Extending Compact-Diagram to Basic Smart Multi-Valued Variable Diagrams

Hélène Verhaeghe[1], Christophe Lecoutre[2], and Pierre Schaus[1]

[1] UCLouvain, ICTEAM, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium,
{firstname.lastname}@uclouvain.be
[2] CRIL-CNRS UMR 8188, Université d'Artois, F-62307 Lens, France,
lecoutre@cril.fr

Abstract. Multi-Valued Decision Diagrams (MDDs), and more generally Multi-Valued Variable Diagrams (MVDs), are instrumental in modeling constrained combinatorial problems. This has led to a number of algorithms for filtering constraints such as mddc, MDD4R and CD (Compact-Diagram). Many compressed forms of tables have also been proposed over the years, leading to a 'smart' hybridization between extensional an intentional representations, which was obtained by embedding simple arithmetic constraints in tuples (of tables). Interestingly, the state-of-the-art algorithm CT (Compact-Table) has been recently extended to deal efficiently with *bs*-tables, i.e., 'basic smart' tables containing expressions of the form '$*$', '$\neq v$', '$\leq v$', '$\geq v$' and '$\in S$'. In this paper, we introduce the concept of *bs*-MVDs by enabling arcs of diagrams to be labelled with similar expressions. We show how such diagrams can be naturally derived from ordinary tables and MDDs, and we extend the state-of-the-art algorithm CD in order to handle *bs*-MVDs (and *bs*-MDDs).

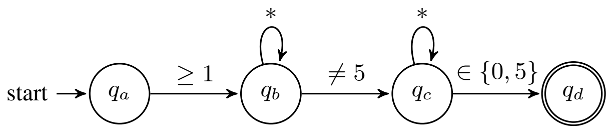
Keywords: Multi-Valued Decision Diagrams, Filtering, Compression, Compact-Table, Bitset

1 Introduction

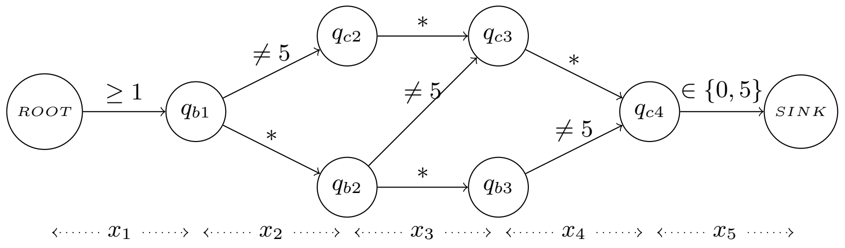
Efficiently representing constraints under extensional forms such as tables and decision diagrams has been a hot research topic for the last decade; concerning MDDs (Multi-Valued Decision Diagrams), see e.g., [1, 2, 4, 5, 13–15, 28]. Actually, two main lines of improvements have been followed when handling extensional forms of constraints. Firstly, high effective filtering techniques have been proposed over the years, such as those based on tabular reduction [18, 19] and bitwise operations [11, 16, 32]. Secondly, compact representation techniques have been intensively studied, mainly by allowing simple constraints to be put in tables as in [17, 20, 30, 31] or by directly using decision diagrams such as MDDs [10, 22, 23].

CD (Compact-Diagram), previously called Compact-MDD, has been recently introduced [29] for Multi-Valued Variable Diagrams (MVDs), which are diagrams generalizing MDDs by authorizing non-determinism. By combining several ideas and techniques, in particular, those proposed in [30, 29], we show how a bitwise filtering algorithm can be conceived for constraints represented by *bs*-MVDS, which are 'basic smart' MVDs accepting arcs to be labelled with unary constraints. These *bs*-MVDs can be seen as a very promising modeling tool.

As a direct application of *bs*-MVDs, we find the possible compact representation of regular constraints [3, 6, 25], imposing that a specified sequence of variables must be accepted by a given automaton (derived from a regular expression). As a user, it is quite natural to express an automaton using expressions on transitions, as illustrated in Fig. 1a for the regular language $[1-9].^*[\hat{}5].^*[05]$. As described in [24], a layered deterministic automaton graph basically defines an MDD constraint; a layered automaton graph obtained from a non-deterministic automaton then naturally leads to an MVD constraint. Figure 1b displays the *bs*-MVD that corresponds to the unfolding of the automaton over a sequence of 5 variables.



(a) Automaton



(b) MVD

Fig. 1: The regex $[1-9].^*[\hat{}5].^*[05]$ describes a number divisible by 5 with at least one of its inner digits being different from 5.

The main contributions of this paper are:

- Different strategies to create *bs*-MVD constraints from table constraints, with improved compression compared to classical MDDs.
- An efficient bitwise filtering algorithm that enforces the property known as Generalized Arc Consistency on *bs*-MVD constraints.
- An experimentation that demonstrates the practical interest of our approach.

2 Technical Background

A *constraint network* is composed of a set of variables and a set of constraints. Each *variable* x has an associated (ordered) domain $dom(x)$ containing the values that can be assigned to it; the *current* domain is included in the *initial* domain $dom^0(x)$. We respectively denote by $min(x)$ and $max(x)$ the smallest and greatest values in $dom(x)$. Each *constraint* c involves an ordered set of variables, called the *scope* of c and denoted by $scp(c)$, and is semantically defined by a *relation* $rel(c)$ containing the tuples allowed for the variables involved in c. The *arity* of a constraint c is $|scp(c)|$. When the domain of a variable x is (becomes) singleton, we say that x is *bound*.

Given a sequence $\langle x_1, \dots, x_r \rangle$ of r variables, an r-tuple τ on this sequence of variables is a sequence of r values $\langle a_1, \dots, a_r \rangle$, where the individual value a_i is also denoted by $\tau[x_i]$. An r-tuple τ is *valid* on an r-ary constraint c iff $\forall x \in scp(c), \tau[x] \in dom(x)$, and τ is *allowed* by c iff $\tau \in rel(c)$. A *support* of c is a tuple which is both valid on c and allowed by c. A *literal* is a pair (x, a) where x is a variable and a a value. A literal (x, a) is *Generalized Arc-Consistent* (GAC) on c iff there is a support τ on c such that $\tau[x] = a$. A constraint c is GAC iff any literal (x, a) such that $x \in scp(c)$ and $a \in dom(x)$ is GAC on c.

A directed graph is composed of a set of nodes and a set of arcs. Each arc has an orientation from one node, the *tail* of the arc, to another node, the *head* of the arc. For a given node ν, the set of arcs with ν as tail (resp., head) is called the set of *outgoing* (resp., *incoming*) arcs of ν. A labelled directed graph is a directed graph such that a label $l(\omega)$ is associated with each arc ω. A node is *in-d* (in-deterministic) iff it does not have two incoming arcs with the same label, *in-nd* otherwise. A node is *out-d* (out-deterministic) iff no two outgoing arcs have the same label, *out-nd* otherwise. A directed acyclic graph (DAG) is a directed graph with no directed cycles. An MVD (Multi-valued Variable Diagram) [1] *for* a constraint c (called an MVD constraint) is a layered DAG, with one special root node at level 0, denoted by `ROOT`, r layers of arcs, one layer $L(x_i)$ for each variable x_i of the scope $\langle x_1, \dots, x_r \rangle$ of c, and one special sink node at level r, denoted by `SINK`. The arcs in $L(x_i)$ going from level $i - 1$ to level i are *on* the variable x_i: any such arc is labelled by a value in $dom^0(x_i)$. A *valid path* in such an MVD is a path p from the root to the sink such that for each variable x_i in $scp(c)$ the label of the arc going in p from level $i - 1$ to i is a value in $dom(x_i)$. The set of supports of an MVD constraint c corresponds to the valid paths in the MVD for c. One classical type of MVD is the Multi-valued Decision Diagram (MDD) [8], which guarantees that each node is out-d (each node at level i has at most $|dom^0(x_i)|$ outgoing arcs, labelled with different values), but possibly in-nd. Another type of MVD is the semi-MDD (sMDD) [29] which guarantees that each node at a level $< \lfloor \frac{r}{2} \rfloor$ is out-d and each node at a level $> \lfloor \frac{r}{2} \rfloor + 1$ is in-d.

CD. Compact-Diagram, or CD (previously called Compact-MDD [29]), is a filtering algorithm (propagator) that uses bitwise operations for MVD constraints. It is based on some ideas behind both Compact-Table [11], a propagator for table constraints that uses bitwise operations as well, and MDD4R [23], a propagator for MDD constraints.

The idea is to keep track of the arcs that remain valid during the filtering process; namely by introducing (reversible sparse) bitsets, one per layer of the MVD (and so, per variable of the constraint). At layer i, one bit, in the bitset `currArcs`$[x_i]$, is associated with each arc: when the bit is set to 1, it means that the arc is considered as valid. This way, the current MVD, which can be seen as a subgraph of the initial MVD, can be identified, and used to remove the values without any supports left.

Algorithm 1: Class ConstraintCD

```
1  Method enforceGAC()
2      updateGraph()
3      filterDomains()
4  Method updateGraph()
5      foreach variable $x \in$ scp do
6          mask[$x$] $\leftarrow 0^{64}$
7      updateMasks()
8      propagateDown($x_1$, false)
9      propagateUp($x_r$, false)
10 Method updateMasks()
11     foreach variable $x \in \{x \in scp : |\Delta_x| > 0\}$ do
12         if $|\Delta_x| < |dom(x)|$ then                          // Incremental update
13             foreach value $a \in \Delta_x$ do
14                 mask[$x$] $\leftarrow$ mask[$x$] | supports[$x, a$]          // bitwise OR
15         else                                                      // Reset-based update
16             foreach value $a \in dom(x)$ do
17                 mask[$x$] $\leftarrow$ mask[$x$] | supports[$x, a$]          // bitwise OR
18             mask[$x$] $\leftarrow \sim$ mask[$x$]                          // bitwise NOT
19 Method propagateDown($x_i$, localChange)
20     if $|\Delta_{x_i}| > 0$ or localChange then
21         currArcs[$x_i$] $\leftarrow$ currArcs[$x_i$] & $\sim$ mask[$x_i$]
22         if currArcs[$x_i$] $= 0$ then
23             throw Backtrack
24         if $x_i \neq x_r$ then
25             localChange $\leftarrow$ false
26             foreach node $\nu \in \{\nu$ : currArcs[$x_{i+1}$] & arcsT[$\nu, x_{i+1}$] $\neq 0^{64}\}$ do
27                 if currArcs[$x_i$] & arcsH[$x_i, \nu$] $= 0^{64}$ then
28                     mask[$x_{i+1}$] $\leftarrow$ mask[$x_{i+1}$] | arcsT[$\nu, x_{i+1}$]
29                     localChange $\leftarrow$ true
30             propagateDown($x_{i+1}$, localChange)
31     else if $x_i \neq x_r$ then
32         propagateDown($x_{i+1}$, false)
33 Method propagateUp($x_i$, localChange)
       /* Similar to propagateDown with $x_1$ instead of $x_r$, $x_{i-1}$ instead of $x_{i+1}$,
       inverted use of arcsT and arcsH.                                          */
34 Method filterDomains()
35     foreach variable $x \in \{x \in scp : |dom(x)| > 1\}$ do
36         foreach value $a \in dom(x)$ do
37             if currArcs[$x$] & supports[$x, a$] $= 0^{64}$ then        // bitwise AND
38                 $dom(x) \leftarrow dom(x) \setminus \{a\}$
```

To ease computations, at each level there are three types of precomputed bitsets: these bitsets are never modified. First, $\texttt{supports}[x, a]$ indicates for each arc on the variable x whether or not the value a is initially supported by this arc (bit is set to 1 iff a is supported). Second, $\texttt{arcsT}[\nu, x]$ and $\texttt{arcsH}[x, \nu']$ indicate for each arc on x whether ν and ν' are respectively the tail and the head of this arc. Finally, a temporary bitset $\texttt{mask}[\texttt{x}_\texttt{i}]$ is associated with each variable x_i to store the results of intermediate computations.

The pseudo-code for enforcing GAC on an MVD constraint is given by Algo. 1, which is, for simplicity, a simplified version of the one given in [29]. In method updateGraph(), after initializing all masks, all arcs that can be trivially removed are handled by calling updateMasks(). This method assumes access to the set of values Δ_x removed from $dom(x)$ since the last call to enforceGAC()[1]. There are two ways of updating the masks (before updating currArcs from these masks, later): either incrementally or from scratch after resetting. In case of an incremental update, we perform the union of the arcs to be removed, whereas in case of a reset-based update, we perform the union of the arcs to be kept, followed by a reverse operation. Next, we need to determine which arcs can be subsequently removed: this is achieved by calling the methods propagateDown() and propagateUp(), which, similarly to MDD4R, perform two passes on the diagram. During the downward (resp., upward) pass, each level is examined from the root (resp., sink) to the sink (resp., root). When there are no more valid arcs entering (resp. exiting) a node, it becomes unreachable and all arcs exiting (resp. entering) the node becomes invalid. Identifying unreachable nodes is done by testing if the intersection between currArcs and arcsT (for outgoing arcs) or arcsH (for incoming arcs) is empty.

The process of filtering domains is very similar to the one described in CT [11]. This is given by method filterDomains() in Algo. 1. For each unbounded variable x and each value a in $dom(x)$, the intersection between the valid arcs on x, $\texttt{currArcs}[x]$, and the arcs allowing value a, $\texttt{supports}[x, a]$, determines if a is still supported. An empty intersection means that a can be deleted from $dom(x)$.

Let us mention an important fact about the bitwise operations performed at Lines 14, 17 and 28 of Algo. 1. As described in [11], bitsets are implemented as arrays of words (long integers). Progressively, while arcs become invalid, some words of currArcs become equal to 0 (all bits set to 0). In practice, operations are only performed on the active (i.e., non-zero) words of currArcs, which are easily retrievable due to the use of a sparse set maintaining the indexes of active words (called validWords). Eventually, the mask is intended for being intersected with currArcs. Therefore, computing values of the words of mask corresponding to a non-valid word of currArcs is not needed and not done.

3 Transforming Tables into *bs*-MVDs

A *bs*-MVD, or basic smart MVD, is defined similarly to a *bs*-table, or basic smart table [30]. Namely, it is an MVD where each arc is labelled by one the following expressions:

[1] In [27], a sparse-set domain implementation for obtaining Δ_x without overhead is described.

'$*$', '$\langle relop\rangle\ v$', '$\in S$' and '$\notin S$' where v is value, S is a set of values, and $\langle relop\rangle$ is an operator in $\{<, \leq, =, \neq, \geq, >\}$. The operator involved in the labeling expression of an arc ω is denoted by $op(\omega)$. There are two main strategies of generating *bs*-MVDs from (ordinary) tables:

1. Through MVDs: the table is first transformed into an MVD, using pReduce [24] (leading to an MDD) or sReduce [29] (leading to an sMDD). Then, the arcs with the same tail and head nodes are processed, targeting the general unary expressions given above. Note that the initial structure of the MVD is preserved. Namely, an MDD becomes a *bs*-MDD, while an sMDD becomes a *bs*-sMDD.
2. Through *bs*-tables: the table is first transformed into a *bs*-table, for example using the algorithms described in [17, 30]. Then, the *bs*-table is transformed into a *bs*-MVD using a slightly modified version of an algorithm transforming tables into diagrams. However, as some (smart) tuples may overlap, the transformation may lead to a *bs*-MVD with nodes that are non deterministic (in-nd and out-nd) at any level. This is not an issue since CD can handle non-deterministic diagrams.

We now describe how to carry out the second step of both approaches.

3.1 From MVDs to *bs*-MVDs: Arc Merging

Generating a *bs*-MVD from an MVD is straightforward. At each level i, we simply process every group G of (at least two) arcs sharing the same tail and head nodes. Specifically, we can compare $V = \{l(\omega) : \omega \in G\}$ with $dom(x_i)$, and consequently apply some rules (given in order of priority) for merging some arcs of G:

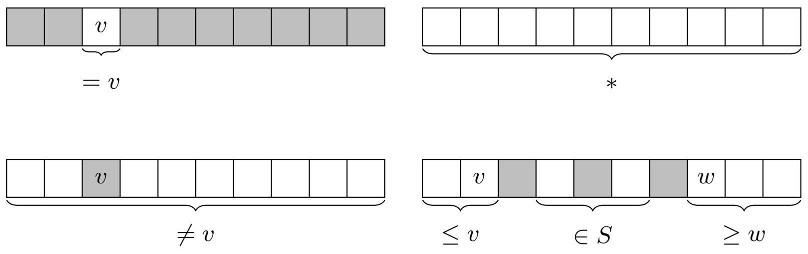
1. if $V = dom(x_i)$, then G is replaced by a unique arc labelled with '$*$';
2. if $\exists a \in dom(x_i)$ s.t. $V \cup \{a\} = dom(x_i)$, then G is replaced by a unique arc labelled with '$\neq a$';
3. if m, defined as $\max\{v : \{v' \in dom(x_i) : v' \leq v\} \subseteq G\}$ is not equal to $min(x_i)$, then $G^m = \{\omega \in G : l(\omega) \leq m\}$ is replaced by a unique arc labelled with '$\leq m$' (otherwise, $G^m = \emptyset$); if M, defined as $\min\{v : \{v' \in dom(x_i) : v' \geq v\} \subseteq G \setminus G^m\}$, is not equal to $max(x_i)$, then $G^M = \{\omega \in G \setminus G^m : l(\omega) \geq M\}$ is replaced by a unique arc labelled with '$\geq M$' (otherwise, $G^M = \emptyset$); with $G' = G \setminus G^m \setminus G^M$, if $|G'| > 1$ then G' is replaced by a unique arc labelled with '$\in S$' where $S = \{l(\omega) : \omega \in G'\}$.

Figure 2a illustrates these merging rules. The variable of interest x_i has a domain (initially) composed of 10 values, and white cells represent the values that are present in G.

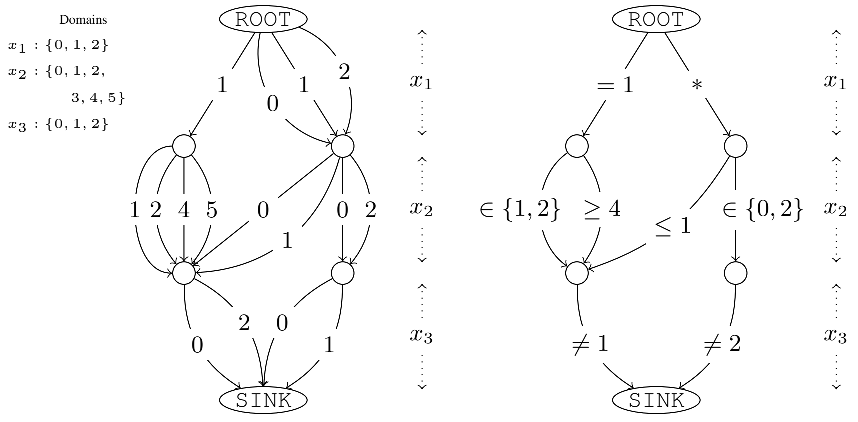
Note that our merging procedure keeps at most three arcs between any two nodes. An example is given in Fig. 2.

3.2 From *bs*-Tables to *bs*-MVDs: pReduce$_{bs}$

To create a *bs*-MVD from a *bs*-table, one can easily adapt the known procedure pReduce (initially introduced for creating MDDs from tables) so as to generate MVDs; the



(a) Illustration of possible merging rules (on a domain of size 10).



(b) An MVD (c) An equivalent *bs*-MVD

Fig. 2: Transforming an MVD into an equivalent *bs*-MVD.

adaptation is called pReduce$_{bs}$. The four steps of the procedure are the following. First, the tuples of the table are sorted using a lexicographic ordering. Second, the corresponding trie (i.e., prefix tree) is created by sharing common prefixes among the tuples. Third, a diagram is derived from the trie by merging all the leaves of the trie to form the sink node. Finally, the diagram is reduced by merging, in a bottom-up way, each pair of nodes having similar sets of outgoing edges. Two sets of outgoing arcs are similar if they have the same cardinality, and for each arc in one set, there is an arc in the other set with the same label (value) and the same head. Actually, for adapting it, we just need to impose a total order on expressions (unary constraints) involved in basic smart tuples. For example, we can simply associate a pair of integers with each expression (unary constraint) such that the first element of the pair represents the type (operator) of the expression and the second element the operand involved in the expression. Figure 3a illustrates the naturally derived lexicographic order.

Figure 3 illustrates through an example the four steps of pReduce$_{bs}$: going from a sorted *bs*-table (Fig. 3b) to a trie (Fig. 3c), then into an MVD (Fig. 3d) and finally into a reduced MVD (Fig. 3e, where the gray node is the result of merging two nodes with similar outgoing sets of edges). This example shows that pReduce$_{bs}$ does not necessar-

Expression	Representation
$= v$	$(0, v)$
$\neq v$	$(1, v)$
$*$	$(2, 0)$
$\leq v$	$(3, v)$
$\geq v$	$(4, v)$
$\in S$	$(5, \sum_{i \in S} 2^i)$
$\notin S$	$(6, \sum_{i \notin S} 2^i)$

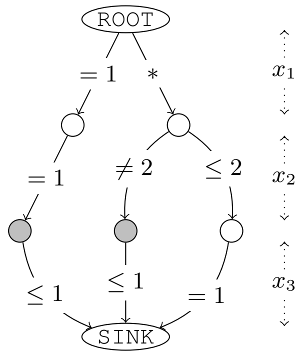
(a) Lexicographic Order on Expressions

x_1	x_2	x_3
$= 1$	$= 1$	≤ 1
$*$	$\neq 2$	≤ 1
$*$	≤ 2	$= 1$

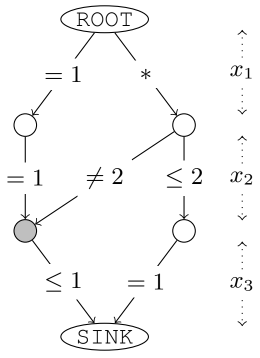
(b) Sorted Table

x_1	x_2	x_3
$= 1$	$= 1$	≤ 1
$*$	$\neq 2$	≤ 1
	≤ 2	$= 1$

(c) Trie



(d) MVD



(e) Reduced MVD

Fig. 3: Turning a *bs*-table into a *bs*-MVD using pReduce$_{bs}$.

ily generate a *bs*-MDD, because some nodes are not out-d, possibly leading to multiple paths for a same tuple as it is the case for $(1, 1, 1)$.

A similar adaptation exists for sReduce [29], the procedure that generates sMDDs, leading to sReduce$_{bs}$.

4 CD^{bs}: Compact-Diagram Handling *bs*-MVDs

CD and CT are quite similar in term of design. Basically, both of them use the bitsets called `supports` to respectively find the arcs and tuples that must be discarded. Recently, the CT^{bs} [30] algorithm, which can deal with *bs*-tables, was proposed as an extension of CT, by only modifying the update procedure. In the same spirit, we show how similar ideas can be reused to adapt the method updateMask() of CD in order to define CD^{bs}. We present first a simple version of CD^{bs}, before introducing an optimized version that strongly relies on a partition of the arcs at each level i, defined as follows:

- $\texttt{C}^{\texttt{bas}}(x_i) = \{\omega \in L(x_i) : op(\omega) \in \{=, \neq, *\}\}$,
- $\texttt{C}^{\texttt{min}}(x_i) = \{\omega \in L(x_i) : op(\omega) \in \{<, \leq\}\}$,
- $\texttt{C}^{\texttt{max}}(x_i) = \{\omega \in L(x_i) : op(\omega) \in \{>, \geq\}\}$,
- $\texttt{C}^{\texttt{set}}(x_i) = \{\omega \in L(x_i) : op(\omega) \in \{\in, \notin\}\}$

Algorithm 2: Simple Version of CD^{bs}

```
1  Method updateMasks()
2      foreach variable $x \in \{x \in scp : |\Delta_x| > 0\}$ do
3          if $|\Delta_x| < |dom(x)|$ and $\texttt{C}^{\texttt{set}}(x) = \phi$ then      // Incremental update
4              foreach value $a \in \Delta_x$ do
5                  $\texttt{mask}[x] \leftarrow \texttt{mask}[x] \mid \texttt{supports}^*[x, a]$      // bitwise OR
6              if $dom(x)$.minChanged() then
7                  $\texttt{mask}[x] \leftarrow \texttt{mask}[x] \mid \sim \texttt{supportsMin}[x, x.min]$
8              if $dom(x)$.maxChanged() then
9                  $\texttt{mask}[x] \leftarrow \texttt{mask}[x] \mid \sim \texttt{supportsMax}[x, x.max]$
10         else                                                  // Reset-based update
11             foreach value $a \in dom(x)$ do
12                 $\texttt{mask}[x] \leftarrow \texttt{mask}[x] \mid \texttt{supports}[x, a]$      // bitwise OR
13             $\texttt{mask}[x] \leftarrow \sim \texttt{mask}[x]$                          // bitwise NOT
```

Simple Adaptation of CT^{bs}. As in CT^{bs}, in addition to bitsets `supports`, we introduce auxiliary bitsets:

- $\texttt{supports}^*[x, a]$, the exclusive supports: for each arc for which the label of arc ω is exactly a ('$= a$'), the bit is set to 1,
- $\texttt{supportsMin}[x, a]$, the lower bound supports: for each arc which would be still valid if the minimum of the domain was a, the bit is set to 1,
- $\texttt{supportsMax}[x, a]$, the upper bound supports: for each arc which would be still valid if the maximum of the domain was a, the bit is set to 1.

Algorithm 2 displays the method updateMasks() for the simple version of CD^{bs}. This is for Compact-Diagram a simple adaptation of the modifications made to pass from CT to CT^{bs}. Resetting (and recomputing) is performed when the number of removed values (i.e., values in Δ_x) is too large by collecting the supports of every value in the current domain (lines 10-18). Otherwise an incremental update is performed. Notice that contrarily to the reset-based update, one needs to also collect invalid arcs for operators in $\{<, \leq, >, \geq\}$ using `supportsMin` and `supportsMax` at lines 7 and 9 of Algo 2. The time complexity of one call to updateMasks(), for a given variable x, is $\Theta(dt)$ where t is the number of valid words and d is $\min(|\Delta_x|, |dom(x)|)$ if $\texttt{C}^{\texttt{set}}(x) = \phi$ and $|dom(x)|$ if not.

Exploiting Partitions of Arcs. The time complexity of Algo. 2 can be improved to reach $\Omega(t)$ and $\mathcal{O}(dt)$. For that, let us consider the hypothetical case of a variable with an operator in $\{<, \leq, >, \geq\}$ for each of its associated arc labels. In such a case, one can collect invalid arcs using lines 7 and 9 from Algo. 2, and there is no need to iterate over the sets $dom(x)$ or Δ_x. This favorable situation can be partially forced by sorting arcs in bitsets `supports` so that the bits in a computer word only represent arcs from a given category ($\texttt{C}^{\texttt{bas}}, \texttt{C}^{\texttt{set}}, \texttt{C}^{\texttt{min}}, \texttt{C}^{\texttt{max}}$). If each computer word is filled with (bits for)

Algorithm 3: Optimized Version of CD^{bs}

```
Method updateMasks()
    foreach variable $x \in \{x \in scp : |\Delta_x| > 0\}$ do
        foreach index $j \in$ currArcs[x].validWords do
            switch currArcs[x].category[j] do
                case C^bas do
                    if $|\Delta_x| < |dom(x)|$ then                    // Incremental update
                        foreach value $a \in \Delta_x$ do
                            $mask[x][j] \leftarrow mask[x][j] \mid supports^*[x, a][j]$
                    else                                               // Reset update
                        foreach value $a \in dom(x)$ do
                            $mask[x][j] \leftarrow mask[x][j] \mid supports[x, a][j]$
                        $mask[x][j] \leftarrow \sim mask[x][j]$
                case C^set do
                    foreach value $a \in dom(x)$ do
                        $mask[x][j] \leftarrow mask[x][j] \mid supports[x, a][j]$
                    $mask[x][j] \leftarrow \sim mask[x][j]$
                case C^min do
                    if $dom(x)$.minChanged() then
                        $mask[x][j] \leftarrow mask[x][j] \mid \sim supportsMin[x, x.min][j]$
                case C^max do
                    if $dom(x)$.maxChanged() then
                        $mask[x][j] \leftarrow mask[x][j] \mid \sim supportsMax[x, x.max][j]$
```

arcs belonging to the same category (dummy invalid arcs are used to complete a word if necessary), then only the required specific operations can be systematically applied to this word. This leads to Algo. 3 that iterates over the valid words and only applies the operations required by the category of the word (note that the category for the jth word is given by `currArcs`$[x]$.`category`$[j]$). Arcs from C$^{\texttt{bas}}$ are updated using `supports`* or `supports` (incremental or reset case). Arcs from C$^{\texttt{set}}$ are updated using `supports` in all cases. Arcs from C$^{\texttt{min}}$ and C$^{\texttt{max}}$ are updated using `supportsMin` and `supportsMax`, respectively. It appears that the categories C$^{\texttt{min}}$ and C$^{\texttt{max}}$ are particularly cheap to treat as they only imply one value.

An Interesting Observation. In Algo. 3, each valid word is associated with a (unique) category. From this fact, one can observe that `supportsMin` and `supportsMax` are useless.

Observation 1 *For any variable x, and any word index j of* $\texttt{currArcs}[x]$*, we have:*

$$\texttt{currArcs}[x].\texttt{category}[j] = \texttt{C}^{\texttt{min}} \Rightarrow \\ \texttt{supportsMin}[x,a][j] = \texttt{supports}[x,a][j]$$

Similarly,

$$\texttt{currArcs}[x].\texttt{category}[j] = \texttt{C}^{\texttt{max}} \Rightarrow \\ \texttt{supportsMax}[x,a][j] = \texttt{supports}[x,a][j]$$

Proof. (sketch for $\texttt{C}^{\texttt{min}}$) By restricting the scope of the definitions of the bitsets to the word (index) j whose bits are exclusively associated with arcs from $\texttt{C}^{\texttt{min}}$, $\texttt{supports}[x,a][j]$ contains arcs represented by this word that accept the value a, i.e. arcs labelled by $\leq v$ with $v \geq a$, whereas $\texttt{supportsMin}[x,a][j]$ contains arcs for which $\exists b \in dom(x)$ accepted by the arcs such as $a \leq b$, i.e., arcs labelled by $\leq v$ with $v \geq a$. The two words end up to be equal: the exact same bits are set for both $\texttt{supports}[x,a][j]$ and $\texttt{supportsMin}[x,a][j]$.

This observation is illustrated by Fig. 4. For any literal (x,a) and any word index j of category $\texttt{C}^{\texttt{min}}$ (resp., $\texttt{C}^{\texttt{max}}$), the word $\texttt{supportsMin}[x,v][j]$ (resp., $\texttt{supportsMax}[x,v][j]$) is equal to the word $\texttt{supports}[x,v][j]$. Therefore, we can simply use supports at lines 19 and 22. It means that the only required auxiliary bitset is $\texttt{supports}^*$ for words attached to $\texttt{C}^{\texttt{bas}}$.

Overall Complexity of the Propagator. Regarding the time complexity of the propagator (and not only the updateMasks() method), CD is $\mathcal{O}(max(n,d)r\frac{a}{w})$ where r is the arity of the constraint, d the greatest domain size, n (resp. a) the maximum number of nodes (resp. arcs) per level and w the size of computer words ($w = 64$ for Java long integer type). CD^{bs} keeps the same complexity. Regarding the space complexity, the maximum number of words of one bitset is $\lceil \frac{a}{w} \rceil + 3$. Per level, there is one currArcs, d supports and $\texttt{supports}^*$ (its length is min 0 words, if $\texttt{C}^{\texttt{bas}} = \phi$ and $\lceil \frac{a}{w} \rceil$ max, if $|\texttt{C}^{\texttt{set}}| \leq w$, $|\texttt{C}^{\texttt{min}}| \leq w$ and $|\texttt{C}^{\texttt{max}}| \leq w$) and n arcsH and arcsT. The space complexity is thus $\mathcal{O}((d+n)r\frac{a}{w})$.

5 Experimental Results

All algorithms described in this paper are implemented in the Oscar solver [21], using 64-bit words (Long). Our implementation benefits from all optimization techniques described in [11] and [29]. Notably, we manage sparse sets in order to avoid handling zero computer words.

All the results of our experiments are displayed using performance profiles [12]. A performance profile is a cumulative distribution of the improved performance of an algorithm $s \in S$ compared to other algorithms of S over a set I of instances: $\rho_s(\tau) = \frac{1}{|I|} \times |\{i \in I : r_{i,\,s} \leq \tau\}|$ where the performance ratio is defined as $r_{i,\,s} = \frac{t_{i,\,s}}{\min\{t_{i,\,s} | s \in S\}}$ with $t_{i,\,s}$ the value of the measured unit (here, either the number of nodes, the number

	x
ω_0	$= 1$
ω_1	≤ 2
ω_2	≥ 1
ω_3	$\in \{1, 3\}$
ω_4	$\neq 1$
ω_5	> 2
ω_6	$\notin \{0, 3\}$
ω_7	< 2
ω_8	$\neq 2$
ω_9	$*$

(a) Labels of Arcs

	word 0				word 1				word 2				word 3			
(Category)	$\mathtt{C}^{\mathtt{bas}}$				$\mathtt{C}^{\mathtt{set}}$				$\mathtt{C}^{\mathtt{min}}$				$\mathtt{C}^{\mathtt{max}}$			
	ω_0	ω_4	ω_8	ω_9	ω_3	ω_6			ω_1	ω_7			ω_2	ω_5		
$[x, 0]$	0	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0
$[x, 1]$	1	0	1	1	1	1	0	0	1	1	0	0	1	0	0	0
$[x, 2]$	0	1	0	1	0	1	0	0	1	0	0	0	1	0	0	0
$[x, 3]$	0	1	1	1	1	0	0	0	0	0	0	0	1	1	0	0
$[x, 4]$	0	1	1	1	0	1	0	0	0	0	0	0	1	1	0	0

(b) Bitsets `supports` for literals on x

	word 0				word 1				word 2				word 3			
(Category)	$\mathtt{C}^{\mathtt{bas}}$				$\mathtt{C}^{\mathtt{set}}$				$\mathtt{C}^{\mathtt{min}}$				$\mathtt{C}^{\mathtt{max}}$			
(From)	`supports`*				no auxiliary				`supportsMin`				`supportsMax`			
	ω_0	ω_4	ω_8	ω_9	ω_3	ω_6			ω_1	ω_7			ω_2	ω_5		
$[x, 0]$	0	0	0	0	-	-	-	-	1	1	0	0	0	0	0	0
$[x, 1]$	1	0	0	0	-	-	-	-	1	1	0	0	1	0	0	0
$[x, 2]$	0	0	0	0	-	-	-	-	1	0	0	0	1	0	0	0
$[x, 3]$	0	0	0	0	-	-	-	-	0	0	0	0	1	1	0	0
$[x, 4]$	0	0	0	0	-	-	-	-	0	0	0	0	1	1	0	0

(c) Auxiliary bitsets for literals on x

Fig. 4: Bitsets related to a variable x, assuming 10 associated arcs $\omega_0, \omega_1, \ldots$ in the *bs*-MVD. The size of computer words is assumed to be 4, for simplicity.

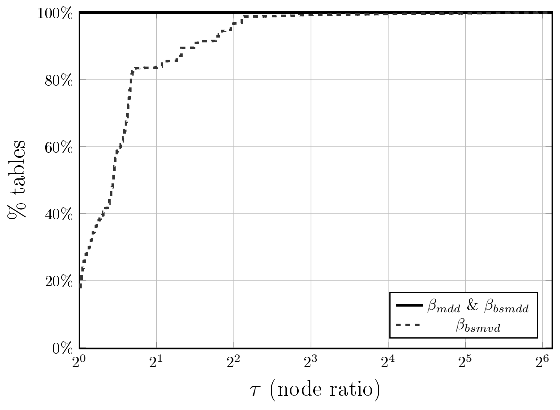
of arcs or the CPU time) obtained with algorithm $s \in S$ on instance $i \in I$. A ratio $r_{i,s} = 1$ thus means that s is the best algorithm for instance i.

Depending on the main data structure (table or diagram) and possible transformation, we use different names to describe the benchmark suite:

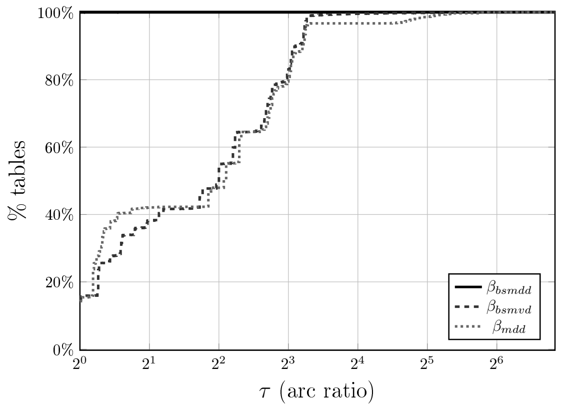
- β_t: the initial benchmark. It is a set of roughly $4,000$ instances only containing (positive) table constraints, and available on the XCSP3 website [7].
- β_{bst}: instances of β_t have been transformed into instances where *bs*-tables replace (ordinary) tables. The compression algorithm detailed in [30] was used.
- β_{mdd}: instances of β_t have been transformed into instances where MDDs replace (ordinary) tables. The algorithm pReduce [24] was used.
- β_{bsmvd}: instances of β_{bst} have been transformed into instances where *bs*-MVDs replace *bs*-tables. The algorithm pReduce$_{bs}$ was used.
- β_{bsmdd}: instances of β_{mdd} have been transformed into instances where *bs*-MDDs replace MDDs.

Our experiments have two main objectives:

1. analyzing the compression quality of the different approaches, when transforming tables,
2. analyzing the speedup obtained by the new filtering algorithms over the existing ones.



(a) Nodes



(b) Arcs

Fig. 5: Performance profile comparing the structure of the graphs from β_{bsmdd}, β_{bsmvd} and β_{mdd}.

5.1 Quality of Compression

To start, we consider the results depicted in Fig. 5. The three benchmarks involving MVDs are β_{mdd}, β_{bsmvd} and β_{bsmdd}. In term of compression, the clear winner is β_{bsmdd} with substantially less arcs than in the diagrams generated by the two other approaches. Let us recall that this approach consists of two main steps: 1) creating a graph, and 2) merging arcs greedily. The alternative approach β_{bsmvd} that creates first a *bs*-table, and then converts it into a *bs*-MVD is worse both in terms of the number of nodes and the number of arcs, even when compared to a standard generation of MDDs (β_{mdd}). One explanation is that, despite starting from smaller tables, there is less chance to merge nodes due to the proliferation of constraint labels in the compressed tables.

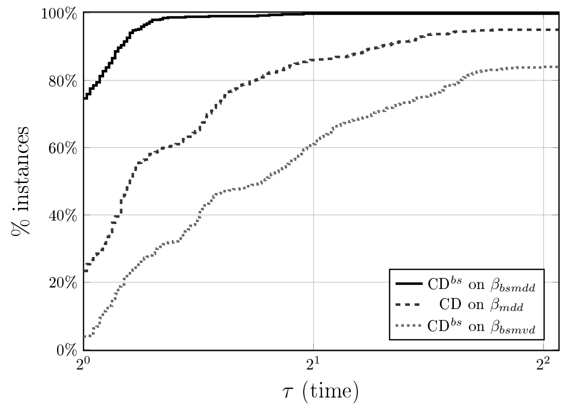


Fig. 6: Performance profile comparing CD^{bs} to CD on various basic smart MVD benchmarks CD on β_{mdd}, CT^{bs} on β_{bsmdd}, β_{bsmvd} and β_{bsmdd}.

5.2 Speedup

Figure 6 shows the results of a comparison between CD and CD^{bs}. The new filtering algorithm CD^{bs}, as it could be expected, obtains a larger speedup when applied on graphs with fewer nodes and arcs, i.e., on instances from β_{bsmdd}.

In particular, we can see that on the benchmark β_{bsmvd} (based on a compression into *bs*-tables, followed by a generation of *bs*-MVDs) CD^{bs} performs worse than CD applied on β_{mdd} (standard MDDs). The reason is that graphs in β_{bsmvd} have generally a greater number of nodes than other equivalent graphs as shown before in Fig. 5a. This follows the same conclusions as in [29] regarding why CD was more efficient on sMDDs (having fewer nodes than MDDs).

An interesting remark is that, contrarily to CT^{bs} (Compact-Table for basic smart tables) [30], the presence of expressions '$\in S$' does not induce any overhead for CD^{bs}. Since the arcs involving expressions of the form '$\in S$' are gathered on the same bitwords, they don't prevent from doing an incremental update when considering the other categories of expressions, as it was the case for CT.

In [29], CT was shown to remain faster than CD despite the introduction of bitwise operations. We revisit the same experiment with the newly presented algorithm. Figure 7 compares four scenarios, including the use of CT: CT on β_t, CT^{bs} (the extension of CT [30] handling directly *bs*-tables) on β_{bst}, CD (Compact-Diagram, the adaptation of CT to MVD [29]) on β_{mdd} and CD^{bs} on β_{bsmdd}.

One can see that CT is still the best approach, followed by CT^{bs}. Nevertheless, as it can be observed in the figure, the gap is shrinking when using the new algorithm CD^{bs}. Also, there is now around 10% of the instances where CD^{bs} is the fastest algorithm. A post analysis has shown that instances with larger domains are the more favorable for CD^{bs}. In such cases, we could observe for some tables a reduction by a factor of up to 8 on the number of arcs.

The main advantage of CD thus lies in the potential compactness of the diagrams, although this is really problem/constraint dependent. On the one side, some graphs, when

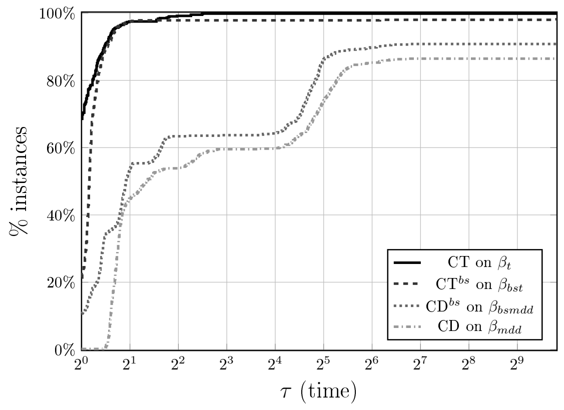


Fig. 7: Performance profile of the comparison of the best-case scenario of CD^{bs}, CD and the tables algorithms CT and CT^{bs}.

expanded into tables, can't even fit in memory. On the other side, some constraints, like AllDifferent [22] are not well suited for an MDD representation because there is almost no compression. When CD can benefit from a large compression, it becomes faster.

For a fair comparison, the choice was made not to evaluate the new algorithm on a priori favorable problems, hence the benchmarks composed of problems that initially contain table constraints. Also the order of variables remained unchanged (order as described in the initial instances used). Optimizing this order may also have an impact on the size of the graphs [9].

In our opinion, having both CT and CD is useful: if, for a given constraint, a high compression (by an MDD or another diagram) is possible, CD should be used, otherwise CT is more suited. Also, the new algorithm should typically be used for solving combinatorial problems with complex constraints that can't even be represented in memory as simple tables. One good example of work in that direction is [26]. Another promising direction for applying this propagator is for solving combinatorial problems on Strings.

6 Conclusion

We have proposed to use a new general form of Multi-valued Variable Diagram (MVD) for representing constraints: the *bs*-MVD that accepts unary constraints as labels of arcs. We have also shown how to generate such diagrams from (ordinary) tables. Finally, we have adapted the propagator CD (Compact-Diagram) to *bs*-MVDs, by inspiring ourselves from the adaptation of CT to *bs*-tables. This new propagator is efficient and makes little closer graph-based approaches and table-based approaches.

Acknowledgements

The second author is supported by the project CPER Data from the "Hauts-de-France".

References

1. Amilhastre, J., Fargier, H., Niveau, A., Pralet, C.: Compiling CSPs: A complexity map of (non-deterministic) multivalued decision diagrams. International Journal on Artificial Intelligence Tools **23**(04) (2014)
2. Andersen, H., Hadzic, T., Hooker, J., Tiedemann, P.: A constraint store based on multivalued decision diagrams. In: Proceedings of CP'07. pp. 118–132 (2007)
3. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In: International Conference on Principles and Practice of Constraint Programming. pp. 107–122. Springer (2004)
4. Bergman, D., Ciré, A., van Hoeve, W.: MDD propagation for sequence constraints. Journal of Artificial Intelligence Research **50**, 697–722 (2014)
5. Bergman, D., Ciré, A., van Hoeve, W., Hooker, J.: Decision diagrams for optimization. Springer (2016)
6. Bessiere, C., Hebrard, E., Hnich, B., Kiziltan, Z., Quimper, C.G., Walsh, T.: Reformulating global constraints: The slide and regular constraints. In: International Symposium on Abstraction, Reformulation, and Approximation. pp. 80–92. Springer (2007)
7. Boussemart, F., Lecoutre, C., Piette, C.: XCSP3: An integrated format for benchmarking combinatorial constrained problems. Tech. Rep. arXiv:1611.03398, CoRR (2016), available from `http://www.xcsp.org`
8. Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **35**(8), 677–691 (1986)
9. Cappart, Q., Goutierre, E., Bergman, D., Rousseau, L.M.: Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In: Proceedings of AAAI'19 (2019)
10. Cheng, K., Yap, R.: An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. Constraints **15**(2), 265–304 (2010)
11. Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J.C., Schaus, P.: Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In: Proceedings of CP'16. pp. 207–223 (2016)
12. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Mathematical programming **91**(2), 201–213 (2002)
13. Gange, G., Stuckey, P., Szymanek, R.: MDD propagators with explanation. Constraints **16**(4), 407–429 (2011)
14. Hadzic, T., Hooker, J., O'Sullivan, B., Tiedemann, P.: Approximate compilation of constraints into multivalued decision diagrams. In: Proceedings of CP'08. pp. 448–462 (2008)
15. Hoda, S., van Hoeve, W., Hooker, J.: A systematic approach to MDD-Based constraint programming. In: Proceedings of CP'10. pp. 266–280 (2010)
16. Ingmar, L., Schulte, C.: Making compact-table compact. In: International Conference on Principles and Practice of Constraint Programming. pp. 210–218. Springer (2018)
17. Le Charlier, B., Khong, M.T., Lecoutre, C., Deville, Y.: Automatic synthesis of smart table constraints by abstraction of table constraints
18. Lecoutre, C.: STR2: Optimized simple tabular reduction for table constraints. Constraints **16**(4), 341–371 (2011)
19. Lecoutre, C., Likitvivatanavong, C., Yap, R.: STR3: A path-optimal filtering algorithm for table constraints. Artificial Intelligence **220**, 1–27 (2015)
20. Mairy, J.B., Deville, Y., Lecoutre, C.: The smart table constraint. In: Proceedings of CPAIOR'15. pp. 271–287 (2015)
21. OscaR Team: OscaR: Scala in OR (2012), available from `https://bitbucket.org/oscarlib/oscar`

22. Perez, G.: Decision diagrams: constraints and algorithms. Ph.D. thesis, Université de Nice (2017)
23. Perez, G., Régin, J.C.: Improving GAC-4 for Table and MDD constraints. In: Proceedings of CP'14. pp. 606–621 (2014)
24. Perez, G., Régin, J.C.: Efficient operations on mdds for building constraint programming models. In: Twenty-Fourth International Joint Conference on Artificial Intelligence (2015)
25. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Proceedings of CP'04. pp. 482–495 (2004)
26. Roy, P., Perez, G., Régin, J.C., Papadopoulos, A., Pachet, F., Marchini, M.: Enforcing structure on temporal sequences: the allen constraint. In: International conference on principles and practice of constraint programming. pp. 786–801. Springer (2016)
27. le Clément de Saint-Marcq, V., Schaus, P., Solnon, C., Lecoutre, C.: Sparse-sets for domain implementation. In: Proceeding of TRICS'13. pp. 1–10 (2013)
28. de Uña, D., Gange, G., Schachte, P., Stuckey, P.J.: Compiling cp subproblems to mdds and d-dnnfs. Constraints **24**(1), 56–93 (2019)
29. Verhaeghe, H., Lecoutre, C., Schaus, P.: Compact-mdd: Efficiently filtering (s) mdd constraints with reversible sparse bit-sets. In: IJCAI. pp. 1383–1389 (2018)
30. Verhaeghe, H., Lecoutre, C., Deville, Y., Schaus, P.: Extending compact-table to basic smart tables. In: Proceedings of CP'17. pp. 297–307 (2017)
31. Verhaeghe, H., Lecoutre, C., Schaus, P.: Extending compact-table to negative and short tables. In: Proceedings of AAAI'17 (2017)
32. Wang, R., Xia, W., Yap, R., Li, Z.: Optimizing Simple Tabular Reduction with a bitwise representation. In: Proceedings of IJCAI'16. pp. 787–795 (2016)