definition

Noname manuscript No.
(will be inserted by the editor)

From MDD to BDD and Arc consistency

Julien Vion & Sylvain Piechowiak

Abstract In this paper, we present a new conversion of multivalued decision diagrams (MDD) to binary decision diagrams (BDD) which can be used to improve MDD-based filtering algorithms such as *MDDC* or *MDD-4R*. We also propose *BDDF*, an algorithm that copies modified parts of the BDD "on the fly" during the search of a solution, and yields a better incrementality than a pure *MDDC*-like approach. *MDDC* is not very efficient when used to represent poorly structured positive table constraints. Our new representation combined with *BDDF* retains the properties of the MDD representation and has comparable performances to the *STR2* algorithm by Ullmann [37] and Lecoutre [22].

1 Introduction

The *Constraint Satisfaction Problem* (CSP) is an extremely versatile generic model for combinatorial problems and one of the most studied NP-complete problems. Problems are decomposed into *constraint models*, i. e., hyper-graphs whose vertices are *variables* and hyper-edges represent *constraints* that define the allowed or forbidden instantiations of variables they are connected to. The CSP consists of deciding whether an instantiation of all variables that satisfies all constraints, i. e., a *solution*, exists. The contents of this article also apply to the NP-hard Constraint *Optimization* Problem (COP). The COP is to find an *optimal* solution to a constraint model, i. e., that maximizes or minimizes the value of one variable.

CSP and COP are traditionally solved using a complete, systematic depth-first search algorithm. At every search node, the problem is *filtered*: values that cannot appear in any solution, i. e., *inconsistent* values, are removed from the domains of the variables [5]. Inconsistent values are detected using properties that can usually be checked in reasonable time and space. This process is usually called *propagation*. What we want is the best trade-off between the complexity of the propagation process, and the complexity of the search.

Consistency properties are usually defined at the constraint level. Constraints are propagated one by one until a fix-point is reached, i. e., no inconsistent value can be identified by any constraint. *Arc consistency* (AC) is obtained when, for each constraint in the model, algorithms are able to detect *all* values that are inconsistent w.r.t. the constraint. Although arc consistency has been historically defined on binary constraints defined in extension only [29, 26], modern solvers are rather based on more versatile generic propagators of any arity [20]. The positive *table* constraint is defined in *extension*, i. e., as a list of allowed instantiations. Such a list forms a *relation*, akin to a *table* in a relational

Université de Valenciennes et du Hainaut Cambrésis
LAMIH CNRS UMR 8201
E-mail: {julien.vion,sylvain.piechowiak}@univ-valenciennes.fr

database [12]. It is an important constraint type which appear in numerous models (e. g., database, configuration) and has been very much studied recently [37, 22, 24, 27].[1] They are the most natural generalization of "historical" 1970s–80s binary constraints to the non-binary case.

One important feature for propagating constraints efficiently is *incrementality*. Indeed, in order to reach a fix-point during constraint filtering, and while exploring the search space using a DFS, the propagation algorithm will be called a very large number of times on the same relation. Avoiding to repeat similar work as much as possible is mandatory for performance. This requires to design algorithms and data structures that have incrementality properties. This is usually not trivial as, during the construction of the DFS, *backtracks* will occur, and the data structures used will have to be backtracked as well efficiently. One key data structure used to implement backtrackable data is the *sparse set* [7][2].

If the relation can be represented by a "flat" positive table of reasonable size, the reference algorithm with the best simplicity *vs.* efficiency ratio is probably *STR2* [22], although theoretically more advanced procedures exists, such as *STR3* [24] or *AC5TC* [27]. Recently, *Compact Table*, a heavily optimized variant based on *bit vectors*, has been proposed [13]. The idea behind *STR* is to filter the table before the search for supports: invalid lines are removed and will not be considered on the current branch of the search tree. When a backtrack is required by the search process, a simplified form of sparse set is used to restore the removed lines in constant time. *STR2* also avoids to control variables that are not modified between two calls to the algorithm, and stops the search for supports of variables whose entire domain has been proved to be consistent.

An efficient approach to handle large tables is to *compress* them, using e. g., *Tries* [18] or *multivalued decision diagrams* (MDD) [9]. Such data structures can represent an exponential number of instantiations in polynomial space. MDD can thus represent relations such as *sliding-sum/sequence* in polynomial space [4, 32]. They can also represent "unrolled" finite-state automatons (i. e., the *regular* global constraint [34]). Handling relations with MDD is a very promising approach, as Cheng and Yap [9] show with their *MDDC* algorithm. *MDDC* uses sparse sets to label invalid vertices in the MDD, which brings incrementality to the algorithm. Gange, Stuckey, and Szymanek [16] and Perez and Régin [33] have already shown that the original algorithm could be improved. In this article, we show how a simple conversion of a MDD to a *binary decision diagram* (BDD) [8] can significantly improve the performances of MDD-based algorithms. Indeed, our representation has a much finer grain than a classic MDD implementation, which allows to reduce the raw size of the data structure and to achieve finer incrementality. We designed two algorithms, namely *BDDC2*[3] and *BDDF*. The former is a direct adaptation of *MDDC* that operates on our BDD conversion. We show both formally and experimentally the improvement of *BDDC2* over *MDDC*. *BDDF* was designed to improve the incrementality of *BDDC2* even further, in a similar way as Perez and Régin [33] did directly on MDD in parallel to this work. This is allowed by a better exploitation of the data provided by coarse-grained propagation queues, and on-the-fly filtering of the BDD inspired by *STR2*.

2 Definitions and notations

Definition 1 (Variable, domain, instantiation, constraint) A constraint model consists of a pair $(\mathscr{X}, \mathscr{C})$, where:

[1] Negative table constraints have received much less attention but are reasonably handled by generic AC algorithms such as AC-3^{rm} [23] and efficient indexation of the table.

[2] Note that sparse sets can only be used when all operations on the set are *monotonic* on a branch of the search tree, i. e., items are only *added* along the whole branch, or *removed* along the whole branch.

[3] The name *BDDC* was already used by Cheng and Yap [10] to denote a preliminary version of *MDDC* restricted to binary domains.

$\mathscr{X}$ is a set of $\boldsymbol{n}$ *variables*; each variable $X \in \mathscr{X}$ is defined on a *domain* $\mathrm{dom}(X)$;

$\mathscr{C}$ is a set of $\boldsymbol{e}$ *constraints*; each constraint $C \in \mathscr{C}$ involves at most $\boldsymbol{k}$ variables $\mathrm{vars}(C) \subseteq \mathscr{X}$ and defines the set of at most $\boldsymbol{\lambda}$ allowed instantiations of these variables.

An *instantiation* I of a set of variables $\mathcal{X}$ binds each variable of $\mathcal{X}$ to one *value* from its domain.

In this article, we consider that all domains are discrete and contains at most $\boldsymbol{d}$ values. An instantiation is said to be *valid* when it binds variables to values from their domains. The *tightness* ($\geq \boldsymbol{t}$) of a constraint is usually defined as the proportion of instantiations *forbidden* by C w.r.t. the number of valid instantiations of $\mathrm{vars}(C)$. In this article, we will rather use the *looseness* metric ($\leq \boldsymbol{l}$), defined as the proportion of instantiations *allowed* by C w.r.t. the number of valid instantiations. We have $l = 1 - t$ and $\lambda \leq ld^k$.

Definition 2 (Arc Consistency, support) Let C be a constraint, $X \in \mathrm{vars}(C)$ and $v \in \mathrm{dom}(X)$. A *support* of v w.r.t. C is a valid instantiation of $\mathrm{vars}(C)$, allowed by C, that binds X to v. v is *arc consistent* (AC) w.r.t. C iff there exists a support of v w.r.t. C.

Enforcing arc consistency on a constraint C consists of removing all values from the domains of the variables of $\mathrm{vars}(C)$ that are not AC for C. This process is called a *constraint propagation* or a *revision*. Applying arc consistency on a constraint model consists of propagating all constraints that may involve non-AC values until a fix-point is reached. For any CSP/COP model, the fix-point is unique.

Example 1 Constraint C involves variables X, Y and Z, whose domains are $\{a, b, c\}$, and allows the following six instantiations of these variables:

$$\begin{aligned}\mathrm{tab}(C) = \{ \ & \langle X = b, Y = c, Z = b \rangle \, , \ \langle X = a, Y = b, Z = a \rangle \, , \\ & \langle X = a, Y = a, Z = a \rangle \, , \langle X = c, Y = a, Z = c \rangle \, , \\ & \langle X = c, Y = a, Z = a \rangle \, , \ \langle X = a, Y = a, Z = c \rangle \}\end{aligned}$$

In this example, we have $k = 3$ variables, $d = 3$ values and $\lambda = 6$ allowed instantiations. The looseness of C is $l = 6/3^3 \approx 22\,\%$.

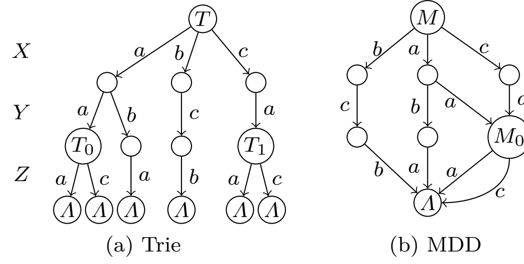
Proving that a value is arc consistent w.r.t. this constraint requires to find a *support* for this value. An instantiation is allowed iff it appears in $\mathrm{tab}(C)$. It is valid iff all values of the instantiation are present in the domain of the variables. Such an instantiation supports all domain values that appear in it. In Example 1, the value b from X's domain is supported by $\langle X = b, Y = c, Z = b \rangle$. If we remove c from Y's domain, the instantiation is not valid anymore: b has no other support and can be removed from the domain of X. In the worst case, a positive table constraint allows $\Theta(d^k)$ instantiations, and enumerating them to find supports may seem impractical.

In the remaining of this article, we only consider constraints that are defined in extension as a list of *allowed* instantiations. However, all discussed techniques can be applied with heterogeneous constraint types, using the most appropriate propagator for each constraint.

We implement our data structures using linked lists, indexed sequences (arrays), and hash tables. In this paper, we use 1 as the base index for arrays: $A[i]$ is the ith element of A. For lists, we define the *prepend* operator "::", e. g., $0 :: \langle 1, 2, 3 \rangle = \langle 0, 1, 2, 3 \rangle$. We will also use the following metrics to evaluate the size of graphs: $|V(G)|$ and $|E(G)|$ respectively denote the number of vertices and edges of graph G.

3 Multi-valued Decision Diagrams

This section describes MDDs, which is the data structure we build upon to represent allowed instantiations in constraints.



(a) Trie (b) MDD

Fig. 1: A Trie T (a) and a MDD M (b) that represents the table of Example 1. T_0 and T_1 are the same: they are merged in one node M_0 in the MDD.

b	c	b
a	a	a
a	a	c
a	b	a
c	a	c
c	a	a

Fig. 2: Relation of Example 1 recursively grouped by prefix, used to build the Trie of Fig. 1a.

A Trie (which stands for *Re*trie*val*) is a tree-based data structure that was initially designed to efficiently represent and perform searches in a dictionary (a set of text strings) [15]. For our purposes, it can be used to represent the set of instantiations allowed by a constraint, which can then be checked in $\Theta(k)$.

Definition 3 (Trie) A Trie T representing a set of lists S can be either:

- a leaf Λ which represents the set composed of the empty list $\{\langle\rangle\}$,[4] or
- an application $T : i \to T'$ such that T' is a Trie that represents the set of lists $\{\tau, \dots\}$ such that $(i :: \tau) \in S$.

A vertex in the tree represents the set of all lists sharing the same prefix. Each leaf of the tree corresponds to an element of *Tab* (cf Figure 1a). A Trie performs a *compression* of the table, i. e., $|E(\mathit{trie})| \leq k\lambda$ (note that since the structure is tree-shaped, each vertex except the root has exactly one ingoing edge). Constructing a Trie from a table can be optimally performed in $\Theta(k\lambda)$. The trick is to "group" the table by prefix recursively, using an efficient hash table implementation. Most functional languages (including the Java 8 *Stream API*) provide a generic $\Theta(\lambda)$ *groupBy* second-order function, which allows to implement the construction algorithm in a few lines of code. Fig. 2 gives an insight on how the algorithm works: first, tuples are grouped according to column $i = 1$. Then, each group is recursively grouped according to column $i + 1$. The result gives the shape of our trie. Purely functional data structures equivalent to hash tables are typically implemented using hash tries, adding an $\Theta(\log d)$ factor[5] to the operations [2].

A trie representing the worst-case "full" relation of d^k instantiations (a table of kd^k cells) has $\sum_{i=1}^{k} d^i$ edges. It follows that $|E(\mathit{trie})| \in O(d^k) \cap O(k\lambda)$. Compression is better when tries represent the allowed instantiations of loose constraints (λ is relatively large, closer to d^k) as the probability that many lines share the same prefix increases.

[4] For most applications, a correct rationale for Λ is that it represents the set of all relations.

[5] Actually $\Theta(N)$ where N is the number of bits used to represent domain values.

MDDs are defined similarly to Tries above. MDD allow identical sub-trees to be merged. A MDD is thus a singly-rooted DAG (cf Figure 1b). For a same set of instantiations, we have $|E(mdd)| \leq |E(trie)|$. A MDD (or Trie) can be "reduced" to a minimal MDD (i. e., such that the MDD does not contain any two identical vertices) in $\Theta(|E(mdd)|)$ by traversing it by depth-first search (DFS) and indexing each vertex [9].[6] A reduced MDD is often called RMDD, but all algorithms in this paper can be applied whether the MDD is fully reduced or not. MDD can also be constructed from ad hoc functions, but memoization [28] (also called *hash consing* [1]) is often used to merge sub-trees. Perez and Régin [32] give some insight on building MDDs efficiently from a finite state automaton or some other functional pattern.

The table depicted on Example 1 contains 6 instantiations of 3 variables, and can be represented using a 6×3 table (i. e., 18 cells), a Trie of 13 edges, or the MDD of 11 edges on Figure 1b. Interestingly, the set of all possible instantiations of 3 variables with 3 values is represented by a table of $3^3 = 27$ lines of 3 cells each (i. e., 81 cells), a Trie of $3^1 + 3^2 + 3^3 = 39$ edges or a MDD of 9 edges.

Although it is not formally required, the order in which variables appear along the branches of Tries/MDD representing table of instantiations is always the same. Such a MDD is often called an Ordered MDD (OMDD). This assumption still considerably simplifies algorithms and usually yields smaller reduced graphs. The size of MDD (i. e., number of edges) is highly dependent on the chosen order. However, finding the optimal order is NP-hard [6] and is outside the scope of this paper. To simplify our descriptions, we consider that all MDD are ordered, but our algorithms can be generalized.

Using Tries or MDDs to represent table constraints was previously considered by e. g., Gent et al. [18] (for Tries) and Cheng and Yap [9] and more recently Gange, Stuckey, and Szymanek [16] and Perez and Régin [33] (for MDDs). Gent et al.'s strategy is to seek for supports by a DFS in a Trie, ignoring edges (and therefore skipping branches) corresponding to non-valid instantiations. For example, one can search for a support for $X = b$ under the hypothesis that $Y = a$ in the Trie T of Figure 1a. We start from the root T and we follow the edge b as we want a support for $X = b$. The next edge is invalid as $Y \neq c$. There is no other available edge, so we deduce that $X = b$ has no support and can be deleted. Processing variables deeper in the tree is of course much more time-consuming, so the authors propose to generate several Tries: in this way, each variable is at the top of some Trie. However, search space exploration by DFS interacts badly with this scheme, e. g., searching for a support for $X = a$ under the hypothesis made by the DFS that $Z = c$ requires to traverse the whole leftmost sub-tree to deduce that no such support exist.

Cheng and Yap [9] developed *MDDC*, another algorithm specifically targeted at MDDs (although it can be applied on Tries). Contrary to Gent et al.'s technique, *MDDC* does not restart the DFS when a support is found, but traverses the whole MDD completely. Every time a leaf is reached, all encountered values so far can be labelled as supported. The trick is that invalid parts of the MDD are labelled and recorded so that they will not be traversed again in the same branch of the search tree. *Timestamps* are used to avoid traversing the same vertex twice in the same pass, and backtrackable *sparse sets* [7] record invalid vertices and bring incrementality to the algorithm. Once the MDD has been traversed, all values that were not labelled as supported can be safely removed from the domain of the variables.

Algorithm 1 is an adaptation of *MDDC* using our notations. It requires a few parameters besides the MDD M: the timestamp *ts* is a global value which is incremented before each call to the algorithm. *Invalid* is a backtrackable set which is initialized to the empty set at the root of the DFS, and is maintained along the branches of the search tree. It can be implemented using e. g., a *sparse set*. *Supported* and p are respectively initialized to 0

[6] This complexity implies that indexing a vertex has a complexity linear in the number of outgoing edges (i. e., does not depend on d), which can be obtained easily by using appropriately chosen data structures, e. g., linked lists or hash tables.

Algorithm 1: MDDC(M, ts, $Invalid$, p, $Supported$)

```
input  :
M  is a MDD vertex that represents allowed instantiations of variables 𝒳.
ts  is the current timestamp s.t. Ts[M] = ts iff M has been visited in the current run of this
     algorithm and is part of at least one support.
Invalid  is a set that contains MDD vertices which lead to no valid instantiations.
p  is the index s.t. 𝒳[p] is the variable corresponding to the current level of the MDD.
Supported  is an application of each variable of 𝒳 to the set of values that have at least one support.
δ  contains the level from which entire domain of all variables are supported.
output : true if and only if the MDD contains at least one support.
1  if (M = Λ) ∨ (ts = Ts(M)) then  return true
2  else if M ∈ Invalid then  return false
3  else
4      X ← 𝒳[p]
5      valid ← false
6      foreach i ∈ dom(X) | MDDC(M[i], ts, Invalid, p + 1, Supported) do
7          valid ← true
8          Supported(X) ← Supported(X) ∪ {i}
9          if (p + 1 = δ) ∧ (Supported(X) = dom(X)) then
10             δ ← p
11             break;
       /* valid is true if and only if M has at least one valid child ⟹ M is
          valid.                                                              */
12     if valid then  Ts(M) ← ts
13     else  Invalid ← Invalid ∪ {M}
14     return valid
```

and {} (the empty set) before each call to the algorithm. The outgoing edges of a MDD vertex are typically implemented using an array data structure. Thus, $M[i]$ is the "child" vertex reached from M through edge i.

The δ variable and Lines 9–11 implement an *early cutoff optimization*: the idea is to early-end the algorithm if entire domains of all variables further down in the MDD already have been identified as supported. It is initialized to k before each call to the algorithm.

Example 2 We describe one run of *MDDC* on the MDD from Fig. 1b. Line 6 builds the set of supported values of variable X by recursive calls to the *MDDC* function. Leftmost branch is traversed first, down to the leaf. When the leaf is reached, we know that all values on the branch are supported ($X = b$, $Y = c$ and $Z = b$). The algorithm then backtracks to the middle branch, which has values $X = a$, $Y = b$ and $Z = a$, then backtracks to level 2 to go trough the branch $Y = a$, $Z = a$ then one backtrack to $Z = c$. Finally, we backtrack to $X = c$, $Y = a$. When M_0 is reached, the timestamp tells us that there exists at least one valid vertex further down, so it is no necessary to continue to label $X = c$ as supported.

If we state the hypothesis $Y = a$, we have to traverse the MDD again to discover supports that include this assumption. The first branch ($X = b$) does not allow to reach the leaf anymore. The vertex is labelled as invalid and will not be traversed again until the domain of Y and the *Invalid* set are restored to a previous state.

MDDC is often compared to *STR2* [22], which operates on a flat table representation of the relation. Indeed, although the compression obtained by MDDs is nearly always interesting, *STR2* operates two extra optimizations: when the *early cutoff optimization* of *MDDC* only triggers when all values of every variable from some level of the MDD is supported, *STR2* allows to skip individual variables regardless of its position in the scope of the constraint. Moreover, contrary to *MDDC*, *STR2* takes into account information about modified variables brought by the solver: there is no need to check the validity of values from unmodified variables. Finally, tables used by *STR2* are usually more CPU-

cache-friendly than standard graph implementations. In conclusion, *MDDC* is interesting over *STR2* only when MDDs allow a high compression of the table ($|E(mdd)| \ll k\lambda$), which requires loose and/or very structured constraints.

Other recent work on MDD-based constraint processing includes *MDDI* by Gange, Stuckey, and Szymanek [16] which implements explanation techniques as well as the inclusion of backtrack-stable *watches* to skip some search for supports. Recently, Perez and Régin [33] developed *MDD-4R*, an algorithm based on *GAC-4* which filters MDDs during search. A trailing technique is used to restore deleted edges on backtrack. Both algorithms improve the incrementality properties of MDD processing. Recent experiments, however, by Demeulenaere et al. [13] show that even the latest *MDD-4R* algorithm hardly improves on *STR2* in practice and that optimizations of *STR2* are still the most competitive options on available benchmarks.[7]

4 An alternative representation for MDDs

Historically, Tries and MDDs have been mainly used to perform fast lookup operations, i. e., checking whether a particular tuple is present in the data structure in $\Theta(k)$. For this purpose, the outgoing edges of a MDD vertex are usually stored using an indexed data structure, e. g., array or hash table. The loop at Line 6 of Algorithm 1 is described here akin to Cheng and Yap's version: it iterates over the domain values, and for each value we check whether a valid child MDD exists. Consequently, known algorithms usually have a complexity in $\Theta(d \cdot |V(mdd)|)$ operations. If an array is used to represent outgoing edges, which is often the case to obtain best performance for lookups, then the space complexity is $\Theta(d \cdot |V(mdd)|)$ as well. This complexity amounts to consider that each vertex has exactly d outgoing edges. A glance at, e. g., Fig. 1b on page 4, encourages us to think that this can be improved. Moreover, we found out that for *MDDC* – as well as other algorithms such as Cheng and Yap's *reduce* function – the lookup operation is not required: if an appropriate data structure is used, one can replace the iteration over domain values by an iteration over outgoing edges.

The following property is easy to show, since each vertex has at most d outgoing edges:

Property 1 For any MDD M, $|E(M)| \leq d \cdot |V(M)|$

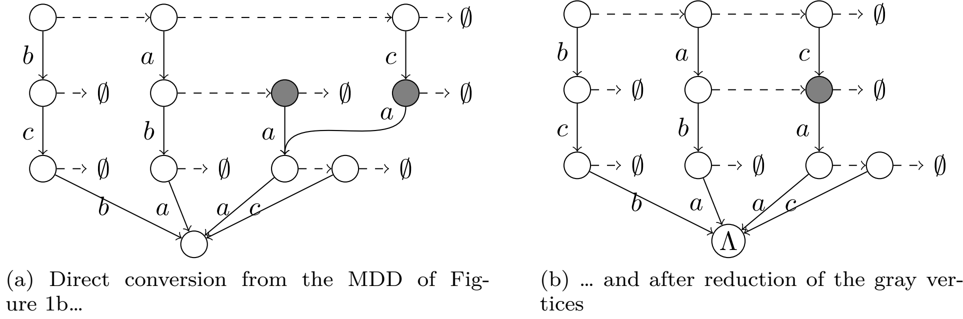
If we carefully implement a *MDDC* variant, we can consider each edge of the MDD at most once, resulting in a complexity in $\Theta(|E(M)|)$, which should be an improvement over $\Theta(d \cdot |V(M)|)$ according to Property 1 above. Actually, using simply linked lists to represent outgoing edges is sufficient to obtain this behavior, but it can be improved. We consider a binary representation of the MDD, i. e., all vertices have at most two outgoing edges. We obtain a so-called *binary decision diagram* (BDD) [8] with the following semantics:

Definition 4 A BDD B representing a set of lists S can be either:

- the empty set $\emptyset$, or
- a leaf Λ which represents the set composed of the empty list, or
- a triple $(i, \mathit{child}, \mathit{sibling})$ where *child* is a BDD that represents all tuples prefixed by i in S, and *sibling* is a BDD that represents the remaining tuples.

This representation is strictly equivalent, memory-wise, to using a list to represent the outgoing edges of a vertex: simply think the "sibling" edge of a BDD vertex as the "tail" edge of a linked list composed of (i, child) cells. On Figure 3, the "linked list tail edges" would correspond to the dashed lines. BDDs and BDD processing algorithms are simpler to implement than MDD, as they avoid the reference/array duality usually required to

[7] Note that the instances used by Demeulenaere et al. does not include any of the heavily structured relations described by e. g., Cheng and Yap [9] (definite state automatons, super-row density, sliding-sum constraints…), that can only be handled by MDD or ad-hoc representations.



(a) Direct conversion from the MDD of Figure 1b…

(b) … and after reduction of the gray vertices

Fig. 3: BDD representation of the relation from Example 1. ⇢ are “sibling” and → are “child” edges. The “∅” vertex is duplicated here for readability.

Algorithm 2: MDDtoBDD(M, ts)

input : M is a MDD vertex.
ts is a timestamp s.t. $Ts[M] = ts$ iff M has been visited in the current run of this algorithm. In this case, $Last[M]$ contains the previously computed result.
output : A BDD vertex equivalent to M w.r.t. Definition 4.

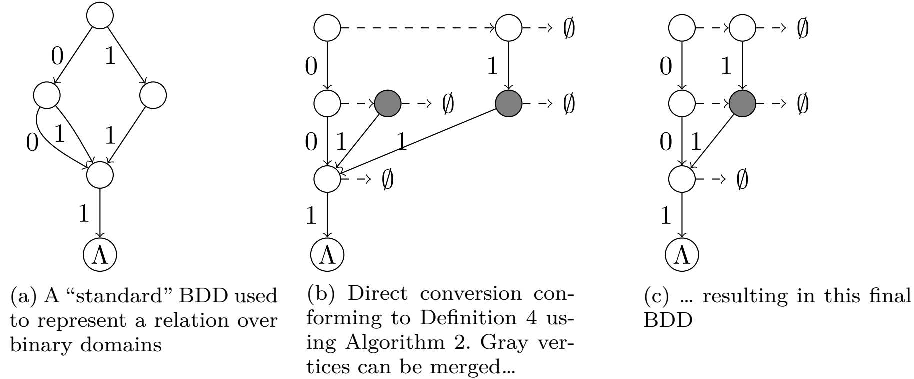
1 **if** $M = \Lambda$ **then return** Λ
2 **else if** $ts \neq Ts(M)$ **then**
3 $\quad Ts(M) \leftarrow ts$
4 $\quad v \leftarrow \{\}$
5 $\quad$ **foreach** *non-empty* $M[i]$ **do**
6 $\quad\quad v \leftarrow (i, \text{MDDtoBDD}(M[i], ts), v)$
7 $\quad Last(M) \leftarrow v$
8 **return** $Last(M)$

represent edges, and they can lead to better compression of the data structures, as we will show later. Converting a MDD to a BDD can be done using Algorithm 2, which is quite straightforward. Timestamps and the *Last* data structure are simply an implementation of memoization [28]. The timestamp is incremented before each call to the algorithm, as for *MDDC*. When a vertex is processed, the result is stored in *Last* and the vertex' timestamp is updated. If the same vertex is encountered twice in the same run of the algorithm, we can reuse the same result from *Last*. This ensures that each vertex is processed only once per run of the algorithm. Access to the *Last* data structure can be implemented in constant-time, using e. g., a hash table indexed with an unique identifier for each vertex. The hash table can also be used to replace the timestamp in order to achieve a purely functional implementation, but the resulting algorithm is unfortunately much slower.

The order in which variables appear in the BDD will be the same as in the original MDD. The order in which values appear depend on the order in which MDD outgoing edges are processed on Line 5 of the algorithm, which typically is the natural (lexicographic) order of the values. The way v is built on Line 6 actually *reverses* the order in which the values appear in the BDD w.r.t. the order in which edges are processed.

Algorithm 2 runs in $\Theta(d \cdot |V(M)|)$ assuming that the loop on Line 5 requires to iterate over all the potential d outgoing edges of the current vertex. This can be optimized by using a linked list or a hash table to represent outgoing edges, but as the algorithm is typically ran only once per MDD, this is not crucial.

An example is given on Figure 3. Figure 3a is the direct conversion from the MDD of Figure 1b. The example shows that this representation allows further reduction of the MDD, as the two nodes labelled in gray on Figure 3a are equivalent and can be merged, resulting in the BDD of Figure 3b. This additional reduction can be performed easily using standard BDD reduction algorithms (e. g., from Bryant [8]). Algorithm 2 generates



(a) A "standard" BDD used to represent a relation over binary domains

(b) Direct conversion conforming to Definition 4 using Algorithm 2. Gray vertices can be merged…

(c) … resulting in this final BDD

Fig. 4: Comparison between standard BDD and Definition 4 BDD

for B exactly one non-terminal vertex for each edge of M, and the additional reduction can reduce this number. If we take into account the two terminal vertices Λ and $\emptyset$, we have the following properties:

Property 2 B is the BDD obtained by applying Algorithm 2 and possibly a reduction algorithm to a MDD M. We have:

$$|V(B)| \leq |E(M)| + 2$$

The additional reduction allowed by the conversion from MDD to BDD is worthy for processing algorithms such as *BDDC2* which is detailed hereafter. Let us recall that the order in which variables are considered in the MDD has an important impact in the size of the reduced MDD, and that finding the optimal variable order is a NP-hard problem [6]. When the MDD is converted to a BDD, the order of the variables is kept, so the reduction of the BDD is at least as good as that of the MDD. The quality of the *additional* reduction, however, depends on the ordering of values in the domains. Indeed, the additional reduction depicted on Figure 3b is only possible if a is after b in the domain of Y. Would a be after c in the domain of Z, then an additional merge is possible. Finding a way to order both variables and domains in order to reduce the size of the BDD is probably worth investigating.

In the obtained BDD, each non-terminal vertex has two children, and terminal vertices have none, so we have this additional property:

Property 3 B is the BDD obtained by applying Algorithm 2 and possibly a reduction algorithm to a MDD M. We have:

$$|E(B)| \leq 2 \cdot |E(M)|$$

On Figure 1b, the MDD has 8 vertices and 11 edges. It can be converted to the BDD of Figure 3b, which has 12 vertices and 20 edges.

We would like to emphasize here that the BDD obtained from Definition 4 and Algorithm 2 are *not* equivalent to "standard" MDDs that would be used to represent relations over binary domains, as described by e. g., Cheng and Yap [10]. Indeed, the standard definition of BDD usually states that the two children of any vertex correspond respectively to one of the two possible values of the binary domain (usually *true* and *false*). This does not have the same semantics as our definition of a BDD, where the "child" and "sibling" successors of a vertex have different meanings. This illustrated by Figure 4a, where a "standard" BDD (composed of 5 vertices and 6 edges) representing a binary relation is converted to another BDD conforming to Definition 4 (composed of 8 vertices and 12 edges). The obtained BDD can be further reduced to 7 vertices and 10 edges.

Algorithm 3: BDDC2($B = (i, child, sibling)$, ts, $Invalid$, p, $Supported$)

```
   input  : Cf. MDDC algorithm.
   output : true iff the BDD contains at least one support.
 1 if (B = Λ) ∨ (ts = Ts(B)) then return true
 2 else if B ∈ Invalid then return false
 3 else
 4     X ← 𝒳[p]
       // A vertex is valid if either child or sibling is valid
 5     valid ← false
 6     if i ∈ dom(X) ∧ BDDC2(child, ts, Invalid, p + 1, Supported) then
           // Child is valid, so current vertex is valid
 7         valid ← true
 8         Supported(X) ← Supported(X) ∪ {i}
 9         if (p + 1 = δ) ∧ (Supported(X) = dom(X)) then
10             δ ← p
11         else
               // Proceed to sibling, return value is not used
12             BDDC2(sibling, ts, Invalid, p, Supported)
13     else
           // Child not valid, so check sibling
14         valid ← BDDC2(sibling, ts, Invalid, p, Supported)
15     if valid then Ts(B) ← ts
16     else Invalid ← Invalid ∪ {B}
17     return valid
```

Hadzic, Hansen, and O'Sullivan [19] proposed another way to convert a MDD to a BDD, by converting multi-valued variables to binary variables using well-known *log* or *direct* encodings (respectively), and MDDs are converted accordingly. This results in BDDs having either $\Theta(\log d \cdot |E(M)|)$ or $\Theta(d \cdot |E(M)|)$ (resp.) vertices. This is worse than the bound we obtained (cf Property 2). Srinivasan et al. [35] also propose an MDD-to-BDD conversion using a technique very similar to log-encoding. In summary, to the best of our knowledge, it is the first time that a linear transformation from MDD to BDD is proposed.

5 *BDDC2*: Arc consistency for BDD constraints

We adapted *MDDC* to perform on the BDD conversion of the MDD. We obtain *BDDC2* (Algorithm 3).

Property 4 *BDDC2* applied on a BDD B has a complexity in $\Theta(|V(B)|)$.

Proof First, we prove that *BDDC2* is in $O(|V(B)|)$.

1. Each vertex is parsed at most once thanks to the timestamp, and
2. all operations on a given vertex (i. e., each line of Algorithm 3) can be implemented in constant-time. Sets can be implemented using sparse sets, which allow constant-time add (Lines 8, 16) and check (Lines 2, 6) operations [7].[8] We have $Supported(X) \subseteq \mathrm{dom}(X)$, so the set comparison operation on Line 9 can be done by checking the domains sizes only, which can be maintained.

Now, we prove that *BDDC2* is in $\Omega(|V(B)|)$: the algorithm processes every vertex in the graph, except if it is in the *Invalid* set (Line 1), or if its depth is higher than δ (Line 9: this means that all variables at deeper levels have their whole domain supported) and the variable corresponding to its depth has its whole domain supported. In the worst case, *Invalid* is empty (all vertices are supported) and one domain value is only supported by the deepest vertex in the graph. In this case, $\delta = k$ and the check on Line 9 will fail. □

[8] Alternative implementations of sets can also be used provided they have expected performance similar to sparse sets.

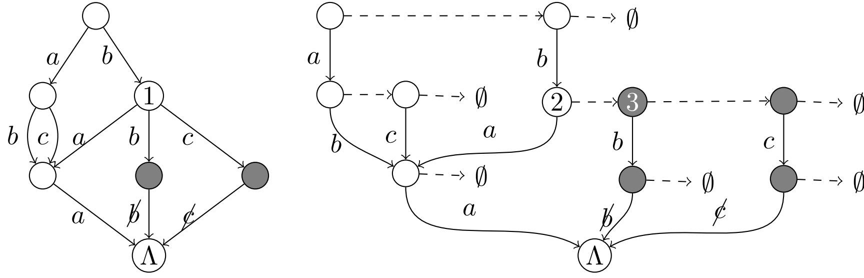


Fig. 5: MDD and BDD representation of the same relation with b and c removed on the third level. Grayed out nodes can be labelled as invalid by *MDDC* and *BDDC2*, respectively. Vertices 1 and 2 represent the same relation, yet Vertex 1 has three outgoing edges, two of them leading to invalid vertices. Vertex 2 has only two outgoing edges, one of which leading to the invalid vertex 3.

BDDC2 is a clear improvement over *MDDC* for three reasons:

1. The time complexity is better, as we stated in the previous section. The loop of Line 6 of Algorithm 1, which iterated over domain values, is replaced by the recursive calls on Lines 12 and 14 in Algorithm 3, which iterates over BDD vertices. As we stated in the beginning of this section, it allows to turn *MDDC*'s complexity of $\Theta(d\cdot|V(M)|)$ into *BDDC2*'s $\Theta(|V(B)|)$, where M and B represent equivalent relations. Combining Properties 1 and 2, we have $|V(B)| - 2 \leq |E(M)| \leq d \cdot |V(M)|$.

Note, however, that this statement is mostly true at the top of the search tree. Indeed, when values are removed from the domains of variables following search hypothesis, backtracks and propagation, d might be significantly reduced. However, we show experimentally in Section 7 that iterating over outgoing edges is much more efficient in practice than over domain values. In any case, the algorithm *BDDF* described in the next section was designed to address this issue.

2. BDD benefit from additional reduction, as seen on Figures 3 and 4a. Algorithm 2 transforms a MDD into a BDD such that $|V(B)| = |E(M)| + 2$. The obtained BDD can then be further reduced (Property 2). Most BDD processing algorithms, including *BDDC2* and *BDDF* discussed hereafter, have a complexity in $\Theta(|V(B)|)$. Reducing the number of BDD vertices is the easiest way to improve the worst-case behavior of these algorithms.

3. Vertices in a BDD have a finer grain than vertices in a MDD. *MDDC* (resp. *BDDC2*) label invalid vertices, which means that the value of $|V(M)|$ (resp. $|V(B)|$) decreases along a branch of the search tree. *BDDC2* is able to label vertices more acutely, and thus skips larger parts of the graph. An example is given on Figure 5. After one execution of *MDDC*, gray nodes are labelled as invalid and will not be considered again in this branch of the search tree. Vertex 1 cannot be marked as invalid because value a on the second level is still valid. This will require all further calls to *MDDC* to check the validity of values b and c. With *BDDC2* and the BDD representation, we can label Vertex 3 as invalid, which allows to skip both values on each execution of the algorithm in the current search tree branch. The rationale is that under the BDD form, *BDDC2* can invalid a BDD vertex if all *remaining* domain values are invalid, whereas under the MDD form, *MDDC* can only invalid a vertex when *all* children are invalid.

Algorithm 4: filterBDD($B = (i, \mathit{child}, \mathit{sibling}), \mathit{ts}, \mathit{Modif}, p$)

input :
B is a BDD vertex which represent a set of allowed instantiations.
ts is the current value of the timestamp s.t. $\mathit{ts} = \mathit{Ts}(B)$ iff B has already been processed in this run of the algorithm. In this case, $\mathit{Last}(B)$ contains the result of the previous computation.
Modif is the list of modified variables.
p is the index s.t. $\mathcal{X}[p]$ is the variable corresponding to the current level of the BDD.

output : A BDD equivalent to B with all invalid instantiations removed.

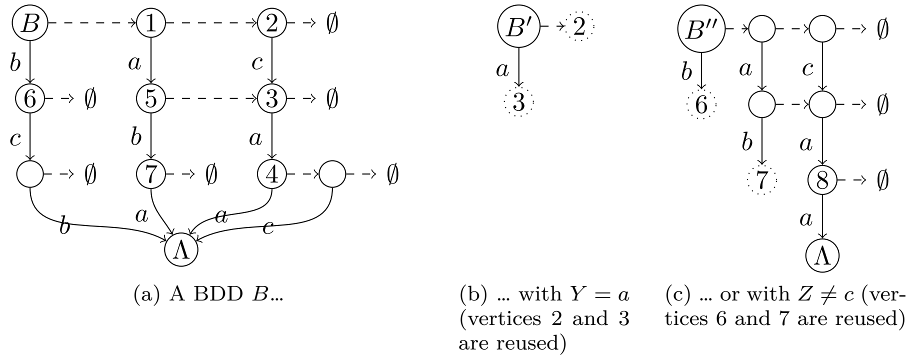
```
1  if (Modif = {}) ∨ (B = {}) ∨ (B = Λ) then return B
2  else if ts ≠ Ts(B) then
3      Ts(B) ← ts
4      s ← filterBDD(sibling, ts, Modif, p)
5      X ← 𝒳[p]
6      if i ∈ dom(X) then
7          c ← filterBDD(child, ts, Modif − X, p + 1)
8          if c = {} then
9              Last(B) ← s
10         else if (c = child) ∧ (s = sibling) then
11             Last(B) ← B
12         else
13             Last(B) ← (i, c, s)
14     else
15         Last(B) ← s
16 return Last(B)
```

6 BDDF (BDD-Filtering)

We designed *BDDF* so as to bring together the advantages of both *MDDC*/*BDDC2* and *STR*, and acts as a replacement for both algorithms. The idea is to "filter" the invalid parts of the relation, similarly to the recent algorithm *MDD-4R* [33] which was developed in parallel to this work.[9] Contrary to *MDD-4R*, we do not rely on "trailing queues" to restore deleted parts of the graphs upon backtracking, as our software rely on functional programming frameworks. *Persistent* data structures [14], sometimes called *immutable*, are standard in functional programming languages such as ML, Haskell, or Scala [31]. These languages dissuade or forbid to change the content of data structures during the execution of a program. Adding or updating information in a data structure requires to "copy" all data, changing appropriate nodes in the process. Original data is left untouched and can still be reached if required. Data structures are often composed of lists or trees (or, more generally, directed acyclic graphs), and large parts of the graphs can generally be reused instead of copied. Guarantees in the absence of side effects usually lead to less bug-prone programming, especially when several threads or processes access the same data simultaneously. Such data structures are useful in the context of constraint solvers, as they allow easy implementation of backtracking, but also have nice perspectives in the design of non-chronological backtracking and multi-core processing.

Algorithm 4 performs the filtering of a BDD, so that each remaining vertex is part of at least one support. It is very close to the *restrict* operator originally proposed by Bryant [8], but takes into account the "value" associated to vertices and the different semantics of its "child" and "sibling" successors. Contrary to the *separation problem* [11], filtering a BDD B has a complexity linear in the number of vertices and the obtained BDD B' cannot be larger than the original ($|V(B')| \leq |V(B)|$). A "new" BDD is returned, but in most cases original sub-graphs of the original BDD can be kept. As stated in Def. 4 on page 7, a BDD vertex is a (*value*, *child*, *sibling*) triple: the new vertices are created only on Line 13, after considering all other options.

[9] We published the idea of filtering MDD in French [39] in 2013.



(a) A BDD B...

(b) ... with $Y = a$ (vertices 2 and 3 are reused)

(c) ... or with $Z \neq c$ (vertices 6 and 7 are reused)

Fig. 6: Filtering a persistent BDD.

The run-time of the algorithm is similar to that of *STR(1)*, with the additional compression gains brought by BDDs: removing a single vertex from the BDD is equivalent to removing a potentially exponential number of lines in the corresponding table. Information on modified variables is used: *Modif* is the set of variables which have lost values since the last filtering operation, as *STR2* does. Unfortunately, for our algorithm we have a restriction which will be detailed later. Variables are removed from *Modif* when they are encountered, during the recursive call on Line 7.[10] If *Modif* is emptied, it means that no variable below X has been modified. This implies that all remaining vertices are still valid, so no more filtering is necessary and no processing is done at further levels. If a vertex is invalid or has no children, it can be removed from the BDD and its sibling is directly returned (Lines 9 or 15). A new vertex can be created every time a vertex is fully processed (Line 13). However, there are many cases where no filtering is done on a valid vertex' child and sibling. In this case, the original vertex can be returned (Lines 10–11).

Example 3 We describe one run of `filterBDD` (Algorithm 4) on the BDD from Figure 6a, under the hypothesis that $Y = a$. We have $\mathit{Modif} = \{Y\}$. $\forall$ vertex v, $\mathit{Ts}(v) = 0$. $\mathit{ts} = 1$.

The algorithm starts with Vertex B. *Modif* is not empty, current timestamp is different from B's timestamp, so the timestamp is updated and a recursive call is done on Line 4 for Vertex 1, which subsequently yields a recursive call for Vertex 2, and then its empty sibling, which is returned. Algorithm continues to check Vertex 2's child. Presence of c in $\mathrm{dom}(X)$ is checked on Line 6 (this can be optimized out since we know from *Modif* that X is unchanged). Recursive call is now done on Line 7 to Vertex 3.

Vertex 3's sibling is the empty BDD which requires no processing. $a \in \mathrm{dom}(Y)$, so a recursive call is performed on Vertex 4 with $\mathit{Modif} = \{Y\} - Y = \{\}$. End condition of the recursive algorithm is encountered on Line 1, so we get back to Vertex 3: its child will be unchanged. Note that this end condition saves us 4 recursive calls. Since Vertex 3's sibling is also unchanged, the case of Line 10 is encountered and we can return Vertex 3 directly. We record this result in the *Last* data structure (Line 11). We get back to Vertex 2 in the call stack.

Vertex 2's sibling and child are both unchanged. Case of Line 10 is encountered again, and Vertex 2 can be returned. We get back to Vertex 1 in the call stack.

A recursive call is done to Vertex 5, which yields a call to Vertex 3. As Vertex 3 has already been processed in this run of the algorithm, we know from the timestamp that we can directly return the previously computed result from *Last*. Back to Vertex 5, the check

[10] If $X \notin \mathit{Modif}$, there is no need to control validity of values on Line 6 for this vertex and all its siblings. The removal of X from *Modif* on Line 7 can also be omitted in this case. These simple optimizations are not included in the algorithms to improve concision and preserve readability, but have some impact on the speed of processing.

Algorithm 5: seekSuppBDD($B = (i, \mathit{child}, \mathit{sibling})$, ts, $\mathit{Supported}$, p, Seek)

input :
B is a BDD vertex which represent a set of supports.
ts is a timestamp s.t. $ts = Ts(B)$ iff B has already been processed in this run of the algorithm.
$\mathit{Supported}$ is an application of each variable of $\mathcal{X}$ to the set of values that have at least one support.
p is the index s.t. $\mathcal{X}[p]$ is the variable that corresponds to the current level of the BDD.
$\mathit{Seek} \subseteq \{1 \ldots k\}$ is the set of variable indices (from $\mathcal{X}$) whose domain contains values that may be unsupported.

output : $\mathit{Supported}$ and Seek are updated by side effect.

```
1 if (B ≠ {}) ∧ (∃p′ ∈ Seek | p′ ≥ p) ∧ (ts ≠ Ts(B)) then
2     Ts(B) ← ts
3     if p ∈ Seek then
4         X ← 𝒳[p]
5         Supported(X) ← Supported(X) ∪ {i}
6         if Supported(X) = dom(X) then
7             Seek ← Seek − p
8     seekSuppBDD(sibling, ts, Supported, p, Seek)
9     seekSuppBDD(child, ts, Supported, p + 1, Seek)
```

on Line 6 is false, as $Y = a$. Vertex 5 can be filtered out and Vertex 3 is directly returned on Lines 15–16. We get back to Vertex 1 in the call stack.

Vertex 1's sibling is unchanged, but its children is now Vertex 3 instead of Vertex 5. We have to create a new vertex B' on Line 13. We get back to Vertex B in the call stack, which yields a recursive call to Vertex 6. Vertex 6's sibling is of course unchanged, on Line 6 we detect that $c \notin \mathrm{dom}(Y)$, which means that Vertex 6 has no child. Its sibling, the empty BDD, is returned. We get back to Vertex B.

Its new child is the empty BDD. Condition of Line 8 is encountered, so B can be filtered out and its sibling B' is returned (Figure 6b). The call stack is now empty and the algorithm ends.

This example is a rather good case (memory-wise), as we can obtain a filtered copy of a BDD by generating only one new vertex B'. The hypothesis $X \neq b$ (resp. $X = c$) would be an ever better case as it would only imply to consider Vertex 1 (resp. Vertex 2) as the new root of the BDD. In general, changes deep in the graph are costlier. Figure 6c is the result of the filtering of B under the hypothesis that $Z \neq c$: here, six new vertices must be created. In the worst case, deleting a vertex v may require to recreate all vertices present in any path from the root of the BDD to v. Experiments in Section 7 show that this can be an issue.

In some cases, two vertices can become identical after filtering (e. g., Vertices 7 and 8 on Figure 6c), so they can be merged. However, detecting this efficiently requires to maintain a caching data structure (e. g., a hash table) of all vertices over all branches of the search tree, and free the cache of outdated vertices upon backtracking. The garbage collector and "weak references" provided by the Java VM allow to implement these features quite easily, but querying the cache at each vertex instantiation is too slow. A quick experiment showed that this "improvement" leads to a $3 \sim 6\times$ slower algorithm which requires $\approx 3\times$ more memory than plain *BDDF*. Maybe this can be improved.

Once the BDD has been filtered, it only contains supports, i. e., all values in the BDD are supported. Algorithm 5 can be called to identify all values that appear in the BDD. Other values can be removed from the domains. This part of the algorithm is similar to *BDDC2*, but since all values are valid, some optimizations can be implemented. Timestamps are used to avoid traversing the same vertex twice (Lines 1 and 2). The *Seek* set is used to early-end the traversal when supports are found for entire domains of all variables below the current level of the BDD. It is initialized to $\{1, \ldots, k\}$ before each call to the algorithm. This is similar to the δ of *BDDC2* but with a finer grain, as it allows us to

skip Lines 4–7 if all values of the current level have been found. The largest value in *Seek* should be maintained so that the end condition on Line 1 can be checked in (amortized) constant time.

BDDF consists of applying Algorithms 4 and 5 successively to detect all supports and restrict the BDD size to bring incrementality. Unsupported values are removed from the domain of the variables. Here follows a more detailed complexity analysis:

Property 5 The worst-case time complexity of one run of Algorithm 4 (`filterBDD`) for a BDD B is $\Theta(|V(B)|)$.

During the filtering process, every vertex of the BDD is traversed at most once, thanks to the use of timestamps. Unfortunately, the *incremental* complexity of BDD filtering is not as good as for *STR2*: we cannot completely ignore the unmodified variables, since we need to traverse the corresponding levels to reach deeper vertices. Value validity checks, however, can be skipped if we know that the variable has not lost values. As each variable can only be modified d times, each edge will trigger at most d value validity checks in one branch of the search tree.

Property 6 The worst-case time complexity of one run of Algorithm 5 (labelling supported values) for a BDD B is in $\Theta(|V(B)|)$.

All tests at Line 1 of the algorithm can be performed in constant time. The *Seek* set can be implemented using a bit vector and a pointer on the largest value. Checks on Line 1 and 3 can thus be made in $O(1)$. Removing any value but the largest is also $O(1)$. When the largest value is removed, the bit vector is parsed from the end to find the new largest value. The vector is parsed at most once during the whole algorithm, thus an additional amortized cost of $O(k)$ which can be discarded.[11]

As $Supported(X) \subseteq \text{dom}(X)$, the test on Line 6 can also be done by comparing cardinalities. This can be done in constant time by maintaining the size of the sets.

Property 7 If all e constraints of a CSP/COP model are represented by BDDs of size less than $|V(B)|$ and propagated with *BDDF*, the worst-case *time* complexity of performing all constraint propagations over a branch of the search tree is in $O(ekd \cdot |V(B)|)$.

On one branch of the search tree, for each of the e constraints, the filtering algorithm can be called up to kd times, i. e., every time a value is removed from the domain of a variable in the scope of each constraint. Although we added several incrementality features to Algorithms 4 and 5, in the worst case, every vertex of the BDD might be traversed at each call of each algorithm. This results in the $O(ekd \cdot |V(B)|)$ worst-case time complexity.

However, the number of *value validity checks* performed by *BDDF* is in $O(ed \cdot |V(B)|)$ for a branch of the search tree, as we can skip checks on a level when a variable is not modified.

We recall here the worst-case incremental complexity of *STR2* which is in $O(ek^2d\lambda)$, with $|V(B)| \leq k\lambda$. *STR2* performs at most $O(ekd\lambda)$ value validity checks on one branch of the search tree.

Property 8 If all e constraints of a CSP/COP model are represented by BDDs of size less than $|V(B)|$ and propagated with *BDDF*, the worst-case *space* complexity of performing all constraint propagations over a branch of the search tree is $O(ekd \cdot |V(B)|)$.

In the worst case, filtering a single BDD may require to create $O(|V(B)|)$ vertices, i. e., copy most of the BDD. Indeed, as we stated before (e. g., Figure 6c), deleting a vertex requires to recreate all paths to the deleted vertex' left sibling. We observe experimentally,

[11] $|V(B)| < k$ requires that the BDD is shallower than the arity of the constraint. This can happen only when all values of the deeper variables are supported by all instantiations of the other variables, which means that the constraint can be easily reformulated to omit these variables. Anyway, this issue did not appear in our experiments.

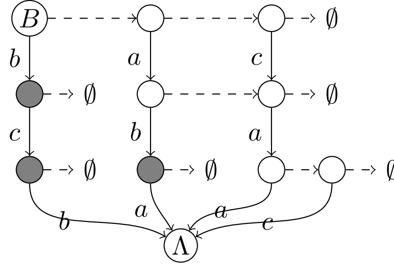


Fig. 7: Vertices in white cannot be invalidated by *BDDC2* as long as the instantiation $\langle X = c, Y = a, Z = c\rangle$ is valid.

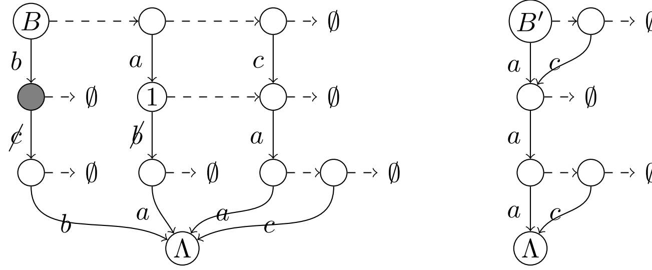


Fig. 8: State of the BDD after the hypothesis that $Y = a$ and one execution of *BDDC2* (left, greyed out vertex is labelled as invalid) and *BDDF* (right).

however, that practical results are very far away from theoretical worst-cases. For example, on experimental results on Figure 12, memory consumption of *BDDF* is at worst 3 times more than *BDDC2*, where in theory it could be up to 50.

MDDC and *BDDC2* implement incrementality by labelling invalid vertices, maintaining this information down the branch of the search tree. A vertex is "valid" iff it is a leaf or at least one edge from that vertex points to a valid vertex. This means that a single valid instantiation can keep "alive" up to dk vertices (cf Fig. 7 for an example). *BDDF* improves on this by removing individual invalid vertices, such that the number of remaining vertices is minimal, i. e., at most k vertices per valid instantiation.

Observe Figure 8: under the hypothesis that $Y = a$, *BDDF* filters the BDD to B'. *BDDC2* keeps the original BDD B, but labels the grey vertex as invalid. Some vertices with invalid values remain: $X = b$ has no support and is deleted by the algorithms. It is invalid, but node B remains because its sibling is still valid. Same for node 1. Experimental results confirm this behavior (cf e. g., Figure 10). Moreover, *BDDF* takes advantage of information about modified variables provided by the underlying propagation framework. Efforts in modifying *BDDC2* to exploit this information were unsuccessful. Indeed, using it efficiently requires the two passes made by *STR2* or *BDDF*: one for filtering (where we can use information on modified variables), and one for labelling supported values. As invalid edges remain in the BDD with *BDDC2*, domain validity checks must be done twice, which slows down the process too much. Gange, Stuckey, and Szymanek [16] proposed to use *watches*: a valid MDD edge is associated to each value. As long as the edge is valid, the value has a support. This allows to skip many support searches. This technique cannot be used with *BDDF* because of the new vertices created during search. Moreover, as with *STR2*, the bottleneck of our algorithm is rather in the filtering process.

On the other side, *BDDF* requires to allocate new vertices during the filtering. Although this has no impact on theoretical time complexities, memory allocations can have

a non-negligible cost in practice, and also require to take care of de-allocation. Garbage collecting is quasi-mandatory to do this properly. Spatial complexity may also be a problem in some configurations. Experimental results hereafter show that although benchmarks exists where *BDDF* is the fastest algorithm (especially on randomly generated problems), it can be beaten by *BDDC2* when domains are small, and by *STR2* when the obtained compression is low (unstructured or very tight constraints). *STR2* can also have an advantage when the arity is high because of the restriction on the use of *Modif* we just described.

Finally, we state that the filtering process defined for BDD can also be applied on MDD directly, resulting in the so-called *MDDF* algorithm which was described in [40] (in French, same authors). We found out that *MDDF* was outperformed by *BDDF* in almost every aspect, so we discarded it from this article for the sake of conciseness.

7 Experiments

All algorithms were implemented using the Scala 2.11 programming language [31] and the Concrete 3 constraint programming platform [38]. We used a Java 8 Runtime Environment provided by the Oracle 64-Bit Server Virtual Machine. 4 GiB heap space was allowed. The operating system uses a Linux 4.3.6-x86_64 kernel running on standard desktop computers with Intel Core i5-3470 CPU @ 3.2 GHz and 8 GiB RAM. *STR2* is used for positive tables, and one of *MDDC*, *BDDC2* or *BDDF* is used to propagate constraints defined by a MDD. For brievety, one call to either algorithm for one constraint (i. e., a constraint propagation) is called a *revision* in this section. MDDs are converted to tables or BDDs, or vice versa, to apply the requested algorithm. The Scala language cannot handle arrays larger than $2^{31} - 1$ elements (i. e., the maximum value of a 32-bit signed integer). When converting a MDD to a flat table, if λ is higher than this number, an *out-of-memory* error is thrown even though the data might have (unlikely) been represented within 4 GiB memory using smarter (but slower) data structures. Most problems use additional propagators for binary or global constraints provided by Concrete 3.

The solver was carefully parameterized so that the very same branching (see below) and propagation ordering (FIFO) strategies are used in all configurations, i. e., search trees are strictly identical.

7.1 Randomly generated problems: mini-benchmarks

We implemented a pseudo-random MDD generator with the four parameters (d, k, λ, q) as proposed by Cheng and Yap [9]. d is the domain size, k is the depth of the MDD, λ is the number of tuples allowed by the MDD, and q is a "structural factor" which defines the probability that a vertex is identical to another. Pseudo-random numbers are obtained from the JDK's builtin linear congruential generator. Random reduced MDD can be generated in $\Theta(k\lambda)$ using Bentley and Floyd [3]'s algorithm to generate random sets, combined with the "grouping" strategy described in Sect. 3 on page 3 and Cheng and Yap [9]'s `reduce` function. The structural factor is enforced by traversing the generated MDD and replacing each vertex with a previously encountered one of the same depth with a probability q. Alternatively, MDD can be generated with the parameters (d, k, l, q). We recall that l is the *looseness* of the constraint, i. e., the proportion of instantiations allowed: $\lambda = l \cdot d^k$. MDD are then "flattened" to tables in $\Theta(k\lambda)$ to apply *STR2*, or converted to BDD in $\Theta(d \cdot |V(M)|)$ using Algorithm 2 and Bryant [8]'s reduction algorithm to apply either *BDDC2* or *BDDF*. The time to generate and convert data structures are not accounted in experimental evaluations.

First, we evaluated the space required to represent a relation using either a reduced MDD, implemented using arrays to represent outgoing edges, and its conversion to reduced

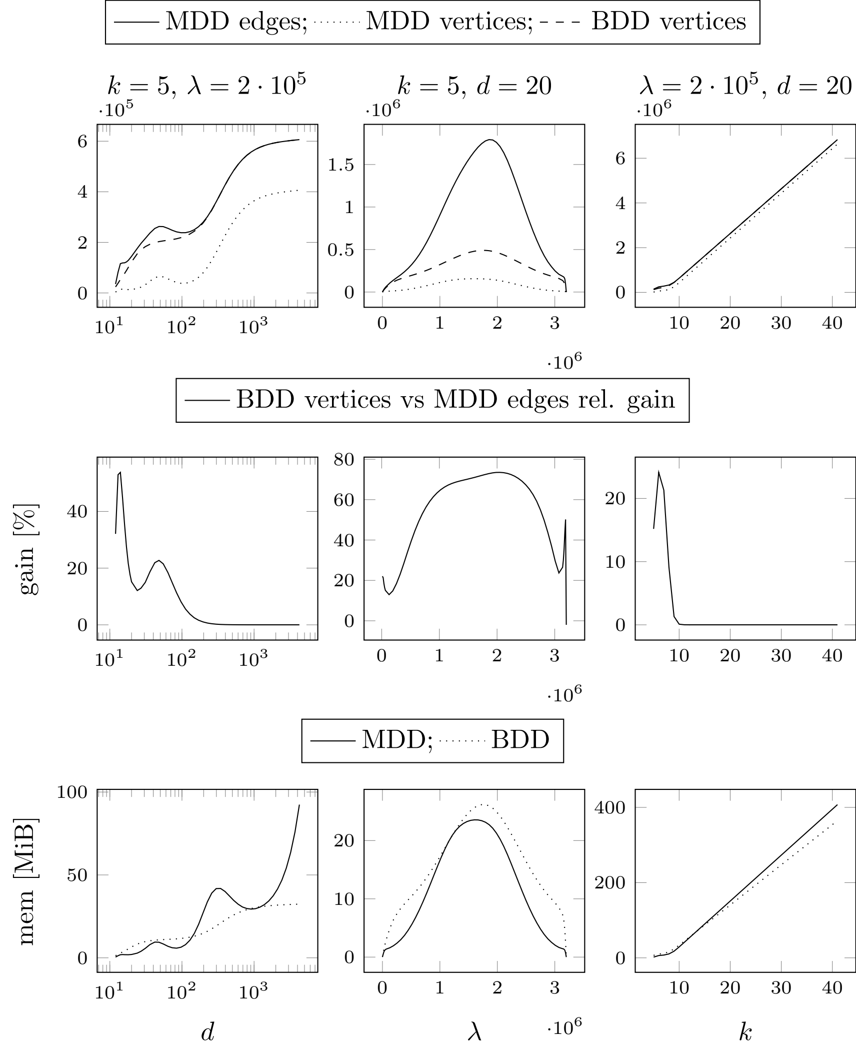


Fig. 9: Space required to represent a relation using a MDD or a BDD with $q = 0\,\%$.

BDD. We measured the number of vertices and edges required, as well as the actual memory occupied by our implementations of the data structures. In this experimentation, we generated pseudo-random MDD around the point $(d, k, \lambda) = (20, 5, 2 \times 10^5)$ and made each parameter vary individually around it. We chose $d = 20$ because it was near the average size in our bank of structured problems (cf Section 7.3). We chose $k = 5$ because we wanted (in the next set of experiments) to plot the impact of varying l on the full range from 0 to 100 %, which results in too large tables/MDDs with higher arities. Similarly, choosing values for λ higher than 2×10^5 resulted in faster memory-outs with only changes in scale on the plots.

Results are presented on Figure 9. Each point in the plots is the average over 10 generated relations. On the left column, we make d vary, keeping k and λ constants. On the middle column, we make λ vary. Finally, on the rightmost column, we make k vary. On the three topmost plots, we plotted the number of edges and vertices of BDD and MDD. The most pertinent comparison is between the number of MDD edges and BDD vertices: the difference between the two values corresponds to the additional reduction that can

be obtained after a MDD is converted to a BDD (cf Figure 4a). BDD edges is merely a factor 2 over BDD vertices, we chose not to plot it to improve readability. The average relationship between MDD edges and MDD vertices is more complicated but not so much relevant for our analysis, as no algorithm or data structure depends only on the number of MDD vertices; the space complexity of MDD and the time complexity *MDDC* are both in $\Theta(d \cdot |V(M)|)$ and Property 1 states that $|E(M)| \leq d \cdot |V(M)|$).

We plotted the reduction gain obtained after the conversion to BDD[12] separately on the three intermediate plots. The gain seems to reach an extrema of about 70 % when the looseness of the constraint approaches 50 %. Similarly, the gain decreases when d and k increase (for a given value of λ). The lower plots show the actual memory consumption of our implementations. The behavior in $\Theta(d \cdot |V(M)|)$ of MDDs is clearly visible on the bottom left plot: memory continues to increase linearly with d whereas the BDD implementation tends towards a constant. Our implementation of BDD is somewhat memory consuming, because a timestamp and (constant-sized) cache data (*Ts* and *Last* that can be spotted on Algorithms 3, 4 and 5) are associated to each vertex even when not in use. This may be optimized out on more specific implementations, i. e., only *BDDC2* is to be implemented. One can expect up to 30 % less memory required in this case.

We then performed a mini-benchmark to evaluate the performance of our propagators on randomly generated MDDs. In this experiment, we added a set of instances around the point $(d, k, l) = (20, 5, 10^{-4})$. Choosing a value for l implies that λ grows polynomially with d and exponentially with k. To evaluate the incrementality properties of our algorithms, we simulated a search tree branch by applying the following procedure 20 times: the propagator is called, then 20 % of randomly selected values from each of 20 % randomly selected variables are removed. Results are shown on Figure 10 for $q = 0$ and Figure 11 for $q = 50\,\%$. On each figure, top 6 plots are generated using the λ parameter, and bottom 6 use the l parameter instead. Plots are organized similarly to Figure 9. Each point in the plots is the median obtained over 50 runs with the same parameters and different pseudo-random seeds. On plots with $q = 50\,\%$, the obtained MDD/BDD are quite small and some measurements may become imprecise (e. g., what appears as constants on top right plots of Fig 11 are theoretically linear with a very low slope). Unfortunately, it is hard to generate larger data due to the complexity of the random generation algorithms.

These plots show that *MDDC* is greatly outperformed by *BDDC2* on almost every aspect, except memory for smaller domains. *BDDF* is also always faster than *BDDC2* on these plots (this behavior is contradicted in further experiments, see below). *BDDF* or *BDDC2* are faster and less memory consuming than *STR2* when λ or l is large or if there is structure in the MDDs ($q > 0\,\%$): if the tables cannot be compressed significantly, *STR2* should be the better algorithm. However, *BDDF* and *BDDC2* are still competitive, whereas *MDDC* can become very slow when domains are too large.

7.2 Randomly generated problems: full benchmarks

Next, we performed benchmarks on fully solved problems randomly generated using the same technique as above, plus two additional n and e parameters. n defines the number of variables and e is the number of constraints in the problem. The problems confer to model RB [41] and exhibit a phase transition around $l = \exp(-n/e \cdot \log d)$ if $l \geq k^{-1}$. We chose to observe how solving performance evolves when l grows on problems near the phase transition with $(d, k) = (8, 5)$, then increased d and k separately to 10 and 6, respectively. We adjusted n so that problems are not excessively hard to solve, i. e., less than 1,200 s with highest parameters. Finally, we tested with $q = 0\,\%$ and $q = 50\,\%$. To remain near the phase transition, we have to increase e together with l. To minimize rounding errors, we adjusted e and calculated the corresponding l according to Model RB theorems. The

12 $\frac{|E(M)|-|E(B)|}{|E(M)|}$

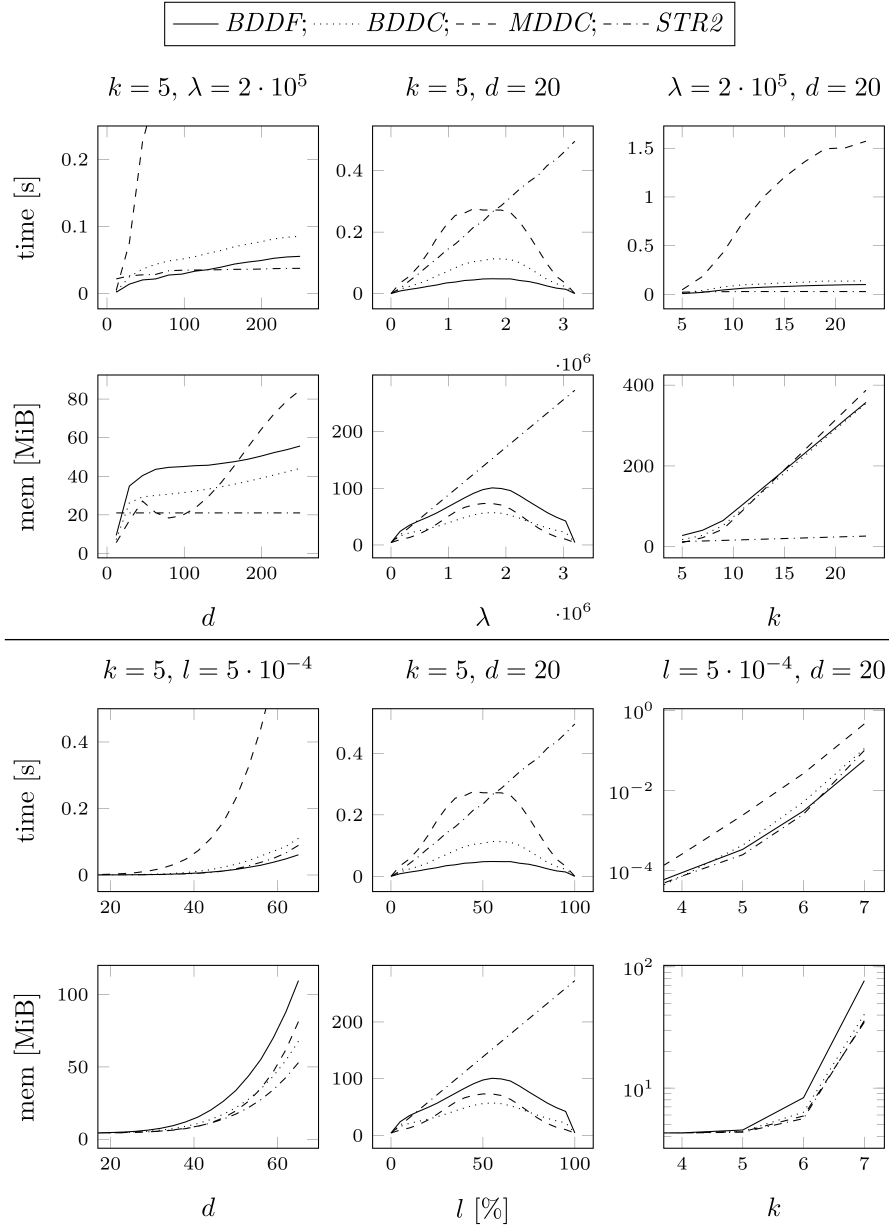


Fig. 10: Time and space to filter 20 levels of a simulated search tree branch with $q = 0\,\%$ (lower is better).

branching strategy used is the $^{dom}/_{wdeg}$ variable ordering heuristic coupled with *random* value ordering heuristic, and geometric restarts. A timeout of 1,200 s was used. Results are shown on Figure 12. Indicative ranges of e and λ are given.

Instead of showing the time to solve the problems, which grows exponentially when e and l increase, we chose to plot the number of revisions per second (i. e., the number of times a revision algorithm is executed per second), which are more readable. When a run timeouts, the number of revisions per second performed before the timeout is kept. Let

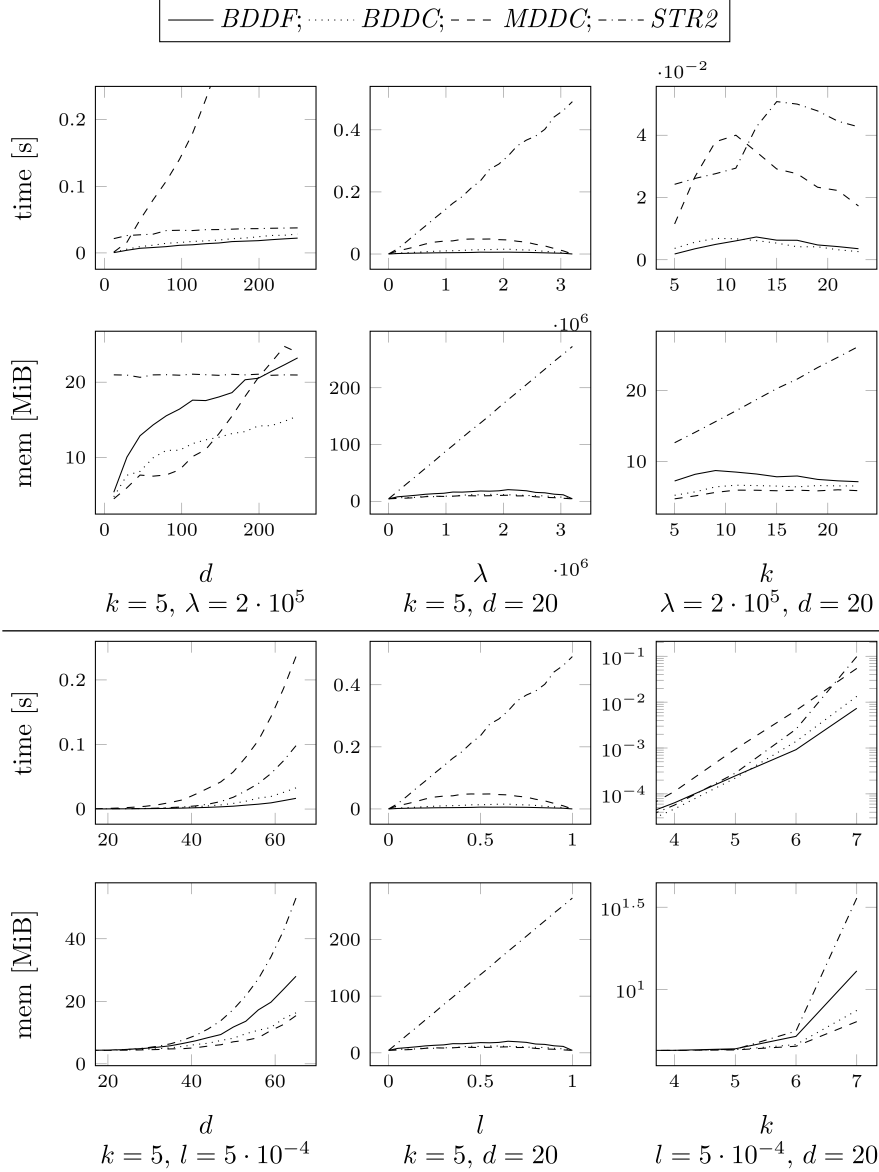
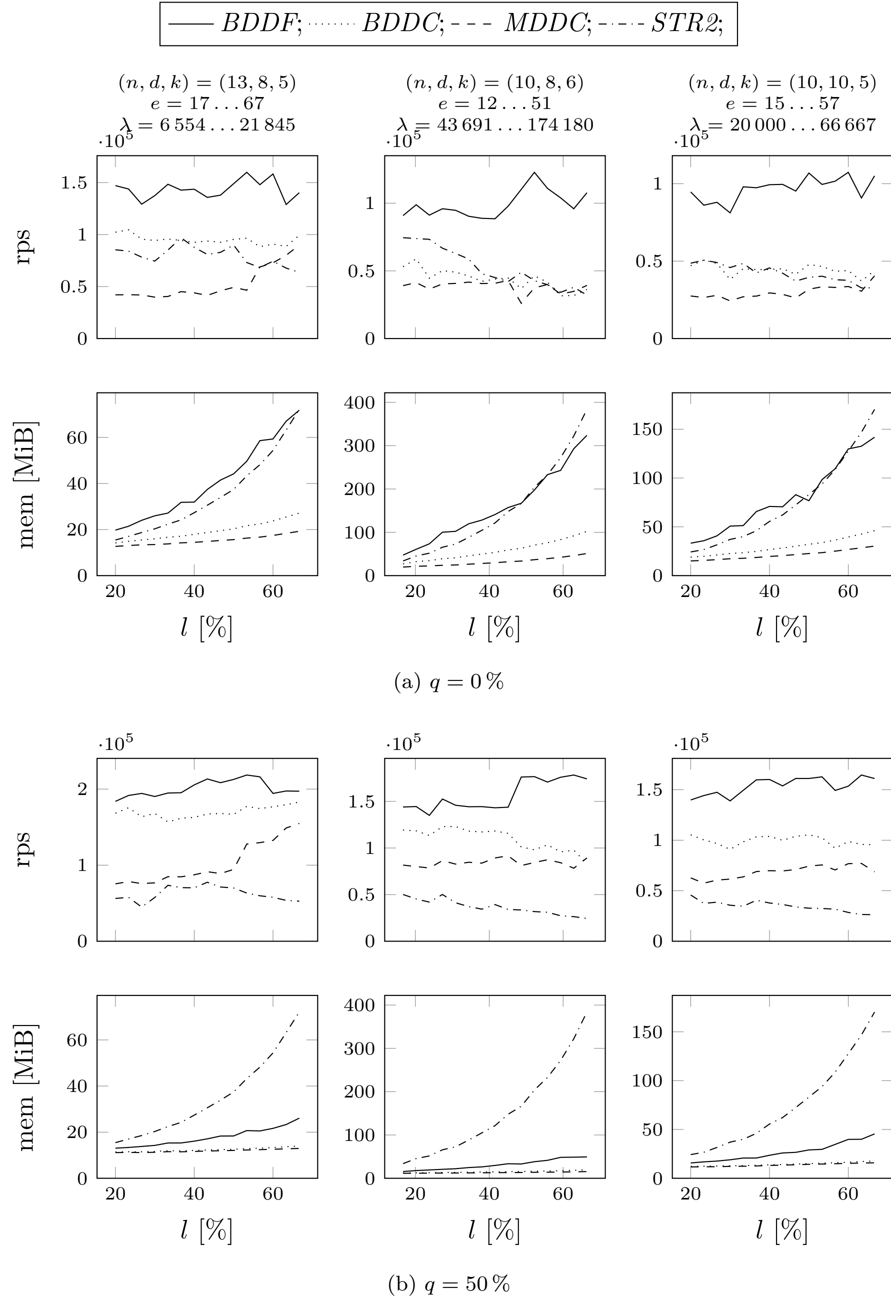


Fig. 11: Time and space to filter 20 levels of a simulated search tree branch with $q = 50\,\%$ (lower is better).

us recall that the search tree and thus the total number of revisions is exactly the same whichever revision algorithm is used. We also plot the memory required by each algorithm.

On these randomly generated problems, *BDDF* is always the fastest algorithm. When $q = 0\,\%$, *MDDC* is always the slowest, except when l is very high, where it collides with *BDDC2* and *STR2*. This is theoretically sound as high l implies that the number of outgoing edges of a MDD vertex is almost always equal to d. Memory-wise, *BDDF* and *STR2* are both high, whereas *BDDC2* and *MDDC* are very competitive. Note that *STR2* is nearly as efficient as *BDDF* when arity is high and looseness is low, i. e., tables are rather small. Indeed, high arities benefit to *STR2* due to its better incrementality properties, and small random tables are not much compressed by MDD or BDD.



(a) $q = 0\,\%$

(b) $q = 50\,\%$

Fig. 12: Propagation speed (revisions per second, higher is better) and memory required (lower is better) to solve randomly generated problems. Each point is a median over 50 problems.

	#	d	k	λ	l [%]	$d\,\lvert V(M)\rvert$	$\lvert E(M)\rvert$	$\lvert V(B)\rvert$
aim	72	2	3	6.8	85.5	12.2	9	10
bdd	70	2	16	32.3 k	21.5	26 k	26 k	20 k
carseq	40	2	248	4×10^{83}	0.5	4.8 k	3.4 k	3.4 k
crossword	687	25	9	14.7 k	0.1	247 k	19 k	16 k
kakuro	551	9	6	49.0 k	1.3	1.8 k	861	844
mdd	44	5	7	39.5 k	50.5	1.8 k	1.7 k	1.4 k
nonograms	185	2	25	80.8 k	0.1	369	250	252
pentominoes	5	13	129	7×10^{424}	5×10^{-4}	110 k	72 k	72 k
pigeonsplus	37	8	6	1.8 M	48.3	91	46	48
proteindesign	5	328	3	9.7 k	0.6	668 k	8.2 k	6.4 k
renault	102	13	4	3.5 k	15.0	4.3 k	1.6 k	1.1 k
tsp	75	168	3	14.6 k	0.3	59 k	15 k	15 k
weirand	100	12	5	56.2 k	22.6	49 k	32 k	22 k

Table 1: Average characteristics of evaluation instances (rounded). Only the largest MDD of each instance (in terms of $|E(M)|$) is accounted.

Increasing q obviously benefits to all BDD/MDD-based algorithms. The minimalist data structures of *BDDC2*/*MDDC* result in very low memory usage in this case.

7.3 Structured problems

Concrete 3 can read problems formulated using either its own API (which was used for random problems), XCSP 2.1 [25] or MiniZinc 2/FlatZinc [30] formats. We selected many families of instances from various sources. Table 1 lists the different classes of instances, number of problems in each class and average characteristics. To compute the characteristics, we only took into account the largest relation (in terms of MDD edges) of each instance. Problems *pentominoes* and *proteindesign* come from the MiniZinc Challenge 2013 [36]. *carseq* comes from the CSPLib [17] and was reformulated in MiniZinc format for this benchmark. All other instances come from C. Lecoutre's XCSP library [21]. *nonograms* include problems from both the MiniZinc Challenge 2012, 2013 and the XCSP library.

The problems from the XCSP library, as well as *proteindesign*, are already modeled using positive extensional constraints. We selected all instances we could find that featured non-random, non-binary extensional constraints. *bdd*, *mdd* and *wei-rand* classes are randomly generated but feature some structure that can be exploited by MDD/BDD ("quasi-random" instances). Note that *aim*, *kakuro*, *renault* and most *nonograms* instances from the XCSP library are very easy (solved in less than one second whichever propagation algorithm is used), and measurements may be imprecise.

carseq problems are modeled using the *sliding-sum* global constraint. *nonograms* in MiniZinc and *pentominoes* are modeled using the *regular* global constraint. These constraints can be converted to MDD efficiently [9, 32]. Such constructs allow to express MDD of reasonable size that represent extremely large relations. It is also the case for the *pigeonplus* instances, although these instances can barely be represented using tables.

Instances modeled with MiniZinc were solved using the search strategy specified in the model. Other problems use $^{dom}/_{wdeg}$ variable ordering heuristic with random tie-breaking, coupled with *random* value ordering heuristic and geometric restarts. We measured the total number of revisions per second (including revisions of non-MDD/BDD/table-based constraints for heterogeneous models) and memory used by Concrete. We defined a time limit of 1,200 s and a memory limit of 4 GiB. When a timeout occurs, we include the revisions per second and memory used so far in the results. When a memory out occurs, we cap the memory required for the instance to > 4 GiB and counted 0 rps. Each instance was

	BDDF		*BDDC2*		*MDDC*		*STR2*	
	krps	mem	krps	mem	krps	mem	krps	mem
aim	63	13	**97**	13	66	**12**	78	15
bdd	35	1,333	28	554	20	**481**	**68**	1,100
carseq	**434**	58	**433**	**25**	318	**26**	1	> 4,096
crossword	**80**	76	**88**	**59**	33	63	**88**	**54**
kakuro	5	13	**7**	15	6	**8**	5	25
mdd	**191**	20	**177**	**14**	93	**15**	41	124
nonograms	20	14	**27**	13	15	**11**	16	> 85
pentominoes	**11**	219	**11**	109	9	**82**	0	> 4,096
pigeonsplus	**463**	11	**500**	11	390	**9**	110	> 727
proteindesign	**156**	**340**	139	**336**	72	**337**	91	**339**
renault	31	11	**43**	**10**	29	**10**	27	18
tsp	**524**	42	383	30	284	**17**	**558**	37
wei-rand	**13**	751	7	**323**	4	**312**	4	437

Table 2: Average experimental results: "krps" is kilo revisions per second (higher is better) and "mem" is memory used in MiB (lower is better). ">" means that at least one instance ran out of memory and was counted as 0 rps and capped to 4 GiB used memory. Results within 10 % of the best result are highlighted.

is faster than ↗	*BDDF*	*BDDC2*	*MDDC*	*STR2*	*BDDF*	*BDDC2*	*MDDC*	*STR2*
	all 1,973 instances				447 hard instances			
BDDF	–	31	**56**	**59**	–	**49**	**80**	**63**
BDDC2	**54**	–	**80**	**74**	33	–	**80**	**57**
MDDC	32	8	–	**49**	10	10	–	42
STR2	31	16	34	–	33	32	**51**	–

Table 3: Pairwise comparison of algorithms on structured instances

run twice with different random seeds for the branching heuristic to reduce measurement imprecision.

Table 2 shows the results. For each class of instances, we highlighted the best result (highest rps and lowest mem), as well as results which are within 10 % of the best one.

We also made a pairwise comparison of algorithms, similarily to what is done to evaluate solvers in the MiniZinc Challenge [36]: for each instance and each pair of algorithms a_1 and a_2, we scored $^{100}/_{|i|\,|c|}$ "points" for a_1 against a_2 if a_2 errored or a_1 was at least 10 % faster (in terms of rps) than a_2. $|c|$ is the number of problem classes, $|i|$ is the number of instances in the current problem class. The latter factor avoids bias towards classes for which we have a very large number of available instances. Results of this comparison is shown on Table 3. For each pair, the "winner algorithm", i. e., the best on a majority of instances, is highlighted. Columns 2 to 5 include results for our full database of 1,973 instances. Columns 6 to 9 are restricted to the 447 harder instances for which at least one algorithm errored or took more than 200 seconds to solve. Note that this rules out all instances from classes *aim*, *kakuro* and *renault* which are all easy.

For example, *BDDC2* is at least 10 % faster than *MDDC* on 80 % of instances, whereas *MDDC* is at least 10 % faster than *BDDC2* on 12 % of instances. Results are within the margin of 10 % for the 8 % remaining instances. The result is similar when considering only the hard instances (80/10 %). Concerning *BDDF*, we see that although it is slower than *BDDC2* on a short majority of 54 % of instances, the result is reversed to 49/33 % when considering harder instances only.

From these tables, it seems that *BDDC2* is a good compromise for problems that are not too hard to solve. It is faster than *MDDC* on a vast majority of instances, both easy and hard. This can also be spotted on Table 2: rps for *BDDC2* are always greater than

for *MDDC*. *BDDC2* is also often faster than *STR2*, especially on easy instances: 74 % *vs* 16 %, but only 57 *vs* 32 % on harder instances.

STR2 is notably faster only when arity is relatively high and λ comparatively low, i. e., constraints are tight. In this case, the lower incrementality of *BDDC2* w.r.t. that of *STR2* is not compensated by the compression. It is notably the case for *crossword* or *bdd* instances. On the other hand, highly structured relations are very badly represented by flat tables. In this case, *STR2* is extremely slow (e. g., *mdd*, *nonograms*, or *renault*) and most often will not even fit into memory (e. g., for *carseq*, *pentominoes* and *pigeonsplus* classes).

It is harder to tell definitely whether *BDDF* is a worthy improvement over *BDDC2*. At a first glance *BDDC2* is faster (and less memory-consuming) than *BDDF* on a majority of instances, despite the better incrementality of the latter algorithm and encouraging results on randomly generated problems. However, when considering only hard instances, *BDDF* has the advantage. When we investigate specific problem classes, we see that *BDDF* is notably faster than *BDDC2* on *mdd*, *proteindesign*, *tsp* and *wei-rand* problems, which seems to correllate with high domain sizes.

These results are theoretically sound. Although *BDDF* clearly has a better theoretical behaviour than *BDDC2*, there is an non-negligible overhead in managing the extra data. On one hand, the improved incrementality of *BDDF* triggers when values towards the beginning of the domains are filtered from the BDDs. This is more likely to happen and have significant impact when the domains are large. On the other hand, the improved incrementality of *BDDF* is more likely to be witnessed on instances for which more backtracks are required to solve.

8 Conclusion and perspectives

In this article, we presented an alternative representation of MDDs used to represent positive relations for extensional constraints, which uses BDDs instead of MDDs. We shown that the proposed BDD data structure can compress tables better than standard MDDs. To the best of our knowledge, it is the first time in the literature that a conversion from MDD to BDD with potentially lower space behavior is presented. These data structures can be used to compress the *tables* used to represent extensional constraints, and even implement some global constraints such as *sliding-sum* or *regular* easily.

We adapted and presented two algorithms, *BDDC2* and *BDDF* that exploit the ideas from *MDDC* and *STR2* to improve the incrementality properties of *MDDC* while keeping the compression properties of MDDs. The two algorithms can be implemented using coarse-grained propagation queues. We shown experimentally that *BDDC2* is faster than *MDDC* on a vast majority of tested problem instances, making it competitive with *STR2* even when the compression is low. Memory can be worse than *MDDC* by a low constant factor when the additional compression is not significant. *BDDF* has better incrementality properties than *BDDC2*, at the cost of a reasonable but sometimes significant memory overhead. Managing this extra memory can make *BDDF* slower than *BDDC2*, especially when domain size is low.

In summary, experimental as well as theoretical results show that for relations that are either large (loose constraints) or structured (efficiently compressed/represented as a MDD/BDD), one should use either *BDDC2* or *BDDF* instead of *STR2*. The "turning point" is highly dependent on the implementation. *BDDC2* is to be preferred over *BDDF* when memory is an issue, domains are small and problems are not too hard.

On one hand, following ideas from Perez and Régin [33], *BDDF* can probably be improved by exploiting fine-grained propagation queues and additional data structures to link domain values to the corresponding vertices in the MDD/BDD. Other perspectives include improving the compression by reordering variables and values, reducing the MDD/BDDs dynamically during search, or even tailoring search strategies to lead the

search towards areas of the search space where the MDD/BDD are smaller. Moreover, a more aggressive usage of *hash-consing* may theoretically reduce the size of data structures, by e. g., sharing data between several similar BDDs in the same problem instances. For *BDDF*, it can bring on-the-fly reduction with no worst-case complexity overhead. Despite disencouraging preliminary results, this possibility may be worth further investigation.

On the other hand, let us recall that *BDDF* is a competitive algorithm that makes use only of purely persistent/functional data structures.[13] Such data structures are crucial in the implementation of non-chronological backtracking and parallel processing, because they do not require any synchronization. This may be the major perspective of this work.

Acknowledgments. This research was partially financed by the French Ministry of National Education, Research and Technology, The Nord/Pas-de-Calais Region, the French National Center of Scientific Research (CNRS) and the International Campus on Safety and Intermodality in Transportation.

References

[1] J. Allen. *Anatomy of LISP*. New York, NY, USA: McGraw-Hill, Inc., 1978. ISBN: 0-07-001115-X.
[2] P. Bagwell. *Ideal Hash Trees*. Tech. rep. EPFL, 2001.
[3] J. Bentley and R. W. Floyd. "Programming pearls: a sample of brilliance". In: *Commun. ACM* 30.9 (Sept. 1987), pp. 754–757.
[4] D. Bergman, A. A. Ciré, W.-J. van Hoeve, and J. Hooker. "MDD Propagation for sequence Constraints". In: *Decision Diagrams for Optimization*. Springer, 2016. Chap. 10, pp. 183–204.
[5] C. Bessière and J.-C. Régin. "MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems". In: *Proc. of the 12th Itl. Conf. on Principle and Practice of Constraint Programming (CP)*. 1996, pp. 61–75.
[6] B. Bollig and I. Wegener. "Improving the variable ordering of OBDDs is NP-complete". In: *IEEE Transactions on Computers* 45.9 (1996), pp. 993–1002.
[7] P. Briggs and L. Torczon. "An Efficient Representation for Sparse Sets". In: *ACM Letters on Programming Languages and Systems* 2.1–4 (1993), pp. 59–69.
[8] R. E. Bryant. "Graph-based algorithms for boolean function manipulation". In: *IEEE Transactions on Computers* 100.8 (1986), pp. 677–691.
[9] K.C.K. Cheng and R.H.C. Yap. "An MDD-based GAC algorithm for positive and negative table constraints and some global constraints". In: *Constraints* 15.2 (2010), pp. 265–304.
[10] K.C.K. Cheng and R.H.C. Yap. "Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints". In: *Proc. ECAI-06*. IOS Press, 2006, pp. 78–82.
[11] A. A. Ciré and J. N. Hooker. "The Separation Problem for Binary Decision Diagrams." In: *Proc. ISAIM*. 2014.
[12] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Communications of the ACM* 13.6 (1970), pp. 377–387.
[13] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J. Régin, and P. Schaus. "Compact-Table: Efficiently Filtering Table Constraints with Reversible Sparse Bit-Sets". In: *Proc. CP'2016*. Vol. 9892. LNCS. Springer, Sept. 2016, pp. 207–223.
[14] J.R. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. "Making Data Structures Persistent". In: *J. of Computer and System Science* 38 (1989), pp. 86–124.
[15] E. Fredkin. "Trie Memory". In: *Comm. ACM* 3.9 (1960), pp. 490–499.

[13] Note that to achieve pure immutability, the *timestamp* technique used in this paper should be replaced by some sort of lookup table, which is likely to be much slower.

[16] G. Gange, P. J. Stuckey, and R. Szymanek. “MDD propagators with explanation”. In: *Constraints* 16.4 (2011), pp. 407–429.

[17] I. P. Gent and T. Walsh. *CSPLib: a benchmark library for constraints*. Tech. rep. APES-09-1999, 1999. URL: `http://www.csplib.org`.

[18] I.P. Gent, C Jefferson, I. Miguel, and P. Nightingale. “Data structures for generalised arc consistency for extensional constraints”. In: *Proc. AAAI'2007*. 2007, pp. 191–197.

[19] T. Hadzic, E. R. Hansen, and B. O'Sullivan. “On Automata, MDDs and BDDs in Constraint Satisfaction”. In: *Proc. ECAI Workshop on Inference Methods based on Graphical Structure of Knowledge (WIGSK)*. 2008.

[20] P. van Hentenryck, Y. Deville, and CM. Teng. “A Generic AC Algorithm and its Specializations”. In: *Artificial Intelligence* 57 (1992), pp. 291–321.

[21] C. Lecoutre. *Benchmarks 2.0 - XML representation of CSP instances*. `http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks`. 2006.

[22] C. Lecoutre. “STR2: Optimized Simple Tabular Reduction for Table Constraints”. In: *Constraints* 16.4 (2011), pp. 341–371.

[23] C. Lecoutre and F. Hemery. “A Study of Residual Supports in Arc Consistency”. In: *Proceedings of IJCAI'2007*. 2007, pp. 125–130.

[24] C. Lecoutre, C. Likitvivatanavong, and R.H.C. Yap. “A path-optimal GAC algorithm for table constraints”. In: *Proc. ECAI'2012*. 2012, pp. 510–515.

[25] C. Lecoutre and O. Roussel. “XML Representation of Constraint Networks, Version 2.1”. In: *The Computing Research Repository* arXiv: 0902.2362v1 (2008).

[26] A.K. Mackworth. “Consistency in Networks of Relations”. In: *Artificial Intelligence* 8.1 (1977), pp. 99–118.

[27] J.-B. Mairy, P. van Hentenryck, and Y. Deville. “Optimal and efficient filtering algorithms for table constraints”. In: *Constraints* 19.1 (2014), pp. 77–120.

[28] D. Michie. “Memo Functions and Machine Learning”. In: *Nature* 218 (1968), pp. 19–22.

[29] U. Montanari. “Network of constraints : Fundamental properties and applications to picture processing”. In: *Information Science* 7 (1974), pp. 95–132.

[30] N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. “Minizinc: Towards a standard CP modelling language”. In: *Proc. CP'2007*. Ed. by C. Bessière. 2007, pp. 529–543.

[31] M. Odersky et al. *The Scala Programming Language*. `http://www.scala-lang.org/`. 2001.

[32] G. Perez and J.-C. Régin. “Efficient Operations on MDDs for Building Constraint Programming Models”. In: *Proc. 24th IJCAI*. 2015, pp. 374–380.

[33] G. Perez and J.-C. Régin. “Improving GAC-4 for Table and MDD Constraints”. In: *Proc. 20th Conference on Principles and Practice of CP*. Ed. by B. O'Sullivan. LNCS 8656. Springer, 2014, pp. 606–621.

[34] G. Pesant. “A Regular Language Membership Constraint for Finite Sequences of Variables”. In: *Proc. CP'2004*. Ed. by M. Wallace. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 482–495.

[35] A. Srinivasan, T. Ham, S. Malik, and R. K. Brayton. “Algorithms for discrete function manipulation”. In: *1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*. Nov. 1990, pp. 92–95. DOI: `10.1109/ICCAD.1990.129849`.

[36] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, and J. Fischer. “The MiniZinc Challenge 2008–2013”. In: *AI Magazine* 35.2 (2014), pp. 55–60.

[37] J.R. Ullmann. “Partition search for non-binary constraint satisfaction”. In: *Information Science* 177 (2007), pp. 3639–3678.

[38] J. Vion. *Concrete: a CSP Solving API for the JVM*. `http://github.com/concrete-cp`. 2006.

[39] J. Vion. “Consistance d’arc par MDD-Réduction”. French. In: *Actes des 9e Journées Francophones de Programmation par Contraintes (JFPC)*. Ed. by C. Truchet. 2013, pp. 323–332.

[40] J. Vion and S. Piechowiak. “Maintenir des MDD persistants pour établir la consistance d’arc”. French. In: *Revue d’Intelligence Artificielle* 28.5 (2014), pp. 547–569.

[41] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. “A simple model to generate hard satisfiable instances”. In: *Artificial Intelligence* 171 (2007), pp. 514–534.