Encoding Multi-valued Decision Diagram Constraints as Binary Constraint Trees

Ruiwei Wang, Roland H. C. Yap

School of Computing, National University of Singapore
13 Computing Drive, 117417, Singapore
{ruiwei,ryap}@comp.nus.edu.sg

Abstract

Ordered Multi-valued Decision Diagram (MDD) is a compact representation used to model various constraints, such as regular constraints and table constraints. It can be particularly useful for representing ad-hoc problem specific constraints. Many algorithms have been proposed to enforce Generalized Arc Consistency (GAC) on MDD constraints. In this paper, we introduce a new compact representation called Binary Constraint Tree (BCT). We propose tree binary encodings to transform any MDD constraint into a BCT constraint. We also present a specialized algorithm enforcing GAC on the BCT constraint resulting from a MDD constraint. Experimental results on a large set of benchmarks show that the BCT GAC algorithm can significantly outperform state-of-the-art MDD as well as table GAC algorithms.

1 Introduction

Many combinatorial problems in real life can be modelled as Constraint Satisfaction Problems (CSPs). Especially, the problems requiring specific constraints which do not fit well with existing known constraints can be modelled with ad-hoc constraints, such as Ordered Multi-valued Decision Diagram (MDD) constraints (Cheng and Yap 2010). Some ad-hoc constraints have been widely studied. For example, many Generalized Arc Consistency (GAC) algorithms have been proposed as the propagation algorithm to handle MDD constraints and (non-binary) table constraints (Yap, Xia, and Wang 2020). In this paper, we present a new representation for MDDs, showing how to transform any MDD constraint into Binary Constraint Trees (BCTs) and design MDD GAC propagators using BCTs.

MDD (Srinivasan et al. 1990) is a compact representation which can be exponentially smaller than its corresponding table representation. MDD constraints have been shown to be useful in modelling other constraints such as regular constraints (Pesant 2004), table constraints and problem specific constraints (Cheng and Yap 2006, 2010). Intuitively, GAC propagators for MDD constraints have the potential to outperform those of table constraints as they exploit structure inside the constraint. However, prior works (Verhaeghe, Lecoutre, and Schaus 2018, 2019) show that table GAC algorithms using bitwise operations, e.g. the CT algorithm (Demeulenaere et al. 2016), outperform MDD GAC algorithms on a large set of benchmarks. As such an open question is whether MDD GAC algorithms can overall outperform the CT algorithm. We answer this, showing solving table constraints as MDDs encoded with BCTs outperforms the state-of-the-art table GAC algorithm CT.

Rather than using GAC for non-binary constraints, we can transform the constraints into binary constraints and use Arc Consistency (AC). Many binary encodings have been proposed to encode table constraints as binary constraints, such as Dual/Double Encoding (Dechter and Pearl 1989; Stergiou and Walsh 1999), Hidden Variable Encoding (HVE) (Rossi, Petrie, and Dhar 1990) and Bipartite Encoding (BE) (Wang and Yap 2020). AC on the encoded instances can achieve GAC or higher-order consistencies on the original CSPs (Bessiere, Stergiou, and Walsh 2008). Recently, it has been shown that specialized AC propagators can work very well on binary encoding instances (Wang and Yap 2019, 2020). A question though is whether the binary encoding approach is able to give compact representations for a MDD constraint, We demonstrate a novel binary encoding which is at least as compact as the MDD it encodes.

In this paper, we introduce a compact representation called BCT which is a set of binary constraints with a tree structure. BCTs can be exponentially smaller than MDDs when representing some constraints. We show that MDD constraints can be transformed into BCT constraints by using Tree Binary Encodings (TBEs). We first propose a basic encoding, the Direct Tree Binary Encoding (DTBE) to encode any MDD constraint as a BCT constraint with the same size as the original MDD constraint. We then give four reduction rules to reduce a BCT constraint by eliminating, merging and reconstructing variables. We propose a specialized propagator to enforce GAC on MDD constraints encoded as BCT constraints taking advantage of our new representation. Experimental results show that the reduction rules significantly reduce the size of BCTs making the GAC propagator on BCT constraints much faster than the state-of-the-art MDD GAC propagator CD (Verhaeghe, Lecoutre, and Schaus 2018) and also the table GAC propagator CT.

2 Background

A CSP P is a pair (X, C) where X is a set of variables, $\mathcal{D}(x_i)$ the domain of a variable x_i, and C is a set of constraints. A *literal* of a variable x is a variable value assignment (x, a). A *tuple* over a set of variables $\{x_{i_1}, x_{i_2}, \ldots, x_{i_r}\}$ is denoted by a *set of literals* $\{(x_{i_1}, a_1), (x_{i_2}, a_2), \ldots, (x_{i_r}, a_r)\}$. Each constraint c_j has a constraint scope $scp(c_j) \subseteq X$ and a relation $rel(c_j)$ defined by a set of tuples over $scp(c_j)$. $NC(x) = \{c \in C | x \in scp(c)\}$ is a set of the constraints including a variable x. $N(x) = \cup_{c \in NC(x)} scp(c)$ and $N^-(x) = N(x) \setminus \{x\}$ denote the neighbors of x. The arity of a constraint c is the number of variables in its scope, i.e. $|scp(c)|$. A constraint c is a *binary constraint* if $|scp(c)| = 2$. A CSP P is called a *binary CSP* if the largest constraint arity is 2. A binary CSP is *normalized* if all constraints have different scopes. Given any set of variables V and literals τ, we use $\tau[V] = \{(x, a) \in \tau | x \in V\}$ to denote a subset of τ, while $T[V] = \{\tau[V] | \tau \in T\}$ is the *projection* of tuples T on V. A tuple τ over X is a solution of P if $\tau[scp(c)] \in rel(c)$ for all constraints $c \in C$ and $a \in \mathcal{D}(x)$ for all $(x, a) \in \tau$. $sol(X, C)$ (or $sol(P)$) denotes all solutions of P.

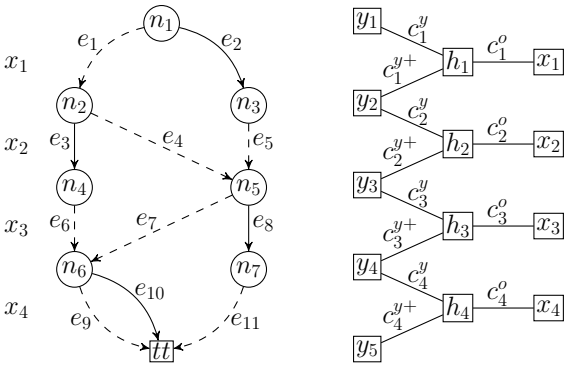
A *support of a value* $a \in \mathcal{D}(x)$ on a constraint c is a tuple $\tau \in rel(c)$ such that $(x, a) \in \tau$ and $b \in \mathcal{D}(y)$ for all $(y, b) \in \tau$. A variable $x \in scp(c)$ is *Generalized Arc Consistent* (GAC) on c if a has a support on c for all $a \in \mathcal{D}(x)$. c is GAC if all variables in $scp(c)$ are GAC on c. A CSP (X, C) is GAC if every constraint in C is GAC. For binary CSPs, GAC is also called *Arc Consistency* (AC).

An *Ordered Multi-valued Decision Diagram (MDD)* (Cheng and Yap 2010) with respect to an order O over a set of r variables (O_i denotes the i-th variable) is a labelled layered directed acyclic graph denoted by the triple, $(\cup_{i=1}^{r+1} L_i, \cup_{i=1}^{r} E_i, label)$, where:

- L_i is a set of nodes at the i^{th} layer. L_1 only includes the root node and L_{r+1} the terminal node;
- E_i is a set of directed edges e pointing from a node $out(e) \in L_i$ to another node $in(e) \in L_{i+1}$, while $label(e)$ maps edge e to a value in $\mathcal{D}(O_i)$;
- there is at most 1 edge $e \in E_i$ such that $out(e) = n_j$ and $label(e) = l$ for each value $l \in \mathcal{D}(O_i)$ and node $n_j \in L_i$;
- for each node (edge), there is at least one path from root to the terminal node including the node (edge).

We use $mdd(c^*, O)$ to denote a MDD representing a constraint c^* w.r.t. an order O over $scp(c^*)$, where every tuple τ in $rel(c^*)$ corresponds to a path from root to the terminal. Moreover, MDDs can be reduced by merging distinct nodes, where two nodes are *distinct* if for each edge from one of two nodes, there is an edge from the other node such that two edges have the same label and point to the same node. All MDDs considered in this paper are ordered. In addition, we use the pReduce (Perez and Régin 2015) algorithm to merge all distinct nodes in a MDD but do not require removing any *redundant nodes* n_j in L_i such that there are $|\mathcal{D}(O_i)|$ edges point from n_j to the same node.[1]

[1]Many MDD GAC propagators, such as MDD4R (Perez and Régin 2014) and CD, also do not require removing redundant nodes.



(a) MDD (b) DTBE

Figure 1: A MDD and the corresponding DTBE where the dashed (and solid) lines in the MDD denote value 0 (and 1).

Example 1. Figure 1a is a MDD representing a *sequence constraint* (Beldiceanu and Contejean 1994) over 4 variables $X = \{x_1, \ldots, x_4\}$ with variable domain is $\{0, 1\}$ and variable order is from x_1 to x_4. The constraint means that for any 2 consecutive variables, the value 1 is assigned to at most 1 variable.

3 Encoding MDD constraints as BCTs

It is well-known that AC on a set of binary constraints having a tree structure is sufficient to achieve global consistency. We exploit this by observing that we can enforce GAC on a constraint by encoding it into a set of binary constraints with a tree structure and maintain AC on these binary constraints. In this section, we propose a transformation reducing constraints in this fashion.

Definition 1. A *Binary Constraint Tree* (BCT) is a normalized binary CSP whose constraint graph is a tree. A *BCT constraint* c is a pair (V, P) such that $P = (X, C)$ is a BCT and $scp(c) = V$ and $V \subseteq X$ and $rel(c) = sol(X, C)[V]$. A *tree binary encoding (TBE)* of a constraint c^* is a BCT $P = (X, C)$ such that $(scp(c^*), P)$ has the same constraint relations as c^* where the variables in $scp(c^*)$ and $X \setminus scp(c^*)$ are called the *original* and *hidden* variables, respectively.

Given any TBE P of a constraint c^*, the constraint graph of P is a tree, thus, if P is AC, every literal of a variable in X is included by at least a solution of P, giving the following.

Proposition 1. Enforcing AC on P achieves GAC on c^*.

Direct tree binary encoding

We now introduce a specific TBE, DTBE, for MDD constraints which directly encodes the nodes and edges in each layer of the MDDs as hidden variables, and then uses binary constraints to connect the edges with the nodes and labels. Its details are given in the following definition.

Definition 2. A *direct tree binary encoding* (DTBE) of a constraint c^* w.r.t. a MDD $mdd(c^*, O)$ is a BCT $dtbe(c^*)$= $(Y \cup H \cup scp(c^*), \{c_1^o, c_1^y, c_1^{y+}, ..., c_r^o, c_r^y, c_r^{y+}\})$ where

- $r = |scp(c^*)|$ and O is an order over $scp(c^*)$;
- $Y = \{y_1, ..., y_{r+1}\}$, $H = \{h_1, ..., h_r\}$ and $scp(c_i^o) = \{O_i, h_i\}$, $scp(c_i^{y+}) = \{y_{i+1}, h_i\}$, $scp(c_i^y) = \{y_i, h_i\}$;
- $\mathcal{D}(y_i)$ is the set of nodes in the i^{th} layer of $mdd(c^*, O)$;
- $\mathcal{D}(h_i)$ is the set of edges which point from the nodes in the i^{th} layer of $mdd(c^*, O)$ to the next layer;
- $rel(c_i^y) = \{\{(h_i, a), (y_i, b)\} | a \in \mathcal{D}(h_i), b = out(a)\}$;
- $rel(c_i^o) = \{\{(h_i, a), (O_i, l)\} | a \in \mathcal{D}(h_i), l = label(a)\}$;
- $rel(c_i^{y+}) = \{\{(h_i, a), (y_{i+1}, b)\} | a \in \mathcal{D}(h_i), b = in(a)\}$.

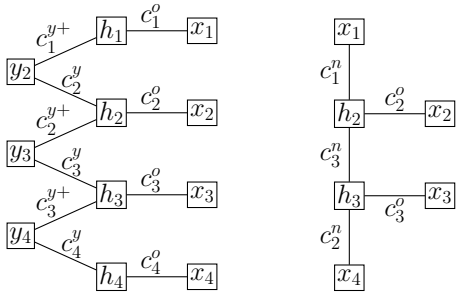
Figure 1b illustrates the DTBE of a constraint c^* w.r.t. the MDD given in Figure 1a. The MDD has 4 (and 5) layers of edges (nodes) corresponding to the hidden variables $H = \{h_1, \ldots, h_4\}$ (and $Y = \{y_1, \ldots, y_5\}$). The domains of the hidden variables are the set of the nodes or edges at each layer of the MDD, e.g. $\mathcal{D}(y_1) = \{n_1\}$ is the set of the nodes in the first layer of the MDD and $\mathcal{D}(h_1) = \{e_1, e_2\}$ is the set of the edges which point from n_1. Every node and edge in Figure 1b denotes a variable and binary constraint respectively. The constraint relations can be directly constructed using definition 2, e.g.
$rel(c_1^o)$=$\{\{(h_1, e_1), (x_1, 0)\}, \{(h_1, e_2), (x_1, 1)\}\}$,
$rel(c_1^y)$=$\{\{(h_1, e_1), (y_1, n_1)\}, \{(h_1, e_2), (y_1, n_1)\}\}$,
$rel(c_1^{y+})$=$\{\{(h_1, e_1), (y_2, n_2)\}, \{(h_1, e_2), (y_2, n_3)\}\}$.
We can see that the constraint graph is a tree and every solution of $dtbe(c^*)$ denotes a path in the MDD which corresponds to a tuple in $rel(c^*)$, thus, $dtbe(c^*)$ is a TBE of c^*. We also call $(scp(c^*), dtbe(c^*))$ as a DTBE constraint.

Example 2. The following constraint over the variables $X = \{x_1, \cdots, x_r\}$ with domains $D = \{1, \cdots, r\}$ and $r > 2$ expresses the relation $\bigvee_{i=1}^{r} \bigvee_{j=i+1}^{r} (x_i = x_j)$. This example constraint is used to show that BCTs can be exponentially smaller than MDDs on representing some constraints. The constraint expresses the negation of the *alldifferent* constraint. The size of the MDD representing the constraint is exponential in r.

The constraint can be represented as a BCT (V, C) where $V = X \cup \{h\}$ and $C = \{c_1, \cdots, c_r\}$ and $\mathcal{D}(h) = \{a_{jkl} | 1 \leq j, k, l \leq r, j \neq k\}$ and for $1 \leq i \leq r$, $scp(c_i) = \{x_i, h\}$ and $rel(c_i) = \{\ \{(h, a_{jkl}), (x_i, a)\} \mid j \neq i, k \neq i, l \in D, a \in D\} \cup \{\ \{(h, a_{jkl}), (x_i, a)\} \mid i \in \{j, k\}, a = l, a \in D\}$. For any tuple τ over X, τ is in the constraint relation, i.e. there exists variables $x_i, x_j \in X$ and a value $a \in D$ such that $\{(x_i, a), (x_j, a)\} \subset \tau$, if and only if $\tau \cup \{(h, a_{ija})\}$ is a solution of (V, C). So, BCT can be exponentially smaller than MDD when representing this constraint.

4 TBE reduction rules

After encoding a MDD constraint c^* into a BCT with the DTBE encoding, we can maintain GAC on c^* by enforcing AC on the BCT (Proposition 1). Our experimental results (see Figure 4) show that the GAC algorithm on DTBE constraints (given in Section 5) is competitive with the state-of-the-art MDD GAC algorithm CD (Verhaeghe, Lecoutre, and Schaus 2018). This shows that DTBE is already good as the MDD representation. Next, we show how to make the representation more compact. We propose four reduction rules to reduce any TBE $P_1 = (X_1, C_1)$ of a constraint c^*.



(a) P_{2a}: Rules 1 (b) P_{2b}: Rules 2

Figure 2: Applying **Rules 1,2** for reducing the DTBE given in Figure 1b, where each rectangle and line respectively denote a variable and a constraint.

Two hidden variable elimination rules

The first rule is to eliminate any hidden variable in P_1 which is only included by one constraint c in C_1. Assume $scp(c) = \{x, y\}$ and let $R_1(P_1, x) = (X_1 \setminus \{x\}, C_1 \setminus \{c\})$ be the CSP generated by eliminating x. Since c is removed, the domain of y is reconstructed as $\{b \in \mathcal{D}(y) | (y, b) \in rel(c_i)[\{y\}]\}$. A tuple over $X_1 \setminus \{x\}$ is a solution of $R_1(P_1, x)$ iff it can be extended to a solution of P_1, so $R_1(P_1, x)$ is a TBE of c^*.

Lemma 1 (Rule 1). $R_1(P_1, x)$ is a TBE of c^*.

Next, we present the second rule that eliminates any hidden variable x that is only included by 2 constraints $c_i, c_j \in C_1$. Assume $scp(c_i) = \{x, y\}$ and $scp(c_j) = \{x, z\}$. Let $R_2(P_1, x) = (X_1 \setminus \{x\}, C \cup \{c'\})$ be the CSP generated by eliminating x where $C = C_1 \setminus \{c_i, c_j\}$ and $scp(c') = \{y, z\}$ and $rel(c') = sol(\{x, y, z\}, \{c_i, c_j\})[\{y, z\}]$. Similarly, a tuple over $X_1 \setminus \{x\}$ is a solution of $R_2(P_1, x)$ iff it can be extended to a solution of P_1, thus, $R_2(P_1, x)$ is a TBE of c^*.

Lemma 2 (Rule 2). $R_2(P_1, x)$ is a TBE of c^*.

For the example DTBE P given in Figure 1b, two variables y_1, y_5 can be eliminated with **Rule 1**. The TBE $P_{2a} = R_1(R_1(P, y_1), y_5)$ generated by applying **Rule 1** is given in Figure 2a. Then five variables h_1, h_4, y_2, y_3, y_4 in Figure 2a can be eliminated by **Rule 2**. The reduced TBE $P_{2b} = R_2(R_2(R_2(R_2(R_2(P_{2a}, h_1), h_4), y_2), y_4), y_3)$ is shown in Figure 2b, where c_1^n, c_2^n, c_3^n are the new constraints generated. The relations of the new constraints can be directly constructed by **Rule 2**, e.g. $rel(c_3^n)$=$\{< e_3, e_6 >, < e_4, e_7 >, < e_4, e_8 >, < e_5, e_7 >, < e_5, e_8 >\}$ where $< e_i, e_j >$ denotes the tuple $\{(h_2, e_i), (h_3, e_j)\}$.

A hidden variable merging rule

The third rule merges any 2 hidden variables x, y which are constrained by a constraint $c_i \in C_1$, i.e. $scp(c_i) = \{x, y\}$, as a new variable m such that $\mathcal{D}(m) = rel(c_i)$, while every constraint $c_j \in C_1$ between x (or y) and another variable $z \notin \{x, y\}$ is replaced with a new constraint c_j' where $scp(c_j') = \{z, m\}$ and $rel(c_j') = \{\{(z, a), (m, \tau)\} \mid \tau \in rel(c_i), \tau \cup \{(z, a)\}$ is in $sol(\{x, y, z\}, \{c_i, c_j\})\}$. We can get a new CSP $R_3(P_1, c_i) = (X \cup \{m\}, C \cup C')$ where $X = X_1 \setminus \{x, y\}$ and $C = \{c \in C_1 | x \notin scp(c), y \notin scp(c)\}$

and $C' = \{c'_j | c_j \in C_1, c_j \notin C, c_j \neq c_i\}$. For each tuple τ over X_1, τ is a solution of P_1 iff $\tau[X] \cup \{(m, \tau[\{x, y\}])\}$ is a solution of $R_3(P_1, c_i)$. So $R_3(P_1, c_i)$ is a TBE of c^*.

Lemma 3 (Rule 3). $R_3(P_1, c_i)$ is a TBE of c^*.

Rule 3 encodes binary constraints as hidden variables by regarding constraint relations as variable domains, where each value in the domain corresponds to a tuple in the relation. We then present a rule to reconstruct hidden variables.

A hidden variable reconstruction rule

We first introduce c-table (Katsirelos and Walsh 2007) before giving the fourth rule. A *c-tuple* τ over a set of variables V is a set of literals such that for each literal $(x, a) \in \tau$, the variable x is included in V. We can see that a tuple is a special case of c-tuples. Every c-tuple τ over variables V represents a set of tuples denoted as $\mathcal{T}(\tau) = \{\tau' \subseteq \tau | \tau'$ is a tuple over $V\}$. A *c-table* is a set of c-tuples. A c-table rel_1 represents a set of tuples rel_2 if the c-tuples in rel_1 represent the tuples in rel_2, i.e. $\cup_{\tau \in rel_1} T(\tau) = rel_2$

The fourth rule reconstructs the domain of any hidden variable with its conditional c-tables defined as follows.

Definition 3 (CC-T)**.** Given a normalized binary CSP P and a variable x in P, the *conditional table* of x is a set of tuples $ct(P, x) = sol(N(x), NC(x))[N^-(x)]$. A *conditional c-table* (CC-T) of x is a c-table representing $ct(P, x)$.

For any variable x in P_1, the domain of x defines a CC-T $cct(P_1, x) = \{t^a \mid a \in \mathcal{D}(x)\}$ where $t^a = \{ (y, b) \mid y \in scp(c), c \in NC(x), \{(y, b), (x, a)\} \in rel(c) \}$ is a c-tuple. For any constraint in $NC(x)$, t^a includes all literals (y, b) such that $\{(y, b), (x, a)\} \in rel(c)$. Therefore, for each solution τ of P_1, if a literal (x, a) is in τ, then the tuple $\tau[N^-(x)]$ is a subset of t^a, otherwise there is a constraint $c \in NC(x)$ such that $\tau[scp(c)] \notin rel(c)$.

For example, $cct(P_{2b}, h_3)$ is a CC-T including 3 c-tuples $t^{e_6}, t^{e_7}, t^{e_8}$, where $t^{e_6} = \{(h_2, e_3), (x_3, 0), (x_4, 0), (x_4, 1)\}$, $t^{e_7} = \{(h_2, e_4), (h_2, e_5), (x_3, 0), (x_4, 0), (x_4, 1)\}$, and $t^{e_8} = \{(h_2, e_4), (h_2, e_5), (x_3, 1), (x_4, 0)\}$. We can see that for any solution τ of P_{2b} including the literal (h_3, e_6), the tuple $\tau[\{h_2, x_3, x_4\}]$ is a subset of the c-tuple t^{e_6}.

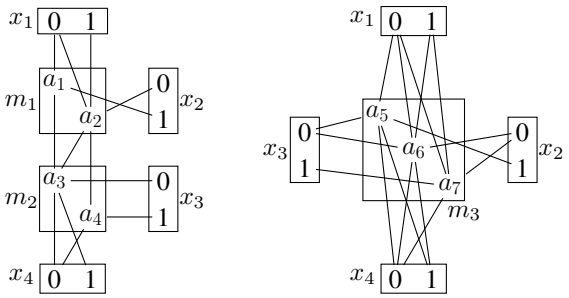
Conversely, given any CC-T rel of x, we can reconstruct x into a new variable x' such that $\mathcal{D}(x') = rel$ and replace each constraint c between x and another variable y with a new binary constraint c' such that $scp(c') = \{x', y\}$ and $rel(c') = \{\{(x', \tau'), (y, a)\} \mid \tau' \in \mathcal{D}(x'), (y, a) \in \tau'\}$. Correspondingly, we can get a new CSP $R_4(P_1, x, rel) = (X \cup \{x'\}, C \cup C')$ where $X = X_1 \setminus \{x\}$ and $C' = \{c' | c \in C_1, x \in scp(c)\}$ and $C = C_1 \setminus NC(x)$. The following lemma shows that $R_4(P_1, x, rel)$ is also a TBE.

Lemma 4 (Rule 4). $R_4(P_1, x, rel)$ is a TBE of c^*.

Proof. P_1 is a BCT and **Rule 4** keeps the structure of P_1, i.e. the constraint graph of $R_4(P_1, x, rel)$ is connected and the number of constraints (edges) is equal to that of variables (nodes) minus 1. Therefore, $R_4(P_1, x, rel)$ is also a BCT. Given any tuple $\tau \in sol(X, C)$:

- If there is a c-tuple $\tau' \in \mathcal{D}(x')$ such that the tuple $\tau \cup \{(x', \tau')\}$ is a solution of $R_4(P_1, x, rel)$, i.e. the tuple τ is in $sol(R_4(P_1, x, rel))[X]$, then the tuple $\tau[scp(c)] \cup \{(x', \tau')\}$ is in $rel(c')$ and $\tau[scp(c)] \subseteq \tau'$ for all $c \in NC(x)$, thus, $\tau[N^-(x)] \subseteq \tau'$. At the same time, $\mathcal{D}(x')$ is a c-table representing $ct(P_1, x)$, thus, $\tau[N^-(x)]$ is in $ct(P_1, x)$. $ct(P_1, x)$ is equal to $sol(N(x), NC(x))[N^-(x)]$, so τ is in $sol(P_1)[X]$.
- Conversely, if τ is in $sol(P_1)[X]$, then $\tau[N^-(x)]$ is in $ct(P_1, x)$ and there is $\tau' \in \mathcal{D}(x')$ such that $\tau[N^-(x)] \subseteq \tau'$, since $\mathcal{D}(x')$ is a c-table representing $ct(P_1, x)$. Correspondingly, for any $c \in NC(x)$, $\tau[scp(c)] \cup \{(x', \tau')\}$ is in $rel(c')$, since $\tau[scp(c)] \subseteq \tau'$, so $\tau \cup \{(x', \tau')\}$ is a solution of $R_4(P_1, x, rel)$ and τ is in $sol(R_4(P_1, x, rel))[X]$.

So we can get that $sol(R_4(P_1, x, rel))[scp(c^*)]$ is equal to $sol(P_1)[scp(c^*)]$ and $rel(c^*)$, which means the BCT $R_4(P_1, x, rel)$ is a TBE of c^*. □



(a) P_{3a}: Rule 4 on P_{2b} (b) P_{3b}: Rule 3 on P_{3a}

Figure 3: Applying **Rules 3,4** for reducing the TBE given in Figure 2b, where every rectangle denotes a variable and the set of values in the rectangle is the domain of the variable, and each line between 2 values in two rectangles denotes a tuple in the relation of the constraint between the 2 variables corresponding to the two rectangles.

For the TBE P_{2b} given in Figure 2b, the hidden variable h_2 can be reconstructed by **Rule 4** as a new hidden variable m_1 with the CC-T $rel_2 = \{a_1, a_2\}$, where $a_1 = \{(x_1, 0), (x_2, 1), (h_3, e_6)\}$ and $a_2 = \{(x_1, 0), (x_1, 1), (x_2, 0), (h_3, e_7), (h_3, e_8)\}$, and then the hidden variable h_3 can be reconstructed as a new hidden variables m_2 with the CC-T $rel_3 = \{a_3, a_4\}$ where $a_3 = \{(x_3, 0), (x_4, 0), (x_4, 1), (m_1, a_1), (m_1, a_2)\}$ and $a_4 = \{(x_3, 1), (x_4, 0), (m_1, a_2)\}$. We show an example of **Rule 4** in Figure 3a which gives TBE $P_{3a} = R_4(R_4(P_{2b}, h_2, rel_2), h_3, rel_3)$. The constraint relations are reconstructed based on the CC-Ts, e.g. the relation of the constraint c_4^n between m_1 and m_2 is equal to $\{\{(m_1, a_1), (m_2, a_3)\}, \{(m_1, a_2), (m_2, a_3)\}, \{(m_1, a_2), (m_2, a_4)\}\}$, which is based on that $(m_1, a_1) \in a_3$, $(m_1, a_2) \in a_3$ and $(m_1, a_2) \in a_3$.

We can also merge the hidden variables m_1 and m_2 in P_{3a} as a new hidden variable m_3 with **Rule 3**. An example using **Rule 4** is Figure 3b giving the TBE $P_{3b} = R_3(P_{3a}, c_4^n)$, where each value in $\mathcal{D}(m_3) = \{a_5, a_6, a_7\}$ corresponds to a tuple in $rel(c_4^n)$. For example, a_5 corresponds to the tuple $\{(m_1, a_1), (m_2, a_3)\}$ and the values connecting to a_5 in Figure 3b also connect to a_1 or a_3 in Figure 3a.

Evaluating TBE sizes

Rules 1-4 can be used to make a new TBE. However, the new TBEs may not always be more compact than the original. Thus, we need to evaluate the sizes of the TBEs before reconstructing them. We do not use rules which do not reduce the TBE as measured by the evaluated size. In this paper, as we represent binary constraints as binary matrices, the size of any binary constraint c is evaluated as $|\mathcal{D}(x) \times \mathcal{D}(y)|$ where $scp(c) = \{x, y\}$. The evaluated size, $size(P)$, of any TBE P is the sum of the evaluated sizes of all binary constraints in P.

The evaluated sizes of the TBEs given in Figures 1b, 2a, 2b, 3a and 3b are 66, 56, 33, 20 and 24, respectively. We can see that P_{3a} is the smallest TBE. The evaluated size of P_{3a} is 3.3 times smaller than that of the original DTBE given in Figure 1b. In addition, $size(P_{3b})$ is greater than $size(P_{3a})$, so we do not use **Rule 3** to merge the hidden variables m_1 and m_2 in P_{3a}.

Constructing CC-Ts

We can reconstruct the domain of a hidden variable x in a TBE P with any CC-Ts of x via **Rule 4**. However, it is NP-hard to get the optimal c-table (Katsirelos and Walsh 2007) representing the conditional table $ct(P, x)$. In this paper, we construct a CC-T by merging some c-tuples in $ct(P, x)$ and $cct(P, x)$. Two c-tuples τ_1 and τ_2 in a CC-T rel_1 of x are *mergeable* w.r.t. a variable $y \in N^-(x)$ if $\tau_1 \setminus \tau_2$ (or $\tau_2 \setminus \tau_1$) is empty or only includes some literals of y, since $\mathcal{T}(\tau_1) \cup \mathcal{T}(\tau_2) = \mathcal{T}(\tau_1 \cup \tau_2)$. This is an equivalence relation defining a partition $par(rel_1, y)$ of rel_1 where all c-tuples in the same subset S are mergeable w.r.t. y. We use $merge(rel_1, y) = \{\cup_{\tau \in S} \tau \mid S \in par(rel_1, y)\}$ to denote a c-table generated by merging c-tuples with $par(rel_1, y)$. Obviously, $merge(rel_1, y)$ is also a CC-T of x.

We then show how to generate a CC-T rel used for **Rule 4**. Let v be a variable in $N^-(x)$ with maximum domain size. If $\prod_{y \in (N^-(x) \setminus \{v\})} |\mathcal{D}(y)| < |\mathcal{D}(x)|$, which means that the number of c-tuples in $merge(ct(P, x), v)$ is less that of $cct(P, x)$, then the CC-T rel is initialized as $merge(ct(P, x), v)$, otherwise it is initialized as $cct(P, x)$. At the same time, we go through the variables y in $N^-(x)$ one by one in an order of decreasing domain size and iteratively reset rel as the CC-T $merge(rel, y)$.

Reducing DTBE

Algorithm 1 is used to reduce any DTBE of a MDD constraint c^* by applying the 4 reduction rules. It is NP-hard to construct the optimal TBE by reducing a DTBE even if we only use **Rule 4**. As such, we propose the following heuristic based on preliminary experiments. We apply the reduction rules in a certain order and also process the hidden variables with order O. The rules are only applied when it reduces the evaluated size of the DTBE. The algorithm first generates a DTBE (X, C) of the constraint c^* at Line 1. Then the following functions applying a reduction rule are used to transform (X, C):

- $ApplyRule1(X, C)$ is called at Line 2. It scans all hidden variables $x \in X$ and applies **Rule 1** to eliminate x if x is only included by one constraint in C.
- $ApplyRule2(X, C)$ is called at Lines 3,5 and 8. It scans all hidden variables x and applies **Rule 2** to eliminate x if x is only included by two constraint c_i, c_j in C such that $|\mathcal{D}(y)| \times |\mathcal{D}(z)|$ is not greater than $|\mathcal{D}(x)| \times (|\mathcal{D}(y)| + |\mathcal{D}(z)|)$ where $scp(c_i) = \{x, y\}$ and $scp(c_j) = \{x, z\}$.
- $ApplyRule3(X, C)$ is called at Line 6. It repeatedly scans the hidden variables from O_1 to O_k and back to O_1 until **Rule 3** cannot be applied to merge any variables, where k is the number of hidden variables. For any hidden variable O_i, if there is a constraint $c \in C$ between O_i and another hidden variable O_j such that the evaluated size of $R_3((X, C), c)$ is not greater than that of (X, C), then O_i and O_j are merged by **Rule 3**.
- $ApplyRule4(X, C)$ is called at Lines 4 and 7. It scans all hidden variables in the order O and uses **Rule 4** to reconstruct the hidden variable domains. The detail of the CC-Ts used by **Rule 4** is given in the last subsection.

Algorithm 1: ReducingDTBE(c^*)

```
1 (X, C) ← dtbe(c*);
2 ApplyRule1(X, C);
3 ApplyRule2(X, C);
4 ApplyRule4(X, C);
5 ApplyRule2(X, C);
6 ApplyRule3(X, C);
7 ApplyRule4(X, C);
8 ApplyRule2(X, C);
  return (X, C);
```

The initial order O over the hidden variables in $dtbe(c^*)$ is set as $y_{r+1} < h_r < y_r < ... < h_1 < y_1$ where $r = |scp(c^*)|$. After a new variable m is added by merging a variable x with its neighbors via **Rule 3** or reconstructing x via **Rule 4**, we update the order as $m < z$ (or $z < m$) for all variables z such that $x < z$ (or $z < x$).

5 A BCT GAC propagator

The BCT is a binary CSP with a special structure, which we exploit to devise more efficient propagators. We follow the ideas of special propagation order in the HTAC algorithm (Wang and Yap 2019). Given any TBE (X, C) of a constraint c^*, the constraint graph of (X, C) is a tree. Hence, we can enforce AC on (X, C) by calling various revise functions to update variable domains from O_1 to O_n and back to O_1 where $n = |X|$ and O is an order over X such that $|\{O_j \in N(O_i) | j > i\}| \leq 1$ for each variable $O_i \in X$.

We use three different variable domain representations, including sparse sets (Briggs and Torczon 1993), sparse bit sets (Demeulenaere et al. 2016) and the default domain representations used in the Abscon solver. To exploit efficient bit operations, we select a subset of hidden variables, $H_1 \subset X$, such that for each constraint $c \in C$, $scp(c) \cap H_1 \neq \emptyset$ or $scp(c) \subseteq scp(c^*)$, representing the domains of these variable as sparse bit sets. The domains of the remaining hidden variables, $H_2 = X \setminus H_1$, are represented

as sparse sets. We do not represent all hidden variable domains as sparse bit sets, since our preliminary experiments show that it is better to use sparse sets for the variables in H_2. Note that we only use one kind of domain representations for each hidden variable to avoid the cost of maintaining consistency between different representations. Finally, for the original variables in $scp(c^*)$, we use the default domain representations in the Abscon solver, i.e. both ordered link and bit set. For the TBEs used in this paper, we set H_1 as $\{h \in X | h \notin scp(c^*), N(h) \cap scp(c^*) \neq \emptyset\}$.

As the variable domain representations are different, different revise functions are needed for the propagators of the binary constraints between various variables. For each binary constraint $c \in C$ between 2 variables x, y, if $x \notin H_1$ or $y \notin H_1$, i.e. there is one variable whose domain is not represented as (sparse) bit set, we can directly use the existing revise functions introduced in (Lecoutre and Vion 2008; Wang and Yap 2019), otherwise we apply a new revise function (Algorithm 2) given below for the case when both variables are sparse bit sets.

A revise function for bit set variable domains

We first introduce the data structures used in our new revise function (see Algorithm 2). For any variable x: (i) $bitDom[x]$ is a bit set representing $\mathcal{D}(x)$ and $bitDom[x][i]$ denotes the i^{th} word, where each '1' bit in the word corresponds to a value in $\mathcal{D}(x)$; (ii) $wordDom[x]$ is a sparse set recording the non-ZERO words in $bitDom[x]$; (iii) $bitSup[c, x, i, p]$ is a bit set recording all values in $\mathcal{D}(y)$ which supports the value corresponding to the p^{th} bit in $bitDom[x][i]$, where $scp(c) = \{x, y\}$ and a value $b \in \mathcal{D}(y)$ supports a value $a \in \mathcal{D}(x)$ if $\{(x, a), (y, b)\} \in rel(c)$.

The revise function (as usual) is used to enforce a variable x to be AC on a constraint c, i.e. eliminates values in $\mathcal{D}(x)$ which are not supported by any value in $\mathcal{D}(y)$, where $scp(c) = \{x, y\}$. The outer loop (Line 1), goes over each xwi in $wordDom[x]$ and a word xw records the values in $bitDom[x][xwi]$ whose supports in $\mathcal{D}(y)$ have not been found. The inner loop (Line 2), scans all ywi in $wordDom[y]$ and removes the values, which are supported by a value in the word $yw = bitDom[x][ywi]$, from xw. At Lines 8 and 9, the values in xw can be eliminated from $bitDom[x][xwi]$, since they have not any support in $\mathcal{D}(y)$. A main part of the function is checking which values in xw have supports in yw. If $bitcount(xw) < bitcount(yw)$ where $bitcount$ counting the number of '1' bits in a word, then the function scans all values in xw, using $numberTrailingZeros$ which return the position of the right most '1' bit in the word, and checks whether they have any supports in yw (between Lines 3 and 4), otherwise it scans all values in yw to compute a union w of the values in $bitDom[x][xwi]$ supported by the values (between Lines 5 and 6). At Line 7, w is used to update xw.

Algorithm 2: revise(c, x)

```
   Assume scp(c) = {x, y};
1  for xwi ∈ wordDom[x] do
       xw ← bitDom[x][xwi];
2      for ywi ∈ wordDom[y] do
           w ← 0;
           yw ← bitDom[y][ywi];
           if bitcount(xw) < bitcount(yw) then
3              word ← xw;
               while word ≠ 0 do
                   p ← numberTrailingZeros(word);
                   word ← word & ¬(1 << p);
                   bs ← bitSup[c, x, xwi, p];
                   if (bs[ywi] & yw) ≠ 0 then
4                      w ← w | (1 << p);
           else
5              word ← yw;
               while word ≠ 0 do
                   p ← numberTrailingZeros(word);
                   word ← word & ¬(1 << p);
6                  w ← w | bitSup[c, y, ywi, p][xwi];
7          xw ← xw & ¬w;
           if xw = 0 then
               break;
8      bitDom[x][xwi] ← bitDom[x][xwi] & ¬xw;
9      update wordDom[x];
   return bitDom is changed;
```

6 Experiments

We evaluate the effectiveness of the reduction rules (compression ratio) and efficiency of GAC algorithms in the Abscon solver.[2] We compare the propagators DTBE (GAC on DTBE constraints) and TBE (GAC on the BCT constraints generated by Algorithm 1) with CD (Compact-MDD), a state-of-the-art MDD GAC propagators also using bitwise operations (Verhaeghe, Lecoutre, and Schaus 2018). We tested the algorithms with the binary branching MAC and geometric restart strategy.[3] The variable and value search heuristics used are Activity (Michel and Van Hentenryck 2012) and lexical value order. Experiments were run on a 3.20GHz Intel i7-8700 machine. Total time is limited to 10 minutes per instance and memory to 12G, where total time is the sum of initialization time, transformation time and solving time. We tested with a set of (structured) benchmarks also used by other papers (Cheng and Yap 2010; Gange, Stuckey, and Szymanek 2011; Verhaeghe, Lecoutre, and Schaus 2018):

- Car Sequencing: we use 30 Caroline Gagne hard instances (denoted as C-1) from CSPlib.[4] The problem is modelled with cardinality constraints and sequence constraints, where the sequence constraints are represented

[2] https://www.cril.univ-artois.fr/%7Elecoutre/#/softwares

[3] The initial $cutoff = 10$ and $\rho = 1.1$. For each restart, $cutoff$ is the allowed number of failed assignments and $cutoff$ increases by ($cutoff \times \rho$) after restart.

[4] www.csplib.org

as Binary Decision Diagrams (Cheng and Yap 2010). The MDDs used in C-1 are small. We create harder instances by increasing the block size and block capacity of the C-1 instances by 2 times (and 3 times) to create new instances with larger MDDs denoted as C-2 (and C-3).

- Pentominoes: we use 5 instances (denoted as P-m) from Minizinc Challenge 2020 and 36 instances from the pentominoes generator website.[5] These instances are modelled with regular constraints. We convert the regular constraints to MDD constraints. The instances are separated into 3 series P-10, P-15 and P-20 where P-10 (P-15 or P-20) includes the instances using a 10×10 (15×15 or 20×20) board and 10 (15 or 20) tiles.
- Nurse Scheduling: we use the model 1 and 2, denoted as N-1 and N-2, from (Gange, Stuckey, and Szymanek 2011) where we only convert the regular constraints to MDD constraints. The MDDs used in N-1 and N-2 are small, so we create more challenging N-3 (N-4) instances with lager MDDs by changing the regular constraint of each nurse in N-2 to restrict that the nurse must work 1 or 2 night shifts every 9 (11) days, and 2 or 3 (3 or 4) days off every 7 (9) days, while the nurse can only work a second shift after 12 hours of the first. For each model, we use 50 instances from the N30 series[6] where the number of nurses in each instance is set to the maximum number of nurses required for any day.
- XCSP: we use all 2559 non-binary instances which only employ table constraints from the XCSP website[7] and the methods from (Cheng and Yap 2010; Perez and Régin 2015) to encode tables into MDDs.

Non-table benchmarks

We first evaluate on problems with MDD constraints, the Car Sequence, Pentominoes and Nurse Scheduling Problems. The constraint relations of the MDD constraints used in these benchmarks are very large, and not practically representable as tables, hence, table GAC propagators are not evaluated. We use the MDDc propagator (Cheng and Yap 2010) as a baseline algorithm for these problems.

The results are shown in Table 1. Many instances cannot be solved in 10 minutes, thus, we show the average number of search nodes per second, i.e. the number of search nodes divided by solving time. We highlight that these benchmarks are hard for the MDD solvers as the search speed can be as very low, e.g. 4 nodes/s (MDDc on P-20). The columns labelled #N and #E show the average number of nodes and edges of each MDD constraint. The ESR column is the mean ratio of "the evaluated size of DTBE constraints" to "the evaluated size of TBE constraints". The RT column is the average CPU time (in seconds) of Algorithm 1.

We see the reduction rules work well on these benchmarks. The mean evaluated size of TBE is up to 347 times smaller than that of DTBE. More significantly, ESR increases as MDD constraints' sizes increase, so compression increases with larger MDDs. The DTBE propagator has similar performance to CD, showing that our basic representation is competitive. After rule reduction, the TBE propagator is much faster than other propagators on the problems tested. We can see that the larger the ESR, the faster the TBE propagator. For example, the mean evaluated size of TBE is 255 times (and 91 times) smaller than that of DTBE on the N-4 series (and N-3 series), while TBE can be 85 times (and 6 times) faster than DTBE. In addition, the mean times of Algorithm 1 are acceptable. The transformation time is mostly small. The largest RT time in Table 1 is 11.13s, i.e. the average reduction time of P-20. This is not significant compared to the improvement, e.g. TBE has 246 search nodes/s on P-20 but MDDc only has 4 nodes/s and CD is 15 nodes/s.

	#I	#N	#E	RT	Nodes/s				ESR
					TBE	DTBE	CD	MDDc	
C-1	30	1366	1903	0.03	**1308**	1198	1219	1253	3
C-2	30	16K	23K	0.13	**1730**	1478	1458	708	5
C-3	30	313K	435K	3.43	**814**	538	530	45	5
P-m	5	8366	95K	0.55	**5083**	2114	2423	602	46
P-10	12	7612	56K	0.28	**4392**	1070	1151	580	60
P-15	12	43K	511K	2.21	**1155**	75	81	46	162
P-20	12	137K	2M	11.13	**246**	14	15	4	347
N-1	50	1788	2319	0.01	**11512**	7818	7933	4211	13
N-2	50	1955	2914	0.01	**13174**	8897	9018	3384	26
N-3	50	13K	23K	0.06	**11618**	1827	1837	607	91
N-4	50	91K	161K	0.39	**3728**	44	43	96	255

Table 1: Non-table benchmarks. "#I", "#N", "#E" stand for the number of "instances", "nodes", "edges" and "RT" and "ESR" stand for "reduction time" and "evaluated size ratio".

Table benchmarks

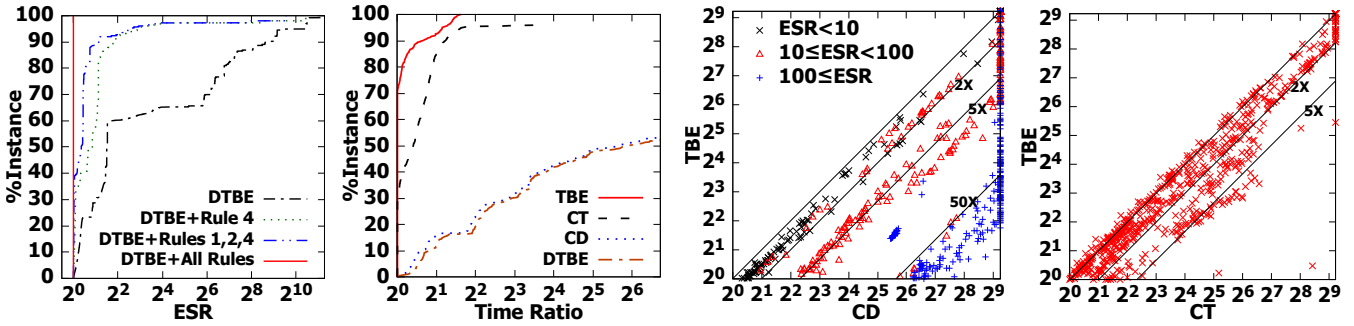
We now evaluate table constraint benchmarks. The evaluation with non-table benchmarks was for ad-hoc constraints beyond the reach of tables. We now evaluate benchmarks when table or MDD constraints are both feasible representations. We use CT (Demeulenaere et al. 2016) as the baseline algorithm as it is a state-of-the-art table GAC algorithm. CT has been shown to overall outperform MDD GAC algorithms on table constraint benchmarks (Verhaeghe, Lecoutre, and Schaus 2018, 2019).

Experimental results are shown in Figure 4. We compare the reduction rules and GAC propagators with performance profiles (Dolan and Moré 2002) in Figures 4a and 4b. The various (colored) lines in 4a "DTBE+*some Rules*" compare the TBE generated with a different subset of rules in Algorithm 1, e.g. "DTBE+Rule 4" means only Rule 4 is used. Every dot (α, β) in Figure 4a (Figure 4b) denotes that there are β percentage instances such that the ratio of "the evaluated size (the total time) of a TBE constraint (a propagator)" to "the evaluated size (the total time) of the *virtual best TBE constraint* (the *virtual best propagator*)" is up to α. The total time includes the time of encoding table constraints as MDDs and BCTs. In Figure 4b, we remove: (i) the instances that cannot be solved by TBE, DTBE, CT or CD within 10 minutes (timeout); and (ii) the instances on which the solving times of CT and TBE are both less than 2 seconds (triv-

[5] https://github.com/zayenz/minizinc-pentominoes-generator

[6] https://www.projectmanagement.ugent.be/nsp.php

[7] http://xcsp.org



(a) Evaluated size (b) Total time (c) Solving time: CD vs TBE (d) Solving time: CT vs TBE

Figure 4: Results of table benchmarks.

ial). We use scatter plots to compare the solving time of TBE with CD and CT. Each dot in Figures 4c and 4d denotes an instance, while the diagonal lines with the label 2X, 5X or 50X mean that the TBE algorithm is 2X, 5X or 50X faster than CD or CT.

Figure 4a shows the effectiveness of the rules in reducing the size of the TBE with "DTBE+All Rules" being the most effective. Figure 4b with the time ratio shows that the DTBE propagator is competitive with CD, while the TBE propagator is the fastest propagator on 71% of instances. TBE can be up to 12 (98) times faster than the CT (CD) algorithms. We found that the more the reduction rules reduce the TBE (larger ESR), the greater the speedup of TBE over CD. The TBE constraints generated using all rules can be up to 2166 times smaller than the original DTBE constraints. The solver performance is shown in Figure 4c, we can see the solving time ratio between TBE and CD is consistent with ESR. On most instances with "100 $\leq$ESR", the TBE can be 50X faster than CD. We turn to the CT comparison, Figure 4d, TBE is faster than CT on most of the instances and only slightly slower on a few. The average CPU time of Algorithm 1 is only 0.43s, thus, TBE can overall outperform the CT algorithm for non-trivial instances. We highlight that the results in (Verhaeghe, Lecoutre, and Schaus 2018, 2019) show CT outperforms CD for table benchmarks, while we outperform both CD and CT.

7 Related work

Cheng and Yap proposed the first MDD GAC algorithm MDDc (Cheng and Yap 2010) using a depth-first search strategy. Then various incremental algorithms, such as incremental MDD (Gange, Stuckey, and Szymanek 2011) and MDD4R (Perez and Régin 2014), have been proposed to avoid redundant traversal. Comparing to table GAC algorithms, MDD GAC algorithms can be more efficient when the MDD constraints are compact, but there is a large gap between the MDD representation's compression ratio and when it outperforms table propagators. Recent works show table GAC algorithms using bitwise operations can overall outperform MDD GAC algorithms on a large set of benchmarks (Wang et al. 2016; Demeulenaere et al. 2016).

In order to reduce the gap, Verhaeghe et al. proposed the MDD GAC CD algorithm (Verhaeghe, Lecoutre, and Schaus 2018) using ideas from CT. The CD algorithm has been also extended with some alternative representations of MDDs, such as semi-MDDs (Verhaeghe, Lecoutre, and Schaus 2018) and basic smart MVDs (Verhaeghe, Lecoutre, and Schaus 2019). Although CD and its variants can be faster than the MDD4R algorithm and reduce the gap, CT still overall outperforms CD and its variants on a large set of benchmarks (see the experimental results in (Verhaeghe, Lecoutre, and Schaus 2018, 2019)). The BCT GAC propagator proposed in this paper is the first algorithm, enforcing GAC on MDD constraints, which is shown to overall outperform the state-of-the-art table GAC algorithm CT.

We are not aware of any binary encoding for MDD constraints. The most related work is the HVE encoding of table constraints. As shown in (Wang and Yap 2019, 2020), the state-of-the-art AC propagator HTAC on HVE instances has similar performance to CT. Recently, the table constraint binary encoding BE (Wang and Yap 2020) also overall outperforms CT. However, BE changes the constraint structures and consistency levels, i.e. it is more than a table GAC propagator. Furthermore, the MDD constraint covers the scenarios when we need ad-hoc constraints but tables are not practically feasible.

8 Conclusion

Ordered Multi-valued Decision Diagram (MDD) is more compact than its table representation, but the state-of-the-art table GAC propagator CT still overall outperforms MDD GAC propagators. In order to reduce the large gap between the MDD representation's compression ratio and efficiency of MDD GAC propagators, we propose a new representation of MDD constraints called BCT. BCTs can be exponentially smaller than MDDs when representing some constraints. We first use DTBE to encode any MDD as a BCT constraint, and then reduce the BCT constraint by eliminating, merging and reconstructing some hidden variables. We also give a BCT GAC propagator using bitwise operations to enforce GAC on the MDD constraints encoded as BCT constraints. Experimental results on a large set of benchmarks show that BCT constraints are very compact and BCT GAC propagator outperforms the state-of-the-art MDD GAC propagator CD and table GAC propagator CT.

Acknowledgements

This work has been supported by grant R-252-000-B58-114.

References

Beldiceanu, N.; and Contejean, E. 1994. Introducing global constraints in CHIP. *Mathematical and computer Modelling*, 20(12): 97–123.

Bessiere, C.; Stergiou, K.; and Walsh, T. 2008. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 172(6-7): 800–822.

Briggs, P.; and Torczon, L. 1993. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4): 59–69.

Cheng, K.; and Yap, R. H. C. 2010. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2): 265–304.

Cheng, K. C.; and Yap, R. H. 2006. Applying Ad-hoc Global Constraints with the case Constraint to Still-Life. *Constraints*, 11(2-3): 91–114.

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

Demeulenaere, J.; Hartert, R.; Lecoutre, C.; Perez, G.; Perron, L.; Régin, J.-C.; and Schaus, P. 2016. Compact-Table: efficiently filtering table constraints with reversible sparse bit-sets. In *International Conference on Principles and Practice of Constraint Programming*, 207–223.

Dolan, E. D.; and Moré, J. J. 2002. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2): 201–213.

Gange, G.; Stuckey, P. J.; and Szymanek, R. 2011. MDD propagators with explanation. *Constraints*, 16(4): 407.

Katsirelos, G.; and Walsh, T. 2007. A compression algorithm for large arity extensional constraints. In *International Conference on Principles and Practice of Constraint Programming*, 379–393.

Lecoutre, C.; and Vion, J. 2008. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters*, 2: 21–35.

Michel, L.; and Van Hentenryck, P. 2012. Activity-based search for black-box constraint programming solvers. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 228–243.

Perez, G.; and Régin, J.-C. 2014. Improving GAC-4 for table and MDD constraints. In *International Conference on Principles and Practice of Constraint Programming*, 606–621.

Perez, G.; and Régin, J.-C. 2015. Efficient operations on MDDs for building constraint programming models. In *International Joint Conference on Artificial Intelligence*, 374–380.

Pesant, G. 2004. A regular language membership constraint for finite sequences of variables. In *International Conference on Principles and Practice of Constraint Programming*, 482–495.

Rossi, F.; Petrie, C. J.; and Dhar, V. 1990. On the Equivalence of Constraint Satisfaction Problems. In *European Conference on Artificial Intelligence*, 550–556.

Srinivasan, A.; Ham, T.; Malik, S.; and Brayton, R. K. 1990. Algorithms for discrete function manipulation. In *IEEE/ACM International Conference on Computer-Aided Design*, 92–95.

Stergiou, K.; and Walsh, T. 1999. Encodings of Non-Binary Constraint Satisfaction Problems. In *AAAI Conference on Artificial Intelligence*, 163–168.

Verhaeghe, H.; Lecoutre, C.; and Schaus, P. 2018. Compact-MDD: Efficiently Filtering (s)MDD Constraints with Reversible Sparse Bit-sets. In *International Joint Conference on Artificial Intelligence*, 1383–1389.

Verhaeghe, H.; Lecoutre, C.; and Schaus, P. 2019. Extending Compact-Diagram to Basic Smart Multi-Valued Variable Diagrams. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 581–598.

Wang, R.; Xia, W.; Yap, R. H. C.; and Li, Z. 2016. Optimizing Simple Tabular Reduction with a Bitwise Representation. In *International Joint Conference on Artificial Intelligence*, 787–795.

Wang, R.; and Yap, R. H. C. 2019. Arc Consistency Revisited. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 599–615.

Wang, R.; and Yap, R. H. C. 2020. Bipartite Encoding: A New Binary Encoding for Solving Non-Binary CSPs. In *International Joint Conference on Artificial Intelligence*, 1184–1191.

Yap, R. H. C.; Xia, W.; and Wang, R. 2020. Generalized Arc Consistency Algorithms for Table Constraints: A Summary of Algorithmic Ideas. In *AAAI Conference on Artificial Intelligence*, 13590–13597.