Compilation de réseaux de contraintes en graphes de décision décomposables multivalués

Frédéric Koriche Jean-Marie Lagniez Pierre Marquis Samuel Thomas
CRIL-CNRS, Université d'Artois, Lens, France
{koriche,lagniez,marquis,thomas}@cril.univ-artois.fr

Résumé

Dans cet article, nous présentons et évaluons un algorithme de compilation de réseaux de contraintes (CNs) en graphes de décision décomposables multivalués (MDDG). Bien que le langage MDDG contienne le langage Decision-DNNF comme sous-ensemble propre, il offre les mêmes requêtes et les mêmes transformations réalisables en temps polynomial, ce qui en fait un langage intéressant pour beaucoup d'applications. Notre large panel d'expérimentations a montré que le compilateur cn2mddg que nous avons développé parvient à compiler des CNs qui se trouvent généralement hors de portée des approches standard basées sur la transformation du réseau en CNF, compilée ensuite en Decision-DNNF. De plus, la taille de la forme compilée obtenue se révèle plus concise, parfois de plusieurs ordres de grandeur.

Abstract

We present and evaluate a top-down algorithm for compiling finite-domain constraint networks (CNs) into the language MDDG of multivalued decomposable decision graphs. Though it includes Decision-DNNF as a proper subset, MDDGoffers the same key tractable queries and transformations as Decision-DNNF, which makes it useful for many applications. Intensive experiments showed that our compiler cn2mddg succeeds in compiling CNs which are out of the reach of standard approaches based on a translation of the input network to CNF, followed by a compilation to Decision-DNNF. Furthermore, the sizes of the resulting compiled representations turn out to be much smaller (sometimes by several orders of magnitude).

1 Introduction

La programmation par contraintes (CP) est depuis longtemps reconnue comme un paradigme de choix pour la représentation et la résolution de problèmes combinatoires [22]. Le problème est représenté de manière compacte et intuitive en utilisant un réseau de contraintes (CN) ; celui-ci consiste en un ensemble de variables, chacune étant associée à un domaine de valeurs, et un ensemble de contraintes qui interconnectent les variables en spécifiant, d'une manière ou d'une autre, leurs n-uplets de valeurs autorisés. Malgré le succès indéniable de cette approche déclarative, un des défis majeurs de la programmation par contraintes est d'offrir des garanties de performance pour répondre aux requêtes posées par l'utilisateur, qui se résument souvent à résoudre des (instances de) problèmes NP-difficiles. Comme il est souligné dans [13], le critère de garantie de performance est crucial pour les applications en ligne, telles que les logiciels de configuration [17] et les systèmes de recommandation [6], où les requêtes posées « à la volée » par l'utilisateur doivent être résolues en temps réel.

Le but de cet article est de répondre à ce critère en utilisant la compilation de connaissances [8]. L'idée générale est de convertir un langage de contraintes vers un langage de compilation cible permettant de résoudre diverses tâches de calcul (que l'on classe souvent en requêtes et transformations) en temps polynomial. Bien que beaucoup de requêtes soient intraitables lorsque le problème est formulé en CN, elles deviennent traitables à partir d'une forme compilée de celui-ci, assurant ainsi une garantie de performance en ligne quand la forme compilée est de taille raisonnable.

Nous présentons un algorithme descendant cn2mddg pour la compilation de CNs, à domaines finis, en graphes de décision décomposables multivalués. cn2mddg reçoit en entrée un CN représenté sous le format XCSP 2.1 [23]. Notre algorithme de compilation retourne en sortie une représentation des solutions du CN dans le langage MDDG (graphes de décision décomposables multivalués). MDDG est l'extension vers les domaines non booléens du langage DDG [11], aussi connu sous le nom de Decision-DNNF [21]. Il est basé sur

des nœuds $\wedge$ décomposables et des nœuds de décision (multivalués). Comme le langage des Decision-DNNF, le langage des MDDG permet le traitement polynomial de plusieurs classes de requêtes, telles que la recherche d'une solution, le comptage de solutions (possiblement pondérées), l'énumération des solutions (avec un délai polynomial), et l'optimisation sous fonction objectif linéaire. Ce langage autorise aussi plusieurs transformations polynomiales, telles que le conditionnement, c'est-à-dire l'instanciation de variables, et plus généralement l'ajout de contraintes unaires.

cn2mddg exploite une technique de cache spécifique, ainsi qu'une nouvelle heuristique de choix de variable fondée sur une notion de centralité considérée en algorithmique des graphes (*betweenness centrality*), et détecte les contraintes universelles pendant la recherche afin d'effectuer des simplifications supplémentaires. cn2mddg utilise une approche *directe* : le CN donné en entrée est compilé directement en MDDG, sans passer par une formule intermédiaire. A l'opposé, les compilateurs standard comme c2d [7] ou Dsharp [20], nécessitent une approche *indirecte* pour prendre en charge les réseaux de contraintes : le CN donné est d'abord traduit en une CNF équivalente, qui est ensuite transformée par le compilateur en Decision-DNNF.

Nous avons expérimenté notre compilateur cn2mddg sur un grand nombre de jeux d'essai (173) provenant de différentes familles (15). A la lumière des résultats expérimentaux, il apparaît que les approches indirectes sont, dans la majorité des cas pratiques, inutilisables : quel que soit l'encodage utilisé, la traduction du CN en formule clausale (CNF) équivalente requiert un grand nombre de variables booléennes, et induit une perte de structure, inhérente à la représentation clausale (comparée à la représentation graphique d'un réseau de contraintes). En compilant directement le CN sans passer par une formule booléenne intermédiaire, notre compilateur cn2mddg est bien plus robuste puisqu'il réussit à compiler beaucoup de CNs qui se sont révélés hors de portée des approches indirectes, utilisant une traduction préliminaire en formule booléenne. De plus, les tailles des formes compilées obtenues sont plus concises, le gain de taille étant parfois réduit de plusieurs ordres de grandeur.

Dans la suite de l'article, nous commençons par introduire quelques notions de base, portant sur les réseaux de contraintes, puis nous présentons le langage MDDG. Nous décrivons ensuite le compilateur cn2mddg, et concluons en présentant quelques résultats obtenus lors de nos expérimentations. Notre compilateur, ainsi que les traducteurs, les jeux d'essai utilisés pour nos expérimentations et des résultats suplémentaires, peuvent être téléchargés à partir de l'adresse suivante : http://www.cril.fr/KC/.

2 Préliminaires

Rappelons qu'un réseau de contraintes (CN) (fini) est un triplet $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, consistant en un ensemble de variables $\mathcal{X} = \{X_1, \ldots, X_n\}$, un ensemble de domaines $\mathcal{D} = \{D_1, \ldots, D_n\}$, et un ensemble de contraintes $\mathcal{C} = \{C_1, \ldots, C_m\}$. Chaque domaine D_i est un ensemble fini contenant les valeurs possibles de la variable X_i. Chaque contrainte C_j pose une restriction sur les combinaisons de valeurs d'un sous-ensemble de variables. Formellement, $C_j = (S_j, R_j)$, où $S_j = \{X_{j1}, \ldots, X_{jk}\}$ est une partie de X, appelée *portée* (ou *scope*) de C_j, et R_j est un prédicat sur le produit cartésien de $D_{j1} \times \ldots \times D_{jk}$, appelé *relation* de C_j. R_j peut être représenté en extension par la liste des n-uplets autorisés (ou la liste des n-uplets interdits), ou en intension via un oracle, c'est-à-dire une fonction $D_{j1} \times \ldots \times D_{jk}$ vers $\{0, 1\}$ calculable en temps polynomial en la taille de l'entrée. L'*arité* d'une contrainte est donnée par le cardinal de sa portée. Les contraintes d'arité 2 sont dites *binaires* et celles d'arité supérieure à 2 sont dites n-aires.

Exemple 1 *Dans la suite, nous considérons le réseau* CN $\mathcal{N}$ *défini sur quatre variables* X_1, X_2, X_3 *et* X_4*, chacune étant associée au même domaine* $\{0, 1, 2\}$*, par les trois contraintes* C_1, C_2 *et* C_3*, caractérisées par les expressions suivantes :*

- $C_1 = (X_1 \neq X_2)$;
- $C_2 = (X_2 = 0) \vee (X_2 = 1) \vee (X_2 = X_3 + X_4 + 1)$;
- $C_3 = (X_3 > X_4)$.

Soit S un sous-ensemble de variables de $\mathcal{X}$. Nous appelons *état* (de décision) sur S, une fonction $\boldsymbol{s}$ qui associe à chaque variable X_i de S un sous-ensemble $\boldsymbol{s}(X_i)$ de valeurs dans D_i. Dans la suite, les états sont aussi notés comme l'union d'affectations élémentaires, c'est-à-dire des ensembles de la forme $\{\langle X_i, x_j \rangle\}$, où $x_j \in \boldsymbol{s}(X_i)$. *scope*$(\boldsymbol{s})$ représente l'ensemble S des variables pour lesquelles $\boldsymbol{s}$ est défini. Un état $\boldsymbol{s}$ est dit *partiel* si *scope*$(\boldsymbol{s})$ est un sous-ensemble strict de $\mathcal{X}$, sinon $\boldsymbol{s}$ est dit *complet*. Une variable X_i de *scope*$(\boldsymbol{s})$ est dite *instanciée* si $\boldsymbol{s}(X_i)$ est un singleton. L'ensemble des variables instanciées de $\boldsymbol{s}$ est noté *single*$(\boldsymbol{s})$. Un état $\boldsymbol{s}$ est dit instancié lorsque toutes ses variables sont instanciées, i.e. *scope*$(\boldsymbol{s})$ = *single*$(\boldsymbol{s})$.

Pour un état $\boldsymbol{s}$ et un sous-ensemble de variables $T \subseteq$ *scope*$(\boldsymbol{s})$, nous notons $\boldsymbol{s}[T]$ la restriction de $\boldsymbol{s}$ à T, autrement dit $\boldsymbol{s}[T]$ est l'ensemble $\{\langle X_i, x_j \rangle \in \boldsymbol{s} \mid X_i \in T\}$. Une instanciation $\boldsymbol{s}$ satisfait une contrainte $C_j = (S_j, R_j)$ si $S_j \subseteq$ *scope*$(\boldsymbol{s})$ et $R_j(x_{j1}, \ldots, x_{jk}) = 1$, pour tout $l \in 1, \ldots, k$, tel que $\langle X_{jl}, x_{jl} \rangle \in \boldsymbol{s}[S_j]$. Une *solution* d'un CN $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ est une instanciation complète $\boldsymbol{s}$ satisfaisant toutes les contraintes C_j de $\mathcal{C}$. Par exemple, $\boldsymbol{s} = \{\langle X_1, 1 \rangle, \langle X_2, 0 \rangle, \langle X_4, 1 \rangle, \langle X_4, 0 \rangle\}$ est une solution du CN de l'exemple 1.

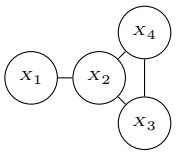


FIGURE 1 – Graphe primal du CN de l'exemple 1.

Soit un CN $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ et un état $\boldsymbol{s}$ défini sur une partie de $\mathcal{X}$. Le *conditionnement* de $\mathcal{N}$ par $\boldsymbol{s}$, noté $\mathcal{N} \mid \boldsymbol{s}$, est le CN $(\mathcal{X}', \mathcal{D}', \mathcal{C}')$ défini par les conditions suivantes : $\mathcal{X}' = \mathcal{X} \setminus \mathit{single}(\boldsymbol{s})$; à chaque domaine D_i de $\mathcal{D}$, on associe le domaine $D'_i \in \mathcal{D}'$, où $D'_i = D_i$ si $X_i \notin \mathit{scope}(\boldsymbol{s})$ et $D'_i = s(D_i)$ sinon ; enfin, à chaque contrainte $C_j = (S_j, R_j)$ de $\mathcal{C}$, on associe la contrainte $C'_j = (S'_j, R'_j)$, où $S'_j = S_j \setminus \mathit{single}(\boldsymbol{s})$ et R'_j est la restriction de R_j à S'_j.

Le graphe primal d'un CN $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ est le graphe non orienté G donné par l'ensemble des sommets $\mathcal{X}$ et l'ensemble des arêtes $\mathcal{E}$, tel que $\{X_p, X_q\} \in \mathcal{E}$ si et seulement si $\{X_p, X_q\}$ est un sous-ensemble de la portée S_j d'une contrainte C_j de $\mathcal{C}$. Par exemple, le graphe primal de l'exemple 1 est décrit dans la figure 1.

3 Le langage MDDG

Dans un premier temps, nous présentons le langage des graphes de décision multivalués (MDG) :

Définition 1 *(MDG) Soit un ensemble fini $\mathcal{X}$ de variables à domaine fini. Le langage MDG des graphes de décision multivalués (à lecture unique) sur $\mathcal{X}$ est l'ensemble des graphes Δ orientés sans circuit à racine unique, tels que chaque feuille est étiquetée par $\top$ (vrai) ou $\bot$ (faux), et chaque nœud interne N est soit un nœud $\wedge$, soit $N = \wedge(N_1, \ldots, N_i)$, soit un nœud de décision N associé à une variable $X_i \in \mathcal{X}$, autrement dit un nœud $\vee$ déterministe $N = \vee(N_1, \ldots, N_j)$ tel que $D_i = \{x_{i1}, \ldots, x_{ij}\}$ et l'arc entre N et $N_k (k \in 1, \ldots, j)$ est étiqueté par l'affectation élémentaire $\{\langle X_i, x_{ik}\rangle\}$. Les chemins dans Δ doivent satisfaire la propriété de « lecture unique » (*read-once*) : pour tout chemin depuis la racine de Δ jusqu'à une feuille, et pour toute variable $X_i \in \mathcal{X}$, au maximum un seul arc peut être étiqueté par une affectation élémentaire portant sur X_i.*

Pour chaque nœud N d'un graphe Δ de MDG, l'ensemble des variables de N, noté $\mathit{Var}(N)$, est inductivement défini par les conditions suivantes :

- si N est une feuille, alors $\mathit{Var}(N) = \varnothing$;
- si $N = \wedge(N_1, \ldots, N_i)$ est un nœud $\wedge$, alors $\mathit{Var}(N) = \bigcup_{k=1}^{i} \mathit{Var}(N_k)$;
- si N est un nœud de décision $N = \vee(N_1, \ldots, N_j)$ associé à une variable X, alors $\mathit{Var}(N) = \{X\} \cup \bigcup_{k=1}^{j} \mathit{Var}(N_k)$.

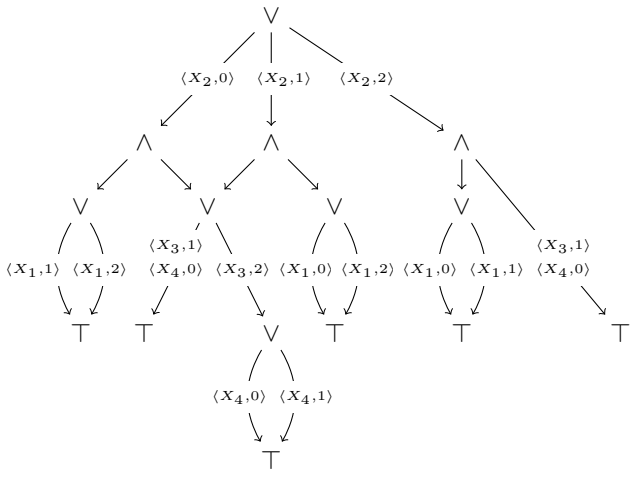


FIGURE 2 – Un MDDG équivalent au CN de l'exemple 1.

Soit $\boldsymbol{s}$ une instanciation complète sur $\mathcal{X}$ et soit Δ un graphe de MDG sur $\mathcal{X}$, ayant pour racine N. Le graphe $\mathit{eval}(N, \boldsymbol{s})$ est défini inductivement par les conditions suivantes :

- si N est une feuille, alors $\mathit{eval}(N, \boldsymbol{s}) = N$;
- si $N = \wedge(N_1, \ldots, N_i)$ est un nœud $\wedge$, alors $\mathit{eval}(N, \boldsymbol{s}) = \wedge(\mathit{eval}(N_1, \boldsymbol{s}), \ldots, \mathit{eval}(N_i, \boldsymbol{s}))$;
- si N est un nœud de décision $N = \vee(N_1, \ldots, N_j)$ associé à la variable X, alors $\mathit{eval}(N, \boldsymbol{s}) = \mathit{eval}(N_k, \boldsymbol{s})$ avec $\langle X, x_k\rangle \in \boldsymbol{s}$.

$\boldsymbol{s}$ est une *solution* de Δ si et seulement si le graphe de MDG sans noeud de décision $\mathit{eval}(N, \boldsymbol{s})$ est évalué à vrai.

Le langage MDDG examiné dans cette étude est un sous-ensemble des MDG contenant des nœuds *décomposables*, c'est-à-dire des nœuds $\wedge$ dont les fils ne partagent aucune variable en commun.

Définition 2 *(MDDG) Soit un ensemble fini $\mathcal{X}$ de variables à domaine fini, le langage MDDG sur $\mathcal{X}$ est le sous-ensemble des représentations en MDG Δ où tout nœud $N = \wedge(N_1, \ldots, N_i)$ de type $\wedge$ est décomposable, c'est-à-dire $\mathit{Var}(N_k) \cap \mathit{Var}(N_l) = \varnothing$ pour tous $k, l \in \{1, \ldots, i\}$, tels que $k \neq l$.*

A titre d'illustration, le graphe MDDG décrit dans la figure 2 est équivalent au CN donné dans l'exemple 1 : les deux représentations ont le même ensemble de solutions. Pour des raisons de lisibilité, les arcs conduisant à une feuille $\bot$ ne sont pas représentés sur la figure et la feuille $\top$ est dupliquée. De plus, les nœuds avec un seul fils sont supprimés et leurs étiquettes (dans le cas

des nœuds de décision) sont regroupées avec celles de leur arc incident. Les affectation élémentaires correspondantes sont typiquement obtenues par propagation (dans ce cas, les nœuds $\vee$ ne sont pas explicitement construits, nous les mentionnons dans la définition du langage MDDG dans un souci de simplicité).

Le langage des Decision-DNNF [21, 11] est un sous-ensemble du langage des MDDG où chaque variable possède un domaine booléen. En plus du gain d'expressivité obtenu en autorisant des domaines non booléens, les MDDG peuvent essentiellement traiter les mêmes requêtes et les mêmes transformations que celles des Decision-DNNF (comptage de solutions pondéré, énumération de solutions, optimisation pour une fonction objectif linéaire et conditionnement). Les algorithmes en temps polynomial utilisés pour résoudre ces requêtes et transformations dans les Decision-DNNF peuvent être étendus aux MDDG, de façon triviale.

MDDG possède aussi des similarités avec le langage MDD considéré dans [2], ainsi qu'avec le langage AOMDD considéré dans [18, 19], mais ne coïncide avec aucun d'entre eux. Ainsi, MDD et MDDG ne sont pas comparables en terme d'inclusion. D'un côté, une représentation en MDD est une structure non déterministe (plus d'un arc sortant d'un nœud de décision étiqueté par une variable X_i peut être étiqueté par une même valeur du domaine D_i de X_i), alors que les nœuds de décision des MDDG sont toujours déterministes. D'un autre côté, les représentations en MDDG autorisent des nœuds de type $\wedge$, alors que les nœuds internes des MDD sont tous des nœuds de décision. De la même manière, AOMDD et MDDG ne sont pas comparables du point de vue de l'inclusion. AOMDD est adapté à la compilation de modèles graphiques (en particulier, des réseaux bayésiens), et permet la représentation de fonctions non booléennes (comme des fonctions d'utilité ou de coût, des distributions de probabilités discrètes, etc.), alors que le langage MDDG ne le permet pas. De plus, les AOMDD sont des structures ordonnées qui respectent un pseudo-arbre induit par un ordre prédéfini d'élimination des variables - c'est un prérequis indispensable pour assurer la canonicité. A l'opposé, les MDDG n'imposent pas une telle condition (les représentations en MDDG ne sont pas canoniques).

4 Un compilateur MDDG de type descendant

Nous avons développé un compilateur cn2mddg de type descendant qui prend en entrée un CN représenté sous le format XCSP 2.1 [23], et retourne une représentation MDDG équivalente à celui-ci, c'est-à-dire ayant les mêmes solutions. Toutes les caractéristiques basiques offertes par le format XCSP 2.1 sont prises en compte. En particulier, les contraintes peuvent être représentées en intension (en tant que « prédicats ») ou en extension (en tant que « relations »). Cependant, seules trois contraintes globales sont supportées par notre compilateur dans sa forme actuelle : *allDifferent*, *weightedSum* (c'est-à-dire les contraintes linéaires), et *element*.[1]

L'architecture de notre compilateur est similaire à celle des compilateurs de type descendant pour les domaines booléens, comme c2d (http://reasoning.cs.ucla.edu/c2d/), ou Dsharp (http://www.haz.ca/research/dsharp/), tout deux ayant pour cible le langage Decision-DNNF. Notre compilateur est fondé sur un algorithme de recherche de solutions, suivant la trace d'un *solver* [15]. Il utilise des techniques similaires à celles proposées dans c2d et dans Dsharp, dont l'analyse de conflits pour guider la recherche, la propagation de contraintes afin de simplifier la formule, la mise en cache (*caching*) de composantes afin d'éviter la duplication de parties identiques pour la forme compilée, et une heuristique de choix de variable dynamique (comme Dsharp qui tire parti de l'heuristique de choix de variable *vsads*).[2] La méthode de *caching* et l'heuristique de choix de variable utilisées dans cn2mddg sont spécifiques à la nature du problème donné en entrée (un CN), qui est plus structuré que le format « plat » des CNF. De plus, notre algorithme exploite une méthode spécifique pour gérer les contraintes universelles, permettant d'effectuer plus de simplifications lors de la compilation.

Les contraintes universelles. Les contraintes universelles sont des contraintes qui sont nécessairement satisfaites, quelles que soient les affectations des variables apparaîssant dans leur portée (suivant le domaine courant des variables). Par exemple, avec le CN donné dans l'exemple 1, quand la contrainte C_2 est conditionnée par une affectation élémentaire $\{\langle X_2, 0\rangle\}$ ou $\{\langle X_2, 1\rangle\}$, C_2 devient universelle. A chaque étape de la compilation, les contraintes universelles sont détectées. L'universalité de chaque contrainte $C_j = (S_j, R_j) \in \mathcal{C}$ pour laquelle il existe au moins un $X_i \in s_j$ tel que D_i a été réduit par propagation à la suite de la dernière instanciation élémentaire est testée. Nous cherchons une instanciation $\boldsymbol{s}$ des variables dans la portée de C_j parmi les valeurs de leur domaine courant tel que $\boldsymbol{s}$ ne satisfait pas R_j. C_j est valide si et seulement si nous ne pouvons pas trouver une telle instanciation $\boldsymbol{s}$. Pour des raisons d'efficacité, $\boldsymbol{s}$ est cherchée d'une façon paresseuse. Une fois une instancia-

1. Ces contraintes peuvent aussi être encodées comme des « relations », mais dans ce cas il ne sera pas possible de profiter des propagateurs dédiés.

2. Dans c2d l'heuristique de choix de variable est statique.

tion $\boldsymbol{s}$ trouvée, elle est sauvegardée, puis reconsidérée en priorité lorsque l'universalité de C_j est à nouveau testée. Une fois détectée, une contrainte universelle C_j est simplement supprimée du réseau courant. Cela simplifie évidemment la suite des traitements (pas besoin de prendre en compte ces contraintes), favorise la décomposition (dû à la suppression d'arcs dans le graphe primal du CN), et influence l'heuristique de choix de variable. Pour les compilateurs de type Decision-DNNF, la gestion des contraintes universelles revient simplement à ignorer toutes les clauses partageant un littéral avec l'interprétation partielle courante.

Le caching. La mise en cache (*caching*) est une technique cruciale pour les compilateurs calculant des représentations à base de graphes orientés sans circuit (DAG). Le but est d'éviter d'explorer le même sous-problème deux fois ou plus, et de dupliquer la partie compilée. En effet, du fait de l'interchangeabilitée (conditionnelle) des valeurs dans beaucoup de réseaux, il arrive souvent que deux états de décision distincts $\boldsymbol{s}_1$ et $\boldsymbol{s}_2$ considérés successivement pendant la recherche conduisent au même sous-problème, autrement dit $\mathcal{N} \mid \boldsymbol{s}_1$ et $\mathcal{N} \mid \boldsymbol{s}_2$ sont équivalents. Dans une telle situation, plutôt que de compiler les deux réseaux, il est plus judicieux de ne compiler que $\mathcal{N} \mid \boldsymbol{s}_1$, puis d'enregistrer dans le cache une clé de $\mathcal{N} \mid \boldsymbol{s}_1$ associée à la racine N du nœud de sa représentation en MDDG et détecter que $\mathcal{N} \mid \boldsymbol{s}_2$ est équivalent à $\mathcal{N} \mid \boldsymbol{s}_1$ en cherchant dans le cache à chaque étape de la compilation. Ainsi, au lieu de calculer le graphe correspondant à $\mathcal{N} \mid \boldsymbol{s}_2$, il suffit de créer un arc pointant vers N. Par exemple, pour le CN donné dans l'exemple 1, la composante sur $\{X_3, X_4\}$ obtenue par décomposition dynamique est la même pour les états $\{\langle X_2, 0\rangle\}$ et $\{\langle X_2, 1\rangle\}$; nous n'avons donc pas besoin de la dupliquer.

Cependant, tester l'équivalence de deux CNs est un problème coNP-complet en général, ce problème devant être considéré un nombre exponentiel de fois durant la compilation. Pour ces raisons, il est impossible de faire du *caching* en force brute, où l'ensemble des réseaux différents $\mathcal{N} \mid \boldsymbol{s}$ rencontrés lors de la recherche seraient considérés (cela demanderait un temps de compilation ingérable). De ce fait, deux réseaux $\mathcal{N} \mid \boldsymbol{s}_1$ et $\mathcal{N} \mid \boldsymbol{s}_2$ sont considérés comme équivalents lorsqu'ils sont identiques.

La principale difficulté du *caching* concerne la taille des entrées du cache, qui doivent rester aussi petites que possible. Pour notre mise en cache, nous enregistrons d'abord les domaines courants des variables du problème, c'est-à-dire $\boldsymbol{s}$ restreint à ses variables non affectées. Stocker l'ensemble des contraintes courantes serait bien trop gourmand en espace. Heureusement, c'est inutile dans la plupart des cas. En effet, toute contrainte $C_j = (S_j, R_j)$ telle que $S_j \cap \mathit{single}(\boldsymbol{s}) = \varnothing$ n'a pas besoin d'être sauvegardée (étant donné que qu'elle n'est pas affectée). De plus, aucune contrainte $C_j = (S_j, R_j)$ binaire du réseau donné en entrée n'a besoin d'être enregistrée à partir du moment où ce réseau est arc-cohérent. En effet, si aucune variable de $S_j = \{X_i, X_k\}$ n'a été instanciée, alors nous sommes dans le cas précédent. Si les deux variables X_i, X_k de S_j sont instanciées alors soit la contrainte est universelle, soit elle est incohérente et donc, il est inutile de la stocker dans aucun des deux cas. Enfin, si une seule variable X_i de S_j est instanciée (disons à la valeur x_i) alors la projection sur la variable restante (non instanciée) X_k de la restriction R_j à $X_i = x_i$, coïncide avec la restriction de $\boldsymbol{s}$ à $\{X_k\}$ quand le réseau courant est arc-cohérent. [3] De manière similaire, nous n'avons pas besoin d'enregistrer les contraintes *allDifferent* qui peuvent être considérées comme des conjonctions de contraintes binaires. Les contraintes restantes sont sauvegardées dans notre cache. Pour les contraintes représentées en intension, les variables instanciées dans $\boldsymbol{s}$ sont remplacées par leurs valeurs dans les prédicats, et on effectue une étape de simplification dans le but de réduire les représentations des contraintes. Les contraintes représentées en extension sont stockées explicitement. Finalement, pour chaque contrainte *weightedSum*, les variables instanciées dans $\boldsymbol{s}$ sont remplacées par leurs valeurs, les contraintes sont simplifiées et seul le terme constant en résultant a besoin d'être sauvegardé. Les contraintes de type *element* qui sont binaires n'ont pas besoin d'être sauvegardées ; pour celles qui sont n-aires, on enregistre en cache $\{\langle X_i, x_i\rangle \in \boldsymbol{s} \mid X_i \in S_j\}$.

L'heuristique de choix de variable. Notre heuristique de choix de variable *bc* est basée sur une notion de centralité (*betweenness centrality*) utilisée en algorithmique des graphes [5]. Étant donné un nœud X_i dans un graphe (dans notre cas, le graphe primal du CN courant dans lequel les nœuds sont identifiés par les variables qu'ils étiquètent), $bc(X_i)$ est donné par le nombre de plus courts chemins entre toute paire de nœuds passant par X_i. Formellement,

$$bc(X_i) = \Sigma_{X_j \neq X_i \neq X_k} \frac{\sigma_{X_i}(X_j, X_k)}{\sigma(X_j, X_k)}$$

où X_i, X_j et X_k sont des nœuds du réseau, $\sigma(X_j, X_k)$ est le nombre de plus courts chemins de X_j à X_k et $\sigma_{X_i}(X_j, X_k)$ est le nombre de ces chemins passant par

3. Cela rappelle le traitement des clauses binaires dans Dsharp, qui n'ont pas besoin de figurer dans le cache du moment où la propagation unitaire a été effectuée [20].

Algorithm 1: cn2mddg($\mathcal{N}$)

entrée: un réseau de contraintes $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, et un état de décision $\mathtt{s}$ sur une partie de $\mathcal{X}$
sortie : la racine N d'une représentation MDDG équivalente à $\mathcal{N} \mid \mathtt{s}$

```
1  s ← ac(N, s)
2  N ← N | s
3  if unsat(N) then return leaf(⊥)
4  if #(X) = 0 then return leaf(⊤)
5  if cache(N) ≠ nil then return cache(N)
6  C ← removeUniversal(C)
7  CoCo ← connectedComponents(N)
8  N∧ ← ∅
9  foreach Co ∈ CoCo do
10     Xi ← selectVariable(Co)
11     N∨ ← ∅
12     foreach xi s.t. ⟨Xi, xi⟩ ∈ s do
13         N∨ ←
           N∨ ∪ cn2mddg(N, s[Co \ {Xi}] ∪ {⟨Xi, xi⟩})
14     N∧ ← N∧ ∪ dNode(Xi, N∨)
15 N ← aNode(N∧)
16 cache(N) ← N
17 return N
```

X_i. Dans le CN de l'exemple 1, X_2 est l'unique variable maximisant la valeur de *bc*. Il apparaît clairement qu'affecter en priorité les variables centrales du graphe primal $(\mathcal{X}, \mathcal{E})$ d'un CN favorise la génération de composantes connexes disjointes de taille similaire. Ceci permet la décomposition du réseau en plusieurs sous-réseaux indépendants (portant sur des ensembles disjoints de variables), de taille proche, pouvant être compilés séparément et rassemblés par un nœud $\wedge$ dans la représentation cible en MDDG. Le calcul de la centralité de tous les nœuds dans $(\mathcal{X}, \mathcal{E})$ peut être effectué en $\mathcal{O}(n.p)$, où $n = \#(\mathcal{X})$ et $p = \#(\mathcal{E})$. En pratique, le calcul de $bc(X_i)$ pour chaque nœud X_i du graphe $(\mathcal{X}, \mathcal{E})$ d'un CN est suffisamment efficace pour être effectué dynamiquement, c'est-à-dire être calculé pour chaque réseau rencontré lors de la compilation.

Le compilateur cn2mddg. L'algorithme 1 présente le pseudo-code du compilateur cn2mddg. La compilation d'un CN $\mathcal{N}$ s'effectue en appellant cn2mddg sur celui-ci et sur l'état de décision initial $\boldsymbol{s} = \{\langle X_i, x_i\rangle \mid X_i \in \mathcal{X}, x_i \in D_i\}$. Tout d'abord (ligne 1), on établit l'arc-cohérence de $\mathcal{N}$ avec $\boldsymbol{s}$ (les valeurs des variables apparaîssant dans $\boldsymbol{s}$ n'étant pas supportées par $\mathcal{N}$ sont supprimées de l'état). Pour des raisons d'efficacité, l'arc-cohérence est assurée au tout début (au premier appel) puis est maintenue dynamiquement à chaque fois qu'une nouvelle affectation élémentaire est considérée (ligne 13). Ensuite, $\mathcal{N}$ est conditionné par l'état courant (ligne 2). [4] A la ligne 3, nous appelons un *solver* CSP afin de déterminer si $\mathcal{N}$ est cohérent. Nous avons développé notre propre *solver*, basé sur une recherche en profondeur avec retour en arrière chronologique, utilisant l'heuristique dirigée par les conflits (devenue standard) *dom/wdeg* [4]. L'arc-cohérence est maintenue à chaque point de choix. Un poids est associé à chaque contrainte C_j de $\mathcal{C}$, et est incrémenté chaque fois qu'un conflit est détecté. Si $\mathcal{N}$ est incohérent, alors il est équivalent à la formule MDDG réduite à la feuille étiquetée par $\bot$, retournée par l'algorithme. La ligne 4 concerne le dernier cas de base, quand toutes les variables de $\mathcal{X}$ ont été considérées. Dans ce cas, $\mathcal{N}$ est équivalent à la formule MDDG $\top$, retournée par l'algorithme. On recherche si le réseau courant $\mathcal{N}$ a déjà été rencontré lors de la compilation (ligne 5). On utilise pour cela la fonction cache qui fait le lien entre les réseaux rencontrés et la racine du MDDG correspondant au réseau. Si $\mathcal{N}$ a déjà été trouvé durant la compilation, alors l'algorithme retourne simplement la racine du nœud de la formule MDDG correpondante.

Sinon on simplifie $\mathcal{N}$ en supprimant les contraintes universelles qu'elle contient (ceci est effectué par la fonction removeUniversal à la ligne 6). Puis, on cherche les composantes connexes du réseau obtenu (ligne 7). La fonction connectedComponents retourne une partition *CoCo* de l'ensemble courant des variables $\mathcal{X}$ correspondant aux composantes connexes du graphe primal de $\mathcal{N}$ (un simple parcours en largeur d'abord du graphe permet de les déterminer). Ensuite, on considère chaque élément *Co* de *CoCo* successivement (ligne 9). *Co* est donc un ensemble de variables indépendant des autres éléments de *CoCo* ; chaque réseau $\mathcal{N}$ correspondant peut donc être compilé séparément, que l'on regroupe dans un ensemble de nœuds $N_\wedge$ préalablement initialisé à l'ensemble vide (ligne 8). Pour chaque réseau *Co*, l'état de décision courant $\boldsymbol{s}$ peut être réduit aux variables apparaîssant dans *Co*. Une variable X_i est choisie grâce à la fonction selectVariable en ligne 10. Ensuite, en ligne 12, on considère successivement chacune des valeurs x_i du domaine courant de X_i. Plus précisément, pour chaque valeur x_i, on construit une affectation élémentaire correspondante $\{\langle X_i, x_i\rangle\}$. On conditionne le réseau par $\boldsymbol{s}$ que l'on a restreint aux variables de *Co* privées de X_i, auquel on

4. Dans notre implémentation, le conditionnement $\mathcal{N} \mid \boldsymbol{s}$ à la ligne 2 est fait implicitement (nous le présentons ainsi pour des raisons de clarté). A chaque étape, seules les listes des variables non instanciées et des domaines courants sont mises à jour (pour des raisons d'efficacité, les contraintes ne sont jamais modifiées).

ajoute l'affectation $\{\langle X_i, x_i\rangle\}$; ce réseau est compilé récursivement (ligne 13). L'ensemble $N_\vee$ des nœuds de décision, initialisé à l'ensemble vide (ligne 11), est mis à jour (ligne 13). Lorsque toutes les valeurs x_i du domaine ont été considérées, on construit un nouveau nœud de décision étiqueté par X_i que l'on ajoute à l'ensemble des nœuds $N_\wedge$ (ligne 14). Une fois traité l'ensemble des éléments de *CoCo*, on rasssemble les éléments de $N_\wedge$ en une conjonction sous forme d'un nœud $\wedge$ N grâce à la fonction aNode à la ligne 15. On associe ce nœud à l'entrée $\mathcal{N}$ du cache (ligne 16). Enfin, en ligne 17, nous retournons la racine du MDDG représentant $\mathcal{N}$.

Il est garanti que l'algorithme 1 termine, puisqu'à chaque étape de récursion, au moins une variable du CN initial est instanciée. Par construction, les solutions du MDDG en sortie sont exactement les mêmes que celles du CN en entrée.

5 Expérimentations

Alors que la compilation de CN est un sujet de recherche actif depuis des années (voir [27, 1, 12]), il n'existe pas à notre connaissance de compilateurs adaptés aux CNs de domaines non booléens et gérant des contraintes représentées en intension. En particulier, le compilateur AOMDD [5] suppose que toutes les contraintes du CN soient représentées en extension par leurs n-uplets autorisés. Heureusement, un grand nombre d'encodages CNF adapté aux CNs existent (entre autres [9, 3, 16, 25, 28, 14]), permettant de comparer notre approche à une compilation en Decision-DNNF précédée d'une transformation du réseau en CNF.

Notre cadre expérimental. Nous avons sélectionné 173 CNs issus de 15 familles différentes. Certaines instances contiennent des contraintes définies en extension, par leurs listes des n-uplets autorisés ou encore par leurs listes de n-uplets interdits. Pour d'autres instances, les contraintes sont données en intension.

Notre but fut de compiler chaque CN en une représentation en MDDG grâce au compilateur cn2mddg, et en une représentation en Decision-DNNF, en traduisant d'abord chaque instance en CNF, puis en utilisant le compilateur Dsharp, qui produit une représentation en Decision-DNNF à partir d'une formule CNF. Pour cela, nous avons considéré deux encodages CNF différents. Tout d'abord, le *sparse encoding*, qui consiste à encoder les domaines des variables, puis les contraintes suivant une méthode mixte (dans le but de minimiser le nombre de clauses générées, chaque contrainte est encodée en utilisant soit le *support encoding* soit le *conflict encoding*). Enfin, le *log encoding*, qui utilise un nombre logarithmique de variables booléennes pour encoder les domaines des variables et qui encode les contraintes suivant le *conflict encoding*. [6]

Pour chaque instance, nous avons calculé le temps de compilation (en secondes) et la taille de la formule compilée (le nombre d'arcs dans le graphe). Pour les approches basées sur une transformation en CNF, nous avons aussi calculé le temps de traduction et la taille de la formule CNF obtenue (le nombre de variables et de clauses). Nous avons lancé nos expérimentations sur un Quad-core Intel XEON X5550 avec 32Gio de mémoire. Pour chaque instance nous avons fixé un temps limite (*time-out*) à 3600 secondes pour la phase de transformation en CNF (ainsi que pour la phase de compilation) et un espace mémoire limite de 8Gio pour stocker la formule CNF (ainsi que pour la représentation de la forme compilée).

Quelques résultats obtenus. Dans le temps et la mémoire alloués, cn2mddg a réussi à compiler 131 instances sur les 173 testées. La compilation s'est terminée sur un *time-out* (TO) pour 32 instances, et sur un *memory-out* (MO) pour 10 instances. On peut observer un très grand écart de performances avec Dsharp qui n'a compilé que 83 instances quand le *sparse encoding* a été utilisé, et 61 instances lorsque le *log encoding* a été utilisé. Plus en détails, la transformation en CNF a conduit à 27 *memory-out* avec le *sparse encoding* (respectivement 35 avec le *log encoding*). Sur les 146 (respectivement 138) instances restantes, la compilation en Decision-DNNF par Dsharp s'est terminée sur un *time-out* pour 24 instances (respectivement 21), et sur un *memory-out* pour 39 instances (respectivement 56).

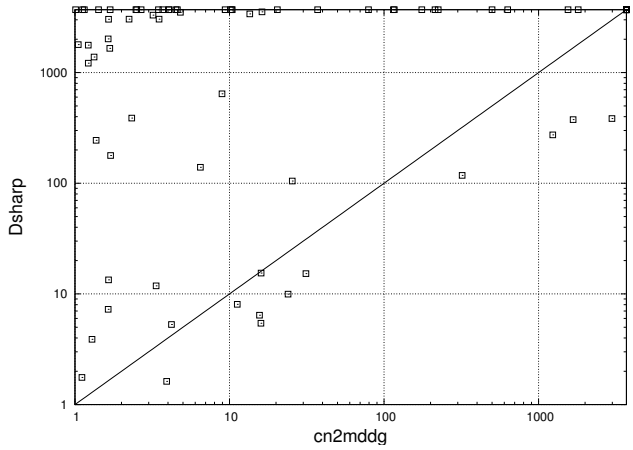
Quel que soit l'encodage considéré (*sparse encoding* ou *log encoding*) l'approche par transformation en CNF suivie d'une compilation en Decision-DNNF s'est révélée peu compétitive dans la plupart des cas. Ceci peut s'expliquer à la fois par le très grand nombre de variables booléennes apparaîssant dans la formule CNF générée, et de la perte d'information sur la structure du problème dû au format CNF (comparé à la représentation en CN). Notre compilateur cn2mddg s'est montré bien plus robuste puisqu'il a réussi à compiler beaucoup de CN qui se sont montrés hors de portée des approches par transformation en CNF puis compilation.

En particulier, chacune des 83 instances compilées avec succès par Dsharp (avec le *sparse encoding* en amont) se sont aussi montrées compilables en MDDG

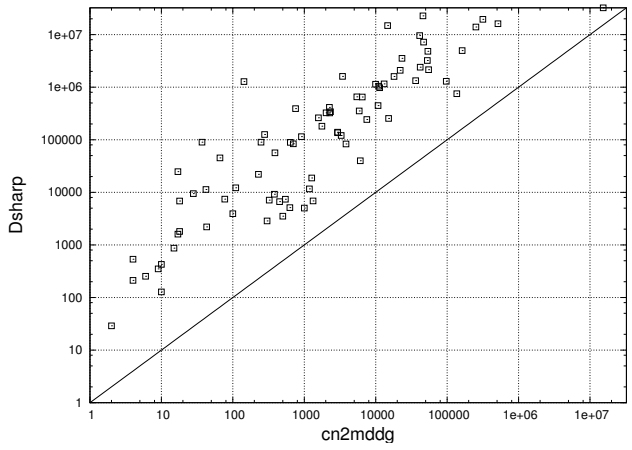
5. Disponible à l'adresse http://graphmod.ics.uci.edu/group/aomdd

6. Certains traducteurs disponibles, comme Sugar [24] ou Azucar [26], s'appuient sur des encodages qui ne préservent pas l'équivalence (ils sont orientés vers la résolution du problème de satisfaction), et ne peuvent donc être utilisés tels quels dans notre cadre.

en utilisant cn2mddg. Les temps de compilation requis pour créer les représentations en MDDG depuis les réseaux donnés en entrée sont souvent plus courts que les temps de compilation nécessaires pour créer les compilations en formules Decision-DNNF depuis les représentations CNF des réseaux donnés. Plus remarquablement encore, les tailles des formules compilées se révèlent toujours plus petites, parfois de plusieurs ordres de grandeur, quand le langage cible est MDDG comparé au langage Decision-DNNF.



(a) Temps



(b) Taille

FIGURE 3 – Comparaison entre temps/taille de compilation en MDDG et temps/taille de compilation en Decision-DNNF.

On peut observer cet écart clairement sur la figure 3, où chaque point représente l'une des 83 instances que Dsharp a réussi à compiler (avec le *sparse encoding* en amont). Le temps nécessaire pour compiler (respectivement la taille de) la représentation en MDDG est donnée sur l'axe des abscisses et le temps nécesssaire pour compiler (respectivement la taille de) la représentation en Decision-DNNF depuis la CNF encodée est donné sur l'axe des ordonnées. [7] Notez bien que toutes les échelles considérées sur le figure sont logarithmiques.

Le tableau 1 présente une sélection des résultats obtenus. Chaque ligne correspond à un réseau de contraintes CN $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ identifié par la colonne la plus à gauche. Les colonnes suivantes donnent respectivement le « type » de CN (en extension ou en intension), son nombre $\#\mathcal{X}$ de variables, son nombre $\#C$ de contraintes, l'arité maximale *maxA* des contraintes, la taille maximale *maxD* des domaines, un majorant *tw* de la largeur d'arbre de son graphe primal, [8] le temps nécessaire pour compiler le CN en MDDG en utilisant cn2mddg, et la taille de la formule compilée.

Pour les deux encodages en CNF que nous avons considérés, nous avons indiqué le nombre *#pv* de variables propositionnelles que contient la CNF, le nombre *#pcl* de clauses obtenues, le temps nécesssaire pour compiler la CNF en Decision-DNNF en utilisant Dsharp, et la taille de représentation en Decision-DNNF résultante.

Les résultats obtenus montrent que l'approche cn2mddg est plus compétitive que l'approche par transformation en CNF suivi d'une compilation en Decision-DNNF. La comparaison est, en effet, en faveur de cn2mddg que l'on considère comme critère de comparaison le nombre d'instances résolues, ou (ce qui est plus important) la taille des formules compilées. Les résultats montrent aussi que la compilation de CNs issus d'applications réelles, et possédant une complexité significative (souvent hors de portée de la compilation en *tree clustering* [10], vu la taille de leur domaine et la largeur d'arbre de leur graphe primal), vers le langage MDDG est réalisable.

Nous avons aussi réalisé une évaluation différentielle de chaque technique nouvelle utilisée dans cn2mddg afin de déterminer son apport. Pour des raisons d'espace, nous ne pouvons pas donner ici le détail de cette évaluation. Néanmoins, soit dom/wdeg+noU la version de cn2mddg pour laquelle l'heuristique de choix de variable *dom/wdeg* est utilisée (à la place de *bc*), et pour laquelle les contraintes universelles ne sont pas gérées spécifiquement ; notre évaluation a montré que dom/wdeg+noU a réussi à résoudre seulement 101 instances (sur 173) dans le temps et la mémoire alloués ; de plus, le nombre d'instances (sur 101) pour lesquelles la taille de la représentation en MDDG obtenue en utilisant cn2mddg (resp. dom/wdeg+noU) est inférieure à $p = \frac{1}{2}$ fois la taille de la représentation en MDDG obtenue en utilisant dom/wdeg+noU (resp. cn2mddg) est 35

7. Notons que la différence de temps en faveur de cn2mddg aurait été bien plus grande encore si nous avions pris en compte le temps nécessaire à l'encodage du réseau en CNF.

8. Calculé en utilisant QuickBB - voir http://www.hlt.utdallas.edu/~vgogate/quickbb.html - avec l'heuristique de choix de variable *random* et un *time-out* de 1800 secondes.

(resp. 6). Le nombre correspondant pour la proportion $p = \frac{1}{10}$ (au lieu de $\frac{1}{2}$) est 12 (resp. 0). Ainsi, pour plus de 10% des instances traitées, l'heuristique de choix de variable *bc* couplée à la gestion des contraintes universelles a conduit à une diminution de la taille de la forme compilée d'un ordre de grandeur.

6 Conclusion

Dans cet article, nous avons présenté un algorithme descendant `cn2mddg` pour la compilation de réseaux de contraintes à domaines finis, en graphes de décision décomposables multivalués. Parmi les différentes techniques qu'il incorpore, notre algorithme tire parti d'une méthode de *caching* spécifique, d'une nouvelle heuristique de choix de variable basée sur la *betweennes centrality*, et une gestion spécifique des contraintes universelles.

Notre large panel d'expérimentations a montré que `cn2mddg` permet de compiler des réseaux de contraintes qui ne peuvent être compilés en Decision-`DNNF` via un encodage préliminaire en `CNF`. Nous avons aussi montré que `cn2mddg` permet typiquement de produire des représentations compilées de bien plus petites tailles.

Ce travail ouvre de nombreuses perspectives. Parmi celles-ci, nous prévoyons d'étendre notre algorithme à la compilation de réseaux de contraintes pondérées, aux réseaux bayésiens et aux réseaux de Markov, et d'évaluer et de comparer le compilateur obtenu aux approches existantes pour traiter ces modèles graphiques.

Références

[1] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs application to configuration. *Artificial Intelligence*, 135(1-2) :199–234, 2002.

[2] J. Amilhastre, H. Fargier, A. Niveau, and C. Pralet. Compiling csps : A complexity map of (non-deterministic) multivalued decision diagrams. *International Journal on Artificial Intelligence Tools*, 23(4), 2014.

[3] O. Bailleux and Y. Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In *Proc. of CP'03*, pages 108–122, 2003.

[4] F. Boussemart, F. Hemery, Ch. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *Proc. of ECAI'04*, pages 146–150, 2004.

[5] U. Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 30(2) :136–145, 2008.

[6] H. Cambazard, T. Hadzic, and B. O'Sullivan. Knowledge compilation for itemset mining. In *Proc. of ECAI'10*, pages 1109–1110, 2010.

[7] A. Darwiche. New advances in compiling cnf into decomposable negation normal form. In *Proc. of ECAI'04*, pages 328–332, 2004.

[8] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17 :229–264, 2002.

[9] J. de Kleer. A comparison of ATMS and CSP techniques. In *Proc. of IJCAI'89*, pages 290–296, 1989.

[10] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3) :353–366, 1989.

[11] H. Fargier and P. Marquis. On the use of partially ordered decision graphs in knowledge compilation and quantified Boolean formulae. In *Proc. of AAAI'06*, pages 42–47, 2006.

[12] H. Fargier and M.-C. Vilarem. Compiling CSPs into tree-driven automata for interactive solving. *Constraints*, 9 :263–287, 2004.

[13] E. Freuder and B. O'Sullivan. Grand challenges for constraint programming. *Constraints*, 19 :150–162, 2014.

[14] I. P. Gent. Arc consistency in SAT. In *Proc. of ECAI'02*, pages 121–125, 2002.

[15] J. Huang and A. Darwiche. The language of search. *Journal of Artificial Intelligence Research*, 29 :191–219, 2007.

[16] K. Iwama and S. Miyazaki. Sat-variable complexity of hard combinatorial problems. In *Proc. of IFIP World Computer Congress'94*, pages 253–258, 1994.

[17] U. Junker. Configuration. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 24. Elsevier, 2006.

[18] R. Mateescu and R. Dechter. Compiling constraint networks into AND/OR multi-valued decision diagrams (AOMDDs). In *Proc. of CP'06*, pages 329–343, 2006.

[19] R. Mateescu, R. Dechter, and R. Marinescu. AND/OR multi-valued decision diagrams (AOMDDs) for graphical models. *Journal of Artificial Intelligence Research*, 33 :465–519, 2008.

[20] Ch.J. Muise, Sh.A. McIlraith, J.Ch. Beck, and E.I. Hsu. Dsharp : Fast d-DNNF compilation

	CN							
Name	type	$\#\mathcal{X}$	$\#C$	maxA	maxD	tw	time	size
rect-packing/rect-packingrpp07-true	I	1168	1273	8	23	15	31.22	5875
rect-packing/rect-packingrpp08-true	I	1584	1714	9	31	17	1234.72	162258
rect-packing/rect-packingrpp09-true	I	2196	2353	10	36	19	1673.33	514754
ghoulomb/ghoulomb3-4-6	I	3524	3543	16	37	?	164.95	74427
ghoulomb/ghoulomb3-4-5	I	2033	2051	11	26	31	15.17	5162
driver/normalized-driverlogw-08c-sat-ext	E	408	9321	2	11	92	15.63	2931
driver/normalized-driverlogw-08cc-sat-ext	E	408	9321	2	11	92	15.96	2931
scheduling/talent-concert	I	325	352	46	316	52	1277.21	404437
fapp/fapp18/normalized-fapp18-0350-8	I	350	2387	2	302	187	0.69	4243
fapp/fapp19/normalized-fapp19-0350-6	I	350	3114	2	802	130	79.34	1694146
costaArray/CostasArray10	I	110	338	4	19	23	10.39	13440
costaArray/CostasArray14	I	210	808	4	27	36	TO	-
photo/photophoto2	I	89	133	21	11	21	499.93	9564220
photo/photophoto1	I	75	102	18	9	18	10.13	210646
rlfap/normalized-scen4	I	680	3967	2	44	30	3.47	52226
rlfap/normalized-scen7-w1-f4	I	400	660	2	40	7	4.60	444483
radiation/radiation04	I	781	569	9	5180	33	-	MO
renault/normalized-renault-mod-32-ext	E	111	154	10	42	11	20.39	160238
renault/normalized-renault-mod-11-ext	E	111	149	10	42	10	16.22	41919
still-life/still-life7x7	I	690	803	50	50	49	1819.87	738478
configit/Aralia/edfpa15r	I	198	110	13	2	28	175.49	2044261
configit/Aralia/edfpa14q	I	505	194	22	2	34	TO	-
configit/Aralia/das9207	I	600	324	8	2	15	8.94	45853

	CNF - sparse mixed encoding				CNF - log conflict encoding			
Name	#pv	#pcl	time	size	#pv	#pcl	time	size
rect-packing/rect-packingrpp07-true	8709	110661	15.26	353048	2378	70087	37.70	976110
rect-packing/rect-packingrpp08-true	17724	248417	273.73	4973120	3225	147307	1139.74	7938679
rect-packing/rect-packingrpp09-true	37044	593518	375.66	16118647	4466	392657	TO	-
ghoulomb/ghoulomb3-4-6	MO	MO	MO	MO	MO	MO	MO	MO
ghoulomb/ghoulomb3-4-5	MO	MO	MO	MO	MO	MO	MO	MO
driver/normalized-driverlogw-08c-sat-ext	9528	62825	6.42	139306	1050	46081	32.78	499796
driver/normalized-driverlogw-08cc-sat-ext	9528	62825	5.43	136501	1050	46081	23.63	425617
scheduling/talent-concert	MO	MO	MO	MO	MO	MO	MO	MO
fapp/fapp18/normalized-fapp18-0350-8	9199429	63582486	-	MO	2810	52179077	-	MO
fapp/fapp19/normalized-fapp19-0350-6	166130802	867243022	-	MO	MO	MO	MO	MO
costaArray/CostasArray10	149564	841930	TO	-	540	3606946	TO	-
costaArray/CostasArray14	988671	5568047	TO	-	1036	34687218	-	MO
photo/photophoto2	685555	14326576	TO	-	204	10923133	-	MO
photo/photophoto1	73095	1328839	TO	-	172	1117281	TO	-
rlfap/normalized-scen4	915553	4875002	-	MO	4060	3058032	TO	-
rlfap/normalized-scen7-w1-f4	102556	683178	-	MO	2392	522995	-	MO
radiation/radiation04	MO	MO	MO	MO	MO	MO	MO	MO
renault/normalized-renault-mod-32-ext	222582	1755876	TO	-	286	138124077	-	MO
renault/normalized-renault-mod-11-ext	223718	1762294	3538.01	2399273	286	138117804	-	MO
still-life/still-life7x7	MO	MO	MO	MO	MO	MO	MO	MO
configit/Aralia/edfpa15r	396	24710	-	MO	396	24710	-	MO
configit/Aralia/edfpa14q	1010	5793030	TO	-	1010	5793030	TO	-
configit/Aralia/das9207	1200	3894	642.68	22707412	1200	3894	97.86	9937506

TABLE 1 – Une sélection de résultats expérimentaux.

with sharpSAT. In *Proc. of AI'12*, pages 356–361, 2012.

[21] U. Oztok and A. Darwiche. On compiling CNF into decision-DNNF. In *Proc. of CP'14*, pages 42–57, 2014.

[22] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier, 2006.

[23] O. Roussel and Ch. Lecoutre. XML Representation of Constraint Networks : Format XCSP 2.1. Technical report, Computing Research Repository (CoRR) abs/0902.2362, feb 2009.

[24] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2) :254–272, 2009.

[25] T. Tanjo, N. Tamura, and M. Banbara. A compact and efficient sat-encoding of finite domain CSP. In *Proc. of SAT'11*, pages 375–376, 2011.

[26] T. Tanjo, N. Tamura, and M. Banbara. Azucar : A sat-based CSP solver using compact order encoding - (tool presentation). In *Proc. of SAT'12*, pages 456–462, 2012.

[27] N.R. Vempaty. Solving constraint satisfaction problems using finite state automata. In *Proc. of AAAI'92*, pages 453–458, 1992.

[28] T. Walsh. SAT v CSP. In *Proc. of CP'00*, pages 441–456, 2000.