Décomposition et Good Recording pour le problème Max-CSP

Philippe Jégou
LSIS
Université d'Aix-Marseille III
Av. Escadrille Normandie-Niemen
13397 Marseille Cedex 20
email : philippe.jegou@univ.u-3mrs.fr

Cyril Terrioux
LSIS
Université d'Aix-Marseille III
Av. Escadrille Normandie-Niemen
13397 Marseille Cedex 20
email : cyril.terrioux@univ.u-3mrs.fr

Résumé

Dans le cadre du problème Max-CSP, la principale tâche à accomplir (en particulier si on s'intéresse à des instances issues du monde réel) est la recherche d'une solution optimale. Dans [TJ03], nous proposons une nouvelle méthode, appelée BTD, afin de résoudre les CSP valués et donc les instances Max-CSP. Cette méthode, qui repose à la fois sur des techniques énumératives et la notion de décomposition arborescente, se contente en fait de calculer le coût d'une solution optimale sans pour autant calculer cette solution. Dans un premier temps, nous mettons en avant les origines de ce problème et nous proposons une méthode afin de calculer efficacement une solution optimale à partir du coût optimal fourni par BTD.

Dans un second temps, nous nous intéressons à l'efficacité pratique de BTD. Grâce à la notion de goods valués, BTD vise à tirer profit de l'efficacité de l'énumération tout en garantissant des bornes de complexité en temps meilleures que celles des méthodes énumératives. Mais qu'en est-il en pratique ? Aucun résultat n'est présenté dans [TJ03]. Aussi, nous montrons, dans cet article, l'intérêt pratique de BTD pour la résolution d'instances aléatoires structurées et d'instances issues du monde réel.

1 Introduction

De nombreux problèmes, comme les problèmes de satisfaction d'une formule booléenne (SAT), de configuration, de coloration de graphes, de planification,..., peuvent être exprimés sous la forme d'un problème de satisfaction de contraintes (CSP). Un CSP est défini par un ensemble de variables (chacune ayant un domaine fini) et un ensemble de contraintes. Chaque contrainte interdit certaines combinaisons de valeurs pour un sous-ensemble de variables donné. Résoudre un CSP requiert d'instancier chaque variable avec une valeur de telle sorte que toutes les contraintes soient satisfaites. Déterminer si un CSP possède ou non une solution est un problème NP-Complet. Lorsqu'on considère des problèmes issus du monde réel, on se rend compte qu'ils font souvent intervenir deux sortes de contraintes : des contraintes dures qui représentent certaines propriétés physiques et des contraintes molles qui expriment des notions comme la possibilité ou la préférence. Les premières doivent être absolument satisfaites tandis que les secondes peuvent être violées. Malheureusement, représenter ces problèmes dans le formalisme CSP (où chaque contrainte doit être satisfaite) conduit souvent à l'obtention de problèmes sur-contraints dépourvus de solution. Toutefois, même s'il n'existe pas de solution idéale, il peut se révéler intéressant et important de trouver une affectation qui optimise un critère donné portant sur la satisfaction de contraintes. Ainsi, récemment, plusieurs extensions du formalisme CSP ont été proposées (comme, par exemple, [FW92, BMR95, SFV95]).

Dans ce papier, nous nous focalisons sur le problème Max-CSP [FW92]. Résoudre une instance Max-CSP équivaut à trouver une affectation des variables qui maximise le nombre de contraintes satisfaites. Par le passé, plusieurs algorithmes de résolution ont été définis. Une partie d'entre eux exploite des techniques énumératives comme le Branch and Bound (BB) ou la notion de consistance d'arc [LMS99, Lar02, LS03, CS04]. D'autres méthodes, elles, reposent sur l'approche par programmation dynamique [VLS96, Kos99, MS00, MS01, MSV02, LD03]. Parmi elles, quelques-unes (par exemple [Kos99, MS00, LMS02, LD03]) tirent profit de la structure du problème. Ces différentes approches ont fourni, dans des cas différents, des résultats intéressants.

Dans [TJ03] nous proposons, dans le cadre du problème des CSP valués ([SFV95] qui constitue une généralisation du problème Max-CSP), une méthode hybride, nommée BTD. Cette méthode est basée, à la fois, sur des techniques énumératives et sur la notion de décomposition arborescente. Son objectif est de bénéficier de l'efficacité pratique de l'énumération tout en garantissant des bornes de complexité en temps meilleures que celles des méthodes énumératives. A la lecture de [TJ03], deux questions fondamentales pour BTD peuvent se poser. La première concerne le calcul d'une solution optimale. En effet, la méthode BTD telle qu'elle a été définie dans [TJ03] se contente de calculer le coût de la meilleure affectation, sans pour autant fournir cette affectation. Or, l'obtention d'une solution optimale est une des tâches les plus importantes pour un solveur, en particulier lorsqu'on considère des instances issues du monde réel. Donc, la question qui se pose est de savoir comment calculer efficacement une solution optimale à partir du coût optimal fourni par BTD. La seconde question porte sur l'efficacité pratique de BTD. Cette méthode a déjà montré son intérêt pratique pour la résolution de CSP classiques [JT03]. En revanche, son comportement sur le problème Max-CSP reste inconnu et doit être étudié. Dans cet article, nous répondons à ces deux questions.

Le papier est organisé ainsi. La section 2 rappelle les notions de base sur les CSP et les Max-CSP. La section 3 est consacrée à la méthode BTD. Puis, nous expliquons comment

calculer efficacement une solution optimale dans la section 4, avant de présenter des résultats expérimentaux concernant le comportement pratique de BTD dans la section 5. Enfin, nous concluons et donnons quelques perspectives pour la méthode BTD dans la section 6.

2 Notions de base

Un *problème de satisfaction de contraintes* (CSP) peut être défini par un quadruplet (X, D, C, R). X est un ensemble $\{x_1, \ldots, x_n\}$ de n variables. Chaque variable x_i prend ses valeurs dans le domaine fini d_{x_i} issu de D. Ces variables sont soumises aux contraintes de C. Chaque contrainte c est définie comme un ensemble $\{x_{c_1}, \ldots, x_{c_k}\}$ de variables. Une relation r_c (issue de R) est associée à chaque contrainte c telle que r_c représente l'ensemble des tuples autorisés sur $d_{x_{c_1}} \times \cdots \times d_{x_{c_k}}$. Notons qu'il est également possible de définir les contraintes en intention en utilisant par exemple des fonctions ou des prédicats. Etant donné $Y \subseteq X$ tel que $Y = \{x_1, \ldots, x_k\}$, une *affectation* (ou *instanciation*) des variables de Y est un tuple $\mathcal{A} = (v_1, \ldots, v_k)$ de $d_{x_1} \times \cdots \times d_{x_k}$. Une contrainte c est dite *satisfaite* par $\mathcal{A}$ si $c \subseteq Y, (v_1, \ldots, v_k)[c] \in r_c$, *violée* sinon. Nous notons l'affectation $(v_1, \ldots, v_k)$ sous la forme plus explicite $(x_1 \leftarrow v_1, \ldots, x_k \leftarrow v_k)$. Dans ce papier, sans perte de généralité, nous considérons des contraintes binaires (c'est-à-dire des contraintes qui impliquent exactement deux variables). Aussi, la structure d'un CSP peut être représentée par le graphe (X, C), appelé *graphe de contraintes*, dont les sommets sont les variables de X et pour lequel il existe une arête entre deux sommets si les variables correspondantes sont liées par une contrainte. Etant donnée une instance, le problème CSP consiste à déterminer s'il existe une instanciation de toutes les variables qui satisfait toutes les contraintes. Ce problème est NP-Complet. Malheureusement, lorsqu'on représente des problèmes issus du monde réel sous la forme de CSP, on obtient fréquemment des problèmes sur-contraints dépourvus de solution. Toutefois, même s'il n'existe pas de solution idéale, il peut se révéler intéressant et important de trouver une affectation qui optimise un critère donné portant sur la satisfaction de contraintes. Ainsi, récemment, plusieurs extensions du formalisme CSP ont été proposées (comme, par exemple, [FW92, BMR95, SFV95]).

Dans ce papier, nous nous concentrons sur le problème Max-CSP [FW92]. Résoudre une instance Max-CSP équivaut à trouver une affectation qui maximise le nombre de contraintes satisfaites. Autrement dit, on va rechercher une affectation des variables qui minimise le nombre de contraintes violées. Le nombre de contraintes violées par une affectation est appelé le *coût* de cette affectation. Par le passé, plusieurs algorithmes de résolution ont été définis. Ils reposent généralement sur des techniques énumératives ou sur des approches par programmation dynamique. Les méthodes énumératives exploitent un minorant, qui sous-estime le coût de la meilleure extension complète de l'affectation courante, et un majorant qui correspond généralement au coût de la meilleure solution connue. Ainsi, si le minorant reste inférieur au majorant, elles étendent l'affectation courante en instanciant une nouvelle variable. Sinon, elles reviennent en arrière en essayant une nouvelle valeur pour la dernière variable instanciée. Si toutes les valeurs ont été essayées, elles reviennent en arrière une nouvelle fois, et ainsi de suite. L'efficacité des méthodes énumératives dépend grandement de la qualité du minorant et du majorant. Plus grand est le minorant (respectivement plus petit est le majorant), plus le pouvoir de coupe (et donc le nombre de nœuds et de tests de contraintes économisés) sera im-

portant. La méthode énumérative de base est le Branch and Bound (BB). Elle utilise simplement comme minorant le coût de l'affectation courante et comme majorant le coût de la meilleure solution connue. Plusieurs améliorations ont été proposées à partir du cadre des CSP classiques. Par exemple, le minorant peut être amélioré en utilisant des techniques prospectives comme le Forward-Checking (FC [FW92]) ou la notion de consistance d'arc [LMS99, Lar02, LS03, CS04]. D'autres méthodes, elles, reposent sur l'approche par programmation dynamique [VLS96, Kos99, MS00, MS01, MSV02, LD03]. Elles divisent le problème en plusieurs sous-problèmes. Puis, elles résolvent chaque sous-problème et mémorisent des informations sur ces résolutions. Ces informations sont exploitées, par la suite, pour résoudre un sous-problème plus important, et ainsi de suite jusqu'à ce que le problème entier soit résolu. Par exemple, dans certains cas, ces informations sont employées pour produire un bon minorant ou un bon majorant comme par exemple dans la méthode RDS (Russian Dolls Search [VLS96]) et ses variantes [MS00, MS01, LMS02, MSV02]. Parmi ces méthodes, quelques-unes (par exemple [Kos99, MS00, LMS02, LD03]) tirent profit de la structure du problème. D'un point de vue pratique, les méthodes énumératives qui exploitent la notion de consistance d'arc obtiennent de bons résultats quand les instances à résoudre ont une taille raisonnable. Cependant, elles semblent éprouver quelques difficultés sur des instances de taille plus importante comme par exemple les instances réelles du CELAR [CdGL$^+$99]. D'autre part, les méthodes basées sur la programmation dynamiques paraissent effectuer des recherches redondantes. Néanmoins, en pratique, elles obtiennent des résultats intéressants. Par exemple, RDS [VLS96] et la méthode structurelle de Koster [Kos99] parviennent à résoudre l'instance SCEN-06 du CELAR (qui est une des plus difficiles).

3 L'algorithme BTD

Dans [TJ03] nous proposons, dans le cadre du problème des CSP valués ([SFV95] qui constitue une généralisation du problème Max-CSP), une méthode hybride, nommée BTD (pour Backtracking with Tree-Decomposition). Il s'agit d'une méthode énumérative qui est guidée par une décomposition arborescente du graphe de contraintes. Une *décomposition arborescente* [RS86] d'un graphe $G = (X, E)$ est un couple $(\mathcal{C}, \mathcal{T})$ avec $\mathcal{T} = (I, F)$ un arbre et $\mathcal{C} = \{\mathcal{C}_i : i \in I\}$ une famille de sous-ensembles de X tels que chaque cluster $\mathcal{C}_i$ est un nœud de $\mathcal{T}$ et vérifie : (1) $\cup_{i \in I} \mathcal{C}_i = X$, (2) pour chaque arête $\{x, y\} \in E$, il existe $i \in I$ avec $\{x, y\} \subseteq \mathcal{C}_i$, (3) pour tout $i, j, k \in I$, si k est sur un chemin allant de i à j dans $\mathcal{T}$, alors $\mathcal{C}_i \cap \mathcal{C}_j \subseteq \mathcal{C}_k$. La largeur d'une décomposition arborescente $(\mathcal{C}, \mathcal{T})$ est égale à $max_{i \in I}|\mathcal{C}_i| - 1$. La *largeur d'arbre* (*tree-width*) de G est la largeur minimale sur toutes les décompositions de G. Remarquons que trouver une décomposition arborescente optimale est un problème NP-Difficile [ACP87]. Cependant, nous pouvons toujours produire une décomposition arborescente satisfaisante en utilisant la notion de *graphes triangulés*. La figure 1(b) présente une décomposition arborescente possible pour le graphe de la figure 1(a). Ainsi, nous obtenons $\mathcal{C}_1 = \{x_1, x_2, x_3\}$, $\mathcal{C}_2 = \{x_2, x_3, x_4, x_5\}$, $\mathcal{C}_3 = \{x_4, x_5, x_6\}$ et $\mathcal{C}_4 = \{x_3, x_7, x_8\}$, et la largeur d'arbre est égale à trois. Ensuite, à partir d'une décomposition donnée, nous considérons l'arborescence (I, F) dont $\mathcal{C}_1$ est la racine et nous notons $Desc(\mathcal{C}_j)$ l'ensemble des variables qui appartiennent à $\mathcal{C}_j$ ou à l'un de ses descendants $\mathcal{C}_k$ dans l'arborescence enracinée en $\mathcal{C}_j$. Par exemple, nous avons $Desc(\mathcal{C}_2) = \mathcal{C}_2 \cup \mathcal{C}_3 = \{x_2, x_3, x_4, x_5, x_6\}$.

La première étape de BTD consiste à calculer une décomposition arborescente du

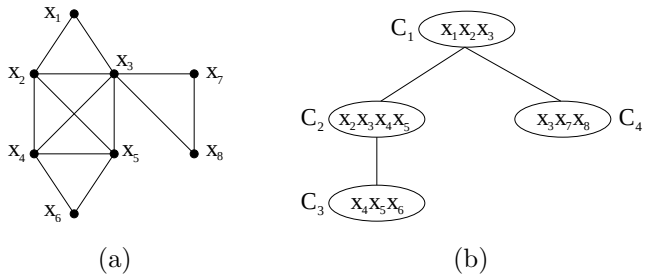

FIG. 1 – (a) Un graphe de contraintes portant sur 8 variables. (b) Une décomposition arborescente de ce graphe.

graphe de contraintes. La décomposition obtenue induit un ordre partiel sur les variables qui permet à BTD d'exploiter certaines propriétés structurelles du graphe et ainsi d'élaguer certaines parties de l'espace de recherche. En fait, les variables sont instanciées selon un parcours en profondeur d'abord de l'arborescence de clusters. En d'autres termes, nous affectons en premier les variables du cluster racine $\mathcal{C}_1$, puis celles de $\mathcal{C}_2$, ensuite celles de $\mathcal{C}_3$, ... A l'intérieur d'un cluster, on peut choisir librement l'ordre d'instanciation des variables. Ainsi, d'un point de vue pratique, BTD pourra utiliser un ordre statique ou dynamique à l'intérieur de chaque cluster. Par exemple, $x_1, x_2, \ldots, x_8$ et x_2, x_1, x_3, x_5, x_4, x_6, x_7, x_8 sont deux ordres possibles sur les variables. En outre, la décomposition arborescente et l'ordre induit sur les variables permettent à BTD de diviser le problème en plusieurs sous-problèmes. Etant donnés deux clusters $\mathcal{C}_i$ et $\mathcal{C}_j$ (avec $\mathcal{C}_j$ un fils de $\mathcal{C}_i$), la définition du sous-problème enraciné en $\mathcal{C}_j$ dépend de l'affectation courante $\mathcal{A}$ sur $\mathcal{C}_i \cap \mathcal{C}_j$. Ce sous-problème est noté $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$. Son ensemble de variables est $Desc(\mathcal{C}_j)$. Le domaine de chaque variable qui appartient à $\mathcal{C}_i \cap \mathcal{C}_j$ est restreint à la valeur que possède cette variable dans $\mathcal{A}$. Concernant l'ensemble des contraintes, il contient les contraintes qui portent sur au moins une variable qui apparaît exclusivement dans $\mathcal{C}_j$ ou dans un de ses descendants. Par exemple, considérons le CSP dont le graphe de contraintes est présenté à la figure 1(a), dont le domaine de chacune des huit variables est $\{1, 2, 3\}$, et dont chaque contrainte $c_{ij} = \{x_i, x_j\}$ signifie $x_i \neq x_j$. Etant donnée l'affectation $\mathcal{A} = (x_2 \leftarrow 2, x_3 \leftarrow 2)$, l'ensemble des variables de $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2}$ est $Desc(\mathcal{C}_2)$, (avec $d_{x_2} = d_{x_3} = \{2\}$ et $d_{x_4} = d_{x_5} = d_{x_6} = \{1, 2, 3\}$) et celui des contraintes $\{c_{24}, c_{25}, c_{34}, c_{35}, c_{45}, c_{46}, c_{56}\}$. Notons que la contrainte c_{23} ne figure pas dans cet ensemble car les variables x_2 et x_3 appartiennent toutes les deux à $\mathcal{C}_1$. Remarquons que cette définition des sous-problèmes permet de définir une partition de l'ensemble de contraintes C. Une telle partition est primordiale puisqu'elle garantit que BTD prend en compte chaque contrainte une et une seule fois et donc que la méthode calcule correctement le coût de chaque affectation. Enfin, la notion de décomposition arborescente permet de définir la notion de *good structurel valué* (par analogie à la notion de nogood). Un good structurel valué de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$ (avec $\mathcal{C}_j$ un fils de $\mathcal{C}_i$) est une paire $(\mathcal{A}, v)$ avec $\mathcal{A}$ une affectation sur $\mathcal{C}_i \cap \mathcal{C}_j$ et v le coût optimal du sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$. Par exemple, si nous considérons l'affectation $\mathcal{A} = (x_2 \leftarrow 2, x_3 \leftarrow 2)$ sur $\mathcal{C}_1 \cap \mathcal{C}_2$, nous obtenons le good $(\mathcal{A}, 0)$ puisque $(x_2 \leftarrow 2, x_3 \leftarrow 2, x_4 \leftarrow 1, x_5 \leftarrow 3, x_6 \leftarrow 2)$ est une affectation optimale sur $Desc(\mathcal{C}_2)$ (qui ne viole aucune contrainte car c_{23} n'est pas une contrainte de $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2}$).

La figure 2 décrit l'algorithme BTD basé sur BB. La méthode BTD explore l'espace de recherche suivant l'ordre sur les variables induit par la décomposition arborescente. Aussi, elle commence par instancier les variables du cluster racine $\mathcal{C}_1$. A l'intérieur d'un cluster $\mathcal{C}_i$, elle procède de façon classique à l'image de BB en affectant une valeur à une variable, en maintenant et comparant un minorant et un majorant, et en revenant en arrière si le minorant dépasse ou égale le majorant. Le minorant et le majorant de BTD sont similaires à ceux de BB, si ce n'est qu'ils ne tiennent compte que des contraintes du sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_{p(i)}/\mathcal{C}_i}$ (avec $\mathcal{C}_{p(i)}$ le cluster parent de $\mathcal{C}_i$ et $\mathcal{A}$ l'affectation courante sur $\mathcal{C}_i \cap \mathcal{C}_{p(i)}$). Le minorant correspond au coût de l'affectation courante sur $Desc(\mathcal{C}_i)$ alors que le majorant est défini comme le coût de la meilleure solution connue pour le sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_{p(i)}/\mathcal{C}_i}$. Quand toutes les variables de $\mathcal{C}_i$ sont instanciées, si le minorant est inférieur au majorant, BTD poursuit sa recherche avec le premier fils de $\mathcal{C}_i$ (s'il en existe un). Plus généralement, considérons un fils $\mathcal{C}_j$ de $\mathcal{C}_i$. Etant donnée l'affectation courante $\mathcal{A}$ sur $\mathcal{C}_i$, BTD vérifie si l'instanciation $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ correspond à un good structurel valué. Si c'est le cas, la méthode BTD ajoute le coût associé à ce good au minorant. Sinon, elle étend l'affectation $\mathcal{A}$ sur $Desc(\mathcal{C}_j)$ afin de calculer le coût optimal v du sous-problème $\mathcal{P}_{\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j],\mathcal{C}_i/\mathcal{C}_j}$. Puis, elle ajoute v au minorant avant de mémoriser le good valué $(\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j], v)$. Si, après avoir examiné le fils $\mathcal{C}_j$, le minorant est toujours inférieur au majorant, BTD continue son exploration avec le prochain fils de $\mathcal{C}_i$. Enfin, si un échec survient, BTD revient en arrière en tentant de modifier l'instanciation courante sur $\mathcal{C}_i$.

Dans la figure 2, étant donnés une affectation $\mathcal{A}$ et un cluster $\mathcal{C}_i$, BTD recherche la meilleure affectation $\mathcal{B}$ sur $Desc(\mathcal{C}_i)$ telle que $\mathcal{A}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}] = \mathcal{B}[\mathcal{C}_i \backslash V_{\mathcal{C}_i}]$ et telle que le coût de $\mathcal{B}$ soit inférieur à $\alpha_{\mathcal{C}_i}$. $V_{\mathcal{C}_i}$ correspond à l'ensemble des variables non instanciées de $\mathcal{C}_i$, $l_{\mathcal{C}_i}$ au minorant et $\alpha_{\mathcal{C}_i}$ au majorant par rapport au sous-problème $\mathcal{P}_{\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_{p(i)}],\mathcal{C}_{p(i)}/\mathcal{C}_i}$. Si BTD trouve une telle affectation, il retourne son coût, sinon il retourne un coût supérieur ou égal à $\alpha_{\mathcal{C}_i}$. L'appel initial est BTD$(\emptyset, \mathcal{C}_1, \mathcal{C}_1, 0, +\infty)$.

Enfin, BTD a une complexité en espace en $O(n.s.d^s)$ pour une complexité en temps en $O(n.s^2.m.\log(d).d^{w+1})$ avec $w+1$ la taille du plus grand cluster $\mathcal{C}_k$ et s la taille de la plus grande intersection $\mathcal{C}_i \cap \mathcal{C}_j$ avec $\mathcal{C}_j$ un fils de $\mathcal{C}_i$ [TJ03]. Bien entendu, ces complexités supposent qu'une décomposition arborescente a été calculée au préalable (les paramètres structurels w et s étant relatifs à cette décomposition).

4 Comment calculer une solution optimale ?

La méthode BTD se contente de fournir le coût optimal α de l'instance à résoudre. En aucun cas, elle ne calcule explicitement une solution optimale de cette instance. En effet, lorsque BTD exploite un good de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$, il n'affecte pas une nouvelle fois les variables de $Desc(\mathcal{C}_j) - (\mathcal{C}_i \cap \mathcal{C}_j)$. Ce qui est appelé dans [JT03, TJ03] un *forward-jump* (par analogie avec la notion de backjump). Par exemple, après avoir instancié les variables x_1, x_2 et x_3 de $\mathcal{C}_1$, BTD va tester si l'affectation sur $\mathcal{C}_1 \cap \mathcal{C}_2 = \{x_2, x_3\}$ correspond à un good. Si tel est le cas, BTD exploite ce good et n'explore pas le sous-problème formé par $Desc(\mathcal{C}_2)$ une nouvelle fois. Il va directement tester si l'affectation sur $\mathcal{C}_1 \cap \mathcal{C}_4$ correspond à un good. Ainsi, comme plusieurs variables ne sont pas instanciées (en l'occurrence les variables x_4, x_5 et x_6), BTD ne peut pas produire une solution du problème à résoudre. Il cherche seulement son coût optimal. Même si calculer le coût optimal peut avoir son intérêt, la principale tâche dans le cadre du problème Max-CSP reste la recherche d'une solution optimale. Aussi, cela soulève une question fondamentale

BTD$(\mathcal{A}, \mathcal{C}_i, V_{\mathcal{C}_i}, l_{\mathcal{C}_i}, \alpha_{\mathcal{C}_i})$
1. **Si** $V_{\mathcal{C}_i} = \emptyset$
2. **Alors**
3. $F \leftarrow Fils(\mathcal{C}_i)$
4. **Tant que** $F \neq \emptyset$ **et** $l_{\mathcal{C}_i} < \alpha_{\mathcal{C}_i}$ **Faire**
5. Choisir $\mathcal{C}_j$ dans F
6. $F \leftarrow F \backslash \{\mathcal{C}_j\}$
7. **Si** $(\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j], v)$ est un good de $\mathcal{C}_i/\mathcal{C}_j$ dans G **Alors** $l_{\mathcal{C}_i} \leftarrow l_{\mathcal{C}_i} + v$
8. **Sinon**
9. $v \leftarrow$ **BTD**$(\mathcal{A}, \mathcal{C}_j, \mathcal{C}_j \backslash (\mathcal{C}_j \cap \mathcal{C}_i), 0, \alpha_{\mathcal{C}_1})$
10. $l_{\mathcal{C}_i} \leftarrow l_{\mathcal{C}_i} + v$
11. Mémoriser le good $(\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j], v)$ de $\mathcal{C}_i/\mathcal{C}_j$ dans G
12. **FinSi**
13. **FinTantque**
14. Retourner $l_{\mathcal{C}_i}$
15. **Sinon**
16. Choisir $x \in V_{\mathcal{C}_i}$
17. $d \leftarrow d_x$
18. **Tant que** $d \neq \emptyset$ **et** $l_{\mathcal{C}_i} < \alpha_{\mathcal{C}_i}$ **Faire**
19. Choisir a dans d
20. $d \leftarrow d \backslash \{a\}$
21. $l_a \leftarrow |\{c = \{x, y\} \in C | y \notin V_{\mathcal{C}_i}$ *et* $\mathcal{A} \cup \{x \leftarrow a\}$ *viole* $c\}|$
22. **Si** $l_{\mathcal{C}_i} + l_a < \alpha_{\mathcal{C}_i}$
23. **Alors** $\alpha_{\mathcal{C}_i} \leftarrow \min(\alpha_{\mathcal{C}_i},$ **BTD**$(\mathcal{A} \cup \{x \leftarrow a\}, \mathcal{C}_i, V_{\mathcal{C}_i} \backslash \{x\}, l_{\mathcal{C}_i} + l_a, \alpha_{\mathcal{C}_i}))$
24. **FinSi**
25. **FinTantque**
26. Retourner $\alpha_{\mathcal{C}_i}$
27. **FinSi**

FIG. 2 – L'algorithme BTD.

pour BTD : comment calculer efficacement une solution optimale à partir du coût optimal fourni par BTD ? Plus généralement, cette question concerne les méthodes qui réalisent des compromis entre le temps et l'espace et qui ainsi ne mémorisent qu'une partie des informations nécessaires à la production d'une solution optimale. C'est, par exemple, le cas de l'adaptation du Tree-Clustering proposée dans [DF01]. Cette adaptation possède une complexité en espace limitée, mais souffre en contrepartie du même défaut que BTD. En effet, elle mémorise des informations uniquement sur les séparateurs et se trouve donc dans l'incapacité de produire une solution sans effectuer une énumération sur les variables non instanciées.

Dans cette section, nous expliquons comment construire une solution optimale à partir du coût optimal α. Une façon basique consiste à utiliser n'importe quel algorithme énumératif pour rechercher une instanciation de coût α. Une telle méthode est clairement inefficace et a une complexité en temps pire que celle de BTD. De plus, en procédant ainsi, nous ne tirons pas parti de la décomposition arborescente, ni des goods mémorisés par BTD durant la recherche. Ainsi, il serait préférable de concevoir une méthode dérivée

de BTD qui exploiterait les goods produits par BTD. Par exemple, étant donné un cluster $\mathcal{C}_i$, nous pouvons chercher une affectation $\mathcal{A}$ sur $\mathcal{C}_i$ telle que pour chaque fils $\mathcal{C}_j$ de $\mathcal{C}_i$, $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ soit un good. Cette méthode a une complexité similaire à celle de BTD. Cependant, il est évident qu'en pratique, elle visite moins de nœuds et effectue moins de tests de contraintes que BTD (sauf dans le cas où il n'y aurait qu'un seul cluster). Cette méthode est meilleure que la première, mais, elle semble encore trop coûteuse car le nombre de goods mémorisés peut être important. Donc, pour des raisons d'efficacité, nous devons restreindre le nombre de goods susceptibles d'être utilisés pour guider la recherche. L'idéal serait d'effectuer la recherche de la solution optimale en ne conservant qu'un seul good par intersection $\mathcal{C}_i \cap \mathcal{C}_j$. En fait, ce cas idéal peut être atteint en mémorisant quelques informations supplémentaires lorsque BTD utilise ou mémorise un good. Notons bien que BTD continue de mémoriser tous les goods qu'il découvre, la restriction à un good par intersection ne s'appliquant qu'à la phase de recherche d'une solution optimale.

Conserver un seul good par intersection signifie que pour chaque intersection, on garde le good qui participe à la solution optimale dont BTD a estimé le coût. Pour chaque intersection, il faut donc pouvoir identifier ce good. Bien entendu, lorsque BTD recherche le coût d'une solution optimale et qu'il mémorise un good, il n'est pas en mesure de dire si ce good participe ou non à une solution optimale. L'identification doit donc s'effectuer après avoir appliqué BTD. Par contre, durant la recherche, BTD peut mémoriser, pour chaque fils $\mathcal{C}_j$ du cluster racine $\mathcal{C}_1$, quel est le good sur $\mathcal{C}_1 \cap \mathcal{C}_j$ qui participe à la meilleure solution connue. Ainsi, à l'issue de BTD, on connaît, pour chaque fils $\mathcal{C}_j$ de $\mathcal{C}_1$, le good sur $\mathcal{C}_1 \cap \mathcal{C}_j$ qui participe à la solution optimale dont BTD vient d'estimer le coût. Pour ne conserver qu'un seul good par intersection, il ne reste plus qu'à établir, pour chaque good g de $\mathcal{C}_1$ par rapport à $\mathcal{C}_j$, quel est le good de $\mathcal{C}_k$ par rapport à $\mathcal{C}_l$ (avec $\mathcal{C}_k$ et $\mathcal{C}_l$ des clusters inclus dans $Desc(\mathcal{C}_j)$) qui est utilisé pour construire g. Autrement dit, dans notre exemple, si on connaît les deux goods sur $\mathcal{C}_1 \cap \mathcal{C}_2$ et sur $\mathcal{C}_1 \cap \mathcal{C}_4$ qui participent à la solution optimale, il ne reste plus qu'à déterminer le good correspondant sur $\mathcal{C}_2 \cap \mathcal{C}_3$. Cette tâche peut être accomplie à moindre coût durant la recherche effectuée par BTD. La principale difficulté provient alors de l'utilisation des forward-jumps durant la recherche. En effet, quand BTD exploite un good de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$, il ne visite pas à nouveau le sous-problème enraciné en $\mathcal{C}_j$. Ainsi, il ne teste pas les goods de $\mathcal{C}_j$ par rapport à n'importe lequel de ses fils. Par exemple, en utilisant un good sur $\mathcal{C}_1 \cap \mathcal{C}_2$, BTD ne teste pas les goods de $\mathcal{C}_2$ par rapport à $\mathcal{C}_3$. Autrement dit, la connaissance du good sur $\mathcal{C}_1 \cap \mathcal{C}_2$ qui participe à la solution optimale recherchée n'indique pas pour autant quel good sur $\mathcal{C}_2 \cap \mathcal{C}_3$ participe à cette solution. Il est donc nécessaire de mémoriser des informations supplémentaires durant la recherche effectuée par BTD. Aussi, quand BTD enregistre un nouveau good g de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$, il doit également mémoriser, pour chaque fils $\mathcal{C}_k$ de $\mathcal{C}_j$, le good sur $\mathcal{C}_j \cap \mathcal{C}_k$ utilisé pour construire le good g. En appliquant récursivement ce principe, on peut, à partir d'un good donné g de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$, établir quel est le good de $\mathcal{C}_k$ par rapport à $\mathcal{C}_l$ (avec $\mathcal{C}_k$ et $\mathcal{C}_l$ des clusters inclus dans $Desc(\mathcal{C}_j)$) qui est utilisé pour construire g. Ainsi, si, pour chaque fils $\mathcal{C}_j$ du cluster racine $\mathcal{C}_1$, on sait quel good sur $\mathcal{C}_1 \cap \mathcal{C}_j$ participe à la solution optimale, on parvient à ne conserver qu'un seul good par intersection. Grâce à une structure de données convenable, l'enregistrement de ces informations supplémentaires ne change pas les complexités en temps et en espace de BTD.

Ensuite, pour construire la solution optimale recherchée, nous instancions d'abord les

variables qui apparaissent dans au moins une intersection $\mathcal{C}_i \cap \mathcal{C}_j$ avec la valeur qu'elles ont dans le good correspondant. A l'issue de cet étape, certaines variables ne sont pas instanciées. Nous notons $\mathcal{U}_{\mathcal{C}_i}$ l'ensemble des variables non instanciés dans chaque cluster $\mathcal{C}_i$. Trivialement, nous avons $\mathcal{U}_{\mathcal{C}_i} = \mathcal{C}_i - (\mathcal{C}_{p(i)} \cup \bigcup_{\mathcal{C}_j \in Fils(\mathcal{C}_i)} \mathcal{C}_j)$. Dans notre exemple, il resterait à instancier les variables x_6, x_7 et x_8. On aurait donc $\mathcal{U}_{\mathcal{C}_1} = \mathcal{U}_{\mathcal{C}_2} = \emptyset, \mathcal{U}_{\mathcal{C}_3} = \{x_6\}$ et $\mathcal{U}_{\mathcal{C}_4} = \{x_7, x_8\}$. Pour chaque cluster $\mathcal{C}_i$, nous considérons le sous-problème défini par le sous-graphe $(\mathcal{C}_i, C \cap (\mathcal{C}_i \times (\mathcal{C}_i \backslash \mathcal{C}_{p(i)})))$ du graphe de contraintes (X, C). Pour chaque sous-problème, les seules variables à instancier sont celles de $\mathcal{U}_{\mathcal{C}_i}$. Pour trouver la valeur que possède chacune de ces variables dans la solution optimale, on va effectuer une recherche énumérative. Nous pouvons remarquer que ces sous-problèmes sont indépendants puisque chaque intersection $\mathcal{C}_i \cap \mathcal{C}_j$ constitue un séparateur du graphe de contraintes (X, C). Donc, nous pouvons les résoudre indépendamment les uns des autres. Pour chaque problème $\mathcal{C}_i$, la valeur initiale du minorant (si nous utilisons BB) est donnée par :

$$|\{c = \{x, y\} \in C | \exists \mathcal{C}_j \in Fils(\mathcal{C}_i), x \in \mathcal{C}_i \text{ et } y \in \mathcal{C}_i \cap \mathcal{C}_j \text{ et } \mathcal{A} \text{ viole } c\}| + \sum_{\mathcal{C}_j \in Fils(\mathcal{C}_i)} \alpha_{\mathcal{C}_j}$$

où $\mathcal{A} = \bigcup_{\mathcal{C}_j \in Fils(\mathcal{C}_i)} \mathcal{A}_{\mathcal{C}_j}$ et $(\mathcal{A}_{\mathcal{C}_j}, \alpha_{\mathcal{C}_j})$ est le good de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$ qui a été conservé. Clairement, les deux termes de cette somme portent sur des ensembles de contraintes différents. Concernant le majorant, il s'agit simplement de $\alpha_{\mathcal{C}_i}$. La complexité en temps de cette méthode est, dans le pire des cas, $O(nmd^k)$ où k est la taille du plus grand ensemble $\mathcal{U}_{\mathcal{C}_i}$. Bien sûr, construire une solution optimale requiert toujours une énumération, mais celle-ci est limitée à son strict minimum. Pour éviter une telle énumération, il faudrait mémoriser, pour chaque good $(\mathcal{A}, v)$ de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$, l'affectation sur $\mathcal{C}_j$ qui participe à une solution optimale du problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$. La complexité en espace de BTD serait alors en $O(n.w.d^s)$ au lieu de $O(n.s.d^s)$. Cette dernière solution n'est pas toujours envisageable en pratique. En effet, le paramètre s permet de contrôler le compromis espace-temps. Ainsi, à partir d'une décomposition arborescente avec une valeur de s donnée, il est possible de produire une décomposition arborescente possédant une valeur de s inférieure (voir [DF01, JT03] pour plus de détails). Diminuer ainsi la valeur de s permet de restreindre la quantité de mémoire requise par BTD, mais cet économie s'effectue au détriment de la taille du plus grand cluster qui peut alors augmenter. Aussi, en pratique, nous avons observé que s est généralement significativement plus petit que $w + 1$. Par conséquent, la quantité de mémoire supplémentaire requise peut être, dans certains cas, trop importante pour mettre en œuvre une telle variante de BTD.

Enfin, considérons le cas où il n'y a qu'un seul cluster. Nous avons trivialement $k = n$ et donc la production d'une solution optimale requiert une énumération sur n variables. Afin d'éviter une telle redondance dans la recherche, nous pouvons doter BTD de la capacité d'enregistrer la meilleure instanciation connue sur le cluster racine. Ce compromis modifie légèrement la complexité en espace ($O(n + n.s.d^s)$ au lieu de $O(n.s.d^s)$) tout en économisant du temps de calcul.

5 Résultats expérimentaux

La seconde question soulevée par la méthode BTD concerne son efficacité pratique. Notre intuition ainsi que les résultats théoriques de BTD laissent à penser que BTD devrait se révéler efficace en pratique pour résoudre certaines instances (en particulier si

elles possèdent une structure adéquate) mais aucun résultat expérimental n'a été présenté dans [TJ03]. Notre intuition est fondée, d'abord, sur la complexité en temps de BTD qui est meilleure que celle des méthodes énumératives puisque $w + 1 \leq n$. Ensuite, grâce à la mémorisation et à l'utilisation de goods, il s'avère que BTD résout chaque sous-problème une et une seule fois, ce qui permet d'économiser du temps et des tests de contraintes. En revanche, BTD emploie des minorants et majorants locaux, qui peuvent donc avoir un pouvoir de coupe limité. Par conséquent, des expérimentations sont nécessaires pour estimer l'intérêt pratique réel de BTD.

Cette section fournit des résultats expérimentaux portant sur des instances aléatoires et des instances issues du monde réel. Dans les deux cas, les expériences sont réalisées sur un PC sous linux doté d'un processeur Intel Pentium IV cadencé à 2,4 GHz et de 512 Mo de mémoire vive. Pour les instances aléatoires, le temps de résolution est limité à une demi-heure. Aussi, parfois, certaines instances peuvent ne pas être résolues. Pour chaque classe d'instances aléatoires, nous résolvons cinquante problèmes. Les résultats présentés sont alors les moyennes des résultats obtenus pour chaque instance. Pour les problèmes aléatoires comme pour les problèmes issus du monde réel, une décomposition arborescente est calculée en triangulant le graphe de contraintes (grâce à l'algorithme proposé dans [RTL76]) et en recherchant les cliques maximales du graphe de contraintes triangulé. A partir de cette décomposition arborescente, nous produisons une décomposition dont le paramètre s ne dépasse pas 5 pour les instances aléatoires et 10 pour les instances issues du monde réel (voir [JT03] pour plus de détails sur le calcul d'une décomposition arborescente pour BTD), ce qui limite la quantité de mémoire requise par BTD. Notons que le temps de calcul d'une décomposition arborescente est négligeable rapporté au temps de résolution du problème. Enfin, pour des raisons évidentes d'efficacité, BTD est basé sur FC (au lieu de BB). On notera donc cette version FC-BTD. L'emploi d'un filtrage de type Forward-Checking dans BTD n'altère en rien les résultats présentés précédemment. A l'intérieur d'un cluster, FC-BTD utilise l'heuristique *dom/deg* pour ordonner dynamiquement les variables. Cette heuristique consiste à choisir d'abord la variable x qui minimise le rapport $|d_x|/|\Gamma_x|$ avec Γ_x l'ensemble des variables qui partagent une contrainte avec x et $|d_x|$ le nombre de valeurs du domaine d_x qui n'ont pas été supprimées par filtrage. Aucune heuristique particulière n'est utilisée pour ordonner les valeurs.

5.1 Expérimentations sur des instances aléatoires

Nous avons d'abord étudié le comportement de FC-BTD sur des instances aléatoires classiques. Dans le cadre des CSP classiques, FC-BTD résout les instances aléatoires classiques aussi efficacement que les meilleures méthodes énumératives [JT03], bien que ces instances ne possèdent pas *a priori* de bonnes propriétés structurelles. Malheureusement, dans le cadre des Max-CSP, il s'avère que dans plusieurs cas, la méthode FC-BTD est moins performante que des algorithmes comme FC ou FC-MRDAC. En effet, comme ces instances ne possèdent pas de bonnes propriétés structurelles, les clusters ainsi obtenus correspondent souvent à des problèmes sous-contraints et donc FC-BTD perd beaucoup de temps à énumérer toutes les solutions possibles (car FC-BTD exploite des minorants et majorants locaux).

Ensuite, nous avons poursuivi notre étude avec des instances aléatoires structurées pour lesquelles nous pouvons penser que FC-BTD sera meilleur que les algorithmes classiques grâce à l'exploitation de la structure. Pour ces instances, nous employons le modèle

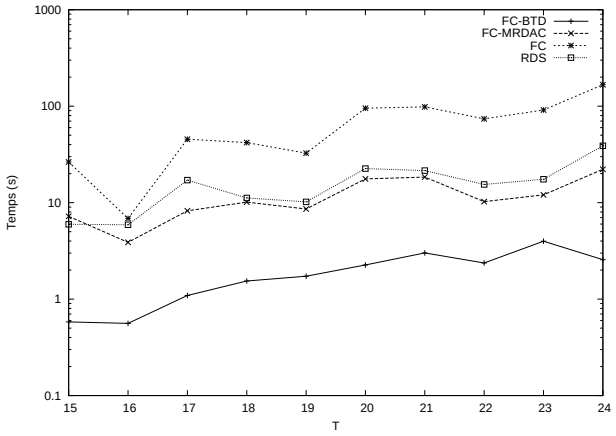

FIG. 3 – Temps d'exécution moyen en secondes (avec une échelle logarithmique) pour FC-BTD, FC-MRDAC, FC et RDS pour la classe (30,5,10,T,5).

d'instances aléatoires structurées proposé dans [JT03]. Nous considérons plusieurs classes $(n, d, r_{max}, T, s_{max})$. Une instance de la classe $(n, d, r_{max}, T, s_{max})$ possède n variables (chacune ayant un domaine de taille d). Son graphe de contraintes forme un arbre de cliques tel que la taille de la plus grande clique est r_{max} et la taille de la plus grande intersection est au plus s_{max}. T désigne la dureté de chaque contrainte. Nous comparons FC-BTD avec trois algorithmes à savoir FC, RDS [VLS96] et FC-MRDAC [LMS99]. Pour FC, nous employons l'heuristique dynamique dom/deg. Pour RDS, l'ordre statique sur les variables est construit initialement en exploitant l'heuristique dom/deg. FC et RDS n'emploient aucune heuristique particulière pour ordonner les valeurs. Pour FC-MRDAC, nous utilisons l'implémentation proposée par J. Larrosa [Lar].

Nous observons d'abord que, pour chaque classe (30,5,10,T,5) présentée à la figure 3, FC-BTD apparaît significativement meilleur que les trois autres méthodes quelle que soit la dureté. Par exemple, FC-BTD résout ces instances de 7 à 14 fois plus rapidement que FC-MRDAC. Ce gain est obtenu grâce à l'exploitation des goods valués structurels. Les goods permettent, en effet, à FC-BTD d'éviter d'explorer certaines parties redondantes de l'espace de recherche. Aussi, FC-BTD effectue moins de tests de contraintes que les autres méthodes testées. Seulement quelques centaines de goods sont mémorisés, mais les goods sont exploités à de nombreuses reprises (en moyenne entre 1 000 et 8 000 fois par good).

Afin de confirmer ces résultats, nous avons comparé FC-BTD, FC-MRDAC, FC et RDS sur quatre autres classes d'instances aléatoires structurées (voir le tableau 1). Nous avons alors constaté que, pour certaines classes, FC-MRDAC ou RDS étaient incapables de résoudre toutes les instances alors que FC-BTD parvient à résoudre chaque instance en seulement quelques secondes. Outre les gains en temps, FC-BTD accomplit aussi significativement moins de tests de contraintes que les trois autres algorithmes, comme le montre le tableau 2 (dans ce tableau, nous ne tenons compte que du nombre de tests pour les instances qui sont résolues par tous les algorithmes). FC-BTD se révèle donc nettement plus efficace que FC-MRDAC, FC et RDS sur ces instances. Il tire pleinement

Classe	FC-BTD	FC-MRDAC	FC	RDS
(30,10,10,78,5)	18,9	280,6 (1)	124,2	154,9 (1)
(40,5,10,15,5)	2,7	144,6 (1)	149,3	152,8 (0)
(40,10,10,55,5)	7,6	318,5 (3)	77,6	503,1 (8)
(40,5,15,9,5)	15,5	160,3 (1)	64,2	109,8 (0)

TAB. 1 – Temps d'exécution moyen en secondes (le nombre entre parenthèses indique le nombre d'instances non résolues) pour FC-BTD, FC-MRDAC, FC et RDS pour quatre classes d'instances aléatoires structurées.

Classe	FC-BTD	FC-MRDAC	FC	RDS
(30,10,10,78,5)	59,4	512,7	495,3	756,2
(40,5,10,15,5)	6,3	137,4	535,6	781,5
(40,10,10,55,5)	40,7	296,3	307,8	871,2
(40,5,15,9,5)	42,1	233,7	360,3	551,1

TAB. 2 – Nombre moyen de tests de contraintes en millions pour FC-BTD, FC-MRDAC, FC et RDS en ne considérant que les instances résolues par tous les algorithmes.

parti de la structure de ces instances. En effet, d'après le tableau 3, comme pour la première classe, seulement quelques goods sont enregistrés mais leur utilisation par FC-BTD lui permet d'élaguer de nombreuses branches et ainsi d'économiser une multitude de tests de contraintes. Autrement dit, FC-BTD réalise un bon compromis entre le temps et l'espace puisque les enregistrements de goods ne requièrent au final que peu de mémoire.

Enfin, nous avons observé (voir la dernière colonne du tableau 3) que la méthode proposée à la section 4 pour calculer une solution optimale nécessite au plus un millier de tests de contraintes supplémentaires, ce qui est négligeable comparé aux millions de tests accomplis pour trouver le coût optimal.

Classe	Nb. goods mémorisés	Nb. utilisations goods (en milliers)	Nb. tests goods (en milliers)	Nb. tests suppl.
(30,10,10,78,5)	5 776	11 906	33 227	698
(40,5,10,15,5)	1 084	3 014	3 976	489
(40,10,10,55,5)	9 362	3 796	11 000	968
(40,5,15,9,5)	519	9 577	10 733	581

TAB. 3 – Nombre moyen total de goods mémorisés, d'exploitations de goods et de tests de goods pour calculer le coût optimal et nombre moyen de tests de contraintes nécessaires pour calculer une solution optimale à partir du coût optimal.

5.2 Expérimentations sur des instances issues du monde réel

A présent, nous étudions le comportement de FC-BTD sur certaines instances réelles issues de l'archive FullRLFAP[1]. Ces instances correspondent à des problèmes d'allocation de fréquences (pour plus de détails, voir [CdGL+99]). Certaines d'entre elles peuvent facilement s'exprimer sous la forme d'instances Max-CSP binaires pondérés. En particulier, nous focalisons notre étude sur la classe SUBCELAR qui contient cinq sous-problèmes produits à partir de l'instance SCEN-06 (une des plus difficiles). Nous exploitons la simplification proposé par T. Schiex [Sch]. Elle consiste à supprimer toutes les contraintes dures d'égalité et à diviser par deux le nombre de variables. Ainsi, nous obtenons des instances de taille plus réduite et donc une meilleure décomposition arborescente. Par exemple, l'instance la plus petite ($SUBCELAR_0$) possède 16 variables et 57 contraintes tandis que la plus grande ($SUBCELAR_4$) a 22 variables et 131 contraintes. Les domaines contiennent 36 ou 44 valeurs.

Comme le montre le tableau 4, FC-BTD parvient à résoudre toutes les instances de la classe SUBCELAR. Ces résultats sont essentiellement dus à l'exploitation des goods. En effet, en moyenne, FC-BTD exploite chaque good entre 9 et 261 fois, ce qui permet d'éviter de nombreuses redondances dans la recherche. Pour information, pour chaque instance, le calcul d'une solution optimale à partir du coût optimal requiert au plus 3 270 tests de contraintes, ce qui est une nouvelle fois négligeable par rapport aux millions de tests requis pour calculer le coût optimal.

Instance	Temps (s)	Nb. goods	Nb. utilisations goods (en milliers)	Nb. tests goods (en millions)
$SUBCELAR_0$	2,5	34 170	306	1,57
$SUBCELAR_1$	308	80 375	1 336	6,48
$SUBCELAR_2$	405	96 980	996	15,64
$SUBCELAR_3$	1 883	515 735	19 661	162,70
$SUBCELAR_4$	122 933	403 282	105 386	844,44

TAB. 4 – Temps de résolution en secondes, nombre total de goods mémorisés, d'exploitations de goods, et de tests de goods pour FC-BTD pour la résolution des instances SUBCELAR sans majorant initial.

Un des avantages d'employer des instances réelles comme celles du CELAR est de pouvoir comparer nos résultats à ceux précédemment obtenus (par exemple [LMS99, Kos99, LMS02, MSV02, LD03]). Hélas, en pratique, cette comparaison est assez difficile à réaliser. D'une part les ordinateurs utilisés ont souvent des architectures complètement différentes. Par exemple, dans [LMS02], la méthode Specialized RDS (une variante de RDS) est expérimentée sur un Sun Ultra 60. Les architectures matérielles étant totalement différentes, nous ne pouvons pas comparer directement les temps des deux approches. Cependant, à titre indicatif, nous rappelons les résultats obtenus par cette méthode dans le tableau 5. D'autre part, les protocoles expérimentaux diffèrent. Par exemple, dans [LMS99], FC-MRDAC résout les instances SUBCELAR en utilisant comme majorant initial le coût optimal tandis qu'à la base, FC-BTD n'exploite aucun majorant initial. A titre indicatif, le tableau 6 présente les résultats obtenus par FC-BTD si on

[1]Nous remercions le Centre d'Electronique de l'Armement (France).

Instance	FC-MRDAC	SRDS	OSRDS
$SUBCELAR_0$	-	89	9,91
$SUBCELAR_1$	2 600	711	92
$SUBCELAR_2$	23 000	1 573	168
$SUBCELAR_3$	68 000	34 671	4 833
$SUBCELAR_4$	260 000	206 314	16 860

TAB. 5 – Temps en secondes obtenus par FC-MRDAC sur un Sun Sparc 2 [LMS99] (avec comme majorant initial le coût optimal de l'instance à résoudre), par Specialized RDS sur un Sun Ultra 60 [LMS02] et par Opportunistic Specialized RDS sur un Pentium IV 1,8 GHz [MSV02].

Instance	Temps (s)	Nb. goods	Nb. utilisations goods (en milliers)	Nb. tests goods (en millions)
$SUBCELAR_0$	1,1	16 056	102,2	0,58
$SUBCELAR_1$	154	4 344	24,0	0,07
$SUBCELAR_2$	195	1 728	18,3	0,06
$SUBCELAR_3$	1 092	67 004	487,2	5,33
$SUBCELAR_4$	58 834	55 452	6 200,5	17,76

TAB. 6 – Temps de résolution en secondes, nombre total de goods mémorisés, d'exploitations de goods, et de tests de goods pour FC-BTD pour la résolution des instances SUBCELAR avec comme majorant initial le coût de la meilleure solution.

utilise comme majorant initial le coût de la solution optimale. Les temps de résolution de FC-MRDAC sont eux rappelés dans le tableau 5. Pour comparaison, FC-MRDAC résout, par exemple, l'instance $SUBCELAR_2$ en environ 23 000 secondes sur un Sun Sparc 2 contre 195 secondes pour FC-BTD sur un PC équipé d'un Pentium IV 2,4 GHz. Toutefois, quand les architectures sont similaires, la comparaison peut être effectuée facilement. Par exemple, [MSV02] présente les résultats expérimentaux obtenus par une version améliorée de SRDS (appelée Opportunistic Specialized RDS et notée OSRDS) sur un Pentium IV 1,8 GHz. La dernière colonne du tableau 5 rappelle ces résultats. La comparaison est favorable tantôt à OSRDS, tantôt à FC-BTD. Elle montre que, grâce à une exploitation plus fine de la structure, FC-BTD peut rivaliser, dans certains cas, avec des méthodes possédant, comme OSRDS, un minorant de meilleure qualité que le sien.

A l'image de FC-BTD, [LD03] exploite la structure du problème pour fournir des bornes de complexité en temps et en espace, mais les résultats expérimentaux présentés sont loin d'être convaincants. En effet, ces résultats portent uniquement sur une sous-instance de l'instance $SUBCELAR_1$ (cette sous-instance étant obtenue en réduisant tous les domaines à leurs dix premières valeurs). Les temps de calcul présentés sont d'au moins 27 000 secondes alors que FC-BTD requiert seulement 675 secondes pour résoudre cette sous-instance. Ce temps de résolution supérieur au temps requis pour résoudre l'instance $SUBCELAR_1$ s'explique par la nature même des contraintes.

La comparaison entre FC-BTD et la méthode de Koster [Kos99], la meilleure méthode connue pour résoudre les problèmes du CELAR, est loin d'être aisée. En effet, cette

méthode, qui exploite comme BTD la notion de décomposition arborescente pour fournir une borne de complexité en temps, emploie plusieurs prétraitements afin de simplifier le problème. En particulier, un de ces prétraitements réduit la taille du graphe de contraintes, ce qui permet d'employer une meilleure décomposition arborescente et donc d'obtenir de meilleurs paramètres structurels. Notre méthode n'utilisant à l'heure actuelle aucun de ces prétraitements, les résultats obtenus par les deux méthodes sont difficilement comparables. De plus, nous n'avons pas encore étudié l'influence de certains paramètres structurels (comme w, s ou la forme de l'arborescence de clusters) sur le comportement de FC-BTD. Aussi, en ajoutant des prétraitements comme ceux de Koster et en effectuant de meilleurs choix pour certains paramètres de FC-BTD, nous pouvons espérer améliorer encore l'efficacité pratique de FC-BTD. En fait, ces améliorations se révèlent même nécessaires pour résoudre des instances plus grandes et plus difficiles comme par exemple l'instance SCEN-06.

6 Conclusion et perspectives

Dans ce papier, nous avons soulevé deux questions concernant la méthode BTD. La première porte sur la construction d'une solution optimale à partir du coût optimal calculé par BTD. Nous avons alors proposé une méthode efficace pour calculer une solution optimale. La seconde question s'intéresse à l'efficacité pratique de BTD. Nous avons étudié expérimentalement la méthode BTD et montré son intérêt pratique pour la résolution d'instances possédant une bonne structure. En effet, la méthode BTD se révèle significativement plus efficace que FC-MRDAC, FC et RDS pour résoudre des instances aléatoires structurés, et, de plus, elle parvient à résoudre toutes les instances SUBCELAR.

Concernant les perspectives, BTD peut être amélioré par la prise en compte des contraintes entre des variables non instanciées (par exemple, en employant la notion de consistance d'arc [LMS99, CS04]). Ensuite, l'étude de l'influence de certains paramètres structurels sur le comportement de BTD peut nous aider à optimiser certains choix pour ces paramètres, ce qui permettrait à BTD d'obtenir des résultats encore meilleurs. Enfin, nous pouvons ajouter à BTD des prétraitements comme ceux de Koster [Kos99].

Références

[ACP87] S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal of Discrete Mathematics*, 8 :277–284, 1987.

[BMR95] S. Bistarelli, U. Montanari, and F. Rossi. Constraint solving over semirings. In *Proceedings of the fourteenth International Joint Conference on Artificial Intelligence*, pages 624–630, 1995.

[CdGL+99] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio Link Frequency Assignment. *Constraints*, 4 :79–89, 1999.

[CS04] M. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154 :199–227, 2004.

[DF01] R. Dechter and Y. El Fattah. Topological Parameters for Time-Space Tradeoff. *Artificial Intelligence*, 125 :93–118, 2001.

[FW92] E. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58 :21–70, 1992.

[JT03] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146 :43–75, 2003.

[Kos99] A. Koster. *Frequency Assignment - Models and Algorithms*. PhD thesis, University of Maastricht, November 1999.

[Lar] J. Larrosa. http ://www.lsi.upc.es/ larrosa/pfc-mrdac.

[Lar02] J. Larrosa. On arc and node consistency. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 48–53, 2002.

[LD03] J. Larrosa and R. Dechter. Boosting Search with Variable Elimination in Constraint Optimization and Constraint Satisfaction Problems. *Constraints*, 8(3) :303–326, 2003.

[LMS99] J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence*, 107(1) :149–163, 1999.

[LMS02] J. Larrosa, P. Meseguer, and M. Sánchez. Pseudo-Tree Search with Soft Constraints. In *Proceedings of the 15th European Conference on Artificial Intelligence*, pages 131–135, 2002.

[LS03] J. Larrosa and T. Schiex. In the quest of the best form of local consistency for Weighted CSP. In *Proceedings of the eighteenth International Joint Conference on Artificial Intelligence*, pages 239–244, 2003.

[MS00] P. Meseguer and M. Sánchez. Tree-based Russian Doll Search. In *Proceedings of Workshop on soft constraint*. CP'2000, 2000.

[MS01] P. Meseguer and M. Sánchez. Specializing Russian Doll Search. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, volume LNCS 2239, pages 464–478, 2001.

[MSV02] P. Meseguer, M. Sánchez, and G. Verfaillie. Opportunistic Specialization in Russian Doll Search. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, volume LNCS 2470, pages 264–279, 2002.

[RS86] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of tree-width. *Algorithms*, 7 :309–322, 1986.

[RTL76] D. Rose, R. Tarjan, and G. Lueker. Algorithmic Aspects of Vertex Elimination on Graphs. *SIAM Journal on computing*, 5 :266–283, 1976.

[Sch] T. Schiex. http ://www.inra.fr/bia/t/schiex/doc/celar.html.

[SFV95] T. Schiex, H. Fargier, and G. Verfaillie. Valued Constraint Satisfaction Problems : hard and easy problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 631–637, 1995.

[TJ03] C. Terrioux and P. Jégou. Bounded backtracking for the valued constraint satisfaction problems. In *Proceedings of the ninth International Conference on Principles and Practice of Constraint Programming*, pages 709–723, 2003.

[VLS96] G. Verfaillie, M. Lemaître, and T. Schiex. Russian Doll Search for Solving Constraint Optimization Problems. In *Proceedings of the thirteenth National Conference on Artificial Intelligence*, pages 181–187, 1996.