Encoding Multi-Valued Decision Diagram Constraints as Binary Constraint Trees

Ruiwei Wang and 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.

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.
Background

A CSP $P$ is a pair $(X, C)$ where $X$ is a set of variables, $D(x_i)$ is 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 $\{x_{i_1}, x_{i_2}, \ldots, x_{i_n}\}$ is denoted by a set of literals $\{(x_{i_1}, a_{i_1}), (x_{i_2}, a_{i_2}), \ldots, (x_{i_r}, a_r)\}$. 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 $\tau$, we use $\tau[V] = \{(x, a) \in \tau|x \in V\}$ to denote a subset of $\tau$, while $T[V] = \{\tau[V]|\tau \in T\}$ is the projection of tuples $T$ on $V$. A tuple $\tau$ over $X$ is a solution of $P$ if $|scp(c)| = rel(c)$ for all constraints $c \in C$ and all $a \in D(x)$ for $\forall (x, a) \in \tau$. $sol(X, C)$ is the solution of a constraint $c$. 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, $(\bigcup_{i=1}^{r+1} L_i, \bigcup_{i=1}^{r+1} E_i, label)$, 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 $D(O_i)$;
- there is at most 1 edge $e \in E_i$ such that $out(e) = n_j$ and $label(e) = l$ for each value $l \in D(O_i)$ 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 $\tau$ in $rel(c^*)$ corresponds to a path from root to 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 Régis 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 $|D(O_i)|$ edges pointing from $n_j$ to the same node.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 order is from $x_1$ to $x_4$. The meaning of this constraint is to require that for any 2 consecutive variables, the value 1 is assigned to at most 1 variable.

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 such a BCT $P = (X, C)$ such that $scp(c^*)$ 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 $dtbe(c^*) = (Y \cup H \cup scp(c^*), \{c_0^+, c_1^+, c_2^+, \ldots, c_r^+, c^+_r\})$ where

1Many MDD GAC propagators, such as MDD4R (Perez and Régis 2014) and CD, also do not require removing redundant nodes.
• $r = |scp(c^*)|$ and $O$ is an order over $scp(c^*)$;
• $Y = \{y_1, \ldots, y_{n_1}\}$, $H = \{h_1, \ldots, h_{n_1}\}$ and $scp(c^*) = \{O_i, h_i\}$, $scp(c_1^*) = \{y_{n_1+1}, h_{n_1}\}$, $scp(c_i^*) = \{y_i, h_i\}$;
• $D(y_1)$ is the set of nodes in the $i$th layer of $mdd(c^*, O)$;
• $D(h_1)$ is the set of edges which point from the nodes in the
  $i$th layer of $mdd(c^*, O)$ to the next layer;
• $rel(c_i^*) = \{(h_i, a), (y_{i+1}, b) | a \in D(h_i), b = \text{out}(a)\}$;
• $rel(c_i^*) = \{(h_i, a), (O_i, l), l = \text{label}(a)\}$;
• $rel(c_i^*) = \{(h_i, a), (y_i, b) | a \in D(h_i), b = \text{in}(a)\}$.

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. $D(y_1) = \{n_1\}$
is the set of the nodes in the first layer of the MDD
and $D(h_1) = \{c_1, c_2\}$ 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.
$rel(c_i^*) = \{(h_i, c_1^i), (x_i, 1), (h_i, c_2^i), (x_i, 1)\}$,
$rel(c_i^*) = \{(h_i, c_1^i), (y_i, 1), (h_i, c_2^i), (y_i, 1)\}$,
$rel(c_i^*) = \{(h_i, c_1^i), (y_i, n_2), (h_i, c_2^i), (y_i, n_2)\}$. We can see that the constraint graph is a tree and every
solution of $dtbe(c^*)$ denotes a path in the MDD which
represents a variable 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 $X = \{x_1, \ldots, x_r\}$ with domains $D = \{1, \ldots, r\}$
and $r > 2$ expresses the relation $\bigvee_{i=1}^{r} \bigvee_{j=i+1}^{r} (x_i = x_j)$. The
constraint in this example is used to show that BCTs can be
very expressive, exponentially smaller than MDDS on
representing some constraints. This constraint expresses the
negation of the all-different 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 $C = \{c_1, \ldots, c_r\}$ and $D = \{a_{ijk} | 1 \le j, k \le r, j \neq k \}$ and for $1 \le i \le r$, $scp(c_i) = \{x_i, h\}$ and
$rel(c_i) = \{(h, a_{ijk}), (x_i, a) | i \neq j, k, i \neq l \in D, a \in D\}$.
For any tuple $\tau$ over $X \cup \{h\}$, $\tau$ is in the constraint relation, i.e.
there exists variables $x_i, x_j \in X$ and a value $a \in D$ such
that $\{x_i, x_j\} \subset \tau$, if and only if $\tau \cup \{(h, a_{ij})\}$ is
a solution of $(V, C)$. So, BCT can be exponentially smaller
than MDD when representing this constraint.

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 8) 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 rep-
resentation more compact. We propose four reduction rules
to reduce any TBE $P_1 = (X_1, C_1)$ of a constraint $c^*$.

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 $[b \in D(y) | (x, b) \in rel(c)]$. A
tuple over $X_1 \setminus \{x\}$ is a solution of $R_1(P_1, x)$ if 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 $c_1, c_2 \in C_1$. Assume
$scp(c_1) = \{x, y\}$ and $scp(c_2) = \{x, z\}$. Let
$R_2(P_1, x) = (X_1 \setminus \{x\}, C \cup \{c\})$ be the CSP
generated by eliminating $x$ where $C = C_1 \cup \{c_1, c_2\}$ and
$scp(c') = \{z, y\}$ and $rel(c') = \{\{x, y, z\}, \{c_1, c_2\}\} \{y, z\}$. Similarly, a
tuple over $X_1 \setminus \{x\}$ is a solution of $R_2(P_1, x)$ if 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
$P_{2a} = R_1(R_1(P_1, y_1), y_5)$ generated by applying Rule 1
is given in Figure 2a. Then five variables $h_1, h_3, y_2, y_3, y_4$ in
Figure 2a can be eliminated by Rule 2. The reduced
TBE $P_{2b} = R_2(R_2(R_2(R_2(P_2a, h_1), h_3), y_2), y_3)$ is shown in Figure 2b, where $c_1^a, c_2^a, c_3^a$ are the new
constraints generated. The relations of the new constraints can be
directly constructed by Rule 2, e.g. $rel(c_1^a) = \{(x, e_3, e_6) >, \}
< e_4, e_7 >, < e_4, e_8 >, < e_5, e_7 >, < e_5, e_8 >$ where
$< e_4, e_7 >$ denotes the tuple $\{(h_2, c_1^a), (h_3, c_1^a)\}$.

A Hidden Variable Merging Rule

The third rule merges any 2 hidden variables $x, y$ which are
constrained by a constraint $c \in C_1$, i.e. $scp(c) = \{x, y\}$,
as a new variable $m$ such that $D(m) = rel(c)$, 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 $c_j'$
$scp(c_j') = \{z, m\}$ and $rel(c_j') = \{(z, a), (m, \tau) \ | \ \tau \in \in rel(c), \tau \cup \{(z, a)\} \in sol(\{x, y, z\}, \{c_1, c_2\})\}$. We can get a new CSP
$R_3(P_1, c_1) = (X \cup \{m\}, C \cup C')$ where
$X = X_1 \setminus \{x, y\}$ and $C = \{c \in C_1 | x \notin scp(c), y \notin scp(c)\}$.
Lemma 4

new binary constraint \( R \)

Lemma 3 (Rule 3).

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 \( \tau \) 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 \( \tau \) over variables \( V \) represents a set of tuples denoted as \( T(\tau) = \{ \tau' \subseteq \tau \mid \tau' \text{ is a tuple over } V \} \). A c-table is a set of c-tuples. A c-table \( rel_l \) represents a set of tuples \( rel_l \) if the c-tuples in \( rel_l \) represent the tuples in \( rel_l \), i.e., \( \cup_{\tau \in rel_l} T(\tau) = rel_l \).

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 \( ct(P_1, x) \) is an \( t^o \) \( \tau \in D(x) \) where \( t^o = \{(y, b) \mid y \in scp(c), c \in NC(x), \{(y, b), (x, a) \} \in rel(c) \} \) is a c-tuple. For any constraint in \( NC(x), t^o \) includes all literals \((y, b) \) such that \( \{(y, b), (x, a) \} \in rel(c) \). Therefore, for each solution \( \tau \) of \( P_1 \), if there is a tuple \((x, a) \) in \( \tau \), then the tuple \( \tau(N^-(x)) \) is a subset of \( t^o \), otherwise there is a constraint \( c \in NC(x) \) such that \( \tau[scp(c)] \notin rel(c) \).

For example, \( ct(P_2b, h_3) \) is a CC-T including 3 c-tuples \( t^o, t^r, t^s \), where \( t^o = \{(h_2, e_3), (x_3, 0), (x_4, 0), (x_4, 1)\} \), \( t^r = \{(h_2, e_3), (h_2, e_5), (x_3, 0), (x_4, 0), (x_4, 1)\} \), and \( t^s = \{(h_2, e_4), (h_2, e_3), (x_3, 1), (x_4, 0), (x_4, 0)\} \). We can see that for any solution \( \tau \) of \( P_2b \) including the literal \((h_3, e_6) \), the tuple \( \tau(\{h_2, x_3, h_4\}) \) is a subset of the c-tuple \( t^o \).

Conversely, given any CC-T \( rel_l \) of \( x \), we can reconstruct \( x \) into a new variable \( x' \) such that \( D(x') = rel_l \) and replace each constraint \( c \) between \( x \) and another variable \( y \) with a new binary constraint \( c' \) such that \( scp(c') = \{x', y\} \) and \( rel(c') = \{((x', \tau'), (y, a)) \mid \tau' \in D(x'), (y, a) \in \tau' \} \). Correspondingly, we can get a new CSP \( R_4(P_1, x, rel) = (X \cup \{x\}, C' \cup C') \) where \( X = X_1 \setminus \{c\} \) and \( C' = \{c' \mid c \in C_1, x \in scp(c) \} \) 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 \( \tau' \in D(x') \) such that the tuple \( \tau \cup \{(x', \tau')\} \) is a solution of \( R_4(P_1, x, rel) \).
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 |D(x) × 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, 3a and 3b are 66, 56, 33, and 20, respectively. We can see that P3a is the smallest TBE. The evaluated size of P3a is 3.3 times smaller than that of the original DTBE given in Figure 1b. In addition, size(P3b) is greater than size(P3a), so we do not use Rule 3 to merge the hidden variables m1 and m2 in P3a.

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 τ1 and τ2 in a CC-T rel1 of x are mergeable w.r.t. a variable y ∈ N−(x) if τ1 \ τ2 (or τ2 \ τ1) is empty or only includes some literals of y, since T(τ1) ∪ T(τ2) = T(τ1 \ τ2). This is an equivalence relation defining a partition par(rel1, y) of rel1 where all c-tuples in the same subset S are mergeable w.r.t. y. We use merge(rel1, y) = \bigcup_{S ∈ S} S ∈ par(rel1, y) to denote a c-table generated by merging c-tuples with par(rel1, y). Obviously, merge(rel1, 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 ∈ N−(x) \ \{v\}} |D(y)| < |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. We remark that it is NP-hard to construct the optimal TBE by reducing a DTBE by using the rules. 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 ∈ 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 c1, c2 in C such that |D(y)| × |D(z)| is not greater than |D(x)| × (|D(y)| + |D(z)|) where scp(c1) = {x, y} and scp(c2) = {x, z}.
- **ApplyRule3(X, C)** is called at Line 6. It repeatedly scans the hidden variables from O1 to O2 and back to O1 until Rule 3 cannot be applied to merge any variables, where k is the number of hidden variables. For any hidden variable Oi, if there is a constraint c ∈ C between Oi and another hidden variable Oj such that the evaluated size of R2((X, C), c) is not greater than that of (X, C), then Oi and Oj 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. 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).

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 O1 to O_k and back to O1 where |{O_j ∈ N(O_i) | j > i}| ≤ 1 for each variable O_i ∈ 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, H1 ⊂ X, such that for each constraint c ∈ C, scp(c) \ H1 ≠ ∅ or scp(c) ⊆ scp(c∗), representing the domains of these variable as sparse bit sets. The domains of the remaining hidden variables, H2 = X \ H1, 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 \(y \notin H_2\), 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 \(D(x)\) and \(bitDom[x][i]\) denotes the \(i^{th}\) word, where each ‘1’ bit in the word corresponds to a value in \(D(x)\); (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 \(D(y)\) which supports the value corresponding to the \(p^{th}\) bit in \(bitDom[x][i]\), where \(scp(c) = \{x, y\}\) and a value \(b \in D(y)\) supports a value \(a \in D(x)\) 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 \(D(x)\) which are not supported by any value in \(D(y)\), if \(scp(c) = \{x, y\}\). The outer loop (Line 1), goes over each \(xwi\) in \(wordDom[x]\) and a word \(yw\) records the values in \(bitDom[x][xwi]\) whose supports in \(D(y)\) have not been found. The inner loop (Line 2), scans all \(yw\) in \(wordDom[y]\) and removes the values, which are supported by a value in the word \(yw = bitDom[x][xwi]\) from \(yw\). At Lines 8 and 9, the values in \(yw\) can be eliminated from \(bitDom[x][xwi]\), since they have not any support in \(D(y)\). A main part of the function is checking which values in \(yaw\) have supports in \(yw\). If \(bitcount(yw) < bitcount(yw)\) where \(bitcount\) counting the number of ‘1’ bits in a word, then the function scans all values in \(yw\), 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 \(xw\).

### Experiments

We evaluate the effectiveness of the reduction rules (compression ratio) and efficiency of GAC algorithms in the Ab-
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. 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 $10 \times 10$ ($15 \times 15$ or $20 \times 20$) 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 larger 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 (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 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 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 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.

### 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 ($\alpha$, $\beta$) in Figure 4a (Figure 4b) denotes that there are $\beta$ 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 $\alpha$. 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-

<table>
<thead>
<tr>
<th>#I</th>
<th>#N</th>
<th>#E</th>
<th>RT</th>
<th>Nodes/s</th>
<th>ESR</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-1</td>
<td>30</td>
<td>1366</td>
<td>1903</td>
<td>0.03</td>
<td>1308</td>
</tr>
<tr>
<td>C-2</td>
<td>30</td>
<td>16K</td>
<td>23K</td>
<td>0.13</td>
<td>1730</td>
</tr>
<tr>
<td>C-3</td>
<td>30</td>
<td>313K</td>
<td>435K</td>
<td>3.43</td>
<td>814</td>
</tr>
<tr>
<td>P-m</td>
<td>5</td>
<td>8366</td>
<td>95K</td>
<td>0.55</td>
<td>5083</td>
</tr>
<tr>
<td>P-10</td>
<td>12</td>
<td>7612</td>
<td>56K</td>
<td>0.28</td>
<td>4392</td>
</tr>
<tr>
<td>P-15</td>
<td>12</td>
<td>43K</td>
<td>511K</td>
<td>2.21</td>
<td>1155</td>
</tr>
<tr>
<td>P-20</td>
<td>12</td>
<td>137K</td>
<td>2M</td>
<td>11.13</td>
<td>246</td>
</tr>
<tr>
<td>N-1</td>
<td>50</td>
<td>1788</td>
<td>2319</td>
<td>0.01</td>
<td>11512</td>
</tr>
<tr>
<td>N-2</td>
<td>50</td>
<td>1955</td>
<td>2914</td>
<td>0.01</td>
<td>13174</td>
</tr>
<tr>
<td>N-3</td>
<td>50</td>
<td>13K</td>
<td>23K</td>
<td>0.06</td>
<td>11618</td>
</tr>
<tr>
<td>N-4</td>
<td>50</td>
<td>91K</td>
<td>161K</td>
<td>0.39</td>
<td>3728</td>
</tr>
</tbody>
</table>

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”.

---

1[https://github.com/zayenz/minizinc-pentominoes-generator](https://github.com/zayenz/minizinc-pentominoes-generator)
3[http://xcsp.org](http://xcsp.org)
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 ≤ 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. Comparing with existing results, the results in (Verhaeghe, Lecoutre, and Schaus 2018, 2019) show that CD is outperformed by CT for table benchmarks. In contrast, our results here outperform both CD and CT on table benchmarks.

Related Work
Cheng and Yap proposed the first MDD GAC algorithm MDDc (Cheng and Yap 2010) using a depth-first search strategy. Later, various incremental algorithms, such as incremental MDD (Gange, Stuckey, and Szymanek 2011) and MDD4R (Perez and Régis 2014), were 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 algorithm CD (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.

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 was supported in part by the National Research Foundation Singapore under its AI Singapore Programme [AISG-RP-2018-005] and grant T1 251RES2024

References