Improving GAC-4 for Table and MDD Constraints

Guillaume Perez, Jean-Charles Régin

Improving GAC-4 for Table and MDD Constraints

Guillaume Perez and Jean-Charles Régin

Université Nice-Sophia Antipolis, I3S UMR 6070, CNRS, France
guillaume.perez06@gmail.com, jcregin@gmail.com

Abstract. We introduce GAC-4R, MDD-4, and MDD-4R three new algorithms for maintaining arc consistency for table and MDD constraints. GAC-4R improves the well-known GAC-4 algorithm by managing the internal data structures in a different way. Instead of maintaining the internal data structures only by studying the consequences of deletions, we propose to reset the data structures by recomputing them from scratch whenever it saves time. This idea avoids the major drawback of the GAC-4 algorithm, i.e., its cost at a shallow search-tree depth. We also show that this idea can be exploited in MDD constraints. Experiments show that GAC-4R is competitive with the best arc-consistency algorithms for table constraints, and that MDD-4R clearly outperforms all classical algorithms for table or MDD constraints.

1 Introduction

We consider table and Multi-valued Decision Diagram (MDD) constraints, which list the allowed combinations of values for the variables in the scopes of the constraints. Those constraints are useful for modeling and solving many real-world problems. They can be specified either directly, by input from the user, or indirectly by synthesizing other constraints or subproblems [17,11].

Table constraints are fundamental and implemented in any CP solver. This is the case for the or-tools[1] solver, which won the 2013 MiniZinc Challenge.[2] The or-tools solver does not implement many global constraints, but has an implementation of GAC-4R, which is our efficient algorithm for enforcing arc consistency on table constraints. In this paper, we introduce GAC-4R, and we adapt it to enforce arc consistency on MDD constraints.

Consider an extensional constraint C. Arc-consistency algorithms for C operate as follows: for each value a in the domain of a variable x, they search for a combination of values in the current domains of the other variables in the scope of C that contains (x, a) and satisfies C. A tuple of C is a combination of values in the domain of the variables in the scope of C. We say the the tuple is *allowed*, or a support, when it appears in the constraint's definition. We say that the tuple is *valid* if and only if its values appear in the *current* domains of the respective variables. Note that a valid tuple is not necessarily allowed. Arc-consistency algorithms can be distinguished depending on how they manage allowed and valid tuples [16]. While the allowed tuples do not

[1] http://code.google.com/p/or-tools/
[2] http://www.minizinc.org/challenge2013/call_for_problems.html

change during search because they are listed in the constraint definition, their validity is determined by the current domains.

While some arc-consistency algorithms operate on allowed tuples to check their validity, others first consider the current domains and look for a combination satisfying the constraint.

Existing algorithms mainly differ by how they operate when a value is deleted. Some algorithms are lazy (e.g., GAC-Schema [3], STR-2 [14], or STR-3 [15]). They try to reduce the operations executed at each modification (i.e., deletion of value of a domain) at the cost of increasing the complexity of the implementation. Others, such as GAC-4 [19], operate more systematically, thus keeping the implementation simple.

GAC-4 associates, to each variable-value pair (x, a), the list $S(x, a)$ of valid tuples involving (x, a) that satisfy C. When a value b is deleted from the domain of a variable y, the tuples associated with (y, b) are no longer valid and must be removed. Consequently, for each tuple $t \in S(y, b)$ and for each variable-value pair (z, c) in t, we remove t from $S(z, c)$. If $S(z, c)$ becomes empty, then no valid tuple involving (z, c) and satisfying C exists. Thus, we can safely remove c from $D(z)$.

GAC-4 is efficient when there are only few tuples for each value, which typically occurs at deeper levels of the search tree. However, at shallower levels its performance is qualitatively different in that maintaining the internal data structures is costly. In this paper, we give a solution to this issue.

Indeed, we show that the performance of GAC-4 can be improved by rebuilding, from scratch, the data structures of GAC-4 when the modifications have reached a given threshold. We illustrate such a situation with an example. Consider a table constraint with k tuples and involving a variable x having 10 values in its domain (the arity is not important here). Assume that the tuples are homogeneously distributed among the values of x. In other words, every value of x appears in about $\frac{k}{10}$ tuples. Now, assume that a is assigned to x. Thus, only about $\frac{k}{10}$ tuples remain valid. GAC-4 will consider and propagate deletions of $\frac{9k}{10}$ tuples although only about $\frac{k}{10}$ tuples remain. Thus, it is more effective to reset the constraint with the elements of $S(x, a)$, in other words, to rebuild the constraint from scratch. In this situation, we restart from a tuple set of only $\frac{k}{10}$ tuples and save a factor of 9. We can determine exactly when it is worthwhile to apply such an operation, which is when the sum of the sizes of S lists of the deleted values of x is larger than the sum of the sizes of S lists of the remaining values in the current domain of x.

In this paper, we introduce GAC-4R, which exploits that idea. The challenge is to maintain this mechanism throughout the search, because we need to undo this operation upon backtracking. To this end, we propose to represent the list of supported tuples for each variable-value pair using sparse sets.

Another way for improving the performance of arc-consistency algorithms for table constraints is to reduce the size of the representation of the tuples because the complexity depends on it. Several algorithms for compressing the allowed tuples of a constraint have been proposed, and arc consistency algorithms adapted for dealing with them [12,9,21]. Multi-valued Decision Diagrams (MDDs) are one of the most advanced and powerful representations. Cheng and Yap provide `mddc`, an efficient algorithm for enforcing arc consistency on MDDs based on GAC-3 [8,7]. In this paper, we define

MDD-4, which adapts GAC-4 to MDD constraints, and introduce MDD-4R, which implements our improvement in MDD-4.

The paper is organized as follows. First, we recall background information. Then, we describe GAC-4R, our new version of GAC-4 based on resetting data structures. Next, we introduce MDD-4 and MDD-4R, which adapts the idea of reset to MDDs. After reviewing main existing GAC algorithms we discuss experiments that empirically establish the advantages of our algorithms in practice. Finally, we conclude this paper.

2 Preliminaries

2.1 Definitions

Constraint network. A finite *constraint network* $\mathcal{N} = (X, \mathcal{D}, \mathcal{C})$ is defined as a set of n *variables* $X = \{x_1, \ldots, x_n\}$, a set of *domains* $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible *values* for variable x_i, and a set $\mathcal{C}$ of *constraints* between variables. A value a for a variable x is often denoted by (x, a).

Constraint. A constraint C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ that specifies the *allowed* combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An element of $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ is called a *tuple on* $X(C)$ and $t[x]$ is the value of the tuple t assigned to x. $|X(C)|$ is the *arity* of C. We will denote by d the size of the largest initial domain and by r the arity.

Arc consistency. Let C be a constraint. A tuple t on $X(C)$ is *valid* iff $\forall x \in X(C)$, $t[x] \in D(x)$; and t is a *support* for (x, a) iff $t[x] = a$ and $t \in T(C)$. A value $a \in D(x)$ is *consistent with* C iff $x \notin X(C)$ or there exists a valid support for (x, a). C is *arc consistent* iff $\forall x \in X(C), D(x) \neq \emptyset$ and $\forall a \in D(x)$, a is consistent with C.

Table and MDD constraints. Those constraints are said to be defined in extension. A table constraint is a constraint whose tuples satisfying the constraint are explicitly given in extension. An MDD constraint is a constraint which is defined thanks to a multi-valued decision diagram which is an efficient way to know whether or not a combination of values of the variables involved in the constraints satisfies the constraint.

2.2 GAC-4

The data structures of GAC-4 are quite simple. GAC-4 associates with each value a of each variable x the list $S(x, a)$ of the tuples of $T(C)$ containing (x, a). It maintains the following invariant:

$\forall x \in X(C), \forall a \in D(x)$: *$S(x, a)$ contains the valid tuples* $t \in T(C)$ *with* $t[x] = a$.

Thus, if a list $S(x, a)$ becomes empty then GAC-4 removes the value a from the domain of x. The initialization is done as follows: for each tuple t and for each value (x, a) belonging to t we add t to $S(x, a)$. Each time there is a deletion of a value of a variable involved in C, this deletion is added to a deletion set and the constraint is pushed in order to be revised later. Function REVISEGAC-4 (See Algorithm 1) is called in order to propagate the consequences of these deletions. It maintains the S lists.

Algorithm 1: REVISEGAC-4

```
REVISEGAC-4(C: constraint; deletionSet: list): Boolean
for each (x, a) ∈ deletionSet do
    for each t ∈ S(x, a) do
        for each (z, c) ∈ t do remove t from S(z, c)
        if S(z, c) = ∅ then remove c from D(z) ; add (z, c) to deletionSet
        if D(z) = ∅ then return False;
return True
```

2.3 Sparse Set

Sparse set is an efficient data structure for manipulating sets with a fixed size universe U [5]. It has been successfully used in CP for representing sets or lists [8,7,14,15]. We will use it in GAC-4R and MDD-4R, so we give details of it.

For convenience, the elements in U are mapped to integers 0 through $|U| - 1$. The representation has three components: two vectors (named `dense` and `sparse`), each $|U|$ elements long and a scalar (named `members`) that records the number of members in the set. The values in the array `dense` from 0 to `members` - 1 corresponds to the elements in the set. The array `sparse` contains indices of the array `dense`. If a number k is a member of the set, it must satisfy two conditions $0 \leq \texttt{sparse}[k] < \texttt{members}$ and $\texttt{dense}[\texttt{sparse}[k]] = k$. It means that $\texttt{sparse}[k]$ is the index i in the array `dense` of the value k, that is, we have $\texttt{dense}[i] = k$. Here is a sparse set:

sparse	5	2	-	0	-	1	-	3	4	-
dense	3	5	1	7	8	0				
members	6									

The membership, addition and deletion functions are defined in Algorithm 2. Function DELETE has been modified from its original definition in [5] in order to be able to restore easily the sparse set after some deletions. Consider the sparse set previously defined. Suppose that member 7 is deleted. Before the deletion the scalar `members` is equal to 6 and after the deletion we have the new sparse set:

sparse	3	2	-	0	-	1	-	5	4	-
dense	3	5	1	0	8	7				
members	5									

When 7 has been deleted, the members 7 and 0 (i.e. the last value of the set) have been exchanged. Precisely, we swap dense[sparse[7]] and dense[sparse[0]] and we swap sparse[7] and sparse[0]. Thanks to these swaps, we can easily restore the sparse set simply by setting the members value to 6. The sparse set contains the same elements but not in the same order.

2.4 Multi-valued Decision Diagram

This presentation is inspired from [22]. Multi-valued decision diagram (MDD) is a method for representing discrete functions. It is a multiple-valued extension of BDD

Algorithm 2: Functions for manipulating a sparse set. k is an element. S is a sparse set with two arrays $S.\texttt{dense}$ and $S.\texttt{sparse}$ and scalar $S.\texttt{members}$.

MEMBER(k, S): return $S.\texttt{sparse}[k] < S.\texttt{members}$ and $S.\texttt{dense}[S.\texttt{sparse}[k]] = k$

ADD(k, S) // assume k is not a member

- $S.\texttt{sparse}[k] \leftarrow S.\texttt{members}$
- $S.\texttt{dense}[S.\texttt{members}] \leftarrow k$
- $S.\texttt{members} \leftarrow S.\texttt{members} + 1$

DELETE(k, S) // assume k is a member

- $ik \leftarrow S.\texttt{sparse}[k]$; $ie \leftarrow S.\texttt{members} - 1$; $e \leftarrow S.\texttt{dense}[ie]$
- $S.\texttt{sparse}[e] \leftarrow ik$
- $S.\texttt{dense}[ik] \leftarrow e$
- $S.\texttt{sparse}[k] \leftarrow ie$
- $S.\texttt{dense}[ie] \leftarrow k$
- $S.\texttt{members} \leftarrow S.\texttt{members} - 1$

[6]. An MDD is a directed acyclic graph (DAG) used to represent some multi-valued function $f : \{0...d-1\}^r \rightarrow \{0...d-1\}$, based on a given integer d. Given the r input variables, the DAG representation is designed to contain r layers of nodes, such that each variable is represented at a specific layer of the graph. Each node on a given layer has d outgoing arcs to nodes in the next layer of the graph. Each arc is labeled by its corresponding integer. The final layer is represented by the terminal nodes having values in the range $\{0, ...d-1\}$.

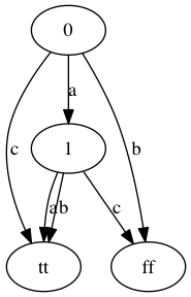

Fig. 1. an MDD representing a function defined on variables x_1 and x_2. Layer 1 (resp. 2) corresponds to x_1 (resp. x_2). Terminal node tt is true while ff is false. The function is true for the tuples (a, a), (a, b), (c, a), (c, b), (c, c)

The function encoded by an MDD may be evaluated on a given set of r input variables by traversing the graph, starting from the root (i.e. the highest-layer variable in the graph), and choosing the outgoing arc at each node corresponding to the input value of the variable represented at the current layer of the graph. This traversal continues until a terminal node is reached. The resulting value of the function is indicated by the value of the terminal node reached along this evaluation path. MDDs have been widely used in CP because there are powerful for modeling problems or for representing some functions or domain store [1,10,11,2]. In CP there are usually two terminal nodes tt, which is true, and ff which is false. Figure 1 gives an example of MDD. We have the path $(0, 1, tt)$ so $f(a, a)$ and $f(a, b)$ are true because tt is true.

An MDD is *ordered* if each variable is encountered at most once on each path from the root to a terminal. An MDD is *fully-reduced* if it does not contain any nodes with

all k outgoing arcs pointing to the same node and does not contain duplicate nodes for a given layer in the DAG. Fully-reduced and ordered MDDs are mainly used.

Arc consistency and MDD. Cheng and Yap provide `mddc`, an algorithm maintaining arc consistency for an MDD constraint based on GAC-3 [8,7]. In an MDD constraint, the MDD models the set of tuples satisfying the constraint. Each variable of the MDD corresponds to a variable of the constraint. An arc associated with an MDD variable corresponds to a value of the corresponding variable of the constraint. It uses a depth first search for traversing the MDD from the root node. Only arcs corresponding to values belonging to the domain of the variables are used. Terminal nodes are either positive or negative. A node x is marked positive if the subgraph traversed from x reaches a positive node. If no terminal node can be reached from x or if only negative nodes can be reached from x then x is marked negative. The algorithm saves the values corresponding to the arcs having a positive terminating node (this means that the value belongs to a valid tuple defined by the path). When the depth first search ends, the values that have not been saved may be safely removed because there is no longer a path from the root to a positive terminal node, which involves an arc corresponding to this value.

In `mddc`, it is important to note that any arc is traversed at most once, because a depth first search is used and because the MDD is not changed during the search by the algorithm. This is only the way the MDD is traversed that depends on the current domains. Note also that it is not straightforward to find an arc corresponding to a value belonging to the current domain and this task is more difficult when the domain size is reduced.

3 GAC-4R

GAC-4 is a simple and easy algorithm to implement. Its worst case complexity is optimal. However it is mainly focused on the study of the consequences of the deletions of values but it could be worthwhile to recompute some data structures instead of maintaining them incrementally. A simple example is the computation of an intersection. Suppose that you want to maintain the intersection C of two sets A and B. You computed this intersection when A and B had 100 values and you determined that C has 50 values. Suppose that a value v of A is deleted. Then, your algorithm will recompute the set C by checking whether C contains v or not, and remove v from C if $v \in C$. Now, suppose that we remove 98 values from A, and we have $A = \{a, b\}$. Then it is faster to check whether a and b belong to B and build C consequently instead of considering the deleted values. Only two operations have to be performed to define the new set C. In this case, we will say that we reset C.

This idea had been applied to define which algorithm should be preferred between AC-2001 and AC-6 [4] or to design an adaptive algorithm [20]. Combining GAC-4 with this idea will save a lot of computations for a shallow depth of the tree search. Such a combination requires to answer two questions:

1. How can we know whether a reset is better or not?
2. How can we perform this reset and the restoration of the previous set efficiently?

The first question answer is simple. Consider a variable x and $\Delta(x)$ the set of values of $D(x)$ that have been deleted and not yet considered by GAC-4 algorithm (i.e.

they belong to deletionSet). The number of tuples that are no longer valid, denoted by $\#T\Delta(x)$, is given by the sum of the size of the S lists of the values in $\Delta(x)$, and the number of remaining tuples is the difference between the total number of tuples and $\#T\Delta(x)$ because a tuple contains only one value per variable. So we have:

Property 1 *Let x be a variable, $\Delta(x)$ be the values of x that have been deleted and that must be propagated, $\#T\Delta(x) = \sum_{a \in \Delta(x)} |S(x,a)|$ be the number of tuples that are no longer valid and T be the current number of tuples.*
If $\#T\Delta(x) > \frac{T}{2}$ then a reset operation will consider less tuples than the application of Function REVISEGAC-4

This property is useful only if we can answer the second question. We show that we can use sparse sets for efficiently computing a reset operation in such a way that the restoration is easy.

The question can be reformulated as follows. Consider S a set with two lists of elements R and Q. The lists are disjoint and their union contains exactly all the elements of S. The sets R and Q are not explicitly given, that is, we do not have a set representing them but we can traverse them (in term of programming language, they are given by an iterator) and their size is known. We want to modify S by removing the set of elements $R \subseteq S$ in order to obtain a set containing only the elements of Q. However, instead of performing $|R|$ operations for this task, we want to have a number of operations bounded by $\min(|R|, |Q|)$. In addition, we have to be able to restore the set S after performing the modifications with a similar complexity (or less).

Algorithm 3: Function re-add of a sparse set S. k is an element.

```
RE-ADD(k, S) // We assume that S.dense[S.sparse[k]] = k
ik ← S.sparse[k]; e ← S.dense[S.members]
S.sparse[k] ← S.members
S.sparse[e] ← ik
S.dense[S.members] ← k
S.dense[ik] ← e
S.members← S.members +1
```

Thanks to sparse sets we can give a nice answer to this question. Let S be represented by a sparse set. If $|R| \leq |Q|$ then we delete the elements of R from S as it is explained in Preliminaries section. The restoration of S consists of modifying the scalar `members` of S. Assume that $|Q| < |R|$. S is recomputed as follows. First, we set S.`members` to 0. Then, we traverse Q and for each element $a \in Q$ we add a to S by calling Function RE-ADD (See Algorithm 3) which is a modified version of Function ADD of the sparse set. It exploits the fact that the value which is added was previously in the set. Thus, it proceeds to a swap in a way similar as the one used by Function DELETE in order to be able to restore the set in the future. More precisely, when an element i is re-added to the sparse set, we swap i and the value j at the index defined by `members`. That is, we exchange the value of i and the value of j in the `sparse` array and we exchange i and j in the `dense` array. For instance, consider the left sparse set:

sparse	3	2	-	0	-	1	-	5	4	-
dense	3	5	1	0	8	7				
members	2									

sparse	3	4	-	0	-	1	-	5	2	-
dense	3	5	8	0	1	7				
members	3									

The set contains the values 3 and 5. If we re-add the value 8 then we will exchange the value of `dense[members]`, i.e. 1, with 8. So we will have `dense`$[2] = 8$; `dense`$[4] = 1$; `sparse`$[8] = 2$; `sparse`$[1] = 4$. We obtain the right sparse set.

The advantage of this method is that the restoration of the scalar `members` is enough for restoring the sparse set. Function RE-ADD has a complexity in $O(1)$ per call. Thus, we can re-add $|Q|$ elements in $O(|Q|)$.

Algorithm 4: GAC-4R. T is the current number of tuples

```
REVISEGAC-4R(C: constraint; deletionSet: list, T: number of tuples): Boolean
    for each x ∈ X(C) do #TΔ(x) ← 0
    for each (x, a) ∈ deletionSet do #TΔ(x) ← #TΔ(x) + |S(x, a)|
    #TΔmax ← max_{x∈X(C)}(#TΔ(x))
    if #TΔmax > T/2 then
        // we reset the data structures
        pick a variable x with #TΔ(x) = #TΔmax
        Tset ← ∅; T ← 0
        for each a ∈ D(x) do add S(x, a) in Tset
        for each y ∈ X(C) do
            for each b ∈ D(y) do S(y, b).members ← 0
        // we re-add valid tuples into the S lists
        for each t ∈ Tset do
            if t is valid then
                for each i = 1...n do RE-ADD(t, S(x_i, t[i]))
                T ← T + 1
        // we remove values having an empty S list.
        for each y ∈ X(C) do
            for each b ∈ D(y) do
                if S(y, b) = ∅ then remove b from D(y)
                if D(y) = ∅ then return False;
    else
        // classical GAC-4 deletion process
        for each (x, a) ∈ deletionSet do
            for each t ∈ S(x, a) do
                for each i = 1...n do
                    DELETE(t[i],S(x_i, t[i]))
                    T ← T − 1
                    if S(x_i, t[i]) = ∅ then remove t[i] from D(x_i)
                    if D(x_i) = ∅ then return False;
    return True
```

A possible implementation of GAC-4R is given by Algorithm 4. Each list S is represented by a sparse set with a fixed size universe equal to $|T(C)|$. For convenience, we will consider that t is a tuple and also the index of the tuple in the table of tuples.

The complexity of GAC-4R remains the same as the complexity of GAC-4, because the deletion of a tuple or the re-addition of a tuple have the same complexity which corresponds to the arity of the constraint. In addition traversing the valid tuples costs at least the cost of traversing all the domains of the variables involved in the constraint and since we do this only when there are less valid tuples than non valid tuples, the traversal of all the domains does not impact the complexity.

4 MDD-4R

We propose to adapt the principles of GAC-4R to be able to deal with an MDD instead of a table. First, we will slightly modify the MDD from which the constraint is defined. Then we design the MDD-4 algorithm. Unlike `mddc`, MDD-4 modifies and maintains the MDD during the search for a solution. Next, we will study the reset principle for MDD-4 and explain when the reset should be done.

4.1 MDD reformulation

ff removal. The node ff is not useful for maintaining arc consistency. So, we remove it and also all the nodes that do not belong to a path from the root to tt. This can be done by performing a depth first search.

Semi-reduced MDD. We relax the fully reduced property of the MDD. We accept to have nodes with all k outgoing arcs pointing to the same node. We will say that we have a semi-reduced MDD. Note our algorithm may also work with MDDs having duplicate nodes, i.e., MDDs that are not even semi-reduced. Figure 2 gives an example of a reformulation. Each reduced arc in a fully reduced MDD is replaced by d arcs in the semi-reduced MDD. We do not find any problem for which it really changes the space complexity.

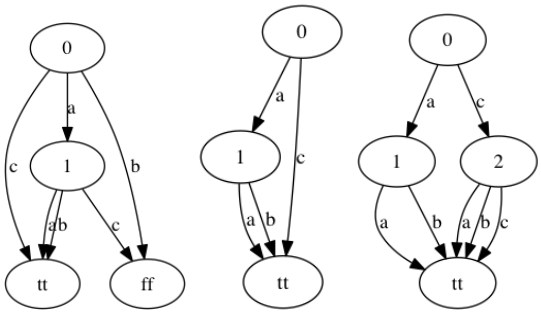

Fig. 2. Reformulation of an MDD. The left graph is the initial MDD. The middle graph represents the deletion of the node ff. The right graph is the semi-reduced MDD that will be used by MDD-4.

4.2 MDD-4 algorithm

The algorithm MDD-4 is a modification of GAC-4 for dealing with MDDs. We differentiate two parts: the maintenance of the MDD during the search for a solution and the maintenance of the S lists.

In GAC-4 the maintenance of the list of valid tuples is made by managing the S lists. With an MDD this is more complex, because the tuples are not explicitly represented in an MDD. The representation is implicit: a valid tuple corresponds to path from the root to the node tt which traverses only arcs corresponding to valid values. In order to avoid checking all the time the validity of the values, like in `mddc`, and to make sure that we do not uselessly traverse a path we propose to delete the arcs corresponding to values that are no longer valid. Then, we delete the nodes and arcs that do not belong to a path from the root to tt. We call this process the maintenance of the MDD. We do not need to explicitly search for such paths. It is sufficient to delete the nodes from which we can no longer reach the root or the node tt. The idea is to check whether a deleted arc (i, j) is the only outgoing arc from i or the only incoming arc to j, because if a node has no longer any outgoing arc or any incoming arc then we can remove it. We implement this process as follows. When values are removed from domains we delete the corresponding arcs in the MDD and push the extremities into two queues $Q\uparrow$ and $Q\downarrow$ (See Function REMOVEARC). Each queue can contain only once a node. If an arc (i, j) is deleted then i is pushed into $Q\uparrow$ and j is pushed into $Q\downarrow$. Then the queues are proceeded by layer (See Lines 2 and 3). The elements of $Q\uparrow$ are taken from the deepest to the shallowest layer. For each element $i \in Q\uparrow$ we remove i from the queue and we check whether there is an outgoing arc from i. If there is none, then i is removed from the graph, so arcs are deleted and new nodes are added to the queues. The elements of $Q\downarrow$ are taken from the shallowest to the deepest layer. For each element $j \in Q\downarrow$ we remove j from the queue and we check whether there is an incoming arc from j. If there is none, then node j is removed from the graph, so arcs are deleted and new nodes are added to the queues. The process is repeated until the queues are empty.

In order to remove values from domain variables we need to have a relation between the values and the arcs of the MDD. In MDD-4 this relation is defined by the S lists, so the S lists contain arcs instead of tuples as in GAC-4. Precisely, for each value (x, a), the list $S(x, a)$ contains the set of arcs in the MDD labeled with the value a at the layer of x. If there is no more arc for a value (x, a) in the MDD, then $S(x, a)$ becomes empty and we can safely remove a from $D(x)$. Once again, S lists are implemented by sparse sets.

A synopsis of the code of MDD-4 is given by Algorithm 5. In order to restore efficiently the MDD we use sparse sets for representing the neighborhood of the nodes.

We can establish an interesting property:

Property 2 *MDD-4 cannot perform more operations than GAC-4 for establishing arc consistency of a table constraint*

sketch of proof The deletion of a value in GAC-4 implies to consider all the current tuples containing the values. The deletion of a tuple in GAC-4 costs r operations (i.e., the arity of the constraint), because the tuple belongs to r S-list (one for each value of

Algorithm 5: MDD-4.

```
REMOVEARC(MDD, Q↓, Q↑, (i, j)):Boolean
    delete the arc (i, j) from the MDD
    y ← variable of the arc (i, j); b ← value of the arc (i, j)
    remove the arc (i, j) from the S(y, b)
    if S(y, b) = ∅ then remove b from D(y)
    push nodes i into Q↓ and j into Q↑ if they are not already in.
    return (D(y) ≠ ∅)
REVISEMDD-4(C: constraint; deletionSet: list): Boolean
  Q↓← ∅; Q↑← ∅
1 for each (x, a) ∈ deletionSet do
    for each arc (i, j) ∈ S(x, a) do
      if ¬ REMOVEARC(MDD, Q↓, Q↑, (i, j)) then return False
  while Q↓≠ ∅ or Q↑≠ ∅ do
2   while Q↑≠ ∅ do
      pick the node i ∈ Q↑ with the lowest layer
      if there is no outgoing arc from i then
        for each arc (j, i) do
          if ¬ REMOVEARC(MDD, Q↓, Q↑, (j, i)) then return False
      remove i from Q↑
3   while Q↓≠ ∅ do
      pick the node j ∈ Q↓ with the highest layer
      if there is no incoming arc to j then
        for each arc (j, i) do
          if ¬ REMOVEARC(MDD, Q↓, Q↑, (j, i)) then return False
      remove j from Q↓
  return True
```

the tuple). When a value(x, a) is deleted in MDD-4, we have to delete all the arcs corresponding to this value. The number of deleted arcs is bounded by the number of tuples involving (x, a). The deletion of (x, a) triggers the maintenance of the extremities of the deleted arcs. Since the deleted arcs belong to deleted tuples, the number of arcs that are considered through the nodes maintenance cannot be greater than the number of elements of all the tuples involving (x, a). Thus the property holds.

4.3 MDD-4R

MDD-4 can be improved by integrating the idea of resetting the data structures instead of being focused only on the deletions. MDD-4 works by layer in the MDD, that is, variable per variable. Let $\#A(x)$ be the total number of arcs associated with a value of a variable x. This number can be stored and maintained for each variable. For a given layer corresponding to the variable x, we have to compute $\#DA(x)$ the number of arcs that will be deleted for this layer. By comparing this number to $\#A(x)$ we will know

whether it is better to reset the layer or not. Resetting the layer means that we rebuild the layer of the graph from the remaining nodes by adding their remaining arcs instead of deleting the arcs and nodes of the layer. The computation of $\#DA(x)$ depends on the type of modification occurring in the MDD. There are two kinds of modifications:

- First, the arcs corresponding to the deletions of values of a variable x are removed. In this case, we have $\#DA(x) = \sum_{a \in \Delta(x)} |S(x,a)|$. This happens only once.
- Second, the consequences of the deletion of arcs and nodes in the MDD are propagated, in other words the MDD is maintained. MDD-4R proceeds by layer. For a given variable y, MDD-4 has the list $Q(y)$ of nodes that must be deleted, which is for the given layer the content of the queue $Q\uparrow$ or $Q\downarrow$ depending on the sense of propagation. We have $\#DA(y) = \sum_{z \in Q(y)} |N(y,z)|$, where $N(y,z)$ is the list of arcs associated with y having z as extremity.

Property 3 *Let x be any variable, If $\#DA(x) > \frac{\#A(x)}{2}$ then a reset operation for the layer of x will consider less arcs than the application of MDD-4 for this layer.*

Note that this property computes exactly whether it is better to reset or not the data structure.

The reset idea can be implemented in an easy way by using sparse sets for representing the neighborhood of the nodes.

5 Related work

Allowed of or-tools. The or-tools solver integrates an efficient algorithm when there are only few tuples. It is based on GAC-3. The algorithm, that we will name `allowed`, indexes the tuples and maintains a bitset of the valid tuples. For each variable x and for each value a, a bitset mask of tuples containing the value (x,a) is maintained. If a value (y,b) is removed then all the tuples of the bitset mask of (y,b) are cleared in the bitset of valid tuples. Then, the algorithm scans the values of all the variables to see if there is an active tuple which supports it thanks to bitwise operators.

STR2. It has been proposed in [14]. It is based on GAC-3. It maintains a sparse set of the valid tuples and improves the test of the validity of a tuple by considering only the variables whose domain recently changed. It also uses the tuples that have been computed valid as support for all the other values involved in the tuple, like in GAC-Schema, and so it decreases the number of values for which we have to search for a new support.

STR3. It has been defined in [15]. It is based on GAC-Schema (or GAC-6). For each variable x and for each value a of x it precomputes the list of tuples involving (x,a). It uses sparse set for maintaining the validity of tuples and to speed-up the test of validity.

We do not include the algorithms of [18] because the authors recognize that when the arity of the table increases, the existing state-of-the-art propagators STR3 and `mddc` are faster than their algorithms.

6 Experiments

Machine: Dell server having four E7- 4870 Intel processors, each having 10 cores with 256 GB of memory and running under Scientific Linux.
Solver: or-tools 3158.
Selected Benchmarks: all problems can be downloaded from the Solver Competition archive [13]. We selected problems having only positive table constraints and at least one variable whose domain is not Boolean. We do not include BDD-based instances and instances involving only binary constraints because we are interested in large arity constraints. rand-8-20-5-18-800 is abbreviated by rand-1 and rand-10-20-10-5-10000 is abbreviated by rand-2. half-n25-d5-e56-r7-1 is abbreviated by half-1 and contains the problems of half-n25-d5-e56-r7 that are solved by `mddc` in less than 1800s and half-n25-d5-e56-r7-2 is abbreviated by half-2 and contains the problems of half-n25-d5-e56-r7 that are not solved by `mddc` in less than 1800s.

We also used random problems that we defined. One of the most difficult parameter to define is the tightness of the constraint that is, the ratio between the number of tuples allowed by the constraint and the total number of tuples. We use ratio from 0.00004% to 1%. For a comparison, we can note that an alldiff constraint defined on a set of k variables sharing the same k values is equal to $\frac{k!}{k^k}$ which is 0.6% for $k = 7$; 0.034% for $k = 10$ and 0.0003% for $k = 15$.
Search Strategy: we select the variable that appears in most constraints and its smallest value as proposed for testing `mddc` [7]. Our algorithm is more robust than the Cheng's one for the strategy, because we maintain the MDD whereas they traverse the initial MDD according to the current domains. Thus `mddc` algorithm can lose time for finding an arc corresponding to a value belonging to the current domain of the associated variable. This problem does not arise in neither MDD-4 nor MDD-4R.
Results: times are expressed in seconds. Time Out (T-O) is set at 1800s. All means are geometric.

6.1 General comparison

Table 1. Geometric means of the time needed to solve some categories of problems.

benchmark	MDD-4	MDD-4R	`mddc`	GAC-4	GAC-4R	STR2	STR3	allowed
nonograms	0,33	**0,27**	0,8	4,3	3,18	2,77	1,52	1,03
cw-m1c-ogd	3,07	**1,72**	32,69	4,03	2,74	13,21	2,69	3,09
cw-m1c-uk	3,96	1,91	21,14	3,24	2,27	8,32	**1,89**	2,22
rand-1	6,06	2,93	13,71	2,90	1,38	1,56	1,93	**1,09**
rand-2	192,47	**50,51**	186,35	241,56	170,36	141,03	T-O	T-O
half-1	975,49	**471,25**	1438,20	T-O	T-O	T-O	T-O	T-O
half-2	1720	**778,28**	T-O	T-O	T-O	T-O	T-O	T-O

Table 1 gives results for the selected benchmarks. MDD-based algorithms perform well in general. Algorithm `mddc` is clearly improved by MDD-4 and MDD-4R algo-

rithms. GAC-4R outperforms GAC-4 and is competitive with other GAC algorithms for table constraints. The reset strategy is quite interesting and MDD-4R clearly outperforms all the other algorithms.

We propose to study in detail the behavior of these algorithms according to the tightness and the domain size. For each graph we select randomly 10 problems and run them 30 times and take the mean.

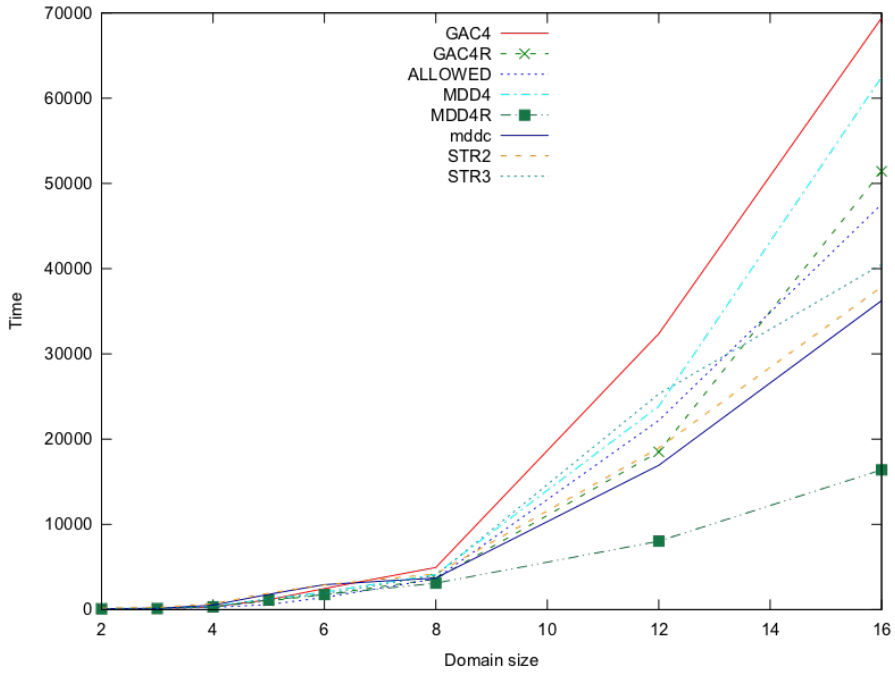

Fig. 3. Time needed to solve problems while increasing the domain size. tightness is 1%.

Domain Size. We study the behavior of the algorithms while changing the domain size. Each problem involves 100 variables and constraints of arity 7. The results are given in Fig. 3. Modifying the domain may change the tightness so we set it at 1%. Clearly MDD-4R is the best algorithm. The reset idea is also a clear improvement for GAC-4 and MDD-4. GAC-4R is competitive with STR2 and STR3.

Tightness. We set the domain size (8) and the arity (7) and we increase the tightness. Each problem involves 40 variables and 40 constraints.

Fig. 4 presents the results. Once again the reset idea is worthwhile. STR2 outperforms STR3 and GAC-4R.

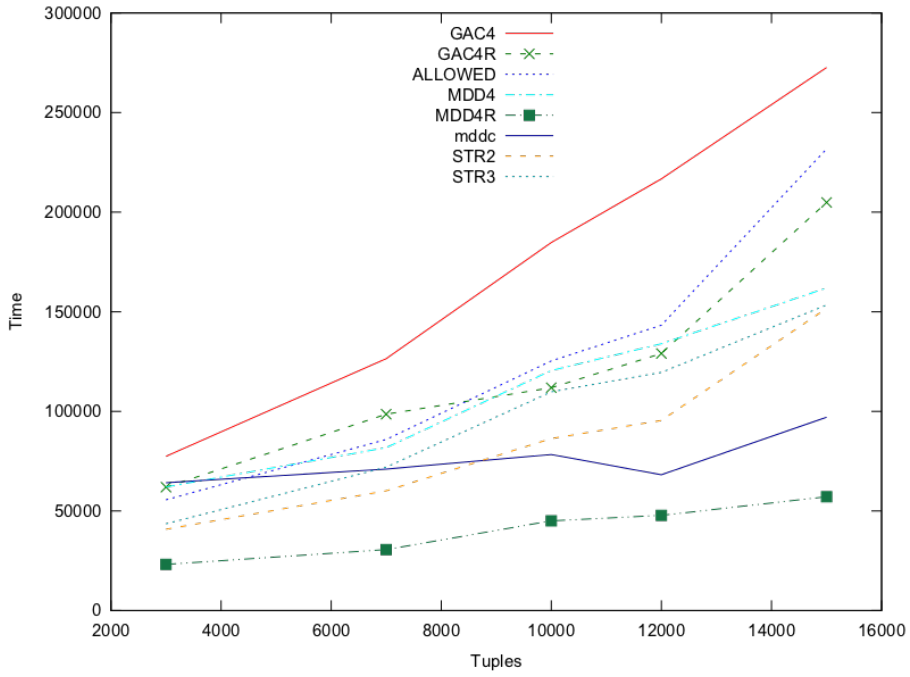

Fig. 4. Time needed to solve problems while increasing the tightness. Domain size is 8.

The conclusion of these experiments is that MDD-based algorithms clearly outperform the algorithms based on table constraints when the tightness and/or the size of the domains grow. The reset idea really improves the algorithms and GAC-4R is a competitive algorithm. The most robust and globally the most efficient algorithm is MDD-4R. This algorithm should be preferred in practice.

7 Conclusion

We have introduced MDD-4 and the two algorithms GAC-4R and MDD-4R which respectively improved the algorithms GAC-4 and `mddc`. These algorithms are mainly based on an adaptive method for maintaining the data structures. If a lot of deletions are made then we reset the data structures from the remaining elements instead of studying the consequences of the deletions. We have defined rules for maintaining the data structures in the best way for GAC-4R and MDD-4R. The experiments show that GAC-4R clearly improves GAC-4 and is competitive with STR2 and STR3. They also show that MDD-4R is a clear improvement of `mddc` and that it outperforms all existing algorithms.

References

1. Henrik Reif Andersen, Tarik Hadzic, John N. Hooker, and Peter Tiedemann. A constraint store based on multivalued decision diagrams. In *CP*, pages 118–132, 2007.
2. David Bergman, Willem Jan van Hoeve, and John N. Hooker. Manipulating mdd relaxations for combinatorial optimization. In *CPAIOR*, pages 20–35, 2011.
3. C. Bessière and J-C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, Nagoya, 1997.
4. C. Bessière and J-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315, Seattle, WA, USA, 2001.
5. Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2:59–69, 1993.
6. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C35(8):677–691, 1986.
7. K. Cheng and R. Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15, 2010.
8. Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad hoc r-ary constraints. In *CP*, pages 509–523, 2008.
9. I. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proc. AAAI'07*, pages 191–197, Vancouver, Canada, 2007.
10. Tarik Hadzic, John N. Hooker, Barry O'Sullivan, and Peter Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In *CP*, pages 448–462, 2008.
11. Samid Hoda, Willem Jan van Hoeve, and John N. Hooker. A systematic approach to mdd-based constraint programming. In *CP*, pages 266–280, 2010.
12. G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Proc. CP'07*, pages 379–393, Providence, USA, 2007.
13. Christophe Lecoutre. Csp/maxcsp/wcsp solver competitions. In *http://www.cril.univ-artois.fr/ lecoutre/benchmarks.html*, 2009.
14. Christophe Lecoutre. Str2: optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.
15. Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap. A path-optimal gac algorithm for table constraints. In *ECAI*, pages 510–515, 2012.
16. O. Lhomme and J-C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proc. AAAI'05*, pages 405–410, Pittsburgh, USA, 2005.
17. Olivier Lhomme. Practical reformulations with table constraints. In *ECAI*, pages 911–912, 2012.
18. Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. Optimal and efficient filtering algorithms for table constraints. *Constraints*, 19(1):77–120, 2014.
19. R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of ECAI-88*, pages 651–656, 1988.
20. Jean-Charles Régin. Ac-*: A configurable, generic and adaptive arc consistency algorithm. In *CP*, pages 505–519, 2005.
21. J-C. Régin. Improving the expressiveness of table constraints. In *CP'11, proceedings workshop ModRef'11*, 2011.
22. Michael Rice and Sanjay Kulhari. A survey of static variable ordering heuristics for efficient bdd/mdd construction. *University of California, Tech. Rep*, 2008.