Vers une décomposition dynamique des CSP

Philippe Jégou **Hanan Kanso** **Cyril Terrioux**

Aix-Marseille Université, CNRS, ENSAM, Université de Toulon, LSIS UMR 7296
13397 Marseille Cedex 20 (France)
{philippe.jegou, hanan.kanso, cyril.terrioux}@lsis.org

Résumé

Dans cet article, nous traitons deux questions clés relatives aux méthodes de résolution basées sur la décomposition arborescente de CSP. Tout d'abord, nous proposons un algorithme de calcul de décompositions qui permet de borner la taille des séparateurs, soit un paramètre crucial pour limiter la complexité en espace, et donc la faisabilité pratique de ce type de méthodes. Ensuite, nous montrons comment il est possible de modifier une décomposition dynamiquement pendant une résolution. Cette modification dynamique permet d'offrir plus de liberté aux heuristiques de choix de variables. Cela permet aussi un meilleur usage des informations obtenues pendant la résolution tout en contrôlant la taille de l'espace mémoire requis.

Abstract

In this paper, we address two key aspects of solving methods based on tree-decomposition. First, we propose an algorithm computing decompositions that allows to bound the size of separators, which is a crucial parameter to limit the space complexity, and thus the feasibility of such methods. Moreover, we show how it is possible to dynamically modify the considered decomposition during the search. This dynamic modification can offer more freedom to the variable ordering heuristics. This also allows to better use the information gained during the search while controlling the size of the required memory.

1 Introduction

Les méthodes de résolution basées sur la décomposition arborescente de CSP ont démontré leur intérêt théorique car elle permettent de garantir des bornes de complexité en temps en $O(exp(w))$ aussi bien qu'en espace en $O(exp(s))$ où w et s sont des paramètres induits par des propriétés structurelles des réseaux de contraintes. Ainsi, quand w est borné par une constante, ces méthodes assurent un temps de résolution polynomial. De plus, en pratique, ces approches sont particulièrement justifiées car il existe de nombreux problèmes réels pour lesquels w est relativement petit [6]. Cependant, deux problèmes majeurs se présentent souvent en pratique. Tout d'abord, le contrôle de la valeur de s n'est pas toujours garanti, en particulier pour des algorithmes de calcul de décompositions tels que $Min\text{-}Fill$ [22] qui peut cependant être considéré comme l'état de l'art [7]. Cela rend ce type d'approches parfois complètement inopérant en pratique car sur ce plan, ce paramètre est crucial [1]. De plus, assurer une complexité en temps en $O(exp(w))$ requiert un parcours de l'espace de recherche qui impose de fortes contraintes sur l'ordre d'affectation des variables, ce qui peut conduire à une très forte détérioration de l'efficacité pratique. Pour répondre à la question relative à l'espace mémoire, nous proposons un nouvel algorithme de calcul de décompositions qui est paramétrable. Il prend en entrée un paramètre S et va permettre un calcul de décomposition qui garantie une taille de séparateurs inférieure ou égale à S. Sa complexité en temps est inférieure à celle de $Min\text{-}Fill$ et il offre de plus des performances pratiques qui sont largement meilleures (environ 1000 fois plus rapide sur un large ensemble de benchmarks). La deuxième partie de l'article propose une approche qui sert à modifier dynamiquement une décomposition pendant une recherche, permettant ainsi d'offrir plus de liberté aux heuristiques tout en poursuivant l'exploitation des décompositions. Cette approche est motivée par le fait que pour être efficace en pratique, il est en général nécessaire de prendre en compte le contexte courant d'une recherche ainsi que les connaissances acquises progressivement pendant la résolution. Ce type d'approche existe au niveau des solveurs CSP qui utilisent des heuristiques basées sur des pondérations attachées aux contraintes comme par exemple pour *dom/wdeg* [4]. Elle est présente également au niveau des solveurs CDCL pour SAT [19] par le biais notamment des techniques d'apprentissage de clauses et de redémarrage.

Dans le cas des méthodes opérant par décomposition, la difficulté fondamentale porte sur l'ordre d'affectation des variables qui est imposé par la décomposition. Pour contourner cette difficulté, nous proposons d'adapter dynamiquement la décomposition pendant la résolution. La première idée concernant une telle approche a été introduite dans [11] mais principalement sur un plan théorique. Aussi, nous montrons comment une telle approche est finalement réalisable. Par ailleurs, nous étendons cette approche en intégrant les techniques de redémarrage comme proposé par [13]. De plus, nous décrivons comment modifier dynamiquement une décomposition, en prenant en compte les connaissances acquises pendant la résolution tout en proposant de conserver une borne sur la taille des séparateurs pendant la recherche.

La section 2 rappelle des notions sur les méthodes de résolution basées sur les décompositions arborescentes tandis que la section 3 présente un algorithme de calcul de décompositions arborescentes qui prend en compte la taille des séparateurs. La section 4 introduit une extension de l'algorithme BTD [12] visant à adapter la décomposition pendant la résolution. Enfin, avant de conclure, la section 5 présente des expérimentations pour évaluer la pertinence de cette approche.

2 Préliminaires

Une instance de CSP (pour Problème de Satisfaction de Contraintes) est définie par la donnée d'un triplet (X, D, C), où $X = \{x_1, \ldots, x_n\}$ est un ensemble de n variables, $D = \{d_{x_1}, \ldots, d_{x_n}\}$ est un ensemble de domaines finis, et $C = \{c_1, \ldots, c_e\}$ est un ensemble de e contraintes. Chaque contrainte c_i est un couple $(S(c_i), R(c_i))$, où $S(c_i) = \{x_{i_1}, \ldots, x_{i_k}\} \subseteq X$ définit la *portée* de c_i, et $R(c_i) \subseteq d_{x_{i_1}} \times \cdots \times d_{x_{i_k}}$ est une *relation de compatibilité*. L'*arité* d'une contrainte c_i est $|S(c_i)|$. Si l'arité de chaque contrainte est égale à 2, on parle alors de CSP *binaire*. La structure d'une instance CSP est donnée par un hypergraphe (un graphe pour les CSP binaires), appelé *(hyper)graphe* de contraintes, dont les sommets correspondent aux variables et les arêtes aux portées des contraintes. Pour simplifier les notations, nous notons l'hypergraphe $(X, \{S(c_1), \ldots S(c_e)\})$ par (X, C). Une affectation d'un sous-ensemble de X est dite *cohérente* si toutes les contraintes portant sur ce sous-ensemble sont satisfaites. Le problème du test d'existence d'une *solution* (i.e. une affectation cohérente de X) est bien connu pour être NP-complet. Aussi, de nombreux travaux ont été réalisés pour améliorer la résolution pratique comme ceux portant sur les heuristiques de choix de variables, l'apprentissage de contraintes, le backtracking non-chronologique, ou encore les techniques de filtrage. Néanmoins, la complexité de la résolution demeure exponentielle, au moins en $O(n.d^n)$ où d est la taille maximum des domaines. Pour contourner cette « intraitabilité » théorique, d'autres approches ont été proposées. Certaines d'elles s'appuient sur la notion de classes polynomiales structurelles [8] basées sur la notion de *décomposition arborescente de graphes* [21].

Définition 1 *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, F est un ensemble d'arêtes T) et $E = \{E_i : i \in I\}$ une famille de sous-ensembles de X, telle que chaque sous-ensemble (appelé cluster) E_i est un nœud 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 un chemin de i à j dans T, alors $E_i \cap E_j \subseteq E_k$. La largeur d'une décomposition arborescente 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.*

Cette notion est seulement définie pour des graphes mais elle peut être considérée pour des hypergraphes en exploitant leur *2-section*[1]. L'avantage des méthodes de résolution exploitant les décompositions arborescentes est d'abord lié à leur complexité théorique en temps d^{w+1} [7] alors que la complexité en espace est en d^s où s est la taille maximum des intersections (appelés *séparateurs* dans la suite) entre clusters. Ainsi, ces méthodes peuvent être efficaces sur des instances de grande taille et de petite tree-width comme c'est le cas, par exemple, pour les problèmes d'allocation de fréquences radio [5]. Ces méthodes procèdent en deux étapes : (1) calcul d'une décomposition arborescente, et (2) résolution d'une instance exploitant la décomposition. Puisque le calcul de décompositions optimales (i.e. de largeur w) est NP-difficile [2], en pratique, la première étape calcule en général des décompositions arborescentes de largeur $w^+ \geq w$, à savoir une approximation de la tree-width. Dans ce contexte, la méthode *Min-Fill* [22] apparaît comme le meilleur compromis entre le temps de calcul ($O(n^3)$) et la qualité des décompositions obtenues. Elle constitue la référence dans l'état de l'art pour de tels algorithmes [7], même si, pour des graphes de quelques dizaines de milliers de sommets, elle est en général inutilisable en pratique.

Cependant, les décompositions calculées ne sont pas nécessairement les plus adaptées du point de vue de la résolution [10, 14]. En premier lieu, *Min-Fill* ne prend pas en compte explicitement les propriétés structurelles des graphes considérés, ce qui peut rendre la résolution inefficace. Par exemple, les décompositions obtenues peuvent contenir des clusters non connexes ce qui peut conduire à une forte dégradation de l'efficacité

1. La 2-section d'un hypergraphe (X, C) est le graphe (X, C') où $C' = \{\{x, y\} | \exists c \in C, \{x, y\} \subseteq c\}$ [3].

de la résolution [14]. De plus, *Min-Fill* peut générer des décompositions telles que s est souvent proche de w^+. En effet, afin de minimiser la largeur, *Min-Fill* produit des clusters avec peu de sommets *propres* (i.e. des sommets appartenant à un cluster mais pas à son cluster parent dans la décomposition), voire même, un seul sommet propre. Cela peut mener à un coût qui s'avère prohibitif en termes d'espace mémoire. Ainsi, la minimisation de s est cruciale pour être efficace en pratique [10]. Par ailleurs, pour garantir la complexité en temps d^{w^+}, les méthodes structurelles efficaces comme BTD [12] utilisent un ordre d'affectation des variables qui est partiellement déterminé par la décomposition considérée. Quand *Min-Fill* est utilisé, cette liberté est même encore plus restreinte du fait du nombre limité de sommets propres dans les clusters. Or, il est bien connu que pour avoir une résolution efficace, il est souhaitable d'avoir une liberté maximale pour le choix des prochaines variables à affecter.

Pour contourner ces difficultés, plusieurs approches sont possibles. La première consiste à s'appuyer sur une décomposition disposant de petits séparateurs, tout en ayant de plus grands clusters. Cela permet de relâcher les contraintes portant sur l'ordre d'affectation [10]. Une autre possibilité porte sur l'exploitation des redémarrages comme proposé dans [13]. Cette approche fonctionne en redémarrant la recherche à partir d'une nouvelle première variable qui ne figure pas nécessairement dans le précédent cluster racine. Ceci conduit, tout en conservant la même décomposition (excepté en termes de racine), à donner plus de liberté à l'ordre. La pertinence d'une telle démarche a été démontrée expérimentalement dans [13]. Une autre possibilité consiste à modifier dynamiquement la décomposition pendant la résolution, tout en maintenant des garanties au niveau de la complexité temporelle. Cette approche a été introduite dans [11], mais principalement sur un plan théorique. Elle propose d'étendre la taille d'un cluster en le fusionnant avec un ou plusieurs clusters voisins. Toutefois, sa pertinence n'a jamais véritablement été démontrée. De plus, telle qu'elle a été introduite dans [11], elle n'est guidée que par des critères d'ordre structurel, sans prendre en compte explicitement l'état de la recherche, et surtout, les connaissances déjà acquises pendant la résolution. Notons que l'exploitation de la structure des instances de manière dynamique a déjà été proposée pour SAT dans [17]. Cela étant, dans ces travaux, aucune garantie ne pouvait être donnée au sujet des bornes de complexité, contrairement à ce que nous proposons ici.

Pour proposer des voies alternatives à ces précédents travaux, nous introduisons dans la suite d'abord un algorithme qui calcule des décompositions dont la taille des séparateurs est bornée. Ensuite, nous proposons un nouvel algorithme de résolution basé sur BTD et qui permet d'adapter dynamiquement les décompositions pendant la recherche en exploitant des informations obtenues durant la résolution.

3 Décomposer en bornant les séparateurs

Dans [9], nous avons proposé un cadre général pour calculer des décompositions arborescentes qui ne repose pas sur la notion de triangulation. Il permet d'implémenter différentes méthodes de décomposition selon les caractéristiques souhaitées pour la décomposition. Par exemple, l'algorithme *BC-TD* [14], qui calcule des décompositions dont tous les clusters sont connexes, est une variante possible de ce cadre. Avec la définition de ce cadre, nos objectifs sont multiples. D'abord, pour pouvoir traiter dynamiquement des décompositions, il faut disposer d'algorithmes efficaces, tant en termes de complexité théorique, que d'efficacité pratique. Ensuite, la complexité de ces algorithmes ne doit pas dépasser $O(n(n+e))$ afin d'être meilleure que celle de *Min-Fill*, ce qui peut être accompli en économisant une étape des plus coûteuses, à savoir la triangulation. Au-delà, il faut veiller à limiter la taille maximale des clusters, ce qui reste et demeure un paramètre important, tout en maîtrisant la taille des séparateurs.

Dans cette partie, nous rappelons le cadre *H-TD-WT* (pour *Heuristic Tree-Decomposition Without Triangulation*) qui calcule une décomposition arborescente du graphe $G = (X, C)$ sans triangulation en temps polynomial (à savoir en $O(n(n + e))$). Comme pour *Min-Fill*, aucune garantie n'est offerte quant à l'optimalité de la largeur obtenue. Tel que présenté dans [9], ce cadre permet cependant différentes paramétrisations en prenant en compte plusieurs critères dont certains sont relatifs à w^+ et/ou s. Nous nous intéressons ici à maîtriser la taille des séparateurs.

La première étape de *H-TD-WT* (ligne 1 dans l'algorithme 1) calcule un premier cluster, noté E_0, à l'aide d'une heuristique. X' qui représente l'ensemble des sommets déjà considérés est initialisé à E_0 (ligne 2). On notera par $X_1, X_2, \ldots X_k$ les composantes connexes du sous-graphe $G[X \backslash E_0]$ induit par la suppression dans G des sommets de E_0 [2]. Chacun de ces ensembles X_i est inséré dans une file d'attente F (ligne 4). Pour chaque élément X_i retiré de F (ligne 6), V_i note l'ensemble des sommets de X' qui sont adjacents à au moins un sommet de X_i (ligne 7). Dans l'exemple de la figure 1, pour $X_i = X_1$, on a $V_i = V_1 = \{x, y, z\}$.

On peut constater que V_i est un séparateur dans le graphe G puisque la suppression de V_i dans G déconnecte G (X_i étant déconnecté du reste de G). Nous considérons alors 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 8)

2. Le sous-graphe $G[Y]$ de $G = (X, C)$ induit par $Y \subseteq X$ est le graphe (Y, C_Y) où $C_Y = \{\{x, y\} \in C | x, y \in Y\}$.

Algorithme 1 : *H-TD-WT*

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

1 Choix d'un premier cluster E_0 dans G
2 $X' \leftarrow E_0$
3 Soient $X_1, \dots X_k$ les composantes connexes de $G[X \setminus E_0]$
4 $F \leftarrow \{X_1, \dots X_k\}$
5 **tant que** $F \neq \emptyset$ **faire** /* calcul d'un nouveau cluster E_i */
6 Enlever X_i de F
7 Soit $V_i \subseteq X'$ le voisinage de X_i dans G
8 Déterminer un sous-ensemble $X_i'' \subseteq X_i$ tel qu'il existe au moins un sommet $v \in V_i$ tel que $N(v, X_i) \subseteq X_i''$
9 $E_i \leftarrow X_i'' \cup V_i$
10 $X' \leftarrow X' \cup X_i''$
11 Soient $X_{i_1}, X_{i_2}, \dots X_{i_{k_i}}$ les composantes connexes de $G[X_i \setminus E_i]$
12 $F \leftarrow F \cup \{X_{i_1}, X_{i_2}, \dots X_{i_{k_i}}\}$

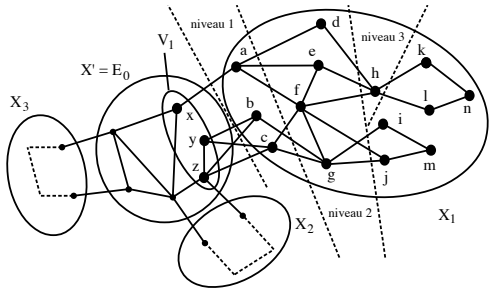


FIGURE 1 – Graphe illustrant H_5.

peut être paramétrée. Elle recherche un sous-ensemble de sommets $X_i'' \subseteq X_i$ tel que $X_i'' \cup V_i$ sera un nouveau cluster E_i de la décomposition. 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 $N(v, X_i) \subseteq X_i''$. Nous définissons alors un nouveau cluster $E_i = X_i'' \cup V_i$ (ligne 9). Puis, nous rajoutons à X' les sommets de X_i'' (ligne 10), avant de calculer les composantes connexes du sous-graphe issu de la suppression des sommets de E_i dans $G[X_i]$, composantes qui sont alors rajoutées à la file F (ligne 11). Ce processus est répété jusqu'à ce que la file F soit vide.

Dans [9], ce schéma était décliné selon quatre heuristiques. La première (notée H_1) vise à minimiser la taille des clusters alors que la seconde (H_2) garantit que tous les clusters sont connexes (voir [14]). La troisième (H_3) tente d'identifier des parties indépendantes dans le graphe et les sépare aussitôt que possible en effectuant un parcours en largeur à partir des sommets de V_i. La quatrième (H_4) tend à limiter la taille des séparateurs de la décomposition. Pour cela, elle considère un paramètre S qui représente la taille maximale autorisée pour un séparateur. Elle ajoute de nouveaux sommets au prochain cluster E_i de la même manière que H_3. Toutefois, la recherche est stoppée à un niveau $l = L$ dès que les toutes les composantes connexes de $G[X_i \setminus E_{i_L}]$ induites sont déconnectées du reste du graphe par des séparateurs de taille au plus S.

Nous introduisons ici une nouvelle heuristique, naturellement notée H_5. Elle vise à raffiner l'heuristique H_4 en détectant plus de séparateurs de taille au plus S. Là où H_4 doit atteindre un niveau de la recherche en largeur à partir duquel tous les séparateurs sont de taille au plus S, H_5 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'exploration, exceptée bien sûr, la partie associée à cette nouvelle composante connexe.

Nous illustrons le fonctionnement de H_5 sur l'exemple de la figure 1. Nous évoquons en premier le calcul de E_1, le second cluster (après E_0) lors du premier passage dans la boucle et nous fixons la valeur du paramètre S à 2. On considère donc l'ensemble $V_1 = \{x, y, z\}$. Les sommets a, b et c constituent le premier niveau. Aucun sous-ensemble de $\{a, b, c\}$ de taille 2 ou moins n'induit de séparateur. Le niveau suivant, constitué des sommets d, e, f et g, est donc visité. À ce niveau, on dispose de 2 séparateurs minimaux, d'une part $\{d, e, f\}$ et d'autre part $\{f, g\}$. Avec l'heuristique H_4, la recherche en largeur se poursuivrait. Avec H_5, elle va certes se poursuivre, mais l'existence du séparateur $\{f, g\}$ 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 auront été enlevés les sommets i, j et m car il ne sont plus connectés au reste des autres sommets. Ainsi, l'ensemble $\{i, j, m\}$ va être inséré dans la fille d'attente F. Comme ceci ne figure pas explicitement dans le schéma de l'algorithme, il faut considérer que cet ajout fait partie intégrante du traitement réalisé à la ligne 8. La recherche en profondeur se poursuit donc mais sur la partie de X_1 qui n'a pas encore été parcourue, et dont $\{i, j, m\}$ a été supprimé, soit $\{h, k, l, n\}$. On constate alors que le niveau suivant, uniquement constitué du sommet h constitue un séparateur de taille 1, donc inférieur à la borne fixée $S = 2$. Le parcours en largeur est stoppé, et le nouveau cluster est maintenant constitué : $E_1 = \{x, y, z, a, b, c, d, e, f, g, h\}$ et $X_1'' = \{a, b, c, d, e, f, g, h\}$. On obtient une composante connexe $X_{1_1} = \{k, l, n\}$ qui est ajoutée à F.

En fait, si on revient au schéma général de l'algorithme *H-TD-WT*, la version associée à l'heuristique H_5 respecte bien les conditions imposées par l'approche, à savoir que l'on construit bien un sous-ensemble $X_i'' \subseteq X_i$ tel qu'il existe au moins un sommet $v \in V_i$ tel que $N(v, X_i) \subseteq X_i''$. Cette observation, conjuguée à la preuve figurant dans [9], permet de s'assurer de la validité de H_5 :

Théorème 1 *H_5 calcule les clusters d'une décomposition arborescente.*

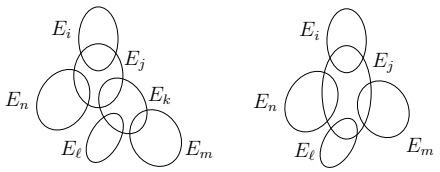


FIGURE 2 – Ensemble de clusters d'une décomposition avant fusion de E_k dans E_j (a) et après (b).

Il en est de même que pour l'analyse de sa complexité.

Théorème 2 *La complexité en temps de l'algorithme H_5-TD-WT est $O(n(n+e))$.*

4 Décomposition dynamique

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. Pour modifier la décomposition, on s'appuie sur des informations recueillies pendant la résolution, notamment celles liées à la sémantique du problème. Cette démarche s'inscrit alors dans la lignée des méthodes dites *adaptatives*. Ces méthodes font des choix non seulement par rapport à l'état courant de la résolution, mais aussi par rapport aux états précédents. Ce faisant, elles ont démontré leur intérêt pratique (comme dans [4, 18, 20]) par rapport aux méthodes classiques. Par exemple, en utilisant des stratégies de choix de variables orientées vers les conflits, on peut identifier les variables les plus problématiques pendant la recherche en enregistrant en permanence des informations sur les conflits rencontrés. Ainsi, cela permet de considérer ces variables plus tôt dans le reste de l'arbre de recherche et, par là même, de résoudre l'instance plus efficacement. Si l'ordre sur les variables est totalement libre, la variable suivante serait donc la variable jugée comme la plus influente pour la suite de la résolution. Par contre, pour des méthodes exploitant une décomposition arborescente comme BTD, l'ordre sur les variables étant partiellement induit par la décomposition, la liberté de l'heuristique de choix de variables s'en trouve d'autant restreinte. Pour y remédier, nous proposons d'adapter la décomposition durant la résolution en procédant à une fusion dynamique de certains clusters.

L'algorithme BTD-MAC+RST+Fusion (voir l'algorithme 3) représente une adaptation de l'algorithme BTD-MAC+RST [13] afin de prendre en compte la fusion dynamique. Pour ces deux algorithmes, l'emploi d'une décomposition arborescente avec un certain cluster racine E_r induit un ordre partiel sur les variables. Si E_j est le cluster courant, le choix de variables se limite alors soit à choisir parmi les variables

Algorithme 2 : BTD-MAC+Fusion

```
   Entrées-Sorties : P = (X, D, C) : CSP
   Entrées : Σ : suite de décisions ; E_i : cluster ; V_{E_i} : ensemble de
             variables
   Entrées-Sorties : G : ensemble de goods ; N : 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_{E_i} = ∅ alors
 2     résultat ← vrai
 3     S ← Fils(E_i) /* Fils(E_i) : ensemble des clusters fils de E_i */
 4     tant que résultat ∉ {faux, inconnu} et S ≠ ∅ faire
 5         Choisir un cluster E_j ∈ S
 6         S ← S\{E_j}
 7         si Pos(Σ)[E_i ∩ E_j] est un nogood dans N 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 alors
11                 résultat ← BTD-MAC+Fusion(P,Σ,E_j,
                     E_j\(E_i ∩ E_j),G,N)
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
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
17                     sinon
18                         si fusion alors Fusionner E_j avec un de
                             ses fils
19                         si non redémarrage alors
20                             S ← S ∪ {E_j}
21                             résultat ← vrai
22     retourner résultat
23 sinon
24     Choisir une variable x ∈ V_{E_i}
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_{E_i}\{x},G,N)
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 ∪ ⟨x ≠ v⟩)[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_{E_i},G,N)
37             sinon retourner faux
38     sinon retourner résultat
```

non encore instanciées du cluster E_j, soit à choisir un prochain cluster parmi les fils de E_j une fois le cluster E_j entièrement instancié. Les deux algorithmes débutent la résolution avec une décomposition calculée en amont de celle-ci. La différence principale entre eux réside dans le fait que BTD-MAC+RST n'utilise que 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-MAC+RST+Fusion va la faire évoluer dynamiquement. Ainsi, l'ordre partiel imposé par la décomposition change pendant la résolution. L'opération per-

Algorithme 3 : BTD-MAC+RST+Fusion

```
   Entrées : P = (X, D, C) : CSP
   Sorties : vrai si P possède une solution, faux sinon
 1 G ← ∅ ; N ← ∅
 2 répéter
 3     Choisir un cluster racine E_r
 4     résultat ← BTD-MAC+Fusion (P,∅,E_r,E_r,G,N)
 5 jusqu'à résultat≠ unknown
 6 retourner résultat
```

mettant de changer la décomposition dans ce contexte est la *fusion*. La fusion consiste à mettre en commun les variables de deux clusters voisins pour former ainsi un seul cluster. La figure 2 montre la fusion des clusters E_j et E_k. À noter que, les fils du cluster fusionné deviennent les fils du cluster résultant de la fusion. Par exemple, dans la figure 2, E_l et E_m, les fils de E_k, deviennent les fils de E_j. Soit D la décomposition initiale et D' la décomposition obtenue après la fusion. Tout ordre sur les variables permis par D reste permis par D'. Cependant, en exploitant D' on peut obtenir plus d'ordres possibles qu'en exploitant D. On en déduit que la fusion préserve les ordres de choix de variables initialement permis tout en offrant plus de liberté. Le choix de fusionner ou non des clusters est conditionné par des informations apprises durant la résolution. Aussi, le comportement de BTD-MAC+RST+Fusion se situe entre celui de BTD-MAC+RST avec une liberté partielle pour l'ordre sur les variables (si aucune fusion n'est effectuée) à celui de MAC [23] 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 donc de pouvoir trouver dynamiquement le bon compromis à partir d'informations recueillies durant la résolution.

L'algorithme BTD-MAC+RST+Fusion exploite l'algorithme BTD-MAC+Fusion (voir l'algorithme 2). Ce dernier ne se différencie de BTD-MAC+NG [13] que par les lignes 17 à 21. Initialement, la suite de décisions Σ ainsi que les ensembles de goods G et de nogoods N sont vides. BTD-MAC+RST+Fusion (à l'instar de BTD-MAC+RST) commence la résolution en assignant les variables du cluster racine 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 $E_i \cap E_{p(i)}$ (avec $E_{p(i)}$ le cluster père de E_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 : des décisions positives $x_i = v_i$ qui assignent la valeur v_i à la variable x_i et des décisions négatives $x_i \neq v_i$ qui assurent que v_i ne peut être assignée à x_i. Supposons que $\Sigma =< \delta_1, ..., \delta_i >$ 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, on procède à un retour-arrière et la dernière décision positive $x_\ell = v_\ell$ est changée en $x_\ell \neq v_\ell$. Lorsque le cluster E_i est choisi comme cluster suivant, on sait 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 de $Desc(E_i)$ (avec $Desc(E_i)$ l'ensemble de variables appartenant à l'union des descendants E_k de 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 décision Σ, on résout le problème enraciné en E_j et induit par $Pos(\Sigma)[E_i \cap E_j]$ (avec $Pos(\Sigma)[E_i \cap E_j]$ l'ensemble des décisions positives impliquant les variables de $E_i \cap E_j$ dans Σ). Si on trouve 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 *nogood structurel* (lignes 15-16). Ces (no)good structurels sont utilisés ultérieurement dans la recherche pour éviter certaines redondances (lignes 7-8). Si un redémarrage est déclenché (ligne 31), la résolution est interrompue. La gestion des redémarrages est faite comme dans [13] et s'accompagne de l'enregistrement de nld-nogoods réduits [16] qui permettent de ne pas réexplorer des parties de l'espace de recherche déjà visitées. L'intérêt du redémarrage réside dans l'exploitation des connaissances acquises auparavant via les (no)good structurels et les nld-nogoods réduits [16]. Le déclenchement d'un redémarrage peut être conditionné par des paramètres globaux (portant sur l'ensemble du problème) ou locaux (relatifs au cluster courant) ou aussi une combinaison des deux. Les lignes 17-21 concernent uniquement la fusion dynamique donc BTD-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 section 2. Le choix de fusionner ou non des clusters va se faire sur la base d'un critère s'appuyant sur l'état courant du problème, mais aussi sur tout ou partie des états précédemment rencontrés durant la résolution. 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-MAC+Fusion va modifier la décomposition courante en fusionnant ces deux clusters (ligne 18). Pour cela, on commence par désaffecter les variables instanciées du cluster courant et par enregistrer des nld-nogoods réduits (ligne 32) comme le ferait un redémarrage classique. Une fois revenu dans le cluster parent, on procède

à la fusion. À ce stade (ligne 19), soit on continue à revenir en arrière si *redémarrage* est vrai, soit la recherche se poursuit avec l'exploration d'un fils du cluster parent. 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 de pouvoir exploiter au plus tôt les variables nouvellement ajoutées dans ce cluster.

L'algorithme BTD-MAC-Fusion est paramétrable notamment par l'heuristique de fusion (nous en proposons une dans la partie expérimentale). Un bon choix pour cette heuristique peut améliorer considérablement la résolution en la rendant plus efficace.

Fusionner deux clusters ne remettant pas en cause la validité des (no)goods structurels et des nld-nogoods, nous pouvons prouver la validité de l'algorithme avant de donner ses complexités en temps et en espace.

Proposition 1 *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 dans le cluster E_x (avec E_y fils de E_x dans (E, T)). Les (no)good structurels de E_i par rapport à 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').*

Théorème 3 *BTD-MAC+RST+Fusion est correct, complet et termine.*

Théorème 4 *BTD-MAC+RST+Fusion a une complexité en temps en $O(R.((n.s^2.e.\log(d) + w'^{+}.N).d^{w'^{+}+2} + n.(w'^{+})^2.d))$ et une complexité en espace en $O(n.s.d^s + w'^{+}.(d+N))$ 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 le nombre de nld-nogoods réduits mémorisés.*

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 convenable. Dans un tel cas de figure, BTD-MAC+RST+Fusion est proche de l'algorithme BDH [11]. Outre l'absence de redémarrages dans BDH, en pratique, il s'en 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 limite fixée au préalable.

5 Évaluation expérimentale

Dans cette section, nous évaluons l'intérêt pratique de l'heuristique H_5 et des décompositions dynamiques.

5.1 Protocole expérimental

En ce qui concerne les décompositions, nous avons retenu *Min-Fill* (en tant qu'heuristique de référence dans la littérature et pour sa bonne approximation de la largeur arborescente), H_2 (pour sa garantie de connexité des clusters), H_3 (dont les clusters ont plusieurs fils) et H_5 (pour sa maîtrise de la taille des séparateurs). Pour H_2, les clusters sont construits en choisissant les sommets de la composante connexe concernée par ordre de degré décroissant jusqu'à ce que le cluster devienne connexe (ce qui correspond à l'heuristique $NV2$ de [14]). Pour H_5, la décomposition est utilisée avec différentes valeurs de S.

La décomposition dynamique repose sur une heuristique de fusion. Cette dernière s'appuie sur l'heuristique de choix de variable pour évaluer la nécessité de fusionner. Plus précisément, étant donné un cluster courant E_j, à chaque fois qu'on choisit la variable suivante dans E_j, on va regarder si cette variable aurait été choisie si l'heuristique de choix de variable avait eu la possibilité de sélectionner une variable parmi toutes les variables non instanciées de $E_j \cup (\bigcup_{E_k \in Fils(E_j)} E_k)$. À chaque fois qu'une variable d'un fils E_k de E_j est préférée à une variable de E_j, un compteur propre à E_k est incrémenté. Lorsque le compteur associé à un fils E_k atteint une certaine limite L (à savoir 100 dans nos expérimentations), on fusionne E_k avec E_j.

Au niveau de la résolution, nous considérons BTD-MAC+RST+Fusion et BTD-MAC+Fusion (c.-à-d. BTD-MAC+RST+Fusion sans redémarrage) pour les méthodes exploitant des décompositions dynamiques, BTD-MAC et BTD-MAC+RST en tant que références pour les méthodes exploitant une décomposition statique, et MAC+RST comme méthode énumérative classique de référence. Nous choisissons comme cluster racine le cluster ayant le plus grand rapport nombre de contraintes sur la taille du cluster moins un. La cohérence d'arc est appliquée en pré-traitement via $AC3^{rm}$ et durant la résolution via $AC8^{rm}$ [15]. Toutes les méthodes de résolution utilisent l'heuristique dom/wdeg [4] pour choisir la prochaine variable à instancier.

Tous les algorithmes sont implémentés en C++ au sein de notre propre bibliothèque. Les expérimentations ont été réalisées sur des serveurs lames sous Linux Ubuntu 14.04 dotés chacun de deux processeurs Intel Xeon E5-2609 à 2,4 GHz et de 32 Go de mémoire. Nous avons sélectionné 1 859 les instances CSP de la compétition CSP 2008 [3]. 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 (car non prises en compte dans notre bibliothèque). Les instances retenues représentent la majo-

3. Voir http ://www.cril.univ-artois.fr/CPAI08.

Algorithme	*Min-Fill*		H_2		H_3		H_5	
	#rés.	temps	#rés.	temps	#rés.	temps	#rés.	temps
BTD-MAC	1 344	43 272	1 405	31 429	1 466	31 469	1 469	33 564
BTD-MAC+RST	1 495	43 557	1 518	35 042	1 529	30 187	1 543	33 049
BTD-MAC+Fusion	1 481	42 505	1 518	37 440	1 523	35 101	1 534	34 048
BTD-MAC+RST+Fusion	1 544	41 622	1 547	32 547	1 554	33 736	1 567	34 432

TABLE 1 – Nombre d'instances résolues et temps d'exécution en s pour BTD-MAC(+RST) et BTD-MAC(+RST)+Fusion selon la méthode de décomposition utilisée.

Algorithme	*Min-Fill*	H_2	H_3	H_5
BTD-MAC	34 669	18 018	18 951	18 243
BTD-MAC+RST	24 026	17 233	17 758	16 288
BTD-MAC+Fusion	25 238	17 575	18 753	17 837
BTD-MAC+RST+Fusion	23 832	16 803	17 602	15 718

TABLE 2 – Temps d'exécution en s pour BTD-MAC(+RST) et BTD-MAC(+RST)+Fusion selon la méthode de décomposition utilisée pour les 1 234 instances résolues par tous les algorithmes.

rité des familles d'instances. Pour chaque instance, le temps d'exécution (incluant le calcul de la décomposition) est limité à 15 minutes.

5.2 H_5 comparée aux autres décompositions

La table 1 fournit le nombre d'instances résolues et le temps d'exécution cumulé pour chaque algorithme et pour chaque méthode de décomposition. D'abord, nous comparons les méthodes de décomposition du point de vue de l'efficacité de la résolution par BTD-MAC. Concernant le nombre d'instances résolues, BTD-MAC avec H_5 (pour $S = 50$) résout le plus grand nombre d'instances (à savoir 1 469) tandis qu'avec *Min-Fill*, il en résout le moins (à savoir 1 344). Clairement, les méthodes de décomposition visant à minimiser la largeur ne conduisent pas aux résolutions les plus efficaces. D'autres paramètres ont plus d'impacts comme la connexité des clusters (cf. H_2), le nombre de fils des clusters (cf. H_3) et la maîtrise de la taille des séparateurs (cf. H_5). Notons que ces résultats sont cohérents avec ceux de [14, 9]. Au-delà, pour H_5, l'efficacité de la résolution dépend bien sûr de la valeur choisie pour S. Par exemple, BTD-MAC résout plus d'instances pour $S = 15$ (à savoir 1 514). Par contre, le choix de $S = 50$ est plus intéressant quand on exploite des décompositions dynamiques.

Au niveau du temps d'exécution, pour une comparaison plus équitable, nous ne considérons, dans la table 2, que les 1 234 instances résolues par tous les algorithmes. BTD-MAC obtient les meilleurs résultats avec H_5 (et H_2) et les pires avec *Min-Fill*. Nous devons souligner que le calcul des décompositions avec H_i ($i = 2, 3, 5$) est nettement plus rapide qu'avec *Min-Fill*. Par exemple, H_5 (pour $S = 50$) requiert seulement 7 s pour calculer les décompositions des 1 234 instances contre 7 582 s pour *Min-Fill*.

Notons enfin que les observations faites ici pour BTD-MAC restent valides quelle que soit la variante de BTD considérée comme le montrent les tables 1-4.

5.3 Décompositions dynamiques contre statiques

D'abord, si nous comparons BTD-MAC à BTD-MAC+Fusion (resp. BTD-MAC+RST à BTD-MAC+RST+Fusion) dans les tables 1-2, nous pouvons voir que, quelle que soit la méthode de décomposition utilisée, les méthodes exploitant des décompositions dynamiques résolvent plus d'instances que leur pendant exploitant une décomposition statique pour des temps de résolution similaires ou meilleurs. Cela montre bien l'intérêt de fusionner dynamiquement des clusters durant la résolution.

Par ailleurs, le concept de fusion a déjà été exploité de manière statique (par exemple dans [10]) une fois la décomposition initiale calculée via n'importe quelle méthode (comme *Min-Fill*, H_2, H_3 ou H_5). Les tables 3-4 fournissent les résultats correspondants pour BTD-MAC(+RST). Ici, nous limitons la taille des séparateurs en fusionnant avec son père tout cluster dont la taille du séparateur avec son père dépasse une certaine valeur (à savoir 15 dans les tables 3-4). Nous pouvons constater que, pour le nombre d'instances résolues, BTD-MAC+Fusion est significativement meilleur que BTD-MAC tandis que BTD-MAC+RST+Fusion est comparable ou légèrement meilleur que BTD-MAC+RST. À nouveau, l'exploitation des décompositions dynamiques permet d'obtenir parmi les meilleurs résultats. Au-delà, via la fusion dynamique, nous adaptons la décomposition en fonction des connaissances sémantiques apprises durant la résolution alors que la fusion statique ne repose que sur des critères structurels et nécessite de choisir une limite pour la taille des séparateurs, ce qui peut constituer une tâche difficile.

Enfin, nous pouvons remarquer que BTD-MAC+RST et BTD-MAC+Fusion sont relativement proches en termes de nombre d'instances résolues ou de temps d'exécution. Cela peut s'expliquer par le choix d'un nouveau cluster racine quand BTD-MAC+RST redémarre, ce qui peut être vu comme une forme simplifié de dynamicité. Par ailleurs, l'exploitation conjointe des redémarrages

et des décompositions dynamiques s'avère très pertinente puisque BTD-MAC+RST+Fusion surclasse à la fois BTD-MAC+RST et BTD-MAC+Fusion. Pour terminer, nous pouvons aussi noter que BTD-MAC+RST+Fusion avec H_5 obtient les meilleurs résultats quels que soient les algorithmes de résolution et/ou de décomposition exploités.

5.4 BTD-MAC+RST+Fusion versus MAC+RST

Nous comparons à présent MAC+RST et BTD-MAC+RST+Fusion (pour $S = 50$). La figure 3(a) représente le nombre d'instances résolues pour MAC-BTD+RST+Fusion, MAC+RST et VBS (c.-à-d. le meilleur solveur virtuel parmi les deux algorithmes). D'abord, BTD-MAC+RST+Fusion résout plus d'instances que MAC+RST (1 567 instances contre 1 548). Ensuite, nous pouvons noter que le comportement de BTD-MAC+RST+Fusion est plus proche de celui de VBS que celui de MAC+RST, ce qui montre clairement la supériorité de BTD-MAC+RST+Fusion sur MAC+RST.

Nous focalisons maintenant nos observations sur les instances les plus difficiles. Parmi les 1 859 instances considérées, certaines sont résolues facilement par MAC+RST (par exemple 284 instances le sont sans aucun retour en arrière). L'exploitation de méthodes structurelles comme BTD ou ses variantes sur de telles instances n'a pas nécessairement de sens. Aussi, nous utilisons le nombre de nœuds développés par MAC+RST comme critère de difficulté. Une instance sera considérée comme difficile si ce nombre de nœuds est supérieur à $100n$ (avec n le nombre de variables). Ainsi, nous avons 577 instances considerées comme difficile. La figure 3(b) présente une comparaison des temps d'exécution de MAC-BTD+RST+Fusion et MAC+RST pour ces instances. Globalement, nous pouvons constater que, pour une bonne partie d'entre elles, MAC-BTD+RST+Fusion et MAC+RST ont un comportement similaire. En effet, pour environ 60 % des instances, l'écart entre les temps d'exécution est inférieur à 10 %. Cependant, pour les instances restantes, MAC-BTD+RST+Fusion surclasse souvent MAC+RST. Pour 16 % d'entre elles, MAC-BTD+RST+Fusion est au moins 10 fois plus rapide que MAC+RST alors que MAC+RST ne l'est que dans 1 % des cas. Enfin, l'exploitation de la structure joue ici un rôle central. En effet, 86 % des instances non résolues par MAC+RST mais résolues par BTD-MAC+RST+Fusion sont des instances structurées ayant un rapport $n/(w+1)$ supérieur à 5.

6 Conclusion

Dans ce papier, nous avons proposé deux contributions complémentaires. D'une part, nous avons présenté un nouvel algorithme pour calculer des décompositions arborescentes (à savoir H_5) permettant de borner la taille des séparateurs, qui est un paramètre crucial pour l'efficacité pratique des méthodes de résolution structurelles comme BTD. Sa complexité en temps est meilleure que celle de *Min-Fill* et il est nettement plus rapide en pratique (environ 1 000 fois plus rapide que *Min-Fill* sur un vaste ensemble d'instances). D'autre part, nous avons décrit une extension non triviale de BTD, à savoir BTD-MAC+RST+Fusion, qui peut adapter la décomposition en fusionnant dynamiquement certains clusters en fonction de la sémantique de l'instance et de connaissances acquises durant la résolution. Ainsi, notre méthode exploite des ordres sur les variables plus flexibles et peut éviter certains inconvénients que peut avoir une décomposition intiale calculée uniquement sur la base de critères structurels. En pratique, nous avons montré que BTD-MAC+RST+Fusion surclasse BTD-MAC+RST quelle que soit la méthode de décomposition exploitée. De plus, son utilisation conjointe avec H_5 conduit à l'obtention des meilleurs résultats.

Plusieurs extensions de ce travail sont possibles. D'abord, d'autres heuristiques de fusion sont envisageables en exploitant d'autres informations sémantiques. Ensuite, la rapidité de H-TD-WT ouvre des perspectives plus larges pour une modification dynamique de la décomposition en permettant des recalculs pendant la résolution et lors des redémarrages. Au-delà, des problèmes plus difficiles (optimisation, comptage ou compilation) pourraient être abordés.

Références

[1] D. Allouche, S. de Givry, and T. Schiex. Towards Parallel Non Serial Dynamic Programming for Solving Hard Weighted CSP. In *CP*, pages 53–60, 2010.

[2] S. Arnborg, D. Corneil, and A. Proskuroswki. Complexity of finding embeddings in a k-tree. *SIAM Journal of Disc. Math.*, 8 :277–284, 1987.

[3] C. Berge. *Graphs and Hypergraphs.* Elsevier, 1973.

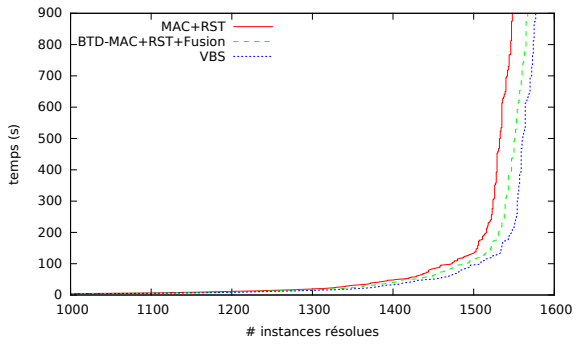
[4] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *ECAI*, pages 146–150, 2004.

[5] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio Link Frequency Assignment. *Constraints*, 4 :79–89, 1999.

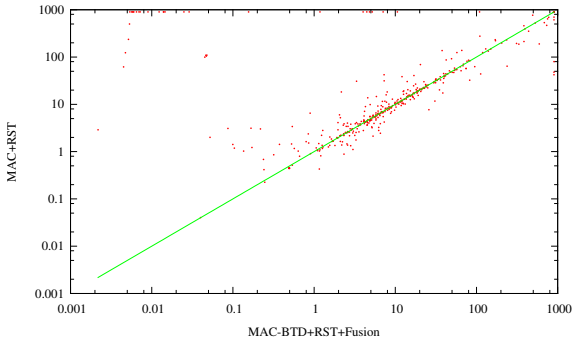
[6] S. de Givry, T. Schiex, and G. Verfaillie. Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP. In *AAAI*, pages 22–27, 2006.

Algorithme	*Min-Fill*		H_2		H_3		H_5	
	#rés.	temps	#rés.	temps	#rés.	temps	#rés.	temps
`BTD-MAC`	1 450	45 988	1 493	35 871	1 504	31 612	1 511	32 097
`BTD-MAC+RST`	1 537	41 722	1 549	33 328	1 553	33 164	1 564	33 145

TABLE 3 – Nombre d'instances résolues et temps d'exécution pour BTD-MAC(+RST) selon la méthode de décomposition utilisée pour une fusion statique limitant la taille des séparateurs à 15.



(a)



(b)

FIGURE 3 – (a) Nombre d'instances résolues pour MAC-BTD+RST+Fusion avec H_5, MAC+RST et VBS, (b) comparaison des temps d'exécution de MAC-BTD+RST+Fusion et MAC+RST pour les 577 instances difficiles.

Algorithme	*Min-Fill*	H_2	H_3	H_5
`BTD-MAC`	32 641	17 914	17 813	16 503
`BTD-MAC+RST`	24 456	17 514	17 050	16 235

TABLE 4 – Temps d'exécution pour BTD-MAC(+RST) selon la méthode de décomposition utilisée pour une fusion statique limitant la taille des séparateurs à 15, pour les 1 234 instances résolues par tous les algorithmes.

[7] R. Dechter. *Constraint processing*. Morgan Kaufmann Publishers, 2003.

[8] G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124 :243–282, 2000.

[9] P. Jégou, H. Kanso, and C. Terrioux. An Algorithmic Framework for Decomposing Constraint Networks. In *ICTAI*, pages 1–8, 2015.

[10] P. Jégou, S. N. Ndiaye, and C. Terrioux. Computing and exploiting tree-decompositions for solving constraint networks. In *CP*, pages 777–781, 2005.

[11] P. Jégou, S.N. Ndiaye, and C. Terrioux. Dynamic Management of Heuristics for Solving Structured CSPs. In *CP*, pages 364–378, 2007.

[12] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146 :43–75, 2003.

[13] P. Jégou and C. Terrioux. Combining Restarts, Nogoods and Decompositions for Solving CSPs. In *ECAI*, pages 465–470, 2014.

[14] P. Jégou and C. Terrioux. Tree-decompositions with connected clusters for solving constraint networks. In *CP*, pages 407–423, 2014.

[15] C. Lecoutre, C. Likitvivatanavong, S. Shannon, R. Yap, and Y. Zhang. Maintaining Arc Consistency with Multiple Residues. *Constraint Programming Letters*, 2 :3–19, 2008.

[16] C. Lecoutre, L. Saïs, S. Tabary, and V. Vidal. Recording and Minimizing Nogoods from Restarts. *JSAT*, 1(3-4) :147–167, 2007.

[17] W. Li and P. van Beek. Guiding Real-World SAT Solving with Dynamic Hypergraph Separator Decomposition. In *ICTAI*, pages 542–548, 2004.

[18] L. Michel and P. Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *CP-AI-OR*, pages 228–243, 2012.

[19] N. Eén and N. Sörensson. An Extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[20] P. Refalo. Impact-based search strategies for constraint programming. In *CP*, pages 557–571, 2004.

[21] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, 7 :309–322, 1986.

[22] D. J. Rose. A graph theoretic study of the numerical solution of sparse positive definite systems of linear equations. In *Graph Theory and Computing*, pages 183–217. Academic Press, 1972.

[23] D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *ECAI*, pages 125–129, 1994.