Un cadre générique pour l'intégration de BTD dans une bibliothèque de programmation par contraintes

Samba Ndojh NDIAYE[1,2] **Christine SOLNON**[1,3]

[1] Université de Lyon, LIRIS, UMR 5205 CNRS
[2] Université Lyon 1, F-69622 France
[3] INSA-Lyon, F-69621, France

{samba-ndojh.ndiaye,christine.solnon}@liris.cnrs.fr

Résumé

La méthode BTD permet de résoudre efficacement des instances CSP en tirant profit de la structure des problèmes à travers une décomposition arborescente du réseau de contraintes. Cette décomposition permet d'identifier des sous-problèmes indépendants dont la résolution peut être effectuée par toute méthode de résolution de CSP classique. Dans cet article, nous introduisons un cadre générique pour intégrer BTD dans une bibliothèque de programmation par contraintes, ce qui nous permet de bénéficier des différents algorithmes de propagation et de recherche implémentés dans la bibliothèque. Ce cadre a été implémenté à l'aide de la bibliothèque Gecode, et nous considérons deux variantes : la première (BTD-G) utilise la recherche Gecode par défaut pour résoudre chaque sous-problème, tandis que la seconde (BTD-GR) utilise une recherche avec *restarts*. Cela nous permet d'évaluer l'intéret d'utiliser une recherche avec *restarts* lors de la résolution des sous-problèmes. Ces deux variantes sont comparées expérimentalement sur les instances de la compétition CSP'2009. Nous constatons que BTD-GR surpasse BTD-G pour beaucoup d'instances, et que BTD-GR surpasse Gecode (dans sa version avec *restarts*) pour un grand nombre d'instances structurées.

Abstract

BTD exploits tree decompositions of constraint networks to speed up their solution process. The decomposition is used to identify independent subproblems which may be solved by any constraint solver provided that it is complete. Since now, BTD has been implemented on top of *ad hoc* solvers which are often restricted to constraints defined in extension. In this paper, we introduce a framework for integrating BTD in a Constraint Programming library, thus allowing us to benefit from various kinds of constraints, propagators, and heuristics to solve subproblems identified by BTD. This framework has been implemented on top of Gecode, and we consider two variants of it : The first one (BTD-G) uses the default Gecode search without restarts, whereas the second one (BTD-GR) uses the restart-based Gecode search. This allows us to evaluate the interest of using restarts when assigning values to the variables of a same cluster. These two variants are experimentally evaluated on the CSP'2009 competition benchmark, showing that BTD-GR outperforms BTD-G on many instances, and that BTD-GR outperforms Gecode (with a restart policy) on many structured instances.

1 Introduction

Le formalisme CSP (problème de satisfaction de contraintes) permet de modéliser très simplement un problème en identifiant un ensemble de variables à affecter avec un ensemble de valeurs en essayant de respecter des contraintes. Les méthodes de résolution structurelles offrent les meilleures garanties théoriques pour la recherche d'une solution d'une instance CSP [6]. Cependant, leur efficacité pratique n'est pas toujours évidente à démontrer. La méthode BTD [11], tout en garantissant parmi les meilleures bornes de complexité théorique [10], a donné des résultats très prometteurs sur des instances binaires souvent générées de manière aléatoire [11, 9, 10]. Des évaluations ont également été menées avec succès sur des instances structurées de la compétition CSP-2008

[2, 13, 12, 15]. Malheureusement, ces dernières expérimentations posent des restrictions sur l'arité des contraintes, sur la définition des relations en extension ou en intention, voire sur la gestion des contraintes globales. Par ailleurs, le fait que BTD puisse reposer sur toute méthode capable de résoudre les sous-problèmes identifiés par la décomposition arborescente a conduit à plusieurs combinaisons ([11, 3, 2, 13]) avec des méthodes telles que FC [7], MAC [22], CBJ [20], Decision Repair [14, 19], mais aussi avec plusieurs heuristiques de choix de variables telles que dom/ddeg [1] et dom/wdeg [4]. [13] a proposé une intégration des politiques de restarts [5] dans BTD. Malgré tous ces travaux, il reste un grand nombre de possibilités qui n'ont pas encore été explorées et beaucoup de types d'instances sur lesquelles cette méthode n'a pas été évaluée. Il semble difficile d'accomplir cette tâche en implémentant directement tous les outils nécessaires du fait du nombre impressionnant de techniques proposées chaque année par la communauté programmation par contraintes. Néanmoins, il faut noter que cette communauté dispose de plusieurs bibliothèques regroupant une grande partie des techniques les plus performantes de résolution de CSP. Nous proposons de nous appuyer sur toutes ces ressources offertes par ces bibliothèques afin de faciliter l'intégration dans BTD des meilleures techniques de résolution de CSP et l'évaluation de ces combinaisons sur un ensemble diversifié d'instances contenues dans les benchmarks disponibles. Pour cela, nous définissons un cadre générique d'intégration de BTD dans une bibliothèque de programmation par contraintes (CP). Ce cadre suppose uniquement l'existence dans la bibliothèque d'une méthode capable d'énumérer l'ensemble des solutions d'une instance CSP. Nous montrons également comment étendre ce cadre à des approches de résolution utilisant des politiques de *restarts*. Ces deux variantes ont été implémentées dans la bibliothèque Gecode [23] et des expérimentations ont été menées sur l'ensemble des instances CSP du benchmark de la compétition CSP'2009. Les résultats démontrent l'efficacité de BTD combinée avec une politique de restarts limitée aux clusters de la décomposition arborescente.

La section 2 fait quelques rappels sur les notions de base liées à la méthode BTD. Dans la section 3, nous présentons le cadre générique, puis nous discutons de l'intégration de politiques de restarts dans la section 4. La section 5 décrit les résultats expérimentaux.

2 Préliminaires

Un *problème de satisfaction de contraintes* (X, D, C) est défini par un ensemble de variables X, une fonction domaine D qui associe à chaque variable $x_i \in X$ l'ensemble $D(x_i)$ des valeurs pouvant être affectées à x_i, et un ensemble de contraintes C à satisfaire. Une contrainte $c_j \in C$ est définie sur un sous-ensemble des variables $\mathit{portée}(c_j) \subseteq X$ et restreint les valeurs que ces variables peuvent prendre simultanément. Etant données deux fonctions domaine D et D', nous notons $D' \subseteq D$ si $D'(x_i) \subseteq D(x_i)$ pour chaque variable $x_i \in X$. Nous disons qu'une fonction domaine D affecte une variable x_i si $|D(x_i)| = 1$. Une solution est une fonction domaine qui affecte toutes les variables et satisfait toutes les contraintes.

La structure d'un CSP (X, D, C) peut être représentée par l'hypergraphe $H_{(X,D,C)} = (X, \mathcal{E})$, appelé *hypergraphe de contraintes*, tel que $\mathcal{E} = \{\mathit{portée}(c) | c \in C\}$. Chaque sommet de cet hypergraphe est associé à une variable du CSP, et chaque hyperarête à une contrainte. La 2-section de $H_{(X,D,C)}$ est le graphe $G_{(X,D,C)} = (X, E)$ où $E = \{\{x_i, x_j\} \subseteq X : \exists c_k \in C, \{x_i, x_j\} \subseteq \mathit{portée}(c_k)\}$. Ainsi, toute contrainte $c_k \in C$ induit une clique dans la 2-section qui contient les sommets associés à $\mathit{portée}(c_k)$. Une décomposition arborescente d'un CSP est définie sur la 2-section de son hypergraphe de contraintes.

Définition 1 *[21] Une décomposition arborescente d'un graphe $G = (X, E)$ est un couple (K, T) où $T = (I, F)$ est un arbre, $K : I \to \mathcal{P}(X)$ est une fonction qui associe un sous-ensemble de variables $K_i \subseteq X$ (appelé* cluster*) à chaque sommet, et les conditions suivantes sont vérifiées :*
(i) $\cup_{i \in I} K_i = X$;
(ii) pour chaque arête $\{x_j, x_k\} \in E$, il existe un sommet $i \in I$ tel que $\{x_j, x_k\} \subseteq K_i$;
(iii) pour tout $i, j, k \in I$, si k est sur le chemin entre i et j dans T, alors $K_i \cap K_j \subseteq K_k$.
La largeur w d'une décomposition arborescente (K, T) est égale à la taille maximum des clusters moins 1, i.e.*, $w = \max_{i \in I} |K_i| - 1$. La largeur d'arbre (ou* treewidth*) w^* de G est la largeur minimum sur l'ensemble de ses décompositions arborescentes.*

Etant donnée une décomposition arborescente (K, T) d'un CSP (X, D, C) et une racine $r \in I$, BTD identifie des sous-problèmes indépendants qui sont résolus séparément. Plus précisément, chaque sous-problème ne contient qu'un sous-ensemble de l'ensemble initial de variables. BTD affecte en premier les variables du cluster racine K_r. S'il n'existe pas d'affectation cohérente de ces variables, alors le problème est incohérent. Sinon, si r a k fils $i_1, \ldots, i_k$ dans l'arbre T, BTD résoud récursivement k sous-problèmes indépendants : chaque sous-problème est associé à un fils i_j de r et contient l'ensemble des variables apparaissant dans les clusters associés aux sommets du sous-arbre de T enraciné en i_j. Ces k sous-problèmes sont indé-

Algorithme 1 – BTD$((X, D, C), (K, T), r)$

Données : une instance CSP (X, D, C), une décomposition arborescente (K, T), et un sommet racine r de T
Résultats : Retourne *succès* s'il existe une fonction domaine $D' \subseteq D$ qui affecte de façon cohérente toutes les variables des clusters associés aux sommets de T ; retourne *échec* sinon.

```
1  model ← buildModel((X, D, C), K_r)
2  tant que model.next() ≠ null faire
3      Soit newD la fonction domaine retournée par next() (telle que newD affecte de façon cohérente toutes les variables de K_r)
4      Pour chaque fils i de r, soit A_i le tuple de valeurs affectées aux variables de K_r ∩ K_i dans newD
5      tant que il n'existe pas de fils j de r tel que A_j est un nogood
6      et il existe un fils i de r tel que A_i n'est pas un good faire
7          Soit T_i le sous-arbre de T enraciné en i
8          si BTD((X, newD, C), (K, T_i), i) = succès alors
9              enregistrer A_i comme good
10         sinon
11             enregistrer A_i comme nogood
12     si pour chaque fils i de r, A_i est un good alors
13         retourne succès
14 retourne échec
```

pendants car, dès lors que les variables du cluster racine K_r ont été affectées, il n'existe plus de contrainte partagée entre deux sous-problèmes différents.

Afin d'éviter d'exlorer plusieurs fois une même zone de l'espace de recherche, BTD enregistre des *(no)goods* structurels. Un good (resp. nogood) structurel est une affectation des variables d'un *séparateur* (*i.e.*, les variables qui appartiennent à l'intersection d'un cluster racine avec un de ses fils) tel que le sous-problème associé à ce fils est cohérent (resp. incohérent) quand on affecte ainsi ces variables. Ces (no)goods structurels permettent de réduire la complexité théorique de BTD à $\mathcal{O}(nd^{w+1})$ où n est le nombre de variables, d la taille du plus grand domaine, et w la largeur de T. La complexité en espace est $\mathcal{O}(nsd^s)$ où s est la taille du plus grand séparateur. Le nombre maximum de (no)goods structurels que BTD peut enregistrer sur un séparateur donné est borné par d^s.

3 Cadre générique d'une intégration de BTD dans une bibliothèque CP

Comme cela a été signalé dès son introduction [11], BTD peut utiliser n'importe quelle approche de résolution de CSP pour rechercher une solution dans un sous-problème défini par un cluster. Nous proposons ici un cadre générique pour l'intégration de BTD dans une bibliothèque CP, permettant ainsi de combiner BTD avec l'ensemble des techniques de résolution, algorithmes de propagation (qui n'altèrent pas la structure du problème en rajoutant des contraintes portant sur des variables qui ne se trouvent pas au sein d'un même cluster) et heuristiques intégrés dans la bibliothèque.

Notre cadre suppose uniquement que la bibliothèque CP fournit deux méthodes :

- une méthode *buildModel*$((X, D, C), K_i)$ qui retourne un modèle CP étant donnés une instance CSP (X, D, C) et un sous-ensemble de variables à instancier $K_i \subseteq X$;
- une méthode *next()* qui retourne une nouvelle fonction domaine $D' \subseteq D$ qui affecte de façon cohérente toutes les variables de K_i, ou bien retourne *null* s'il n'existe plus de solution.

La fonction générique qui intègre BTD dans une telle bibliothèque CP est décrite dans l'algorithme 1. Cette fonction prend en entrée une instance CSP (X, D, C), une décomposition arborescente (K, T), et un sommet racine r de T. *BTD* appelle tout d'abord la méthode *buildModel* afin de construire le modèle CP, où l'ensemble des variables à affecter est restreint à celles du cluster racine K_r. Ensuite, elle appelle itérativement la méthode *next* (ligne 2) pour rechercher des affectations cohérentes des variables de K_r, jusqu'à ce que soit une affectation puisse être étendue à une solution pour tous les sous-problèmes de r (de sorte que le problème est résolu - ligne 12), ou bien il n'existe plus d'autre affectation cohérente (de sorte que le problème est incohérent - ligne 13). Dans les lignes 3 à 10, *BTD* tente d'étendre l'affectation courante des variables de K_r à une solution pour tous les sous-problèmes : pour chaque fils i de r dans T, s'il existe un (no)good pour les valeurs affectées aux variables du séparateur $(K_r \cap K_i)$, alors ce sous-problème a déjà été résolu précédemment ; sinon *BTD* est récursivement appelée sur le sous-arbre de T enraciné en i, et le (no)good correspondant à ce sous-problème est enregistré. La boucle lignes 5 à 10 se termine soit quand un nogood est trouvé pour un fils (dans ce cas,

la recherche passe à l'affectation suivante des variables de K_r), soit quand il existe un good pour tous les fils (dans ce cas, la recherche s'arrête sur un succès, ligne 12).

Etant donné un modèle retourné par la méthode *buildModel*$((X, D, C), K_r)$, si la complexité en temps de l'ensemble des appels à la méthode *next()* (pour rechercher toutes les affectations cohérentes des variables de K_r) est $\mathcal{O}(|K_r|^d)$ où $d = \max_{x_i \in K_r} |D(x_i)|$, alors la complexité en temps de l'algorithme 1 est la même que celle de l'algorithme BTD introduit dans [11].

4 Intégration de politiques de restarts dans le cadre générique

Les bibliothèques CP récentes intègrent des politiques de restarts qui améliorent fortement les performances des méthodes de résolution de CSP. Une recherche avec restarts est composée de plusieurs recherches successives, chacune de ces recherches étant limitée (borne sur le temps ou sur le nombre d'échecs, par exemple) : quand la limite est atteinte, la recherche courante est arrêtée et une nouvelle recherche est lancée avec une nouvelle limite (typiquement, une limite plus grande). Chaque nouvelle recherche considère des heuristiques différentes, par exemple, de sorte que des zones différentes de l'espace de recherche sont explorées. Par ailleurs, des informations peuvent être transmises d'une recherche à l'autre, concernant par exemple l'ordre dans lequel affecter les variables ou bien des nogoods liés aux échecs [17].

L'intégration de restarts dans BTD peut se faire de deux manières orthogonales qui peuvent être combinées. La première qui a été proposée dans [13] consiste à changer de cluster racine à chaque tour. Cela conduit évidemment à modifier l'ordre d'affectation des variables et donc à explorer différentes parties de l'espace de recherche.

La seconde que nous explorons ici consiste à utiliser une méthode de résolution à base de restarts pour la résolution des clusters (recherche d'une solution dans le cluster courant). Plus précisément, quand nous demandons à la bibliothèque CP de rechercher une affectation cohérente des variables du cluster racine K_r (appel de la méthode *next()*, ligne 2 de l'algorithme 1), nous paramétrons la bibliothèque CP afin qu'elle utilise une politique de restarts. Cependant, les approches à base de restarts ne permettent pas directement de lister toutes les solutions d'un problème. En effet, rechercher une nouvelle solution juste après avoir trouvé une première solution peut conduire exactement à la même solution. Il est nécessaire donc de rajouter une contrainte (un nogood) interdisant la solution trouvée pour pouvoir en trouver potentiellement d'autres. Nous proposons de combiner ces nogoods avec des nogoods structurels (*i.e.*, des nogoods sur les séparateurs des clusters) : quand une affectation cohérente pour les variables du cluster racine K_r a été trouvée, BTD essaie récursivement de l'étendre à tous les sous-arbres de r ; s'il réussit (ligne 11), alors BTD s'arrête sur un succès et il n'est pas nécesssaire d'ajouter un nogood dans la mesure où le problème enraciné en r est résolu ; sinon, nous ajoutons au modèle Gecode le nogood correspondant aux valeurs affectées aux variables de $K_r \cap K_i$ où i est le fils de r pour lequel BTD a retourné un échec. Concrêtement, cela est réalisé en modifiant la ligne 10 de l'algorithme 1 pour ajouter le nogood A_i à *model*, en plus d'enregistrer A_i comme un nogood structurel.

Notons que les nogoods ajoutés au modèle Gecode sont des nogoods visant à empêcher la méthode *next()* de Gecode de retourner une solution qu'elle a déjà retournée précédemment. L'objectif est donc différent des nogoods introduits par Lecoutre *et al* dans [17], qui visent à éviter d'explorer plusieurs fois des zones de l'espace de recherche menant à des échecs. De fait, lorsque Gecode utilise une politique de restarts, ces nogoods liés aux échecs sont automatiquement gérés par Gecode.

Théorème 1 *La complexité en temps de l'algorithme 1 utilisant une politique de restarts est* $\mathcal{O}(nb_{restarts}.n^2d^{w+1+s})$ *avec* $nb_{restarts}$ *le nombre de restarts, n le nombre de variables, d la taille maximum des domaines, w la largeur de la décomposition arborescente et s la taille maximum des intersections entre clusters.*

Si une solution est trouvée sur un cluster, soit elle mène à une solution globale dans le sous-problème enraciné en ce cluster et celui-ci est résolu, soit elle ne mène pas à une solution globale dans ce sous-problème et un nogood est rajouté et on cherche la solution suivante. Sachant que la recherche de la solution suivante ne commence pas nécessairement là où s'était arrêté la dernière, son coût est en $O(d^{w+1})$. Cette recherche de la solution suivante est effectuée autant de fois que la solution précédente a conduit à un échec et donc à l'enregistrement d'un nogood. Le nombre maximum de nogoods qu'il est possible d'avoir est borné par $maxf.d^s$, avec $maxf$ le nombre maximum de fils du cluster courant. En outre, $maxf$ peut être borné par n. De ce fait, la complexité de cette version de BTD est en $O(n^2d^{w+1+s})$ pour chaque restart. Par ailleurs, nous avons noté un nombre de restarts limité sur les expérimentations que nous avons menées. Il est en moyenne de 4 par instance avec un maximum de 460 pour une instance très difficile contenant 680 variables.

5 Évaluation expérimentale

5.1 Description du benchmark et des conditions expériementales

Nous avons choisi d'évaluer les méthodes proposées sur les 3285 instances CSP du benchmark de la compétition CP'2009 (Fourth International CSP Solver Competition : http ://www.cril.univ-artois.fr/CSC09/). Ces instances ont été regroupées en cinq classes :

- la classe C1, qui contient 556 instances définies avec des contraintes globales (*alldiff, cumulative, element, weightedsum*),
- la classe C2, qui contient 709 instances avec des contraintes d'arité quelconque définies en intention,
- la classe C3, qui contient 686 instances avec des contraintes binaires définies en intention,
- la classe C4, qui contient 699 instances avec des contraintes d'arité quelconque définies en extension,
- la classe C5, qui contient 635 instances avec des contraintes binaires définies en extension.

Nous avons utilisé le modèle proposé dans [18] pour définir ces instances au sein du solveur Gecode. Toutes les expérimentations ont été menées sur des processeurs Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz, avec 20480 Ko de cache et 3Go de RAM. Chaque méthode a été exécutée sur ces instances avec une durée limite fixée à 30mn au-delà de laquelle cette instance est considérée comme non résolue par la méthode. Les temps reportés dans la suite de cette section incluent le temps nécessaire au prétraitement de l'instance et au calcul de la décomposition arborescente (détaillés en 5.2). Si la mémoire nécessaire pour la résolution d'une instance dépasse les 3Go de RAM, alors cette instance est également considérée comme non résolue par la méthode.

5.2 Approches considérées et détails d'implémentation

Nous avons implémenté l'algorithme 1 à l'aide de la bibliothèque Gecode [23], de façon très naturelle dans la mesure où Gecode fournit les deux méthodes *buildModel* et *next*, et permet d'ajouter des nogoods à un modèle. Nous appelons *BTD-G* (resp. *BTD-GR*) la variante utilisant Gecode sans restarts (resp. Gecode avec restarts) quand la méthode *next* est appelée.

Paramètres de Gecode. Une première série d'expérimentations nous a permis de constater que la configuration de Gecode (sans BTD) qui obtient les meilleurs résultats sur les instances considérées est celle qui utilise une politique de restarts combinée à l'heuristique de choix de variables INT_VAR_AFC_SIZE_MAX (correspondant à minDom/wdeg [4] avec un choix aléatoire en cas d'égalités) et un choix de valeurs aléatoire. La politique de restarts est basée sur une progression géométrique avec un facteur d'échelle fixé à 2000 et une base à 2. Les restarts sont combinés avec l'enregistrement d'un nogood à chaque fois qu'une affectation partielle incohérente est trouvée, la profondeur maximale d'enregistrement de ces nogoods étant limitée à sa valeur par défaut (*i.e.*, 128). Cette configuration de Gecode est appelée GecodeR dans la suite.

Nous utilisons les mêmes heuristiques de choix de variables et de valeurs que GecodeR pour BTD-G et BTD-GR, et la même politique de restarts pour BTD-GR.

Calcul d'une décomposition arborescente pour BTD-G et BTD-GR. BTD repose sur une décomposition arborescente d'un CSP, et sa complexité en temps dépend de la taille des clusters. Plus les clusters sont petits, plus le problème est simple à résoudre en théorie. Cependant, le calcul d'une décomposition minimisant la taille maximum des clusters est un problème NP-difficile. Ainsi, une heuristique est souvent utilisée pour calculer une triangulation de la 2-section de l'hypergraphe de contraintes (de sorte que tout cycle de longueur supérieure ou égale à 4 possède une arête reliant deux sommets non consécutifs). Cette propriété assure qu'on peut facilement calculer une décomposition arborescente du graphe dont les clusters sont ses cliques maximales. Dans notre cas, nous avons choisi d'utiliser l'approche consistant à construire d'abord un graphe dont les sommets représentent les cliques maximales et il existe une arête entre deux sommets si les cliques maximales correspondantes ont une intersection qui n'est pas vide. Ensuite, un arbre recouvrant est calculé depuis ce graphe. Cet arbre induit une décomposition arborescente. Nous utilisons l'heuristique de triangulation Minfill [16] qui est connue pour sa capacité à produire des clusters de taille réduite.

L'espace mémoire nécessaire au stockage des (no)goods structurels dépend de la taille maximale des séparateurs entre les clusters. Il est très important de limiter cette valeur [8]. Comme [8], nous avons fixé la borne maximum à ne pas dépasser à 10. Si deux clusters ont une intersection dont la taille dépasse cette valeur, ils sont fusionnés pour former un seul cluster.

Une fois que la décomposition arborescente a été calculée, nous devons choisir un cluster racine, à partir duquel la recherche BTD sera commencée. Ce cluster racine est choisi aléatoirement parmi l'ensemble des clusters qui ont le plus grand nombre de variables. Les fils de chaque cluster sont ordonnés par taille de sépa-

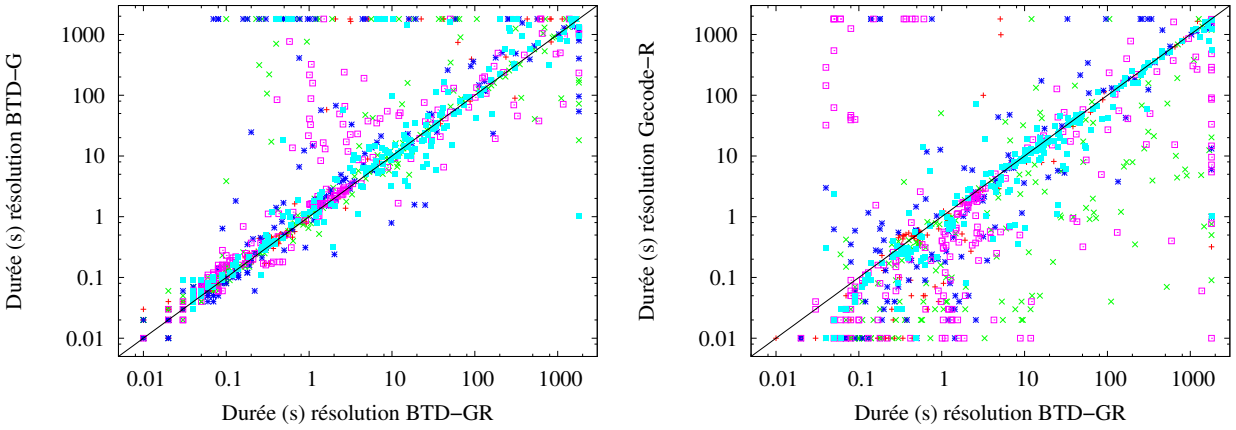

FIGURE 1 – Comparaison de BTD-G, BTD-GR et GecodeR. Chaque point (x, y) de la courbe de gauche (resp. de droite) correspond à une instance résolue en x secondes par BTD-GR, et y secondes par BTD-G (resp. GecodeR). La couleur et la forme des points dépendent de la classe de l'instance : $+$ pour C1, $\times$ pour C2, $*$ pour C3, $\square$ pour C4, et $\blacksquare$ pour C5.

rateur croissante : dans la boucle des lignes 5 à 10 de l'algorithme 1, nous choisissons en premier le fils i de r tel que $|K_i \cap K_r|$ soit minimal, puis le deuxième plus petit, etc.

Prétraitement sur les instances. Pour certaines instances, la 2-section n'est pas connexe. Dans ce cas, nous décomposons tout d'abord la 2-section en composantes connexes, et nous résolvons chaque sous-problème correspondant à chaque composante connexe séquentiellement. Cela est fait pour toutes les approches considérées (y compris GecodeR). Notons que le fait de résoudre chaque composante connexe séparément n'implique pas nécessairement que le sous-graphe de la 2-section induit par les variables d'un cluster soit nécessairement connexe, comme cela a été remarqué dans [13].

Pour chaque instance, nous demandons à Gecode de propager ses contraintes afin de filtrer ses domaines avant de commencer sa résolution. Gecode ne garantit pas que ce filtrage assure la cohérence d'arc généralisée pour toutes les contraintes : sur certaines, il peut être moins fort pour réduire le temps de calcul nécessaire. Néanmoins, ce filtrage suffit à résoudre 12 instances du benchmark. Par ailleurs, ce filtrage permet de réduire le domaine d'au moins une variable à une seule valeur pour 672 instances. Pour ces instances, nous éliminons de la 2-section les sommets correspondant à des variables dont le domaine est réduit à une seule valeur, avant de calculer la décomposition arborescente. Cela permet souvent d'avoir une meilleure décomposition en termes de taille des clusters : la largeur est diminuée pour 86% des 672 instances, et augmentée pour 6% de ces instances ; quand elle est diminuée, elle diminue de 37% (en moyenne, avec une diminution maximale de 98% pour une instance) ; quand elle est augmentée, elle augmente de 20% (en moyenne, avec une augmentation maximale de 31%).

5.3 Résultats expérimentaux

Parmi les 3285 instances du benchmark, 1880 ne sont pas structurées du tout dans le sens où leur décomposition arborescente ne contient qu'un seul cluster (contenant toutes les variables). Pour ces 1880 instances, BTD-G (resp. BTD-GR) se comportent comme Gecode sans restarts (resp. comme GecodeR). La seule différence est due au coût de calcul de la décomposition arborescente, qui est généralement très petit comparé au temps de résolution : en moyenne sur les 1880 instances non structurées, le coût de calcul de la décomposition arborescente est égal à 2% du temps CPU de résolution par GecodeR.

Parmi les 1405 instances pour lesquelles la décomposition arborescente comporte plus d'un cluster, 427 ne sont résolues par aucune des approches considérées dans la limite de temps CPU considérée (30 minutes). La figure 1 compare BTD-G avec BTD-GR (partie gauche), et BTD-GR avec GecodeR (partie droite) pour les 978 instances restantes (dont la décomposition comporte plus d'un cluster, et qui sont résolues par au moins une approche). La partie gauche nous montre que l'utilisation d'une politique de restarts pour chercher une affectation cohérente du cluster courant améliore la résolution pour un grand nombre d'instances, tandis que cela dégrade la résolution pour un nombre réduit d'instances. La partie droite nous montre que, si

	GecodeR					BTD-GR				
	C1	C2	C3	C4	C5	C1	C2	C3	C4	C5
1s	**4**	**13**	5	**46**	2	0	1	**6**	22	2
5s	1	**12**	2	12	**4**	1	1	**6**	**22**	2
10s	2	**12**	4	7	**5**	**3**	1	**7**	**21**	2
50s	1	**7**	3	9	**4**	**3**	4	**7**	**17**	1
100s	1	**6**	3	7	**3**	**3**	3	**7**	**16**	0
500s	1	**5**	3	5	0	**2**	2	**9**	**14**	0
1000s	1	2	3	3	0	1	2	**9**	**13**	0
1800s	1	**3**	3	0	**1**	1	2	**9**	**11**	0

TABLE 1 – Nombre d'instances bien structurées résolues par GecodeR et pas BTD-GR (partie gauche) et nombre d'instances bien structurées résolues par BTD-GR et non GecodeR (partie droite), pour différentes limites de temps CPU.

pour beaucoup d'instances la décomposition dégrade les performances, pour certaines instances elle l'améliore fortement. En particulier, 16 instances qui ne sont pas résolues par GecodeR dans la limite de temps de 1800 secondes sont résolues par BTD-GR en moins d'une seconde.

Regardons maintenant d'un peu plus près les instances pour lesquelles la décomposition arborescente est bonne, c'est-à-dire, les instances pour lesquelles le plus grand cluster de la décomposition arborescente comporte au plus cinquante pour cent des variables. Parmi les 1405 instances pour lesquelles la décomposition arborescente comporte plus d'un cluster, il y a 520 instances pour lesquelles la décomposition est bonne : 70 (resp. 126, 80, 103, and 141) instances de la classe C1 (resp. C2, C3, C4, et C5).

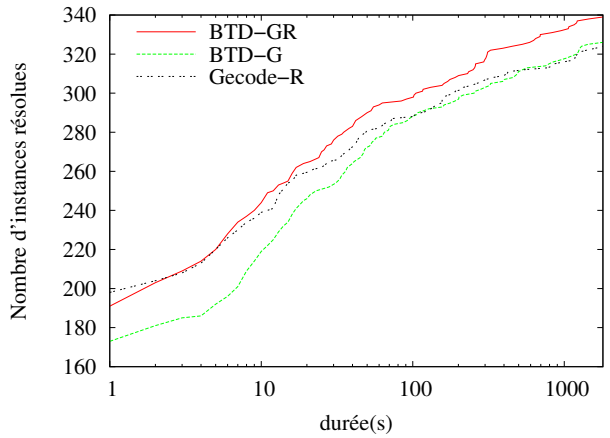

FIGURE 2 – Evolution du nombre d'instances bien structurées résolues par BTD-GR, BTD-G, et GecodeR en fonction du temps CPU.

La figure 2 nous montre que, pour ces instances bien structurées, BTD-GR est clairement plus performant que GecodeR, pour des limites de temps supérieures à 10 secondes, tandis que pour des limites de temps inférieures, les deux approches ont des résultats relativement similaires. Nous constatons également que BTD-GR est plus performant que BTD-G, ce qui confirme bien le fait que l'intégration d'une politique de restarts pour résoudre chaque cluster améliore le processus de résolution. Au bout de 1800 secondes, BTD-GR (resp. BTD-G et GecodeR) ont été capables de résoudre 339 (resp. 326 et 324) instances bien structurées.

La table 1 nous montre que chacune des deux approches BTD-GR et GecodeR est capable de résoudre des instances que l'autre n'est pas capable de résoudre. Il est intéressant de noter que les instances qui ne sont résolues que par BTD-GR appartiennent à toutes les classes, même si une majorité d'entre elles appartiennent à la classe C4 des instances n-aires définies en intention.

6 Conclusion

Nous avons montré dans cet article comment intégrer la méthode BTD dans une bibliothèque de programmation par contraintes, utilisée comme une boite noire pour résoudre des sous-problèmes. Cette intégration nous permet de bénéficier des algorithmes de résolution implémentés dans la bibliothèque (propagation de contraintes globales, heuristiques de choix de variables, restarts, etc). Nous avons réalisé cette intégration à l'aide de la bibliothèque Gecode. Les premières expérimentations sur le benchmark de la compétition CSP 2009 montrent tout le potentiel de cette intégration. En particulier, certaines instances difficiles pour la version considérée de Gecode sont bien mieux résolues en exploitant une décomposition arborescente.

Cette étude a également permis de montrer qu'une stratégie de restart peut être appliquée à l'intérieur de

chaque cluster, et que cette stratégie améliore les performances. Cette stratégie est complémentaire de celle proposée dans [13], et ces deux stratégies pourraient être combinées.

Cette première intégration de BTD dans une bibliothèque de contraintes nous ouvre de nombreuses perspectives de recherche. Dans la version actuelle, nous avons fait le choix de considérer toutes les contraintes initiales dans chaque cluster, comme cela est proposé dans [11, 13]. Cela permet à Gecode de filtrer au maximum les domaines, mais bien évidemment, cela peut être assez coûteux en temps. Une autre possibilité aurait été de restreindre l'ensemble des contraintes à celles dont la portée est incluse dans le cluster courant, ce qui est moins coûteux en temps mais peut également moins réduire les domaines des variables. Nous travaillons actuellement à l'extension de notre cadre d'intégration de BTD dans une bibliothèque de programmation par contraintes afin de pouvoir considérer différents sous-ensembles de contraintes, pour chaque sous-problème.

Références

[1] C. Bessière and J.-C. Régin. MAC and Combined Heuristics : Two Reasons to Forsake FC (and CBJ ?) on Hard Problems. In *Proceedings of CP*, pages 61–75, 1996.

[2] L. Blet, S. N. Ndiaye, and C. Solnon. Experimental comparison of BTD and intelligent backtracking : Towards an automatic per-instance algorithm selector. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP-2014)*, pages 190–206, 2014.

[3] Loïc Blet, Samba Ndojh Ndiaye, and Christine Solnon. A generic framework for solving csps integrating decomposition methods. In *CP doctoral program*, Quebec, Canada, 2012.

[4] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *ECAI-2004*, volume 16, page 146, 2004.

[5] C. P Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 431–437, 1998.

[6] G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124 :343–282, 2000.

[7] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3) :263–313, 1980.

[8] P. Jégou, S. N. Ndiaye, and C. Terrioux. Computing and exploiting tree-decompositions for solving constraint networks. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP-2005)*, pages 777–781, 2005.

[9] P. Jégou, S. N. Ndiaye, and C. Terrioux. Dynamic Management of Heuristics for Solving Structured CSPs. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP-2007)*, pages 364–378, 2007.

[10] P. Jégou, S. N. Ndiaye, and C. Terrioux. Combined Strategies for Decomposition-based Methods for solving CSPs. In *Proceedings of the International Conference on Tools with Artificial Intelligence (ICTAI 2009)*, pages 115–122, 2009.

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

[12] P. Jégou and C. Terrioux. Combining Restarts, Nogoods and Decompositions for Solving CSPs. In *Proceedings of the European Conference on Artifcial Intelligence (ECAI 2014)*, pages 465–470, 2014.

[13] P. Jégou and C. Terrioux. Tree-Decompositions with Connected Clusters for Solving Constraint Networks. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP-2014)*, pages 407–423, 2014.

[14] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artif. Intell.*, 139(1) :21–45, 2002.

[15] S. Karakashian, R. J. Woodward, and B. Y. Choueiry. Reformulating R(*, m)C with Tree Decomposition. In *Ninth International Symposium on Abstraction, Reformulation and Approximation (SARA 2011)*, pages 62–69, 2011.

[16] U. Kjaerulff. Triangulation of Graphs - Algorithms Giving Small Total State Space. Technical report, Judex R.R. Aalborg., Denmark, 1990.

[17] Christophe Lecoutre, Lakhdar Sais, SÃ©bastien Tabary, and Vincent Vidal. Recording and minimizing nogoods from restarts. *JSAT*, 1(3-4) :147–167, 2007.

[18] M. Morara, J. Mauro, and M. Gabbrielli. Solving xcsp problems by using gecode. In *26th Italian*

Conference on Computational Logic (CILC), volume 810 of *CEUR Workshop Proceedings*, pages 401–405. CEUR-WS.org, 2011.

[19] Cédric Pralet and Gérard Verfaillie. Travelling in the world of local searches in the space of partial assignments. In *CPAIOR*, pages 240–255, 2004.

[20] Patrick Prosser. Forward checking with backmarking. In *Constraint Processing, Selected Papers*, pages 185–204. Springer-Verlag, 1995.

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

[22] D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the eleventh ECAI*, pages 125–129, 1994.

[23] C. Schulte, G. Tack, and M. Z. Lagerkvist. Modeling. In Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist, editors, *Modeling and Programming with Gecode*. 2013. Corresponds to Gecode 4.2.1.