$ au graphe G_i,
- Calculer CM l'ensemble de cliques maximales de G'_i,
- Choisir E_i qui consiste en une clique de CM incluant V_i.

La complétion de V_i consiste à ajouter dans G les arêtes absentes entre chaque paire de sommets de V_i. L'étape suivante consiste à trianguler $G_i = G[V_i \cup X_i]$ en utilisant Min-$Fill$ et à construire un graphe triangulé G'_i des sommets de $V_i \cup X_i$. Nous précisons qu'il n'est pas nécessaire d'ordonner tous les sommets du graphe selon Min-$Fill$ mais uniquement ceux de V_i. Une fois les sommets de V_i ordonnés, les autres sommets restants peuvent être ignorés sachant que nous sommes uniquement intéressés par les cliques contenant V_i. Les cliques maximales de G'_i sont ensuite mémorisées dans CM. Comme $G[V_i]$ est une clique, l'inclusion de V_i dans au moins une des cliques mémorisées dans CM est garantie. En fait, cette clique va constituer le nouveau cluster noté E_i qui sera rajouté à l'ensemble des clusters déjà calculés.

Nous considérons maintenant le graphe de la figure 3.11 et nous illustrons le calcul de cluster suivant E_1 par Min-$Fill$-MG. L'application de Min-$Fill$ au graphe $G[V_1 \cup X_1]$ génère l'ensemble des cliques (si nous calculons toutes les cliques) $CM = \{\{x_3, x_4, x_7\}, \{x_3, x_5, x_6, x_7\}, \{x_5, x_6, x_7, x_9\}, \{x_5, x_8\}, \{x_9, x_{10}, x_{11}\}\}$. Comme $V_1 \subset \{x_3, x_4, x_7\}$ alors $E_1 =$

$\{x_3, x_4, x_7\}$ et donc $X_{1_1} = \{x_5, x_6, x_8, x_9, x_{10}, x_{11}\}$ sera ajoutée à F. Le fait que E_i soit une des cliques maximales de CM garantit que E_i ne serait jamais inclus dans un cluster formé ultérieurement.

3.3.4 Validité de H-TD

Pour chacune des heuristiques mentionnées ci-dessus, qu'elle soit basée sur une triangulation ou pas, le calcul de E_i est suivi du calcul de nouvelles composantes connexes induites par la suppression des sommets de E_i dans $G[X_i]$. Cela permet notamment de séparer les composantes indépendantes du graphe et de calculer des décompositions plus adaptées à la topologie du graphe. Ces composantes connexes sont insérées à leur tour dans la file d'attente F pour être traitées ultérieurement. Le graphe à décomposer sera entièrement recouvert par des clusters lorsque F devient vide. Ainsi, nous obtenons une collection de clusters correspondant à une décomposition valide du graphe. Avant de s'intéresser à la validité de la décomposition induite par les clusters calculés, nous démontrons que les heuristiques H_i-TD et l'heuristique Min-$Fill$-MG satisfont la propriété 3.

Lemme 3 *Les heuristiques H_i-TD satisfont la propriété 3.*

Preuve : Nous démontrons cette propriété en montrant qu'à la ligne 9 de l'algorithme 3.2, nous garantissons que $N(v, X_i) \subseteq X_i''$. Cette propriété est en effet garantie par construction. En ce qui concerne l'heuristique H_1-TD, elle consiste à choisir un sommet v de V_i ayant le plus petit nombre de voisins dans X_i. Une fois v choisi, tout le voisinage de v est ajouté à E_i. Ainsi, $N(v, X_i) \subseteq X_i''$. Quant à H_2-TD, la connexité est vérifiée tout en veillant à ce qu'il existe au moins un sommet v tel que $N(v, X_i) \subseteq X_i''$. En ce qui concerne les heuristiques H_3-TD, H_4-TD et H_5-TD, quel que soit le niveau auquel elles s'arrêtent, elles vont inclure le niveau $u = 1$ de X_i dans E_i. Ainsi, E_i inclut forcément $N(V_i, X_i)$. Par conséquent, il existe nécessairement $v \in V_i$ tel que $N(v, X_i) \subseteq X_i''$. Nous en déduisons que $N(v, X_i) \subseteq X_i''$ pour toutes les heuristiques H_i-TD. Le fait qu'au moins un nouveau sommet de X_i est ajouté au cluster E_i qui n'est présent dans aucun cluster déjà formé, garantit que E_i ne peut être égal à aucun autre cluster. En outre, comme tout le voisinage du sommet v est ajouté à E_i, le sommet v n'apparaîtra dans aucun cluster pouvant être formé ultérieurement à partir des nouvelles composantes connexes calculées $X_{i_1}, X_{i_2}, \ldots X_{i_{k_i}}$. En conséquence, $\forall E_i, \nexists E_j$ tel que $i \neq j$ et $E_i \subseteq E_j$. □

Lemme 4 *L'heuristique Min-$Fill$-MG satisfait la propriété 3.*

Preuve : Nous commençons par démontrer qu'au moins un nouveau sommet de X_i sera ajouté à E_i. Comme nous choisissons à chaque étape une clique maximale issue de la triangulation de $G[V_i \cup X_i] = G_i$ qui contient V_i, il suffit de démontrer que ce cluster inclura V_i strictement (donc que $V_i \subset E_i$). Tout graphe triangulé possède au moins deux sommets *simpliciaux* (sommets dont l'ensemble des voisins forment une clique) qui ne sont pas voisins si le graphe n'est pas complet (cf. théorème de Dirac [Dirac, 1961]). Ainsi, nous savons qu'il existe deux sommets x et x' dans G_i' qui sont simpliciaux. Nous avons alors deux possibilités :

- x ou $x' \in V_i$. Sans manque de généralité, considérons $x \in V_i$. Puisque $x \in V_i$, nous savons que x possède au moins un voisin v dans X_i. De plus x est simplicial alors l'ensemble de ses voisins forme une clique. Or, comme V_i a été complété, x possède comme voisins au moins $(V_i \backslash \{x\}) \cup \{v\}$ qui est une clique de G_i'. Ainsi, $V_i \cup \{v\}$

est aussi une clique de G'_i. Cette clique est nécessairement contenue dans une clique maximale de G'_i et nous avons ainsi au moins une clique E_i de CM telle que $V_i \subset E_i$.

- Ni x, ni x' ne sont dans V_i. Nous démontrons le résultat par induction sur la taille de X_i. L'hypothèse est que pour $|X_i| = k \geq 2$ (car ni x, ni x' ne sont dans V_i et si $|X_i| = 1$, nous nous retrouvons dans le premier cas), il existe une clique de G'_i incluant strictement V_i. Pour la base de l'induction, nous avons $|X_i| = 2$. Dans ce cas, comme $G[X_i]$ est connexe, nécessairement x et x' sont voisins avant triangulation. Après triangulation, x et x' seront toujours voisins puisque la triangulation ne supprime pas d'arêtes. Ainsi nous concluons qu'une fois triangulé G_i n'aura pas deux sommets simpliciaux non voisins. Par conséquent, d'après la contraposée du théorème de Dirac, G'_i forme une clique car nous ne pouvons pas avoir deux sommets de X_i non voisins et simpliciaux. Ainsi, il existe bien une clique de G'_i induit par tous les sommets G'_i incluant strictement V_i.

  Nous supposons maintenant que la proposition est vérifiée pour $|X_i| = k \geq 2$ et nous prouvons qu'elle est aussi vérifiée pour $|X_i| = k + 1$. Considérons maintenant x. Si x inclut V_i dans son voisinage, $V_i \cup \{x\}$ est une clique de G'_i et le résultat est vérifié. Si x n'inclut pas V_i dans son voisinage, le sous-graphe de G'_i induit par la suppression de x possède k sommets et est triangulé. Dans ce cas, par hypothèse d'induction, il contient une clique incluant strictement V_i. *A fortiori*, G'_i inclut cette clique et le résultat est vérifié. Il faut noter que si x inclut une partie de V_i dans son voisinage, comme x inclut nécessairement au moins un autre sommet v de X_i dans son voisinage (sinon l'hypothèse de connexité de X_i avant triangulation de $G[V_i \cup X_i]$ ne serait pas vérifiée) et que x est simplicial, v sera également voisin de ces sommets dans le sous-graphe de G'_i induit par la suppression de x.

En conséquence, nous avons bien un élément E_i de CM tel que $V_i \subset E_i$. Comme au moins un nouveau sommet de X_i est ajouté au cluster E_i qui n'est présent dans aucun cluster déjà formé, E_i, cela garantit que E_i n'est égal à aucun autre cluster.

En outre, comme le cluster E_i est une clique maximale de G'_i, alors il ne sera inclus dans aucun autre cluster formé ultérieurement. En effet, supposons que E_i induit une composante connexe X_{i_p} dont le voisinage s'avère être E_i. Alors tout sommet de E_i est lié à au moins un sommet de X_{i_p}. L'ensemble de ces arêtes est noté $C_{\subseteq}$. Cependant, comme G'_i est le graphe triangulé correspondant à $G[V_i \cup X_i]$ et comme tout sous-graphe d'un graphe triangulé l'est aussi, le sous-graphe de G'_{i_p} correspondant à $G[E_i \cup X_{i_p}]$ est triangulé. Ce dernier contient nécessairement les arêtes de $C_{\subseteq}$ puisque la triangulation ne supprime pas d'arêtes. Or, E_i est une clique maximale. En termes de décomposition arborescente, E_i constituerait alors un cluster. En plus, les arêtes de $C_{\subseteq}$ doivent être forcément recouvertes. Nous savons également que les sommets de $C_{\subseteq}$ qui n'appartiennent pas à E_i appartiennent à la même composante connexe si le cluster E_i est supprimé de $G[E_i \cup X_{i_p}]$. Ainsi, ces sommets appartiennent aux clusters de $Desc(E_j)$ telle que E_j est un cluster fils de E_i dans la décomposition enracinée en E_i. Finalement, nous pouvons constater que ces sommets ne peuvent alors appartenir qu'à un seul cluster en raison de la troisième condition d'une décomposition arborescente. Nous obtenons alors un cluster contenant tous les sommets de E_i en plus d'autres sommets ; il inclut ainsi E_i. Cela est contradictoire avec le fait que E_i est une clique maximale. Par conséquent, E_i ne peut pas être inclus dans de futurs clusters. □

Théorème 5 *H-TD calcule les clusters d'une décomposition arborescente et garantit qu'aucun cluster n'est inclus dans un autre.*

Preuve : Il suffit de prouver la correction des lignes 6 à 13 de l'algorithme. Nous montrons d'abord que l'algorithme s'arrête. À chaque passage dans la boucle, au moins un sommet de X_i sera rajouté à l'ensemble X' et ce sommet n'apparaîtra pas plus tard dans un nouvel élément de la file d'attente puisque ces éléments sont définis par les composantes connexes de $G[X_i \backslash E_i]$, un sous-graphe qui contient strictement moins de sommets qu'il n'y en a dans X_i. Ainsi, après un nombre fini d'étapes, l'ensemble $X_i \backslash E_i$ sera un ensemble vide, et donc aucun nouvel ajout à F ne sera possible.

Nous montrons maintenant que l'ensemble des clusters $E_0, E_1, \dots E_l$ induit bien une décomposition arborescente de G. Nous le prouvons par une induction sur l'ensemble des clusters ajoutés en montrant que tous ces clusters vont induire une décomposition arborescente du graphe $G[X']$. Initialement, le premier cluster E_0 induit une décomposition arborescente du graphe $G[E_0] = G[X']$. L'hypothèse d'induction est que l'ensemble des clusters déjà ajoutés $E_0, E_1, \dots E_{i-1}$ induit une décomposition arborescente du graphe $G[E_0 \cup E_1 \cup \dots \cup E_{i-1}]$. Considérons maintenant l'ajout de E_i. Nous montrons que par construction, $E_0, E_1, \dots E_{i-1}$ et E_i induit une décomposition arborescente du graphe $G[X']$ en montrant que les trois conditions (i), (ii) et (iii) de la définition des décompositions arborescentes sont satisfaites.

(i) Chaque nouveau sommet ajouté dans X' appartient à E_i

(ii) Chaque nouvelle arête de $G[X']$ est recouverte par le cluster E_i.

(iii) Nous pouvons considérer deux cas différents pour un sommet $x \in E_i$, sachant que pour les autres sommets, la propriété est déjà satisfaite par l'hypothèse d'induction. Si $x \in V_i$, la propriété a déjà été vérifiée par hypothèse d'induction. Si $x \in E_i \backslash V_i$, x n'apparaît pas dans un autre cluster que E_i et la propriété est vérifiée.

Finalement, il est facile de voir que nous obtenons bien les clusters d'une décomposition arborescente du graphe $G[X']$, et par extension de G puisqu'en fin de traitement, nous avons $X' = X$. En addition, le fait que les heuristiques sur lesquelles se base *H-TD* satisfont la propriété 3 garantit qu'il n'existe aucun cluster de la décomposition arborescente qui est inclus dans un autre. □

3.3.5 Complexité de H-TD

Nous nous intéressons, dans cette partie, à la complexité du cadre de calcul de décompositions *H-TD*. Comme nous l'avons déjà vu, ce cadre peut être instancié par une heuristique de calcul du premier cluster E_0 et une heuristique qui se charge de calculer le prochain cluster E_i. La liste des heuristiques possibles risque d'être très longue. En effet, ces heuristiques peuvent avoir des objectifs très diversifiés cherchant à satisfaire des critères très différents l'un de l'autre allant des plus simples au plus sophistiqués. Ainsi, selon ce critère, l'heuristique peut être plus ou moins coûteuse. C'est pourquoi la complexité de *H-TD* dépend donc de la complexité de ces deux heuristiques.

Théorème 6 *La complexité en temps de H-TD est* $O(n(n+e)) + Complexité(H_{E_0}) + Complexité(H_{E_i})$ *avec* $Complexité(H_{E_0})$ *la complexité de l'heuristique de calcul du cluster* E_0 *et* $Complexité(H_{E_i})$ *la complexité de l'heuristique de calcul du cluster* E_i*.*

Preuve : La ligne 1 et les lignes 3-5 sont réalisables en temps linéaire, soit $O(n+e)$, puisque le coût de calcul des composantes connexes de $G[X_0 \backslash E_0]$ est borné par $O(n+e)$.

La complexité de la ligne 2 dépend de l'heuristique employée pour le calcul du cluster E_0. Sa complexité est alors $Complexité(H_{E_0})$. Nous analysons maintenant le coût de la boucle (ligne 6). Tout d'abord, notons qu'il y a nécessairement moins de n insertions dans la file F car, à chaque passage, dans la boucle, nous sommes assurés qu'au moins un nouveau sommet aura été rajouté dans X', et donc supprimé de l'ensemble des sommets n'ayant pas encore été traités. Nous analysons maintenant le coût de chaque traitement associé à l'ajout d'un nouveau cluster, dont nous donnons pour chacun, sa complexité globale.

- Ligne 7 : l'obtention du premier élément X_i de F est faisable en $O(n)$, c'est-à-dire en $O(n^2)$ globalement.
- Ligne 8 : l'obtention du voisinage $V_i \subseteq X'$ de X_i dans G est faisable en $O(n+e)$, et donc en $O(n(n+e))$ globalement.
- Ligne 9 : le coût de cette étape dépend de l'heuristique choisie pour le calcul du prochain cluster E_i et est $Complexité(H_{E_i})$.
- Lignes 10 et 11 : chacune de ces étapes est réalisable en $O(n)$, c'est-à-dire globalement en $O(n^2)$.
- Ligne 12 : le coût de la recherche des composantes connexes de $G[X_i \backslash E_i]$ est en $O(n+e)$. Ainsi le coût de cette étape est en $O(n(n+e))$.
- Ligne 13 : l'insertion de X_{i_j} dans F est réalisable en $O(n)$, c'est-à-dire globalement en $O(n^2)$ puisqu'il y a moins de n insertions dans F. □

Notons que si les heuristiques choisies ont des complexités ne dépassant pas $O(n(n+e))$ la complexité globale de H-TD serait alors en $O(n(n+e))$. Le coût de calcul de la décomposition reste alors raisonnable. En particulier, cette complexité dépend uniquement du nombre initial d'arêtes e et non pas de e' comme c'est le cas de certaines méthodes de calcul de décomposition basées sur la triangulation.

Nous allons voir dans ce qui suit que les heuristiques H_1-TD, H_2-TD, H_3-TD, H_4-TD et H_5-TD peuvent garantir une telle complexité.

Théorème 7 *La complexité en temps de H_1-TD, H_2-TD, H_3-TD, H_4-TD et H_5-TD est $O(n(n+e))$.*

Preuve : Pour chaque heuristique, nous fournissons le coût de la ligne 9.

- Pour H_1-TD, cette étape est réalisée en $O(n)$, c'est-à-dire en $O(n^2)$ globalement.
- Pour H_2-TD, cette étape est réalisée en $O(n+e)$, c'est-à-dire en $O(n(n+e))$ globalement. En effet, trouver X_i'' tel que $V_i \cup X_i''$ soit connexe est faisable en $O(n(n+e))$ globalement. Plus précisément, le test de connexité de $G[E_i]$ qui est faisable en $O(n+e)$, est fait une seule fois pour chaque sommet ajouté.
- Pour H_3-TD, la recherche des sommets au niveau u, c'est-à-dire les voisins des sommets au niveau $u-1$ dans $X_i \backslash E_{i_{u-1}}$ est réalisable en $O(n+e)$. Le coût du calcul des composantes connexes de $G[X_i \backslash E_{i_{u-1}}]$ est faisable en $O(n+e)$. Comme dans le pire cas, chaque sommet est à un niveau différent, cette étape est faite au plus n fois, c'est-à-dire un coût global en $O(n(n+e))$.

- Le cas de H_4-TD est similaire à celui de H_3-TD. En effet, la recherche des sommets au niveau u est faisable en $O(n+e)$. En outre, le calcul des composantes connexes de $G[X_i \backslash E_{i_{u-1}}]$ et de leur séparateurs associés est réalisable en $O(n+e)$. D'où le coût global de cette étape est en $O(n(n+e))$.

- Pour H_5-TD, l'analyse est identique à H_4-TD.

Finalement, la complexité temporelle des algorithmes H_1-TD, H_2-TD, H_3-TD, H_4-TD et H_5-TD est de $O(n(n+e))$. □

Théorème 8 *La complexité en temps de l'algorithme Min-Fill-MG est* $O(n^2(n+e'))$ *avec* e' *le plus grand nombre d'arêtes obtenu suite à une triangulation de Min-Fill.*

Preuve : La différence entre *Min-Fill-MG* et les autres heuristiques H_i-TD réside dans l'emploi de la triangulation lors de la construction du prochain cluster E_i. Plus précisément, à la ligne 9 de l'algorithme 3.2, le sous-ensemble X_i'' est calculé grâce à *Min-Fill*. En effet, *Min-Fill-MG* calcule une triangulation de $G_i = G[V_i \cup X_i]$ en utilisant l'heuristique *Min-Fill* et construit le cluster E_i qui n'est qu'une clique issue de la triangulation contenant V_i. Comme *Min-Fill* a une complexité de $O(n(n+e'))$, alors globalement le coût est en $O(n^2(n+e'))$ avec e' le plus grand nombre d'arêtes obtenu suite à une triangulation d'un G_i par *Min-Fill*. Ainsi, la complexité temporelle de l'algorithme *Min-Fill-MG* est de $O(n^2(n+e'))$. □

Notons que les complexités des heuristiques H_i-TD et *Min-Fill-MG* supposent que la complexité du calcul du premier cluster n'excèdent par leur complexités globales.

3.4 Étude expérimentale

Dans cette section, nous évaluons le cadre de calcul de décompositions H-TD. D'une part, nous nous intéressons à son utilisation dans l'optique de calculer des décompositions minimisant la largeur. D'autre part, nous évaluons son intérêt du point de vue de la résolution d'instances CSP et WCSP.

Benchmark utilisé Nous considérons 3 ensembles d'instances :

- I_1 : Le premier ensemble I_1 rassemble 33 instances issues du dépôt UCI sur les Réseaux de Croyance et quelques instances du problème de coloration de graphes. Il s'agit d'instances dont la largeur arborescente est connue [Berg and Järvisalo, 2014]. Le nombre n de sommets des graphes varie de 11 à 128 sommets et le nombre e d'arêtes varie de 20 à 2 556 arêtes.

- I_2 : Le deuxième ensemble I_2 porte principalement sur 1 859 instances CSP issues de la compétition CSP de 2008 [CP0, 2008]. Pour établir cette sélection, nous avons exclu les instances ayant des décompositions triviales (par exemple les instances ayant un graphe complet) ainsi que les instances ayant des contraintes globales. Les instances retenues représentent la majorité des familles d'instances. Le nombre n de variables varie de 6 à 14 930 et le nombre de contraintes m varie de 5 à 77 305. Le nombre e d'arêtes varie de 25 à 2 641 722 arêtes.

- I_3 : Le troisième ensemble I_3 rassemble le benchmark d'instances disponible à l'adresse `http://genoweb.toulouse.inra.fr/~degivry/evalgm`. Il est composé de plus de 3 000 instances incluant, entre autres, des modèles graphiques stochastiques de l'évaluation UAI de 2008 et 2010, des instances de la compétition PIC 2011 et des instances de la compétition MiniZinc 2012 et 2013. Nous avons écarté les instances résolues pendant la phase de prétraitement qui sera décrite dans la partie 3.4.3 pour obtenir au final un benchmark composé de 2 444 instances. Le nombre n de variables varie de 4 à 137 808 avec des domaines d'une taille variant de 2 à 503. Le nombre de fonctions de coûts varie de 11 à 1 799 150 et leur arité est comprise entre 2 à 372.

À noter que parmi toutes ces instances, certaines correspondent en fait à des hypergraphes dont les décompositions arborescentes sont obtenues en travaillant sur leur 2-section.

Réalisation des expérimentations Les méthodes de décompositions sont implémentées en C++ au sein de notre bibliothèque. Toutes les expérimentations ont été réalisées sur des serveurs lame sous Linux Ubuntu 14.04 dotés chacun de deux processeurs Intel Xeon E5-2609 à 2,4 GHz et de 32 Go de mémoire.

3.4.1 Minimisation de la largeur de la décomposition calculée

Dans cette sous-section, nous nous focalisons sur la largeur des décompositions produites par *Min-Fill* et par les deux heuristiques de *H-TD* visant la minimisation de la largeur de la décomposition : H_1-TD et *Min-Fill-MG*. Nous évoquons tout d'abord le protocole expérimental.

3.4.1.1 Protocole expérimental

Pour *Min-Fill* et *Min-Fill-MG*, les égalités rencontrées au niveau du nombre d'arêtes à ajouter pour compléter le voisinage d'un sommet, sont cassées grâce à un critère sur 3 niveaux qui privilégie :

- le sommet ayant le plus petit nombre de voisins non encore numérotés dans le graphe courant triangulé,
- en cas d'égalité, le sommet ayant le plus petit degré dans le graphe courant triangulé,
- en cas d'égalité, le sommet apparaissant le premier dans l'ordre lexicographique.

Pour H_1-TD, le choix du premier cluster (E_0) consiste à calculer une clique maximale dans le graphe. Pour *Min-Fill-MG*, le choix du premier cluster consiste à prendre la première clique calculée par *Min-Fill*.

Les expérimentations portent sur les ensembles d'instances I_1 et I_2.

3.4.1.2 Observations et analyse des résultats

Comparaison de la largeur obtenue par rapport à la largeur exacte Afin de comparer la largeur des décompositions arborescentes produites par *Min-Fill*, H_1-TD et *Min-Fill-MG* par rapport à la largeur exacte, nous considérons les instances du benchmark I_1 dont la largeur arborescente est connue. La table 3.1 présente les résultats obtenus pour quelques-unes d'entre elles. Le nombre d'instances considéré peut paraître faible, mais il faut se rappeler que déterminer la largeur arborescente d'un graphe quelconque est un problème NP-difficile. D'ailleurs, les méthodes exactes ne sont vraiment opérationnelles

Instances	n	e	w	*Min-Fill*	H_1-*TD*	*Min-Fill-MG*
				w^+	w^+	w^+
myciel4	23	71	10	**11**	**11**	**11**
mildew	35	80	4	**4**	7	**4**
queen6_6	36	290	25	26	**25**	**25**
barley	48	126	7	**7**	10	**7**
eil51.tsp	51	140	8	10	**9**	**9**
celar02	100	311	10	**10**	11	**10**
miles500	128	1170	22	23	28	**22**
child	20	30	3	**3**	**3**	**3**
hailfinder	56	99	4	**4**	6	**4**
hepar2	70	158	6	**6**	**6**	**6**

TABLE 3.1 – Nombre de sommets et d'arêtes, largeur arborescente (optimum) et largeur des décompositions produites par *Min-Fill*, H_1-*TD* et *Min-Fill-MG* pour des instances dont la largeur w est connue. Ces instances proviennent du dépôt UCI sur les Réseaux de Croyance et du problème de coloration de graphes (benchmark I_1). Les meilleures largeurs obtenues pour chaque instance sont en gras.

que sur des graphes de petite taille. Une alternative consiste à procéder à un encadrement de la largeur arborescente par des bornes inférieures et supérieures. Mais, faute de disposer de bornes de qualité, cette solution n'aboutit que très rarement à déterminer w.

Nous avons observé que *Min-Fill-MG* trouve une décomposition optimale pour 25 instances, et que *Min-Fill* trouve une décomposition optimale pour 23 instances tandis que H_1-*TD* n'y parvient que pour 12 instances. Dans les cas où la décomposition n'est pas optimale, la largeur obtenue est souvent proche de l'optimum. Ceci illustre bien l'aptitude de ces méthodes heuristiques à calculer des décompositions de qualité en temps raisonnable. En effet, chaque décomposition est ici calculée en moins de $0,06$ s. Cela justifie l'intérêt de ces heuristiques par rapport aux méthodes exactes. À titre d'exemple, l'instance *eil51* nécessite près de deux heures pour démontrer que la largeur arborescente est 8 sur une machine Intel Xeon quad core sous Linux Ubuntu 10.04 à 2,8 GHz et 32 Go de mémoire dans [Berg and Järvisalo, 2014].

Comparaison des largeurs obtenues selon l'heuristique considérée À présent, nous comparons les différentes heuristiques de décomposition entre elles. Nous considérons pour cela les instances du benchmark I_2. La table 3.2 fournit les largeurs des décompositions obtenues pour une sélection d'instances représentatives des différentes tendances observées. En comparant H_1-*TD* à *Min-fill*, nous nous apercevons que H_1-*TD* produit des décompositions ayant une largeur inférieure ou égale à celles de *Min-Fill* pour 1 119 des 1 859 instances. Pour 786 de ces instances, H_1-*TD* améliore la largeur des décompositions produites par *Min-Fill*. L'amélioration peut être très significative comme c'est la cas, par exemple, pour l'instance bqwh-18-141-37_ext avec une largeur de 49 pour H_1-*TD* contre 73 pour *Min-Fill*. Concernant *Min-Fill-MG*, les largeurs produites sont strictement meilleures que celles de *Min-Fill* pour 613 instances et de qualité égale pour 924 instances. À nouveau, le gain peut être important. Par exemple, la largeur de la décomposition de l'instance ii-8e2 est de 149 pour *Min-Fill-MG* contre 179 pour *Min-Fill*. Enfin, H_1-*TD* obtient des largeurs strictement inférieures à celles de *Min-Fill-MG* pour 751 instances, égale pour 322 instances et strictement supérieures pour 786 instances.

En ce qui concerne le temps d'exécution, *Min-Fill* calcule l'ensemble des décomposi-

Instances	n	e	*Min-Fill*		H_1-*TD*		*Min-Fill-MG*	
			tps	w^+	tps	w^+	tps	w^+
1-insertions-6-4	607	6 337	0,25	187	0,02	**173**	3,5	201
ii-8e2	1 740	10 785	0,57	179	0,05	167	7,76	**149**
3-fullins-4-7	405	3 524	0,07	123	0,01	**96**	0,5	106
aim-200-1-6-3	400	1 119	0,03	91	0,01	90	0,55	**87**
blast-floppy1-6	719	7 818	0,02	**44**	0,02	61	0,26	45
bmc-ibm-02-04	2 810	14 610	0,34	150	0,16	**139**	5,5	148
cache-inv8-ucl	2 355	7 896	0,15	71	0,1	147	3,5	**67**
graph4	400	2 244	0,05	100	0,01	101	0,43	**96**
js-taillard-20-100-2	400	4 180	0,12	152	0,02	**134**	1,64	143
elf-rf8	6 059	23 172	1,71	177	1,19	430	32,24	**165**
fapp17-0300-9	300	2 056	0,1	153	$<$ 0,01	149	1,9	**146**
geo50-20-d4-75-85_ext	50	418	$<$ 0,01	**17**	$<$ 0,01	21	$<$ 0,01	**17**
hanoi4	1 436	9 031	0,86	226	0,13	353	26,36	**203**
bqwh-18-141-37_ext	141	883	0,01	73	$<$ 0,01	**49**	0,1	67
ii-32c3	558	9 198	0,06	**92**	0,06	101	1,37	**92**
will199GPIA-6	701	6 772	0,12	106	0,02	**103**	2,65	105

TABLE 3.2 – Nombre de sommets et d'arêtes, largeur des décompositions produites par *Min-Fill*, H_1-*TD* et *Min-Fill-MG* pour des instances CSP de la compétition de solveurs de 2008 (benchmark I_2). Les meilleures largeurs obtenues pour chaque instance sont en gras.

tions en 12 460 *s* contre 21 643 *s* pour *Min-Fill-MG* tandis que H_1-*TD* n'a besoin que de 928 *s*. En effet, les instances décomposées en moins d'une seconde représentent 88, 27 % des instances pour *Min-Fill*, 94, 99 % pour H_1-*TD* et 70, 84 % pour *Min-Fill-MG*. En outre, les instances décomposées en moins d'une minute représentent 96, 28 % pour *Min-Fill*, 99, 94 % pour H_1-*TD* et 90, 47 % pour *Min-Fill-MG*. Ainsi, H_1-*TD* est clairement plus rapide que *Min-Fill* tandis que *Min-Fill-MG* requiert un temps d'exécution plus long que les deux autres méthodes. Ces résultats étaient prévisibles au regard de la complexité des algorithmes. Le coût élevé de *Min-Fill-MG* vient essentiellement du coût de la triangulation réalisée à chaque étape pour calculer le prochain cluster E_i. La qualité de la décomposition calculée par H_1-*TD* est parfois moins bonne que celle calculée par *Min-Fill-MG*. Ceci est notamment dûe au choix de l'ensemble X_i'' qui exige l'ajout de tous les voisins d'un sommet v de V_i dans X_i à X_i''. En particulier, si le sommet v est l'unique sommet de V_i il sera nécessairement choisi quel que soit son nombre de voisins dans X_i. Ce cas pourrait être très pénalisant pour le calcul de w^+.

Bilan En conclusion, H_1-*TD* et *Min-Fill-MG* ont clairement montré leur capacité à produire souvent des décompositions de meilleure qualité que celles obtenues par la méthode de référence *Min-Fill*. H_1-*TD* a notamment mis en évidence l'intérêt des méthodes de décomposition ne s'appuyant pas sur la triangulation, non seulement par la qualité des décompositions calculées, mais aussi et surtout par le temps de décomposition en permettant de décomposer l'ensemble des instances 13 fois plus rapidement que *Min-Fill*. Quant à *Min-Fill-MG*, elle produit particulièrement des décompositions de meilleure qualité sur une majorité des instances, mais est malheureusement pénalisée par un temps de décomposition qui peut être parfois long. D'ailleurs, *Min-Fill-MG* ne parvient pas à calculer certaines décompositions dans un temps limite de 15 minutes ce qui n'est pas sans

impact sur le nombre total d'instances où *Min-Fill-MG* améliore les décompositions de *Min-Fill* et H_1-TD. H_1-TD et *Min-Fill-MG* prouvent finalement que la prise en compte de la topologie du graphe permet d'améliorer le calcul des décompositions en qualité et/ou en temps.

3.4.2 Efficacité de la résolution pour le problème CSP

Nous comparons ici les décompositions calculées du point de vue de l'efficacité de la résolution des instances CSP. Par souci de simplicité, nous notons parfois H_i l'heuristique H_i-TD. Nous évoquons tout d'abord le protocol expérimental.

3.4.2.1 Protocole expérimental

Au niveau des décompositions, nous considérons :

- *Min-Fill* : c'est l'heuristique de l'état de l'art qui est connue pour sa bonne approximation de la largeur arborescente,
- H_1-TD et *Min-Fill-MG* : leur intérêt au regard de la minimisation de la largeur de la décomposition w^+ a été mis en évidence dans la partie 3.4.1,
- H_2-TD : pour sa garantie de construire des clusters connexes,
- H_3-TD : pour sa construction de clusters ayant plusieurs fils,
- H_4-TD et H_5-TD : pour leur contrôle de la taille des séparateurs.

L'heuristique H_2-TD a été introduite dans [Jégou and Terrioux, 2014b] et présentée en plusieurs versions selon le choix des sommets suivants permettant de construire le cluster connexe. Nous choisissons ici l'heuristique qui consiste à choisir comme sommet suivant le sommet ayant le nombre maximum de voisins dans le cluster (ce qui correspond à l'heuristique $NV4$ dans [Jégou and Terrioux, 2014b]). Au niveau des heuristiques H_4-TD et H_5-TD, elles exploitent une taille maximale de séparateurs dépendant de l'instance, fixée à 5% du nombre de variables de l'instance dans la limite d'au moins 4 variables et au plus 50 et sont notées $H_4^{5\%}$ et $H_5^{5\%}$.

En ce qui concerne les algorithmes exploités, comme référence des méthodes énumératives, nous considérons l'algorithme MAC+RST exploitant les techniques de redémarrage et l'enregistrement des nogoods comme décrit dans les parties 2.2.4.3 et 2.2.4.7. Pour la résolution en se basant sur une décomposition arborescente, nous prenons en compte BTD que nous désignons par BTD-MAC vu son exploitation de MAC pour résoudre chaque cluster comme décrit dans la partie 2.2.5.1. Aussi, nous exploitons BTD-MAC+RST intégrant en plus de BTD-MAC les techniques de redémarrage et d'enregistrements de nogoods comme décrit dans la partie 2.2.5.1. Nous choisissons comme premier cluster racine le cluster ayant le plus grand rapport nombre de contraintes sur la taille du cluster moins un. L'intuition derrière cette heuristique découle du principe *first-fail*. Nous essayons de choisir le cluster induisant le sous-problème le plus contraint. Il en est de même pour le choix du cluster racine à chaque redémarrage. En effet, à chaque redémarrage le cluster racine choisi est celui ayant la somme maximale de poids de contraintes (pondération de contraintes de dom/wdeg) qui intersectent le cluster. Comme le cluster racine est le premier cluster à être affecté, veiller à ce que ses variables soient jugées parmi les plus pertinentes peut être très avantageux pour la suite de la résolution. Au niveau du choix du prochain cluster, lorsqu'un cluster possède plusieurs fils, celui qui a le séparateur de

Algorithme	*Min-Fill*		H_1		*Min-Fill-MG*	
	#rés	temps	#rés	temps	#rés	temps
BTD-MAC	1 348	34 416	1 305	28 768	1 281	39 113
BTD-MAC+RST	1 507	33 632	1 485	28 433	1 425	40 117

TABLE 3.3 – Nombre d'instances résolues et temps d'exécution en secondes pour *BTD-MAC* et *BTD-MAC+RST* selon les décompositions minimisant la largeur w^+.

Algorithme	H_2		H_3		$H_4^{5\%}$		$H_5^{5\%}$	
	#rés	temps	#rés	temps	#rés	temps	#rés	temps
BTD-MAC	1 424	21 860	1 487	28 792	1 527	26 170	1 524	26 143
BTD-MAC+RST	1 536	22 854	1 558	26 418	1 588	24 038	1 586	24 520

TABLE 3.4 – Nombre d'instances résolues et temps d'exécution en secondes pour *BTD-MAC* et *BTD-MAC+RST* selon les décompositions satisfaisant d'autres critères.

plus petite taille est choisi d'abord. La cohérence d'arc est appliquée en prétraitement via *AC*-3^{rm} et durant la résolution via *AC*-8^{rm} (visitées dans la partie 2.2.4.3). En effet, les expérimentations montrent que ces deux méthodes de filtrage sont parmi les méthodes les plus efficaces. Toutes les méthodes de résolution utilisent l'heuristique dom/wdeg pour choisir la prochaine variable à instancier. En ce qui concerne les algorithmes exploitant les redémarrages, ils utilisent une politique de redémarrage géométrique avec un ratio de 1,1 et 100 comme nombre initial de retours en arrière autorisés.

Pour chaque instance, le temps d'exécution (incluant le temps de calcul de la décomposition dans le cas échéant) est limité à 15 minutes et l'utilisation mémoire à 16 Go.

Les expérimentations portent sur l'ensemble d'instances I_2.

3.4.2.2 Observations et analyse des résultats

Comparaison générale des décompositions Les deux tables 3.3 et 3.4 montrent le nombre d'instances résolues parmi les 1 859 instances et le temps total d'exécution pour *BTD-MAC* et pour *BTD-MAC+RST* selon les décompositions exploitées. La table 3.3 montre les résultats pour les décompositions visant à minimiser la largeur de la décomposition w^+ tandis que la table 3.4 s'occupe des décompositions dont le but est de satisfaire des critères jugés pertinents au regard de l'efficacité de la résolution comme la connexité des clusters, le nombre de fils d'un cluster et la taille des séparateurs de la décomposition. Nous remarquons que quel que soit l'algorithme utilisé (sans ou avec redémarrages) l'exploitation des décompositions minimisant la largeur n'est pas bénéfique à l'égard de la résolution. Certes, le nombre d'instances résolues par *BTD-MAC+RST* augmente par rapport à *BTD-MAC* pour *Min-Fill*, H_1 et *Min-Fill-MG*. En effet, l'exploitation des redémarrages permet, entre autres, de créer plus de dynamicité au niveau de l'heuristique de choix de variables en rendant possible le changement du cluster racine à chaque redémarrage. Ainsi, cela compense en partie les restrictions imposées par la décomposition notamment celles qui visent à minimiser la taille des clusters, c'est-à-dire ayant le moins de variables propres comme décrit dans la partie 3.2.2. Cependant, l'emploi des redémarrages semble tout de même insuffisant au vu du nombre d'instances résolues par les décompositions cherchant à satisfaire d'autres critères comme le montre la table 3.4.

Algorithme	*Min-Fill*	H_2	H_3	$H_4^{5\%}$	$H_5^{5\%}$
BTD-MAC+RST	350 432	313 554	297 318	267 938	270 220

TABLE 3.5 – Temps d'exécution en secondes pour *BTD-MAC+RST* selon les décompositions pour toutes les instances de I_2 (non résolues incluses).

Comparaison de *Min-Fill* vis-à-vis de $H_{\{2,3,4,5\}}$ Par la suite, nous ne retenons que *Min-Fill* parmi les heuristiques de la table 3.3 étant donné qu'il s'agit de l'heuristique de l'état de l'art et celle donnant les meilleurs résultats dans la table 3.3. En plus, nous ne considérons que les méthodes exploitant les redémarrages vu que la tendance reste la même en comparant *BTD-MAC+RST* à *BTD-MAC* selon les différentes décompositions exploitées. En comparant le comportement de *BTD-MAC+RST* selon la décomposition employée, nous constatons que la prise en compte de la connexité des clusters permet d'augmenter le nombre d'instances résolues par rapport à celles résolues grâce à *Min-Fill*. D'ailleurs, dans ce benchmark, seulement 10% des décompositions calculées par *Min-Fill* ne contiennent que des clusters connexes. Mais, comme expliqué dans la partie 3.2.2, la présence des clusters non connexes a un impact négatif sur l'efficacité de la résolution. Ainsi, la construction de clusters connexes permet d'augmenter le nombre d'instances résolues. En particulier, si *Min-Fill* permet de résoudre 1 507 instances, H_2 en résout 1 536. Nous avons aussi examiné la connexité des clusters des autres décompositions. En ce qui concerne H_3-*TD*, pour 91% des instances tous les clusters sont connexes. Quant à H_4-*TD* et H_5-*TD*, pour respectivement 86% et 84% des instances la décomposition ne contient que des clusters connexes. Nous remarquons que pour ces heuristiques, un bon pourcentage des décompositions n'inclut que des clusters connexes. En effet, pour calculer le prochain cluster E_i, ces heuristiques effectuent un parcours en largeur à partir de V_i en rajoutant à E_i tous les sommets du niveau parcouru. Ce faisant, elles augmentent leur chance de construire des clusters connexes. L'heuristique H_3 montre aussi son intérêt pratique. Plus précisément, 1 558 instances sont résolues grâce à son exploitation. Ces résultats prouvent que la prise en compte de la topologie du graphe en essayant de détecter ses parties indépendantes lors du calcul des décompositions permet d'améliorer l'efficacité de la résolution. Finalement, H_4 et H_5 visent à limiter la taille des séparateurs produits. Nous observons une augmentation significative du nombre d'instances résolues atteignant respectivement 1 588 et 1 586 instances pour $H_4^{5\%}$ et $H_5^{5\%}$. Ces résultats soulignent l'intérêt de borner la taille des séparateurs déjà évoqué dans la partie 3.2.2. Notons aussi que ces résultats sont cohérents avec ceux de [Jégou et al., 2005], qui insistent sur ce paramètre pour renforcer l'efficacité de la résolution. L'augmentation du nombre d'instances résolues ne semble pas pénaliser les temps cumulés d'exécution. En effet, l'augmentation du nombre d'instances résolues grâce à H_2 (29 instances en plus par rapport à *Min-Fill*) est accompagnée d'une diminution du temps total de résolution qui passe de 33 632 *s* à seulement 22 854 *s* pour H_2. Le temps d'exécution de *BTD-MAC+RST* exploitant H_3 s'élève à 26 418 *s* s'expliquant par le nombre d'instances additionnelles résolues grâce à H_3. Quant à $H_4^{5\%}$ et $H_5^{5\%}$, elles permettent de résoudre le plus grand nombre d'instances en moins de 24 500 *s*. Notons que les temps de décompositions sont assez remarquables du fait que *Min-Fill* requiert 13 895 *s* pour décomposer l'ensemble des instances tandis que les autres décompositions n'ont pas besoin de plus de 550 *s*, voire seulement 15 *s* pour $H_4^{5\%}$. D'ailleurs, si nous retranchons les temps de décomposition des temps cumulés d'exécution, le temps d'exécution avec *Min-Fill* devient 24 719 *s* (8 913 *s* pour décomposer les instances résolues). En revanche, il n'y a pas de changements significatifs des temps d'exécution de *BTD* avec les autres décompositions. Finalement, nous montrons dans la table 3.5 les

Algorithme	$Min\text{-}Fill^{5\%}$		$H_2^{5\%}$		$H_3^{5\%}$	
	#rés	temps	#rés	temps	#rés	temps
BTD-MAC+RST	1 561	32 640	1 578	28 396	1 576	24 993

TABLE 3.6 – Nombre d'instances résolues et temps d'exécution en secondes pour *BTD-MAC+RST* selon les décompositions après l'application de la stratégie de fusion.

temps d'exécution de *BTD-MAC+RST* avec les décompositions $Min\text{-}Fill^{5\%}$, $H^{5\%}_{\{2,3,4,5\}}$-*TD* pour toutes les instances de I_2. Autrement dit, nous ajoutons 900 *s* par instance non résolue au temps d'exécution cumulé. Nous remarquons alors que le fossé s'élargit entre *Min-Fill* et $H^{5\%}_{\{4,5\}}$ pour le temps d'exécution cumulé total.

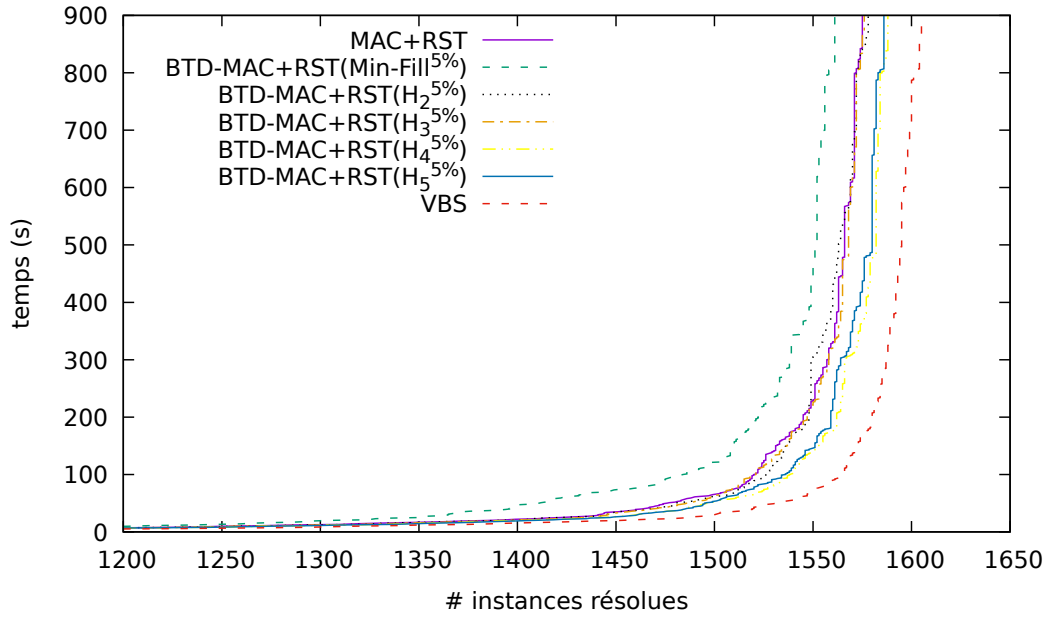


FIGURE 3.12 – Le nombre cumulé d'instances résolues grâce à chaque décomposition pour les 1 859 instances considérées du benchmark I_2.

Comparaison de *Min-Fill* vis-à-vis de $H_{\{2,3,4,5\}}$ pour la même taille de séparateur Dans le but de comparer d'une façon équitable la qualité des différentes décompositions vis-à-vis de l'efficacité de la résolution, nous appliquons la stratégie de fusion [Jégou et al., 2005] pour les décompositions calculées par *Min-Fill*, H_2 et H_3 (ainsi notées $Min\text{-}Fill^{5\%}$, $H_2^{5\%}$ et $H_3^{5\%}$). Nous limitons ainsi la taille des séparateurs à 5% du nombre de variables de l'instance dans la limite d'au moins 4 variables et au plus 50. Ainsi, n'importe quel cluster dont la taille de son séparateur dépasse cette limite sera fusionné avec son père (en mettant en commun des sommets du cluster et de son père pour former un seul cluster). Ce processus est répété jusqu'à ce qu'il n'y ait plus de séparateurs de taille non convenable. Comme prévu, le tableau 3.6 montre que toutes les décompositions permettent désormais de résoudre plus d'instances. Plus précisément, $Min\text{-}Fill^{5\%}$ permet de résoudre 1 561 instances, soit 54 instances de plus que *Min-Fill*. $H_2^{5\%}$ et $H_3^{5\%}$ permettent aussi à leur tour de résoudre plus d'instances avec respectivement 1 578 et 1 576 instances résolues au lieu de 1 536 et 1 558 instances. La tendance reste alors presque la même, avec $H_4^{5\%}$ et $H_5^{5\%}$ surpassant les autres heuristiques de décomposition. Notons

que l'augmentation du nombre d'instances résolues s'accompagne soit d'une amélioration de temps de résolution cumulé, soit d'une légère augmentation de temps justifiée par le nombre additionnel d'instances résolues (cf. tables 3.4 et 3.6).

Comparaison de BTD-MAC+RST avec les décompositions Min-$Fill^{5\%}$ et $H^{5\%}_{\{2,3,4,5\}}$ vis-à-vis de MAC+RST et du VBS Nous nous intéressons également à la comparaison de ces heuristiques à l'égard de MAC+RST et du VBS (pour *Virtual best solver*) qui correspond au meilleur temps de résolution parmi MAC+RST et BTD-MAC+RST utilisant chacune des différentes décompositions. La figure 3.12 montre le nombre cumulé d'instances résolues par BTD-MAC+RST avec chaque décomposition, par MAC+RST ou par le VBS. De son côté, MAC+RST résout 1 575 instances en 25 567 s. Quant au VBS, il résout 1 605 instances en 22 693 s. Il paraît clairement que BTD-MAC+RST exploitant n'importe quel $H^{5\%}_i$ est maintenant beaucoup plus compétitif face à MAC+RST notamment avec $H^{5\%}_4$ et $H^{5\%}_5$ qui permettent de surpasser certainement MAC+RST. Cependant, comme prévu, MAC+RST permet d'obtenir de meilleurs résultats qu'avec BTD exploitant Min-$Fill^{5\%}$. Finalement, les résultats de $H^{5\%}_4$ et $H^{5\%}_5$ semblent très intéressants au vu des résultats du VBS qui ne résout que 17 instances additionnelles.

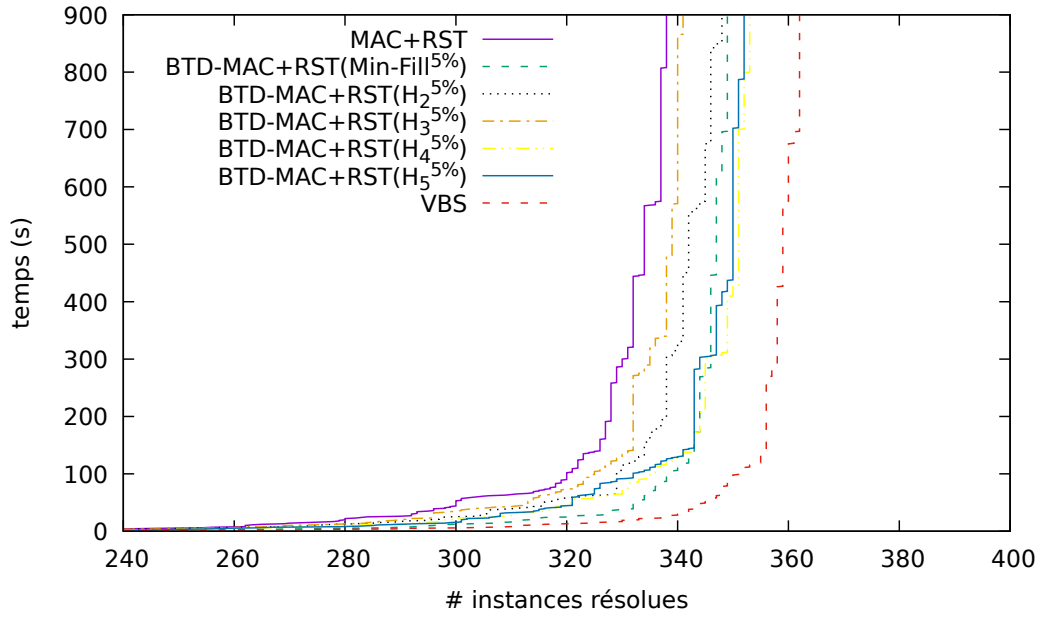


FIGURE 3.13 – Le nombre cumulé d'instances résolues grâce à chaque décomposition uniquement pour les instances ayant une décomposition de w^+ tel que $\frac{n}{w^+} \geq 5$ parmi les instances du benchmark I_2.

Nous nous limitons maintenant aux instances ayant de bonnes propriétés topologiques, c'est-à-dire une largeur w^+ (celle calculée par Min-$Fill$) telle que $\frac{n}{w^+} \geq 5$. Nous disposons alors de 417 instances. Nous constatons particulièrement dans la figure 3.13 que Min-$Fill^{5\%}$ a une meilleure performance sur ces instances grâce à sa bonne approximation de la largeur arborescente. Ainsi, Min-$Fill^{5\%}$ permet de résoudre plus d'instances qu'avec $H^{5\%}_2$, $H^{5\%}_3$ et MAC+RST qui résout le plus petit nombre d'instances. Néanmoins, nous remarquons que BTD-MAC+RST parvient toujours à résoudre plus d'instances avec $H^{5\%}_4$ et $H^{5\%}_5$ qu'avec Min-$Fill^{5\%}$. Notons que, pour ces instances, le pourcentage de décompositions ne contenant que des clusters connexes est de 20% pour Min-$Fill^{5\%}$, 66%

pour $H_3^{5\%}$, 60% pour $H_4^{5\%}$ et de 52% pour $H_5^{5\%}$. Ainsi, sur ces instances, bien que 80% des décompositions de *Min-Fill* contiennent des clusters non connexes, son exploitation permet d'améliorer BTD avec $H_2^{5\%}$ ou $H_3^{5\%}$. Notons d'ailleurs que si la non-connexité d'un cluster E_i correspond au cas de la figure 3.4, elle n'induit aucun problème au niveau de la résolution. En effet, seule la non-connexité de $G[E_i \backslash (E_i \cap E_{p(i)})]$ est susceptible d'avoir une influence sur l'efficacité de la résolution. Celle-ci dépend donc du choix du cluster racine qui peut changer à chaque redémarrage. En tous cas, nous tenons à rappeler que la non-connexité ne dégrade pas systématiquement l'efficacité de la résolution.

Bilan Pour conclure, les expérimentations affirment clairement que les heuristiques de calcul de décomposition visant uniquement à minimiser la largeur de la décomposition ne sont pas les mieux adaptées pour résoudre les instances CSP. Au vu de l'efficacité des algorithmes classiques énumératifs comme MAC+RST, la question de l'intérêt des méthodes basées sur une décomposition se pose légitimement. Cependant, l'introduction du cadre de calcul de décompositions H-TD a permis de proposer d'autres heuristiques de calcul de décompositions dont le but ne se limite pas à s'occuper de la taille des clusters mais se soucie au-delà d'autres critères qui semblent plus intéressants à l'égard de la résolution des instances CSP. C'est ainsi que la connexité des clusters a pu être prise en compte avec H_2-TD, en plus du nombre de fils d'un cluster avec H_3-TD et de la taille des séparateurs avec H_4-TD et H_5-TD. Toutes ces heuristiques ont particulièrement montré que les méthodes basées sur une décomposition peuvent être compétitives vis-à-vis des algorithmes basés sur MAC pour la résolution des instances CSP, voire les surpasser comme avec H_4-TD et H_5-TD.

3.4.3 Efficacité de la résolution pour le problème WCSP

Dans cette sous-section, nous évaluons l'intérêt pratique du cadre de calcul de décompositions H-TD pour la résolution des problèmes d'optimisation sous contraintes WCSP. Mais, au préalable, nous décrivons le protocole expérimental suivi.

3.4.3.1 Protocole expérimental

Nous considérons les algorithmes $HBFS$ et BTD-$HBFS$ décrits respectivement dans la partie 2.4.4.1 et la partie 2.4.4.2. Nous exploitons leurs implémentations fournies dans *Toulbar2* [TOU, 2006]. Ces deux algorithmes sont connus pour leur efficacité incontestable. Leur point fort principal est leur comportement *anytime* leur permettant d'améliorer la borne inférieure et la borne supérieure en permanence. Le paramétrage de $HBFS$, à savoir les valeurs de α_{hbfs}, de β_{hbfs} et de N_{hbfs}, sont identiques à celles utilisées dans [Allouche et al., 2015], c'est-à-dire respectivement 5%, 10% et 10 000.

En ce qui concerne les décompositions, nous considérons *Min-Fill* en tant qu'heuristique de l'état de l'art en utilisant son implémentation fournie dans *Toulbar2*. Une deuxième version de *Min-Fill* est aussi utilisée et serait référencée par *Min-Fill*[4]. Elle se distingue de *Min-Fill* par l'absence de séparateurs de taille supérieure à 4. Elle résulte de l'application de la stratégie de fusion utilisée dans [Jégou et al., 2005] comme dans la partie 3.4.2. Son utilisation avec BTD-$HBFS$ correspond à l'algorithme BTD-$HBFS^{r_4}$ dans [Allouche et al., 2015]. Du côté de H-TD, nous retenons les décompositions H_2, H_3 et H_5. Nous ne rapportons pas les résultats obtenus par H_1 et par *Min-Fill-MG* qui comme pour le problème CSP (cf. la partie 3.4.2) n'ont pas montré d'intérêt vis-à-vis de l'efficacité de la résolution. Nous choisissons de représenter uniquement les résultats de H_5 qui sont très proches de ceux de H_4. Toutefois, nous gardons H_5 vu qu'elle permet de

Algorithme	*Min-Fill*		*Min-Fill*^{4}	
	#rés.	temps	#rés.	temps
BTD-HBFS	1 712	26 291	1 995	91 232

TABLE 3.7 – Nombre d'instances résolues et temps d'exécution en secondes pour *BTD-HBFS* selon les décompositions de l'état de l'art.

Algorithme	H_2		H_3		H_5^{25}		$H_5^{5\%}$	
	#rés.	temps	#rés.	temps	#rés.	temps	#rés.	temps
BTD-HBFS	1 946	76 018	1 989	57 348	2 006	58 826	2 028	58 043

TABLE 3.8 – Nombre d'instances résolues et temps d'exécution en secondes pour *BTD-HBFS* selon les décompositions de *H-TD*.

détecter plus de séparateurs que H_4. Elle est déclinée en deux variantes notées H_5^{25} et $H_5^{5\%}$ permettant de limiter respectivement la taille maximale des séparateurs à 25 et à 5% du nombre de variables de l'instance dans la limite d'au moins 4 variables et au plus 50. Les décompositions H_i sont calculées au sein de notre propre bibliothèque et communiquées à *Toulbar2* par l'intermédiaire d'un fichier. Notons qu'à l'heure actuelle, compte tenu de leur efficacité pratique [Allouche et al., 2015], *HBFS* et *BTD-HBFS* avec *Min-Fill*^{4} peuvent être considérés comme étant les références en tant qu'algorithmes de résolution des instances WCSP respectivement sans et avec exploitation de la structure.

En ce qui concerne la configuration de *Toulbar2*, un prétraitement reposant sur la cohérence d'arc est appliqué via *VAC* [Cooper et al., 2008, 2010] en plus de l'application de l'algorithme *MSD* (pour *Min Sum Diffusion*) [Kovalevsky and Koval, 1975; Cooper et al., 2010] avec 1 000 itérations. La cohérence d'arc est ensuite maintenue pendant la résolution grâce à *EDAC* [Givry et al., 2005]. L'heuristique de choix de variables est dom/wdeg associée à l'heuristique du dernier conflit toutes les deux décrites dans la partie 2.2.4.6. Pour les algorithmes comme *BTD*, le cluster racine choisi est le plus grand cluster (pour *Min-Fill* car c'est l'heuristique choisie dans [Allouche et al., 2015]) ou le cluster maximisant le ratio entre le nombre de contraintes du cluster et sa taille (pour les autres décompositions).

Pour chaque instance, chacun des algorithmes de résolution considérés dispose de 20 minutes (ce temps incluant, le cas échéant, le temps requis pour calculer une décomposition) et de 16 Go de mémoire.

L'ensemble d'instances utilisé est le benchmark I_3.

3.4.3.2 Observations et analyse des résultats

Nous comparons le comportement de *BTD-HBFS* en fonction des différentes décompositions exploitées. En ce qui concerne le temps de calcul des décompositions, nous constatons que le calcul des décompositions avec H_i ($i \in \{2, 3, 5\}$) est nettement plus rapide qu'avec *Min-Fill*. Plus précisément, le temps total de calcul des décompositions ne dépasse pas 1 200 s pour les 2 430 instances décomposées par H_i ($i \in \{3, 5\}$) et 4 655 s pour les 2 427 instances décomposées par H_2 tandis que *Min-Fill* requiert 19 044 s pour décomposer 2 415 instances.

Comparaison de *H-TD* à *Min-Fill* et *Min-Fill*^{4} Les tables 3.7 et 3.8 fournissent le nombre d'instances résolues et le temps d'exécution cumulé pour *BTD-HBFS* respectivement avec les décompositions de l'état de l'art et les décompositions de *H-TD*. Tout

d'abord, notons que *Min-Fill* résout le plus petit nombre d'instances en 20 minutes (seulement 1 712 instances). Cela montre que, malgré la difficulté du problème de l'optimisation sous contraintes, les heuristiques de décomposition cherchant uniquement à minimiser la taille des clusters ne sont pas les plus efficaces. Les résultats montrent aussi que les autres paramètres ont plus d'impact, comme la connexité des clusters (H_2), le nombre de fils d'un cluster (H_3) ou la taille des séparateurs ($Min\text{-}Fill^4$ et H_5). Le fait de borner la taille des séparateurs semble jouer un rôle crucial dans l'augmentation de l'efficacité de la résolution. En effet, si $BTD\text{-}HBFS$ avec H_2 et H_3 résout respectivement 1 946 et 1 989 instances, les décompositions dont la taille de séparateurs est bornée permettent de résoudre 1 995 pour $Min\text{-}Fill^4$ et plus de 2 000 pour H_5 (2 006 pour H_5^{25} et 2 028 pour $H_5^{5\%}$). La comparaison de $Min\text{-}Fill^4$ avec H_5 montre cependant que H_5 permet à $BTD\text{-}HBFS$ de résoudre plus d'instances notamment avec $H_5^{5\%}$ qui résout 2 028 instances contre 1 995 instances pour $Min\text{-}Fill^4$. La figure 3.14 compare les temps d'exécution de $BTD\text{-}HBFS$ avec $H_5^{5\%}$ à $BTD\text{-}HBFS$ avec $Min\text{-}Fill^4$. Elle montre que la plupart des instances sont résolues plus rapidement par $BTD\text{-}HBFS$ avec $H_5^{5\%}$ que par $BTD\text{-}HBFS$ avec $Min\text{-}Fill^4$. En

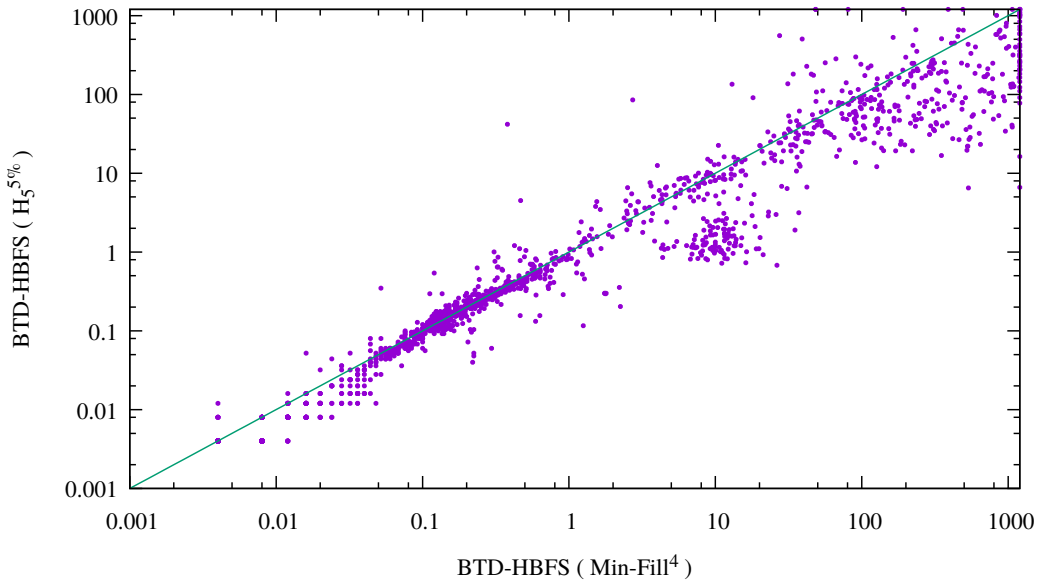


FIGURE 3.14 – Comparaison des temps d'exécution de $BTD\text{-}HBFS$ avec $H_5^{5\%}$ à $BTD\text{-}HBFS$ avec $Min\text{-}Fill^4$ pour les 2 444 instances du benchmark I_3.

outre, $BTD\text{-}HBFS$ associé à $Min\text{-}Fill^4$ requiert 91 232 s en temps d'exécution cumulé contre seulement 58 043 s pour $H_5^{5\%}$. La figure 3.15 montre le nombre cumulé d'instances résolues grâce à chaque décomposition. La figure 3.15 illustre l'analyse réalisée ci-dessus. Elle montre de nouveau clairement d'un côté l'inefficacité de *Min-Fill* par rapport aux autres décompositions pour la résolution des instances WCSP et de l'autre côté l'intérêt des heuristiques bornant la taille des séparateurs notamment $H_5^{5\%}$. Notons enfin que ces résultats sont cohérents avec ceux obtenus pour le problème de décision CSP dans [Jégou et al., 2015a] et ceux obtenus dans la partie 3.4.2.

Comparaison de $BTD\text{-}HBFS(H_5^{5\%})$ à $HBFS$ Nous comparons maintenant $BTD\text{-}HBFS$ à $HBFS$. La figure 3.15 permet de situer $HBFS$ par rapport à $BTD\text{-}HBFS$ selon la décomposition employée. Nous retenons par la suite la décomposition $H_5^{5\%}$ qui permet

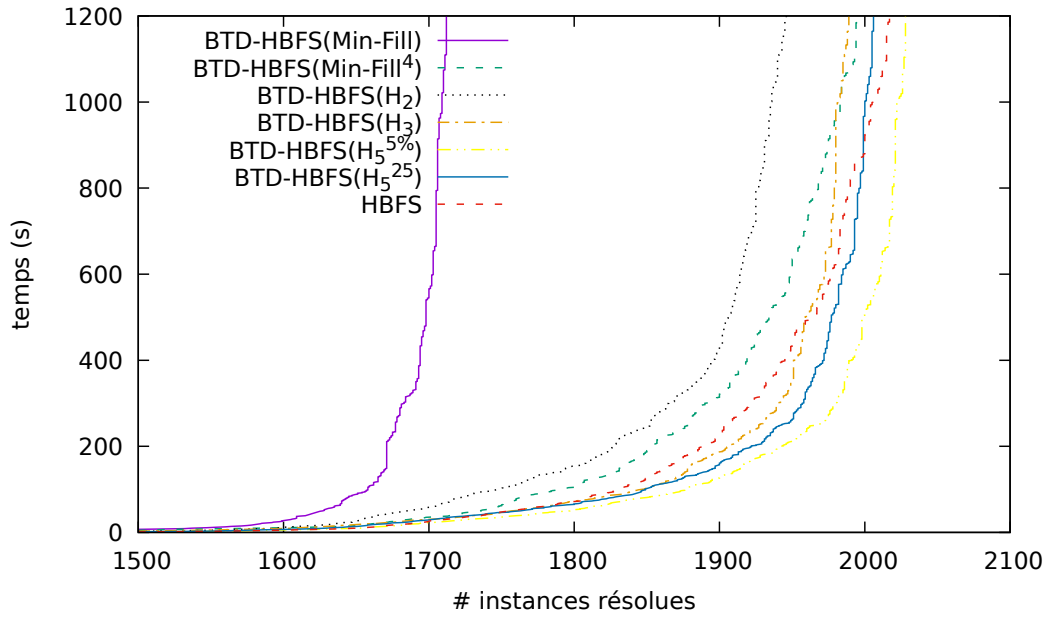


FIGURE 3.15 – Le nombre cumulé d'instances résolues grâce à chaque décomposition et par $HBFS$ pour les 2 444 instances considérées du benchmark I_3.

d'obtenir les meilleurs résultats avec $BTD\text{-}HBFS$ par rapport aux autres décompositions. $BTD\text{-}HBFS$ résout plus d'instances que $HBFS$, à savoir 2 028 instances contre 2 017 instances pour $HBFS$. En plus, $BTD\text{-}HBFS$ ne nécessite que 58 043 s en temps cumulé d'exécution contre 84 657 s pour $HBFS$. Cela peut s'expliquer essentiellement par les enregistrements faits par $BTD\text{-}HBFS$ au niveau des séparateurs, à savoir une borne inférieure et une borne supérieure de l'optimum d'un sous-problème. De plus, $BTD\text{-}HBFS$ détecte les indépendances entre les sous-problèmes, ce qui n'est pas le cas de $HBFS$. La figure 3.16 présente une comparaison des temps d'exécution de $BTD\text{-}HBFS$ à $HBFS$. $BTD\text{-}HBFS$ associée à $H_5^{5\%}$ prouve son intérêt pratique par rapport à $HBFS$. En effet, la plupart des instances qui sont mieux résolues par $HBFS$ ont un temps de résolution comparable à celui de $BTD\text{-}HBFS$. Cependant, nous pouvons remarquer qu'il y a de nombreuses instances qui sont résolues bien plus rapidement avec $BTD\text{-}HBFS$ qu'avec $HBFS$. Lorsque nous nous limitons aux instances non triviales pour $HBFS$ (i.e. résolues par $HBFS$ en plus de 10 secondes ou non résolues), les résultats sont encore plus favorables pour $BTD\text{-}HBFS$. Ce résultat est montré par la figure 3.17 qui compare les temps d'exécution cumulés de $BTD\text{-}HBFS$ et $HBFS$ sur ces instances. Finalement, en examinant parmi ces instances, les instances résolues par les deux méthodes nous obtenons 350 instances. Ces instances sont résolues par $HBFS$ en 74 019 s et en 46 192 s par $BTD\text{-}HBFS$.

Nous comparons aussi les bornes supérieures et les bornes inférieures rapportées par $HBFS$ et $BTD\text{-}HBFS$ dans le cas de dépassement du temps limite. Parmi les 377 instances qui ne sont pas résolues ni par $HBFS$, ni par $BTD\text{-}HBFS$, pour 151 instances, la borne inférieure calculée par $BTD\text{-}HBFS$ est strictement supérieure à celle de $HBFS$ contre 109 pour $HBFS$. En outre, pour 233 instances, la borne supérieure calculée par $BTD\text{-}HBFS$ est strictement inférieure à celle de $HBFS$ contre seulement 73 pour $HBFS$. Cela montre que même lorsque l'instance n'est pas résolue, $BTD\text{-}HBFS$ est capable de donner des approximations de meilleure qualité que $HBFS$. Pour avoir une idée plus

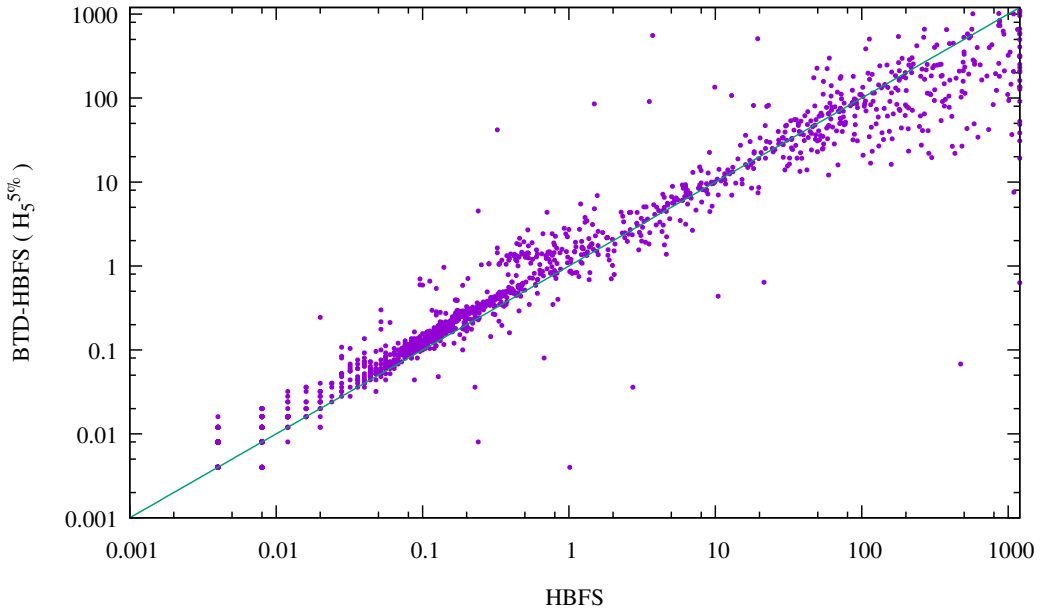


FIGURE 3.16 – Comparaison des temps d'exécution de $BTD\text{-}HBFS$ avec $H_5^{5\%}$ à $HBFS$ pour les 2 444 instances du benchmark I_3.

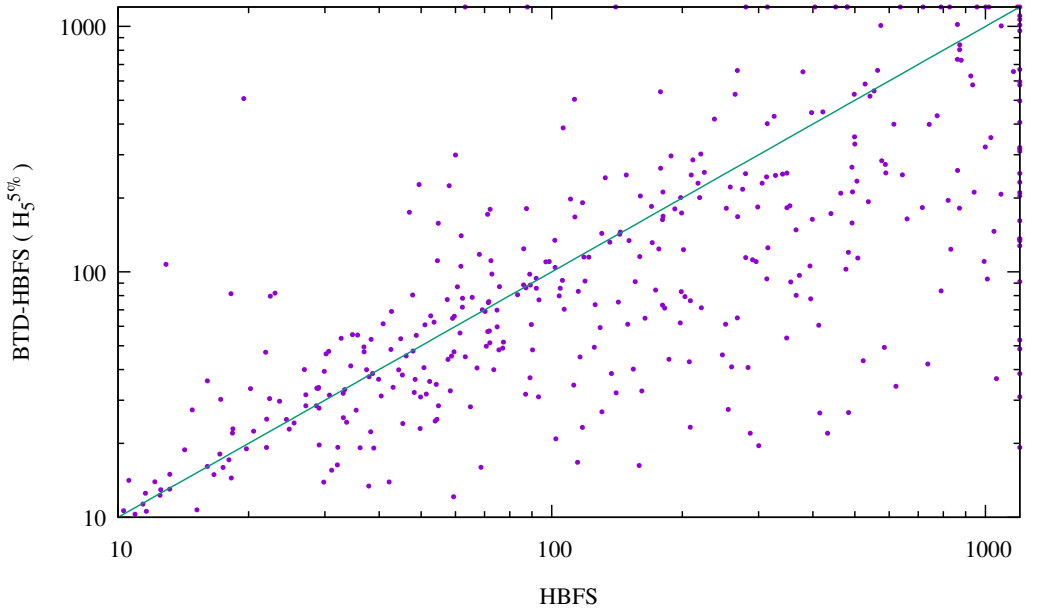


FIGURE 3.17 – Comparaison des temps d'exécution de $BTD\text{-}HBFS$ avec $H_5^{5\%}$ à $HBFS$ sur les instances non résolues par $HBFS$ en moins de 10 secondes du benchmark I_3.

précise sur la progression de la recherche menée par les deux méthodes, nous comparons les écarts obtenus entre la borne supérieure et la borne inférieure pour chaque instance. La figure 3.18 montre une comparaison de l'écart entre la borne supérieure et la borne inférieure pour $HBFS$ et $BTD\text{-}HBFS$. Dans la figure 3.19, nous limitons l'écart maximum à 10^5. Nous constatons à travers les deux figures que $BTD\text{-}HBFS$ semblent avoir des écarts plus petits que ceux obtenus avec $HBFS$. $BTD\text{-}HBFS$ est alors dans de nombreux

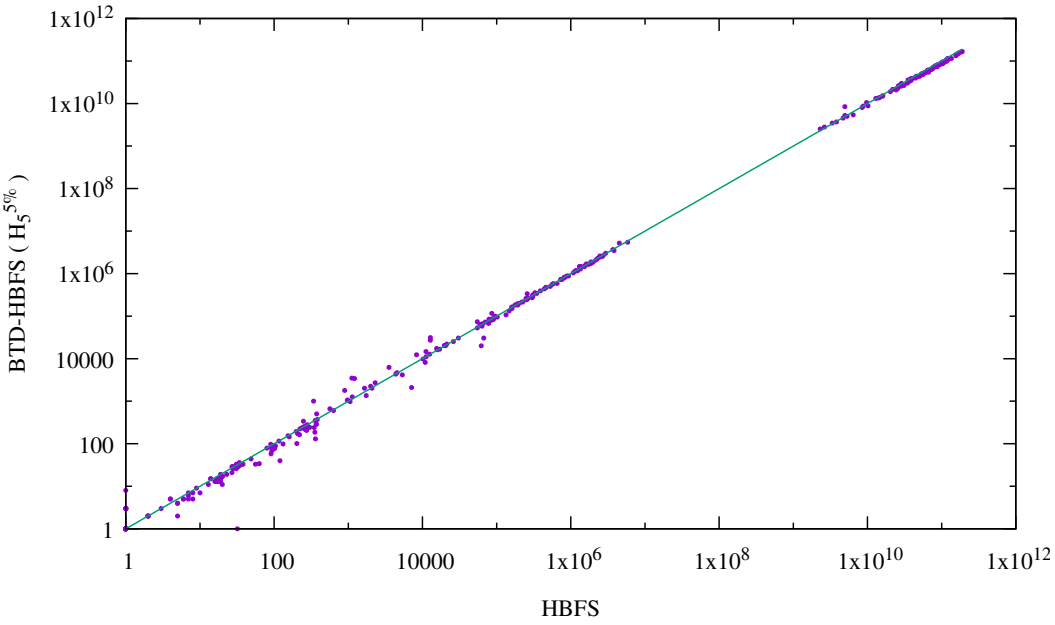


FIGURE 3.18 – Comparaison de l'écart entre la borne supérieure et la borne inférieure pour les 377 instances non résolues.

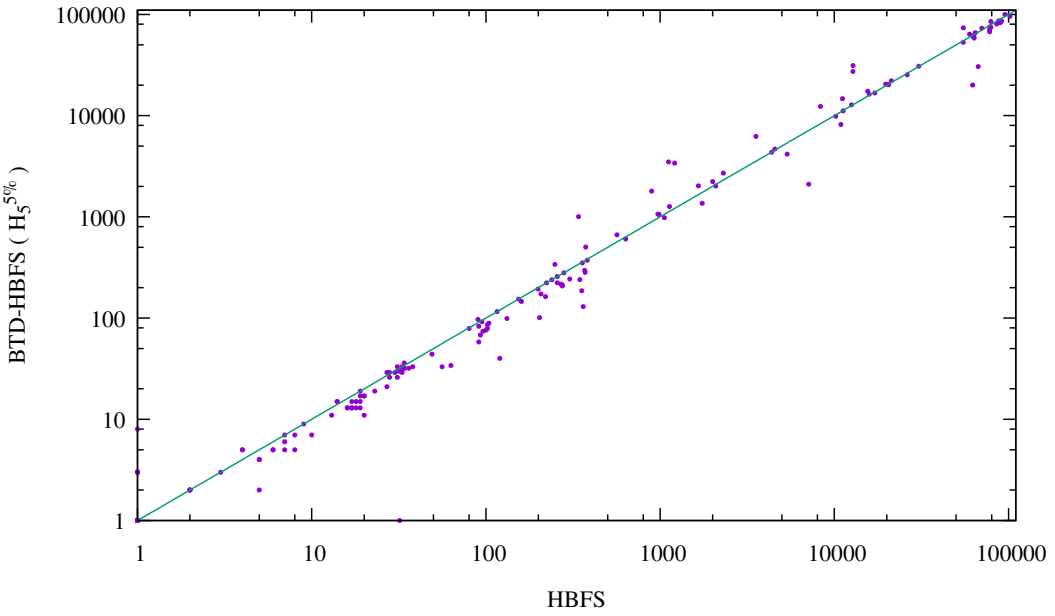


FIGURE 3.19 – Comparaison de l'écart entre la borne supérieure et la borne inférieure pour les 377 instances non résolues (écart limité à 10^5).

cas le plus proche de trouver l'optimum.

Bilan En conclusion, l'évaluation de *H-TD* dans le cadre de la résolution du problème *WCSP* rejoint les conclusions réalisées lors de son évaluation pour la résolution des instances *CSP*. En effet, l'utilisation de *Min-Fill* conjointement avec *BTD-HBFS* montre que les heuristiques dont l'optique est de minimiser la taille des clusters semble avoir un

intérêt moindre à l'égard de la résolution des instances WCSP. En revanche, l'exploitation des décompositions H_2, H_3 et H_5 améliore la performance de $BTD\text{-}HBFS$ et souligne son efficacité notamment avec H_5, en bornant la taille des séparateurs de la décomposition. Ces décompositions permettent également de mettre en valeur $BTD\text{-}HBFS$ vis-à-vis de $HBFS$, la référence en tant que méthode n'exploitant pas la structure du problème. Plus précisément, l'exploitation de $BTD\text{-}HBFS$ avec $H_5^{5\%}$ permet de résoudre plus d'instances que $HBFS$ tout en réalisant un temps cumulé nettement inférieur à celui de $HBFS$.

3.5 Conclusion

Les méthodes de résolution des instances (W)CSP basées sur la décomposition arborescente ont démontré leur intérêt théorique car elles permettent de garantir une borne de complexité en temps en $O(exp(w))$ tout en ayant une complexité en espace en $O(exp(s))$. Lorsque w est borné par une constante, ces méthodes assurent ainsi un temps de résolution polynomial. Le fait de calculer une décomposition optimale ayant une largeur $w^+ = w$ est un problème NP-difficile. C'est ainsi que les efforts se sont focalisés sur l'élaboration de méthodes heuristiques capables d'estimer au mieux cette largeur w dans un temps raisonnable.

Ces heuristiques, dont $Min\text{-}Fill$ fait office de référence dans la communauté CP et bien au-delà [Darwiche, 2009; Koller and Friedman, 2009; Dechter, 2013], sont, pour la plupart d'entre elles, basées sur la triangulation. Toutefois, la triangulation souffre de multiples défauts. D'une part, le fait qu'elle soit réalisée uniquement en se basant sur des critères ne tenant pas compte de la topologie du graphe, comme le degré des sommets dans le cas de $Min\text{-}Fill$, permet de renforcer *l'effet boule de neige*. Cela induit l'ajout d'arêtes non nécessaires du point de vue de la triangulation. D'autre part, l'ajout excessif d'arêtes provoque une augmentation du temps de triangulation et ainsi celui de la décomposition. Plus important, il pourrait être à l'origine de la sur-estimation du w^+ de la décomposition obtenue au final. Au-delà, même si la théorie affirme l'intérêt de minimiser w^+, la pratique ne semble pas en faveur de cette démarche, du moins pour la résolution des instances CSP et WCSP. La conception des méthodes de calcul des décompositions n'étant pas à la base faite en vue de résoudre des instances (W)CSP, elle s'oriente essentiellement vers la minimisation de ce paramètre. En revanche, elle néglige d'autres paramètres comme la connexité des clusters, la taille des séparateurs et la liberté de l'heuristique de choix de variables lors de la résolution.

Nous avons alors proposé un nouveau cadre algorithmique de calcul de décompositions, appelé $H\text{-}TD$. Il permet de calculer des décompositions arborescentes en traversant le graphe, en se basant sur des propriétés liées aux séparateurs et leurs composantes connexes associées. Il n'a pas alors forcément recours à une triangulation ce qui a permis d'améliorer considérablement le temps de calcul des décompositions en pratique. En plus, ce cadre est paramétrable afin de prendre en compte divers critères et répondre aux multiples besoins concernant les caractéristiques des décompositions à calculer. Ces critères peuvent effectivement être liés, par exemple, à la taille des clusters ou à la taille des séparateurs. Grâce à $H\text{-}TD$, nous avons alors introduit deux nouvelles heuristiques, $H_1\text{-}TD$ et $Min\text{-}Fill\text{-}MG$, dans le but de minimiser w^+. $H_1\text{-}TD$, contrairement à $Min\text{-}Fill$, n'est pas basée sur la triangulation ce qui lui permet d'être 13 fois plus rapide que $Min\text{-}Fill$ sur le benchmark I_2 (cf. la partie 3.4.1). Toutefois, $H_1\text{-}TD$ est capable de produire des décompositions de bonne qualité par rapport à $Min\text{-}Fill$. De son côté, $Min\text{-}Fill\text{-}MG$ emploie la triangulation de façon mieux guidée afin de prendre en compte la topologie du graphe. Bien que son temps de calcul soit élevé, $Min\text{-}Fill\text{-}MG$ permet d'obtenir des décompositions d'une

qualité significativement meilleure que $Min\text{-}Fill$. Malgré leur intérêt au niveau de la minimisation de w^+, ces heuristiques ne semblent pas être efficaces vis-à-vis de la résolution. C'est pourquoi nous avons proposé les heuristiques H_2-TD, H_3-TD, H_4-TD et H_5-TD qui tiennent compte respectivement de la connexité des clusters, du nombre de fils et de la taille des séparateurs. Qu'il s'agisse de résoudre des instances CSP ou WCSP, ces heuristiques ont permis d'améliorer l'efficacité des méthodes de résolution structurelles notamment en exploitant les décompositions bornant la taille des séparateurs comme H_5-TD. En outre, les algorithmes basés sur une décomposition sont désormais plus compétitifs face à ceux n'exploitant pas sur une décomposition comme MAC et $HBFS$ (cf. les parties 3.4.2 et 3.4.3).

Finalement, les décompositions employées dans ce chapitre sont calculées en amont de la résolution uniquement sur la base de propriétés structurelles du graphe. Ces décompositions seront utilisées tout au long de la résolution. Nous remarquons ainsi l'importance de cette première et seule décomposition calculée qui va imposer partiellement un ordre de choix de variables durant toute la résolution. Toutefois, cet ordre peut ne pas être en totale adéquation avec la nature de l'instance à résoudre. L'ordre de choix de variables étant particulièrement déterminant pour permettre une résolution efficace, il est donc primordial d'avoir plus de liberté quant au choix de la prochaine variable à instancier que celle offerte à travers l'ordre d'origine. Dans le prochain chapitre, nous proposons ainsi un cadre algorithmique permettant de relâcher progressivement les restrictions imposées par la décomposition. Ce faisant, l'algorithme vise à s'adapter aux spécificités de l'instance à résoudre en combinant l'*exploitation de la décomposition* et la *liberté d'instanciation de variables*.

Chapitre 4

Fusion dynamique de la décomposition dans le cas du problème CSP

Sommaire

4.1 Introduction

Nous avons vu dans le chapitre précédent que le calcul de la décomposition arborescente se fait habituellement en amont de la résolution. Ce calcul se fait essentiellement sur la base de critères structurels comme la taille des clusters, la taille des séparateurs ou la connexité des clusters. Même si certains paramètres sont connus pour jouer un rôle crucial dans l'efficacité de la résolution comme la taille des séparateurs [Jégou et al., 2005, 2015a] que nous avons pu valider expérimentalement dans le chapitre précédent, cela ne garantit pas que la décomposition exploitée pendant la résolution soit la plus adaptée au contexte de la résolution. En effet, une décomposition induit un ordre partiel sur les variables qui sera imposé à la recherche tout au long de la résolution. Or, cet ordre n'est pas nécessairement pertinent du point de vue de la résolution. Notamment, il peut être assez différent d'un ordre choisi par une heuristique de choix de variables ayant une totale liberté, ou ne pas respecter le principe du *first-fail*. En outre, le fait que la décomposition soit calculée avant la résolution, empêche de prendre en compte, lors de son calcul, certains critères qui relèvent de *la sémantique* du problème. Par voie de conséquence, l'ordre de choix de variables qui en découle n'intégrera en aucun cas de tels critères. Parmi les critères sémantiques, nous nous intéressons aux caractéristiques de l'instance à résoudre qui se dévoilent au fur et à mesure de l'avancement de la résolution. Par exemple, pour certaines contraintes nous pourrions effectivement calculer en amont le nombre de tuples autorisés par rapport au nombre de tuples possibles, ce qui pourrait constituer un indicateur de la difficulté de la contrainte. Toutefois, sa difficulté réelle est celle constatée pendant la résolution suite aux affectations des variables. En accumulant des informations correspondant aux états précédents de la résolution et à son état courant, nous construisons le contexte de la résolution. Les techniques permettant d'intégrer la sémantique du problème et l'exploiter par les méthodes de résolution sont des techniques dites *adaptatives*. La principale motivation derrière est de tirer profit du contexte courant de la recherche et aussi de son historique pour permettre à la recherche d'effectuer des choix plus judicieux et plus opportuns. Sur un plan plus général, l'adaptativité permettrait de construire des solveurs plus robustes et plus efficaces. De tels solveurs sont très utiles puisque leur utilisation ne requiert pas de l'expertise. L'adaptativité leur permet d'avoir un comportement relativement proche du meilleur comportement qu'il aurait pu avoir en faisant un choix différent au niveau de son paramétrage. Dans ce contexte, l'adaptativité vise à trouver la meilleure décomposition possible pour la résolution de l'instance considérée. Nous nous focalisons alors sur les techniques adaptatives dans la section suivante et nous expliquons la difficulté rencontrée par BTD et plus généralement par les méthodes relevant de ce type d'approche pour profiter pleinement des techniques adaptatives. Nous proposons ainsi dans la section 4.3 une exploitation dynamique adaptative des décompositions dans le cadre du problème CSP qui permettrait à BTD de se libérer de plus en plus des restrictions imposées par la décomposition. Nous illustrons son intérêt pratique et sa pertinence par une étude expérimentale dans la section 4.4.

4.2 Prise en compte du contexte courant de la résolution

Dans cette partie, dans un premier temps, nous allons revoir quelques méthodes utilisées dans le but de s'adapter au contexte de la résolution. Dans un second temps, nous allons parler des obstacles qui empêchent BTD d'exploiter pleinement leurs avantages.

4.2.1 Stratégies existantes

Les techniques que nous abordons tout d'abord concernent les heuristiques de choix de variables, dont les améliorations qui ont été apportées ont été sensiblement avantageuses aux algorithmes de recherche arborescente. Les heuristiques de choix de variables qui s'avèrent les plus efficaces aujourd'hui sont les heuristiques *adaptatives*. Non seulement elles sont dynamiques, mais elles permettent de s'adapter au contexte de la résolution. En effet, elles permettent de concilier les deux aspects de la résolution : l'aspect rétrospectif et l'aspect prospectif. L'aspect rétrospectif détermine la démarche à effectuer à la rencontre d'une incohérence. Il décide du choix de retour-arrière à faire ainsi que du choix des informations à apprendre si une telle approche est prévue. De son côté, l'aspect prospectif concerne l'assignation de la prochaine variable, soit le choix du couple $(variable, valeur)$ à faire ainsi que la réalisation des changements résultant de ce choix notamment en termes de filtrage de valeurs incohérentes et de détection d'une éventuelle incohérence. Les techniques adaptatives fusionnent ces deux points de vue ; elles permettent d'apprendre des informations sur les états précédents de la recherche et les exploitent, en plus des informations sur son état courant, afin de faire des choix plus judicieux pour la suite de la résolution. Par la suite, nous détaillons quelques-unes parmi les plus connues.

Diriger la recherche par les conflits [Boussemart et al., 2004] Cette heuristique, notée $dom/wdeg$, est la plus utilisée parmi les heuristiques de choix de variables existantes principalement à cause de son efficacité et de la simplicité de sa mise en œuvre. Elle vise à diriger la recherche vers les parties les plus difficiles et les plus problématiques. Elle s'inspire évidemment du principe *first-fail* [Haralick and Elliott, 1980]. Cette heuristique s'avère pertinente essentiellement dans le cas des problèmes où certaines contraintes occupent plus d'importance que d'autres et ainsi où certaines parties du problème semblent plus difficiles que d'autres, voire incohérentes. Par exemple, supposons que deux problèmes sont combinés (sans interaction entre les deux) et que le premier est facile tandis que le deuxième est incohérent. Dans ce cas, les incohérences rencontrées vont se concentrer au niveau des contraintes du sous-problème incohérent tandis que les contraintes du sous-problème facile seront beaucoup moins souvent sujettes à une impasse. Le but de cette heuristique est d'éviter le phénomène du *trashing* pouvant résulter d'une telle configuration. En effet, une heuristique de choix de variables naïve pourrait commencer par affecter les variables du sous-problème facile avant d'affecter celles du sous-problème incohérent et rencontrer à chaque fois une incohérence tardivement. Une heuristique plus « intelligente » constaterait cette différence d'importance entre les contraintes et privilégierait l'affectation des variables impliquées dans les contraintes les plus difficiles. Ce faisant, l'incohérence du problème est facilement prouvé. C'est fondamentalement ce que $dom/wdeg$ cherche à faire. Elle se base sur la notion de pondération de contraintes. En effet, elle affecte un poids $wdeg$ à chaque contrainte du problème et l'initialise à 1. Ce compteur sera incrémenté à chaque fois que le domaine d'une de ses variables devient vide. L'adaptation au contexte de la résolution se fait alors par le biais de la pondération des contraintes suite aux conflits rencontrés durant la recherche. En pratique, pendant la progression de la résolution, les contraintes les plus dures subissent une augmentation remarquable de leur poids associé $wdeg$ par rapport aux poids des autres contraintes. L'aspect apprentissage se renforce alors grâce à l'accumulation et l'enregistrement des conflits rencontrés via le poids $wdeg$. De son côté, l'aspect progression exploite ses enregistrements en choisissant comme variable suivante la variable x_i ayant le plus petit rapport $dom(x_i)/\alpha_{wdeg}(x_i)$ où $dom(x_i)$ est la taille du domaine courant de x_i et $\alpha_{wdeg}(x_i) = \sum_{x_i \in S(c_i):|Fut(c_i)|>1} wdeg(c_i)$ avec $Fut(c_i)$ l'ensemble de variables de $S(c_i)$ non encore assignées. Il est à noter qu'au début

dom/wdeg est identique à *dom/deg* vu que le poids de chaque contrainte est initialisé à 1. En comparant la démarche de cette heuristique et celle des autres heuristiques dynamiques, nous constatons, qu'à un instant donné, l'heuristique *dom/wdeg* dispose des informations sur les états précédents de la recherche ainsi que sur son état courant. Cependant, les heuristiques dynamiques traditionnelles exploitent uniquement des informations correspondant à son état courant comme la taille courante du domaine d'une variable. C'est ainsi que les heuristiques adaptatives sont capables de faire de choix plus judicieux et plus appropriés permettant une résolution plus efficace. Cette heuristique a également un intérêt majeur lorsque les redémarrages sont exploités. Elle permet, lors d'un redémarrage, de choisir d'abord les variables jugées comme les plus pertinentes et ainsi de diversifier les choix selon la connaissance qu'il a pu acquérir jusqu'à cet instant. L'heuristique *dom/wdeg* a montré son efficacité sur un large spectre de problèmes réels, académiques et aléatoires.

Diriger la recherche selon l'impact des variables [Refalo, 2004] L'impact d'une variable est une mesure qui indique l'importance d'une variable dans la réduction de la taille de l'espace de recherche. La taille de l'espace de recherche $\mathcal{E}$ est définie comme étant le produit de la taille courante des domaines des variables. Sachant que l'affectation d'une variable $x_i \leftarrow v_i$ réduit la taille des domaines des autres variables (par propagation), son impact est alors défini par $I(x_i \leftarrow v_i) = 1 - \frac{\mathcal{E}_{après}}{\mathcal{E}_{avant}}$ où $\mathcal{E}_{avant}$ et $\mathcal{E}_{après}$ désignent respectivement la taille de l'espace de recherche avant et après la réalisation de l'affectation. L'impact d'une affectation est ainsi plus élevé lorsque la réduction de l'espace de recherche est plus importante. L'impact d'une variable est déduit de l'impact des valeurs de son domaine et est calculé en fonction des impacts obtenus pendant les étapes précédentes de la recherche. L'heuristique de choix de variables basée sur la notion d'impact, notée IBS (pour *Impact-Based Search*), est dite adaptative ; elle choisit comme variable suivante la variable ayant le plus grand impact sur l'espace de recherche qui est une notion calculée en tenant compte de l'impact de cette variable pendant les étapes précédentes de la recherche. En privilégiant d'abord l'affectation des variables ayant l'impact le plus grand, nous dirigeons la recherche vers les parties les plus contraintes du problème. Si au début de la recherche la valeur de l'impact de chaque variable n'est pas encore suffisamment fiable, la progression de la recherche augmente de plus en plus l'exactitude de cette information. Lorsque les redémarrages sont exploitées conjointement, l'algorithme de recherche est capable de modifier les décisions réalisées vers la racine de l'arbre de recherche en se basant sur les dernières mises à jour des impacts des variables. Cette heuristique peut ainsi contribuer à diminuer la taille de l'arbre de recherche. L'exploitation des impacts des variables a permis de résoudre efficacement certains problèmes comme le problème de *complétion des carrés latins*, le problème du *sac à dos* et le problème du *carré magique* [Refalo, 2004].

Diriger la recherche selon l'activité des variables [Michel and Hentenryck, 2012] L'activité d'une variable est une mesure qui reflète le nombre de fois où cette variable est filtrée par propagation. L'heuristique basée sur la notion d'activité, notée ABS (pour *Activity-Based Search*), vise à exploiter les algorithmes de filtrage qui impliquent l'utilisation de mécanismes sophistiqués et à en tirer des informations pertinentes pour mieux guider la résolution. Elle associe à chaque variable un compteur qui sera mis à jour à chaque nœud de l'arbre de recherche. Selon cette heuristique, l'application de l'algorithme de filtrage suite à une décision, partitionne les variables en deux parties :

- les variables $X_f \subseteq X$ dont le domaine est réduit,

- les variables $X \backslash X_f$ dont le domaine reste inchangé.

Soit γ un réel tel que $0 \leq \gamma \leq 1$ qui permet de faire vieillir une valeur. L'activité d'une variable x_i, notée $A(x_i)$, est alors mise à jour selon les deux règles suivantes :

- $\forall x_i \in X$ tel que $|D_{x_i}| > 1$, $A(x_i) = A(x_i) \times \gamma$
- $\forall x_i \in X_f$, $A(x_i) = A(x_i) + 1$

C'est ainsi que toutes les variables non encore affectées et dont le domaine contient au moins deux valeurs subissent un vieillissement de la valeur de leur activité afin de donner de moins en moins d'importance aux conséquences des anciennes affectations. Au contraire, les activités des variables de X_f sont toutes incrémentées. Cette heuristique tend à guider la recherche en privilégiant d'abord l'instanciation de la variable ayant l'activité la plus élevée. L'intuition découle du principe *first-fail* en admettant que les variables filtrées le plus fréquemment semblent les plus difficiles à instancier. L'heuristique est clairement adaptative parce qu'elle maintient, via les activités des variables, des informations sur les états précédents de la résolution et décide du choix de la variable suivante à assigner sur la base de ces informations. L'heuristique ABS, comme $dom/wdeg$, peut être mise en œuvre sans difficulté particulière et sans être coûteuse en temps contrairement à IBS qui dépend de la taille des domaines des variables du problème. L'avantage de ABS par rapport à $dom/wdeg$ est qu'elle est orientée vers les variables et non pas vers les contraintes, dans le sens où $dom/wdeg$ donne le même poids à toutes les variables d'une contrainte lors de sa violation même si seulement un sous-ensemble de ses variables est éventuellement impliqué dans l'échec. L'heuristique ABS semble être la plus robuste, face à IBS et $dom/wdeg$, sur une sélection de benchmarks dont les instances du problème du *sac à dos* et celles du problème du *carré magique* [Michel and Hentenryck, 2012].

Adaptativité pour les redémarrages [Biere, 2008] Les techniques adaptatives peuvent être également utilisées pour mieux gérer les redémarrages. Redémarrer fréquemment la recherche a permis d'améliorer l'efficacité de la résolution de certaines instances. Cependant, le redémarrage très fréquent peut aussi être contre-productif dans d'autres cas. Ce problème a été abordé dans les solveurs SAT. Dans [Biere, 2008], Biere compte sur les techniques adaptatives pour déterminer la fréquence des redémarrages. Il introduit la notion d'*agilité* qui indique le taux des affectations récemment modifiées (affectation à 0 qui passe à 1 ou vice versa). Plus précisément, les affectations qui sont au centre d'intérêt sont les affectations forcées et non pas celles relevant d'une décision ou d'un choix. Une agilité élevée permet de déduire que les redémarrages doivent être plus espacés dans le temps, voire être suspendus à partir d'un certain seuil. Cependant, une agilité basse favorise des redémarrages plus fréquents. L'agilité est une mesure globale initialisée à 0 et mise à jour à chaque fois que l'affectation d'une variable est forcée. À l'instar de la notion d'activité d'une variable, l'agilité subit un vieillissement qui vise à donner plus de poids aux informations récoltées récemment. Les expérimentations ont montré une hausse considérable du nombre d'instances résolues pour les instances SAT dites *crafted*, notamment pour des instances incohérentes.

Adaptativité dans les solveurs CDCL (pour *Conflict Driven Clause Learning*) [Eén and Sörensson, 2003] Les solveurs SAT modernes explorent l'espace de recherche

en affectant les variables et en construisant l'arbre de recherche correspondant. Conjointement, ils utilisent l'apprentissage de clauses afin de limiter la redondance dans l'exploration de l'arbre de recherche. L'adaptativité est très présente dans ces solveurs et a fortement contribué à leur essor. Elle est particulièrement constatée à travers les heuristiques de choix de variables qui sont guidées par les conflits. Elle est aussi mise en relief par les retours en arrière non chronologiques qui utilisent les informations des clauses apprises pour déterminer la profondeur à laquelle retourner dans l'arbre de recherche en cas d'échec. En outre, ils exploitent les techniques de redémarrage en conservant à chaque relance certaines informations apprises. Ces techniques jouent d'ailleurs un rôle primordial dans l'orientation de la recherche aux sous-espaces les plus prometteurs.

4.2.2 BTD : obstacles et motivations

Nous avons rappelé dans la partie précédente quelques techniques adaptatives parmi les plus connues. Les heuristiques de choix de variables et les techniques de redémarrage adaptatives amassent des informations tout au long de la résolution afin de conserver l'historique de celle-ci. La variable suivante à instancier ou la fréquence des redémarrages sont ainsi choisies en se basant sur l'état courant et les états précédents de la recherche. L'apport des techniques adaptatives est tellement impressionnant que leur exploitation par BTD demeure incontournable. Or, vu l'emploi d'une décomposition, bénéficier pleinement de certaines techniques adaptatives semble impossible. En particulier, l'intérêt des heuristiques de choix de variables adaptatives se trouve plus ou moins limité selon la décomposition utilisée. Dans cette partie, nous nous focalisons sur les heuristiques de choix de variables et nous nous intéressons aux limitations qui y sont imposées par l'utilisation de la décomposition arborescente.

4.2.2.1 Ordre de choix de variables sous BTD

Pendant la résolution, BTD exploite une décomposition calculée en amont de celle-ci. Une décomposition donnée de largeur w^+ impose un ordre partiel (ou même total parfois) pour le choix de variables afin de garantir une complexité temporelle en $O(exp(w^+))$.

Pour préciser l'ordre de choix de variables Λ exploité, nous utilisons la notation suivante :

- $\Lambda = [x_1, x_2, \ldots, x_p]$ est un ordre total pour le choix de variables sur p variables qui signifie que la variable x_1 apparaît nécessairement avant x_2 qui apparaît à son tour avant x_3 et ainsi de suite (une séquence de variables). Autrement dit, $x_1 \preceq_X x_2 \preceq_X \cdots \preceq_X x_p$.

- $\Lambda = \{x_1, x_2, \ldots, x_p\}$ est un ordre partiel pour le choix de variables sur p variables qui signifie que les variables $x_1, x_2, \ldots, x_p$ peuvent apparaître dans n'importe quel ordre (un ensemble de variables).

- Soit Λ_1 un ordre sur p_1 variables, Λ_2 un ordre sur p_2 variables, $\ldots$, et Λ_q un ordre sur p_q variables tel que $\Lambda_i, i \in \{1, \ldots, q\}$ peut être un ordre partiel ou total et $\Lambda_i \cap \Lambda_j = \emptyset$ si $i \neq j$. $\Lambda = [\Lambda_1, \Lambda_2, \ldots, \Lambda_q]$ est un ordre sur $p_1 + p_2 + \cdots + p_q$ variables qui signifie que les variables de Λ_1 doivent forcément apparaître avant toutes celles de Λ_2 qui doivent forcément apparaître avant toutes celles de Λ_3 et ainsi de suite.

- Soit Λ_1 un ordre sur p_1 variables, Λ_2 un ordre sur p_2 variables, $\ldots$, et Λ_q un ordre sur p_q variables tel que $\Lambda_i, i \in \{1, \ldots, q\}$ peut être un ordre partiel ou total et $\Lambda_i \cap \Lambda_j = \emptyset$ si $i \neq j$. $\Lambda = \{\Lambda_1, \Lambda_2, \ldots, \Lambda_q\}$ est un ordre sur $p_1 + p_2 + \cdots + p_q$ variables qui signifie

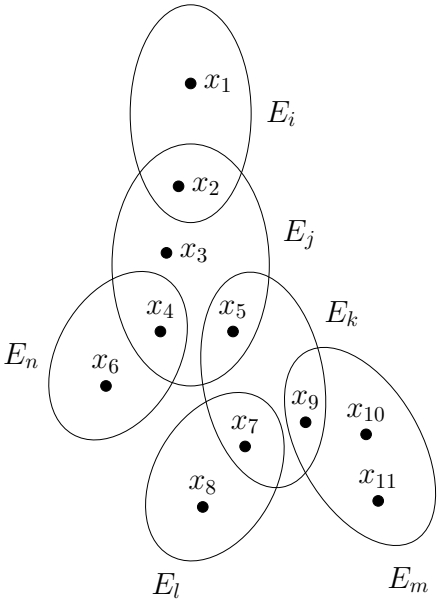


FIGURE 4.1 – La répartition des variables dans les clusters d'une décomposition.

que quel que soit Λ_i et Λ_j de Λ, les variables de Λ_i peuvent apparaître avant toutes celles de Λ_j et vice versa.

Il est à noter que si $\Lambda = \{x_i\}$ ou que $\Lambda = [x_i]$, Λ peut être noté tout simplement x_i. Par exemple, si $\Lambda = [x_1, \{x_2, x_3\}]$ alors x_1 doit nécessairement apparaître avant x_2 et x_3 tandis que x_2 et x_3 peuvent apparaître dans n'importe quel ordre. Les deux ordres totaux possibles sont alors $[x_1, x_2, x_3]$ ou $[x_1, x_3, x_2]$. Aussi, si $\Lambda = [x_1, \{x_2, \{x_3, x_4\}\}]$ alors les ordres totaux possibles sont : $[x_1, x_2, x_3, x_4]$, $[x_1, x_2, x_4, x_3]$, $[x_1, x_3, x_4, x_2]$ ou $[x_1, x_4, x_3, x_2]$.

Les algorithmes de résolution n'exploitant pas une décomposition bénéficient d'une liberté totale quant au choix de la variable suivante à instancier. Au contraire, les méthodes structurelles comme BTD subissent une restriction de la liberté de l'heuristique de choix de variables (cf. la partie 2.2.5.1). L'ordre de choix de variables induit par la décomposition (E, T) découle de l'ordre de visite des clusters qui à son tour est partiellement imposé par la décomposition. En effet, cet ordre est compatible avec la décomposition s'il peut être produit par un parcours en profondeur de l'arbre correspondant T à partir du cluster racine E_r. Étant donné un ordre sur les clusters $<$, un ordre de choix de variables compatible est alors exploité par BTD. Plus précisément, $\forall x \in E_i, \forall y \in E_j$, avec $E_i < E_j$, x doit forcément apparaître avant y dans Λ, c'est-à-dire $\Lambda = [\ldots, x, \ldots, y, \ldots]$ ou $x \preceq_X y$. Dans le but de respecter un ordre de visite des clusters compatible avec un parcours en profondeur de l'arbre T, la liberté de l'heuristique de choix de variables se limite à :

- Choisir librement la variable suivante parmi les variables propres (variables appartenant à un cluster mais pas à son parent) du cluster courant E_i. Toutefois, à chaque fois que le cluster E_i est revisité l'ordre d'instanciation de ses variables propres peut changer.

- Choisir librement un prochain cluster parmi les clusters fils de E_i une fois le cluster E_i entièrement instancié. De même, l'ordre de visite des clusters fils de E_i peut changer à chaque fois que E_i est entièrement instancié.

La figure 4.1 montre la répartition des variables dans les clusters d'une décomposition. Il s'agit d'une décomposition formée de 6 clusters $E = \{E_i, E_j, \ldots, E_n\}$ d'une instance

ayant 11 variables. Supposons que le cluster racine $E_r = E_i$. L'ordre partiel de choix de variables imposé par la décomposition est :
$\Lambda = [\{x_1, x_2\}, \{x_3, x_4, x_5\}, \{x_6, [\{x_7, x_9\}, \{x_8, \{x_{10}, x_{11}\}\}]\}]$. En effet, le cluster E_i est le premier à être instancié, c'est-à-dire les variables x_1 et x_2, selon l'ordre choisi par l'heuristique de choix de variables, suivi par le cluster E_j incluant les variables propres x_3, x_4 et x_5. Puis, un choix entre les deux clusters fils E_k et E_n est réalisé. Lorsque le cluster E_k est choisi et les variables x_7 et x_9 instanciées, de nouveau, un choix est fait entre les clusters E_l et E_m.

Dans [Jégou and Terrioux, 2014a], BTD acquiert un niveau additionnel de liberté (cf. la partie 2.2.5.1). En effet, lors d'un redémarrage, BTD a l'opportunité de choisir un nouveau cluster racine. Le fait de changer le cluster racine résulte en un changement de l'ordre partiel induit par la décomposition employée. Supposons que le nouveau cluster racine de la décomposition dans la figure 4.1 est $E_r = E_j$. Le nouvel ordre partiel induit par la décomposition est alors : $\Lambda = [\{x_2, x_3, x_4, x_5\}, \{x_1, x_6, [\{x_7, x_9\}, \{x_8, \{x_{10}, x_{11}\}\}]\}]$. Le fait de changer le cluster racine et ainsi l'ordre imposé par la décomposition permet d'exploiter potentiellement un ordre plus pertinent à l'égard de la résolution. En particulier, le choix du nouveau cluster racine peut contribuer à exploiter plus tôt certaines variables jugées intéressantes pour la résolution.

Bien que BTD dispose ainsi d'une liberté plus grande pour le choix de la prochaine variable, l'exploitation de la décomposition demeure très contraignante vis-à-vis de la résolution :

- La même décomposition est utilisée tout au long de la résolution. En effet, l'ensemble E de clusters de la décomposition reste inchangé pendant la résolution même si l'ordre exploité sur ces clusters peut éventuellement changer.
- Les clusters de E sont calculés uniquement sur la base de critères structurels avant le début de la résolution. Les ordres pouvant être induits par la décomposition ne sont pas étudiés au préalable. En outre, la compatibilité entre ces ordres et un ordre souhaité à l'égard de la résolution n'est jamais évoquée. En tous cas, beaucoup de caractéristiques de l'instance (comme la difficulté à satisfaire les contraintes ou l'impact des variables) ne sont dévoilées que pendant la résolution.
- Toutes ces contraintes imposées sur l'heuristique de choix de variables empêchent BTD de profiter pleinement des heuristiques de choix de variables adaptatives qui puisent leur intérêt dans leur capacité à offrir un choix plus libre pour la prochaine variable selon les critères définis par cette heuristique.
- Les contraintes imposées sur l'heuristique de choix de variables pourront également empêcher BTD de bénéficier de tous les avantages des redémarrages. En effet, lors d'un redémarrage certaines variables peuvent être jugées très importantes. Un bon algorithme de recherche souhaiterait ainsi les affecter le plus tôt possible dans l'arbre de recherche. Malheureusement, l'emploi d'une décomposition arborescente peut empêcher une telle affectation.

Tous ces éléments pourront détériorer significativement l'efficacité de la résolution.

4.2.2.2 Vers une exploitation plus opportune de la décomposition

L'idée que nous proposons dans ce chapitre s'inspire des travaux réalisés dans [Jégou et al., 2007]. Plus précisément, ces travaux peuvent être considérés comme étant les fondements théoriques de notre travail. Le but de [Jégou et al., 2007] est de trouver un compromis entre les bonnes bornes théoriques induites par l'utilisation des décompositions et

la nécessité absolue d'exploiter des heuristiques de choix de variables efficaces en pratique. Les auteurs proposent une extension de BTD, appelé BDH (pour *Backtracking on Dynamic covering by acyclic Hypergraphs*). Elle est basée sur une exploitation dynamique du recouvrement par un hypergraphe acyclique (CAH). L'approche vise à intégrer des heuristiques de choix de variables plus dynamiques, un critère jugé essentiel pour une résolution efficace en pratique. Ils se focalisent sur des recouvrements acycliques déduits d'un recouvrement dit *de référence*. À partir de ce recouvrement *de référence*, une grande variété de classes d'hypergraphes acycliques peut être définie. Ces classes peuvent être définies selon des critères liés à la nature du recouvrement ou aux relations existantes avec la méthode de résolution. Nous pouvons par exemple citer le recouvrement visant à borner la valeur de certains paramètres comme la largeur ou la taille des séparateurs, la préservation des séparateurs du recouvrement de référence ou la capacité d'implémenter des heuristiques efficaces comme celles qui sont dynamiques. En particulier, les auteurs mettent en avant la classe qui se focalise sur le concept de séparateur. Ce choix est simplement justifié par l'intérêt de ce paramètre vis-à-vis de la complexité spatiale. Ainsi, les recouvrements acycliques qui peuvent être exploités en pratique peuvent être déduits du recouvrement *de référence* en se limitant à un sous-ensemble des séparateurs de ce dernier. En d'autres termes, aucun nouveau séparateur ne peut effectivement être créé puisque ces recouvrements consistent à mettre en commun plusieurs hyperarêtes. Cette extension de BTD a essentiellement permis de proposer un grand nombre d'heuristiques de choix de variables du fait qu'elle se libère de la structure initiale. En effet, BDH n'a pas besoin de déterminer une décomposition unique à exploiter tout au long de la résolution. Au contraire, pendant la recherche, l'hypergraphe courant peut être modifié dans le but de prendre en compte l'évolution du problème. En plus, BDH est capable d'exploiter tous les (no)goods structurels enregistrés pour tous les séparateurs du recouvrement *de référence* incluant des séparateurs qui sont actuellement dans de plus grandes hyperarêtes du recouvrement courant exploité. Les résultats reportés dans [Jégou et al., 2007] se situent principalement sur un plan théorique notamment en ce qui concerne les changements induits vis-à-vis de la complexité temporelle.

Ce que nous proposons dans ce chapitre est d'offrir aux méthodes structurelles telles que BTD l'opportunité d'exploiter - dans le même esprit que BDH - différemment les décompositions. Notre objectif consiste à alléger les contraintes qui handicapent l'heuristique de choix de variables et qui sont à l'origine de la non-compétitivité des méthodes structurelles par rapport aux méthodes non structurelles dans la plupart des cas. Cependant, nous avons choisi de changer dynamiquement la décomposition employée pendant la résolution de façon à suivre ce qui serait souhaitable à l'égard de l'heuristique de choix de variables. En d'autres termes, l'extension de BTD que nous proposons se distingue par des fusions déclenchées dynamiquement sur la base d'informations relatives à la résolution en cours alors que BDH se contente de fusionner des clusters en ne faisant pas croître la taille des clusters au-delà d'une certaine limite fixée au préalable. La démarche que nous proposons s'inscrit dans la lignée des méthodes adaptatives parce qu'elle vise à adapter la décomposition, au sens de l'ensemble de clusters qui la constituent, au contexte de la résolution en se basant sur des informations concernant les états précédents considérés durant la résolution de l'instance ainsi que son état courant.

Par la suite, la section 4.3 présente une exploitation dynamique de la décomposition dans le cadre du problème CSP.

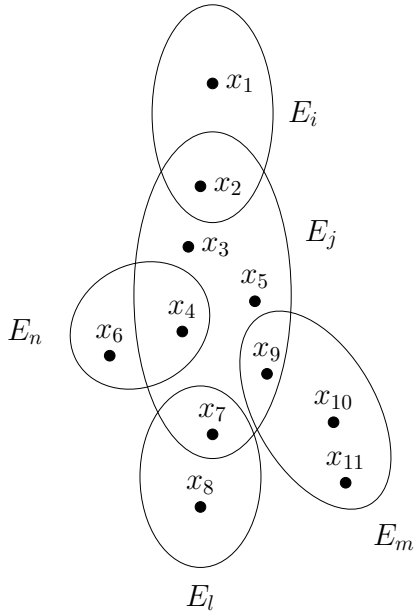


FIGURE 4.2 – Ensemble de clusters de la décomposition de la figure 4.1 après la fusion de E_j et de E_k.

4.3 Modification dynamique de la décomposition via la fusion pour le problème CSP : BTD-MAC+RST+Fusion

Nous visons à offrir à BTD plus de liberté quant à l'heuristique de choix de variables et à ne pas se contenter du niveau de liberté actuellement existant. La thèse que nous défendons ici est que la dynamicité de la décomposition i.e. sa modification pendant la résolution, permet d'adapter la décomposition à la nature de l'instance à résoudre. Par sa modification, nous visons la modification de l'ensemble de clusters E exploités. L'opération permettant de changer la décomposition dans ce contexte est la *fusion* des clusters. Dans ce qui suit, nous proposons l'algorithme $BTD\text{-}MAC{+}RST{+}Fusion$ permettant d'intégrer la modification dynamique de la décomposition via la *fusion* des clusters. Ces travaux ont fait l'objet des publications [Jégou et al., 2016b,c].

L'algorithme $BTD\text{-}MAC{+}RST{+}Fusion$ (voir l'algorithme 4.2) représente une adaptation de l'algorithme $BTD\text{-}MAC{+}RST$ [Jégou and Terrioux, 2014a] (cf. la partie 2.2.5.1) afin de prendre en compte la fusion dynamique. Pour ces deux algorithmes, la décomposition arborescente est calculée en amont de la résolution. Comme décrit dans la section 4.2.2, l'emploi de cette décomposition enracinée en un cluster E_r induit un ordre partiel sur les variables. La différence principale entre eux réside dans le fait que $BTD\text{-}MAC{+}RST$ utilise cette décomposition initiale durant toute la résolution (au sens de l'ensemble des clusters qui la définissent puisque la racine peut changer), tandis que $BTD\text{-}MAC{+}RST{+}Fusion$ va la faire évoluer dynamiquement. Ainsi, l'ordre partiel imposé par la décomposition change pendant la résolution.

Nous donnons tout d'abord des détails plus amples sur la fusion dynamique.

4.3.1 Fusion dynamique

La fusion consiste à mettre en commun les variables de deux clusters voisins pour former ainsi un seul cluster.

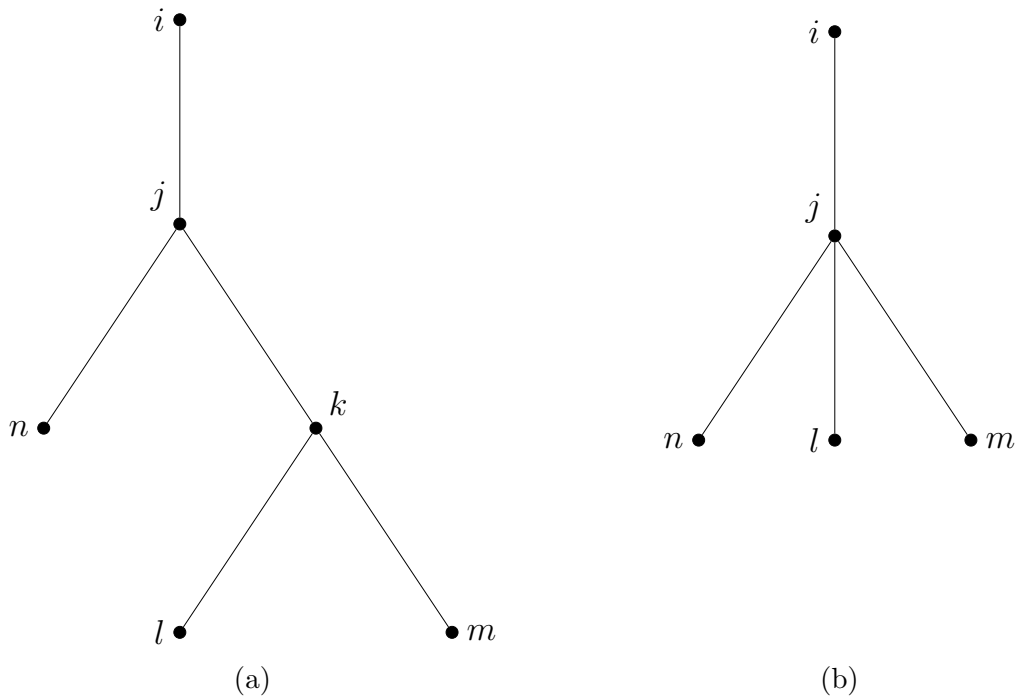


FIGURE 4.3 – L'arbre correspondant à la décomposition avant la fusion (a) et après la fusion (b).

La figure 4.2 montre la fusion des clusters E_j et E_k de la décomposition de la figure 4.1. Le cluster résultant de la fusion du cluster E_j et du cluster E_k est un nouveau cluster E_j. Il est à noter que, les fils du cluster fusionné deviennent les fils du cluster résultant de la fusion. Par exemple, dans la figure 4.1, E_l et E_m, les fils de E_k, deviennent les fils de E_j qui est le cluster résultant de la fusion des clusters E_j et E_k dans la figure 4.2. La figure 4.3 montre à son tour l'arbre T correspondant aux décompositions avant et après la réalisation de la fusion. Soit $\mathcal{D}$ la décomposition initiale et $\mathcal{D}'$ la décomposition après la fusion. Pour le même cluster racine, tout ordre sur les variables permis par $\mathcal{D}$ reste permis par $\mathcal{D}'$. Cependant, en exploitant $\mathcal{D}'$ nous pouvons obtenir plus d'ordres possibles qu'en exploitant $\mathcal{D}$. Nous en déduisons que la fusion préserve les ordres de choix de variables initialement permis tout en offrant plus de liberté. Par exemple, pour $E_r = E_i$, si l'ordre partiel induit par la décomposition dans la figure 4.1 est : $[\{x_1, x_2\}, \{x_3, x_4, x_5\}, \{x_6, [\{x_7, x_9\}, \{x_8, \{x_{10}, x_{11}\}\}]\}]$, l'ordre partiel induit par la décomposition dans la figure 4.2 est : $[\{x_1, x_2\}, \{x_3, x_4, x_5, x_7, x_9\}, \{x_6, x_8, \{x_{10}, x_{11}\}\}]$. Ainsi, l'ordre $[x_1, x_2, x_7, x_9, x_3, x_4, x_5, x_6, x_8, x_{10}, x_{11}]$ est par exemple désormais permis par la décomposition de la figure 4.2 alors qu'il ne l'était pas auparavant par la décomposition de la figure 4.1. Le choix de fusionner ou non des clusters est conditionné par des informations apprises durant la résolution. Aussi, le comportement de $BTD\text{-}MAC{+}RST{+}Fusion$ se situe entre celui de $BTD\text{-}MAC{+}RST$ avec une liberté partielle pour l'ordre sur les variables (si aucune fusion n'est effectuée) et celui de $MAC{+}RST$ avec une liberté totale (si, après une série de fusions, la décomposition ne contient plus qu'un seul cluster). L'intérêt de ce nouvel algorithme est de pouvoir trouver dynamiquement le bon compromis à partir d'informations recueillies durant la résolution. En d'autres termes, son but consiste à exploiter les informations fournies durant la résolution pour adapter en permanence la décomposition et ainsi la liberté de choix de variables au contexte courant de la résolution.

La différence primordiale entre la fusion dite *statique* [Jégou et al., 2005] et la fusion

dynamique réside dans la raison du déclenchement de la fusion des clusters. La fusion dynamique est réalisée pendant la résolution en se basant sur des informations récoltées depuis le début de la résolution jusqu'au moment de la fusion. La fusion statique est faite au préalable de la résolution. Elle se fait habituellement sur la base de critères purement structurels avec tous les inconvénients que cela peut comporter. Par exemple, le but de la fusion statique pourrait consister à limiter la taille des séparateurs en supprimant ceux dans la taille dépasse la limite choisie via la fusion des deux clusters impliqués. D'autres objectifs peuvent être visés comme la création de clusters connexes ou l'augmentation du nombre de variables propres de chaque cluster.

4.3.2 Description de l'algorithme BTD-MAC+RST+Fusion

4.3.2.1 Similitudes avec BTD-MAC+RST

L'algorithme $BTD\text{-}MAC{+}RST{+}Fusion$ exploite l'algorithme $BTD\text{-}MAC{+}Fusion$ (voir l'algorithme 4.1). Ce dernier ne se différencie de $BTD\text{-}MAC$ (cf. la partie 2.2.5.1) que par les lignes 17 à 21. Initialement, la suite de décisions Σ ainsi que les ensembles de goods G^d et de nogoods N^d sont vides. $BTD\text{-}MAC{+}RST{+}Fusion$ (à l'instar de $BTD{+}MAC{+}RST$) commence la résolution en assignant les variables du cluster E_r avant de passer à un cluster fils. En exploitant le nouveau cluster E_i, seules les variables non assignées du cluster E_i seront instanciées. En d'autres termes, seules les variables de E_i n'appartenant pas à $E_i \cap E_{p(i)}$ sont instanciées. Pour résoudre chaque cluster les deux algorithmes s'appuient sur MAC (lignes 24-29 et 35-37). Durant la résolution MAC peut prendre deux types de décisions :

- Décisions positives $x_i = v_i$ qui assignent la valeur v_i à la variable x_i (ligne 27),
- Décisions négatives $x_i \neq v_i$ qui assurent que v_i ne peut être assignée à x_i (ligne 35).

Supposons que $\Sigma =< \delta_1, \ldots, \delta_i >$ est la suite de décisions courante où chaque δ_j peut être une décision soit positive, soit négative. Une nouvelle décision positive $x_{i+1} = v_{i+1}$ est prise et un filtrage par cohérence d'arc (AC) est accompli (ligne 27). Si aucune incohérence n'est détectée, la recherche continue normalement (ligne 28). Sinon la valeur v_{i+1} est supprimée de $D_{x_{i+1}}$ et un nouveau filtrage par AC est effectué (ligne 35). Si une incohérence est détectée, nous procédons à un retour-arrière et la dernière décision positive $x_l = v_l$ est changée en $x_l \neq v_l$. Lorsque le cluster E_i est choisi comme cluster suivant, nous savons que la prochaine décision positive implique une variable de E_i. Étant donné que le séparateur $E_i \cap E_{p(i)}$ est instancié, le filtrage par AC impacte uniquement les variables des clusters de $Desc(E_i)$. Une fois le cluster E_i complètement instancié de façon cohérente (ligne 1), chaque sous-problème enraciné en un cluster E_j, fils de E_i, sera résolu (ligne 11). Plus précisément, pour un cluster fils E_j et une suite de décisions Σ, nous résolvons le problème enraciné en E_j et induit par $Pos(\Sigma)[E_i \cap E_j]$ où $Pos(\Sigma)[E_i \cap E_j]$ est l'ensemble des décisions positives impliquant les variables de $E_i \cap E_j$ dans Σ. Si nous trouvons une extension cohérente de Σ sur $Desc(E_j)$, $Pos(\Sigma)[E_i \cap E_j]$ est enregistré comme *good structurel* (ligne 12-13). Si, au contraire, la résolution montre qu'il n'existe aucune extension cohérente de Σ sur $Desc(E_j)$, $Pos(\Sigma)[E_i \cap E_j]$ est enregistré comme un *nogood structurel* (lignes 15-16). Ces (no)good structurels sont utilisés ultérieurement dans la recherche pour éviter certaines redondances (lignes 7-8 et 10). Si un redémarrage est déclenché (ligne 31), la résolution est interrompue. La gestion des redémarrages est faite comme dans [Jégou and Terrioux, 2014a] et s'accompagne de l'enregistrement de *nld-nogoods réduits* [Lecoutre et al., 2007e] qui permettent de ne pas réexplorer des parties de l'espace de recherche déjà

Algorithme 4.1 : BTD-MAC+Fusion $(P, \Sigma, E_i, V_{E_i}, G^d, N^d)$

Entrées-Sorties : $P = (X, D, C)$: une instance CSP
Entrées : Σ : suite de décisions ; E_i : cluster ; V_{E_i} : ensemble de variables
Entrées-Sorties : G^d : ensemble de goods ; N^d : ensemble de nogoods
Sorties : *vrai* si une solution au sous-problème de P enraciné en E_i et induit par Σ a été trouvée, $faux$ s'il est prouvé qu'il n'en possède pas, *inconnu* sinon

```
1  si V_Ei = ∅ alors
2      résultat ← vrai
3      Q_Ei ← Fils(E_i)                    /* Fils(E_i) : ensemble des clusters fils de E_i */
4      tant que résultat ∉ {faux, inconnu} et Q_Ei ≠ ∅ faire
5          Choisir un cluster E_j ∈ Q_Ei
6          Q_Ei ← Q_Ei \ {E_j}
7          si Pos(Σ)[E_i ∩ E_j] est un nogood dans N^d alors
8              résultat ← faux
9          sinon
10             si Pos(Σ)[E_i ∩ E_j] n'est pas un good de E_i par rapport à E_j dans G^d alors
11                 résultat ← BTD-MAC+Fusion(P,Σ,E_j, E_j\(E_i ∩ E_j),G^d,N^d)
12                 si résultat = vrai alors
13                     Enregistrer Pos(Σ)[E_i ∩ E_j] comme good de E_i par rapport à E_j dans G^d
14                 sinon
15                     si résultat = faux alors
16                         Enregistrer Pos(Σ)[E_i ∩ E_j] comme nogood de E_i par rapport à E_j dans N^d
17                     sinon
18                         si fusion alors Fusionner E_j avec un de ses fils
19                         si non redémarrage alors
20                             Q_Ei ← Q_Ei ∪ {E_j}
21                             résultat ← vrai
22     retourner résultat
23 sinon
24     Choisir une variable x ∈ V_Ei
25     Choisir une valeur v ∈ D_x
26     D_x ← D_x \ {v}
27     si AC (P,Σ ∪ ⟨x = v⟩) alors
28         résultat ← BTD-MAC+Fusion(P,Σ∪ ⟨x = v⟩,E_i, V_Ei\{x},G^d,N^d)
29     sinon résultat ← faux
30     si résultat = faux alors
31         si redémarrage ou fusion alors
32             Enregistrer nld-nogoods par rapport à la suite de décisions Σ[E_i]
33             retourner inconnu
34         sinon
35             si AC (P,Σ ∪ ⟨x ≠ v⟩) alors
36                 retourner BTD-MAC+Fusion(P,Σ ∪ ⟨x ≠ v⟩,E_i, V_Ei,G^d,N^d)
37             sinon retourner faux
38     sinon  retourner résultat
```

visitées. L'intérêt du redémarrage réside dans l'exploitation des connaissances acquises auparavant via les (no)good structurels et les nld-nogoods réduits [Lecoutre et al., 2007e]. Le déclenchement d'un redémarrage peut être conditionné par des paramètres globaux

Algorithme 4.2 : BTD-MAC+RST+Fusion (P)

Entrées : $P = (X, D, C)$: une instance CSP
Sorties : *vrai* si P possède une solution, $faux$ sinon

```
1 G^d ← ∅ ; N^d ← ∅
2 répéter
3 |   Choisir un cluster racine E_r
4 |   résultat ← BTD-MAC+Fusion (P,∅,E_r,E_r,G^d,N^d)
5 jusqu'à résultat ≠ inconnu
6 retourner résultat
```

(portant sur l'ensemble du problème) ou locaux (relatifs au cluster courant) ou aussi une combinaison des deux. Pour plus de détails sur les redémarrages sous BTD, nous invitons les lecteurs à se référer à la partie 2.2.5.1 ou aux travaux cités ci-dessus.

4.3.2.2 Modifications réalisées pour BTD-MAC+RST+Fusion

Les lignes 17-21 concernent uniquement la fusion dynamique donc $BTD\text{-}MAC{+}Fusion$. La fusion dynamique vise, comme expliqué ci-dessus, à donner plus de liberté à l'heuristique de choix de variables, et en même temps, à limiter l'impact des défauts potentiels mis en avant dans la partie 4.2.2. Le choix de fusionner ou non des clusters va se faire sur la base des critères s'appuyant sur l'état courant du problème, mais aussi sur tout ou partie des états précédemment rencontrés durant la résolution. Ce choix est implémenté via la fonction $fusion$ qui retourne *vrai* si une fusion est nécessaire et $faux$ sinon. Cette fonction peut être librement implémentée par l'utilisateur qui pourrait intégrer les critères qu'il souhaite faire entrer en jeu dans la décision de la fusion ou de la non-fusion des clusters. Si aucune fusion n'est requise ($fusion$ renvoie $faux$), la résolution se poursuit normalement. Au contraire, si ce critère considère que fusionner le cluster courant avec un de ses fils permettrait de rendre la suite de la résolution plus efficace (par exemple en permettant d'instancier plus tôt certaines variables jouant un rôle clé), $fusion$ renvoie *vrai* et $BTD\text{-}MAC{+}Fusion$ va modifier la décomposition courante en fusionnant ces deux clusters (ligne 18). La figure 4.4 montre l'affectation des variables du cluster E_j (les variables affectées sont colorées en rouge). Supposons, par exemple, qu'après l'affectation de la variable x_4, la fonction $fusion$ retourne *vrai* pour le cluster E_k. La décomposition va alors être modifiée et les clusters E_j et E_k fusionnés. Pour cela, nous commençons par désaffecter les variables instanciées du cluster courant et par enregistrer des nld-nogoods réduits (ligne 32) comme le ferait un redémarrage classique. Dans la figure 4.5, les variables x_3 et x_4 sont alors désaffectées et la recherche retourne au cluster E_i. Une fois revenu dans le cluster parent, nous procédons à la fusion. À ce stade (ligne 19), soit nous continuons à revenir en arrière si *redémarrage* est vrai, soit la recherche se poursuit avec l'exploration d'un fils du cluster parent. Par exemple, la figure 4.6 montre que $BTD\text{-}MAC{+}Fusion$ explore le nouveau cluster E_j et est désormais capable d'affecter d'abord les variables x_7 et x_9. Notons que désaffecter les variables du cluster courant avant de le fusionner avec un de ses fils n'est pas une nécessité. Toutefois, ce choix devrait permettre d'exploiter au plus tôt les variables nouvellement ajoutées dans ce cluster.

L'algorithme $BTD\text{-}MAC{+}Fusion$ est paramétrable notamment par l'heuristique de fusion (nous en proposons une dans la partie expérimentale). Un bon choix pour cette heuristique devrait permettre d'améliorer considérablement la résolution en la rendant plus efficace.

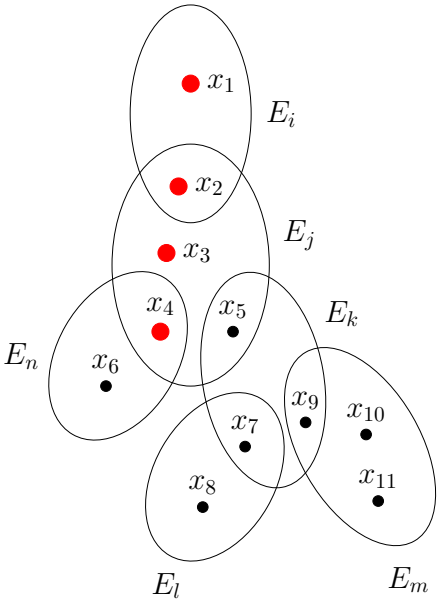


FIGURE 4.4 – Illustration de l'affectation du cluster E_j.

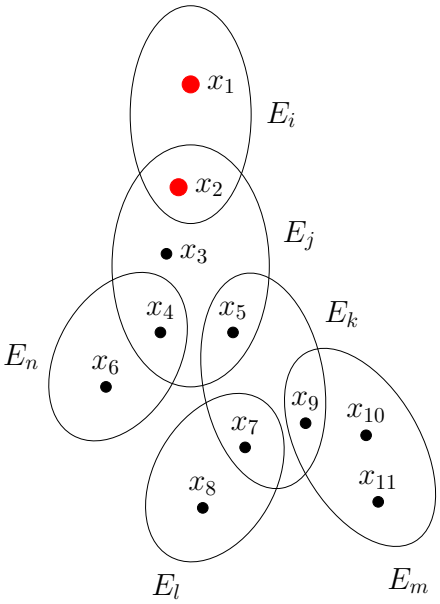


FIGURE 4.5 – Illustration du retour-arrière vers le cluster E_i.

4.3.3 Fondements théoriques

Nous montrons dans cette partie la validité de notre approche. Tout d'abord, nous prouvons que le fait de fusionner deux clusters ne remet pas en cause la validité des (no)goods structurels et des nld-nogoods.

Propriété 5 *Soit (E', T') la décomposition arborescente d'un graphe G obtenue à partir de la décomposition (E, T) de G en fusionnant le cluster E_y avec le cluster E_x (avec E_y fils de E_x dans (E, T)).*
Les (no)good structurels de tout cluster E_i par rapport à un cluster fils E_j (avec $E_j \neq E_y$) et les nld-nogoods réduits enregistrés vis-à-vis de (E, T) restent valides vis-à-vis de (E', T'),

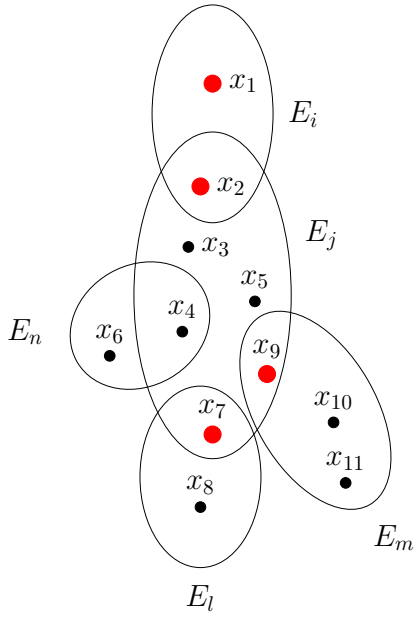


FIGURE 4.6 – Illustration de l'affectation du nouveau cluster E_j.

c'est-à-dire que leur utilisation lors de l'exploitation de (E', T') *ne nuit pas à la validité de BTD-MAC+RST+Fusion.*

Preuve : Soit Δ un good structurel de E_i par rapport à son fils E_j enregistré pour (E, T). Sachant que E_j et E_y sont différents, le sous-problème de P enraciné en E_j dans (E, T) est identique au sous-problème de P enraciné en E_j dans (E', T'). Ainsi, si Δ peut être étendu d'une façon cohérente sur le premier sous-problème, il peut également l'être sur le deuxième sous-problème. En conséquence, Δ est un good structurel valide de E_i par rapport à E_j dans (E', T'). Le raisonnement est similaire pour un nogood structurel.

Un nld-nogood réduit est un nogood quelle que soit la décomposition considérée. Nous devons uniquement vérifier qu'un nld-nogood Δ est valide pour la décomposition (E', T'). En d'autres termes, nous devons vérifier qu'il existe un cluster de E' incluant toutes les variables de Δ. Par construction, il existe nécessairement un cluster E_p de (E, T) couvrant Δ. Si $E_p \neq E_x$ et $E_p \neq E_y$, alors $E_p \in E'$. Sinon, après fusion, nous avons $E_p \subset E_x$ et $E_x \in E'$. Par conséquent, dans les deux cas, les variables de Δ sont toutes recouvertes par un seul cluster de E' et ainsi Δ est valide pour (E', T').□

Nous prouvons maintenant la validité de notre algorithme.

Théorème 9 *BTD-MAC+RST+Fusion est correct, complet et termine.*

Preuve : Considérons premièrement *BTD-MAC+Fusion* qui diffère de *BTD-MAC* par l'exploitation de la fusion. Supposons que nous obtenons (E', T') à partir de la décomposition (E, T) après la fusion de d'un cluster E_i et de son fils E_j. Soit Σ_f la suite de décisions pour laquelle la fonction *fusion* renvoie *vrai* lors de l'affectation des variables de E_i. Certains nld-nogoods réduits sont ainsi enregistrés et la recherche opère un retour-arrière au cluster $E_{p(i)}$ parent du cluster courant E_i. Ainsi, nous obtenons la suite Σ'_f qui correspond à la suite de décisions Σ_f restreinte aux variables des clusters présents dans la branche allant du cluster racine E_r au cluster $E_{p(i)}$. *BTD-MAC+Fusion* continue sa recherche à partir de $E_{p(i)}$ avec Σ'_f en exploitant la décomposition (E', T'). Le cluster

résultant de la fusion (le nouveau cluster E_i) peut éventuellement être le cluster suivant visité ou être visité ultérieurement.

Nous constatons que :

- L'arbre de recherche exploré par $BTD\text{-}MAC{+}Fusion$ depuis son premier appel avec une suite de décisions vide jusqu'à la suite de décisions Σ_f est le même que celui développé par $BTD\text{-}MAC$ dans les mêmes circonstances pour la décomposition (E, T).
- En plus, après la fusion, l'arbre de recherche développé par $BTD\text{-}MAC{+}Fusion$ depuis la suite de décisions Σ'_f jusqu'à sa terminaison est identique à celui développé par $BTD\text{-}MAC$ dans les mêmes circonstances pour la décomposition (E', T').

Nous savons que $BTD\text{-}MAC$ est complet (si aucun redémarrage n'a eu lieu), correct et termine [Jégou and Terrioux, 2014a]. En outre, selon la proposition 5, les (no)goods structurels et les nld-nogoods réduits enregistrés pour (E, T) restent valides pour la nouvelle décomposition (E', T'). Ainsi, la correction, la terminaison et la complétude de l'algorithme ne sont pas mises en danger. Quant aux redémarrages, l'enregistrement de nld-nogoods réduits à chaque redémarrage empêche l'exploration d'une partie de l'espace de recherche déjà explorée. Ainsi, $BTD\text{-}MAC{+}Fusion$ est complet (si aucun redémarrage n'a eu lieu), correct et termine.

En plus, lorsque plusieurs opérations de fusion sont réalisées, le même raisonnement peut être appliqué pour chaque fusion en partitionnant l'arbre de recherche.

Les redémarrages arrêtent la recherche sans changer le fait que si une solution existait dans l'espace de recherche visité par $BTD\text{-}MAC{+}Fusion$ alors $BTD\text{-}MAC{+}Fusion$ l'aurait trouvée. Comme $BTD\text{-}MAC{+}RST{+}Fusion$ fait seulement plusieurs appels à $BTD\text{-}MAC{+}Fusion$, il est correct.

L'algorithme $BTD\text{-}MAC{+}RST{+}Fusion$ est complet, correct et termine :

- En ce qui concerne la complétude, si l'appel à $BTD\text{-}MAC{+}Fusion$ n'est pas arrêté par un redémarrage (ce qui est nécessairement le cas du dernier appel à $BTD\text{-}MAC{+}Fusion$ si $BTD\text{-}MAC{+}RST{+}Fusion$ termine), la complétude de $BTD\text{-}MAC{+}Fusion$ implique celle de $BTD\text{-}MAC{+}RST{+}Fusion$.
- En outre, l'enregistrement des nld-nogoods réduits à chaque redémarrage évite l'exploration d'une partie de l'espace de recherche déjà explorée par un appel antérieur à $BTD\text{-}MAC{+}Fusion$. Il en découle que, suite aux appels successifs à l'algorithme $BTD\text{-}MAC{+}Fusion$, l'exploration de l'espace de recherche se fait sur une partie de plus en plus strictement réduite. Ainsi, la terminaison et la complétude de $BTD\text{-}MAC{+}RST{+}Fusion$ sont garanties par l'enregistrement non limité des nogoods réalisé suite aux différents appels à $BTD\text{-}MAC{+}Fusion$ et aussi grâce à la terminaison et la complétude de $BTD\text{-}MAC{+}Fusion$.

□

Nous nous intéressons maintenant aux complexités en temps et en espace et nous énonçons le théorème suivant :

Théorème 10 *L'algorithme $BTD\text{-}MAC{+}RST{+}Fusion$ a une complexité en temps en $O(R.((n.s^2.m.\log(d) + w'^{+}.|N_r|).d^{w'^{+}+2} + n.(w'^{+})^2.d))$ et une complexité en espace en $O(n.s.d^s + w'^{+}.(d + |N_r|))$ avec w'^{+} la largeur de la décomposition arborescente finale, s la taille de la plus grande intersection $E_i \cap E_j$ de la décomposition initiale, R le nombre de redémarrages et $|N_r|$ le nombre de nld-nogoods réduits mémorisés.*

Preuve : L'algorithme $BTD\text{-}MAC{+}RST$ a une complexité en temps en $O(((n.s^2.m.\log(d)+ w^+.|N_r|).d^{w^+ +2} + n.(w^+)^2.d).R)$ et une complexité spatiale en $O(n.s.d^s + w^+.(d + |N_r|))$ [Jégou and Terrioux, 2014a] (cf. la partie 2.2.5.1). En ce qui concerne l'algorithme $BTD\text{-}MAC{+}RST{+}Fusion$, l'application des opérations de fusion implique que la taille des clusters peut éventuellement augmenter. Ainsi, les complexités théoriques sont exprimées en fonction de w'^+ au lieu de w^+. Les opérations de fusion ne créent pas de nouveaux séparateurs, mais au contraire, en suppriment quelques-uns. Ainsi, la taille maximale des séparateurs de la décomposition initiale représente une borne supérieure sur la taille des séparateurs utilisés durant la recherche. C'est pourquoi, les éléments des complexités spatiales et temporelles relatifs à la taille des séparateurs ne sont pas modifiés. Pour ce qui est des nld-nogoods réduits enregistrés après une opération de fusion, bien qu'ils induisent des coûts additionnels en espace et en temps, ces coûts sont déjà pris en compte par les coûts des nld-nogoods réduits enregistrés lors des redémarrages. En conséquence, la complexité en temps est de $O(R.((n.s^2.m.\log(d) + w'^+.|N_r|).d^{w'^+ +2} + n.(w'^+)^2.d))$ et celle en espace est de $O(n.s.d^s + w'^+.(d + |N_r|))$.□

Notons qu'il est possible de limiter l'augmentation de la largeur de la décomposition finale par rapport à la décomposition initiale en utilisant une heuristique de fusion prenant en compte ce paramètre.

4.4 Étude expérimentale

Dans cette section, nous évaluons l'intérêt pratique des décompositions dynamiques. Nous présentons tout d'abord le protocole expérimental.

4.4.1 Protocole expérimental

Nous reprenons pour ces expérimentations le protocole expérimental de la partie 3.4.2 du chapitre 3. Au niveau des décompositions exploitées, nous retenons $Min\text{-}Fill$, H_2, H_3 et H_5 qui a montré plus d'intérêt que H_4 dans le cadre de l'exploitation dynamique de la décomposition. Pour H_5, la décomposition est exploitée sans limite sur la taille des séparateurs pour laisser plus de latitude à la fusion. Ainsi, tout séparateur trouvé lors du calcul des clusters de H_5 est conservé.

La décomposition dynamique exploite une heuristique de fusion. Cette dernière se base sur les conseils de l'heuristique de choix de variables pour évaluer le besoin de la fusion des clusters. Plus précisément, étant donné un cluster courant E_i, à chaque fois que nous choisissons la variable suivante à affecter dans E_i, on teste si l'heuristique de choix de variables aurait choisi une autre variable si elle avait l'opportunité de choisir parmi les variables non affectées de $(\bigcup_{E_j \in Fils(E_i)} E_j) \cup E_i$. Si une variable d'un fils E_j de E_i est préférée à une variable de E_i, un compteur relatif à E_j est incrémenté. Lorsque le compteur relatif à E_j atteint une limite L (à savoir 100 dans nos expérimentations), le cluster E_j est fusionné avec son parent E_i. Ainsi, les variables appartenant auparavant à E_j appartiennent désormais à E_i. Elles peuvent ainsi être assignées avant toute variable propre de E_i d'origine. La valeur de 100 utilisée dans les expérimentations n'est sûrement pas garantie d'être la valeur optimale pour toutes les instances, même pas pour une instance donnée. Il s'agit en revanche d'une valeur qui a permis une résolution efficace pour un grand nombre d'instances du benchmark considéré. Elle assure un compromis entre une valeur très élevée qui diminuerait fortement la fréquence des fusions en les rendant très contraintes et entre une valeur très basse qui induirait une fusion de E_i avec E_j sans que

Algorithme	*Min-Fill*		H_2		H_3		H_5^∞	
	#rés	temps	#rés	temps	#rés	temps	#rés	temps
`BTD-MAC`	1 348	34 416	1 424	21 860	1 487	28 792	1 430	23 408
`BTD-MAC+RST`	1 507	33 632	1 536	22 854	1 558	26 418	1 538	25 649
`BTD-MAC+Fusion`	1 497	32 206	1 533	25 475	1 547	28 630	1 550	22 828
`BTD-MAC+RST+Fusion`	1 558	35 444	1 572	27 291	1 584	27 809	1 592	27 185

TABLE 4.1 – Nombre d'instances résolues et temps de résolution pour *BTD-MAC*, *BTD-MAC+RST*, *BTD-MAC+Fusion* et *BTD-MAC+RST+Fusion* selon les décompositions exploitées.

la nécessité de cette opération ne soit suffisamment constatée.

4.4.2 Observations et analyse des résultats

Idée générale sur l'intérêt de la fusion dynamique Le tableau 4.1 fournit le nombre d'instances résolues et le temps de résolution pour chaque algorithme selon chaque décomposition considérée. La comparaison des décompositions avec *BTD-MAC* et *BTD-MAC+RST* a été déjà faite dans la partie 3.4.2 du chapitre 3. Elle a montré l'intérêt des décompositions H_2, H_3 et H_5 vis-à-vis de *Min-Fill*, notamment celui de H_5 qui limite la taille des séparateurs. Nous considérons cependant maintenant une version de H_5 qui ne limite pas la taille des séparateurs puisque cela est plus intéressant pour l'exploitation dynamique de la décomposition même si son utilisation est moins efficace que H_3 avec *BTD-MAC* et *BTD-MAC+RST*. Elle est notée H_5^∞.

Nous constatons que quelle que soit la décomposition utilisée *BTD-MAC+Fusion* résout plus d'instances que *BTD-MAC*. En occurrence, *BTD-MAC+Fusion* résout 149 instances de plus avec *Min-Fill* par rapport à *BTD-MAC*, 109 instances de plus avec H_2, 60 instances de plus avec H_3 et finalement 120 instances de plus avec H_5^∞. La figure 4.7

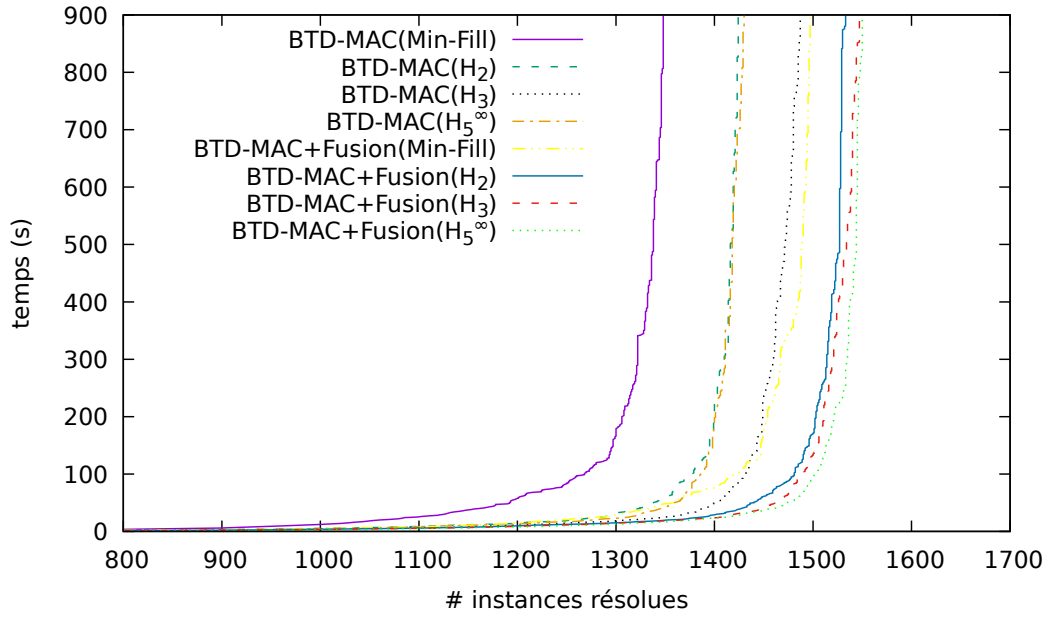


FIGURE 4.7 – Le nombre cumulé d'instances résolues pour les algorithmes *BTD-MAC* et *BTD-MAC+Fusion* selon la décomposition utilisée.

montre le nombre cumulé d'instances résolues pour $BTD\text{-}MAC$ et $BTD\text{-}MAC{+}Fusion$ avec les décompositions $Min\text{-}Fill$, H_2, H_3 et H_5^∞. La figure montre clairement l'amélioration apportée par $BTD\text{-}MAC{+}Fusion$ par rapport à $BTD\text{-}MAC$. Ceci reste valide avec l'exploitation des redémarrages. La figure 4.8 montre également l'intérêt de la fusion dyna-

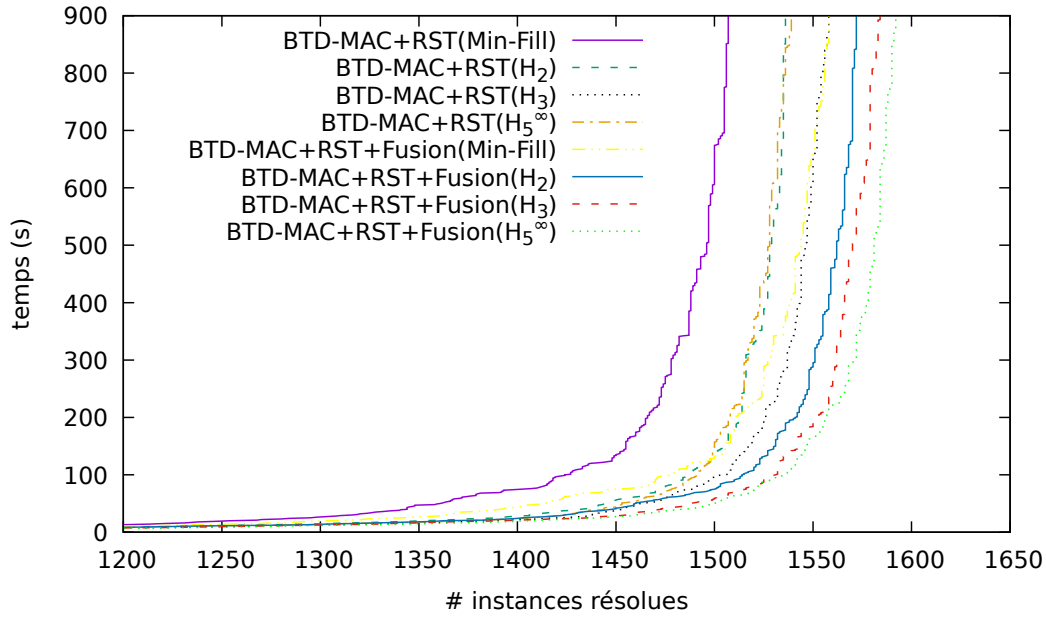


FIGURE 4.8 – Le nombre cumulé d'instances résolues pour $BTD\text{-}MAC{+}RST$ et $BTD\text{-}MAC{+}RST{+}Fusion$ selon la décomposition utilisée.

mique utilisée conjointement avec les redémarrages. En effet, $BTD\text{-}MAC{+}RST{+}Fusion$ résout 51 instances de plus avec $Min\text{-}Fill$ par rapport à $BTD\text{-}MAC{+}RST$, 36 instances de plus avec H_2, 26 instances de plus avec H_3 et 54 instances en plus avec H_5^∞. Les temps de résolution sont soit comparables, soit meilleurs avec la fusion dynamique que sans la fusion dynamique. D'ailleurs, même lorsque le nombre d'instances résolues augmentent considérablement, le temps de résolution cumulé correspondant à la fusion dynamique reste comparable à celui de l'algorithme opérant sans fusion. Il peut même diminuer comme dans le cas de $Min\text{-}Fill$ (34 416 s avec $BTD\text{-}MAC$ et 32 206 s avec $BTD\text{-}MAC{+}Fusion$). L'exploitation de la fusion dynamique avec la décomposition H_5^∞ permet de résoudre le plus grand nombre d'instances sans et avec redémarrage. Vu que $Min\text{-}Fill$ est considéré comme étant l'heuristique de l'état de l'art et comme $BTD\text{-}MAC{+}RST$ est une méthode référence parmi les méthodes structurelles, nous comparons dans la figure 4.9 les deux combinaisons $BTD\text{-}MAC{+}RST$ avec $Min\text{-}Fill$ et $BTD\text{-}MAC{+}RST{+}Fusion$ avec H_5^∞. La figure montre que pour la plupart d'instances du benchmark, soit l'instance est résolue nettement plus rapidement avec $BTD\text{-}MAC{+}RST{+}Fusion$ qu'avec $BTD\text{-}MAC{+}RST$, soit les temps de résolution sont comparables pour les deux méthodes. L'intérêt de l'exploitation dynamique de la décomposition par rapport à son exploitation statique est ainsi mis en évidence.

Apport de la fusion dynamique selon la décomposition ou l'algorithme utilisé L'apport de la fusion dynamique semble d'autant plus important que la qualité de la décomposition vis-à-vis de la résolution est notoirement moins bonne. Par exemple, $BTD\text{-}MAC{+}Fusion$ améliore $BTD\text{-}MAC$ d'une façon plus remarquable avec $Min\text{-}Fill$

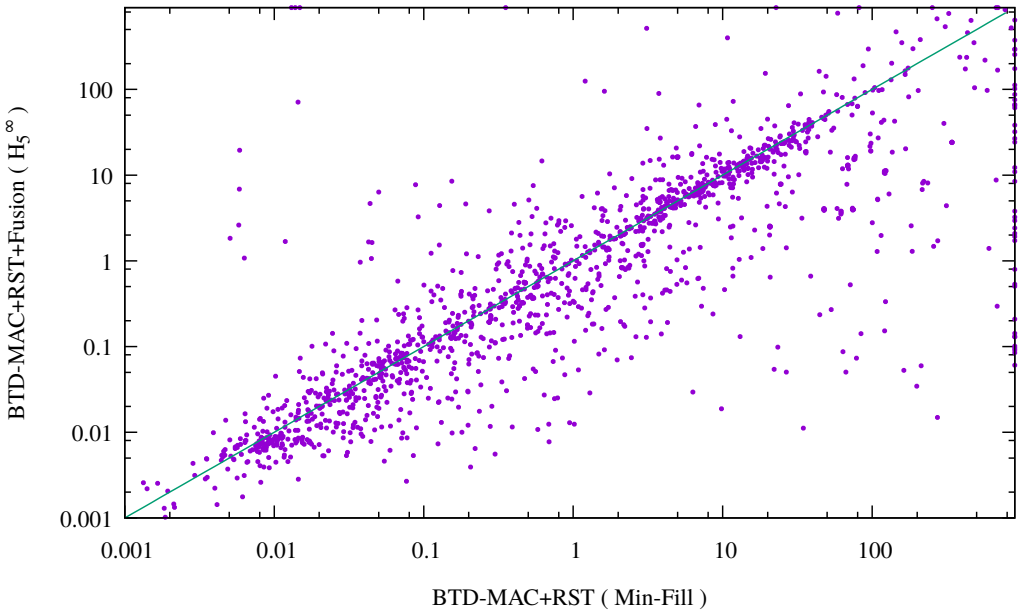


FIGURE 4.9 – Comparaison des temps d'exécution de $BTD\text{-}MAC{+}RST$ avec $Min\text{-}Fill$ et de $BTD\text{-}MAC{+}RST{+}Fusion$ avec H_5^∞.

qu'avec une autre décomposition comme H_3. La raison principale est que l'ordre de choix de variables induit par $Min\text{-}Fill$ est souvent plus contraignant que celui pouvant être induit par une décomposition telle que H_3. Ainsi, l'exploitation de la fusion dynamique peut plus facilement améliorer $BTD\text{-}MAC$ lorsque $Min\text{-}Fill$ est employée que lors de l'utilisation de H_3. En plus, l'amélioration produite par la fusion est plus perceptible lorsque les redémarrages ne sont pas exploitées. En effet, l'exploitation des redémarrages compense en partie les limitations imposées par la décomposition sur l'heuristique de choix de variables, ce qui augmente l'efficacité de $BTD\text{-}MAC{+}RST$ par rapport à $BTD\text{-}MAC$. Plus précisément, lorsque $BTD\text{-}MAC{+}RST$ fait un redémarrage un nouveau cluster racine est choisi. Cela peut être vu comme une forme de dynamicité pour la décomposition. Le même raisonnement réalisé pour la comparaison entre l'apport de la fusion dynamique avec $Min\text{-}Fill$ et celui avec H_3 peut être appliqué pour la comparaison entre l'apport de la fusion dynamique avec redémarrages et celui sans redémarrage. Nous pouvons remarquer aussi que $BTD\text{-}MAC{+}Fusion$ et $BTD\text{-}MAC{+}RST$ se rapprochent vis-à-vis du nombre d'instances résolues ou du temps de résolution. En plus, l'exploitation des redémarrages conjointement avec la décomposition dynamique est vraiment pertinente vu que $BTD\text{-}MAC{+}RST{+}Fusion$ surpasse $BTD\text{-}MAC{+}RST$ et $BTD\text{-}MAC{+}Fusion$.

Comparaison des temps cumulés d'exécution sur les instances résolues par toutes les combinaisons algorithme/décomposition Concernant les temps de résolution, pour une comparaison plus équitable, nous considérons dans le tableau 4.2, les 1 232 instances résolues par tous les algorithmes. Encore une fois, nous obtenons les meilleurs temps de résolution avec H_5^∞ et les pires avec $Min\text{-}Fill$. Plus précisément, le meilleur temps d'exécution cumulé est réalisé par H_5^∞ avec $BTD\text{-}MAC{+}RST$, $BTD\text{-}MAC{+}Fusion$ ou $BTD\text{-}MAC{+}RST{+}Fusion$ (la différence entre les trois méthodes est négligeable) tandis que le pire est celui réalisé par le couple $BTD\text{-}MAC$ et $Min\text{-}Fill$. L'exploitation de la fusion et/ou des redémarrages améliore le temps d'exécution cumulé

Algorithme	*Min-Fill*	H_2	H_3	H_5^∞
BTD-MAC	25 141	12 277	11 371	9 993
BTD-MAC+RST	16 042	10 157	9 478	9 088
BTD-MAC+Fusion	16 579	10 153	9 763	9 039
BTD-MAC+RST+Fusion	16 151	10 166	9 794	9 197

TABLE 4.2 – Le temps de résolution de *BTD-MAC*, *BTD-MAC+RST*, *BTD-MAC+Fusion* et *BTD-MAC+RST+Fusion* selon les décompositions exploitées pour les 1 232 instances résolues par tous les algorithmes.

Algorithme	*Min-Fill*$^{5\%}$		$H_2^{5\%}$		$H_3^{5\%}$		$H_5^{5\%}$	
	#rés	temps	#rés	temps	#rés	temps	#rés	temps
BTD-MAC	1 483	32 998	1 514	25 413	1 519	27 251	1 524	26 143
BTD-MAC+RST	1 561	32 640	1 578	28 396	1 576	24 993	1 586	24 520

TABLE 4.3 – Nombre d'instances résolues et temps de résolution pour *BTD-MAC* et *BTD-MAC+RST* selon les décompositions exploitées avec une fusion statique limitant la taille des séparateurs à 5% du nombre de variables du problème (taille limitée entre 4 et 50).

pouvant être obtenu avec *Min-Fill* tout en restant supérieur aux temps réalisés sous d'autres décompositions.

Fusion dynamique vs fusion statique Nous comparons maintenant la fusion dynamique avec le concept de la fusion statique proposé dans [Jégou et al., 2005], qui peut être réalisée après le calcul d'une décomposition arborescente indépendamment de l'algorithme utilisé pour la calculer (par exemple, *Min-Fill*, H_2, H_3 ou même H_5). La fusion statique effectue des fusions en se basant sur la taille des séparateurs qui est limitée à 5% de n. D'après le tableau 4.3, nous pouvons noter que, concernant le nombre d'instances résolues, *BTD-MAC+Fusion* est significativement meilleur que *BTD-MAC* tandis que *BTD-MAC+RST+Fusion* et *BTD-MAC+RST* sont plus comparables. Plus précisément, *BTD-MAC+RST+Fusion* est plus efficace pour H_3 et H_5 que *BTD-MAC+RST* et est légèrement moins efficace que *BTD-MAC+RST* pour *Min-Fill* et H_2. Ainsi, la fusion dynamique permet d'obtenir de résultats nettement meilleurs par rapport à la fusion statique si les redémarrages ne sont pas exploités et des résultats comparables si les redémarrages sont utilisés. Au-delà, en fusionnant les clusters pendant la résolution, nous adaptons la décomposition selon des connaissances sémantiques de l'instance tandis que la fusion statique se base uniquement sur des critères structurels et requiert de choisir une limite convenable sur la taille des séparateurs ce qui pourrait être particulièrement difficile.
En effet, cette taille dépend en grande partie de l'instance considérée.

Finalement, nous retenons le couple *BTD-MAC+RST+Fusion* et H_5^∞ qui permet d'obtenir les meilleurs résultats en termes du nombre d'instances résolues parmi toutes les combinaisons d'un algorithme de résolution et d'une méthode de calcul de décomposition arborescente.

BTD-MAC+RST+Fusion vs MAC+RST Nous comparons à présent l'algorithme *BTD-MAC+RST+Fusion* contre *MAC+RST* vis-à-vis de l'efficacité de la résolution. La figure 4.10 représente le nombre cumulé d'instances résolues pour l'algorithme *BTD-*

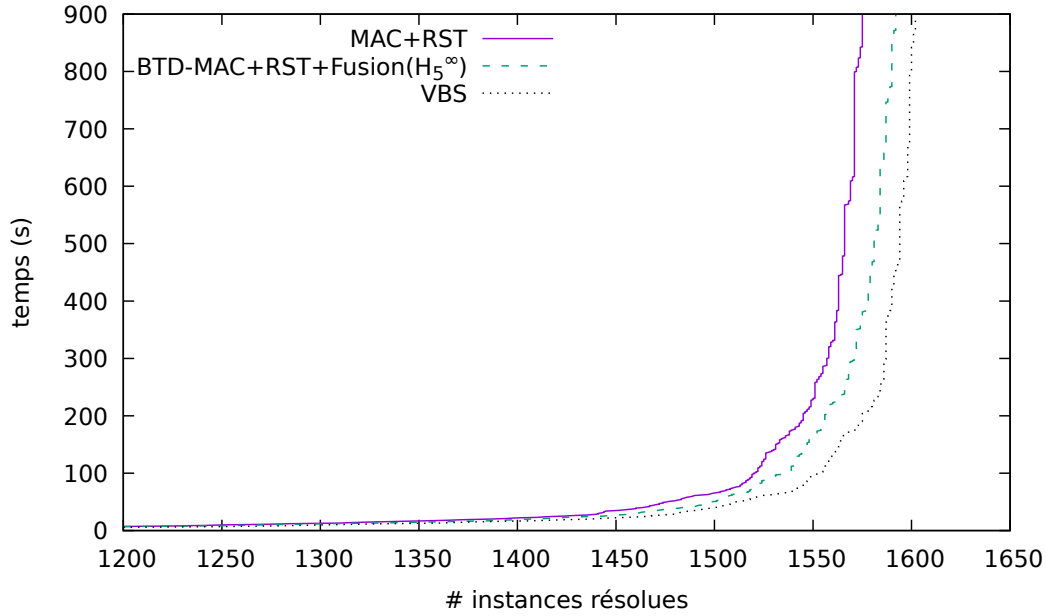


FIGURE 4.10 – Le nombre cumulé d'instances résolues pour $BTD\text{-}MAC{+}RST{+}Fusion$ avec H_5^∞, $MAC{+}RST$ et VBS.

$MAC{+}RST{+}Fusion$, $MAC{+}RST$ et VBS (pour Virtual Best Solver parmi les deux algorithmes). Premièrement, l'algorithme $BTD\text{-}MAC{+}RST{+}Fusion$ résout plus d'instances que $MAC{+}RST$ (1 592 instances contre 1 575). Aussi, nous pouvons noter que le comportement de $BTD\text{-}MAC{+}RST{+}Fusion$ est plus proche de celui du VBS que $MAC{+}RST$, ce qui montre clairement que $BTD\text{-}MAC{+}RST{+}Fusion$ a une meilleure performance que $MAC{+}RST$. Notons finalement que sur ce benchmark, dans 46% des cas, $BTD\text{-}MAC{+}RST{+}Fusion$ ne réalise aucune fusion tandis que dans 6% des cas il fusionne au contraire tous les clusters de la décomposition conduisant à une décomposition à un seul cluster.

Nous focalisons maintenant nos observations sur les instances les plus difficiles. Parmi les 1 859 instances considérées, quelques unes d'entre elles sont facilement résolues par $MAC{+}RST$ (par exemple 286 instances sont résolues sans aucun retour-arrière). L'exploitation des méthodes structurelles comme BTD ou ses variantes pour résoudre de telles instances n'est pas nécessairement pertinente. Ainsi, nous nous basons ici sur le nombre de nœuds développés par $MAC{+}RST$ comme critère de difficulté. Une instance est considérée comme difficile si le nombre de nœuds développés par $MAC{+}RST$ est plus grand que $100 \times n$. Ce faisant, nous disposons de 675 instances considérées comme difficiles. La figure 4.11 fournit la comparaison de temps de résolution pour $BTD\text{-}MAC{+}RST{+}Fusion$ et $MAC{+}RST$ sur ces instances. Globalement, nous observons que les deux algorithmes $BTD\text{-}MAC{+}RST{+}Fusion$ et $MAC{+}RST$ ont un comportement similaire sur une grande partie de ces instances. En effet, pour 66% des instances, la différence du temps de résolution entre les deux méthodes est inférieure à 10%. Cependant, pour le reste des instances, $BTD\text{-}MAC{+}RST{+}Fusion$ est généralement plus rapide que $MAC{+}RST$. D'ailleurs, $BTD\text{-}MAC{+}RST{+}Fusion$ cumule un temps de résolution de 18 741 s sur ces instances tandis que $MAC{+}RST$ requiert 35 116 s. Pour 16% des instances, $BTD\text{-}MAC{+}RST{+}Fusion$ est au moins 10 fois plus rapide que $MAC{+}RST$ tandis que ce dernier est 10 fois plus rapide pour seulement 1% des instances. Finalement, l'exploita-

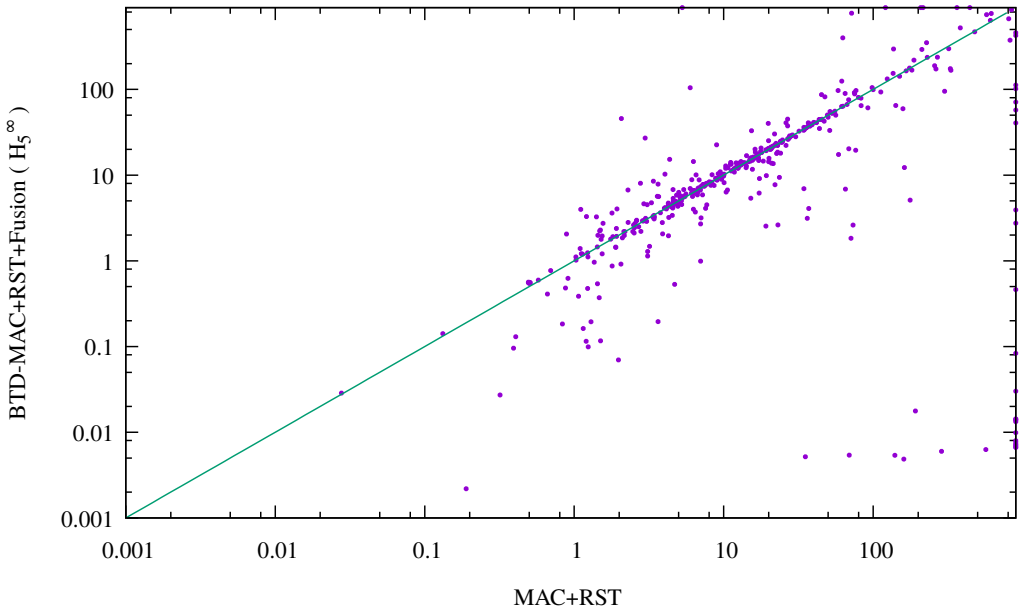


FIGURE 4.11 – La comparaison des temps de résolution de $BTD\text{-}MAC{+}RST{+}Fusion$ et de $MAC{+}RST$ pour les 675 instances.

tion de la structure joue un rôle central. En effet, nous pouvons noter que 100% (respectivement 67%) des instances non résolues par $MAC{+}RST$ mais résolues par $BTD\text{-}MAC{+}RST{+}Fusion$ ont un ratio $n/(w^+ + 1)$ plus grand que 2 (respectivement 5).

Récemment, nous avons participé à la compétition XCSP3 2017 [XC1, 2017]. Nous avons participé au track standard (solveur BTD) et au track mini-solver (solveur miniBTD) pour la résolution séquentielle d'instances CSP. Le temps de résolution par instance est limité à 40 minutes et l'utilisation mémoire à 15500 Mio. Nous exploitons la décomposition $H_5\text{-}TD$, l'heuristique $dom/wdeg$, des redémarrages géométriques avec un seuil initial de 100 retours en arrière et un facteur de 1.1. Le premier cluster racine choisi est celui ayant le ratio maximum du nombre de contraintes par rapport à sa taille moins un. Ensuite, lors de chaque redémarrage, le cluster racine choisi est celui qui maximise la somme des poids des contraintes dont la portée intersecte ce dernier. Finalement, la cohérence d'arc est maintenue en prétraitement via $AC\text{-}3^{rm}$ et pendant la résolution via $AC\text{-}8^{rm}$. Les instances choisies pour le track standard contiennent des contraintes globales non manipulées par notre solveur. Nous avons tout de même réussi à résoudre 47% des 510 instances considérées et 57% des instances résolues par le VBS. Au niveau du track mini-solver, nous avons occupé le deuxième rang et nous avons pu résoudre 67% des 242 instances considérées et 86% des instances du VBS. Le solveur qui a occupé le premier rang résout 95% des instances du VBS.

Bilan En conclusion, nous avons montré l'intérêt pratique de l'exploitation dynamique de la décomposition vis-à-vis de son exploitation statique standard. Nous avons aussi montré que l'exploitation simultanée des redémarrages et de la fusion dynamique augmente l'efficacité des algorithmes bénéficiant des redémarrages. Ainsi, l'intérêt de la fusion dynamique s'ajoute à celui des redémarrages. En outre, nous avons comparé la fusion dynamique à la fusion statique réalisée en amont de la résolution une fois la décomposition calculée. La fusion dynamique met en avant la pertinence des critères sémantiques qu'elle

intègre par rapport à la fusion statique qui se base uniquement sur des critères structurels. Cet aspect est notamment souligné lorsque les redémarrages ne sont pas exploités. En effet, ces techniques compensent dans certains cas l'absence de la fusion dynamique. Finalement, nous avons comparé $BTD\text{-}MAC{+}RST{+}Fusion$ avec H_5^∞, qui permet d'obtenir les meilleurs résultats, à $MAC{+}RST$. $BTD\text{-}MAC{+}RST{+}Fusion$ s'est principalement démarqué sur les instances difficiles. D'ailleurs, il ne faut pas oublier que l'exploitation de la fusion dynamique met en place des techniques sophistiquées dont l'utilisation n'a pas de sens dans le cas où les instances sont faciles, voire triviales.

4.5 Conclusion

Les décompositions arborescentes ont déjà été exploitées avec succès pour résoudre des instances du problème CSP. Pour autant, leur potentiel n'a pas été pleinement exploité. L'amélioration de la qualité des décompositions calculées grâce au cadre général de calcul de décomposition $H\text{-}TD$ (cf. chapitre 3) a permis d'augmenter l'efficacité de la résolution. En effet, ce cadre offre l'opportunité d'intégrer à la décomposition calculée plusieurs critères qui semblent avoir plus d'intérêt à l'égard de la résolution que le paramètre classique qui est la largeur de la décomposition. Ainsi, l'exploitation de ce cadre a permis de calculer, par exemple, des décompositions ayant des clusters connexes, des clusters avec plusieurs fils et aussi des décompositions avec des séparateurs de taille bornée. Ce dernier semble surpasser par son intérêt les autres paramètres vis-à-vis de la résolution des instances CSP. Cependant, cette décomposition est calculée en amont de la résolution et sera utilisée tout au long de la résolution. Elle est calculée en se basant uniquement sur des critères structurels ce qui ne garantit pas que cette décomposition soit la plus appropriée à l'égard du contexte de la résolution. En effet, cette décomposition ne permet pas d'intégrer des éléments relevant de la sémantique du problème.

Quant aux méthodes non structurelles comme MAC, elles profitent librement des approches adaptatives. Ces approches sont connues pour être capables de s'adapter au contexte de la résolution en faisant des choix qui prennent en considération l'état actuel de la résolution mais aussi ses états précédents. En intégrant ces approches, les méthodes de résolution ont témoigné une avancée remarquable. Ainsi, les heuristiques de choix de variables utilisées sont désormais le plus souvent adaptatives. Par exemple, en utilisant l'heuristique de choix de variables $dom/wdeg$, la recherche est dirigée vers les parties les plus difficiles et les plus conflictuelles du problème. Dans le même esprit, la recherche a pu être guidée suivant l'impact de la variable ou son activité. Les méthodes de résolution comme MAC peuvent profiter pleinement de ces méthodes vu qu'il n'y a aucune restriction imposée sur l'heuristique de choix de variables. D'ailleurs, celle-ci peut potentiellement choisir consécutivement des variables qui ne semblent pas avoir de lien structurel et qui sont relativement éloignées dans l'hypergraphe de contraintes. Au contraire, les méthodes structurelles sont désavantagées sur ce point. En effet, la décomposition sur laquelle elle se base induit un ordre partiel de choix de variables. C'est ainsi que ces méthodes ne peuvent pas se laisser complètement guider par une heuristique adaptative tel que $dom/wdeg$. En conséquence, les méthodes à base d'une décomposition voient leur efficacité se dégrader par rapport aux méthodes non structurelles.

Nous avons alors proposé dans ce chapitre de relâcher les contraintes imposées par la décomposition pour permettre plus de liberté aux méthodes structurelles. L'idée est de pouvoir changer de décomposition (au sens de l'ensemble de clusters qui la constituent) pendant la résolution pour pouvoir, par voie de conséquence, changer l'ordre partiel imposé par la décomposition. Le changement de décomposition ne se fait pas d'une

manière aléatoire mais plutôt d'une façon adaptative. En effet, nous nous inspirons des approches adaptatives pour guider le changement de la décomposition selon le contexte de la résolution. La modification de la décomposition se fait en se basant sur des informations recueillies pendant la résolution dans le but d'adapter la décomposition au contexte de celle-ci. Ainsi, nous proposons de modifier la décomposition via *la fusion* des clusters. Deux clusters fusionnés offrent plus de liberté quant aux choix de l'heuristique de choix de variables. Cette fusion se fait grâce aux recommandations de l'heuristique de choix de variables. La fusion dynamique permet aussi à un algorithme tel que BTD de mieux s'adapter aux instances peu structurées qui induisent souvent une « mauvaise » décomposition. En effet, ces instances sont généralement mieux résolues par une approche qui a toute la liberté quant au choix de la prochaine variable. L'utilisation de la fusion dynamique convergerait vers une décomposition qui permettrait de choisir plus librement la prochaine variable voire vers une décomposition ne contenant qu'un seul cluster.

La fusion dynamique de la décomposition a permis d'augmenter l'efficacité pratique de la résolution par rapport à son utilisation statique en termes de nombre d'instances résolues et du temps de résolution. En particulier, l'exploitation de la fusion dynamique avec la décomposition H_5^∞ a sensiblement amélioré la performance de $BTD\text{-}MAC{+}RST$. Nous avons aussi montré l'intérêt de la fusion dynamique par rapport à la fusion statique qui se fait avant la résolution et uniquement à la base de critères structurels. Non seulement, la fusion dynamique a amélioré l'exploitation de la décomposition mais a permis aussi de mettre en avant la compétitivité des méthodes structurelles vis-à-vis des méthodes non structurelles, voire les surpasser dans certains cas notamment pour les instances difficiles.

Dans le chapitre suivant, nous nous intéressons à une autre forme de dynamicité appliquée à la résolution des instances WCSP.

Chapitre 5

Exploitation dynamique de la décomposition dans le cas du problème WCSP

Sommaire

5.1 Introduction

Nous nous sommes intéressés dans le chapitre précédent à l'amélioration de la prise en compte du contexte de la résolution tout en exploitant une décomposition arborescente pour résoudre des instances CSP. Nous avons montré qu'à travers la fusion dynamique des clusters, nous libérons davantage l'heuristique de choix de variables en nous laissant guider par cette dernière pour décider de la fusion ou non des clusters. Nous pouvons ainsi profiter plus intensément des approches adaptatives dont l'emploi est notoirement d'une très grande importance pour assurer une résolution efficace. La fusion dynamique a permis de modifier le comportement des algorithmes basés sur BTD en assurant un compromis entre un algorithme non structurel tel que MAC et un algorithme comme BTD. Plus précisément, selon les instances, le nouvel algorithme pourra avoir un comportement allant d'un simple $BTD\text{-}MAC$ jusqu'à MAC. Cette approche permettrait de faciliter l'emploi du solveur par des utilisateurs non experts. En effet, les solveurs auxquels nous aspirons ne requiert pas l'intervention de l'utilisateur pour sélectionner, par exemple, la décomposition convenable qui sera utilisée pendant la résolution. Cette tâche n'est pas aisée surtout que ce choix peut porter préjudice à l'efficacité du solveur. Notre approche va permettre d'avoir une décomposition choisie par défaut et qui va être remise en cause pendant la résolution. Le solveur peut éventuellement déduire que l'utilisation d'une décomposition n'est pas avantageuse. Dans le même esprit, nous proposons dans ce chapitre une approche différente basée également sur une exploitation dynamique de la décomposition pour la résolution d'instances WCSP. Elle vise à tirer profit à la fois des avantages des méthodes structurelles et des méthodes non structurelles. En d'autres termes, son objectif consiste à imiter les méthodes non structurelles au sens de la liberté du choix de la prochaine variable à affecter tout en bénéficiant des enregistrements réalisés par les méthodes structurelles. La méthode résultante pourrait avoir le comportement de l'un comme le comportement de l'autre ou aussi un comportement intermédiaire permettant de résoudre des instances ne pouvant être résolues ni par l'un, ni par l'autre. Afin de décider du comportement du nouvel algorithme, nous avons recours à une approche adaptative. Elle rassemble des constatations faites jusqu'à un instant t de la résolution et décide tout en se basant sur son état à cet instant du comportement ultérieur de l'algorithme. Nous proposons ainsi dans ce chapitre un nouvel algorithme basé sur $BTD\text{-}DFS$ (cf. la partie 2.4.4.2) qui exploite la décomposition d'une façon plus avancée vis-à-vis de son exploitation classique. Cet algorithme sera décrit en détails dans la section 5.3. Son intérêt pratique est mis en avant dans la section 5.4 avant de conclure.

Tout d'abord, nous nous focalisons dans la section suivante sur les objectifs de la gestion dynamique de la décomposition dans le cadre WCSP.

5.2 Adaptation au contexte de la résolution dans le cadre WCSP

Nous présentons dans cette partie les objectifs de l'utilisation dynamique de la décomposition pour la résolution d'instances WCSP.

Tout d'abord, l'utilisation dynamique de la décomposition pour la résolution d'instances WCSP rejoint le but de la fusion dynamique employée dans le cadre CSP qui consiste à améliorer l'exploitation des approches adaptatives dans les méthodes structurelles. D'ailleurs, l'emploi d'une décomposition arborescente pour la résolution des instances WCSP pose le même problème que son exploitation pour la résolution des instances CSP en termes de liberté de l'heuristique de choix de variables. En permettant des ordres

partiels non compatibles avec la décomposition d'origine, tout en étant moins contraignants, nous augmentons la chance de l'algorithme de trouver un ordre de choix de variables plus opportun à l'égard de l'instance à résoudre. Par exemple, en ayant le maximum de liberté, une heuristique de choix de variables telle que $dom/wdeg$ est plus susceptible de localiser plus rapidement les parties les plus difficiles de l'espace de recherche.

En plus de leur association avec les approches adaptatives, les algorithmes de résolution des instances WCSP ont connu une amélioration remarquable au niveau de leur stratégie de recherche. Plus précisément, avec $HBFS$, la combinaison des deux stratégies de recherche, à savoir le parcours en profondeur (DFS) et le parcours du meilleur d'abord (BFS) a amélioré significativement la résolution des problèmes d'optimisation sous contraintes [Allouche et al., 2015] (cf. la partie 2.4.4.1). Naturellement, si les limitations imposées par la décomposition au niveau de l'heuristique de choix de variables sont drastiques, l'efficacité pratique de cette hybridation peut s'en retrouver détériorée. Ainsi, relâcher les contraintes imposées par la décomposition offre l'opportunité d'avoir une plus grande sélection de variables lors du choix de la prochaine variable à instancier. Par conséquent, une variable de meilleure borne inférieure peut être trouvée lorsque BFS est exploité.

Ensuite, l'utilisation dynamique de la décomposition pour la résolution d'instances WCSP vise à capturer des instances résolues facilement, voire trivialement, par des méthodes non structurelles. En effet, selon la nature de l'instance à résoudre, il se peut que l'instance soit résolue en quelques fractions de secondes par une méthode comme DFS ou $HBFS$ tandis qu'elle nécessiterait des heures d'exécution avant d'être résolue par une méthode basée sur BTD. Pour y parvenir, la méthode proposée devrait pouvoir supprimer dans certains cas toute contrainte imposée sur l'heuristique de choix de variables afin de lui permettre d'acquérir une liberté totale.

L'exploitation dynamique de la décomposition vise également à permettre la résolution de certaines instances plus difficiles, habituellement résolues par une exploitation classique de la décomposition. Plus précisément, elle vise à exploiter les indépendances détectées entre les sous-problèmes ainsi que les enregistrements pouvant être réalisés. Afin d'atteindre cet objectif, la méthode proposée devrait pouvoir se comporter dans certains cas comme une méthode classique basée sur BTD.

La combinaison des avantages des méthodes structurelles et non structurelles a aussi pour objectif de résoudre de nouvelles instances en exploitant conjointement les avantages des méthodes non structurelles et ceux des méthodes structurelles. À cette fin, l'exploitation dynamique de la décomposition devrait permettre d'exploiter la décomposition d'une façon plus légère que son exploitation classique sans toutefois perdre tout l'intérêt des méthodes à base de décomposition.

En outre, au vu de l'efficacité des approches adaptatives et de la grande variété d'instances ciblées par l'exploitation dynamique de la décomposition, cette dernière vise à s'adapter à la nature de l'instance afin que la décomposition couramment employée soit la plus opportune possible. Il peut s'agir de la décomposition d'origine, d'une décomposition formée d'un seul cluster ou d'une décomposition intermédiaire. Son but est alors de se baser sur l'historique de l'instance et sur son état actuel afin de décider quelle décomposition utiliser.

Finalement, l'utilisation dynamique de la décomposition permet d'attribuer moins d'importance à la décomposition originale calculée. En effet, dans le chapitre 3, nous nous sommes intéressés à la qualité de la décomposition calculée vis-à-vis de la résolution. Nous avons aussi vu que selon la décomposition utilisée pendant la résolution, cette dernière peut être plus ou moins efficace. Permettre de se libérer de cette décomposition initialement calculée atténue les inconvénients de celle-ci s'il s'avère qu'elle est à l'origine de la

détérioration de l'efficacité de la résolution.

5.3 Exploitation de la décomposition « si besoin » pour le problème WCSP : BTD-DFS+DYN

Dans le même esprit des travaux réalisés pour le problème CSP dans le chapitre 4, nous visons dans cette partie à exploiter le concept de dynamicité de la décomposition pour le problème WCSP.

Comme évoqué dans la partie précédente, nous visons à profiter de l'efficacité des heuristiques de choix de variables adaptatives comme l'heuristique *dom/wdeg* et du progrès réalisé par les algorithmes de recherche à parcours hybride comme $HBFS$. Ainsi, nous tentons de libérer les méthodes exploitant une décomposition. En revanche, l'utilisation d'une décomposition peut être parfois très bénéfique pour la résolution des instances WCSP. En particulier, l'enregistrement des informations au niveau des séparateurs entre clusters permet d'élaguer des parties de l'espace de recherche et de rendre la résolution plus efficace. Ces enregistrements permettent de mémoriser, pour chaque sous-problème considéré, les meilleures bornes inférieures et supérieures calculées, ou mieux encore, son optimum.

La question qui se pose légitimement est alors de savoir comment exploiter conjointement les avancées dont ont bénéficié les algorithmes de recherche au niveau des heuristiques de choix de variables et du type de parcours pour le problème WCSP et les avantages offerts par la décomposition arborescente. Afin de concilier ces deux points de vue, nous proposons dans cette partie d'exploiter la décomposition dynamiquement d'une façon plus flexible et plus appropriée et adaptée au vu du contexte courant de la résolution. Cela permet de s'adapter progressivement à la nature de l'instance à résoudre. La forme de dynamicité que nous proposons dans ce cadre consiste à utiliser la décomposition correspondant à un sous-problème uniquement lorsque la recherche sans décomposition semble stagner. La décomposition arborescente est alors exploitée d'une façon plus adéquate en évitant de l'utiliser systématiquement que ce soit globalement sur tout le problème ou localement sur un sous-problème. Ainsi, nous proposons dans ce qui suit l'algorithme $BTD\text{-}DFS{+}DYN$ qui intègre ce type de fonctionnement.

Il est très important de noter qu'au vu de l'efficacité pratique de $BTD\text{-}HBFS$ par rapport à celle de $BTD\text{-}DFS$ [Allouche et al., 2015], nous sommes naturellement plus intéréssés par $BTD\text{-}HBFS{+}DYN$ que par $BTD\text{-}DFS{+}DYN$. Toutefois, nous présentons dans cette partie $BTD\text{-}DFS{+}DYN$ par souci de simplicité. Cependant, l'extension de $BTD\text{-}HBFS$ peut être déduite sans difficulté de celle de $BTD\text{-}DFS$.

L'algorithme $BTD\text{-}DFS{+}DYN$ (voir l'algorithme 5.1) se base sur l'algorithme $BTD\text{-}DFS$. Les modifications apportées représentent une adaptation de l'algorithme afin de prendre en compte l'exploitation dynamique de la décomposition. Les deux algorithmes se basent sur une décomposition arborescente calculée en amont de la résolution et qui est enracinée en un cluster E_r. En ce qui concerne $BTD\text{-}DFS$, la décomposition induit classiquement un ordre partiel sur les variables comme tous les algorithmes basés sur BTD et comme nous l'avons déjà vu dans la partie 4.2.2. Cependant, $BTD\text{-}DFS{+}DYN$ exploite la décomposition différemment.

5.3.1 Décomposition si besoin

$BTD\text{-}DFS{+}DYN$ a pour objectif d'exploiter la décomposition à bon escient, c'est-à-dire quand le besoin s'en fait sentir. Plus précisément, la décomposition calculée initialement ne sera exploitée, pour un sous-problème donné, que si la résolution sans décom-

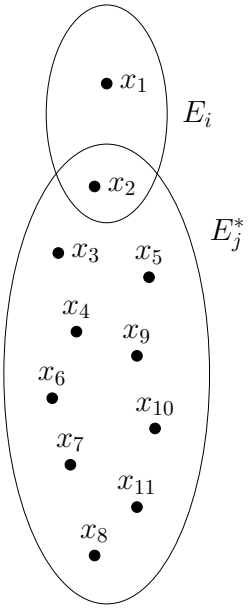


FIGURE 5.1 – Ensemble de clusters de la décomposition de la figure 4.1 lorsque E_j est fusionné avec ses descendants (cluster E_j^*).

position est jugée inefficace. Prenons l'exemple de la décomposition de la figure 4.1 du chapitre 4 représentée par l'ensemble de clusters $E = \{E_i, E_j, E_k, E_l, E_m, E_n\}$ enracinée en E_i. Soit E_j le cluster courant et $\mathcal{A}$ l'affectation courante. Nous nous intéressons au sous-problème $P_j|\mathcal{A}$ enraciné en E_j induit par l'affectation $\mathcal{A}$ du séparateur $E_i \cap E_j$ avec E_i le cluster parent de E_j. La résolution du sous-problème $P_j|\mathcal{A}$ n'utilisera pas forcément la décomposition, soit l'ensemble des clusters $\{E_j, E_k, E_l, E_m, E_n\}$. En effet, au début, la résolution est basée sur le cluster E_j^* résultant de la fusion du cluster E_j avec ses descendants E_k, E_l, E_m et E_n. D'où, formellement $E_j^* = \cup_{E_p \in Desc(E_j)} E_p$. Cette fusion consiste en une mise en commun de toutes les variables des clusters de la descendance de E_j (voir figure 5.1). Ainsi E_j^* contient toutes les variables des clusters descendants de E_j, E_j inclus. En ce qui concerne l'heuristique de choix de variables, elle acquiert une liberté totale quant au choix des variables suivantes sur E_j^*. Plus précisément, l'ordre partiel induit par la décomposition est : $\Lambda = [\{x_1, x_2\}, \{x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}\}]$.
À ce niveau, deux cas se présentent :

- Le sous-problème $P_j|\mathcal{A}$ est facilement résolu en se basant sur E_j^*. Dans ce cas, l'exploitation de la décomposition ne semble pas utile et ne sera pas exploitée.
- La résolution de $P_j|\mathcal{A}$ est au contraire inefficace. Dans ce cas, l'exploitation de la décomposition semble judicieuse. Le cluster E_j est alors réexploité et le même raisonnement est répété au niveau du cluster E_k qui sera au début considéré fusionné avec les clusters de sa descendance formant le cluster E_k^* comme le montre la figure 5.2. L'ordre partiel ainsi induit par la décomposition est alors :
  $\Lambda = [\{x_1, x_2\}, \{x_3, x_4, x_5\}, \{x_6, \{x_7, x_8, x_9, x_{10}, x_{11}\}\}]$

Quant aux clusters feuilles (n'ayant pas de fils comme E_n), $BTD\text{-}DFS{+}DYN$ se comporte comme $BTD\text{-}DFS$. À noter que ce raisonnement est répété pour chaque nouvelle affectation de $E_i \cap E_j$. Ce choix est motivé par le fait qu'une affectation $\mathcal{A}$ du séparateur $E_i \cap E_j$ induit un sous-problème différent de celui induit par une affectation $\mathcal{A}'$. À noter aussi qu'au niveau du cluster racine, $BTD\text{-}DFS{+}DYN$ se comporte comme l'algorithme

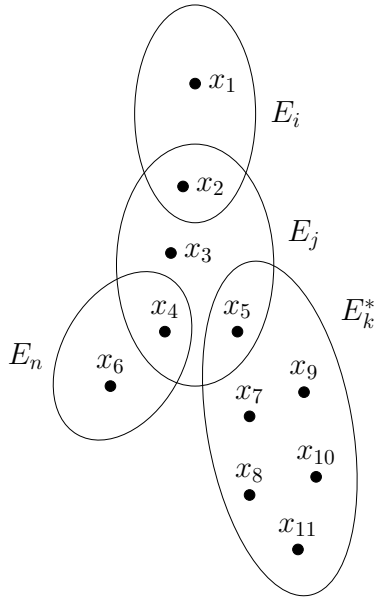


FIGURE 5.2 – Ensemble de clusters de la décomposition de la figure 4.1 lorsque E_j est exploité et E_k fusionné avec ses descendants (cluster E_k^*).

de base ayant toute la liberté concernant le choix des variables du problème. Finalement, la décomposition initiale est utilisée complètement (tous ses clusters sont exploités) si à tous les niveaux $BTD\text{-}DFS{+}DYN$ décide d'exploiter le cluster lui-même plutôt que sa descendance.

5.3.2 Description de l'algorithme BTD-DFS+DYN

Nous décrivons dans cette sous-section l'algorithme $BTD\text{-}DFS{+}DYN$ en mettant en avant les similitudes avec $BTD\text{-}DFS$ ainsi que les modifications qui y sont apportées.

5.3.2.1 Similitudes avec BTD-DFS

L'algorithme $BTD\text{-}DFS{+}DYN$ prend en paramètres l'affectation courante $\mathcal{A}$, le cluster courant E_i, l'ensemble V_{E_i} des variables non affectées de E_i, l'ensemble V_{desc_i} des variables non affectées de la descendance $Desc(E_i)$ de E_i et les bornes inférieure et supérieure courantes *clb* et *cub*. Il vise à calculer l'optimum du problème enraciné en E_i et induit par l'affectation $\mathcal{A}$. Lorsque l'optimum est calculé, la valeur de *cub* retournée correspond à ce dernier. Par souci de clarté, le paramètre *cub* donné en entrée sera noté cub^e et celui en sortie sera désigné par cub^s. Deux cas sont possibles :

- Si la valeur de cub^s est strictement inférieure à celle de cub^e, alors cub^s est l'optimum correspondant à ce sous-problème.
- Sinon, cub^s définit une borne inférieure de l'optimum.

L'appel initial est $BTD\text{-}DFS{+}DYN(\emptyset, E_r, V_{E_r}, V_{desc_r}, lb(P_r|\emptyset), k)$ avec V_{desc_r} l'ensemble des variables de la descendance de E_r, $lb(P_r|\emptyset)$ la borne inférieure initiale déduite grâce à l'application de la cohérence locale au problème initial et k le coût maximal. Rappelons que tout au long de l'algorithme, $w_\emptyset^i$ désigne la borne inférieure localisée relative au cluster E_i. Comme $BTD\text{-}DFS$, $BTD\text{-}DFS{+}DYN$ suppose que le problème donné en entrée

Algorithme 5.1 : BTD-DFS+DYN $(\mathcal{A}, E_i, V_{E_i}, V_{desc_i}, clb, cub)$

Entrées : L'affectation courante $\mathcal{A}$, le cluster courant E_i, la borne inférieure clb
Entrées-Sorties : L'ensemble V_{E_i} des variables non instanciées de E_i, l'ensemble V_{desc_i} des variables non instanciées de $V_{Desc(E_i)}$, la borne supérieure courante cub

1 **si** $Fusion(E_i)$ **alors**
2 $V' \leftarrow V_{desc_i}$
3 **sinon**
4 $V' \leftarrow V_{E_i}$
5 **si** $V' \neq \emptyset$ **alors**
6 $x \leftarrow$dépiler(V') `/* Choisir une variable de` V' `*/`
7 Mettre à jour V_{E_i} et V_{desc_i}
8 $v \leftarrow$dépiler(D_x) `/* Choisir une valeur */`
9 Appliquer la cohérence locale au sous-problème $P_i|\mathcal{A} \cup \{(x = v)\}$
10 $clb' \leftarrow max(clb, lb(P_i|\mathcal{A} \cup \{(x = v)\}))$
11 **si** $clb' < cub$ **alors**
12 $cub \leftarrow$BTD-DFS+DYN$(\mathcal{A} \cup \{(x = v)\}, E_i, V_{E_i}, V_{desc_i}, clb', cub)$
13 **si** $max(clb, lb(P_i|\mathcal{A})) < cub$ **alors**
14 Appliquer la cohérence locale au sous-problème $P_i|\mathcal{A} \cup \{(x \neq v)\}$
15 $clb' \leftarrow \max(clb, lb(P_i|\mathcal{A} \cup \{(x \neq v)\}))$
16 **si** $clb' < cub$ **alors**
17 $cub \leftarrow$ BTD-DFS+DYN $(\mathcal{A} \cup \{(x \neq v)\}, E_i, V_{E_i}, V_{desc_i}, clb', cub)$
18 **sinon**
19 **si** $\neg Fusion(E_i)$ **alors**
20 $Q_{E_i} \leftarrow Fils(E_i)$
 `/* Résoudre les fils dont l'optimum n'est pas connu */`
21 **tant que** $Q_{E_i} \neq \emptyset$ **et** $lb(P_i|A) < cub$ **faire**
22 $E_j \leftarrow$dépiler(Q_{E_i}) `/* Choisir un cluster fils */`
23 **si** $LB_{P_j|\mathcal{A}} < UB_{P_j|\mathcal{A}}$ **alors**
24 $cub' \leftarrow \min\,(UB_{P_j|\mathcal{A}}, cub - [lb(P_i|\mathcal{A}) - lb(P_j|\mathcal{A})])$
25 $cub'' \leftarrow$ BTD-DFS+DYN $(\mathcal{A}, E_j, E_j \backslash (E_i \cap E_j),$ $Desc(E_j)\backslash(E_i \cap E_j), lb(P_j|\mathcal{A}), cub')$
26 Mettre à jour $LB_{P_j|\mathcal{A}}$ et $UB_{P_j|\mathcal{A}}$ en se basant sur cub''
27 $cub \leftarrow \min(cub, w^i_\emptyset + \sum_{E_j \in Fils(E_i)} UB_{P_j|\mathcal{A}})$
28 **sinon**
29 $cub \leftarrow \min(cub, \sum_{E_j \in Desc(E_i)} w^j_\emptyset)$
30 **retourner** cub

est cohérent selon la cohérence locale utilisée et renvoie son optimum. Les lignes 5 à 17 de $BTD\text{-}DFS{+}DYN$ visent à instancier les variables de V' comme le ferait $BTD\text{-}DFS$ pour les variables de V_{E_i}. Un couple (variable, valeur), (x, v), est sélectionné selon les heuristiques de choix de variables et de valeurs employées. Comme le type de branchement utilisé est binaire, ce choix se décline en deux branches $x = v$ (ligne 9) et $x \neq v$ (ligne 14). Pour chaque branche, la cohérence locale est appliquée au sous-problème enra-

ciné en E_i et induit par l'affectation courante permettant d'obtenir une borne inférieure $lb(P_i|\mathcal{A} \cup \{(x = v)\})$ si une décision positive est faite et $lb(P_i|\mathcal{A} \cup \{(x \neq v)\})$ sinon (cf. la partie 2.4.4.2 et l'algorithme $Lc\text{-}BTD^+$ pour de plus amples détails). Si le maximum clb' entre la borne inférieure $lb(P_i|\mathcal{A} \cup \{(x = v)\})$ (ou $lb(P_i|\mathcal{A} \cup \{(x \neq v)\})$ lorsqu'il s'agit d'une décision négative), déduite par la cohérence locale, et la borne inférieure courante *clb*, est toujours inférieure à *cub*, la recherche continue ; sinon le sous-arbre correspondant est élagué. Les lignes 20 à 27 de $BTD\text{-}DFS{+}DYN$ résolvent les sous-problèmes enracinés en chaque cluster fils E_j de E_i de la même façon que $BTD\text{-}DFS$. Le sous-problème $P_j|\mathcal{A}$ n'est exploré que si son optimum n'est pas déjà connu. D'ailleurs, $BTD\text{-}DFS$ et $BTD\text{-}DFS{+}DYN$ mémorisent pour chaque sous-problème $P_j|\mathcal{A}$ deux valeurs, notées $LB_{P_j|\mathcal{A}}$ et $UB_{P_j|\mathcal{A}}$, représentant respectivement les meilleures bornes inférieure et supérieure connues pour $P_j|\mathcal{A}$. Si $LB_{P_j|\mathcal{A}} = UB_{P_j|\mathcal{A}}$, l'optimum de P_j est déjà calculé. L'appel récursif correspondant au fils E_j (ligne 25) exploite une borne supérieure initiale non triviale calculée à la ligne 24 comme expliqué dans [Givry et al., 2006]. Plus précisément, bien que l'utilisation d'un majorant initial trivial (k dans ce cas), garantisse effectivement que l'optimum de $P_j|\mathcal{A}$ sera correctement calculé, cela ne permet pas d'élaguer efficacement l'espace de recherche en n'ayant pas une coupe initiale efficace. En outre, en résolvant le sous-problème $P_j|\mathcal{A}$ indépendamment des autres sous-problèmes, nous pourrons détériorer l'efficacité de la résolution. En effet, l'optimum trouvé pour $P_j|\mathcal{A}$ additionné aux coûts des clusters déjà affectés et aux bornes inférieures calculées pour d'autres, peut éventuellement largement dépasser la borne supérieure courante et ne pas ainsi être capable de participer à une solution globale. Ainsi, $P_j|\mathcal{A}$ exploite une borne supérieure non triviale qui est le minimum entre la meilleure borne supérieure enregistrée pour ce problème $UB_{P_j|\mathcal{A}}$ et une deuxième borne supérieure déduite de la différence entre la borne supérieure courante *cub* et le minorant de $P_i|\mathcal{A}$ duquel nous retranchons le minorant de $P_j|\mathcal{A}$. En exploitant cette nouvelle borne supérieure, si une solution de coût strictement inférieur à cette borne est trouvée, l'optimum est ainsi trouvé. Sinon, nous pouvons uniquement déduire une borne inférieure de l'optimum. L'appel récursif est suivi par une mise à jour de $LB_{P_j|\mathcal{A}}$ et de $UB_{P_j|\mathcal{A}}$ (ligne 26). En effet, si à la ligne 25 l'optimum est trouvé, les valeurs de $LB_{P_j|\mathcal{A}}$ et $UB_{P_j|\mathcal{A}}$ seront toutes les deux égales à l'optimum. Sinon, $LB_{P_j|\mathcal{A}}$ sera mis à jour en fonction de la nouvelle borne inférieure trouvée. L'exploration de tous les clusters fils permet enfin de mettre à jour *cub* (ligne 27). Cette mise à jour se fait en prenant le minimum entre la borne supérieure courante *cub* et celle déduite de la somme du coût de l'affectation courante du cluster E_i représenté par $w^i_\emptyset$ et des meilleures bornes supérieures $UB_{P_j|\mathcal{A}}$ trouvées pour chaque sous-problème $P_j|\mathcal{A}$.

5.3.2.2 Modifications réalisées pour BTD-DFS+DYN

Les similitudes entre $BTD\text{-}DFS$ et $BTD\text{-}DFS{+}DYN$ rappelées, nous nous intéressons maintenant aux modifications faites. La fonction $Fusion$ représente l'heuristique chargée de faire le choix d'exploiter le cluster E_i ou le cluster E^*_i. L'appel à $Fusion$ avec le cluster E_i en entrée renvoie vrai si et seulement si E^*_i est exploité. Sinon, le cluster E_i est exploité et la liberté de choix de variables est restreinte aux variables de V_{E_i}. L'un des deux paramètres V_{E_i} ou V_{desc_i} est effectivement utilisé selon que nous exploitons E_i ou E^*_i. Le choix entre V_{E_i} et V_{desc_i} est fait dans les lignes 1 à 4 et est retenu dans V'. Ces ensembles sont mis à jour convenablement à la ligne 7. Si $V' = \emptyset$ et que E_i n'est pas fusionné avec sa descendance, $BTD\text{-}DFS{+}DYN$ se comporte comme $BTD\text{-}DFS$ (lignes 20-27). Si $V' = \emptyset$ et que E_i est fusionné avec sa descendance, $BTD\text{-}DFS{+}DYN$ n'a plus de variables à instancier vu que toutes les variables de la descendance de E_i sont déjà affectées. Dans ce cas, $BTD\text{-}DFS{+}DYN$ met à jour *cub* (ligne 29) en se référant à la

borne inférieure localisée $w_\emptyset^j$ de chaque cluster E_j appartenant à $Desc(E_i)$. En effet, la nouvelle borne supérieure *cub* est alors le maximum entre la borne supérieure courante *cub* et la somme des coûts des affectations des clusters de la descendance de E_i, $Desc(E_i)$. Si $V' \neq \emptyset$ (lignes 5-17), $BTD\text{-}DFS{+}DYN$ se comporte de la même façon indépendamment du choix de l'exploitation de E_i ou de E_i^* en essayant d'instancier les variables de V'.

L'algorithme $BTD\text{-}DFS{+}DYN$ est paramétrable par l'heuristique $Fusion$ (nous en proposons une dans la partie expérimentale) qui se charge de décider de passer de l'exploitation du cluster E_i^* à E_i, si E_i est le cluster courant. Un choix pertinent de cette heuristique est, bien sûr, essentiel pour améliorer la résolution.

5.3.3 Fondements théoriques

Nous nous intéressons à présent à la validité de $BTD\text{-}DFS{+}DYN$. La clé de la validité de $BTD\text{-}DFS{+}DYN$ réside dans la validité des bornes supérieure et inférieure enregistrées pour chaque sous-problème. En effet, pour un cluster E_i, quels que soit les clusters inclus dans $Desc(E_i)$, $LB_{P_i|\mathcal{A}}$ et $UB_{P_i|\mathcal{A}}$ resterons valides puisque le sous-problème $P_i|\mathcal{A}$ ne change pas. Nous pouvons alors énoncer le théorème suivant :

Théorème 11 *$BTD\text{-}DFS{+}DYN$ est correct, complet et termine.*

Preuve : La correction, la complétude et la terminaison de $BTD\text{-}DFS{+}DYN$ se base sur la correction, la complétude et la terminaison de $BTD\text{-}DFS$ [Givry et al., 2006]. Soit (E, T) la décomposition de base enracinée en E_r pouvant être exploitée par $BTD\text{-}DFS{+}DYN$.

Si tout au long de la résolution, la décomposition effectivement employée est celle contenant un seul cluster (le cas où le seul cluster utilisé de la décomposition est E_r^*) la validité de $BTD\text{-}DFS{+}DYN$ est trivialement garantie grâce à la validité de DFS.

Supposons maintenant que la décomposition employée à un instant donné contient au moins deux clusters, E_i et E_j^*. Autrement dit, pour l'affectation $\mathcal{A}$ du séparateur $E_i \cap E_j$, $BTD\text{-}DFS{+}DYN$ commence à exploiter le cluster E_j^*. Tant que pour l'affectation $\mathcal{A}[E_i \cap E_j]$ $BTD\text{-}DFS{+}DYN$ exploite E_j^*, $BTD\text{-}DFS{+}DYN$ se comporte comme $BTD\text{-}DFS$ sur la base d'une décomposition en deux clusters E_i et E_j^*. De ce fait, comme $BTD\text{-}DFS$ est correct, complet et termine, $BTD\text{-}DFS{+}DYN$ l'est aussi. $BTD\text{-}DFS{+}DYN$ peut éventuellement trouver l'optimum de $P_j|\mathcal{A}$ en exploitant E_j^* et l'enregistrer pour le séparateur $E_i \cap E_j$ qui est le même que celui de $E_i \cap E_j^*$.

Supposons maintenant que l'heuristique $Fusion$ décide d'exploiter le cluster E_j au lieu du cluster E_j^* avant que l'optimum ne soit trouvé. Les bornes inférieures et supérieures $LB_{P_j|\mathcal{A}}$ et $UB_{P_j|\mathcal{A}}$ calculées auparavant restent valides quelle que soit la décomposition employée pour le sous-problème P_j. Elles peuvent ainsi être utilisées en toute sécurité pour renseigner la recherche sur la nouvelle décomposition du problème P_j. Désormais, pour la même affectation $\mathcal{A}$ du séparateur $E_i \cap E_j$, $BTD\text{-}DFS{+}DYN$ exploitera définitivement le cluster E_j. De même, $BTD\text{-}DFS{+}DYN$ se comporte maintenant comme $BTD\text{-}DFS$ sur P_j à base d'une décomposition formée du cluster E_j et des clusters correspondants à la fusion de chaque cluster fils de E_j avec ses descendants. En répétant le même raisonnement à tous les niveaux de la décomposition et pour toutes les affectations d'un séparateur, nous en déduisons grâce à la validité de $BTD\text{-}DFS$, la validité de $BTD\text{-}DFS{+}DYN$. □

L'exploitation dynamique de la décomposition induit un changement au niveau des complexités en temps et en espace.

Théorème 12 *$BTD\text{-}DFS{+}DYN$ a une complexité temporelle en $O(exp(w^{*+}{+}1))$ et une complexité spatiale en $O(exp(s^*))$ avec $w^{*+}{+}1$ la taille du plus grand cluster effectivement exploité et s^* (avec $s^* \leq s$) la taille du plus grand séparateur effectivement utilisé.*

Preuve : Bien que $BTD\text{-}DFS{+}DYN$ se base initialement sur une décomposition précalculée en amont de la résolution, la décomposition n'est pas exploitée traditionnellement. En particulier, $BTD\text{-}DFS{+}DYN$ peut ne jamais exploiter la décomposition, l'exploiter partiellement ou aussi utiliser l'ensemble de tous ses clusters. La complexité temporelle dépend ainsi de la taille du plus grand cluster exploité $w^{*+}+1$ tandis que la complexité spatiale dépend de la taille du plus grand séparateur utilisé s^*. De ce fait, les complexités spatiales et temporelles dépendent de l'heuristique de $Fusion$ utilisée. □

Le comportement de $BTD\text{-}DFS{+}DYN$ pouvant aller d'un $BTD\text{-}DFS$ standard à un DFS classique, il en résulte que la complexité temporelle de $BTD\text{-}DFS{+}DYN$ est comprise entre $O(exp(w^{+}{+}1))$ et $O(exp(n))$ avec w^+ la largeur de la décomposition calculée en amont de la résolution. Toutefois, il est possible de limiter cette complexité en utilisant une heuristique convenable qui se charge d'interdire l'exploitation de la fusion des descendants d'un cluster pour tout cluster situé à un profondeur faible par rapport à la racine. En particulier, l'heuristique de fusion pourrait interdire l'exploitation du cluster E_r^* et ainsi empêcher $BTD\text{-}DFS{+}DYN$ de se comporter comme DFS. Au niveau de la complexité en espace, dans le pire des cas, la complexité en espace est la même que celle de $BTD\text{-}DFS$ si le plus grand séparateur de la décomposition est utilisé puisque $s^* \leq s$.

5.4 Étude expérimentale

Dans cette sous-section, nous évaluons la pertinence de l'exploitation dynamique de la décomposition. Nous évoquons tout d'abord le protocole expérimental.

5.4.1 Protocole expérimental

Nous reprenons, pour ces expérimentations, le protocole expérimental de la partie 3.4.3 du chapitre 3. Nous considérons les algorithmes $HBFS$, $BTD\text{-}HBFS$ et $BTD\text{-}HBFS{+}DYN$. Pour les deux premiers algorithmes, nous exploitons leurs implémentations fournies dans $Toulbar2$ [TOU, 2006]. Nous avons également implémenté l'algorithme $BTD\text{-}HBFS{+}DYN$ au sein de $Toulbar2$. Notre implémentation se base sur celle de $BTD\text{-}HBFS$ tout en apportant les modifications nécessaires. Pour tous les algorithmes, $\alpha_{hbfs} = 5\%, \beta_{hbfs} = 10\%$ et $N_{hbfs} = 10\ 000$ comme dans [Allouche et al., 2015]. L'efficacité pratique de $BTD\text{-}HBFS$ nous incite à le considérer au lieu de $BTD\text{-}DFS$. L'extension $BTD\text{-}HBFS{+}DYN$ peut être facilement déduite de $BTD\text{-}HBFS$ d'une façon similaire à la déduction de $BTD\text{-}DFS{+}DYN$ à partir de $BTD\text{-}DFS$. Concernant les décompositions, nous considérons de nouveau les décompositions $Min\text{-}Fill$ et $Min\text{-}Fill^4$ dont l'implémentation est fournie dans $Toulbar2$. En outre, nous considérons les décompositions H_2, H_3 et H_5 avec ses deux variantes H_5^{25} et $H_5^{5\%}$. La configuration de $Toulbar2$ est identique à celle décrite dans la partie 3.4.3 du chapitre 3. Les instances de I_3 sont toutes retenues. Les algorithmes de résolution ont un temps limite de 20 minutes incluant pour les algorithmes basés sur BTD le temps de calcul de la décomposition et 16 Go d'espace mémoire.

Heuristique de fusion $\mathcal{F}$ L'exploitation dynamique de la décomposition se base sur l'heuristique $\mathcal{F}$ de fusion qui est responsable de décider d'exploiter le cluster initial E_i

Algorithme	*Min-Fill*		*Min-Fill*^{4}	
	#rés.	temps	#rés.	temps
BTD-HBFS	1 712	26 291	1 995	91 232
BTD-HBFS+DYN	1 905	45 450	2 019	74 159

TABLE 5.1 – Nombre d'instances résolues et temps d'exécution en secondes pour *BTD-HBFS* et *BTD-HBFS+DYN* selon les décompositions de l'état de l'art.

Algorithme	H_2		H_3		H_5^{25}		$H_5^{5\%}$	
	#rés.	temps	#rés.	temps	#rés.	temps	#rés.	temps
BTD-HBFS	1 946	76 018	1 989	57 348	2 006	58 826	2 028	58 043
BTD-HBFS+DYN	2 023	50 445	2 023	56 978	2 039	52 304	2 038	60 674

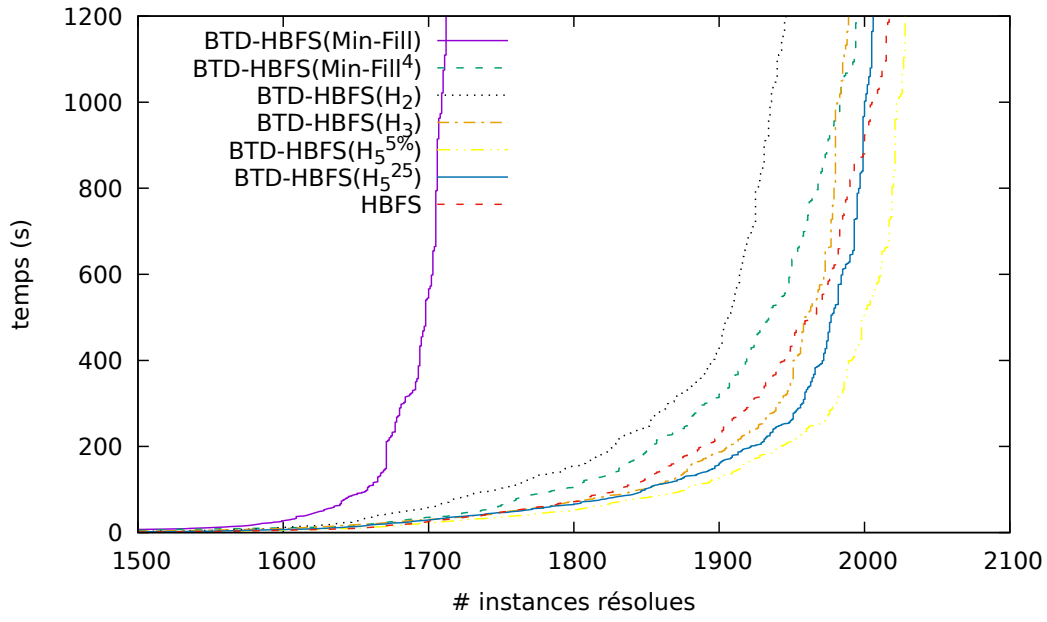
TABLE 5.2 – Nombre d'instances résolues et temps d'exécution en secondes pour *BTD-HBFS* et *BTD-HBFS+DYN* selon les décompositions de *H-TD*.

ou le cluster fusionné E_i^*. L'heuristique $\mathcal{F}$ se base sur le retour d'informations fait par *BTD-HBFS*. Elle exploite notamment son comportement *anytime* et les mises à jour permanentes des bornes inférieures et supérieures reportées pour chaque sous-problème. En effet, chaque appel à *BTD-HBFS* sur un sous-problème $P_i|\mathcal{A}$ prend en entrées une borne inférieure *clb* et une borne supérieure *cub*. Si, dans la limite du nombre de retours en arrière autorisé, *BTD-HBFS* ne réussit pas à améliorer une des deux bornes, $\mathcal{F}$ considère que la résolution de $P_i|\mathcal{A}$ n'avance pas et incrémente un compteur qui lui est propre. Lorsque le compteur relatif à $P_i|\mathcal{A}$ atteint une certaine limite, nous exploitons par la suite E_i en stoppant l'exploitation de E_i^*. Ce faisant, *BTD-HBFS+DYN* accumule des informations relatives aux états précédents de la résolution. Grâce à ces enregistrements, *BTD-HBFS+DYN* est capable de s'adapter au contexte de la résolution et aux particularités de l'instance à résoudre. La limite que nous choisissons dans nos expérimentations est 5. Ce choix est le fruit des expérimentations intensives qui ont permis d'évaluer l'intérêt des différentes valeurs pour cette limite. Une limite trop élevée s'avère contre-productive puisque *BTD-HBFS+DYN* se comporterait souvent comme *HBFS*. Au contraire, une limite inférieure à 5 empêcherait *BTD-HBFS+DYN* quand il le faut de bénéficier de la liberté totale offerte à l'heuristique de choix de variables sur un sous-problème. La limite de 5 est alors capable de donner de bons résultats en moyenne sur l'ensemble des instances du benchmark même si bien sûr, rien ne garantit que cette limite soit la plus adéquate pour chaque instance.

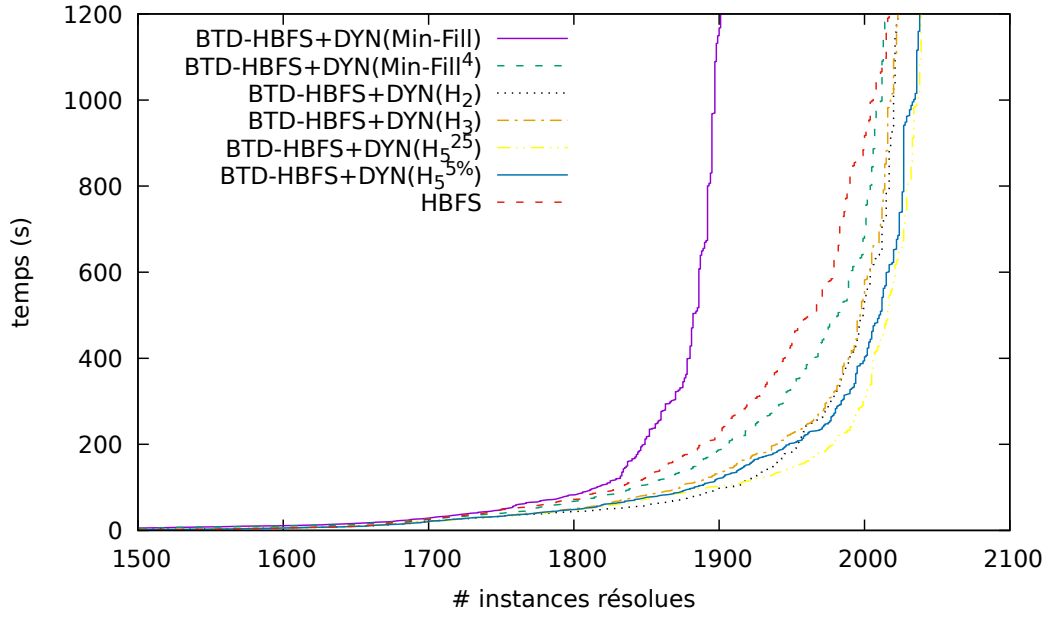
D'autres heuristiques peuvent être également proposées selon le besoin de l'utilisateur. Notons finalement que l'heuristique $\mathcal{F}$ proposée est fondée sur les retours d'informations de *BTD-HBFS*. L'utilisation d'un algorithme différent de *HBFS* nécessiterait un changement de l'heuristique de fusion exploitée.

5.4.2 Observations et analyse des résultats

Apport de l'utilisation si besoin de la décomposition par rapport à son utilisation traditionnelle Nous évaluons à présent *BTD-HBFS+DYN*. Les deux tables 5.1 et 5.2 ainsi que la figure 5.3 montrent, que quelle que soit la décomposition exploitée, *BTD-HBFS+DYN* résout plus d'instances que *BTD-HBFS*. L'augmentation du nombre d'instances résolues peut être considérable comme c'est le cas pour *Min-Fill* qui permet de résoudre 1 905 instances au lieu de 1 712 instances. L'utilisation de H_2 et H_3 avec *BTD-HBFS+DYN* permet de résoudre 2 023 instances contre 1 946 et 1 989 résolues par *BTD-HBFS*. L'exploitation de *Min-Fill*^{4} et H_5 est aussi améliorée avec



(a)



(b)

FIGURE 5.3 – Le nombre cumulative d'instances résolues pour $BTD\text{-}HBFS$ et $HBFS$ (a) et pour $BTD\text{-}HBFS{+}DYN$ et $HBFS$ (b) pour les instances de I_3.

$BTD\text{-}HBFS{+}DYN$ par rapport à $BTD\text{-}HBFS$. En particulier, H_5^{25} permet de résoudre 2 039 instances, ce qui constitue le plus grand nombre d'instances résolues parmi toutes les combinaisons d'algorithmes de résolution et de décomposition présentées. L'augmentation du nombre d'instances résolues est souvent accompagnée d'une diminution du temps total d'exécution, sauf pour $Min\text{-}Fill$, mais ce dernier point s'explique naturellement par le nombre d'instances supplémentaires résolues par $BTD\text{-}HBFS{+}DYN$ par rapport à $BTD\text{-}HBFS$. C'est ainsi que $BTD\text{-}HBFS{+}DYN$ avec $Min\text{-}Fill^4$ résout 2 019 instances en 74 159 s tandis que $BTD\text{-}HBFS$ avec $Min\text{-}Fill^4$ requiert 91 232 s pour résoudre 1 995

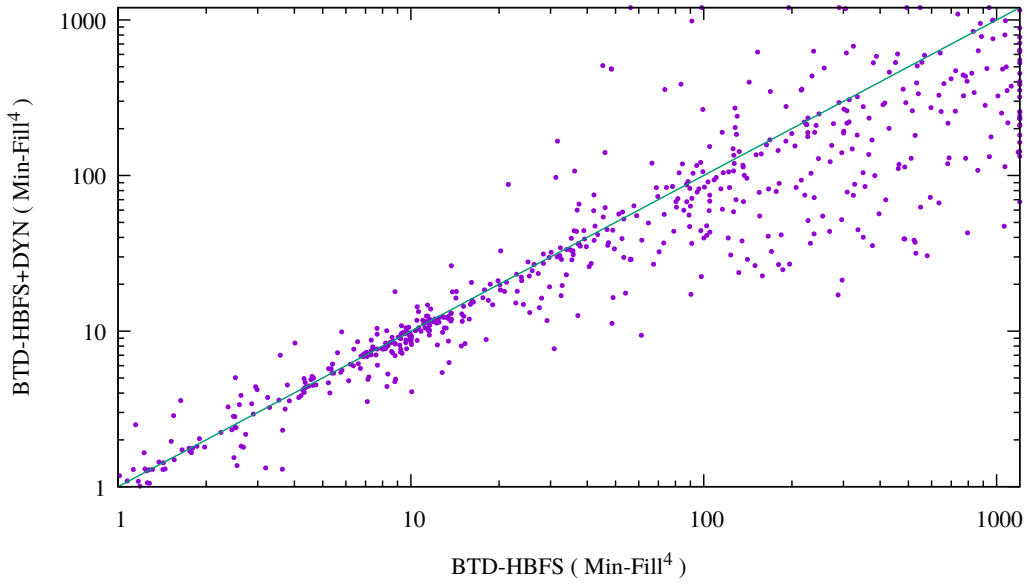


FIGURE 5.4 – Comparaison des temps d'exécution de $BTD\text{-}HBFS{+}DYN$ avec $Min\text{-}Fill^4$ à $BTD\text{-}HBFS$ avec $Min\text{-}Fill^4$ pour les 2 444 instances.

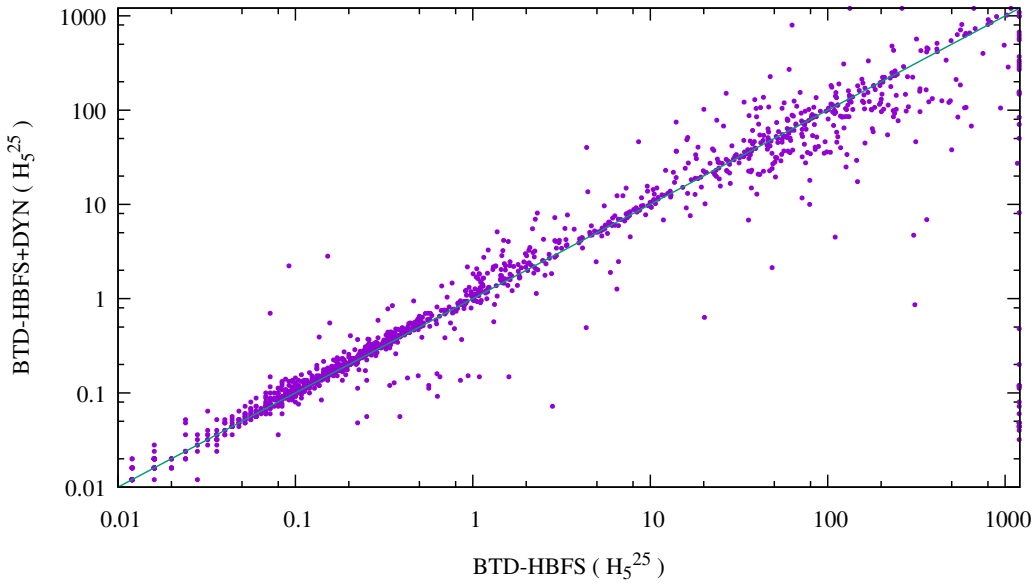


FIGURE 5.5 – Comparaison des temps d'exécution de $BTD\text{-}HBFS{+}DYN$ avec H_5^{25} à $BTD\text{-}HBFS$ avec H_5^{25} pour les 2 444 instances.

instances. La figure 5.4 compare les temps de d'exécution de $BTD\text{-}HBFS{+}DYN$ avec $Min\text{-}Fill^4$ à $BTD\text{-}HBFS$ avec $Min\text{-}Fill^{r_4}$. Elle montre que l'efficacité de la résolution a sensiblement augmenté avec l'exploitation dynamique de la décomposition. Concernant H_5^{25}, $BTD\text{-}HBFS{+}DYN$ utilisant H_5^{25} requiert 52 304 s pour résoudre 2 039 instances tandis que $BTD\text{-}HBFS$ nécessite 58 826 s pour résoudre 2 006 instances. La figure 5.5 qui compare les temps de d'exécution $BTD\text{-}HBFS{+}DYN$ avec H_5^{25} à $BTD\text{-}HBFS$ avec

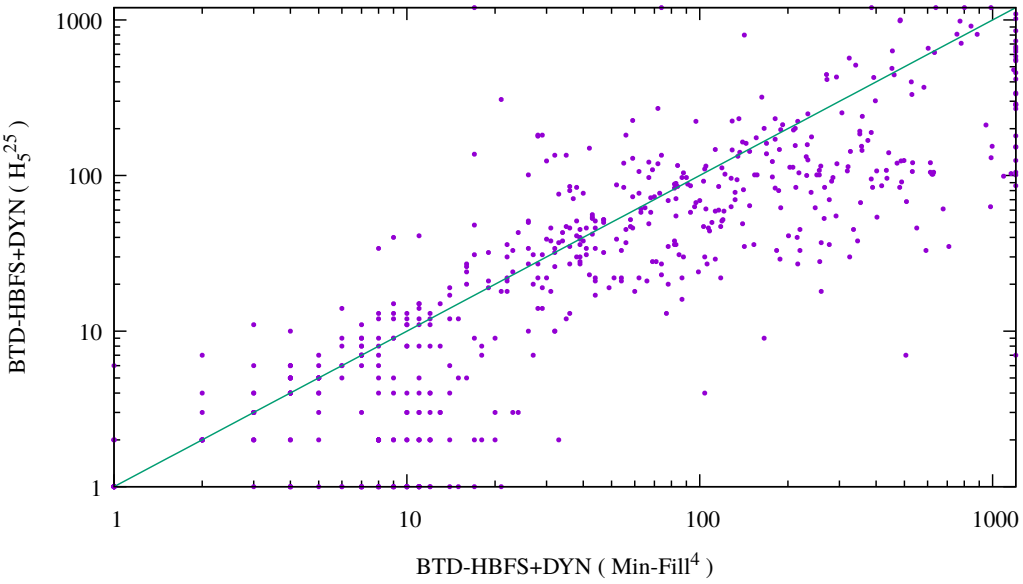


FIGURE 5.6 – Comparaison des temps d'exécution de $BTD\text{-}HBFS{+}DYN$ avec H_5^{25} à $BTD\text{-}HBFS{+}DYN$ avec $Min\text{-}Fill^4$ pour les 2 444 instances.

H_5^{25} indique que les temps d'exécution sont plus souvent comparables que dans le cas de $Min\text{-}Fill^4$. Nous nous intéressons finalement à la comparaison de H_5^{25} à l'heuristique de l'état de l'art $Min\text{-}Fill^4$ avec $BTD\text{-}HBFS{+}DYN$. La figure 5.6 présente une comparaison des temps d'exécution de $BTD\text{-}HBFS{+}DYN$ avec H_5^{25} à $BTD\text{-}HBFS{+}DYN$ avec $Min\text{-}Fill^4$. $BTD\text{-}HBFS{+}DYN$ associé à H_5^{25} prouve clairement son intérêt pratique. Pour une comparaison plus équitable des temps cumulés d'exécution, nous nous appuions sur le benchmark des instances résolues à la fois par $BTD\text{-}HBFS{+}DYN$ avec $Min\text{-}Fill^4$ et par $BTD\text{-}HBFS{+}DYN$ avec H_5^{25}. Il compte 2 013 instances résolues par $BTD\text{-}HBFS{+}DYN$ avec $Min\text{-}Fill^4$ en 71 249 s contre seulement 41 372 s par $BTD\text{-}HBFS{+}DYN$ avec H_5^{25}. Nous retenons pour la suite la décomposition $Min\text{-}Fill^4$ vu que son exploitation avec $BTD\text{-}HBFS$ est considérée comme la méthode structurelle de référence pour la résolution d'instances WCSP. Nous retenons également la décomposition H_5^{25} en raison de son intérêt vis-à-vis de la résolution avec $BTD\text{-}HBFS(+DYN)$.

$HBFS$ et $BTD\text{-}HBFS$ avec $Min\text{-}Fill^4$ sont considérés les méthodes de l'état de l'art respectivement sans et avec exploitation de la structure.

BTD-HBFS+DYN avec H_5^{25} vs BTD-HBFS avec $Min\text{-}Fill^4$ Nous comparons tout d'abord l'algorithme $BTD\text{-}HBFS$ avec $Min\text{-}Fill^4$ à $BTD\text{-}HBFS{+}DYN$ avec H_5^{25}. $BTD\text{-}HBFS{+}DYN$ surpasse significativement $BTD\text{-}HBFS$ en nombre d'instances et en temps de résolution. D'ailleurs, $BTD\text{-}HBFS{+}DYN$ avec H_5^{25} résout 2 039 instances en 52 304 s tandis que $BTD\text{-}HBFS$ résout 1 995 instances avec $Min\text{-}Fill^4$ en 91 232 s. La figure 5.7 comparant les temps d'exécution des deux méthodes confirment cette tendance.

BTD-HBFS+DYN avec H_5^{25} vs HBFS Nous comparons à présent l'algorithme $BTD\text{-}HBFS{+}DYN$ basé sur H_5^{25} à $HBFS$. Nous pouvons noter que $BTD\text{-}HBFS{+}DYN$ avec H_5^{25} a une meilleure performance que $HBFS$ (voir la figure 5.3). Plus précisément, $BTD\text{-}HBFS{+}DYN$ résout plus d'instances que $HBFS$ (2 039 instances contre 2 017 ins-

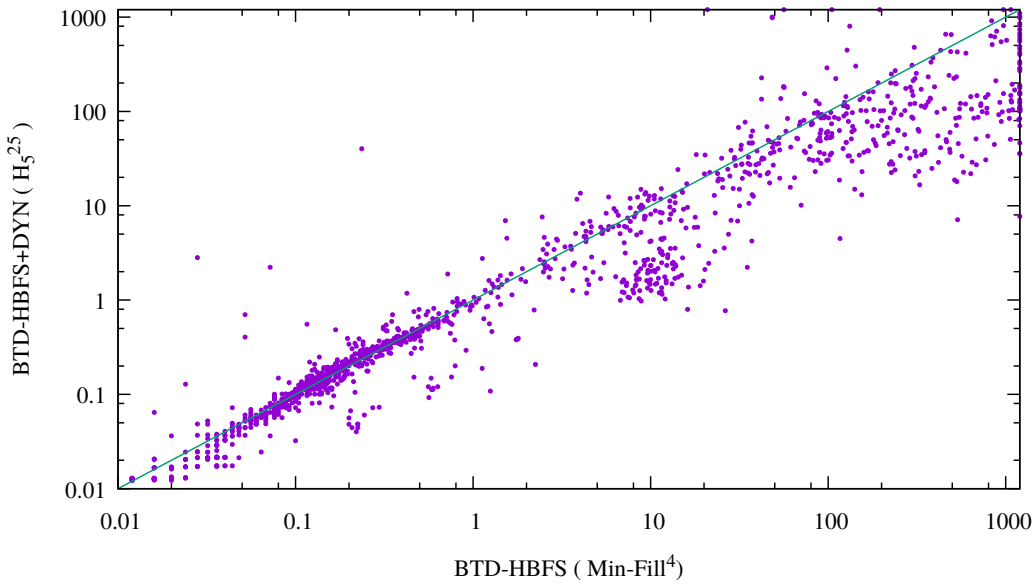


FIGURE 5.7 – Comparaison des temps d'exécution de $BTD\text{-}HBFS{+}DYN$ avec H_5^{25} à $BTD\text{-}HBFS$ avec $Min\text{-}Fill^4$ pour les 2 444 instances.

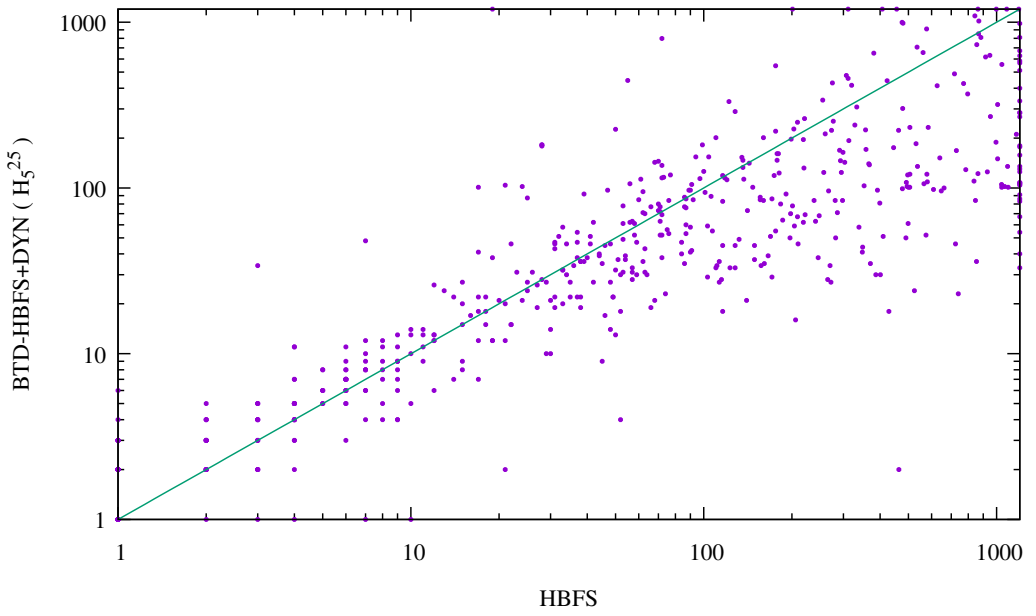


FIGURE 5.8 – Comparaison des temps d'exécution de $BTD\text{-}HBFS{+}DYN$ avec H_5^{25} à $HBFS$ pour les 2 444 instances.

tances). En addition, $BTD\text{-}HBFS{+}DYN$ a un meilleur temps de résolution que $HBFS$, à savoir 52 304 s contre 84 657 s pour $HBFS$. La figure 5.8 montre que ce constat reste valide pour le temps de résolution de la plupart d'instances. Afin de comparer de façon équitable leur temps de résolution, nous considérons le benchmark résolu à la fois par $BTD\text{-}HBFS{+}DYN$ et par $HBFS$. Nous obtenons alors un total de 2 008 intances résolues en 79 020 s par $HBFS$ contre seulement 43 914 s pour $BTD\text{-}HBFS{+}DYN$.

Ces résultats peuvent être essentiellement expliqués par les enregistrements réalisés par $BTD\text{-}HBFS{+}DYN$, par la distinction des différentes composantes connexes du problème et la gestion dynamique de la décomposition qui atténue les inconvénients de l'exploitation traditionnelle de la décomposition. Le couplage de $BTD\text{-}HBFS{+}DYN$ avec H_5^{25} assure notamment que les méthodes basées sur la décomposition sont compétitives vis-à-vis des méthodes n'exploitant pas la décomposition comme $HBFS$. Tous ces résultats montrent que la dynamicité exploitée au niveau de la décomposition semble permettre à $BTD\text{-}HBFS{+}DYN$ de mieux s'adapter à la nature de l'instance et d'éviter de se baser sur une décomposition lorsqu'une résolution sans décomposition s'avère plus efficace que ce soit globalement ou localement.

BTD-HBFS+DYN avec H_5^{25} sur les instances les plus difficiles Nous focalisons maintenant nos observations sur le comportement de $BTD\text{-}HBFS{+}DYN$ associé à H_5^{25}. Nous écartons à ce stade les instances faciles, voire parfois triviales (c'est-à-dire celles résolues en moins de 10 s par $HBFS$). Notons que ces instances sont aussi facilement résolues par $BTD\text{-}HBFS{+}DYN$ vu qu'au niveau du cluster racine $BTD\text{-}HBFS{+}DYN$ se comporte comme $HBFS$ (lorsque nous exploitons la fusion des descendants relatifs au cluster racine). Nous pouvons distinguer trois partitions sur ce benchmark de 794 instances :

- Pour 279 instances au moins un cluster feuille de la décomposition est exploité. Autrement dit, il existe au moins une branche de la décomposition qui est entièrement exploitée (au sens de l'ensemble de clusters d'origine de la décomposition).
- Pour 423 instances, la décomposition n'est jamais exploitée. Cela signifie que $BTD\text{-}HBFS{+}DYN$ se comporte comme $HBFS$.
- Pour les instances restantes, la décomposition est exploitée jusqu'à une certaine profondeur sans pour autant atteindre un cluster feuille de la décomposition.

Donc, en pratique, $BTD\text{-}HBFS{+}DYN$ peut se comporter simplement comme $HBFS$ ou faire des choix d'exploitation des clusters d'origine de la décomposition (au lieu de la fusion de leur descendants) jusqu'à atteindre le comportement de $BTD\text{-}HBFS$.

BTD-HBFS+DYN avec H_5^{25} vs HBFS en cas de dépassement du temps limite Nous comparons les bornes inférieures et supérieures rapportées par $HBFS$ et $BTD\text{-}HBFS{+}DYN$ dans le cas de dépassement du temps limite. Nous disposons de 375 instances qui ne sont pas résolues ni par $HBFS$, ni par $BTD\text{-}HBFS{+}DYN$ avec :

- Pour 252 instances, la borne supérieure calculée par $BTD\text{-}HBFS{+}DYN$ est strictement inférieure à celle de $HBFS$ contre seulement 52 instances pour $HBFS$.
- Pour 229 instances, $BTD\text{-}HBFS{+}DYN$ calcule une borne inférieure strictement supérieure à celle de $HBFS$ contre 146 instances pour $HBFS$.

La figure 5.9 compare l'écart entre les bornes inférieures et les bornes supérieures pour les deux algorithmes. Clairement, $BTD\text{-}HBFS{+}DYN$ offre un écart plus réduit que celui de $HBFS$. Cela montre que même lorsque l'instance n'est pas résolue, $BTD\text{-}HBFS{+}DYN$ est capable de donner des approximations de meilleure qualité que $HBFS$.

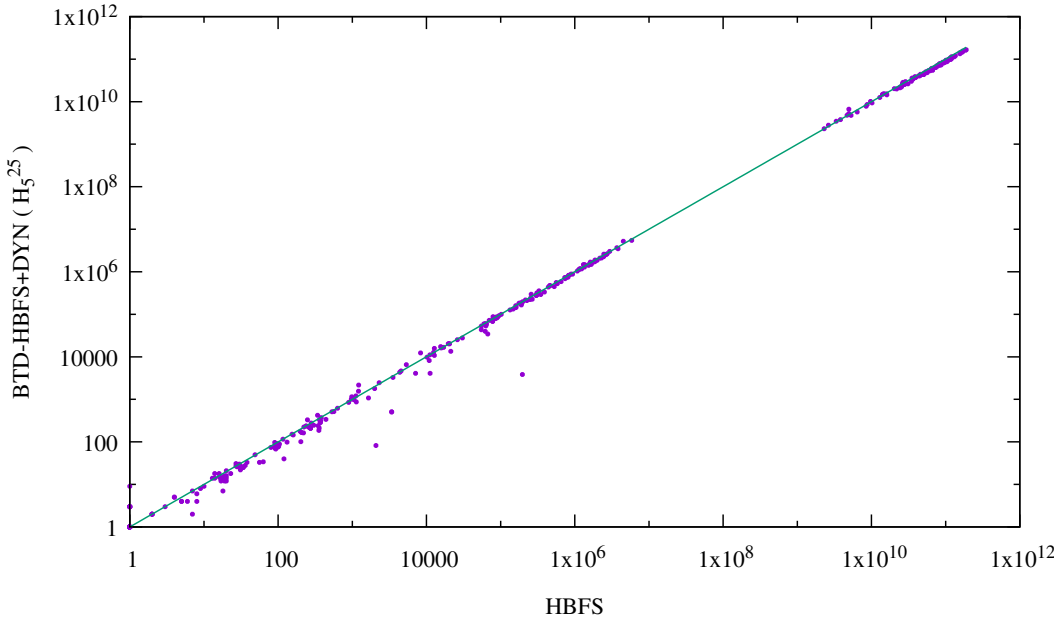


FIGURE 5.9 – Écart entre les bornes inférieures et supérieures pour $BTD\text{-}HBFS{+}DYN$ avec H_5^{25} et $HBFS$.

Bilan Au terme de l'analyse, nous avons démontré que $BTD\text{-}HBFS{+}DYN$ améliore $BTD\text{-}HBFS$, quelle que soit la décomposition utilisée, en nombre total d'instances résolues, en temps, mais également pour la qualité de bornes obtenues pour les cas des instances non résolues. Aussi, grâce à cette extension, les algorithmes basés sur une décomposition présentent davantage d'intérêt par rapport à ceux non basés sur une décomposition comme $HBFS$. Finalement, il semble nécessaire de donner des commentaires additionnelles sur la comparaison entre les temps d'exécution respectifs des différentes approches. En fait, une méthode peut être considérée comme plus efficace si son temps d'exécution est meilleur. Cependant, le nombre d'instances résolues doit être pris en compte. Plus précisément, nous considérons la comparaison des algorithmes ($HBFS$, $BTD\text{-}HBFS$, $BTD\text{-}HBFS{+}DYN$) et des décompositions ($Min\text{-}Fill$, $Min\text{-}Fill^4$, H_2, H_3, H_5^{25}, $H_5^{5\%}$) donnée dans les tableaux 5.1 et 5.2. Lorsque nous rapportons que $BTD\text{-}HBFS$ utilisant $Min\text{-}Fill$ résout 1 712 instances en 26 291 s, tandis que $BTD\text{-}HBFS{+}DYN$ utilisant H_5^{25} résout 2 039 instances en 52 304 s, nous pourrions être intéressés par le benchmark résolu par au moins une méthode parmi les deux méthodes qui incluent 2 049 instances. Nous pouvons constater que $BTD\text{-}HBFS$ exploitant $Min\text{-}Fill$ a résolu seulement 1 712 instances parmi les 2 049 instances en consommant 430 691 s. En effet, nous ajoutons ici le coût des 337 instances non résolues ($2\,049 - 1\,712 = 337$) en 20 minutes, qui est de 404 400 s, qui doit être ajouté aux 26 291 s utilisés pour résoudre les 1 712 instances. En ce qui concerne $BTD\text{-}HBFS{+}DYN$ utilisant H_5^{25}, après l'ajout du coût des 10 instances non résolues parmi les 2 049 instances, nous obtenons un total de 64 304 s. Ce faisant, nous déduisons que $BTD\text{-}HBFS{+}DYN$ (utilisant H_5^{25}) est 6,7 fois plus rapide que $BTD\text{-}HBFS$ (utilisant $Min\text{-}Fill$) tandis qu'il résout 327 instances additionnelles.

5.5 Conclusion

Dans le chapitre 4, nous avons proposé une exploitation dynamique de la décomposition arborescente pour la résolution des instances CSP. Cette dynamicité a été implémentée via la fusion des clusters. La fusion dynamique a permis d'améliorer significativement l'efficacité de la résolution en augmentant le nombre d'instances résolues et en diminuant en général le temps de résolution de chaque instance. Le succès du schéma dynamique dans le cadre du problème CSP nous a incité à proposer une gestion dynamique de la décomposition pour la résolution d'instances WCSP.

Comme pour le problème CSP, la décomposition arborescente a été déjà exploitée pour la résolution d'instances WCSP. La qualité souvent médiocre des décompositions utilisées a été un frein à leur exploitation pour la résolution. La proposition du nouveau cadre généraliste de calcul de décompositions $H\text{-}TD$ a été sensiblement bénéfique pour la résolution. Il n'a cependant pas permis de libérer suffisamment l'heuristique de choix de variables. Or, la liberté de l'heuristique de choix de variables joue un rôle essentiel dans l'augmentation de l'efficacité de la résolution. Les algorithmes de résolution non structurels comme $HBFS$ jouissent d'une liberté totale pour le choix de la prochaine variable. Ceci leur permet notamment de profiter pleinement des approches adaptatives dont l'intérêt peut être majeur.

Pour y remédier, nous avons proposé un nouvel algorithme de résolution d'instances WCSP qui vise à rendre l'exploitation de la décomposition moins contrainte. L'idée consiste à n'utiliser la décomposition sur un sous-problème que lorsque la résolution sans décomposition semble difficile. Elle sera ainsi utilisée « si besoin ». Cela évite de restreindre inutilement la liberté de l'heuristique de choix de variables pour des sous-problèmes « faciles ». En effet, la décomposition est habituellement utilisée pour pouvoir résoudre des instances difficiles du point de vue d'un algorithme tel que $HBFS$.

L'utilisation « si besoin » de la décomposition a rendu $BTD\text{-}HBFS$ plus performant. Elle a augmenté le nombre d'instances résolues et a significativement diminué le temps de résolution pour certaines instances. L'adoption du couplage de $BTD\text{-}HBFS{+}DYN$ et de H_5^{25} a clairement montré son intérêt lors de sa comparaison à $BTD\text{-}HBFS$ avec $Min\text{-}Fill^4$ ou à $HBFS$, les deux méthodes de l'état de l'art. Il a également montré qu'en pratique, l'algorithme peut se comporter comme un simple $HBFS$ notamment pour les instances faciles, avoir un comportement intermédiaire entre $HBFS$ et $BTD\text{-}HBFS$ ou même se comporter comme un $BTD\text{-}HBFS$ classique. Les expérimentations montrent finalement qu'en cas de dépassement du temps limite, $BTD\text{-}HBFS{+}DYN$ est généralement capable de fournir des bornes inférieures et supérieures de meilleure qualité que celles fournies par $HBFS$.

Finalement, toutes les modifications de la décomposition restent complètement transparentes à l'utilisateur et ne demande aucune intervention de sa part. Pour résoudre une instance donnée, il n'a pas désormais à choisir entre $HBFS$ ou $BTD\text{-}HBFS$ par exemple. En effet, si le solveur exploite l'algorithme $BTD\text{-}HBFS{+}DYN$, ce dernier s'occupera de trouver le bon compromis entre profiter pleinement la structure et avoir une liberté totale pour le choix de la prochaine variable.

Dans le prochain chapitre, nous nous intéressons à l'amélioration de l'exploitation de la décomposition arborescente pour la résolution du problème #CSP.

Chapitre 6

Amélioration des méthodes basées sur une décomposition dans le cas du problème #CSP

Sommaire

6.1 Introduction

La résolution du problème #CSP est extrêmement difficile sur le plan théorique comme sur le plan pratique. Ainsi, ce problème a été étudié depuis longtemps et reste l'objet de plusieurs travaux pendant les dernières années. D'un point de vue pratique, des méthodes de résolution ont été proposées. Cependant, en raison de la difficulté théorique et pratique de ce problème, la plupart des travaux se sont focalisés sur les méthodes qui se contentent d'approximer le nombre de solutions ou de fournir une borne inférieure sur le nombre de solutions du problème. En effet, il est souvent difficile de résoudre ces instances exactement ou, en d'autres termes, d'obtenir le nombre exact de leurs solutions. Au contraire, en exploitant certaines propriétés des instances, il est possible de proposer des méthodes exactes qui peuvent être efficaces en théorie comme en pratique. Notamment, dans ce chapitre, nous nous intéressons aux méthodes de recherche qui exploitent la structure de l'instance comme BTD. Contrairement à [Favier et al., 2009] qui utilise essentiellement $\#BTD$, l'adaptation de BTD au problème du comptage, telle une sous-routine pour une méthode d'approximation, nous visons dans ce chapitre à améliorer directement cette approche pour le comptage exact. En particulier, $\#BTD$ effectue beaucoup de calculs qui peuvent s'avérer inutiles dans le contexte du comptage. C'est ainsi que pour une affectation partielle donnée, $\#BTD$ peut compter le nombre de ses extensions cohérentes pour un sous-problème sans avoir la garantie que cette affectation possède au moins une extension cohérente sur tout le problème. Le fait que ces calculs inutiles peuvent être coûteux peut conduire à une dégradation de la performance de $\#BTD$. L'objectif du nouvel algorithme appelé $\#EBTD$ est alors de garantir, lors du comptage du nombre d'extensions d'une affectation partielle pour un sous-problème donné, que cette dernière admet au moins une extension cohérente sur tout le problème.

Le plan de ce chapitre est le suivant. Dans la section 6.2, nous évoquons comment les similitudes entre les deux problèmes CSP et #CSP ont permis d'étendre les méthodes de résolution pour le premier aux méthodes de résolution pour le second. Nous nous focalisons dans cette section sur la méthode BTD qui a été adaptée en $\#BTD$ pour le problème #CSP et nous montrons un inconvénient de cette méthode. Ensuite, dans la section 6.3, nous décrivons les modifications apportées à $\#BTD$ et nous expliquons l'algorithme $\#EBTD$. Nous illustrons par la suite son intérêt par une étude expérimentale avant de conclure.

6.2 Similitudes entre les problèmes CSP et #CSP

Trivialement, les deux problèmes CSP pour la décision et #CSP pour le comptage se rapprochent du fait de la nature de la question à laquelle ils répondent. Étant donnée une instance, le problème de décision CSP consiste à *dire si cette instance possède une solution*. Quant au problème du comptage #CSP, il vise à *compter le nombre de solutions de cette instance*. Évidemment, si nous ne sommes pas capables de décider si une instance a une solution, nous ne serons pas en mesure de compter son nombre de solutions. De même, si nous pouvons compter efficacement le nombre de solutions d'une instance, nous répondons automatiquement à la question de la cohérence de l'instance.

La relation étroite entre ces deux problèmes a permis d'étendre les méthodes de résolution du problème CSP naturellement aux méthodes de comptage. Ainsi, les méthodes énumératives classiques de résolution du problème CSP telles que MAC et RFL ont été étendues et exploitées pour le comptage du nombre de solutions d'une instance. Dans ce cas, elles explorent l'espace de recherche afin d'énumérer l'ensemble des solutions du

problème et ne se contentent pas de la détection d'une seule solution. La façon dont ces méthodes procèdent induit une redondance des sous-espaces de recherche visités. Ces redondances sont encore plus pénalisantes dans le cadre du problème du comptage puisque l'effort effectué pour chaque sous-problème est plus important vu que l'espace de recherche visité serait potentiellement plus grand que dans le cas du problème de décision. De ce fait, ces méthodes ne sont pas efficaces en pratique notamment pour les problèmes ayant un très grand nombre de solutions (sur le benchmark auquel nous allons nous intéresser dans ce chapitre certaines instances possèdent un nombre de solutions de l'ordre de 10^{250} solutions). Dans ce cas, la capacité de l'énumération est dépassée.

Quant aux méthodes structurelles, elles ont été adaptées au problème #CSP. Leur intérêt réside dans leur aptitude à fournir des méthodes exactes de comptage qui ont une complexité théorique en temps beaucoup plus intéressante que celle des méthodes énumératives, exponentielle en n. En pratique, ces méthodes ont également prouvé une amélioration importante en termes d'efficacité. Dans cet esprit, dans [Favier et al., 2009], il a été proposé d'adapter la méthode BTD au problème #CSP aboutissant ainsi à une méthode appelée $\#BTD$. $\#BTD$ a recours à l'enregistrement à la façon de BTD ce qui le rend efficace en pratique contrairement aux méthodes classiques. En effet, BTD enregistre pour chaque sous-problème induit par une affectation donnée un (no)good structurel qui indique si cette affectation admet une extension sur ce sous-problème. Quant à $\#BTD$, c'est le nombre d'extensions de cette affectation qui sera enregistré. Si ultérieurement pendant la recherche, cette même affectation est réalisée, $\#BTD$ réutilise le nombre de solutions enregistré pour le sous-problème déjà visité et par voie de conséquence évite certaines redondances.

Nous nous focalisons dans la partie suivante sur $\#BTD$ et nous expliquons comment cet algorithme compte le nombre de solutions des instances CSP grâce à la décomposition arborescente de leur (hyper)graphe de contraintes.

6.2.1 Adaptation de BTD à #BTD

La méthode $\#BTD$ se base sur les mêmes principes que la méthode BTD utilisée pour résoudre le problème CSP. Elle exploite une décomposition calculée préalablement avant le début de la recherche. Dans la suite de ce chapitre, nous nous basons sur l'exemple de la décomposition de la figure 6.1 pour illustrer le comportement de l'algorithme $\#BTD$ et plus tard celui de l'algorithme $\#EBTD$.

$\#BTD$, à l'instar de BTD, exploite la propriété essentielle de la décomposition arborescente. En effet, le fait d'assigner les variables d'un séparateur entre deux clusters de la décomposition sépare le problème initial en deux problèmes, qui peuvent être résolus indépendamment. Si le séparateur en question sépare deux clusters E_i et E_j de la décomposition (avec E_j le cluster fils de E_i), le premier sous-problème est celui enraciné en E_j. Il contient l'ensemble des variables des clusters de $Desc(E_j)$ (noté $V_{Desc(E_j)}$) et est noté $P_j|\mathcal{A}[E_i \cap E_j]$ (ou simplement P_j lorsqu'il n'y a pas d'ambiguïté). En outre, à l'image de BTD, la décomposition induit un ordre de choix de variables partiel comme cela a été décrit dans la sous-section 4.2.2. En particulier, si le cluster racine $E_r = E_g$ dans la figure 6.1, l'ordre partiel de choix de variables est : $\Lambda = [\{x_1, x_4, x_5\}, \{\{x_2, x_3\}, [\{x_8, x_9\},$ $\{\{x_6, x_7\}, [\{x_{10}, x_{11}\}, \{x_{12}, x_{13}, x_{14}\}], [\{x_{15}, x_{16}\}, \{x_{17}, x_{18}, x_{19}\}]\}]\}]$. Lorsqu'un cluster E_i est totalement assigné, pour chaque cluster E_j dans $Fils(E_i)$, le sous-problème $P_j|\mathcal{A}[E_i \cap E_j]$ est résolu indépendamment. Comme dans le cas du problème de décision où BTD enregistre le résultat de l'exploration du problème $P_j|\mathcal{A}[E_i \cap E_j]$ (enregistrement de *goods* et de *nogoods*), $\#BTD$ évite certaines redondances en enregistrant également les informations nécessaires. En particulier, $\#BTD$ enregistre le nombre exact de solutions de

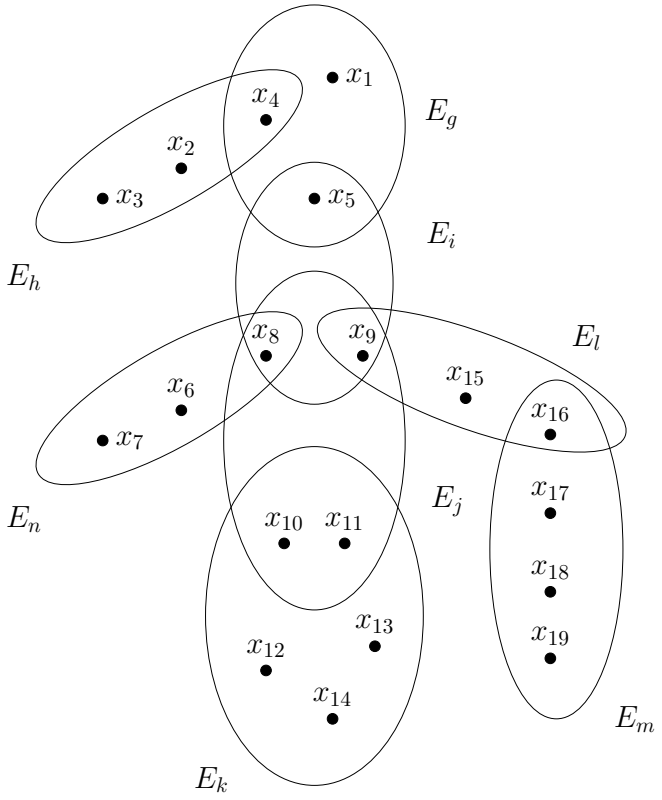


FIGURE 6.1 – L'ensemble des clusters d'une décomposition.

$P_j|\mathcal{A}[E_i \cap E_j]$, noté $\#sol_{E_j}$, comme un *#good structurel*($\mathcal{A}$,$\#sol_{E_j}$). Ce faisant, le nombre de solutions ne sera jamais recalculé pour la même affectation $E_i \cap E_j$. C'est ainsi que $\#BTD$, comme BTD, est capable de maintenir une complexité exponentielle en fonction de la taille du plus grand cluster de la décomposition.

$\#BTD$ est décrit dans l'algorithme 6.1. Étant donnés une affectation $\mathcal{A}$ et un cluster E_i, $\#BTD$ cherche le nombre d'extensions cohérentes $\mathcal{B}$ de $\mathcal{A}$ sur $Desc(E_i)$ telles que $\mathcal{A}[E_i \backslash V_{E_i}] = \mathcal{B}[E_i \backslash V_{E_i}]$. Le premier appel réalisé est $\#BTD(P, (E, T), \emptyset, E_r, E_r, \emptyset)$ et retourne le nombre de solutions du problème global. $\#BTD$ commence par assigner les variables du cluster racine. Si, dans l'exemple de la figure 6.1, $E_r = E_g$, les premières variables à assigner sont les variables x_1, x_4 et x_5. Au sein de chaque cluster à affecter, $\#BTD$ procède classiquement en assignant une valeur à une variable et en faisant un retour-arrière si la recherche rencontre une incohérence (lignes 15-22). Des algorithmes tels que $\#BT$, $\#MAC$ ou $\#RFL$ peuvent être utilisés. La présentation est basée sur $\#BT$. Soit E_i le cluster courant. Une fois toutes les variables de E_i instanciées de façon cohérente (ligne 1), $\#BTD$ calcule le nombre de solutions du sous-problème induit par chaque cluster fils de E_i, s'il en a (lignes 2-12). Dans le cas de la figure 6.1, E_i possède 3 fils : E_n, E_j et E_l. Considérons par exemple le cluster fils E_j. Étant donnée une affectation courante $\mathcal{A}$ de E_i, $\#BTD$ vérifie d'abord si l'affectation $\mathcal{A}[E_i \cap E_j]$ correspond à un #good (ligne 7). Si $\mathcal{A}[E_i \cap E_j]$ est effectivement un #good, $\#BTD$ multiplie le nombre de solutions enregistré avec le nombre de solutions de E_i avec $\mathcal{A}$ comme affectation (ligne 8). Sinon, $\#BTD$ étend $\mathcal{A}$ sur les variables restantes de $Desc(E_j)$ dans le but de calculer le nombre d'extensions cohérentes $\#sol_{E_j}$ (ligne 10). Les variables impliquées dans le cas de la figure 6.1 sont x_{10}, x_{11}, x_{12}, x_{13} et x_{14}. Ensuite, il enregistre le #good $(A[E_i \cap E_j], \#sol_{E_j})$ (ligne 11). Après, $\#BTD$ calcule le nombre de solutions de chaque sous-problème induit par le

Algorithme 6.1 : #BTD (P, (E,T), $\mathcal{A}$, E_i, V_{E_i}, G^d)

Entrées : Une instance CSP $P = (X, D, C)$, (E,T) la décomposition arborescente, l'affectation courante $\mathcal{A}$, le cluster courant E_i, l'ensemble V_{E_i} des variables non assignées de E_i
Entrées-Sorties : L'ensemble G^d de #goods enregistrés
Sorties : Le nombre de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$

```
1  si V_{E_i} = ∅ alors
2      Q_{E_i} ← Fils(E_i)
3      #sol ← 1
4      tant que Q_{E_i} ≠ ∅ et #sol ≠ 0 faire
5          Choisir un cluster E_j ∈ Q_{E_i}
6          Q_{E_i} ← Q_{E_i} \ {E_j}
7          si (A[E_i ∩ E_j], #sol_{E_j}) est un #good dans G^d alors
8              #sol ← #sol × #sol_{E_j}
9          sinon
10             #sol_{E_j} ← #BTD(P, (E,T), A,E_j, V_{E_j} \ (E_i ∩ E_j),G^d)
11             Enregistrer (A[E_i ∩ E_j], #sol_{E_j}) comme #good de E_i par rapport à E_j
                dans G^d
12             #sol ← #sol × #sol_{E_j}
13     retourner #sol
14 sinon
15     Choisir x ∈ V_{E_i}
16     d ← D_x
17     #sol_x ← 0
18     tant que d ≠ ∅ faire
19         Choisir v ∈ d
20         d ← d − {v}
21         si A ∪ {x ← v} satisfait toutes les contraintes de C alors
22             #sol_x ← #sol_x+#BTD(P, (E,T), A ∪ {x ← v}, E_i, V_{E_i} \ {x},G^d)
23     retourner #sol_x
```

prochain fils de E_i (les clusters E_l et E_n dans notre exemple). Finalement, lorsque chaque cluster fils de E_i est examiné, $\#BTD$ essaye de modifier l'affectation courante de E_i. Le nombre de solutions au niveau de E_i est la somme du nombre de solutions pour chaque affectation cohérente de E_i.

Les complexités en temps et en espace de $\#BTD$ sont les mêmes que pour BTD, à savoir $O(n.w^+.log(d).d^{w^+ +1})$ et $O(n.s.d^s)$ respectivement [Favier et al., 2009].

$\#BTD$ a permis d'exploiter la structure de l'instance et de bénéficier des enregistrements réalisés au niveau des séparateurs pour rendre les méthodes de comptage plus efficaces. Cependant, dans [Favier et al., 2009], les expérimentations ont montré qu'en pratique, $\#BTD$ dépasse souvent la limite du temps ou de mémoire. En effet, la manière dont $\#BTD$ procède pour calculer le nombre de solutions de P présente un défaut majeur que nous détaillons dans la partie suivante.

6.2.2 Inconvénient de #BTD

La façon dont $\#BTD$ procède pour calculer le nombre de solutions d'un problème peut effectivement engendrer des calculs coûteux et inutiles. L'idée de base est qu'étant donnée une affectation partielle $\mathcal{A}$, un appel à $\#BTD$, comme celui de $\#sol_{E_j} \leftarrow \#\text{BTD}(P, (E,T), A, E_j, V_{E_j} \backslash (E_i \cap E_j))$ (ligne 10), compte toutes les solutions du sous-problème courant. Pour autant BTD n'a pas la garantie que l'affectation partielle s'étend à une solution sur tout le problème.

Nous illustrons cet inconvénient de $\#BTD$ sur l'exemple de la figure 6.1. Soit $E_r = E_g$ le cluster racine de la décomposition par lequel $\#BTD$ débute le comptage. Supposons que E_i est le cluster courant et qu'il vient d'être instancié d'une façon cohérente (E_g et E_h sont déjà instanciés). $\#BTD$ traite alors les clusters fils de E_i l'un après l'autre en calculant le nombre exact de solutions correspondant à chaque sous-problème enraciné en chaque cluster fils. Il considère, par exemple, le cluster E_j et calcule le nombre exact de solutions du problème $P_j|\mathcal{A}[E_i \cap E_j]$, puis considère le cluster E_l et calcule celui de $P_l|\mathcal{A}[E_i \cap E_l]$ et enfin considère le cluster E_n afin de calculer celui de $P_n|\mathcal{A}[E_i \cap E_n]$. L'inconvénient d'une telle approche est que toutes les solutions de $P_j|\mathcal{A}[E_i \cap E_j]$ sont calculées avant de s'assurer qu'il existe au moins une solution pour $P_l|\mathcal{A}[E_i \cap E_l]$ et pour $P_n|\mathcal{A}[E_i \cap E_n]$. Plus précisément, le nombre de solutions d'un sous-problème donné est calculé avant de vérifier que l'affectation courante peut s'étendre en une solution globale, c'est-à-dire une affectation cohérente de toutes les variables du problème. En effet, dans le pire des cas, $\#BTD$ pourrait calculer le nombre de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$ et de $P_l|\mathcal{A}[E_i \cap E_l]$ avant d'établir que $P_n|\mathcal{A}[E_i \cap E_n]$ ne possède aucune solution. Dans ce cas, le nombre de solutions calculé et enregistré pour les sous-problèmes $P_j|\mathcal{A}[E_i \cap E_j]$ et $P_l|\mathcal{A}[E_i \cap E_l]$ s'avère inutile (sauf si le #good enregistré pour un sous-problème est utilisé ultérieurement). La situation devient de plus en plus pénalisante pour $\#BTD$ lorsque le nombre de sous-problèmes examinés avant l'exploration du sous-problème pour lequel il n'existe aucune extension cohérente augmente. Pour résumer, $\#BTD$ n'exploite pas le fait qu'il est inutile de compter le nombre d'extensions d'une affectation sur un sous-problème si cette affectation ne s'étend pas en une solution globale, c'est-à-dire une solution du point de vue du problème de décision. Ce principe s'applique aussi localement pour un sous-problème en particulier. C'est ainsi que pour compter le nombre d'extensions cohérentes d'une affectation donnée de E_i sur $Desc(E_i)$, il est inutile de compter le nombre d'extensions cohérentes de cette affectation pour un fils donné de E_i si elle n'admet pas au moins une extension sur toutes les variables de $Desc(E_i)$.

En conséquence, cet inconvénient peut détériorer l'efficacité de $\#BTD$ vu que des sous-espaces de recherche sont explorés inutilement ce qui consomme du temps mais aussi de l'espace mémoire en enregistrant des informations qui ne seront éventuellement pas réutilisées par la suite. Ainsi, une première possibilité pour améliorer $\#BTD$ consiste à éviter ces recherches inutiles. Dans la section suivante, nous exploitons cette idée dans le but de définir un algorithme plus sophistiqué que nous appellerons $\#EBTD$. Ce travail a fait l'objet de la publication [Jégou et al., 2016a].

6.3 Recherche plus adaptée au comptage : #EBTD

Dans cette section, nous détaillons l'algorithme $\#EBTD$ et nous montrons comment il est capable d'améliorer la recherche faite par $\#BTD$. Notons que si l'idée sur laquelle se base $\#BTD$ est simple et naturelle, sa mise en œuvre est complexe.

6.3.1 Comptage aveugle vs comptage conscient

Tout d'abord, nous illustrons l'objectif de $\#EBTD$.

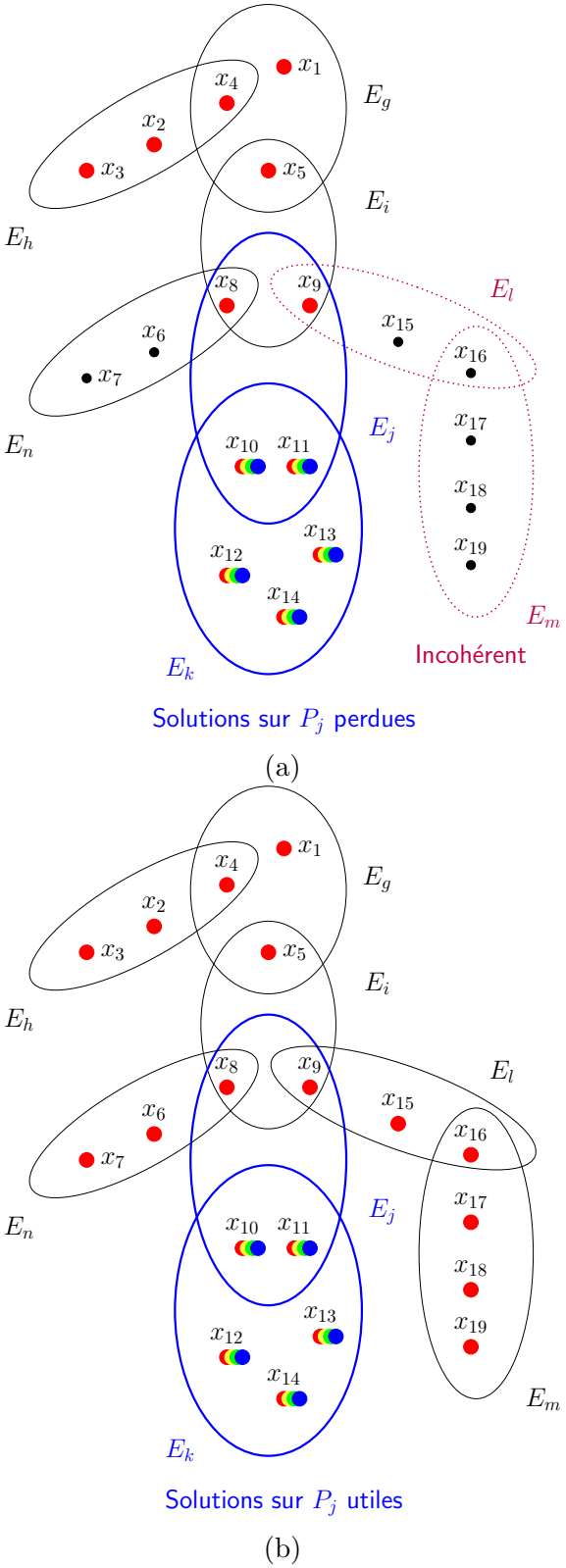


FIGURE 6.2 – Illustration du comptage aveugle (a) et du comptage conscient (b).

Dans la figure 6.2, les variables affectées sont colorées en rouge. Les clusters E_j et E_k du sous-problème P_j pour lequel nous avons compté le nombre d'extensions cohérentes de l'affectation du séparateur $E_i \cap E_j$ sont représentés en bleu. Les différentes solutions sont colorées en : rouge, jaune, vert et bleu. Nous distinguons deux cas de figure :

- Pour la figure 6.2(a), après l'affectation des variables x_1, x_2, x_3, x_4, x_5, x_8 et x_9, nous comptons le nombre d'extensions cohérentes de $\mathcal{A}[\{x_8, x_9\}]$ pour P_j. C'est ce que nous appelons le *comptage aveugle*.
- En ce qui concerne la figure 6.2(b), le comptage du nombre d'extensions cohérentes de $\mathcal{A}[\{x_8, x_9\}]$ pour P_j n'est réalisé qu'après l'affectation de toutes les variables du problème de façon cohérente. Nous parlons ici du *comptage conscient*.

La différence entre les deux cas est que dans la figure 6.2(a), le calcul de $\#sol_{E_j}$ est fait sans que la présence d'une solution globale ne soit garantie. En revanche, dans la figure 6.2(b), le calcul de $\#sol_{E_j}$ n'est réalisé que lorsque l'affectation $\mathcal{A}[\{x_8, x_9\}]$ est garantie extensible de façon cohérente sur le reste des variables du problème. Dans la figure 6.2(a), une fois le nombre de solutions calculé sur P_j, une incohérence est détectée pour P_l. En effet, aucune extension cohérente n'est trouvée sur ce sous-problème. Ainsi, le nombre de solutions trouvé pour P_j s'avère inutile. Cependant, le nombre de solutions trouvé sur P_j dans la figure 6.2(b) est utile et participe au calcul du nombre de solutions de P.

L'objectif de $\#EBTD$ est d'imiter le cas de la figure 6.2(b) et de *garantir lors du comptage du nombre d'extensions d'une affectation sur un sous-problème que cette même affectation est extensible de façon cohérente sur les variables du reste du problème.*

6.3.2 Extension du type d'enregistrements

À l'instar de $\#BTD$, $\#EBTD$ (pour *Enhanced Backtracking with Tree-Decomposition*), est basé sur la notion de la décomposition arborescente des graphes. $\#EBTD$ accomplit aussi une recherche basée sur le retour-arrière en utilisant un ordre de variables compatible avec la décomposition. La première différence avec $\#BTD$ réside dans la définition du concept de goods qui est étendu ainsi :

Définition 64 (Goods structurels exacts et partiels) *Soient (E, T) une décomposition arborescente, E_i et E_j deux clusters de E avec E_j un cluster fils de E_i et $\mathcal{A}$ une affectation cohérente de $E_i \cap E_j$. Un* good structurel exact *est un triplet ($\mathcal{A}$,=,$\#sol_{E_j}$) avec $\#sol_{E_j}$ le nombre exact de solutions de $P_j|\mathcal{A}$. Un* good structurel partiel *est un triplet ($\mathcal{A}$,$\geq$,$\#sol_{E_j}$) avec $\#sol_{E_j}$ une borne inférieure sur le nombre de solutions de $P_j|\mathcal{A}$.*

Notons que les goods exacts structurels ($\mathcal{A}$,=,$\#sol_{E_j}$) sont identiques aux #good($\mathcal{A}$,$\#sol_{E_j}$) structurels exploités dans $\#BTD$ [Favier et al., 2009]. En outre, nous exploitons aussi la notion du nogood structurel :

Définition 65 (nogood structurel [Jégou and Terrioux, 2003]) *Soient (E, T) une décomposition arborescente et E_i et E_j deux clusters de E tel que E_j est un cluster fils de E_i. Un* nogood structurel *de E_i par rapport à E_j est une affectation cohérente $\mathcal{A}$ des variables de $E_i \cap E_j$ de sorte que $P_j|\mathcal{A}$ n'admet aucune solution.*

Bien qu'un nogood structurel soit équivalent à un good exact tel que le nombre de solutions attribué est nul, l'exploitation d'un nogood s'avère plus efficace. En effet, un nogood est exploité *dès que possible* et permet de détecter une incohérence au plus tôt rendant la résolution plus efficace.

Algorithme 6.2 : #EBTD (P, (E,T), $\mathcal{A}$, E_i, V_{E_i}, Z, G^d, N^d)

Entrées : Une instance CSP $P = (X, D, C)$, une décomposition arborescente (E, T), l'affectation courante $\mathcal{A}$, le cluster courant E_i, l'ensemble V_{E_i} des variables non instanciées de E_i

Entrées-Sorties : Z une pile de clusters, l'ensemble G^d de goods enregistrés, l'ensemble N^d de nogoods enregistrés

Sorties : (Le nombre de solutions trouvées pour $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$, le cluster vers lequel nous faisons un retour-arrière)

```
si V_{E_i} = ∅ alors
    #sol ← 1
    Q_{E_i} ← Fils(E_i)
    Z_{good≥} ← ∅
    Z_{inconnu} ← ∅
    tant que Q_{E_i} ≠ ∅ faire
        Choisir un cluster E_j ∈ Q_{E_i}
        Q_{E_i} ← Q_{E_i}\{E_j}
        suivant A[E_i ∩ E_j] faire
            cas où good (A[E_i ∩ E_j], =, #sol_{E_j}) de E_i vis-à-vis de E_j dans G^d faire
                #sol ← #sol * #sol_{E_j}
            cas où good (A[E_i ∩ E_j], ≥, #sol_{E_j}) de E_i vis-à-vis de E_j dans G^d faire
                Z_{good≥} ← Z_{good≥} ∪ {E_j}
            autres cas faire
                Z_{inconnu} ← Z_{inconnu} ∪ {E_j}
                Z ← Z ∪ {E_j}
    si Z ≠ ∅ alors
        E_j ← Premier(Z)
        Z ← Z\{E_j}
        (#sol_{E_j}, E_bt) ← #EBTD(P,(E,T),A,E_j,E_j\(E_{p(j)} ∩ E_j), Z,G^d,N^d)
        si #sol_{E_j} > 0 alors
            si E_bt = E_j alors
                Enregistrer (A[E_{p(j)} ∩ E_j], =, #sol_{E_j}) comme good de E_{p(j)} vis-à-vis de E_j dans G^d
            sinon
                Enregistrer (A[E_{p(j)} ∩ E_j], ≥, #sol_{E_j}) comme good de E_{p(j)} vis-à-vis de E_j dans G^d
                si E_i = E_bt alors
                    Z ← Z\Fils(E_i)
                    retourner (0, E_i)
                sinon retourner (#sol, E_bt)
        sinon
            Enregistrer A[E_{p(j)} ∩ E_j] comme nogood de E_{p(j)} vis-à-vis de E_j dans N^d
            si E_i = E_{p(j)} alors
                Z ← Z\Fils(E_i)
                retourner (0, E_i)
            sinon retourner (#sol, E_{p(j)})
    pour chaque E_j ∈ Z_{inconnu} faire
        #sol ← #sol * #sol_{E_j}
    tant que Z_{good≥} ≠ ∅ faire
        Choisir un cluster E_j ∈ Z_{good≥}
        Z_{good≥} ← Z_{good≥}\{E_j}
        (#sol_{E_j}, E_bt) ← #EBTD(P,(E,T),A,E_j,E_j\(E_i ∩ E_j), Z,G^d,N^d)
        Enregistrer (A[E_i ∩ E_j], =, #sol_{E_j}) comme good de E_i vis-à-vis de E_j dans G^d
        #sol ← #sol * #sol_{E_j}
    retourner (#sol, E_i)
sinon
    Choisir x ∈ V_{E_i}
    d ← D_x
    #sol_x ← 0
    E_bt ← E_i
    répéter
        Choisir v ∈ d
        d ← d − {v}
        si A ∪ {x ← v} satisfait toutes les contraintes de C ∪ N^d alors
            (#sol_{xv}, E_bt) ← #EBTD(P,(E,T),A ∪ {x ← v},E_i, V_{E_i}\{x},Z,G^d,N^d)
            #sol_x ← #sol_x + #sol_{xv}
    jusqu'à d = ∅ ou x ∉ E_bt
    retourner (#sol_x,E_bt)
```

6.3.3 Description de l'algorithme #EBTD

6.3.3.1 Entrées et Sorties

$\#EBTD$ est décrit dans l'algorithme 6.2. Étant données une instance CSP $P = (X, D, C)$ et une décomposition arborescente (E, T), l'appel $\#EBTD$ (P, (E, T), $\mathcal{A}$, E_i, V_{E_i}, Z, G^d, N^d) vise à calculer le nombre de solutions du sous-problème $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$ avec $\mathcal{A}$ l'affectation courante, E_i le cluster courant et V_{E_i} l'ensemble de variables non assignées de E_i. G^d et N^d représentent respectivement l'ensemble de goods (exacts et partiels) et de nogoods structurels qui sont enregistrés durant la recherche. L'algorithme exploite aussi une pile Z qui contient les prochains clusters à traiter. $\#EBTD$ retourne une paire $(\#sol, E_{bt})$. E_{bt} représente le cluster vers lequel $\#EBTD$ fait un retour-arrière ce qui permet de réaliser un saut au cluster pertinent lorsqu'un nogood est trouvé. Concernant le nombre de solutions $\#sol$, nous distinguons trois cas possibles :

- Si $\#sol > 0$ et $E_{bt} = E_i$, alors $\#sol$ est le nombre exact de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$. Ce cas se produit lorsque l'affectation $\mathcal{A}[E_i \backslash V_{E_i}]$ possède au moins une extension cohérente sur tout le problème.

- Si $\#sol > 0$ et $E_{bt} \neq E_i$, alors $\#sol$ est une borne inférieure du nombre de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$. Ce cas se produit lorsque l'affectation $\mathcal{A}[E_i \backslash V_{E_i}]$ admet au moins une extension cohérente sur P_i alors qu'il existe un cluster E_k tel que $\mathcal{A}[E_k \cap E_{p(k)}]$ est un nogood.

- Si $\#sol = 0$ alors $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$ ne possède aucune solution. Ainsi, $\mathcal{A}[E_i \backslash V_{E_i}]$ ne possède aucune extension cohérente sur P_i.

La pile Z résultante peut être définie comme étant l'ensemble de clusters E_j tels que $E_{p(j)}$ est totalement instancié de façon cohérente et tel que $\mathcal{A}[E_{p(j)} \cap E_j]$ est de nature inconnue (i.e. ne correspond ni à un good exact ni à un good partiel de $E_{p(j)}$ par rapport à E_j). Notons qu'elle est nécessairement vide dans le cas où $\#sol > 0$ et $E_{bt} = E_i$. En effet, d'après la spécification de $\#EBTD$, ce dernier ne calcule le nombre exact de solutions d'un sous-problème induit par une affectation $\mathcal{A}$ que lorsque cette affectation s'étend de façon cohérente au reste du problème comme dans le cas de la figure 6.2(b). D'après la définition de la pile Z, cette dernière contient les clusters E_k tel que $\mathcal{A}[E_k \cap E_{p(k)}]$ ne correspond pas à un good. Or, comme pour chaque cluster E_k, l'affectation $\mathcal{A}[E_k \cap E_{p(k)}]$ est garantie extensible de façon cohérente sur $Desc(E_k)$, aucun cluster ne serait inclus dans Z.

6.3.3.2 Similitudes avec #BTD

L'appel initial est $\#EBTD(P, (E, T), \mathcal{A}, E_r, E_r, Z, G^d, N^d)$ où Z, G^d et N^d sont vides. À la façon de $\#BTD$, $\#EBTD$, démarre sa recherche en assignant d'une manière cohérente les variables du cluster racine. Nous considérons par la suite la décomposition de la figure 6.1 avec $E_r = E_g$. Les premières variables à assigner sont alors x_1, x_4 et x_5. Par la suite, $\#EBTD$ explore les clusters fils du cluster courant. Lorsque $\#EBTD$ explore un nouveau cluster E_i, vu que les variables de son cluster parent $E_{p(i)}$ sont déjà instanciées (et ainsi celles de son séparateur), il doit uniquement instancier les variables qui apparaissent dans $E_i \backslash (E_i \cap E_{p(i)})$. Par exemple, si nous considérons le cluster E_i de la figure 6.1, vu que son instanciation se fait après celle de E_g, les seules variables restant à affecter sont x_8 et x_9. Dans le but de résoudre chaque cluster (lignes 46-57), $\#EBTD$ (comme $\#BTD$) peut exploiter n'importe quel algorithme qui n'altère pas la structure. Par souci de simplicité,

la version présentée dans l'algorithme 6.2 est basée sur un simple algorithme de retour-arrière chronologique (BT). Toutefois, dans les expérimentations fournies dans la section suivante, nous considérons un algorithme plus puissant qui est RFL. Ainsi, $\#EBTD$ choisit en premier la prochaine variable x à assigner parmi les variables de V_{E_i} (ligne 46). Après, il sélectionne une valeur v de D_x selon l'heuristique de choix de valeur (lignes 51-52) et l'assigne à la variable x. Si la nouvelle affectation est cohérente vis-à-vis des contraintes initiales de C et des nogoods structurels de N^d (ligne 53), la recherche continue en choisissant une nouvelle variable et en effectuant une nouvelle affectation. Sinon, $\#EBTD$ essaye d'assigner à x une autre valeur de D_x. Lorsque toutes les valeurs possibles sont essayées, $\#EBTD$ fait un retour-arrière.

Lorsque $\#EBTD$ a instancié d'une façon cohérente toutes les variables du cluster courant et du moment où il est certain que cette affectation admet une extension cohérente sur tout le problème, il vise ensuite à compter le nombre de solutions de chaque sous-problème enraciné en chaque cluster fils du cluster courant comme ferait $\#BTD$. Plus précisément, si E_i est, par exemple, le cluster courant dans la figure 6.1 (E_g et E_h sont déjà instanciés), son but est de calculer le nombre de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$, de $P_l|\mathcal{A}[E_i \cap E_l]$ et de $P_n|\mathcal{A}[E_i \cap E_n]$ vu que E_i possède dans ce cas trois fils E_j, E_l et E_n. Lorsque le nombre de solutions exact de chaque sous-problème enraciné en un cluster fils est calculé $\#EBTD$ enregistre un good exact (ligne 23). Ainsi, dans le cas de la figure 6.1, $\#EBTD$ enregistre les goods exacts $(\mathcal{A}[E_i \cap E_j], =, \#sol_{E_j})$, $(\mathcal{A}[E_i \cap E_l], =, \#sol_{E_l})$ et $(\mathcal{A}[E_i \cap E_n], =, \#sol_{E_n})$. Similairement, $\#BTD$ enregistrerait les #goods correspondants à chaque sous-problème (ligne 11), c'est-à-dire les #good $(\mathcal{A}[E_i \cap E_j], \#sol_{E_j})$, $(\mathcal{A}[E_i \cap E_l], \#sol_{E_l})$ et $(\mathcal{A}[E_i \cap E_n], \#sol_{E_n})$.

6.3.3.3 Modifications réalisées pour #EBTD

La différence principale entre $\#EBTD$ et $\#BTD$ réside dans l'exploration des clusters fils. Si dans l'exemple $\#BTD$ calcule consécutivement le nombre de solutions exact de $P_j|\mathcal{A}[E_i \cap E_j]$, $P_l|\mathcal{A}[E_i \cap E_l]$ et de $P_n|\mathcal{A}[E_i \cap E_n]$ (s'il visite les clusters fils dans l'ordre E_j, E_n puis E_l), $\#EBTD$ ne procède pas ainsi. Il vise au contraire à éviter les inconvénients de cette approche décrits dans la partie 6.2.2. $\#EBTD$ explore alors les clusters fils différemment. Plus précisément, il vise à garantir qu'il ne compte exactement le nombre de solutions du sous-problème enraciné en un cluster fils que s'il existe une solution globale du problème compatible avec l'affectation courante, c'est-à-dire une extension de l'affectation courante qui s'étend sur toutes les variables du problème. Ce faisant, le good exact ainsi enregistré est nécessairement utilisé au moins une fois. Pour illustrer ce principe, dans la figure 6.1, après l'affectation du cluster E_i, $\#EBTD$ essaye d'instancier, par ordre, les clusters E_j, E_k, E_l, E_m et E_n ($\#EBTD$ explore, dans ce cas, les clusters fils de E_i dans l'ordre E_j, E_l et E_n). Une fois tous les clusters instanciés, nous savons qu'une solution globale existe. C'est à ce moment que $\#EBTD$ calcule, $\#sol_{E_n}$, $\#sol_{E_l}$ et finalement $\#sol_{E_j}$. Au contraire, si après l'affectation des variables de $Desc(E_j)$ et de $Desc(E_l)$, celles de $Desc(E_n)$ ne peuvent pas être instanciées d'une façon cohérente, l'affectation $\mathcal{A}[E_i \cap E_n]$ est enregistrée comme un nogood de E_i par rapport à E_n tandis que $\mathcal{A}[E_i \cap E_l]$ et $\mathcal{A}[E_i \cap E_j]$ sont enregistrées comme des goods partiels (respectivement $(\mathcal{A}[E_i \cap E_l], \geq, 1)$ de E_i par rapport à E_l et $(\mathcal{A}[E_i \cap E_j], \geq, 1)$ de E_i par rapport à E_j). Le nombre de solutions associé à ces goods partiels est 1 puisqu'une seule solution a été trouvée pour chaque sous-problème. Ces (no)goods structurels peuvent être utilisés ultérieurement afin d'éviter des parties redondantes de l'arbre de recherche Ce principe est appliqué récursivement à tous les niveaux des clusters lors du calcul du nombre de solutions exact d'un sous-problème. Par exemple, si $E_r = E_g$ est le cluster courant qui vient d'être instancié, le nombre de

solutions exact de $P_i|\mathcal{A}[E_g \cap E_i]$ ne peut être calculé avant de s'être assuré qu'il existe au moins une extension cohérente pour $P_h|\mathcal{A}[E_g \cap E_h]$ et vice versa.

La condition $x \notin E_{bt}$ (ligne 56) suspend l'énumération des solutions relatives à E_i lorsqu'un nogood de $E_{p(\ell)}$ par rapport à E_ℓ est enregistré avec E_ℓ est un cluster exploré après E_i. Dans ce cas, $E_{bt} = E_{p(\ell)}$ et la recherche fait retour-arrière vers le cluster $E_{p(\ell)}$. Par exemple, dans la figure 6.1, si $\#EBTD$ assigne E_g puis les clusters de $Desc(E_i)$ mais ne réussit pas à étendre l'affectation pour les variables de $Desc(E_h)$, $\mathcal{A}[\{x_4\}]$ est enregistrée comme un nogood de E_g par rapport à E_h. Le test à la ligne 56 bloque ensuite l'énumération des extensions de $\mathcal{A}[\{x_5\}]$ sur E_i puisque dans ce cas $E_{bt} = E_g$ et ainsi le test échoue. Lorsque toutes les variables de E_i sont instanciées d'une façon cohérente (ligne 1), $\#EBTD$ considère chaque cluster fils de E_i (lignes 2-16). Si $\mathcal{A}[E_i \cap E_j]$ correspond à un exact good (ligne 10), le good est exploité et la recherche passe au prochain cluster fils. $\#EBTD$ utilise deux piles locales $Z_{inconnu}$ et $Z_{good_\geq}$. Il ajoute à Z et à $Z_{inconnu}$ chaque cluster fils E_j de E_i pour lequel $\mathcal{A}[E_i \cap E_j]$ ne correspond pas ni à un good, ni à un nogood de E_i par rapport à E_j. Au contraire, si $\mathcal{A}[E_i \cap E_j]$ correspond à un good partiel de E_i par rapport à E_j, E_j est ajouté à $Z_{good_\geq}$. $Z_{good_\geq}$ est l'ensemble de clusters fils E_j de E_i pour lesquels $\mathcal{A}[E_i \cap E_j]$ correspond à un good partiel de E_i par rapport à E_j avec une borne inférieure sur le nombre de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$. Pour chaque cluster de $Z_{good_\geq}$, le nombre exact de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$ est calculé ultérieurement (lignes 38-43) uniquement si cela est nécessaire, c'est-à-dire si l'affectation courante $\mathcal{A}$ a pu être étendue à une solution globale. En plus, pour chaque cluster de $Z_{inconnu}$, à la ligne 36, le nombre exact de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$ est garanti être calculé (lignes 17-35) et peut ainsi être exploité. Si la pile Z n'est pas vide (ligne 17), $\#EBTD$ continue la recherche sur le cluster E_j, premier élément de la pile Z (ligne 20). Si $\#EBTD$ réussit à étendre $\mathcal{A}$ au sous-problème enraciné en E_j, il enregistre $\mathcal{A}[E_{p(j)} \cap E_j]$ comme un good. Si $\mathcal{A}$ peut être étendue à une solution globale ($E_{bt} = E_j$), le nombre associé à ce good est le nombre exact de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$ et nous disposons d'un good exact (ligne 22-23). Sinon, il correspond à une borne inférieure du nombre de solutions et nous parlons d'un good partiel (lignes 25-29). Au contraire, si $\mathcal{A}$ n'a pas pu être étendue au sous-problème enraciné en E_j, $\mathcal{A}[E_{p(j)} \cap E_j]$ est enregistrée comme un nogood (lignes 31-35). Il est à noter que $E_{p(j)}$ n'est pas forcément E_i. Les lignes 27 et 33 permettent finalement de supprimer de la pile Z les fils du cluster E_{bt} (s'il y en a) en raison de l'enregistrement d'un nogood de E_{bt} par rapport à un de ses fils.

6.3.4 Fondements théoriques

Concernant $\#EBTD$, nous distinguons deux phases qui s'alternent :

- La première phase vise à s'assurer que pour une solution partielle, il existe une solution globale, c'est-à-dire une affectation qui s'étend de façon cohérente sur toutes les variables du problème.
- La deuxième consiste à énumérer, une fois la présence d'une solution globale garantie, l'ensemble des extensions cohérentes d'une affectation induisant un sous-problème pour ce dernier.

Ces deux phases garantissent que, chaque fois que le nombre de solutions d'un sous-problème est calculé, celui-ci est exploité pour le calcul du nombre de solutions du problème. Nous allons donc maintenant démontrer que $\#EBTD$ est capable de calculer le nombre exact de solutions d'un problème tout en garantissant d'éviter certains calculs inutiles au sens évoqué précédemment.

Théorème 13 *#EBTD est correct, complet et termine.*

Dans ce qui suit, nous exploitons les piles Z, $Z_{good_{\geq}}$ et $Z_{inconnu}$. Nous rappelons alors le contenu de celles-ci :

- Z est une pile globale contenant des clusters E_k à traiter tels que $\mathcal{A}[E_{p(k)} \cap E_k]$ ne correspond pas à un (no)good.
- $Z_{good_{\geq}}$ est une pile locale au cluster courant E_i qui contient chaque cluster fils E_j de E_i pour lequel $\mathcal{A}[E_i \cap E_j]$ correspond à un good partiel de E_i par rapport à E_j.
- $Z_{inconnu}$ est une pile locale au cluster courant E_i qui contient chaque cluster fils E_j de E_i pour lequel $\mathcal{A}[E_i \cap E_j]$ ne correspond pas à un good de E_i par rapport à E_j, ni à un nogood.

Preuve :
Soit $G_{=}(E_i)$ l'ensemble des affectations correspondant aux goods exacts de $E_p(i)$ par rapport à E_i dans G. Nous nous basons dans cette preuve sur l'ensemble de variables $VAR(V_{E_i}, Z, \mathcal{A}, G^d)$ défini par :

$$V_{E_i} \cup (\bigcup_{E_j \in Fils(E_i) | \mathcal{A}[E_i \cap E_j] \notin G_{=}(E_j)} V_{Desc(E_j)} \backslash (E_i \cap E_j)) \cup (\bigcup_{E_k \in Z} V_{Desc(E_k)} \backslash (E_{p(k)} \cap E_k)).$$

$VAR(V_{E_i}, Z, \mathcal{A}, G^d)$ contient les variables qui doivent être explorées par $\#EBTD$ afin de déterminer si une solution globale contenant $\mathcal{A}$ existe et le cas échéant de compter le nombre exact de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$. Plus précisément, $VAR(V_{E_i}, Z, \mathcal{A}, G^d)$ contient :

- V_{E_i} : l'ensemble de variables non assignées du cluster courant E_i.
- $V_{Desc(E_j)} \backslash (E_i \cap E_j)$: l'ensemble des variables de $Desc(E_j)$ sans compter les variables de $E_i \cap E_j$ tel que E_j est un cluster fils de E_i pour lequel $\mathcal{A}[E_i \cap E_j] \notin G_{=}(E_j)$. Si $\mathcal{A}[E_i \cap E_j] \in G_{=}(E_j)$, le good $\mathcal{A}[E_i \cap E_j]$ est exploité selon la définition d'un good exact et le sous-problème correspondant n'est pas visité. Si $\mathcal{A}[E_i \cap E_j]$ est un good partiel, le sous-problème enraciné en E_j n'est pas exploré lors de la recherche d'une solution globale (vu qu'au moins une extension cohérente existe pour ce sous-problème). Cependant, une fois une solution globale trouvée, il sera visité afin de calculer $\#sol_{E_j}$. Sinon, $\mathcal{A}[E_i \cap E_j]$ est inconnu et le sous-problème correspondant doit être exploré pour déterminer si une solution globale existe. Notons que dans le cas où $\mathcal{A}[E_i \cap E_j]$ est un nogood aucune solution globale contenant $\mathcal{A}$ n'existe et le sous-problème P_j ne sera pas visité.
- $V_{Desc(E_k)} \backslash (E_{p(k)} \cap E_k)$: l'ensemble des variables de $Desc(E_k)$ sans compter les variables de $E_{p(k)} \cap E_k$ tel que $E_k \in Z$. Le cluster E_k est inséré dans Z vu que $\mathcal{A}[E_{p(k)} \cap E_k]$ n'est ni un good partiel, ni un good exact ou un nogood. Ainsi, Z contient les clusters dont les variables de leur descendance doivent être explorées pour déterminer si une solution globale existe.

Pour prouver ce théorème, nous procédons par induction sur $VAR(V_{E_i}, Z, \mathcal{A}, G^d)$ et nous considérons la propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ définie par :
"$\#EBTD(P, (E, T), \mathcal{A}, E_i, V_{E_i}, Z, G^d, N^d)$ retourne une paire $(\#sol, E_{bt})$ où $\#sol$ est le nombre exact de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$ si $E_i = E_{bt}$, et une borne inférieure sinon". Lorsqu'il n'y a pas d'ambiguïté, $VAR(V_{E_i}, Z, \mathcal{A}, G^d)$ est noté VAR.

- Cas de base : Considérons $\mathcal{P}(\emptyset)$. Si E_i est un cluster feuille, $\#EBTD$ retourne $(1, E_i)$ vu que la seule extension possible de $\mathcal{A}$ est $\mathcal{A}$. Sinon, comme $VAR = \emptyset$ alors les trois termes de VAR sont vides. Alors, pour chaque cluster E_j fils de E_i, $\mathcal{A}[E_i \cap E_j] \in G_=(E_j)$. Donc, en considérant pour chaque cluster fils E_j de E_i (lignes 10-11), $\#EBTD$ met à jour successivement le nombre de solutions de $\#sol$. Vu que pour chaque cluster fils E_j de E_i, $\mathcal{A}[E_i \cap E_j]$ est un good exact, $Z_{good_\geq}$, $Z_{inconnu}$ et Z restent vides. Ainsi $\#EBTD$ retourne $(\#sol, E_i)$ où $\#sol$ est le exact nombre de solutions de $P_i|\mathcal{A}[E_{p(i)} \cap E_i]$. En conséquence, la propriété $\mathcal{P}(\emptyset)$ est valide.

- Cas général : Considérons maintenant $\mathcal{P}(VAR)$ avec $VAR \neq \emptyset$. Nous supposons que $\forall VAR' \subset VAR$, $\mathcal{P}(VAR')$ est valide.

  - Si $V_{E_i} \neq \emptyset$: la boucle (lignes 50-56) explore le cluster E_i comme $\#BT$. La seule différence est que cette boucle est suspendue lorsque $E_{bt} \neq E_i$ ce qui est nécessaire si un nogood est trouvé ultérieurement pendant la recherche. Si $E_{bt} = E_i$, la boucle est équivalente à celle de $\#BT$ et $\#sol_x$ est le nombre de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$ pour les valeurs v déjà affectées à x. Lorsque $\#EBTD$ essaye d'assigner v à x, deux sous-cas existent :
    * $\mathcal{A} \cup \{x \leftarrow v\}$ est incohérente, il n'y a pas d'extension possible et $\#sol_x$ et E_{bt} restent inchangés.
    * $\mathcal{A} \cup \{x \leftarrow v\}$ est cohérente, nous faisons un appel récursif à $\#EBTD$ (ligne 54) sur $V_{E_i} \backslash \{x\}$. Dans ce cas, d'après l'hypothèse d'induction, $\mathcal{P}(VAR \backslash \{x\})$ est valide. Si $E_{bt} = E_i$, alors $\#sol_{xv}$ est le nombre exact de solutions du problème $P_i|\mathcal{A} \cup \{x \leftarrow v\}$. $\#sol_x$ est alors mis à jour pour la valeur v. Si $E_{bt} \neq E_i$, $\#sol_{xv}$ est une borne inférieure sur le nombre de solutions de $P_i|\mathcal{A} \cup \{x \leftarrow v\}$ et $\#sol_x$ est alors mis à jour. Comme x est affectée pour la première fois en E_i alors $x \notin E_{bt}$ et la boucle est alors suspendue retournant une borne inférieure sur le nombre de solutions de $P_i|\mathcal{A}[E_i \backslash V_{E_i}]$.

    Ainsi, la propriété $\mathcal{P}(VAR)$ est valide dans les deux sous-cas.
  - Si $V_{E_i} = \emptyset$: Comme $VAR \neq \emptyset$, pour chaque cluster fils E_j de E_i, $\mathcal{A}[E_i \cap E_j]$ peut être un good ou une affectation inconnue. Aussi, Z peut être vide ou non. Les lignes 6-16 considèrent chaque cluster fils E_j et mettent à jour, si nécessaire, Z, $Z_{good_\geq}$, $Z_{inconnu}$ et $\#sol$. Ainsi, si $\mathcal{A}[E_i \cap E_j]$ correspond à un good exact $\#sol$ est alors mis à jour en fonction du nombre de solutions enregistré $\#sol_{E_j}$. Si $\mathcal{A}[E_i \cap E_j]$ correspond à un good partiel, le cluster E_j est inséré dans $Z_{good_\geq}$. Sinon, E_j est inséré dans Z et $Z_{inconnu}$. Les lignes 17-35 permettent de lancer l'exploration des clusters de Z dans le but de déterminer si une solution globale contenant $\mathcal{A}$ existe. En ce qui concerne l'appel récursif à la ligne 20, nous montrons que $\mathcal{P}(VAR(V_{E_j}, Z, \mathcal{A}, G^d))$ est valide. Si $VAR(V_{E_j}, Z, \mathcal{A}, G^d) \subset VAR(V_{E_i}, Z, \mathcal{A}, G^d)$, par hypothèse d'induction, $\mathcal{P}(VAR(V_{E_j}, Z, \mathcal{A}, G^d))$ est valide. Sinon, nous avons $VAR(V_{E_j}, Z, \mathcal{A}, G^d) = VAR(V_{E_i}, Z, \mathcal{A}, G^d)$. Toutefois, nous savons que pour l'appel suivant, $V_{E_j} \neq \emptyset$ parce que nous explorons le cluster E_j comme dans les lignes 46-57. Cela correspond au cas $V_{E_j} \neq \emptyset$ décrit ci-dessus. Alors la propriété $\mathcal{P}(VAR(V_{E_j}, Z, \mathcal{A}, G^d))$ est valide. Nous distinguons à ce stade trois sous-cas possibles :
    * Si $\#sol_{E_j} = 0$, $P_j|\mathcal{A}[E_{p(j)} \cap E_j]$ n'admet aucune solution et l'affectation $\mathcal{A}[E_{p(j)} \cap E_j]$ est enregistrée comme un nogood structurel (ligne 31). En raison de l'enregistrement d'un nogood, la recherche doit faire un retour-arrière vers le cluster impliqué dans cette incohérence qui est $E_{p(j)}$. Si

$E_i = E_{p(j)}$ (ligne 32), les fils de E_i sont supprimés de la pile Z vu que leur exploration est désormais inutile (ligne 33). Ainsi, $\#EBTD$ retourne $(0, E_i)$ puisqu'il n'existe aucune extension possible de $\mathcal{A}$ sur les variables de $Desc(E_j)$ et le cluster E_i doit être modifié (ligne 34). Sinon, $E_i \neq E_{p(j)}$ et la recherche doit faire un retour-arrière plus loin. Ainsi, $\#EBTD$ retourne $(\#sol, E_{p(j)})$ avec $\#sol$ une borne inférieure sur le nombre de solutions de $P_i|\mathcal{A}[E_i]$ et le nouveau cluster E_{bt} est $E_{p(j)}$ (ligne 35). En conséquence, la propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ est valide.

* Si $\#sol_{E_j} > 0$ et $E_{bt} \neq E_j$ alors le nombre de solutions retourné est nécessairement une borne inférieure sur le nombre de solutions $P_j|\mathcal{A}[E_{p(j)} \cap E_j]$. $\mathcal{A}[E_{p(j)} \cap E_j]$ est alors enregistrée comme un good partiel qui peut être exploité ultérieurement, si nécessaire. Le raisonnement des lignes 26-29 est similaire à celui pour les lignes 32-35. Soit le cluster courant E_i est impliqué dans l'incohérence (lignes 26-28), soit la recherche fait un retour-arrière jusqu'au cluster E_{bt} et essaye de modifier l'affectation de ses variables (ligne 29). Ainsi, la propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ est valide.
* Si $\#sol_{E_j} > 0$ et $E_{bt} = E_j$ alors le nombre de solutions retourné est celui du nombre exact de solutions de $P_j|\mathcal{A}[E_{p(j)} \cap E_j]$. $\mathcal{A}[E_{p(j)} \cap E_j]$ est enregistrée comme un good exact. À ce niveau, Z est vide puisque sinon l'exploration des clusters de Z aurait continué jusqu'à atteindre le dernier cluster. En d'autres termes, l'enregistrement des goods exacts ne peut pas se faire avant que tous les clusters de Z ne soient explorés. En outre, sachant que la traversée de la décomposition se fait grâce à une pile, si $\mathcal{A}[E_{p(j)} \cap E_j]$ est enregistrée comme un good exact alors tous les clusters E_k insérés dans Z avant E_j sont déjà exhaustivement explorés et l'affectation $\mathcal{A}[E_{p(k)} \cap E_k]$ est déjà enregistrée comme un good exact. Ainsi, la propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ est valide.

La propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ est alors valide pour $Z \neq \emptyset$ pour les trois sous-cas possibles. À la ligne 36, $Z = \emptyset$, une solution globale a été déjà trouvée et pour chaque cluster fils $E_j \in Z_{inconnu}$, $\#sol_{E_j}$ est calculé et $\#sol$ est mis à jour (ligne 37). Si $Z_{good_\geq} \neq \emptyset$, chaque cluster $E_j \in Z_{good_\geq}$ est exploré (ligne 41). Si $VAR(V_{E_j}, Z, \mathcal{A}, G^d) \subset VAR(V_{E_i}, Z, \mathcal{A}, G^d)$, par hypothèse d'induction, $\mathcal{P}(VAR(V_{E_j}, Z, \mathcal{A}, G^d))$ est valide. Sinon, nous avons $VAR(V_{E_j}, Z, \mathcal{A}, G^d) = VAR(V_{E_i}, Z, \mathcal{A}, G^d)$. Cependant, comme pour l'appel à la ligne 20, nous savons que pour l'appel suivant $V_{E_j} \neq \emptyset$ et $\mathcal{P}(VAR(V_{E_j}, Z, \mathcal{A}, G^d))$ est valide. Par définition d'un good partiel, $P_j|\mathcal{A}[E_i \cap E_j]$ a au moins une seule solution. Comme Z est vide (une solution globale existe), l'appel récursif à $\#EBTD$ retourne nécessairement le nombre exact de solutions de $P_j|\mathcal{A}[E_i \cap E_j]$. Une fois, tous les clusters fils E_j considérés, $\#sol$ est mis-à-jour et $\#EBTD$ retourne $(\#sol, E_i)$. En conséquence, la propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ est valide pour $V_{E_i} \neq \emptyset$.

En conséquence, nous pouvons déduire que la propriété $\mathcal{P}(VAR(V_{E_i}, Z, \mathcal{A}, G^d))$ est valide. Au niveau de la terminaison, nous pouvons constater que la taille de VAR décroît à chaque appel. En effet, dans le cas où $V_{E_i} \neq \emptyset$, l'appel récursif fait par $\#EBTD$ considère l'ensemble $V_{E_i} \backslash \{x\}$ qui contient strictement moins de variables que V_{E_i} (une variable en moins). Ainsi, la taille de VAR est désormais plus petite. Si $V_{E_i} = \emptyset$, comme nous l'avons déjà vu, les appels récursifs aux lignes 20 et 41 garantissent que $VAR(V_{E_j}, Z, \mathcal{A}, G^d) \subseteq VAR(V_{E_i}, Z, \mathcal{A}, G^d)$. En cas d'égalité, nous exploitons le fait que pour l'appel suivant

$V_{E_j} \neq \emptyset$ et ainsi l'appel récursif à la ligne 54 garantit que le nombre de variables de $VAR(V_{E_j}, Z, \mathcal{A}, G^d)$ diminuera.

L'algorithme $\#EBTD$ est alors correct, complet et termine.□

Finalement, nous donnons les complexités en temps et en espace qui sont comparables avec celles de $\#BTD$.

Théorème 14 *$\#EBTD$ a une complexité en temps en $O(n.(r.m + n.s).d^{w^+ + 1})$ et en espace en $O(n.s.d^s)$.*

Preuve : Le nombre maximal de contraintes impliquées dans chaque vérification de cohérence est m et le coût de chaque vérification de contrainte est de $O(r)$. En outre, nous supposons que les nogoods d'un cluster E_i par rapport à un de ses clusters fils E_j sont stockés comme une contrainte dont la portée est l'ensemble de variables de $E_i \cap E_j$ et qu'il y a au plus n-1 séparateurs (vu que le nombre de clusters est majoré par n). De plus, la vérification de l'existence d'un nogood peut être réalisée en $O(s)$. Ainsi, la vérification de cohérence de la ligne 53 est réalisée en $O(r.m + n.s)$.

Dans le pire des cas, $\#EBTD$ explore tous les clusters de la décomposition et essaye toutes les valeurs possibles de chaque variable du cluster. Comme w^+ est la largeur de la décomposition arborescente employée, la taille maximale d'un cluster est de $w^+ + 1$. Alors, le nombre d'affectations d'un cluster est borné par $d^{w^+ + 1}$. Un good exact enregistré assure que le cluster correspondant n'est pas exploré plus d'une fois pour la même affectation des variables de son séparateur. Si un good partiel est enregistré, le cluster est, au plus, visité deux fois avec la même affectation sachant que la deuxième fois le sous-problème correspondant est entièrement exploré et ainsi le good partiel est remplacé par un good exact. En plus, supposons que les goods sont mémorisés pour chaque séparateur comme les nogoods. Mémoriser un good ou vérifier son existence est alors réalisé en $O(s)$. En conséquence, $\#EBTD$ a une complexité temporelle en $O(n.(r.m + n.s).d^{w^+ + 1})$.

En ce qui concerne la complexité spatiale, $\#EBTD$ enregistre des goods exacts, partiels et des nogoods. Ces enregistrements sont réalisés par rapport à un séparateur $E_i \cap E_j$ où E_j est un fils de E_i. Étant donné que s est la taille maximale des séparateurs, la taille d'un (no)good est bornée par s. Pour chaque séparateur, il existe au plus d^s affectations possibles. Ainsi, comme le nombre de séparateurs est borné par n, la complexité spatiale est en $O(n.s.d^s)$. □

Notons que, lorsque $\#EBTD$ explore, pour la deuxième fois, un sous-problème donné dans le but de calculer un good exact, nous pouvons éviter de redémarrer la recherche du début. En effet, si nous supposons que nous utilisons un ordre de valeurs lexicographique, il suffit d'enregistrer la solution partielle correspondante au good partiel conjointement avec ce good partiel. Ce faisant, si $\#EBTD$ a besoin de réexplorer le sous-problème correspondant il peut en toute sécurité démarrer à partir de cette solution partielle. Un tel problème ne change pas la complexité temporelle tandis que la complexité spatiale devient $O(n.w^+.d^s)$. Cependant, d'un point de vue pratique, les gains peuvent être significatifs.

Avant d'évaluer expérimentalement l'intérêt pratique de $\#EBTD$, nous devons préciser que suite à la publication de ce travail, nous avons été informés de l'existence d'un travail similaire présenté dans la thèse de Karakashian [Karakashian, 2013]. L'idée est fondamentalement la même. L'algorithme proposé évite de compter le nombre de solutions pour un sous-problème avant de s'être assuré que l'affectation partielle s'étend en une solution globale. Néanmoins, du point de vue théorique, certaines faiblesses peuvent être soulignées. L'algorithme fourni ne traduit pas exactement le déroulement décrit. Aucune distinction

n'est faite entre un good partiel et un good exact. En plus, il n'est fourni ni preuve de validité, ni preuve de complexité. Finalement, la partie expérimentale donnée est peu développée et la comparaison entre $\#BTD$ et le nouvel algorithme proposé ne met pas ce dernier suffisamment en valeur.

6.4 Étude expérimentale

Nous évaluons dans cette section l'intérêt pratique de notre approche.

6.4.1 Benchmark utilisé

Nous considérons des instances CSP représentées dans le format XCSP3 [Boussemart et al., 2016]. Tous les détails nécessaires peuvent être trouvés dans [XCS, 2017]. Nous considérons 4 069 instances cohérentes. Elles rassemblent des instances provenant notamment des applications réelles comme la famille *Renault* ou la famille $Rlfap$. Une description de telles familles peut être retrouvée dans [XCS, 2017]. Une majorité d'instances représentent des problèmes académiques dont une partie est générée aléatoirement. Le nombre de variables des instances est situé entre 4 et 28 000 variables avec des domaines d'une taille variant de 1 à 6 561. Le nombre de contraintes varie de 2 à 139 500 d'une arité allant de 2 à 1 000. Il s'agit des contraintes de divers types exprimées en intention ou en extension ou sous forme de contraintes globales (*AllEqual*, *AllDiff*, *Sum*, *Element*) [Beldiceanu et al., 2005; van Hoeve and Katriel, 2006]. Ces instances peuvent être composées d'une seule composante connexe ou d'un plus grand nombre allant jusqu'à 458 composantes connexes. Le nombre de solutions varient d'une instance à l'autre. Certaines instances ne possèdent qu'une seule solution alors que d'autres peuvent avoir un nombre de solutions de l'ordre de 10^{303} solutions. Ce benchmark sera noté I_4.

6.4.2 Protocole expérimental

En ce qui concerne les décompositions, nous retenons les décompositions suivantes (cf. chapitre 3 pour plus de détails sur H_i et $Min\text{-}Fill\text{-}MG$) :

- $Min\text{-}Fill$: considérée comme étant l'heuristique de l'état de l'art de calcul de décompositions dans la communauté CP ; elle vise à minimiser w^+.
- H_1 : elle vise également à minimiser w^+ sans recourir à la triangulation (cf. chapitre 3).
- H_2 : son objectif consiste à construire des décompositions composées uniquement de clusters connexes.
- H_3 : elle permet de calculer une décomposition telle que chaque cluster possède plusieurs clusters fils.
- H_5 : elle vise à limiter la taille des séparateurs de la décomposition (H_4 est écartée parce qu'elle détecte moins de séparateurs que H_5). Toutefois, nous l'utilisons sans limite de taille pour les séparateurs afin de trouver le plus de séparateurs possibles. Elle sera notée H_5^∞.
- $Min\text{-}Fill\text{-}MG$: elle vise à minimiser w^+ en triangulant d'une façon plus attentionnée que $Min\text{-}Fill$.

Au niveau des algorithmes de résolution, nous considérons les algorithmes suivants :

Algorithme	*Min-Fill*		H_1		*Min-Fill-MG*	
	#rés.	temps	#rés.	temps	#rés.	temps
#*EBTD*	3 023	158 998	2 942	134 147	3 029	162 559

TABLE 6.1 – Nombre d'instances résolues et temps d'exécution en secondes pour $\#EBTD$ selon *Min-Fill*, H_1 et *Min-Fill-MG* pour le benchmark I_4.

Algorithme	H_2		H_3		H_5^∞	
	#rés.	temps	#rés.	temps	#rés.	temps
#*EBTD*	2 970	141 380	2 791	169 357	2 739	148 512

TABLE 6.2 – Nombre d'instances résolues et temps d'exécution en secondes pour $\#EBTD$ selon H_2, H_3 et H_5 pour le benchmark I_4.

- $\#EBTD$: nous considérons une version de $\#EBTD$ basée sur RFL que nous implémentons dans notre propre bibliothèque. Nous choisissons RFL au lieu de MAC vu que RFL réalise un branchement d-aire et que pour le comptage toutes les valeurs d'une variable seront testées. La cohérence d'arc est renforcée en prétraitement via $AC\text{-}3^{rm}$ [Lecoutre et al., 2007c] et pendant la résolution via $AC\text{-}8^{rm}$ (AC-8 [Chmeiss and Jégou, 1998] avec des résidus multi-directionnels). La prochaine variable à affecter est choisie grâce à l'heuristique $dom/wdeg$ [Boussemart et al., 2004]. Le cluster racine est celui qui maximise la somme des poids $wdeg$ des contraintes qui intersectent le cluster.

- $\#BTD$: nous considérons l'implémentation de $\#BTD$ fournie dans *Toulbar*2 [Favier et al., 2009]. La décomposition employée par $\#BTD$ est *Min-Fill* également fournie dans Toulbar2. Le cluster racine est celui ayant la plus grande taille.

- $\#RFL$: nous implémentons une version dans notre bibliothèque.

- *cn*2*mddg* [Koriche et al., 2015] (cf. chapitre 2) : nous considérons l'implémentation du compilateur fournie à l'adresse suivante `http://www.cril.univ-artois.fr/KC/mddg.html`.

- #SAT solveurs (cf. chapitre 2) : nous considérons *cachet* [Sang et al., 2004], *c*2*d* [Darwiche, 2001a], *relsat* [Bayardo and Pehoushek, 2000] et *sharpsat* [Thurley, 2006].

Les expérimentations ont été réalisées sur des serveurs lame sous Linux Ubuntu 14.04 dotés chacun de deux processeurs Intel Xeon E5-2609 à 2,4 GHz et de 32 Go de mémoire.

Chaque instance dispose de 20 minutes (incluant le cas échéant, le temps de calcul de la décomposition) et de 16 Go de mémoire.

6.4.3 Observations et analyse des résultats

#EBTD selon la décomposition utilisée Nous comparons tout d'abord le comportement de $\#EBTD$ selon la décomposition utilisée. La table 6.1 montre le nombre d'instances résolues et le temps cumulé d'exécution pour $\#EBTD$ selon les décompositions visant à minimiser w^+ qui sont : *Min-Fill*, H_1 et *Min-Fill-MG*. Quant à la table 6.2, elle montre le nombre d'instances résolues et le temps cumulé d'exécution pour $\#EBTD$ selon les décompositions dont le but est d'augmenter l'efficacité de la résolution des instances CSP. Nous remarquons que les décompositions dont l'objectif est de minimiser w^+ sont généralement plus efficaces vis-à-vis du comptage du nombre de solutions que les

Paramètre	*Min-Fill*	H_1	*Min-Fill-MG*	H_2	H_3	H_5^∞
p_1	122	108	122	57	34	37
p_2	42	41	42	29	15	18
p_3	29	29	29	46	57	48

TABLE 6.3 – La valeur des paramètres p_1, p_2, p_3 pour chaque décomposition pour le benchmark I_4 : p_1 est la moyenne des nombres des séparateurs, p_2 est la moyenne des pourcentages du nombre des séparateurs par rapport à n, p_3 est la moyenne des pourcentages de w^+ par rapport à n.

Min-Fill	H_1	*Min-Fill-MG*	H_2	H_3	H_5^∞
111 150	114 379	107 478	97 596	116 930	112 016

TABLE 6.4 – Temps d'exécution en secondes pour $\#EBTD$ selon les différentes décompositions pour les instances résolues par $\#EBTD$ avec n'importe quelle décomposition du benchmark I_4.

autres décompositions. En effet, *Min-Fill-MG* permet de résoudre le plus grand nombre d'instances (3 029), suivie par *Min-Fill* (3 023), puis par H_2 (2 970), ensuite par H_1 (2 942) et enfin par H_3 (2 791) et H_5 (2 739). Ces résultats sont également représentés dans la figure 6.3. Il semble ainsi que les critères jugés pertinents pour permettre une résolution efficace des instances CSP ne sont pas les plus intéressants pour la résolution des instances #CSP. Si les décompositions telles que H_3 et H_5 permettent souvent de trouver rapidement une première solution (comme dans le cas des CSP), l'étape du comptage se trouve être pénalisée. En conclusion, la taille des clusters semble être un paramètre déterminant pour une résolution efficace des instances #CSP. Nous examinons maintenant les différentes décompositions de plus près. La table 6.3 montre pour chaque décomposition pour le benchmark I_4 la valeur de trois paramètres p_1, p_2 et p_3. p_1 est la moyenne des nombres des séparateurs, p_2 est la moyenne des pourcentages du nombre des séparateurs par rapport à n et p_3 est la moyenne des pourcentages de w^+ par rapport à n. Vu les résultats de la table 6.3, il semble que les décompositions $H_{2,3,5}$ sont pénalisées, en plus de la taille des clusters, par le nombre restreint de séparateurs. Ainsi, nous pouvons établir un lien entre le nombre de séparateurs et l'efficacité du comptage. En effet, plus le nombre de séparateurs augmente, plus l'efficacité du comptage est susceptible d'augmenter. Ceci est dû à l'exploitation des séparateurs et l'augmentation du taux d'enregistrements pouvant être réalisés. Notons que ce taux peut être à l'origine de la saturation de l'espace mémoire disponible. Par exemple, $\#EBTD$ explose en mémoire pour 32 instances avec *Min-Fill*, pour 53 instances avec H_2, pour une instance avec H_3 et H_5^∞, pour 165 instances avec H_1 et pour 25 instances avec *Min-Fill-MG*. Ainsi, il faut veiller à équilibrer le taux d'enregistrements réalisés. Lorsque nous nous limitons aux instances dont le ratio $\frac{n}{w^+}$ est supérieur ou égal à 10, les résultats sont représentés dans la figure 6.4. Ils montrent notamment l'importance de la connexité sur cette catégorie d'instances. Au niveau des temps d'exécution, nous comparons les temps d'exécution de $\#EBTD$ avec les différentes décompositions sur le benchmark formé des instances résolues avec toutes les décompositions. Ce benchmark contient 2 619 instances. Les résultats sont montrés par la table 6.4. Les temps d'exécutions cumulés ne sont pas significativement éloignés. Les meilleurs temps de résolution sont ceux obtenus avec H_2 et *Min-Fill-MG*. Nous retenons pour la suite la décomposition *Min-Fill-MG*. Il est à noter que $\#EBTD$ avec *Min-Fill-MG* est capable de calculer, soit le nombre exact de solutions, soit une borne inférieure non nulle sur le nombre de solutions pour environ 86% des instances considérées.

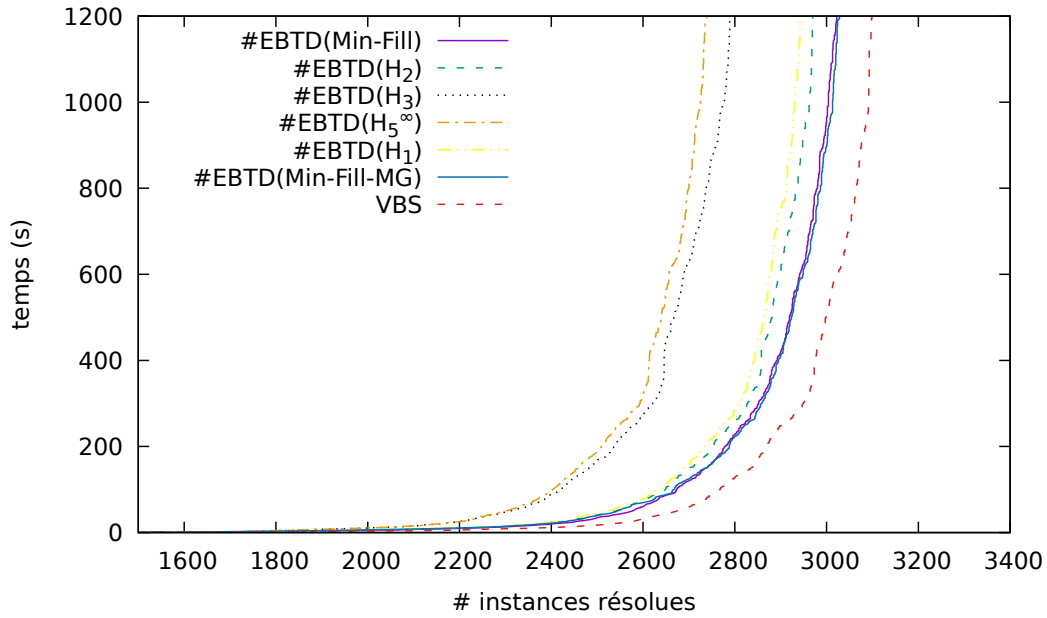


FIGURE 6.3 – Le nombre cumulé d'instances résolues pour $\#EBTD$ selon la décomposition employée et leur VBS.

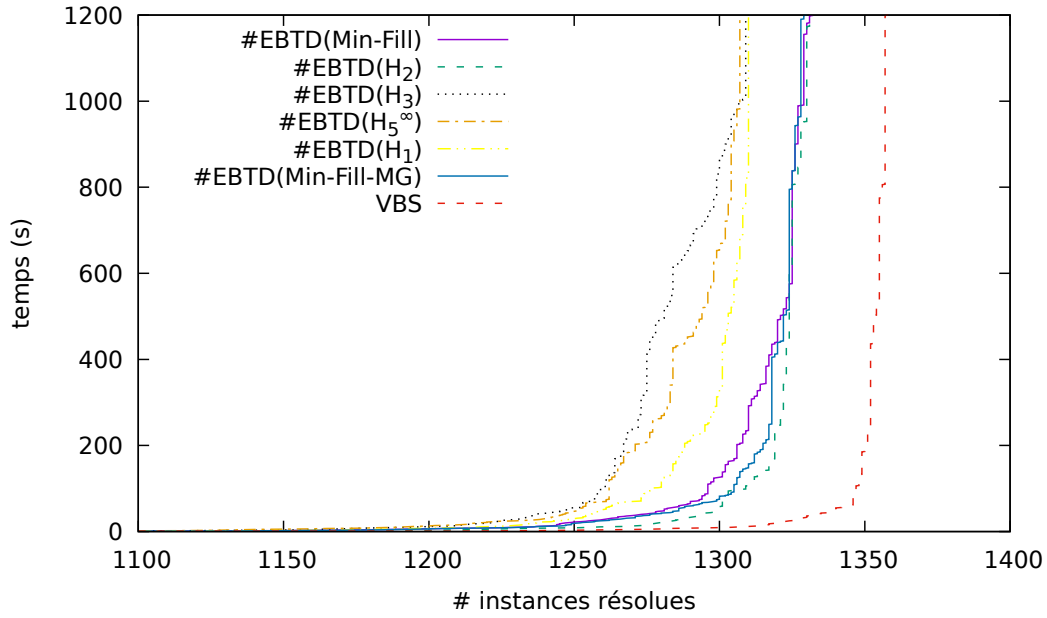


FIGURE 6.4 – Le nombre cumulé d'instances résolues pour $\#EBTD$ selon la décomposition employée et leur VBS pour les instances telles que $\frac{n}{w^+} \geq 10$ du benchmark I_4.

#EBTD vs #RFL Nous comparons à présent le comportement de $\#EBTD$ vis-à-vis de $\#RFL$. $\#RFL$ permet de compter exactement le nombre de solutions de 2 607 instances contre 3 029 pour *Min-Fill-MG*. $\#EBTD$ est alors significativement meilleur que $\#RFL$ en nombres d'instances résolues exactement. La figure 6.5 montre le nombre cumulé d'instances résolues pour $\#RFL$, $\#EBTD$ et leur VBS (pour *Virtual Best Solver*). La figure montre clairement l'intérêt de $\#EBTD$ par rapport à $\#RFL$. Elle montre

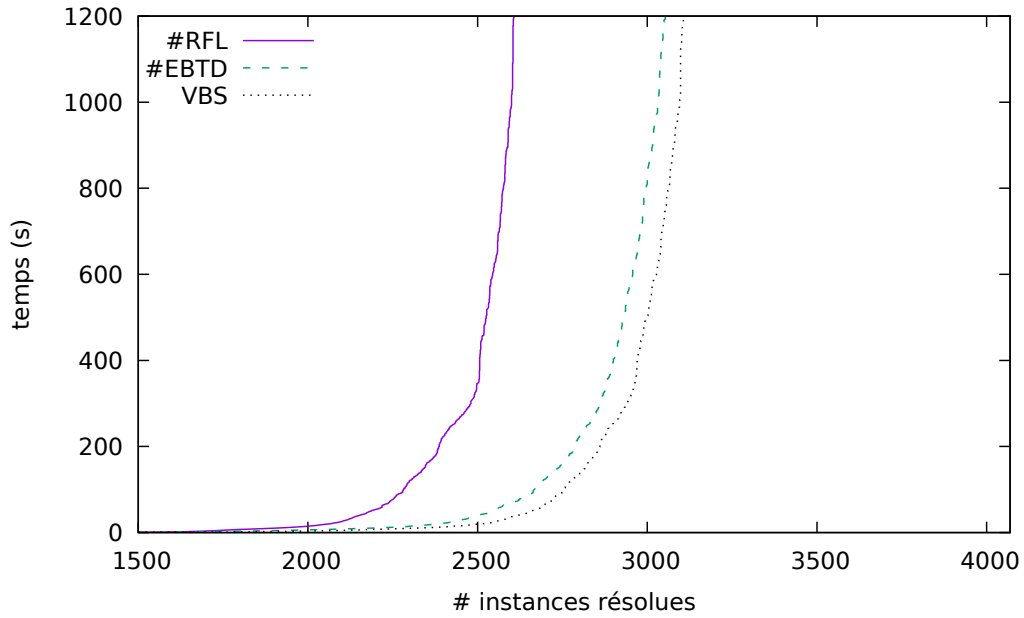


FIGURE 6.5 – Le nombre cumulé d'instances résolues pour $\#RFL$, $\#EBTD$ et leur VBS pour le benchmark I_4.

également que $\#EBTD$ a un comportement proche de celui du VBS. Il y a cependant 56 instances résolues par $\#RFL$ et pas par $\#EBTD$. Parmi ces 56 instances, pour 35 instances $\#EBTD$ ne parvient à trouver aucune solution. Ce fait met en avant la difficulté que rencontre $\#EBTD$ à trouver la première solution dans certains cas. Ceci n'est pas étonnant vu les résultats obtenus dans le cadre du problème CSP. En effet, la décomposition *Min-Fill-MG* est loin d'être efficace pour résoudre une instance CSP par rapport à une méthode telle que MAC ou RFL. Nous comparons maintenant les temps d'exécution cumulés des deux algorithmes. Afin de comparer d'une façon équitable les temps, nous nous basons sur l'ensemble d'instances résolues à la fois par $\#EBTD$ et $\#RFL$. Il compte 2 551 instances résolues par $\#RFL$ en environ 120 000 s et par $\#EBTD$ en 130 000 s. Lorsque $\#RFL$ et $\#EBTD$ comptent tout les deux exactement le nombre de solutions et que l'instance est résolue plus rapidement avec $\#RFL$ qu'avec $\#EBTD$, ceci est *a fortiori* dû à l'exploitation de la liberté totale de l'heuristique de choix de variables. Il s'agit le plus souvent d'instances ayant un nombre de solutions relativement faible (une solution unique dans certains cas). En effet, lorsque le nombre de solutions d'une instance est considérablement élevé, la capacité de l'énumération est dépassée en l'absence de plusieurs composantes connexes. En plus, la redondance des espaces de recherche visités est handicapante et entraîne une forte dégradation de l'efficacité de la recherche. D'ailleurs, le plus souvent lorsque le nombre de solutions trouvées par $\#RFL$ est élevé, l'instance en question contient plusieurs composantes connexes. Ainsi, ce nombre de solutions élevé résulte de la multiplication du nombre de solutions de chaque composante connexe. Nous comparons finalement les bornes inférieures sur le nombre de solutions calculées par $\#RFL$ et $\#EBTD$ lorsque le nombre exact de solutions n'est pas trouvé en raison du dépassement du temps limite ou de l'espace mémoire disponible. 544 instances sont concernées. Pour 278 instances la borne inférieure calculée par $\#RFL$ est strictement inférieure à celle calculée par $\#EBTD$. Cependant, pour 118 instances la borne inférieure calculée par $\#EBTD$ est strictement inférieure à celle calculée par $\#RFL$. Pour le reste d'instances, la borne

inférieure calculée par les deux algorithmes est la même. Cette comparaison est illustrée

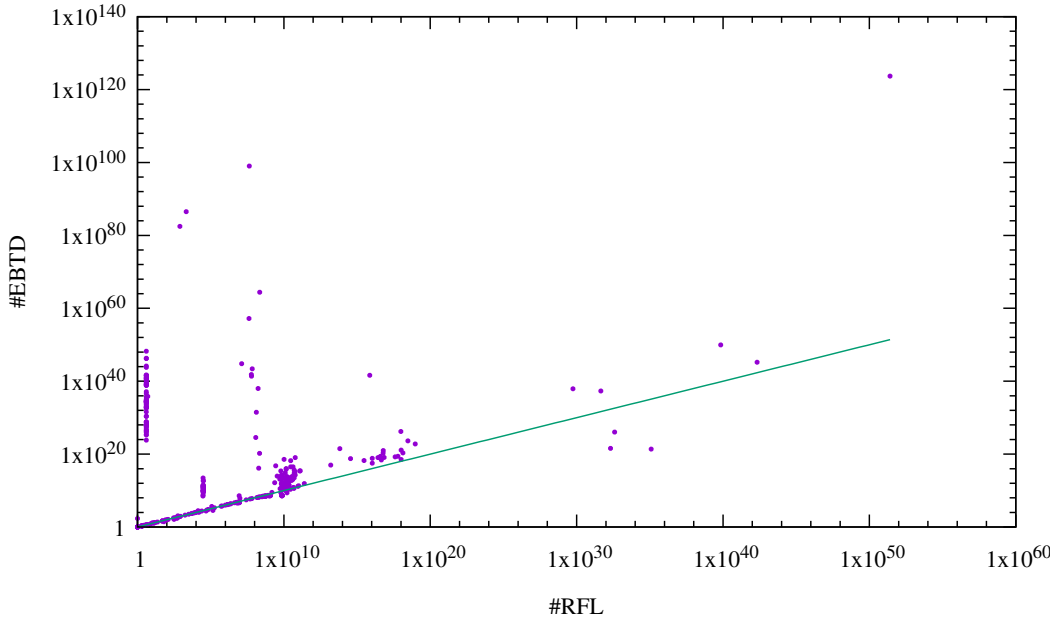


FIGURE 6.6 – Comparaison des bornes inférieures pour le nombre exact de solutions calculées par $\#RFL$ et $\#EBTD$.

par la figure 6.6.

#EBTD vs #BTD Nous comparons à présent $\#EBTD$ à $\#BTD$, la méthode structurelle de l'état de l'art pour le comptage du nombre de solutions. Afin d'y parvenir, nous encodons les instances considérées en un format supporté par *Toulbar*2. Pour $\#BTD$, nous encodons les instances CSP en format WCSP. Cet encodage nécessite la mise à plat des contraintes. Une telle opération pourrait être particulièrement coûteuse en temps ainsi qu'en espace mémoire notamment pour les instances contenant des contraintes en intention ou des contraintes globales. Ainsi, pour notre comparaison, nous considérons 3 956 instances que nous avons réussi à convertir dans moins de 25 minutes et en limitant la taille du fichier résultat à $200Mo$. Ce benchmark est noté $I_{4.1}$ avec $I_{4.1} \subset I_4$. Notons que dans ces expérimentations les temps de conversion du format XCSP3 à n'importe quel autre format ne sont pas inclus dans les temps de résolution des méthodes considérant les instances converties. Sur cet ensemble d'instances, $\#BTD$ résout 2 508 instances tandis que $\#EBTD$ en résout 2 996. $\#BTD$ résout donc considérablement moins d'instances que $\#EBTD$. En outre, le temps cumulé réalisé par $\#EBTD$ est d'environ 153 000 s tandis que celui de $\#BTD$ est d'environ 192 000 s. Ainsi, l'augmentation du nombre d'instances dont $\#EBTD$ calcule le nombre exact de solutions est accompagnée d'une baisse significative du temps de résolution. La figure 6.7 montre que $\#EBTD$ améliore clairement $\#BTD$. Finalement, la figure 6.8 montre l'apport important de $\#EBTD$ par rapport à $\#BTD$. Il est à noter que $\#BTD$ n'est pas capable de donner une borne inférieure sur le nombre de solutions en cas de dépassement de limite de temps ou d'espace mémoire. Cependant, $\#EBTD$ est capable de fournir une borne inférieure non nulle pour 370 instances additionnelles.

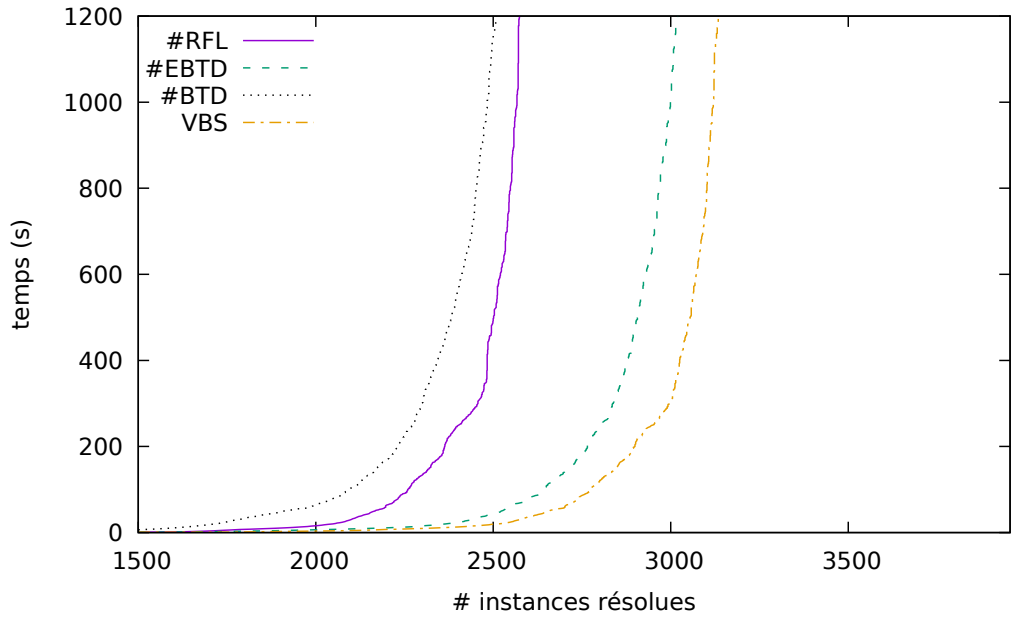


FIGURE 6.7 – Le nombre cumulé d'instances résolues pour $\#RFL$, $\#EBTD$, $\#BTD$ et leur VBS pour le benchmark $I_{4.1}$.

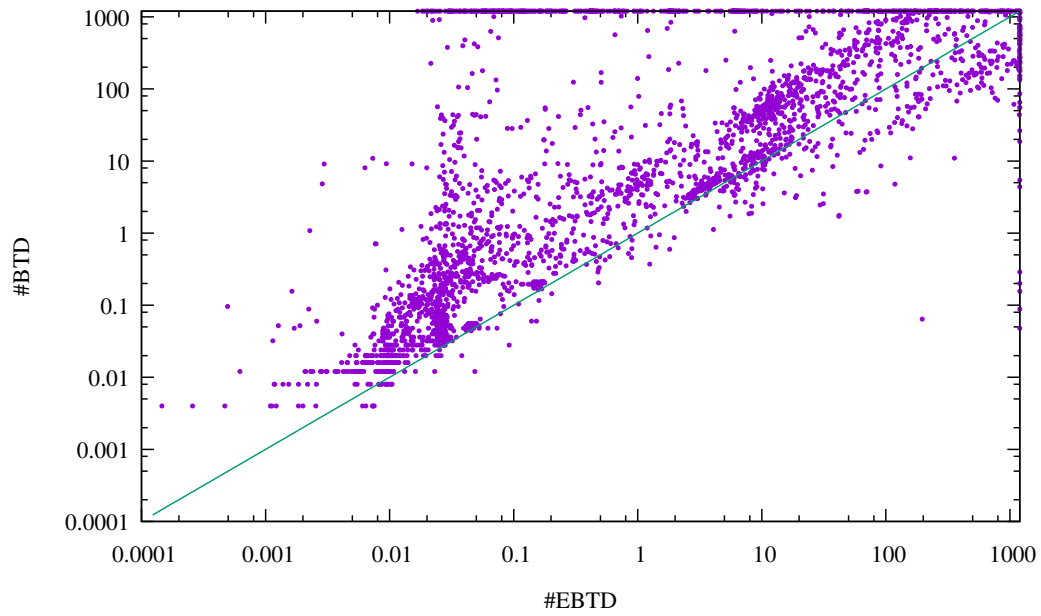


FIGURE 6.8 – Comparaison des temps d'exécution cumulés de $\#EBTD$ et de $\#BTD$.

#EBTD vs cn2mddg Nous comparons maintenant $\#EBTD$ à *cn2mddg* qui prend en entrée des instances au format XCSP2.1 [Roussel and Lecoutre, 2009]. C'est pourquoi la conversion nous oblige également à nous limiter aux 3 956 instances du benchmark $I_{4.1}$. *cn2mddg* permet de compter exactement le nombre de solutions de 3 177 instances en 146 430 s. La compilation réalisée par *cn2mddg* semble permettre de compter le nombre de solutions d'une instance plus efficacement que $\#EBTD$. La figure 6.9 montre que *cn2mddg* a une meilleure performance que les autres méthodes sur l'ensemble total des instances.

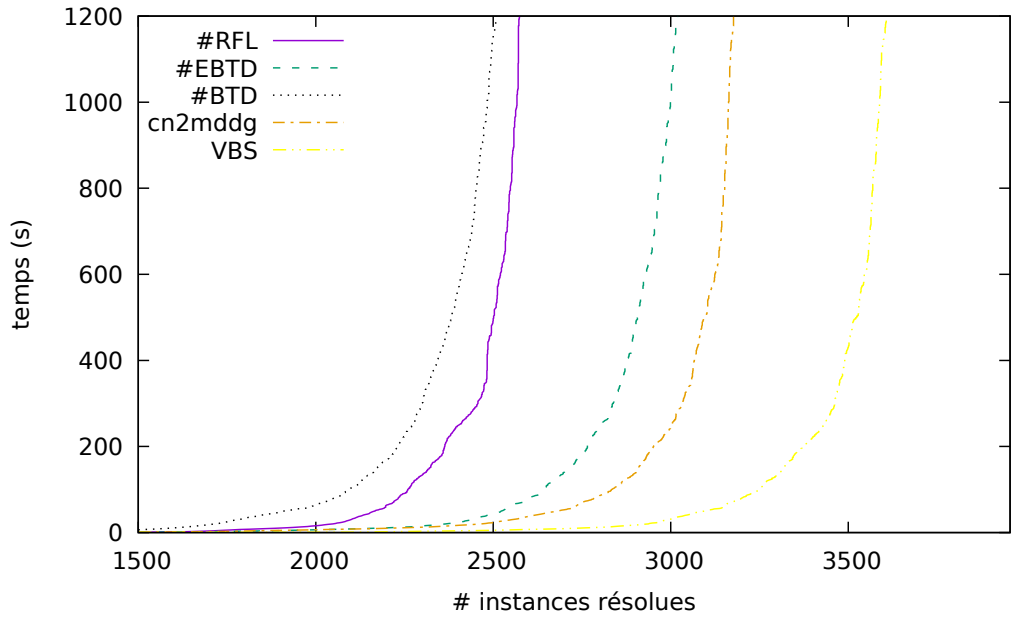


FIGURE 6.9 – Le nombre cumulé d'instances résolues pour $\#RFL$, $\#EBTD$, $\#BTD$, $cn2mddg$ et leur VBS pour le benchmark $I_{4.1}$.

Le comportement du VBS est davantage éloigné de celui du reste des méthodes. Cela montre que les différentes méthodes ne résolvent pas les mêmes instances. Il est à noter que contrairement à $\#EBTD$, le nombre de solutions d'une instance est soit compté exactement, soit aucune solution n'est trouvée dans le cas de $cn2mddg$. Ensuite, nous regardons l'évolution du nombre d'instances résolues à l'exact par $\#EBTD$ et $cn2mddg$ en se restreignant à un ensemble d'instances de plus en plus difficile pour $cn2mddg$. Par exemple, la deuxième ligne du tableau 6.5 indique que nous nous limitons aux instances du benchmark $I_{4.1}$ résolues en plus de 10 s. Les résultats sont montrés dans la table 6.5 et la figure 6.10. Nous remarquons à travers ce tableau que l'augmentation de la difficulté des instances pour $cn2mddg$ met de plus en plus en valeur $\#EBTD$. En effet, pour des instances faciles pour $cn2mddg$, le temps de résolution pourrait potentiellement même être inférieur au temps de calcul de la décomposition. Dans ces cas, l'utilisation d'une décomposition n'est pas justifée. Ce fait met en évidence la complémentarité des deux approches. Ainsi, lorsque l'instance est difficile pour $cn2mddg$, l'utilisation de $\#EBTD$ semble être une alternative intéressante.

#EBTD vs #SAT solveurs Dans cette partie, nous comparons $\#EBTD$ aux #SAT solveurs. Nous devons donc considérer un format supporté par les différents #SAT solveurs. Nous exploitons alors l'encodage direct du format CSP au format SAT [Walsh, 2000], l'encodage logarithmique [Walsh, 2000] et l'encodage tuple [Hurley et al., 2016]. La conversion étant coûteuse en temps et en espace, nous sommes contraints de considérer un sous-ensemble d'instances pour les différents encodages. Nous considérons les instances que nous avons réussi à convertir dans un limite de 25 minutes et dont la taille en arrivée ne dépasse pas $200Mo$. Pour l'encodage direct, nous considérons un benchmark de 2 573 instances noté $I_{direct} \subset I_{4.1}$. En ce qui concerne l'encodage logarithmique, nous nous limitons à un benchmark de 2 424 instances noté $I_{log} \subset I_{4.1}$. Cependant, nous considérons un benchmark de 3 052 instances pour l'encodage tuple noté $I_{tuple} \subset I_{4.1}$.

Temps de résolution de *cn2mddg*	#rés par $\#EBTD$	#rés par *cn2mddg*
≥ 0	2 996	3 177
≥ 10	882	980
≥ 20	644	711
≥ 50	444	493
≥ 100	336	344
≥ 200	248	226
≥ 300	200	144
≥ 400	179	111
≥ 500	166	80
≥ 600	153	52
≥ 700	145	35
≥ 800	138	25
≥ 900	138	20
≥ 1000	134	14
≥ 1100	133	9
≥ 1200	129	0

TABLE 6.5 – Le nombre d'instances résolues à l'exact par $\#EBTD$ et *cn2mddg* en se restreignant à un ensemble d'instances de plus en plus difficile pour le compilateur *cn2mddg*.

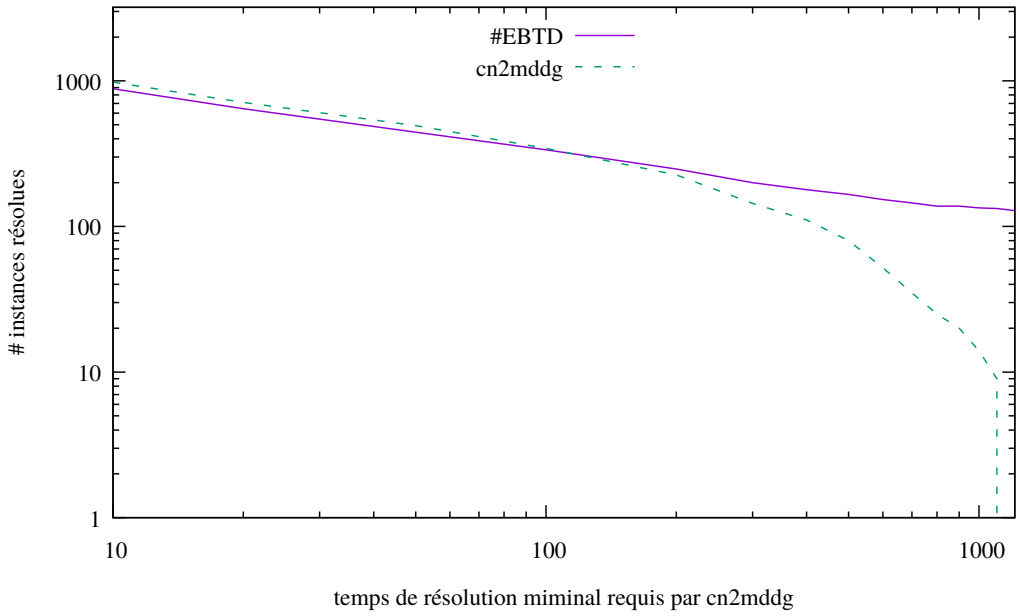


FIGURE 6.10 – Évolution du nombre d'instances résolues par $\#EBTD$ et *cn2mddg* selon les instances considérées du benchmark $I_{4.1}$. Nous nous restreignons à un benchmark de plus en plus difficile pour *cn2mddg*.

Les tables 6.6, 6.7 et 6.8 ainsi que les figures 6.11, 6.12 et 6.13 montrent que les #SAT solveurs ne sont pas compétitifs vis-à-vis des autres compteurs. Nous constatons aussi qu'à chaque fois le VBS a une performance bien meilleure que tous les algorithmes. Nous pouvons alors déduire que les algorithmes évalués résolvent des instances différentes. Lorsque l'encodage logarithmique est utilisé, nous pouvons facilement déduire que les compteurs #SAT ont une performance très médiocre. Nous pouvons alors écarter ces compteurs.

Algorithme	#rés.	temps (s)
$\#RFL$	1 788	86 043
$\#EBTD$	2 231	106 070
$\#BTD$	2 007	156 906
cn2mddg	2 254	97 938
c2d	1 008	223 265
relsat	1 411	130 884
cachet	1 088	208 630
sharpsat	1 990	232 123
VBS	2 388	65 862

TABLE 6.6 – Nombre d'instances résolues et temps d'exécution en secondes pour les différents algorithmes pour le benchmark I_{direct}.

Algorithme	#rés.	temps (s)
$\#RFL$	1 661	86 658
$\#EBTD$	2 102	106 235
$\#BTD$	1 880	157 842
cn2mddg	2 115	96 102
c2d	703	60 737
relsat	649	38 079
cachet	657	168 061
sharpsat	886	132 225
VBS	2 251	68 594

TABLE 6.7 – Nombre d'instances résolues et temps d'exécution en secondes pour les différents algorithmes pour le benchmark I_{log}.

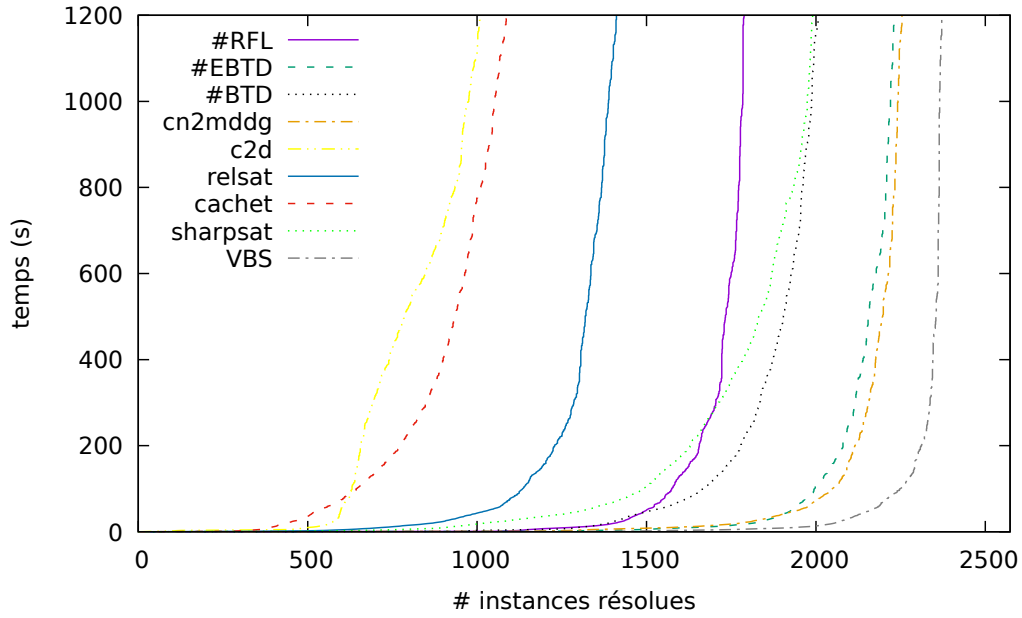


FIGURE 6.11 – Le nombre cumulé d'instances résolues pour tous les algorithmes pour le benchmark I_{direct}.

Algorithme	#rés.	temps (s)
$\#RFL$	2 313	102 351
$\#EBTD$	2 753	123 995
$\#BTD$	2 394	180 354
cn2mddg	2 758	102 240
c2d	1 074	172 571
relsat	1 430	72 787
cachet	1 942	160 521
sharpsat	2 321	293 360
VBS	2 902	89 533

TABLE 6.8 – Nombre d'instances résolues et temps d'exécution en secondes pour les différents algorithmes pour le benchmark I_{tuple}.

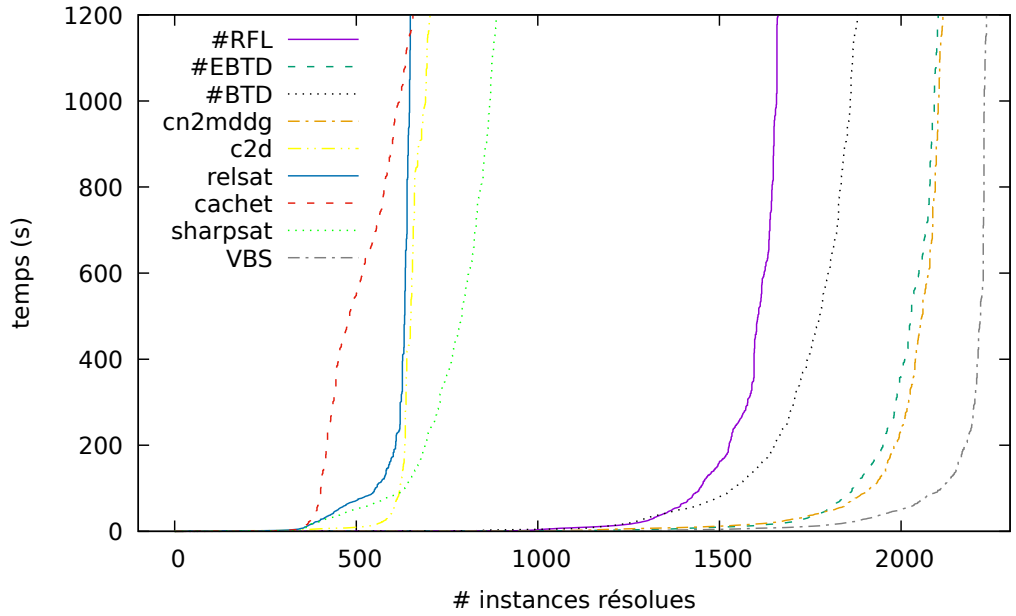


FIGURE 6.12 – Le nombre cumulé d'instances résolues pour tous les algorithmes pour le benchmark I_{log}.

Pour les deux autres encodages, le compteur qui semble le plus efficace est *sharpsat*. Nous nous intéressons maintenant au benchmark résolu à la fois par $\#EBTD$ et *sharpsat*. Sur les deux figures 6.14 et 6.15, nous pouvons constater que malgré l'existence de certaines instances résolues plus rapidement avec *sharpsat* qu'avec $\#EBTD$, $\#EBTD$ domine fortement *sharpsat*. En effet, pour la majorité des instances $\#EBTD$ est capable de compter le nombre exact de solutions plus rapidement que *sharpsat*. D'ailleurs, les temps cumulés d'exécution de $\#EBTD$ et de *sharpsat* sont respectivement 77 611 s et 223 859 s lorsque l'encodage direct est exploité et 34 973 s et 274 256 s si l'encodage tuple est utilisé. Il est à noter finalement qu'aucun compteur $\#SAT$ ne fournit une borne inférieure lorsqu'un dépassement de limite de temps ou d'espace a lieu.

Bilan Dans cette partie, nous avons bien mis en avant l'intérêt pratique de $\#EBTD$. Nous l'avons comparé à de multiples compteurs comme $\#RFL$, $\#BTD$, *cn2mddg*, *c2d*,

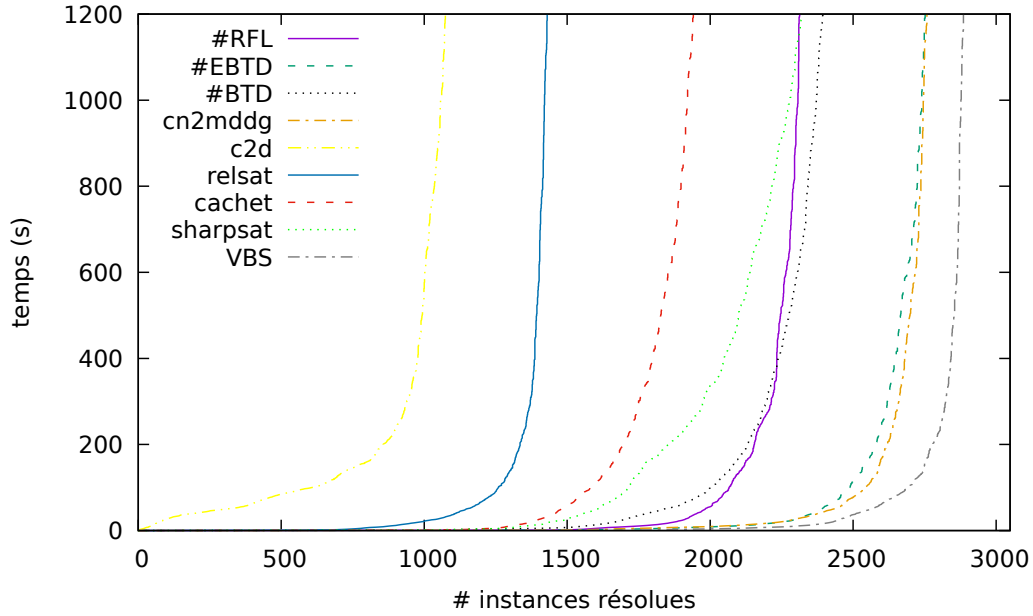


FIGURE 6.13 – Le nombre cumulé d'instances résolues pour tous les algorithmes pour le benchmark I_{tuple}.

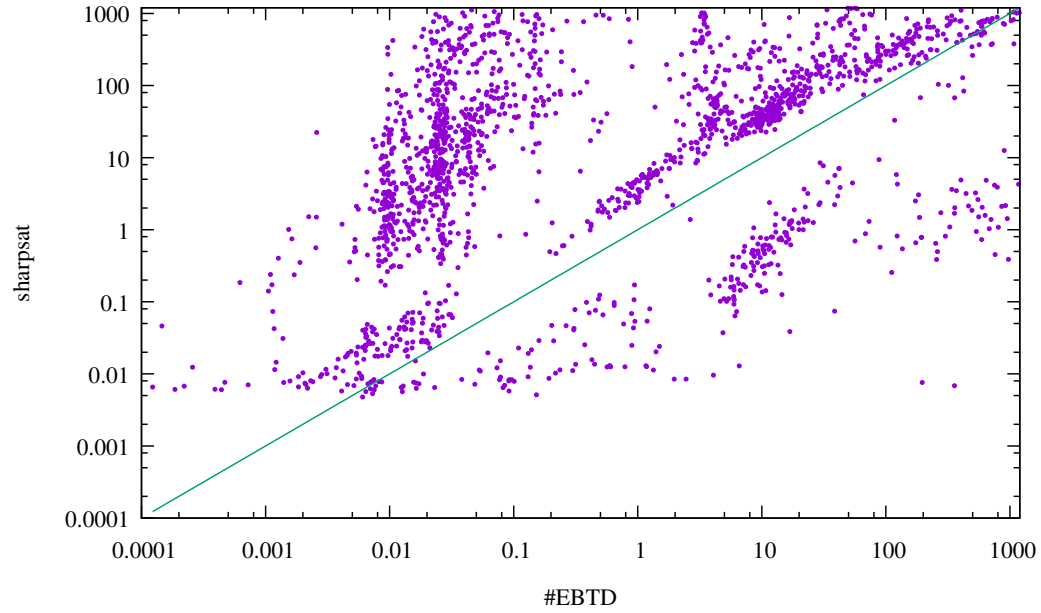


FIGURE 6.14 – Comparaison des temps de résolution de $\#EBTD$ et de sharpsat pour les 1 883 instances résolues à la fois par les deux algorithmes sur le benchmark I_{direct}.

relsat, *cachet* et *sharpsat*. La comparaison du comportement de $\#EBTD$ selon la décomposition utilisée révèle que les décompositions qui visent à minimiser w^+ ainsi que la décomposition H_2 sont les plus efficaces vis-à-vis du comptage du nombre exact de solutions d'une instance. Ainsi, la décomposition $Min\text{-}Fill\text{-}MG$ qui détériorait significativement l'efficacité de BTD pour la résolution d'instances CSP, prouve un intérêt majeur dans le cadre du comptage. Malgré la liberté de choix de variables dont profite $\#RFL$, la

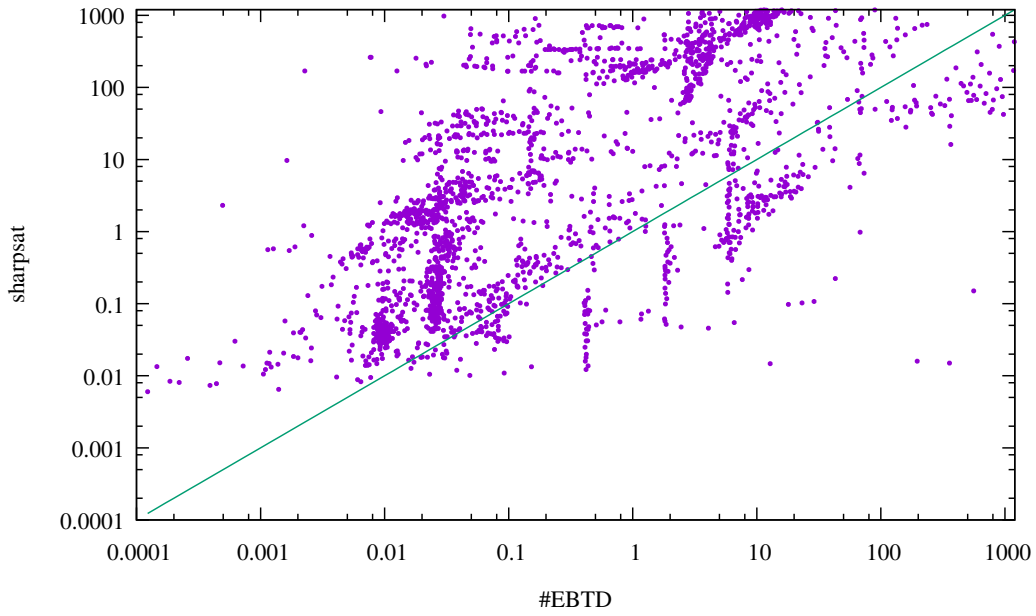


FIGURE 6.15 – Comparaison des temps de résolution de $\#EBTD$ et de sharpsat pour les 2 231 instances résolues à la fois par les deux algorithmes sur le benchmark I_{tuple}.

saturation de la capacité de l'énumération et la redondance des sous-espaces de recherche visités empêche souvent $\#RFL$ de résoudre des instances ayant un très grand nombre de solutions. Ces problèmes sont évités par une méthode à base de décomposition. Dans cette catégorie, la comparaison entre $\#BTD$ et $\#EBTD$ montre que $\#EBTD$ surpasse $\#BTD$ en nombre d'instances et en temps de résolution. Nous comparons aussi $\#EBTD$ au compilateur *cn*2*mddg* qui s'avère particulièrement efficace. L'avantage de $\#EBTD$ par rapport à *cn*2*mddg* est qu'il est capable de fournir une borne inférieure une fois le temps limite dépassé sans que le nombre exact de solutions soit compté. En plus, $\#EBTD$ est mis en valeur pour des instances difficilement résolues par *cn*2*mddg*. Ceci semble évident du fait que le calcul de la décomposition peut être coûteux ne justifiant pas son utilisation pour des instances dont le nombre de solutions peut être facilement compté par *cn*2*mddg*. Finalement, la comparaison de $\#EBTD$ aux #SAT solveurs montre que ces derniers sont clairement surclassés par $\#EBTD$. Elle met en avant aussi les problèmes de conversion entre le format CSP et le format SAT qui s'avère coûteuse en temps et en espace. Plus important, il semble que la structure des instances CSP sera moins présente lorsque cette dernière est convertie en format SAT.

6.5 Conclusion

Dans ce chapitre, nous avons abordé le problème du comptage. Ce problème est très intéressant, du fait des applications qui en découlent en intelligence artificielle et bien au-delà dans d'autres domaines plus éloignés de l'informatique comme la physique ou la chimie mais surtout parce que ce problème constitue un défi sur le plan théorique et le plan pratique. En effet, sa difficulté en théorie et en pratique nous incite à étudier ce problème de plus près. D'une part, des études théoriques ont été réalisées visant à analyser ce problème du point de vue de la complexité théorique en élaborant des classes traitables [Slivovsky and Szeider, 2013] ou en analysant leur difficulté par le biais des théorèmes de

dichotomie [Bulatov and Dalmau, 2003; Bulatov, 2008; Dyer and Richerby, 2013]. D'autre part, d'un point de vue pratique des méthodes de comptage ont été proposées. La difficulté de ce problème a orienté la plupart des travaux vers les méthodes d'approximation qui calculent soit une approximation du nombre de solutions, soit une borne inférieure sur le nombre exact de solutions [Wei and Selman, 2005; Gomes et al., 2006, 2007a; Gogate and Dechter, 2007; Gomes et al., 2007b; Kroc et al., 2008; Gogate and Dechter, 2008]. Malheureusement, même les méthodes d'approximation rencontrent des difficultés pour pouvoir fournir des approximations de qualité.

Ainsi, nous nous sommes intéressés dans ce chapitre aux méthodes exactes. Afin de proposer des méthodes efficaces en théorie et en pratique, nous exploitons les propriétés structurelles de l'instance. En particulier, nous nous concentrons sur le paramètre de la largeur arborescente car les instances ayant une tree-width bornée par une constante peuvent être résolues en temps polynomial. Le fait d'exploiter la structure du problème fournit des bornes de complexité en temps et en espace comme c'est le cas de $\#BTD$. En plus de son intérêt théorique, $\#BTD$ a prouvé son intérêt pratique en permettant de compter le nombre de solutions des instances ayant un nombre élevé - voire gigantesque - de solutions. Cet algorithme exploite une décomposition arborescente et évite certaines redondances en empêchant l'exploration répétée du même sous-espace de recherche grâce aux enregistrements auxquels il a recours.

Bien que certaines redondances soient évitées, la façon dont $\#BTD$ parcourt la décomposition induit des calculs inutiles qui peuvent être coûteux en temps et en espace. C'est pourquoi, nous avons proposé l'algorithme $\#EBTD$ qui vise à éviter ce défaut. Plus précisément, $\#BTD$ compte le nombre d'extensions d'une affectation pour un sous-problème sans la garantie que cette affectation partielle s'étend de façon cohérente sur tout le problème. Ainsi, l'objectif de $\#EBTD$ est de s'assurer qu'une affectation partielle admet au moins une extension cohérente sur toutes les variables du problème avant de calculer le nombre de solutions d'un sous-problème. Ce faisant, le nombre de solutions d'un sous-problème n'est calculé que si nécessaire.

$\#EBTD$ n'a pas vocation à améliorer la complexité théorique mais l'efficacité pratique. En effet, les expérimentations ont montré qu'il surclasse certains compteurs de l'état de l'art. En particulier, $\#EBTD$ surclasse $\#BTD$, $\#RFL$ et certains compteurs #SAT. La comparaison entre $\#EBTD$ et *cn2mddg* est plus intéressante notamment sur les instances difficilement résolues par *cn2mddg*. Finalement, à travers ces expérimentations, nous pouvons déduire que vu la difficulté du problème du comptage, les décompositions visant à minimiser la largeur w^+ sont désormais mises en valeur. En raison de la difficulté du problème, un temps de calcul de décompositions élevé peut être toléré contrairement au problème de décision. En plus, la minimisation du paramètre central de la complexité théorique semble constituer l'une des clés pour compter efficacement le nombre de solutions d'une instance.

Nous arrivons donc à un constat paradoxal. En effet, la décomposition doit minimiser w^+ et augmenter le nombre de séparateurs pour pouvoir permettre le dénombrement efficace des solutions. Cependant, la résolution doit aussi considérer des décompositions mieux adaptées à la recherche d'une première solution, celles-ci n'étant pas nécessairement celles minimisant la largeur. Concilier ces deux points de vue antagonistes peut être très prometteur et peut contribuer à améliorer considérablement l'efficacité des méthodes du comptage basées sur l'exploitation des décompositions.

Conclusion

La configuration usuelle d'un solveur consiste à répondre aux questions suivantes :

- Quel type de branchement faut-il ?
- Est-ce qu'un filtrage doit être exploité en prétraitement ou durant la résolution et si oui quel niveau de cohérence doit être maintenu ?
- Des enregistrements sont-ils employés ?
- Les retours en arrière réalisés sont-ils chronologiques ou non chronologiques ?
- Quelles heuristiques utiliser pour le choix de la prochaine variable et de la prochaine valeur ?
- Sera-il possible d'interrompre la résolution et de procéder à un redémarrage ?

Un solveur se caractérise par la réponse qu'il donne à chacune de ces questions.

Par ailleurs, nous avons vu tout au long de cette thèse que l'approche structurelle peut être particulièrement intéressante sur le plan théorique et sur le plan pratique. Contrairement aux méthodes de résolution énumératives classiques dont la complexité est exponentielle en n, les méthodes structurelles sont exponentielles en fonction de la notion de largeur accompagnant la décomposition capturant cette structure. En pratique, certaines décompositions comme la décomposition arborescente [Robertson and Seymour, 1986] ont montré que leur exploitation peut être fortement recommandée en raison de leur apport remarquable. Malheureusement, les solveurs exploitent rarement la structure et n'intègrent pas habituellement une telle « brique » pour gérer les mécanismes correspondants. Dans cette thèse, nous avons prêté attention à ce point. Nous avons essayé de montrer que son intégration aux solveurs et son utilisation à bon escient peut considérablement améliorer l'efficacité de la résolution. L'ambition de cette thèse est alors de faire que *l'exploitation de la structure devienne l'une des briques clés d'un solveur.*

Les choix des différents paramètres sont tous plus ou moins importants et stratégiques. En outre, le choix d'un paramètre peut influencer le choix des autres paramètres. Par exemple, le choix de l'heuristique de choix de variables *dom/wdeg* [Boussemart et al., 2004] suppose qu'un filtrage des valeurs des domaines est mis en place. Ainsi, les différents choix d'un solveur doivent être combinés adéquatement. Or, cette tâche semble difficile en pratique. Un solveur « idéal » défini par les choix « parfaits » qu'il fait n'existe pas à ce jour et n'existera probablement jamais. D'ailleurs, avoir réalisé les meilleurs choix pour la résolution d'une instance ne signifie pas qu'il en sera de même pour d'autres instances. Cependant, des solveurs efficaces sont disponibles aujourd'hui et permettent de résoudre une grande sélection d'instances réelles et académiques (voir la compétition CSP 2017 dans le cadre du problème CSP ou *Toulbar*2 dans le cadre du problème WCSP).

L'intégration de la brique *structure* au solveur nécessite de gérer un certain nombre de paramètres additionnels dont le choix de la décomposition utilisée. Comme nous cherchons actuellement à concevoir des solveurs dont l'emploi est simple, nous devons garantir qu'un bon choix de la décomposition peut être fait indépendamment du niveau d'expertise de l'utilisateur comme cela a été le cas pour les autres paramètres du solveur. Nous avons alors tenté dans cette thèse d'étudier les inconvénients découlant de la façon dont sont actuellement calculées les décompositions sur l'efficacité de la résolution. Nous avons constaté que :

- La décomposition calculée ne capture pas toujours des critères pertinents vis-à-vis de la résolution,
- La décomposition calculée en amont de la résolution n'est pas en mesure d'intégrer des critères adéquats aux besoins réels de la résolution,
- L'utilisation de la décomposition n'est pas toujours opportune vis-à-vis de la nature du problème traité comme dans le cas du problème du comptage.

Le solveur auquel nous aspirons devra être capable de faire des choix judicieux sur ces trois niveaux sans requérir la moindre intervention de la part de l'utilisateur. Un point crucial sur lequel nous avons insisté est l'*adaptation* de la décomposition au contexte de la résolution. L'adaptativité a été considérée pour d'autres briques dans les solveurs classiques comme pour l'heuristique de choix de variables. La proposition de l'heuristique adaptative *dom/wdeg*, par exemple, vise à prendre en compte le contexte de la résolution lors du choix de la prochaine variable à instancier. En ce qui concerne la brique structurelle, implémenter l'adaptativité s'avère primordial. En effet, le choix de la décomposition influence significativement la résolution vu son impact sur l'heuristique de choix de variables. Malheureusement, le choix de la décomposition au départ peut être peu pertinent et peut nécessiter une remise en cause pendant la résolution. Nous parlons ainsi d'une décomposition adaptative qui change tout au long de la résolution grâce aux nouvelles informations apprises. L'adaptativité renforce ainsi l'aspect « boîte noire » du solveur. L'utilisateur n'a pas ainsi la responsabilité de sélectionner une décomposition convenable pour la résolution d'une instance donnée.

Nous détaillons par la suite les résultats obtenus pour chaque contribution et les perspectives envisagées. Un point crucial que nous avons tenté de traiter est le choix de la décomposition utilisée pendant la résolution. Ce choix concerne le calcul de la décomposition fait en amont de la résolution (chapitre 3), mais aussi son évolution pendant la résolution (chapitres 4 et chapitre 5) pour les problème (W)CSP. Dans le chapitre 6, nous nous sommes concentrés sur l'utilisation à bon escient de la décomposition pour le problème #CSP.

Calcul de la décomposition arborescente (chapitre 3) Le nouveau cadre de calcul de décompositions H-TD a permis de montrer que la minimisation de w^+ sans recourir à la triangulation (H_1-TD) ou en employant une triangulation mieux guidée par la topologie (Min-$Fill$-MG) peut améliorer le temps de décomposition et/ou la qualité de l'estimation de la largeur arborescente. Il a également montré l'intérêt de la connexité des clusters (H_2-TD), du nombre de fils d'un cluster (H_3-TD) et la taille des séparateurs ($H_{4,5}$-TD) pour la résolution des instances (W)CSP. Au contraire, la minimisation de la largeur de la décomposition (comme dans le cas de Min-$Fill$, H_1-TD et Min-$Fill$-MG) s'est avérée moins pertinente pour la résolution d'instances (W)CSP. Finalement, ce cadre a augmenté la compétitivité des méthodes structurelles vis-à-vis des méthodes

énumératives classiques comme $MAC{+}RST$. Plusieurs pistes d'amélioration sont envisageables. Il serait intéressant d'étudier plus profondément les critères de la décomposition qui pourraient potentiellement être plus pertinents à l'égard de la résolution d'instances (W)CSP. En particulier, nous sommes intéressés par l'amélioration de la prise en compte de la sémantique de l'instance lors du calcul de la décomposition. Calculer des décompositions de meilleure qualité pourrait considérablement augmenter l'efficacité de la résolution vu son impact sur l'heuristique de choix de variables. En outre, certaines décompositions ne peuvent pas être calculées via $H\text{-}TD$ (au sens de l'ensemble de clusters pouvant être créés par ce cadre). La généralisation du cadre $H\text{-}TD$ afin de pouvoir calculer une plus grande variété de décompositions constitue une piste à explorer.

Fusion dynamique de la décomposition dans le cas du problème CSP (chapitre 4) L'emploi de la fusion dynamique pendant la résolution des instances CSP a permis d'améliorer la performance de la résolution vis-à-vis de l'exploitation statique de la décomposition en nombre d'instances résolues et en temps. Il a également montré qu'exploiter la décomposition H_5^∞ dans le cadre de la fusion dynamique améliore sensiblement l'efficacité de la résolution. En outre, la fusion dynamique a amélioré l'efficacité de la résolution par rapport à l'emploi de la fusion statique qui vise à « corriger » la décomposition en amont de la résolution. Finalement, la fusion dynamique des clusters a mis encore plus en valeur les méthodes structurelles par rapport aux méthodes classiques notamment pour les instances difficiles.

Pour de futurs travaux, nous proposons de mettre en œuvre d'autres heuristiques de fusion en exploitant plus d'informations récoltées pendant la résolution et qui relèveraient de la sémantique de l'instance à résoudre. Nous proposons aussi de tirer profit de la rapidité du calcul de décompositions offerte par $H\text{-}TD$ pour recalculer la décomposition durant la résolution, notamment lors des redémarrages, afin de ne pas se contenter de modifier la décomposition initiale.

Exploitation dynamique de la décomposition dans le cas du problème WCSP (chapitre 5) Dans le cadre du problème WCSP, l'exploitation « si besoin » de la décomposition a permis d'augmenter le nombre d'instances résolues et de diminuer en général significativement le temps de résolution. Elle a aussi montré qu'employer la décomposition H_5^{25} dynamiquement améliore les méthodes de référence pour la résolution d'instances WCSP [Allouche et al., 2015]. Finalement, l'exploitation si besoin de la décomposition permet d'obtenir de meilleures bornes inférieures et supérieures en cas de dépassement du temps limite vis-à-vis des méthodes de l'état de l'art.

Plusieurs extensions de ce travail sont pertinentes. Le calcul de la décomposition peut être à son tour fait dynamiquement et en cas de besoin. En fait, la décomposition n'est souvent utilisée que partiellement, voire jamais dans certains cas. Ainsi cela permettrait d'économiser le temps de calcul de la décomposition pour les sous-problèmes pour lesquels elle n'est pas utilisée. Plus important, le calcul dynamique de la décomposition permettrait d'obtenir des décompositions plus adaptées au contexte de la résolution. D'autres heuristiques d'exploitation dynamique de la décomposition dans le cadre du problème WCSP pourraient être testées.

Amélioration de #BTD dans le cas du problème #CSP (chapitre 6) L'amélioration de l'exploitation de la décomposition pour le comptage du nombre de solutions permet de surclasser $\#BTD$, $\#RFL$ et les compteurs #SAT. Elle permet d'obtenir des performances comparables au compilateur $cn2mddg$ sur les instances difficilement résolues

par ce dernier, voire meilleures. Finalement, l'importance de la minimisation de la largeur w^+ a été soulignée pour le comptage contrairement aux problèmes de décision et d'optimisation.

En termes de perspectives, nous proposons de considérer de nouvelles heuristiques de calcul de décompositions mieux adaptées au problème du comptage. En particulier, nous sommes intéressés par les décompositions minimisant w^+, maximisant le nombre de séparateurs tout en surveillant le besoin en espace mémoire. Une extension particulièrement intéressante est de mieux gérer les aspects contradictoires liés au calcul de la décomposition. En effet, la décomposition utilisée doit d'une part avoir une largeur minimum pour permettre de compter efficacement et d'autre part doit satisfaire les critères jugés pertinents pour trouver la première solution efficacement comme dans le cas du problème CSP. Finalement, nous proposons d'étudier, pour les instances où le comptage exact semble difficile, l'utilisation $\#EBTD$ pour fournir de meilleures approximations.

Un extension générale de ces travaux est d'aller au-delà de BTD. En effet, bien que la méthode BTD soit ici la méthode référence employée pour évaluer l'intérêt des nouvelles méthodes de calcul ou d'exploitation de la décomposition arborescente, les objectifs de la thèse ne se limitent pas à l'amélioration de BTD. Ainsi, une extension générale de ce travail est de l'appliquer aux autres méthodes structurelles comme celles basées sur la décomposition hyperarborescente [Gottlob et al., 1999], AND/OR search [Marinescu and Dechter, 2005b,a, 2007; Dechter and Mateescu, 2007], Bucket elimination [Dechter and Pearl, 1987; Dechter, 1999; Larrosa and Dechter, 2003], *recursive conditioning* [Darwiche, 2001c] ... Cette approche est particulièrement justifiée par les similitudes existantes entre les différentes méthodes notamment en termes de la liberté restreinte accordée à l'heuristique de choix de variables. En outre, malgré l'emploi de différentes notions de largeur, ces méthodes empruntent aux méthodes à base de décomposition arborescente la façon dont la décomposition est calculée comme c'est le cas avec l'utilisation de $Min\text{-}Fill$ pour calculer la décomposition hyperarborescente [Dermaku et al., 2008], par exemple.

En conclusion, dans cette thèse, notre ambition est d'intégrer la brique structurelle aux briques standards des solveurs actuels. Nous avons tenté d'améliorer les méthodes structurelles et de montrer qu'elles peuvent être compétitives vis-à-vis des méthodes classiques. Notre ambition est d'autant plus réaliste que l'intégration des méthodes telles que BTD s'avère être d'une simplicité remarquable. En effet, l'algorithme BTD peut être implémenté dans des bibliothèques existantes (comme *Gecode* [GEC, 2006], *Toulbar2* [TOU, 2006], *Choco* [CHO, 2008] ...) facilement en imposant des limitations sur l'heuristique de choix de variables selon la décomposition utilisée.

Bibliographie

(2006). Gecode : Generic Constraint Development Environment. http://www.gecode.org.

(2006). Toulbar2 : an open source exact cost function network solver. https://mulcyber.toulouse.inra.fr/projects/toulbar2/.

(2008). Choco : an open source java constraint programming library. http://choco-solver.net/.

(2008). Csp competition 2008. http://www.cril.univ-artois.fr/CPAI08.

(2016). Dimacs graph coloring instances for pace competition. http://mat.gsia.cmu.edu/COLOR/instances.html.

(2016). Luebeck university implementation for pace competition. https://github.com/maxbannach/Jdrasil.

(2016). Meiji university implementation for pace competition. https://github.com/TCS-Meiji/treewidth-exact.

(2016). Repository of control flow graphs for pace competition. https://github.com/freetdi/CFGs.

(2016). Repository of named graphs for pace competition. https://github.com/freetdi/named-graphs.

(2016). Utrecht university implementation for pace competition. https://github.com/TomvdZanden/BZTreewidth.

(2017). 2017 XCSP3 Competition. http://xcsp.org/competition.

(2017). XCSP3 series. http://xcsp.org/series.

Adler, I., Gottlob, G., and Grohe, M. (2007). Hypertree width and related hypergraph invariants. *European Journal of Combinatorics*, pages 2167–2181.

Allouche, D., Givry, S. D., Katsirelos, G., Schiex, T., and Zytnicki, M. (2015). Anytime hybrid best-first search with tree decomposition for weighted CSP. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 12–29.

Amir, E. (2001). Efficient approximation for triangulation of minimum treewidth. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 7–15.

Amir, E. (2010). Approximation algorithms for treewidth. *Algorithmica*, pages 448–479.

Angelsmark, O. and Jonsson, P. (2003). Improved algorithms for counting solutions in constraint satisfaction problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 81–95.

Apsel, U. and Brafman, R. I. (2012). Lifted MEU by Weighted Model Counting. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 1861–1867.

Apt, K. (2003). *Principles of constraint programming*. Cambridge University Press.

Arnborg, S., Corneil, D., and Proskurowski, A. (1987). Complexity of finding embeddings in ak-tree. *SIAM Journal on Algebraic Discrete Methods*, pages 277–284.

Arnborg, S., Lagergren, J., and Seese, D. (1991). Easy problems for tree-decomposable graphs. *Journal of Algorithms*, pages 308–340.

Aschinger, M., Drescher, C., Gottlob, G., Jeavons, P., and Thorstensen, E. (2011). Structural decomposition methods and what they are good for. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 12–28.

Austrin, P., Pitassi, T., and Wu, Y. (2012). Inapproximability of treewidth, one-shot pebbling, and related layout problems. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 13–24. Springer.

Bacchus, F., Dalmao, S., and Pitassi, T. (2002). Value elimination : Bayesian inference via backtracking search. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 20–28.

Bacchus, F. and Grove, A. (1995). On the forward checking algorithm. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 292–309.

Bachoore, E. H. and Bodlaender, H. L. (2005). New upper bound heuristics for treewidth. In *International Workshop on Experimental and Efficient Algorithms*, pages 216–227.

Bachoore, E. H. and Bodlaender, H. L. (2006). A branch and bound algorithm for exact, upper, and lower bounds on treewidth. In *International Conference on Algorithmic Applications in Management (AAIM)*, pages 255–266.

Baget, J. and Tognetti, Y. S. (2001). Backtracking through biconnected components of a constraint graph. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 291–296.

Bailleux, O. and Chabrier, J. (1996). Approximate resolution of hard numbering problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 169–174.

Baptista, L., Lynce, I., and Marques-Silva, J. P. (2001). Complete search restart strategies for satisfiability. In *In IJCAI Workshop on Stochastic Search Algorithms*.

Bayardo, J. R. and Pehoushek, J. D. (2000). Counting models using connected components. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 157–162.

Bayardo, R. J. and Miranker, D. P. (1996). A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 298–304.

Beek, P. V. (2006). Backtracking search algorithms. *Foundations of Artificial Intelligence*, pages 85–134.

Beeri, C., Fagin, R., Maier, D., and Yannakakis, M. (1983). On the desirability of acyclic database schemes. *Journal of the ACM (JACM)*, pages 479–513.

Beldiceanu, N., Carlsson, M., Demassey, S., and Petit, T. (2007). Global constraint catalogue : Past, present and future. *Constraints*, pages 21–62.

Beldiceanu, N., Carlsson, M., and Rampon, J. (2005). Global constraint catalog. `http://sofdem.github.io/gccat/`.

Berg, J. and Järvisalo, M. (2014). SAT-Based Approaches to Treewidth Computation : An Evaluation. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 328–335.

Berge, C. (1973). *Graphes et hypergraphes*. Elsevier.

Berry, A., Blair, J. R., Heggernes, P., and Peyton, B. W. (2004). Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, pages 287–298.

Berry, A., Bordat, J., Heggernes, P., Simonet, G., and Villanger, Y. (2006). A wide-range algorithm for minimal triangulation from an arbitrary ordering. *Journal of Algorithms*, pages 33–66.

Berry, A., Heggernes, P., and Simonet, G. (2003a). The minimum degree heuristic and the minimal triangulation process. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 58–70.

Berry, A., Heggernes, P., and Villanger, Y. (2003b). A vertex incremental approach for dynamically maintaining chordal graphs. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 47–57.

Bertele, U. and Brioschi, F. (1972). *Nonserial dynamic programming*. Academic Press.

Bessiere, C. (1994). Arc-consistency and arc-consistency again. *Artificial intelligence*, pages 179–190.

Bessiere, C. (2006). Constraint propagation. *Foundations of Artificial Intelligence*, pages 29–83.

Bessière, C., Chmeiss, A., and Sais, L. (2001). Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 565–569.

Bessiere, C., Hebrard, E., Hnich, B., and Walsh, T. (2007). The complexity of reasoning with global constraints. *Constraints*, pages 239–259.

Bessière, C., Meseguer, P., Freuder, E. C., and Larrosa, J. (2002). On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, pages 205–224.

Bessiere, C. and Régin, J. (1996). MAC and combined heuristics : Two reasons to forsake FC (and CBJ ?) on hard problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 61–75.

Bessiere, C. and Régin, J. (1997). Arc consistency for general constraint networks : preliminary results. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Bessière, C. and Régin, J. (2001). Refining the Basic Constraint Propagation Algorithm. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 309–315.

Bessière, C., Régin, J., Yap, R. H., and Zhang, Y. (2005). An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, pages 165–185.

Bibel, W. (1988). Constraint satisfaction from a deductive viewpoint. *Artificial Intelligence*, pages 401–413.

Biere, A. (2008). Adaptive restart control for conflict driven SAT solvers. *Proceedings of the International conference on theory and applications of satisfiability testing (SAT)*.

Biere, A., Heule, M., and van Maaren, H. (2009). *Handbook of satisfiability*. IOS press.

Birnbaum, E. and Lozinskii, E. L. (1999). The good old Davis-Putnam procedure helps counting models. *Journal of Artificial Intelligence Research (JAIR)*, pages 457–477.

Bitner, J. R. and Reingold, E. M. (1975). Backtrack programming techniques. *Communications of the ACM*, pages 651–656.

Blair, J. R., Heggernes, P., and Telle, J. A. (2001). A practical algorithm for making filled graphs minimal. *Theoretical Computer Science*, pages 125–141.

Blet, L. (2015). *Configuration automatique d'un solveur generique integrant des techniques de decomposition arborescente pour la resolution de problemes de satisfaction de contraintes*. PhD thesis, INSA de Lyon.

Bodlaender, H. (1996). A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, pages 1305–1317.

Bodlaender, H. L., Drange, P. G., Dregi, M. S., Fomin, F. V., Lokshtanov, D., and Pilipczuk, M. (2016). A $c^k n$ 5-Approximation Algorithm for Treewidth. *SIAM Journal on Computing*, pages 317–378.

Bodlaender, H. L., Fomin, F. V., Koster, A. M., Kratsch, D., and Thilikos, D. M. (2006). On exact algorithms for treewidth. In *Proceedings The Annual European Symposium on Algorithms (ESA)*, pages 672–683.

Bodlaender, H. L., Fomin, F. V., Koster, A. M., Kratsch, D., and Thilikos, D. M. (2012). On exact algorithms for treewidth. *ACM Transactions on Algorithms (TALG)*, page 12.

Bodlaender, H. L., Gilbert, J. R., Hafsteinsson, H., and Kloks, T. (1995). Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms*, pages 238–255.

Bodlaender, H. L. and Koster, A. M. (2010). Treewidth computations I. Upper bounds. *Information and Computation*, pages 259–275.

Bodlaender, H. L. and Koster, A. M. (2011). Treewidth computations II. Lower bounds. *Information and Computation*, pages 1103–1119.

Bondy, J. A. and Murty, U. S. R. (1976). *Graph theory with applications*. Springer London.

Boole, G. (1854). *An investigation of the laws of thought*. Cambridge University Press.

Bouchitté, V., Kratsch, D., Müller, H., and Todinca, I. (2004). On treewidth approximations. *Discrete Applied Mathematics*, pages 183–196.

Bouchitté, V. and Todinca, I. (2001). Treewidth and minimum fill-in : Grouping the minimal separators. *SIAM Journal on Computing*, pages 212–232.

Bouchitté, V. and Todinca, I. (2002). Listing all potential maximal cliques of a graph. *Theoretical Computer Science*, pages 17–32.

Boussemart, F., Hemery, F., Lecoutre, C., and Sais, L. (2004). Boosting Systematic Search by Weighting Constraints. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 146–150.

Boussemart, F., Lecoutre, C., and Piette, C. (2016). XCSP3 : An integrated format for benchmarking combinatorial constrained problems. *arXiv*.

Brandstädt, A., Le, V. B., and Spinrad, J. P. (1999). *Graph classes : a survey*. SIAM.

Brélaz, D. (1979). New methods to color the vertices of a graph. *Communications of the ACM*, pages 251–256.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *Transactions on Computers*, pages 677–691.

Bui-Xuan, B., Telle, J., and Vatshelle, M. (2011). Boolean-width of graphs. *Theoretical Computer Science*, pages 5187–5204.

Bulatov, A. A. (2008). The complexity of the counting constraint satisfaction problem. In *The International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 646–661.

Bulatov, A. A. and Dalmau, V. (2003). Towards a dichotomy theorem for the counting constraint satisfaction problem. In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*, pages 562–571.

Buneman, P. (1974). A characterisation of rigid circuit graphs. *Discrete mathematics*, pages 205–212.

Burton, R. and Steif, J. E. (1994). Non-uniqueness of measures of maximal entropy for subshifts of finite type. *Ergodic Theory and Dynamical Systems*, pages 213–235.

Cabon, B., Givry, S. D., Lobjois, L., Schiex, T., and Warners, J. P. (1999). Radio link frequency assignment. *Constraints*, pages 79–89.

Cambazard, H., Hadzic, T., and O'Sullivan, B. (2010). Knowledge Compilation for Itemset Mining. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 1109–1110.

Chavira, M. and Darwiche, A. (2008). On probabilistic inference by weighted model counting. *Artificial Intelligence*, pages 772–799.

Chen, X. (2000). *A Theoretical Comparison of Selected Csp Solving and Modeling Techniques*. PhD thesis, University of Alberta.

Chmeiss, A. and Jégou, P. (1998). Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, pages 121–142.

Choi, A., Kisa, D., and Darwiche, A. (2013). Compiling probabilistic graphical models using sentential decision diagrams. In *Proceedings of European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 121–132.

Clautiaux, F., Moukrim, A., Nègre, S., and Carlier, J. (2004). Heuristic and metaheuristic methods for computing graph treewidth. *RAIRO-Operations Research*, pages 13–26.

Cohen, D., Jeavons, P., and Gyssens, M. (2008). A unified theory of structural tractability for constraint satisfaction problems. *Journal of Computer and System Sciences*, pages 721–743.

Cohen, D. A., Jeavons, P., and Gyssens, M. (2005). A Unified Theory of Structural Tractability for Constraint Satisfaction and Spread Cut Decomposition. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 72–77.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Annual ACM symposium on Theory of computing*, pages 151–158.

Cooper, M. and Schiex, T. (2004). Arc consistency for soft constraints. *Artificial Intelligence*, pages 199–227.

Cooper, M. C. (1989). An optimal k-consistency algorithm. *Artificial Intelligence*, pages 89–95.

Cooper, M. C. (2003). Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, pages 311–342.

Cooper, M. C., Givry, S. D., Sanchez, M., Schiex, T., and Zytnicki, M. (2008). Virtual Arc Consistency for Weighted CSP. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 253–258.

Cooper, M. C., Givry, S. D., Sánchez, M., Schiex, T., Zytnicki, M., and Werner, T. (2010). Soft arc consistency revisited. *Artificial Intelligence*, pages 449–478.

Cooper, M. C., Givry, S. D., and Schiex, T. (2007). Optimal Soft Arc Consistency. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 68–73.

Courcelle, B. (1990). The monadic second order theory of Graphs I : Recognisable 662 sets of finite graphs. *Information and Computation*, page 663.

Courcelle, B. and Olariu, S. (2000). Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, pages 77–114.

Dahlhaus, D. (1997). Minimal elimination ordering inside a given chordal graph. In *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 132–143.

Darwiche, A. (2001a). Decomposable negation normal form. *Journal of the ACM (JACM)*, pages 608–647.

Darwiche, A. (2001b). On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, pages 11–34.

Darwiche, A. (2001c). Recursive conditioning. *Artificial Intelligence*, pages 5–41.

Darwiche, A. (2004). New advances in compiling CNF into decomposable negation normal form. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 328–332.

Darwiche, A. (2009). *Modeling and reasoning with Bayesian networks*. Cambridge University Press.

Darwiche, A. (2011). SDD : A new canonical representation of propositional knowledge bases. In *SDD : A new canonical representation of propositional knowledge bases*, page 819.

Darwiche, A. and Marquis, P. (2001). A perspective on knowledge compilation. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 175–182.

Darwiche, A. and Marquis, P. (2002). A knowledge compilation map. *Journal of Artificial Intelligence Research (JAIR)*, pages 229–264.

Davies, J. and Bacchus, F. (2007). Using more reasoning to improve #SAT solving. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, page 185.

Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, pages 394–397.

Debruyne, R. and Bessiere, C. (1997a). From restricted path consistency to max-restricted path consistency. *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 312–326.

Debruyne, R. and Bessiere, C. (1997b). Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Dechter, R. (1986). Learning while searching in constraint-satisfaction problems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.

Dechter, R. (1990). Enhancement schemes for constraint processing : Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, pages 273–312.

Dechter, R. (1992). *Constraint networks*. Information and Computer Science, University of California, Irvine.

Dechter, R. (1996). Topological parameters for time-space tradeoff. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 220–227.

Dechter, R. (1999). Bucket elimination : A unifying framework for reasoning. *Artificial Intelligence*, pages 41–85.

Dechter, R. (2003). *Constraint processing*. Morgan Kaufmann.

Dechter, R. (2006). Tractable structures for constraint satisfaction problems. *Foundations of Artificial Intelligence*, pages 209–244.

Dechter, R. (2013). Reasoning with probabilistic and deterministic graphical models : Exact algorithms. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, pages 1–191.

Dechter, R. and Fattah, Y. E. (2001). Topological Parameters for Time-space Tradeoff. *Artificial Intelligence*, pages 93–118.

Dechter, R. and Mateescu, R. (2004). The impact of AND/OR search spaces on constraint satisfaction and counting. *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 731–736.

Dechter, R. and Mateescu, R. (2007). AND/OR search spaces for graphical models. *Artificial intelligence*, pages 73–106.

Dechter, R. and Meiri, I. (1989). *Experimental evaluation of preprocessing techniques in constraint satisfaction problems*. UCLA, Computer Science Department.

Dechter, R. and Meiri, I. (1994). Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, pages 211–241.

Dechter, R. and Pearl, J. (1986). *The cycle-cutset method for improving search performance in AI applications*. University of California, Computer Science Department.

Dechter, R. and Pearl, J. (1987). Network-based heuristics for constraint-satisfaction problems. *Artificial intelligence*, pages 1–38.

Dechter, R. and Pearl, J. (1989). Tree clustering for constraint networks. *Artificial Intelligence*, pages 353–366.

Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J., and Schaus, P. (2016). Compact-table : Efficiently filtering table constraints with reversible sparse bit-sets. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 207–223.

Dent, M. J. and Mercer, R. E. (1994). Minimal forward checking. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 432–438.

Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B., Musliu, N., and Samer, M. (2008). Heuristic methods for hypertree decomposition. In *Proceedings of the Mexican International Conference on Artificial Intelligence (MICAI)*, pages 1–11.

Diestel, R. and Müller, M. (2017). Connected tree-width. *Combinatorica*.

Díez, F. J. (1996). Local conditioning in Bayesian networks. *Artificial Intelligence*, pages 1–20.

Dirac, G. (1961). On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburg 25*, pages 71–76.

Domshlak, C. and Hoffmann, J. (2006). Fast Probabilistic Planning through Weighted Model Counting. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 243–252.

Dubois, D., Fargier, H., and Prade, H. (1993). The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proceedings of the International Conference on Fuzzy Systems*, pages 1131–1136. IEEE.

Dyer, M. and Richerby, D. (2013). An effective dichotomy for the counting constraint satisfaction problem. *SIAM Journal on Computing*, pages 1245–1274.

Eén, N. and Sörensson, N. (2003). An extensible SAT-solver. In *Proceedings of the International conference on theory and applications of satisfiability testing (SAT)*, pages 502–518.

Even, S. (1979). *Graph algorithms*. Cambridge University Press.

Fargier, H. and Lang, J. (1993). Uncertainty in constraint satisfaction problems : a probabilistic approach. In *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 97–104.

Fargier, H., Lang, J., Martin-Clouaire, R., and Schiex, T. (1995). A constraint satisfaction framework for decision under uncertainty. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 167–174.

Fargier, H., Lang, J., and Schiex, T. (1993). Selecting preferred solutions in fuzzy constraint satisfaction problems. In *Proceedings of the European Congress on Fuzzy and Intelligent Technologies (EUFIT)*.

Fargier, H. and Marquis, P. (2009). Knowledge Compilation Properties of Trees-of-BDDs, Revisited. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 772–777.

Favier, A., Givry, S. D., and Jégou, P. (2009). Exploiting Problem Structure for Solution Counting. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 335–343.

Feige, U., Hajiaghayi, M., and Lee, J. R. (2008). Improved approximation algorithms for minimum weight vertex separators. *SIAM Journal on Computing*, pages 629–657.

Fellows, M. R. and Downey, R. G. (1999). *Parameterized complexity*. New York : Springer.

Flum, J. and Grohe, M. (2006). *Parameterized complexity theory*. Springer Science & Business Media.

Fomin, F., Kratsch, D., and Todinca, I. (2004). Exact (exponential) algorithms for treewidth and minimum fill-in. In *The International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 568–580.

Fomin, F. V. and Villanger, Y. (2012). Treewidth computation and extremal combinatorics. *Combinatorica*, pages 289–308.

Freuder, E. and Quinn, M. (1985). Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1076–1078.

Freuder, E. C. (1978). Synthesizing constraint expressions. *Communications of the ACM*, pages 958–966.

Freuder, E. C. (1982). A sufficient condition for backtrack-free search. *Journal of the ACM (JACM)*, pages 24–32.

Freuder, E. C. (1990). Complexity of K-Tree Structured Constraint Satisfaction Problems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.

Freuder, E. C. and Elfe, C. . D. (1996). Neighborhood inverse consistency preprocessing. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 202–208.

Freuder, E. C. and O'Sullivan, B. (2014). Grand challenges for constraint programming. *Constraints*, pages 150–162.

Freuder, E. C. and Wallace, R. J. (1992). Partial constraint satisfaction. *Artificial Intelligence*, pages 21–70.

Freuder, E. C. and Wallace, R. J. (1995). Generalizing Inconsistency Learning for Constraint Satisfaction. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 563–571.

Frost, D. and Dechter, R. (1994). Dead-end driven learning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 294–300.

Frost, D. and Dechter, R. (1995). Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 572–578.

Fukunaga, A. S. (2003). Complete restart strategies using a compact representation of the explored search space. *In IJCAI Workshop on Stochastic Search Algorithms*.

Fulkerson, D. and Gross, O. (1965). Incidence matrices and interval graphs. *Pacific journal of mathematics*, pages 835–855.

Gajarskỳ, J., Lampis, M., and Ordyniak, S. (2013). Parameterized Algorithms for Modular-Width. In *International Symposium IPEC*, pages 163–176.

Gale, D. and Sotomayor, M. (1985). Semiring-based constraint solving and optimization. *American Mathematical Monthly*, pages 261–268.

Gaschnig, J. (1979). Performance measurement and analysis of certain search algorithms. Technical report, Department Of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Gavril, F. (1974). The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, pages 47–56.

Geelen, P. (1992). Dual Viewpoint Heuristics for. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, volume 92, pages 31–35.

Gent, I. P., Jefferson, C., and Miguel, I. (2006). Minion : A fast scalable constraint solver. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 98–102.

Gent, I. P., Jefferson, C., Miguel, I., and Nightingale, P. (2007). Data structures for generalised arc consistency for extensional constraints. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 191–197.

Gergov, J. and Meinel, C. (1994). Efficient Boolean Manipulation with OBDD's Can be Extended to FBDD's. *Transactions on Computers*, pages 1197–1209.

Ginsberg, M. L. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research (JAIR)*, pages 25–46.

Givry, S. D., Heras, F., Zytnicki, M., and Larrosa, J. (2005). Existential arc consistency : Getting closer to full arc consistency in weighted CSPs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 5, pages 84–89.

Givry, S. D., Schiex, T., and Verfaillie, G. (2006). Exploiting Tree Decomposition and Soft Local Consistency In Weighted CSP. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 22–27.

Glorian, G., Lagniez, J. M., Boussemart, F., Lecoutre, C., and Mazure, B. (2017). Combinaison de nogoods extraits au redémarrage. In *Actes des Journées Francophones de Programmation par Contraintes (JFPC)*, pages 55–64.

Gogate, V. and Dechter, R. (2004). A complete anytime algorithm for treewidth. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 201–208.

Gogate, V. and Dechter, R. (2007). Approximate counting by sampling the backtrack-free search space. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, page 198.

Gogate, V. and Dechter, R. (2008). Approximate solution sampling (and counting) on and/or spaces. *Lecture Notes in Computer Science*, page 534.

Golomb, S. W. and Baumert, L. D. (1965). Backtrack programming. *Journal of the ACM (JACM)*, pages 516–524.

Golumbic, M. C. (2004). *Algorithmic graph theory and perfect graphs*. Elsevier.

Gomes, C. P., Hoffmann, J., Sabharwal, A., and Selman, B. (2007a). From Sampling to Model Counting. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2293–2299.

Gomes, C. P., Sabharwal, A., and Selman, B. (2006). Model counting : A new strategy for obtaining good bounds. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 54–61.

Gomes, C. P., Selman, B., Crato, N., and Kautz, H. (2000). Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, pages 67–100.

Gomes, C. P., Selman, B., and Kautz, H. (1998). Boosting combinatorial search through randomization. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 431–437.

Gomes, C. P., van Hoeve, W. J., Sabharwal, A., and Selman, B. (2007b). Counting CSP solutions using generalized XOR constraints. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 204–209.

Gottlob, G., Hutle, M., and Wotawa, F. (2002). Combining hypertree, bicomp, and hinge decomposition. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 161–165.

Gottlob, G., Leone, N., and Scarcello, F. (1999). Hypertree decompositions and tractable queries. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 21–32.

Gottlob, G., Leone, N., and Scarcello, F. (2000). A comparison of structural CSP decomposition methods. *Artificial Intelligence*, pages 243–282.

Graham, M. H. (1980). On the universal relation. Technical report, University of Toronto.

Grohe, M. and Marx, D. (2006). Constraint solving via fractional edge covers. In *Proceedings of the annual ACM-SIAM symposium on Discrete algorithm*, pages 289–298.

Grohe, M. and Marx, D. (2014). Constraint solving via fractional edge covers. *ACM Transactions on Algorithms (TALG)*, page 4.

Gyssens, M., Jeavons, P., and Cohen, D. (1994). Decomposing constraint satisfaction problems using database techniques. *Artificial intelligence*, pages 57–89.

Gyssens, M. and Paredaens, J. (1982). A Decomposition Methodology for Cyclic Databases. *Advances in data base theory*, pages 85–122.

Habbas, Z., Amroun, K., and Singer, D. (2016). Generalized Hypertree Decomposition for solving non binary CSP with compressed table constraints. *RAIRO-Operations Research*, pages 241–267.

Hajnal, A. and Surányi, J. (1958). Über die Auflösung von Graphen in vollständige Teilgraphen. *Ann. Univ. Sci. Budapest, Eötvös Sect. Math*, pages 113–121.

Hamann, M. and Weißauer, D. (2016). Bounding connected tree-width. *SIAM Journal on Discrete Mathematics*, pages 1391–1400.

Hammerl, T., Musliu, N., and Schafhauser, W. (2015). Metaheuristic algorithms and tree decomposition. In *Handbook of Computational Intelligence*, pages 1255–1270. Springer.

Han, C. and Lee, C. (1988). Comments on Mohr and Henderson's path consistency algorithm. *Artificial Intelligence*, pages 125–130.

Haralick, R. M. and Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, pages 263–313.

Harvey, W. D. (1995). *Nonsystematic backtracking search*. PhD thesis, Stanford university.

Harvey, W. D. and Ginsberg, M. L. (1995). Limited discrepancy search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 607–615.

Heggernes, P. (2006). Minimal triangulations of graphs : A survey. *Discrete Mathematics*, pages 297–317.

Heggernes, P., Telle, J. A., and Villanger, Y. (2005). Computing minimal triangulations in time O $(n\alpha \log n)$= O $(n^{2.376})$. In *Proceedings of the Annual ACM-SIAM symposium on Discrete algorithms*, pages 907–916.

Heggernes, P. and Villanger, Y. (2002). Efficient implementation of a minimal triangulation algorithm. In *Proceedings The Annual European Symposium on Algorithms (ESA)*, pages 175–176.

Held, M. and Karp, R. M. (1962). A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, pages 196–210.

Hentenryck, P. V., Deville, Y., and Teng, C. (1992). A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, pages 291–321.

Hoos, H. H. and Stützle, T. (2004). *Stochastic local search : Foundations and applications*. Elsevier.

Hoos, H. H. and Tsang, E. (2006). Local search methods. *Foundations of Artificial Intelligence*, pages 135–167.

Huang, J. and Darwiche, A. (2004). Using DPLL for efficient OBDD construction. In *Proceedings of the International conference on theory and applications of satisfiability testing (SAT)*, pages 157–172.

Hurley, B., O'Sullivan, B., Allouche, D., Katsirelos, G., Schiex, T., Zytnicki, M., and Givry, S. D. (2016). Multi-language evaluation of exact solvers in graphical model discrete optimization. *Constraints*, pages 413–434.

Hwang, J. and Mitchell, D. . G. (2005). 2-way vs. d-way branching for CSP. *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 343–357.

Janssen, P., Jégou, P., Nouguier, B., and Vilarem, M. (1989). A filtering process for general constraint-satisfaction problems : achieving pairwise-consistency using an associated binary representation. In *Proceedings of the IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427.

Jarnìk, V. (1930). About a certain minimal problem. *Práce Moravské Prırodovedecké Spolecnosti*, pages 57–63.

Jégou, P. (1990). Cyclic-clustering : a compromise between tree-clustering and cycle-cutset method for improving search efficiency. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 369–371.

Jégou, P. (1993). On the consistency of general constraint-satisfaction problems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 114–119.

Jégou, P., Kanso, H., and Terrioux, C. (2015a). An Algorithmic Framework for Decomposing Constraint Networks. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1–8.

Jégou, P., Kanso, H., and Terrioux, C. (2015b). De nouvelles approches pour la décomposition de réseaux de contraintes. *Actes des Journées Francophones de Programmation par Contraintes (JFPC)*, pages 140–149.

Jégou, P., Kanso, H., and Terrioux, C. (2016a). Improving Exact Solution Counting for Decomposition Methods. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 327–334.

Jégou, P., Kanso, H., and Terrioux, C. (2016b). Towards a Dynamic Decomposition of CSPs with Separators of Bounded Size. In *Proceedings of the International Conference on Principle and Practice of Constraint Programming (CP)*, pages 298–315.

Jégou, P., Kanso, H., and Terrioux, C. (2016c). Vers une décomposition dynamique des réseaux de contraintes. In *Actes des Journées Francophones de Programmation par Contraintes (JFPC)*, pages 123–132.

Jégou, P., Kanso, H., and Terrioux, C. (2017a). Adaptive and Opportunistic Exploitation of Tree-decompositions for Weighted CSPs. Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI). To appear.

Jégou, P., Kanso, H., and Terrioux, C. (2017b). Vers une exploitation dynamique de la décomposition pour les CSPs pondérés. *Actes des Journées Francophones de Programmation par Contraintes (JFPC).*

Jégou, P., Ndiaye, S., and Terrioux, C. (2009). Combined strategies for decomposition-based methods for solving CSPs. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 184–192.

Jégou, P., Ndiaye, S. N., and Terrioux, C. (2005). Computing and exploiting tree-decompositions for solving constraint networks. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 777–781.

Jégou, P., Ndiaye, S. N., and Terrioux, C. (2006a). An extension of complexity bounds and dynamic heuristics for tree-decompositions of CSP. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 741–745.

Jégou, P., Ndiaye, S. N., and Terrioux, C. (2006b). Strategies and heuristics for exploiting tree-decompositions of constraint networks. In *Inference methods based on graphical structures of knowledge (WIGSK'06), ECAI workshop*, pages 13–18.

Jégou, P., Ndiaye, S. N., and Terrioux, C. (2007). Dynamic Management of Heuristics for Solving Structured CSPs. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 364–378.

Jégou, P., Ndiaye, S. N., and Terrioux, C. (2008). A new Evaluation of Forward Checking and its Consequences on Efficiency of Tools for Decomposition of CSPs. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 486–490.

Jégou, P. and Terrioux, C. (2003). Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, pages 43–75.

Jégou, P. and Terrioux, C. (2004a). A time-space trade-off for constraint networks decomposition. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 234–239.

Jégou, P. and Terrioux, C. (2004b). Decomposition and good recording for solving Max-CSPs. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 196–200.

Jégou, P. and Terrioux, C. (2014a). Combining Restarts, Nogoods and Decompositions for Solving CSPs. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 465–470.

Jégou, P. and Terrioux, C. (2014b). Tree-Decompositions with Connected Clusters for Solving Constraint Networks. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 407–423.

Jégou, P. and Terrioux, C. (2017). Combining restarts, nogoods and bag-connected decompositions for solving CSPs. *Constraints*, pages 191–229.

Jensen, F. V., Lauritzen, S. L., and Olesen, K. G. (1990). Bayesian updating in causal probabilistic networks by local computations. *Computational statistics quarterly*, pages 269–282.

Jeong, J., Sæther, S., and Telle, J. (2015). Maximum matching width : new characterizations and a fast algorithm for dominating set. *arXiv*.

Jlifi, B. and Ghédira, K. (2003). On the enhancement of the informed backtracking algorithm. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 967–967.

Jlifi, B. and Ghédira, K. (2004). A Study of Backtracking Based Informed Algorithms. In *Proceedings of the Starting AI Researchers'Symposium (STAIRS)*, page 147.

Johnson, D. S. (1973). Approximation algorithms for combinatorial problems. In *Proceedings of the Annual ACM symposium on Theory of computing*, pages 38–49.

Jordan, M. I. (2004). Graphical models. *Statistical Science*, pages 140–155.

Junker, U. (2006). *Configuration*, chapter 24, pages 837–868. Handbook of constraint programming [Rossi et al., 2006]. Elsevier.

Jussien, N., Debruyne, R., and Boizumault, P. (2000). Maintaining arc-consistency within dynamic backtracking. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 249–261.

Karakashian, S. (2013). *Practical tractability of CSPs by higher level consistency and tree decomposition*. PhD thesis, The University of Nebraska-Lincoln.

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer.

Karp, R. M. and Luby, M. (1985). Monte-Carlo algorithms for the planar multiterminal network reliability problem. *Journal of Complexity*, pages 45–64.

Kask, K., Dechter, R., and Gogate, V. (2004). Counting-based look-ahead schemes for constraint satisfaction. *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 317–331.

Katsirelos, G. and Bacchus, F. (2003). Unrestricted nogood recording in CSP search. *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 873–877.

Katsirelos, G. and Bacchus, F. (2005). Generalized nogoods in CSPs. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 390–396.

Kitching, M. and Bacchus, F. (2008). Exploiting decomposition in constraint optimization problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 478–492.

Kjaerulff, U. (1990). Triangulation of Graphs - Algorithms Giving Small Total State Space. Technical report, Judex R.R. Aalborg, Denmark.

Kolaitis, P. G. and Vardi, M. Y. (1998). Conjunctive-query containment and constraint satisfaction. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 205–213.

Koller, D. and Friedman, N. (2009). *Probabilistic graphical models : principles and techniques*. MIT press.

Koriche, F., Lagniez, J., Marquis, P., and Thomas, S. (2013). Knowledge Compilation for Model Counting : Affine Decision Trees. In *SDD : A new canonical representation of propositional knowledge bases*, pages 947–953.

Koriche, F., Lagniez, J., Marquis, P., and Thomas, S. (2015). Compiling Constraint Networks into Multivalued Decomposable Decision Graphs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 332–338.

Koster, A. M., Bodlaender, H. L., and Hoesel, S. P. V. (2001). Treewidth : computational experiments. *Electronic Notes in Discrete Mathematics*, pages 54–57.

Koster, A. M. C. A. (1999). *Frequency assignment : Models and algorithms*. PhD thesis, University of Maastricht.

Kovalevsky, V. and Koval, V. (1975). A diffusion algorithm for decreasing energy of max-sum labeling problem. Glushkov Institute of Cybernetics, Kiev, USSR.

Kratsch, D. and Spinrad, J. (2006). Minimal fill in $O(n^{2.69})$ time. *Discrete mathematics*, pages 366–371.

Kroc, L., Sabharwal, A., and Selman, B. (2008). Leveraging belief propagation, backtrack search, and statistics for model counting. *Proceedings of the International Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming (CPAIOR)*, pages 127–141.

Kumar, T. K. (2002). A model counting characterization of diagnoses. Technical report, Knowledge Systems Laboratory Stanford University.

Kwan, A. C. and Tsang, E. P. (1996). Minimal forward checking with backmarking and conflict-directed backjumping. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 290–298.

Lagergren, J. (1996). Efficient parallel algorithms for graphs of bounded tree-width. *Journal of Algorithms*, pages 20–44.

Lagniez, J. and Marquis, P. (2017a). An Improved Decision-DNNF Compiler. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Lagniez, J. and Marquis, P. (2017b). On preprocessing techniques and their impact on propositional model counting. *Journal of Automated Reasoning*, pages 413–481.

Lagniez, J., Marquis, P., and Paparrizou, A. (2017). Defining and Evaluating Heuristics for the Compilation of Constraint Networks. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 172–188.

Land, A. H. and Doig, A. G. (1960). An automatic method of solving discrete programming problems. *Econometrica : Journal of the Econometric Society*, pages 497–520.

Larrosa, J. (2000). Boosting search with variable elimination. *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 291–305.

Larrosa, J. (2002). Node and arc consistency in weighted CSP. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 48–53.

Larrosa, J. and Dechter, R. (2003). Boosting search with variable elimination in constraint optimization and constraint satisfaction problems. *Constraints*, pages 303–326.

Larrosa, J., Meseguer, P., and Sánchez, M. (2002). Pseudo-tree search with soft constraints. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 131–135.

Larrosa, J. and Schiex, T. (2003). In the quest of the best form of local consistency for weighted CSP. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 239–244.

Larrosa, J. and Schiex, T. (2004). Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, pages 1–26.

Lauritzen, S. L. (1996). *Graphical models*. Clarendon Press.

Lawler, E. L. and Wood, D. E. (1966). Branch-and-bound methods : A survey. *Operations research*, 14(4) :699–719.

Lecoutre, C. (2011). STR2 : optimized simple tabular reduction for table constraints. *Constraints*, pages 341–371.

Lecoutre, C. (2013). *Constraint Networks : Targeting Simplicity for Techniques and Algorithms*. John Wiley & Sons.

Lecoutre, C., Boussemart, F., and Hemery, F. (2003). Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 480–494.

Lecoutre, C., Cardon, S., and Vion, J. (2007a). Conservative dual consistency. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 237–242.

Lecoutre, C., Cardon, S., and Vion, J. (2007b). Path consistency by dual consistency. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 438–452.

Lecoutre, C., Cardon, S., and Vion, J. (2011). Second-order consistencies. *Journal of Artificial Intelligence Research (JAIR)*, pages 175–219.

Lecoutre, C., Hemery, F., et al. (2007c). A Study of Residual Supports in Arc Consistency. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 125–130.

Lecoutre, C., Likitvivatanavong, C., and Yap, R. H. (2015). STR3 : A path-optimal filtering algorithm for table constraints. *Artificial Intelligence*, pages 1–27.

Lecoutre, C., Sais, L., Tabary, S., and Vidal, V. (2007d). Nogood Recording from Restarts. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 131–136.

Lecoutre, C., Sais, L., Tabary, S., and Vidal, V. (2007e). Recording and Minimizing Nogoods from Restarts. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, pages 147–167.

Lecoutre, C., Saïs, L., Tabary, S., and Vidal, V. (2009). Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, pages 1592–1614.

Lecoutre, C. and Szymanek, R. (2006). Generalized arc consistency for positive table constraints. *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 284–298.

Lecoutre, C. and Vion, J. (2008). Enforcing arc consistency using bitwise operations. *Constraint Programming Letters (CPL)*, pages 21–35.

Lee, J. H., Schulte, C., and Zhu, Z. (2016). Increasing Nogoods in Restart-Based Search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 3426–3433.

Lee, J. H. and Zhu, Z. (2014). An increasing-nogoods global constraint for symmetry breaking during search. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 465–480.

Lekkeikerker, C. and Boland, J. (1962). Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae*, pages 45–64.

Lerman, I. and Rouat, V. (1999). Segmentation de la sériation pour la résolution de# SAT. *Mathematiques Informatique et Sciences Humaines*, pages 113–134.

Lhomme, O. and Régin, J. (2005). A fast arc consistency algorithm for n-ary constraints. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 405–410.

Liberatore, P. (2000). On the complexity of choosing the branching literal in DPLL. *Artificial intelligence*, pages 315–326.

Luby, M., Sinclair, A., and Zuckerman, D. (1993). Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, pages 173–180.

Lynce, I., Baptista, L., and Marques-Silva, J. (2001). Stochastic systematic search algorithms for satisfiability. *Electronic Notes in Discrete Mathematics*, pages 190–204.

Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial intelligence*, pages 99–118.

Mairy, J., Hentenryck, P. V., and Deville, Y. (2014). Optimal and efficient filtering algorithms for table constraints. *Constraints*, pages 77–120.

Mann, M., Tack, G., and Will, S. (2007). Decomposition during search for propagation-based constraint solvers. *arXiv*.

Marinescu, R. and Dechter, R. (2005a). Advances in and/or branch-and-bound search for constraint optimization. In *Proceedings of the International Workshop on Preferences and Soft Constraints*.

Marinescu, R. and Dechter, R. (2005b). AND/OR branch-and-bound for graphical models. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 224–229.

Marinescu, R. and Dechter, R. (2007). Best-first AND/OR search for graphical models. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 1171–1176.

Markowitz, H. P. (1957). The elimination form of the inverse and its application to linear programming. *Management Science*, pages 255–269.

Marques Silva, J. P., Lynce, I., and Malik, S. (2009). *Conflict-Driven Clause Learning SAT Solvers*, pages 131–153. Handbook of Satisfiability. IOS Press.

Mehta, D. and van Dongen, M. (2004). Two new lightweight arc consistency algorithms. In *Proceedings of the International Workshop on Constraint Propagation and Implementation (CPAI)*, pages 109–123.

Meseguer, P. and Sánchez, M. (2000). Tree-based Russian doll search : Preliminary results. In *Proceedings of the CP Workshop on Soft Constraints*.

Meseguer, P. and Sánchez, M. (2001). Specializing russian doll search. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 464–478.

Meseguer, P., Sánchez, M., and Verfaillie, G. (2002). Opportunistic specialization in russian doll search. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 264–279.

Michel, L. and Hentenryck, P. V. (2012). Activity-based search for black-box constraint programming solvers. In *Proceedings of the International Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming (CPAIOR)*, pages 228–243.

Minton, S., Johnston, M. D., Philips, A. B., and Laird, P. (1992). Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, pages 161–205.

Mitchell, D. G. (2003). Resolution and constraint satisfaction. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 555–569.

Mohr, R. and Henderson, T. C. (1986). Arc and path consistency revisited. *Artificial intelligence*, pages 225–233.

Montanari, U. (1974). Networks of constraints : Fundamental properties and applications to picture processing. *Information sciences*, pages 95–132.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff : Engineering an efficient SAT solver. In *Proceedings of the Annual Design Automation Conference*, pages 530–535.

Muise, C. J., McIlraith, S. A., Beck, J. C., and Hsu, E. I. (2012). Dsharp : Fast d-DNNF Compilation with sharpSAT. In *Proceedings of the Canadian Conference on Artificial Intelligence*, pages 356–361.

Müller, M. (2012). Connected tree-width. *arXiv*.

Nadel, B. A. (1988). Tree search and arc consistency in constraint satisfaction algorithms. In *Search in artificial intelligence*, pages 287–342. Springer.

Nešetřil, J. and Mendez, P. (2012). Bounded Height Trees and Tree-Depth. *Sparsity*, pages 115–144.

Niedermeier, R. (2006). *Invitation to fixed-parameter algorithms*. Oxford Press.

Nielsen, T. D. and Jensen, F. V. (2009). *Bayesian networks and decision graphs*. Springer Science & Business Media.

Ohtsuki, T. (1976). A fast algorithm for finding an optimal ordering for vertex elimination on a graph. *SIAM Journal on Computing*, pages 133–145.

Ohtsuki, T., Cheung, L. K., and Fujisawa, T. (1976). Minimal triangulation of a graph and optimal pivoting order in a sparse matrix. *Journal of Mathematical Analysis and Applications*, pages 622–633.

Otten, L. and Dechter, R. (2012). Anytime AND/OR depth-first search for combinatorial optimization. *AI Communications*, pages 211–227.

Oum, S. and Seymour, P. (2006). Approximating clique-width and branch-width. *Journal of Combinatorial Theory, Series B*, pages 514–528.

Oztok, U. and Darwiche, A. (2014). On compiling CNF into decision-DNNF. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 42–57.

Palacios, H., Bonet, B., Darwiche, A., and Geffner, H. (2005). Pruning Conformant Plans by Counting Models on Compiled d-DNNF Representations. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 141–150.

Parter, S. (1961). The use of linear graphs in Gauss elimination. *SIAM review*, pages 119–130.

Pearl, J. (1984). *Heuristics : Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Pearl, J. (1988). Probabilistic reasoning in intelligent systems : Networks of plausible inference.

Pearl, J. (2014). *Probabilistic reasoning in intelligent systems : networks of plausible inference*. Morgan Kaufmann.

Pearson, J. and Jeavons, P. G. (1997). A survey of tractable constraint satisfaction problems. Technical report, Royal Holloway, University of London.

Perez, G. and Régin, J. (2014). Improving GAC-4 for table and MDD constraints. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 606–621.

Pesant, G. (2005). Counting solutions of CSPs : A structural approach. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 260–265.

Pesant, G., Quimper, C., and Zanarini, A. (2012). Counting-based search : Branching heuristics for constraint satisfaction problems. *Journal of Artificial Intelligence Research (JAIR)*.

Peyton, B. W. (2001). Minimal orderings revisited. *SIAM journal on matrix analysis and applications*, pages 271–294.

Pinto, C. and Terrioux, C. (2009). A generalized Cyclic-Clustering Approach for Solving Structured CSPs. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 724–728.

Pipatsrisawat, K. and Darwiche, A. (2009). Width-Based Restart Policies for Clause-Learning Satisfiability Solvers. In *Proceedings of the International conference on theory and applications of satisfiability testing (SAT)*, pages 341–355.

Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, pages 268–299.

Puget, J. (2004). The next challenge for CP : Ease of use. In *Invited Talk at CP-2004*.

Raghavendra, P. and Steurer, D. (2010). Graph expansion and the unique games conjecture. In *Proceedings of the ACM symposium on Theory of computing*, pages 755–764.

Reed, B. A. (1992). Finding approximate separators and computing tree width quickly. In *Proceedings of the Annual ACM symposium on Theory of computing*, pages 221–228.

Refalo, P. (2004). Impact-based search strategies for constraint programming. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 557–571.

Régin, J. (1996). Generalized arc consistency for global cardinality constraint. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 209–215.

Richards, E. T. and Richards, B. (1996). Nogood learning for constraint satisfaction. In *Proceedings of the CP Workshop on Constraint Programming Applications*.

Robertson, N. and Seymour, P. D. (1983). Graph minors. I. Excluding a forest. *Journal of Combinatorial Theory, Series B*, pages 39–61.

Robertson, N. and Seymour, P. D. (1986). Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, pages 309–322.

Robertson, N. and Seymour, P. D. (1995). Graph minors. XIII. The disjoint paths problem. *Journal of combinatorial theory, Series B*, pages 65–110.

Rose, D. J. (1972). A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. *Graph theory and computing*, page 217.

Rose, D. J., Tarjan, R. E., and Lueker, G. S. (1976). Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on computing*, pages 266–283.

Rosenfeld, A., Hummel, R. A., and Zucker, S. W. (1976). Scene labeling by relaxation operations. *IEEE Transactions on Systems, Man, and Cybernetics*, pages 420–433.

Rossi, F., Beek, P. V., and Walsh, T. (2006). *Handbook of constraint programming*. Elsevier.

Roth, D. (1996). On the hardness of approximate reasoning. *Artificial Intelligence*, pages 273–302.

Roussel, O. and Lecoutre, C. (2009). Xml representation of constraint networks : Format xcsp 2.1. *arXiv*.

Ruppert, D. (2001). Probabilistic networks and expert systems.

Ruttkay, Z. (1994). Fuzzy constraint satisfaction. In *Proceedings of the International Conference on Fuzzy Systems*, pages 1263–1268.

Ryvchin, V. and Strichman, O. (2008). Local restarts in SAT. *Constraint Programming Letters (CPL)*, pages 3–13.

Sabin, D. and Freuder, E. (1994). Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*, pages 125–129.

Sabin, D. and Freuder, E. C. (1997). Understanding and improving the MAC algorithm. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 167–181.

Samer, M. and Szeider, S. (2011). Tractable cases of the extended global cardinality constraint. *Constraints*, pages 1–24.

Samer, M. and Veith, H. (2009). Encoding treewidth into SAT. In *Proceedings of the International conference on theory and applications of satisfiability testing (SAT)*, pages 45–50.

Sanchez, M., Allouche, D., Givry, S. D., and Schiex, T. (2009). Russian Doll Search with Tree Decomposition. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 603–608.

Sandholm, T. (2002). Algorithm for optimal winner determination in combinatorial auctions. *Artificial intelligence*, pages 1–54.

Sang, T., Bacchus, F., Beame, P., Kautz, H. A., and Pitassi, T. (2004). Combining Component Caching and Clause Learning for Effective Model Counting. In *Proceedings of the International conference on theory and applications of satisfiability testing (SAT)*.

Sang, T., Beame, P., and Kautz, H. A. (2005). Performing Bayesian inference by weighted model counting. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 475–481.

Schiex, T. (1992). Possibilistic constraint satisfaction problems or how to handle soft constraints ? In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 268–275.

Schiex, T. (2000). Arc consistency for soft constraints. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 411–425.

Schiex, T., Fargier, H., and Verfaillie, G. (1995). Valued constraint satisfaction problems : Hard and easy problems. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 631–639.

Schiex, T. and Verfaillie, G. (1993). Two approaches to the solution maintenance problem in dynamic constraint satisfaction problems. In *Proceedings of the IJCAI Workshop on Knowledge-based Production Planning, Scheduling and Control*.

Schiex, T. and Verfaillie, G. (1994a). Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, pages 187–207.

Schiex, T. and Verfaillie, G. (1994b). Stubborness : A Possible Enhancement for Backjumping and Nogood Recording. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 165–172.

Seymour, P. and Thomas, R. (1994). Call routing and the ratcatcher. *Combinatorica*, pages 217–241.

Shachter, R. D., Andersen, S. K., and Szolovits, P. (1994). Global conditioning for probabilistic inference in belief networks. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 514–522.

Shafer, G. R. and Shenoy, P. P. (1990). Probability propagation. *Annals of Mathematics and Artificial Intelligence*, pages 327–351.

Shoikhet, K. and Geiger, D. (1997). A practical algorithm for finding optimal triangulations. In *Proceedings of the National Conference on Artificial Intelligence (AAAI) and of the Conference on Innovative Applications of Artificial Intelligence (IAAI)*, pages 185–190.

Slivovsky, F. and Szeider, S. (2013). Model counting for formulas of bounded clique-width. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 677–687.

Smith, B. (1999). The Brélaz heuristic and optimal static orderings. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 405–418.

Smith, B. M. (1994). The phase transition and the mushy region in constraint satisfaction problems. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 100–104.

Smith, B. M. and Sturdy, P. (2005). Value ordering for finding all solutions. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 311–316.

Stallman, R. M. and Sussman, G. J. (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial intelligence*, pages 135–196.

Subbarayan, S., Bordeaux, L., and Hamadi, Y. (2007). Knowledge compilation properties of tree-of-BDDs. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, page 502.

Tarjan, R. E. and Yannakakis, M. (1984). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on computing*, pages 566–579.

Terrioux, C. and Jégou, P. (2003). Bounded backtracking for the valued constraint satisfaction problems. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 709–723.

Thurley, M. (2006). sharpSAT-counting models with advanced component caching and implicit BCP. In *Proceedings of the International conference on theory and applications of satisfiability testing (SAT)*, pages 424–429.

Toda, S. (1991). PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, pages 865–877.

Ullmann, J. R. (1966). Associating parts of patterns. *Information and Control*, pages 583–601.

Ullmann, J. R. (1976). An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, pages 31–42.

Ullmann, J. R. (2007). Partition search for non-binary constraint satisfaction. *Information Sciences*, pages 3639–3678.

Valiant, L. G. (1979). The complexity of computing the permanent. *Theoretical computer science*, pages 189–201.

van Hoeve, W. and Katriel, I. (2006). *Global constraints*, pages 245–280. Handbook of constraint programming [Rossi et al., 2006]. Elsevier.

Verfaillie, G. (1993). Problemes de satisfaction de contraintes : production et révision de solution par modifications locales. In *Proceedings of the International Avignon Workshop*, pages 277–286.

Verfaillie, G., Lemaître, M., and Schiex, T. (1996). Russian doll search for solving constraint optimization problems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 181–187.

Villanger, Y. (2006). Improved exponential-time algorithms for treewidth and minimum fill-in. In *Proceedings of the Latin American Theoritical Informatics Symposium (LATIN)*, pages 800–811.

Walsh, T. (1999). Search in a small world. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1172–1177.

Walsh, T. (2000). SAT v CSP. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, pages 441–456.

Waltz, D. L. (1972). Generating Semantic Descriptions From Drawings of Scenes With Shadows. Technical report, Cambridge, MA, USA.

Wang, R., Xia, W., Yap, R. H., and Li, Z. (2016). Optimizing Simple Tabular Reduction with a Bitwise Representation. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 787–795.

Wei, W. and Selman, B. (2005). A new approach to model counting. In *Proceedings of the International conference on theory and applications of satisfiability testing (SAT)*, pages 324–339.

Wollan, P. (2015). The structure of graphs not admitting a fixed immersion. *Journal of Combinatorial Theory, Series B*, pages 47–66.

Yannakakis, M. (1981). Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic Discrete Methods*, pages 77–79.

Zhang, Y. and Yap, R. H. (2001). Making AC-3 an optimal algorithm. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 316–321.

Résumé

L'importance des problèmes CSP, WCSP et #CSP est reflétée par la part considérable des travaux, théoriques et pratiques, dont ils font l'objet en intelligence artificielle et bien au-delà. Leur difficulté est telle qu'ils appartiennent respectivement aux classes NP-complet, NP-difficile et #P-complet. Aussi, les méthodes qui permettent de résoudre efficacement leurs instances ont une complexité en temps exponentielle. Les travaux de recherche de cette thèse se focalisent sur les méthodes de résolution exploitant la notion de décomposition arborescente. Ces méthodes ont suscité un vif intérêt de la part de la communauté scientifique du fait qu'elles soient capables de résoudre en temps polynomial certaines classes d'instances. Cependant, en pratique, elles n'ont pas encore montré toute leur efficacité vu la qualité de la décomposition employée ne prenant en compte qu'un critère purement structurel, sa largeur. Premièrement, nous proposons un nouveau cadre général de calcul de décompositions qui a la vertu de calculer des décompositions qui capturent des paramètres plus pertinents à l'égard de la résolution que la seule largeur de la décomposition. Ensuite, nous proposons une exploitation dynamique de la décomposition pendant la résolution pour les problèmes (W)CSP. Le changement de la décomposition pendant la résolution vise à adapter la décomposition selon la nature de l'instance. Finalement, nous proposons un nouvel algorithme de comptage qui exploite la décomposition d'une façon différente de celle des méthodes standards afin d'éviter des calculs inutiles. L'ensemble des contributions ont été évaluées et validées expérimentalement.

Mots clés : CSP, WCSP, #CSP, intelligence artificielle, décomposition arborescente, classes traitables, résolution.

Abstract

The importance of CSP, WCSP and #CSP problems is reflected by the considerable amount of theoretical and practical work of which they are subject in artificial intelligence and far beyond. Their difficulty is such that they belong respectively to the NP-complete, NP-hard and #P-complete classes. Hence, the methods that are able to solve efficiently their instances have a complexity in exponential time. The research works of this thesis focus on the solving methods exploiting the notion of tree-decomposition. These methods have aroused a keen interest from the scientific community because they are able to solve some classes of instances in polynomial time. Nevertheless, in practice, they have not shown yet their full efficiency given the quality of the used decomposition that takes only into account a purely structural criterion, its width. First, we propose a new generic framework for computing decompositions which has the virtue of computing decompositions that capture more relevant parameters in the context of solving than the width. Then, we propose a dynamic exploitation of the decomposition during the solving for (W)CSP problems. The modification of the decomposition during the solving aims to adapt the decomposition to the nature of the instance. Finally, we propose a new counting algorithm that exploits the decomposition in a different way than standard methods in order to avoid unnecessary computations. All the contributions have been evaluated and validated experimentally.

Keywords : CSP, WCSP, #CSP, artificial intelligence, tree-decomposition, tractable classes, solving.