Constructions and In-place Operations For MDDs Based Constraints

Guillaume Perez and Jean-Charles Régin

Université Nice-Sophia Antipolis, CNRS, I3S UMR 7271, 06900 Sophia Antipolis, France
guillaume.perez06@gmail.com, jcregin@gmail.com

Abstract. This papers extends in three ways our previous work about efficient operations on Multi-valued Decision Diagrams (MDD) for building Constraint Programming models. First, we improve the existing methods for transforming a set of tuples, Global Cut Seeds or sequences of tuples into MDDs. Then, we present in-place algorithms for adding and deleting tuples from an MDD. Finally, we describe an incremental version of an algorithm which reduces an MDD. We show on a real-life application that in-place operations on MDDs combined with this incremental algorithm outperform classical operations. Furthermore, we give some experimental results showing that the creation algorithms we propose strongly improve upon existing ones.

1 Introduction

Table constraints are useful constraints for modeling and solving many real-world problems. They are explicitly defined by the set of elements of the Cartesian product of the variables, also called tuples, that are allowed. The complexity of arc consistency algorithm associated with table constraints mainly depends on the number of involved tuples. Thus, Cheng and Yap proposed to compress the tuple set of the constraint by using Multi-valued Decision Diagrams (MDD) leading to MDD-based constraints. They designed `mddc`, one of the first filtering algorithms establishing arc consistency for them [8,7]. Recently, we have presented MDD-4R, a new algorithm which improves `mddc` [17]. MDD-4R proceeds like GAC-4R (an efficient arc consistency algorithm for table constraints) and, unlike `mddc`, maintains the MDD during the search for a solution. MDD-4R outperforms table constraints when the compression is effective.

MDDs can also be directly used to express complex constraints that cannot be represented by Table constraints because the number of tuples would be exponential. We have introduced efficient algorithms for creating and reducing an MDD and some powerful algorithms for combining MDDs [18]. Thanks to these new algorithms, some experiments based on real-life applications have shown that the MDD approach becomes competitive with ad-hoc approaches like the filtering algorithms associated with the regular or the knapsack constraints. More precisely we have shown that modeling a complex problem by a succession of operations between MDDs may be a competitive approach with the design of a complex ad-hoc algorithm.

In this paper, we extends our work by showing that MDDs can also be used to efficiently implement partially compressed table constraints like the ones defined by

Global Cut Seeds (GCS) or tuple sequences and by proposing some in-place algorithms for combining MDDs in order to avoid using intermediate MDDs, and by introducing an incremental reduction algorithm.

Table constraints can be specified either directly, by input from the user, or indirectly by synthesizing other constraints or subproblems [15,14]. They have been reinforced in order to deal either from tuple sets or from sequences of tuples [10,12,19]. This has two advantages: it improves their expressiveness and it reduces the number of tuples that are explicitly used and so decreases the practical complexity of the filtering algorithms because they mainly depends on that number.

GCSs and tuple sequences are partial compression of table constraints. This compression can be improved by transforming tables defined by GCS or tuple sequences into MDDs. In the first part of this paper we propose such transformations and we show that the obtained MDDs always uses less space than a set of GCS or tuple sequence for representing the same table. We will also present the first linear algorithm for building an MDD from a list of tuples.

Next, we consider in-place deletion and addition operations, that is operations that do not create a new MDD. Instead they directly modify the current MDD. In-place operations have three advantages: it avoids some memory consumption, it decreases the computation time, and it allows the design of more efficient reduction algorithms because they can be incremental. In this part, we show that the addition and the deletion of one tuple from an MDD can be efficiently done by using the method which consists of isolating the path of the MDD corresponding to the tuple in case of deletion and to the common prefix of the tuple in case of addition. These operations make addition/deletion operations easier on MDDs. Then, we generalize the algorithm for the addition/deletion of a set of tuples.

After each modification of an MDD the reduction operation must be applied and since the deletion or the addition of tuples may modify only a few nodes, we introduce IPREDUCE an incremental version of the reduction operation which allows us to reduce the complexity of the pair of operations formed by the modification and the reduction.

Before concluding, we present some experiments on a real life application showing some strong improvements brought by our algorithms notably compared to the ones previously proposed. We also empirically establish the advantages of the new creation algorithms we propose.

2 Background

MDD. Multi-valued decision diagram (MDD) is a data structure for representing discrete functions. It is a multiple-valued extension of BDDs [6]. An MDD, as used in CP [1,13,14,3,11], is a rooted directed acyclic graph (DAG) used to represent some multi-valued function $f : \{0...d-1\}^r \rightarrow \{true, false\}$, 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 at most d outgoing arcs to nodes in the next layer of the graph (i.e. one per value). We will denote by $L[i]$ the nodes in layer i and by $\omega^+(x)$ the set of outgoing arcs of the node x. Each arc is labeled by its corresponding value. The final layer

is represented by the true terminal node (the false terminal node is typically omitted). There is an equivalence between $f(v_1, ..., v_r) = true$ and the existence of a path from the root node to the true terminal node whose arcs are labeled $v_1, ..., v_r$. Nodes without any outgoing arc or without any incoming arc are removed.

MDD constraint. In an MDD constraint, the MDD models the set of tuples satisfying the constraint, such that every path from the root to the true terminal node corresponds to an allowed tuple. 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. Fig. 1 gives the MDD representing the tuples {a,a}, {a,b}, {c,a}, {c,b} and {c,c}. For each tuple, there is a path from the root node (node 0) to the terminal node (node tt) which is labeled by the tuple values.

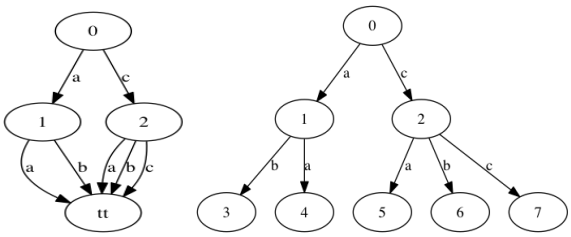

Fig. 1. An MDD (left graph) and a trie (right graph) representing the tuple set {{a,a},{a,b},{c,a},{c,b},{c,c}}

MDD reduction. The reduction of an MDD is one of the most important operations. It consists of merging equivalent nodes, i.e. nodes having the same set of outgoing neighbors associated with the same labels. Usually, a reduction algorithm merges nodes until there is no more equivalent nodes. Most of the time, only reduced MDDs are considered mainly because they are smaller. Fig. 5 exhibits an MDD having two equivalent nodes: b and e. These nodes will be merged by the reduction operation. Note that the reduction operation cannot increase the number of nodes or arcs. Recently, a new reduction algorithm with a linear space and time complexity has been proposed [18].

For convenience, we will denote by d the maximum number of values in the domain of a variable; and by (x, v, y) an arc from x to y labeled by v.

3 Transformations

In this section, we improve existing algorithms for building MDDs from tuple sets and we introduce new algorithms for building MDDs from compressed tuple sets.

3.1 From trie to MDD

A trie is a data structure used by Gent et al. for compressing tuple sets [12]. Each path from the root to a leaf represents an allowed tuple. A trie representing a set of T tuples

will have $|T|$ leaves. Each variable corresponds to a layer of the trie. A node has a maximum of d children, where d is the size of the domain of the corresponding variable of the node. An example of trie is given in Fig. 1. A trie can be transformed into an MDD by merging all the leaves into the terminal node tt and by applying the reduction operation [7].

3.2 From table to MDD

A table is a data structure where each row represents a tuple and where each column corresponds to a value of a variable.

Cheng and Yap build an MDD from a table by defining a trie. Tuples are successively added to the trie. First, a common node is created: the root of the trie. Then paths starting from the root are created. The rooted subpaths common to several tuples are merged together in order to be represented only once. Afterwards, all the leaves are merged and the MDD is reduced. The drawback of this approach is the addition of a tuple, because we need to compute the common subpath of the tuple and the MDD. This operation can be performed in linear time only if we have d entries per node, so we increase the space complexity. Alternatively, we can keep a linear space complexity if we accept to increase the time complexity.

We propose a simple method with a linear time and space complexity: we lexicographically sort the table and we build the trie from the sorted table. Here is an example:

table					sorted table					trie				
a	a	c	a	a	a	a	b	a	b	a	a	b	a	b
a	b	a	b	b	a	a	b	a	c					c
a	a	b	a	c	a	a	c	a	a			c	a	a
a	a	b	a	b	a	b	a	a	b		b	a	a	b
a	b	a	a	b	a	b	a	b	b				b	b

This can be done efficiently because all tuples are consecutive and so there is no need to search for any position for a tuple: the last one is always the correct one. So we do not need the random access to children and this step can be achieved in linear time. In addition, the sort can be performed in linear time because a tuple can be viewed as numbers having r digits where a digit can take on up to d values. Thus we can sort a table containing t tuples in $O(r(t+d))$ by using a radix sort, which is linear in its size. Since, the merge of the leaves and the reduction can be performed in linear time, we obtain a linear time algorithm.

3.3 From GCS and Tuple Sequence to MDD.

Compressed tuples improve the expressiveness of table constraints and reduce the complexity of the filtering algorithms. Therefore, it is interesting to represent them by MDDs in order to reinforce the compression.

A GCS (Global Cut Seed), is a compact representation of a tuple set [10]. A GCS is defined by a set of value sets: $\{\{v_{1,1}, v_{1,2}, ..., v_{1,k_1}\}, ..., \{v_{n,1}, v_{n,2}, ..., v_{n,k_n}\}\}$, where each value set corresponds to a variable. The Cartesian product of these sets defines the

represented tuples. For instance, given D ={1,2,3,4}, the GCS $c = \{D, D, D, D\}$ represents the tuple set { {1,1,1,1}, {1,1,1,2},..., {4,4,4,3}, {4,4,4,4}}. One GCS may represent an exponential number of tuples. However all the tuples cannot be compressed by only one GCS. Two tuples can be represented by the same GCS if they have a Hamming distance equals to 1. For instance, the tuples {1,1,1} and {1,1,2} may be compressed into {1,1,{1,2}}. By contrast the tuples {1,1,1} and {1,2,2} have an Hamming distance equals to 2 and so cannot be represented by only one GCS. So, the compression of a table by a set of GCSs may require a huge number of GCSs. In order to remedy this problem, tuple sequences have been introduced [19]. They generalize GCSs.

A tuple sequence encapsulates a GCS and two tuples: t_{min} a minimum tuple, and t_{max} a maximum tuple. It bounds the lexicographic enumeration of the tuples of the GCS by these two tuples. For instance, let $D = \{1, 2, 3, 4\}$ then the tuple sequence $s = \{\{D, D, D, D\}, \{1, 2, 2, 2\}, \{3, 1, 3, 2\}\}$ represents the tuple set {{1,2,2,2}, {1,2,2,3}, ..., {3,1,3,1}, {3,1,3,2}}.

Since a tuple sequence is a generalization of a GCS, a method transforming a tuple sequence into an MDD could also be used for transforming a GCS into an MDD.

First, we propose an algorithm for representing one tuple sequence by an MDD. Then, we will show how we can deal with several tuple sequences. Let $s = (g, t_{min}, t_{max})$ be a tuple sequence. For transforming s into an MDD we introduce special nodes: wild card nodes. There is at most one wild card node per layer i which is denoted by $w[i]$. The wild card nodes are linked together. All the arcs outgoing from $w[i]$ are incoming arcs of node $w[i + 1]$ and all arcs outgoing $w[n - 1]$ are incoming arcs of tt.

The MDD representing s is built in three steps:

1. The paths corresponding to t_{min} and t_{max} are created.
2. Arcs from the nodes of the paths previously created to wild card nodes are created as follows. Consider the path created for t_{min}. For each layer i, let $val[i]$ be the value set of g for the layer i. For each value $a \in val[i]$ such that $a > t_{min}[i]$ we create an arc from the node n_i of the path representing t_{min} to the wild card node $w[i + 1]$. We repeat this process for the path created for t_{max}. In addition, we add a particular treatment when a node is shared by the two initial paths: instead of considering all values of $val[i]$, we consider only the values in the interval $val[i] \cap]t_{min}[i], t_{max}[i][$.
3. From nodes $w[i]$ to node $w[i + 1]$ we add as many arcs as there are values in $val[i + 1]$.

Fig. 2 shows the resulting MDD. The left graph contains the two paths representing the minimum and maximum tuples. The right graph represents with dashed lines the added arcs to wild card nodes. For instance, for node a each value in {1,2,3,4} greater than 2 labels an arc to node w_2. Arcs joining wild card nodes together and with tt are represented by dotted lines.

Let r be the the number of involved variables. The number of nodes of the obtained MDD is bounded by $3(r-1)+2$. There are $2r$ arcs for the paths corresponding to t_{min} and t_{max}. There are at most $|val[i]|$ arcs from nodes of the t_{min} (resp. t_{max}) path to wild card nodes; There are $|val[i+1]|$ arcs from node $w[i]$ to node $w[i+1]$. Thus, there are at most $2\sum_{i=1}^{r} |val[i]| + 2r$ arcs in the MDD. This is equivalent to the number of values of the tuple sequence.

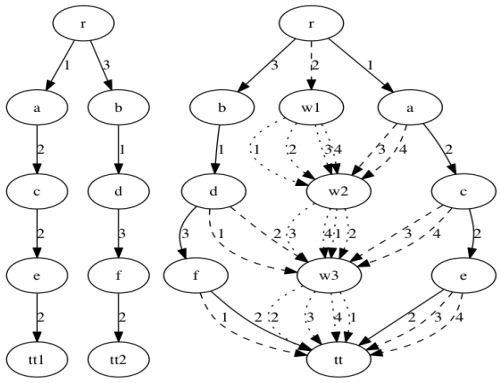

Fig. 2. Creation of an MDD from a tuple sequence

Now, suppose that we have a set of tuple sequences. We can consider successively each tuple sequence and build for each sequence an MDD with the previous algorithm. Then, there are two possibilities. Either the tuple sequences are disjoint or not. The former case arises frequently (for instance when the tuple sequences represent a set of forbidden tuples). We just have to make the union of MDDs. This can be easily done because they are disjoint. The resulting MDD has a space complexity equivalent to the set of tuple sequences and we have:

Property 1 *A set of disjoint tuple sequences can be represented by an MDD having an equivalent space complexity.*

The latter case is more complex. A set of disjoint tuple sequences may be computed from a set of non disjoint tuple sequences and each disjoint tuple sequence can be represented by an MDD. Nevertheless, it may create an exponential number of tuple sequences [19] so an exponential number of MDDs.

4 Addition and Deletions of tuples from an MDD

In this section, we define in-place algorithms for the addition/deletion of tuples from an MDD. Some work have been carried out for performing operations on BDDs. For instance, Bryant define some algorithms for applying different operators [6,5]. However, the described algorithms are not in-place (i.e. there is the creation of a resulting BDD) and it is not easy to generalize some algorithms designed for BDDs to MDDs mainly because some Booleans rules are no longer true when we have d values in the domain and because the complexity of some algorithms is multiplied by $O(d)$ when dealing with d values. Some generic algorithms have been proposed for applying operators on MDDs [2,18], but they are not in-place. An in-place algorithm has been given by Ciré and Hooker [9] but it only deal with partial assignments and has no incremental reduction.

4.1 Deletion of tuples from an MDD

First, we give an algorithm for deleting one tuple from an MDD. Then, we generalize it for a set of tuples.

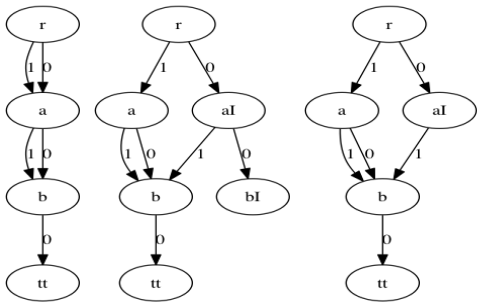

Fig. 3. Tuple {0,0,0} is removed from the left MDD. The isolation of the path corresponding to the tuple is performed (middle MDD) and then the reduction is applied (right MDD). Nodes aI and bI are created from nodes a and b during the path isolation.

The deletion of a tuple τ from an MDD is based on an operation named path isolation, which is a kind of local decompression. The idea is to build a specific path whose arcs are labeled by the values of τ. Furthermore, arcs equivalent to the ones of the isolated path are deleted from the MDD. After the isolation process, the MDD is reduced. Let $\tau[i]$ be the value for the variable $x[i]$. The isolation is performed in 3 steps:
Step 1. We identify $a_1 = (root, \tau[1], n_1)$ the arc of the first layer labeled by $\tau[1]$ the first value of the tuple. We create the node ne_1, the arc $(root, \tau[1], ne_1)$ and we delete the arc a_1. We set x_{mdd} (a node of the MDD) to n_1 and x_{path} (an isolated node) to ne_1.
Step 2. For each layer i from 2 to $r-1$ we repeat the following operation. We identify $a_i = (x_{mdd}, \tau[i], n_{i+1})$ the outgoing arc from x_{mdd} labeled with $\tau[i]$. We create the node ne_{i+1} and the arc $(x_{path}, \tau[i], ne_{i+1})$. For each arc (x_{mdd}, w, y) such that $w \neq \tau[i]$ we create the arc (x_{path}, w, y). We set x_{mdd} to n_{i+1} and x_{path} to ne_{i+1}.
Step 3. For each arc (x_{mdd}, w, tt) such that $w \neq \tau[r]$ we create the arc (x_{path}, w, tt).

If at any moment we cannot identify an arc then it means that τ does not belong to the MDD. Fig. 3 shows the application of this algorithm. The complexity of the deletion of a tuple is bounded by $O(rd)$ because for each isolated node we need to recreate its arcs. However, in practice it is often close to $O(d)$.

Deletion of a set of tuples. We propose a better method than repeating the previous algorithm for each tuple. We transform the set of tuples into an MDD and we subtract this new MDD from the initial one by following the same steps of the previous algorithm. We isolate nodes having a common path in both MDDs, then we remove the common arcs to the isolated nodes of the second last layer. At last, we call the incremental reduction algorithm.

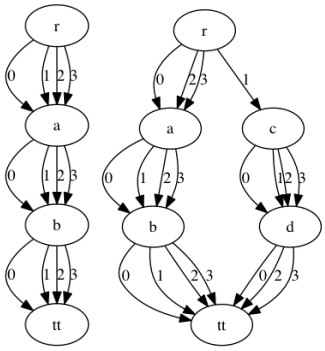

Fig. 4. The left MDD represents all the possible tuples for the values {0,1,2,3}. The right MDD represents the deletion of the GCS {1,{0,1,2,3},1} from the left MDD.

Fig. 4 shows the subtraction of the GCS {1,{0,1,2,3},1} from the MDD representing all the tuples possible for the values {0,1,2,3}. The GCS is isolated from the MDD. Then, the deletion of the arc labeled 1 of node d correspond to the deletion of only the tuples contained in the GCS. It is difficult to bound the complexity of the deletion of T tuples, because the MDD created from them may compress the information.

4.2 Addition of tuples to an MDD

The addition of tuples into MDD follows the same principles as for the deletion. In this case, the isolated path contains arcs labeled by the values of the tuple that must be added. It is performed by applying the same steps as for the deletion.

First, we consider the addition of one tuple τ. The two first steps are very similar to the ones of the deletion. Excepted that at a point, there will be no more path in the MDD having the same subpath as τ. Otherwise, it would mean that τ is already in the MDD. Thus, at a certain moment we will not be able to identify any arc $(x_{mdd}, \tau[i], n_{i+1})$ as in step 2 in the deletion algorithm. When this case arises we can stop step 2 and directly create the path from the current isolated node to the terminal node. This path will be labeled by the values of τ for the remaining layers. Step 3 can be skipped. At last, we call the incremental reduction algorithm. The complexity of the addition of a tuple is in $O(rd)$ because for each isolated node we need to recreate its arcs.

Addition of a set of tuples. Let mdd_1 be the initial MDD. We transform the set of tuples into an MDD, named mdd_2. We add mdd_2 to mdd_1 by following the same steps as for the previous algorithm. We isolate nodes having a common path in both MDDs. When an arc belongs to mdd_2, we create an isolated node and we create an arc from the current isolated node to it. When an arc belongs only to mdd_1, we create an arc from the current isolated node to the node in mdd_1.

Fig. 5 shows the effect of the addition of the tuple {1,2,1} in the MDD given in Fig. 4. We can see the usefulness of the path isolation for avoiding the addition of the

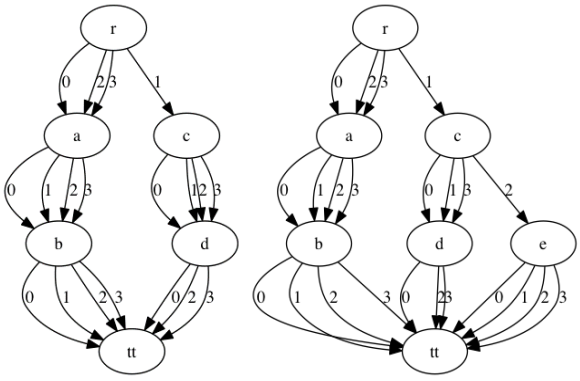

Fig. 5. The right MDD represents the addition of the tuple {1,2,1} to the left MDD, before the reduction.

tuples {1,{0,1,3},1}. The right MDD shows the impact of the reduction on the MDD: nodes e and b are merged because they have the same outgoing arcs. It is difficult to bound the complexity of the addition of T tuples, because the MDD created from them may compress the information.

Algorithm 1 is a possible implementation of the in-place deletion and addition operations.

4.3 Incremental Reduction

A reduction step is needed after the deletion/addition of tuples. Using a generic algorithm is costly because it will traverse all the nodes of the MDD and merge the equivalent ones. Since we consider that we add/delete tuples from an MDD which is reduced we can save some computations for the reduction applied after the operation. Only certain nodes have to be considered:

Property 2 *After the application of an in-place operator, if two nodes are merged then one of these nodes must be an isolated node.*

proof: Two nodes are merged if and only if they have the same set of outgoing neighbors associated with the same labels. Before the operation the MDD is reduced, so no two pairs of nodes can be merged. By induction from the terminal node to the root we prove the property: it is obvious for two nodes of the last layer. Then, from the definition of the path isolation, if a merge exists then it necessarily involved an isolated node because it was not possible to merge nodes of the MDD existing before the operation. □

Thus, we can easily adapt PREDUCE algorithm [18] by rejecting a pack of nodes if it does not involve any isolated node. In addition, it is easy to identify isolated nodes because they belong to the list L of the in-place algorithms. The advantage of this approach is that the reduction step does not increase the complexity of the addition or deletion operations. This new algorithm is named IPREDUCE.

Algorithm 1: In-place Deletion and Addition Algorithms

```
DELETION(L, mdd_1, mdd_2)
    for each (root(mdd_1), v, y_1) ∈ ω^+(root(mdd_1)) do
        if ∃(root(mdd_2), v, y_2) ∈ ω^+(root(mdd_2)) then
            ADDARCANDNODE(L, 1, root(mdd_1), v, y_1, y_2)
            DELETEARC(root(mdd_1), v, y_1)
    for each i ∈ 1..r − 2 do
        L[i] ← ∅
        for each node x ∈ L[i − 1] do
            get x_1 and x_2 from x = (x_1, x_2)
            for each (x_1, v, y_1) ∈ ω^+(x_1) do
                if ∃(x_2, v, y_2) ∈ ω^+(x_2) then
                    ADDARCANDNODE(L, i, x, v, y_1, y_2)
                else CREATEARC(L, i, x, v, y_1)
    for each node x ∈ L[r − 1] do
        get x_1 and x_2 from x = (x_1, x_2)
        for each (x_1, v, tt) ∈ ω^+(x_1) do
            if ∄(x_2, v, y_2) ∈ ω^+(x_2) then
                CREATEARC(L, r, x, v, tt)
    IPREDUCE(L)

ADDITION(L, mdd_1, mdd_2)
    for each v ∈ ω^+(root(mdd_1)) ∪ ω^+(root(mdd_2)) do
        if ∃ (root(mdd_1), v, y_1) ∈ ω^+(root(mdd_1)) then
            if ∃ (root(mdd_2), v, y_2) ∈ ω^+(root(mdd_2)) then
                ADDARCANDNODE(L, 1, root(mdd_1), v, y_1, y_2)
                DELETEARC(L, i, root(mdd_1), v, y_1)
        else ADDARCANDNODE(L, 1, root(mdd_1), v, nil, y_2)
    for each i ∈ 1..r − 2 do
        L[i] ← ∅ for each node x ∈ L[i − 1] do
            get x_1 and x_2 from x = (x_1, x_2)
            // If x_1 is nil then ω^+(x_1) is empty
            for each v ∈ ω^+(x_1) ∪ ω^+(x_2) do
                if ∃ (x_1, v, y_1)  ω^+(x_1) then
                    if ∃ (x_2, v, y_2) ∈ ω^+(x_2) then
                        ADDARCANDNODE(L, i, x, v, y_1, y_2)
                    else CREATEARC(L, i, x, v, y_1)
                else ADDARCANDNODE(L, i, x, v, nil, y_2)
    for each node x ∈ L[r − 1] do
        // If x_1 is nil then ω^+(x_1) is empty
        for each v ∈ ω^+(x_1) ∪ ω^+(x_2) do
            CREATEARC(L, i, x, v, tt)
    IPREDUCE(L)

ADDARCANDNODE(L, i, x, y_1, v, y_2)
    if ∄y ∈ L[i] s.t. y = (y_1, y_2) then
        y ← CREATENODE(y_1, y_2)
        add y to L[i]
    CREATEARC(x, v, y)
```

5 Experiments

The algorithms have been developed on top of or-tools 3158, a constraint programming solver developed by Google. The experiments have been executed on a MacBook Pro (Intel Core I7, 2.3GHz, 8GB memory).

Real life application. We consider the problem given in [16] which deals with Markov Sequence Generations on corpus having more than 10,000 words. The goal is to generate phrases having 24 words where all successions of 4 words come from the corpus and where there is no sequence of more than 8 words coming from the corpus. This problem can be modeled by using MDDs expressing sequences of words [18]. Values of variables are words of the corpus, so we have a huge number of values. From an initial MDD representing allowed sequences of 4 words, 20 intersections of MDDs are performed until obtaining mdd_r the final MDD. The main issue with this approach is the size of the MDDs because mdd_r has 1,208,219 nodes and 188,035,203 arcs. With the operators given in [18] were able to compute mdd_r in 425s. This requires to perform 20 intersections and 20 reductions of huge MDDs.

In this problem, twice a deletion followed by a reduction of the MDD are made. The results are given in the table below. Times are expressed in seconds. In the "Classic" columns, the algorithms given in [18] are used whereas the algorithm proposed in this paper are used in the "In-place" columns. These results clearly show the advantage of using the new algorithms. Using in-place algorithms instead of building intermediate MDDs reduces the memory consumption of the resolution of the whole problem from 52GB to 32GB.

	Classic			In-place		
	deletion	reduction	total	deletion	reduction	total
First Operation	2	1.7	3.7	**1.3**	**0.9**	**2.2**
Second Operation	23.9	14.6	38.5	**1.5**	**6.3**	**7.8**

Operations and Reduction We propose to compare the performance of the classical and the in-place algorithms and the performance of the classical and the incremental reduction algorithms. We use random instances obtained from the real life instances. The first number corresponds to the number of the tuples represented by the MDD whereas the second number is the number of tuples that are removed from the MDD. Table 1 gives the results we obtain. Our algorithms clearly improve the previous ones.

	Classic		In-place	
instances	deletion	reduction	deletion	reduction
30*300K-300K	35,4	4.2	24.8	1.8
300K - 1K	5.3	0.7	1.2	0.6
90K-30K	2.1	0.2	1.6	0.2
300K-10	4.7	0.6	0.002	0.2

Table 1. Arity 12, domain size 10. Average deletion time (s) for random instances.

We also proposes a table summarizing the advantages of the different algorithms. We add results for the BDD and MDD packages proposed in [20,4] (See column Bryant). "P&R15" represents the results we previously obtained and "in-place" column corresponds to the new algorithms. Table 2 gives some resultats for MDD representing 10,000s tuples. Note that huge MDD are not tractable with some old methods.

#tuples	#deleted	Bryant	P&R15	in-place
20000	1000	159	11.5	6
40000	2000	291	40	21
40000	20000	663	51	33
80000	40000	2643	174	114
40000	10	466	185	19

Table 2. Arity 12, domain size 10. Average deletion time (ms) for random instances.

From tables to MDDs We study the performance of the new creation algorithms. The times for sorting the elements are included into our results. First, we tested our algorithm on the instances of the XCSP competition. We give the results for the most representative ones. Sorted creation corresponds to our algorithm, unsorted creation is the classical creation.

instances	creation	
	sorted (ms)	unsorted (ms)
crossword-m1c-ogd	**31.5**	66.2
crossword-m1c-uk-vg	**9.6**	23.1
nonogram-gp	**25.1**	34.5
rand-10-60-20-30	**70.9**	179.9
bdd-21-2713	**8.1**	11.6
bdd-21-133	**98.23**	122.3

These experiments show that it is always better to sort the table and use our creation algorithm.

On the other hand, we tested both algorithms on random instances. We have tested instances having 22 variables, 1,000 tuples and we increased the domain size. The results given in Fig. 6 show that the domain size does not influence the creation time. We can see that even if we increase the number of tuples or the number of variables, our creation algorithm outperforms the existing one. We have also tested instances for all the combinations with domain size in the set {2, 4, 8, 12, 20, 25, 30, 45, 60}, arity in the set {6, 10, 14, 18, 22, 25, 30} and number of tuples in the set{30, 100, 150, 200, 250, 300, 500, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 7500, 10000, 12500, 15000, 17500, 20000, 24000, 28000, 30000}. For all these cases, our method was better.

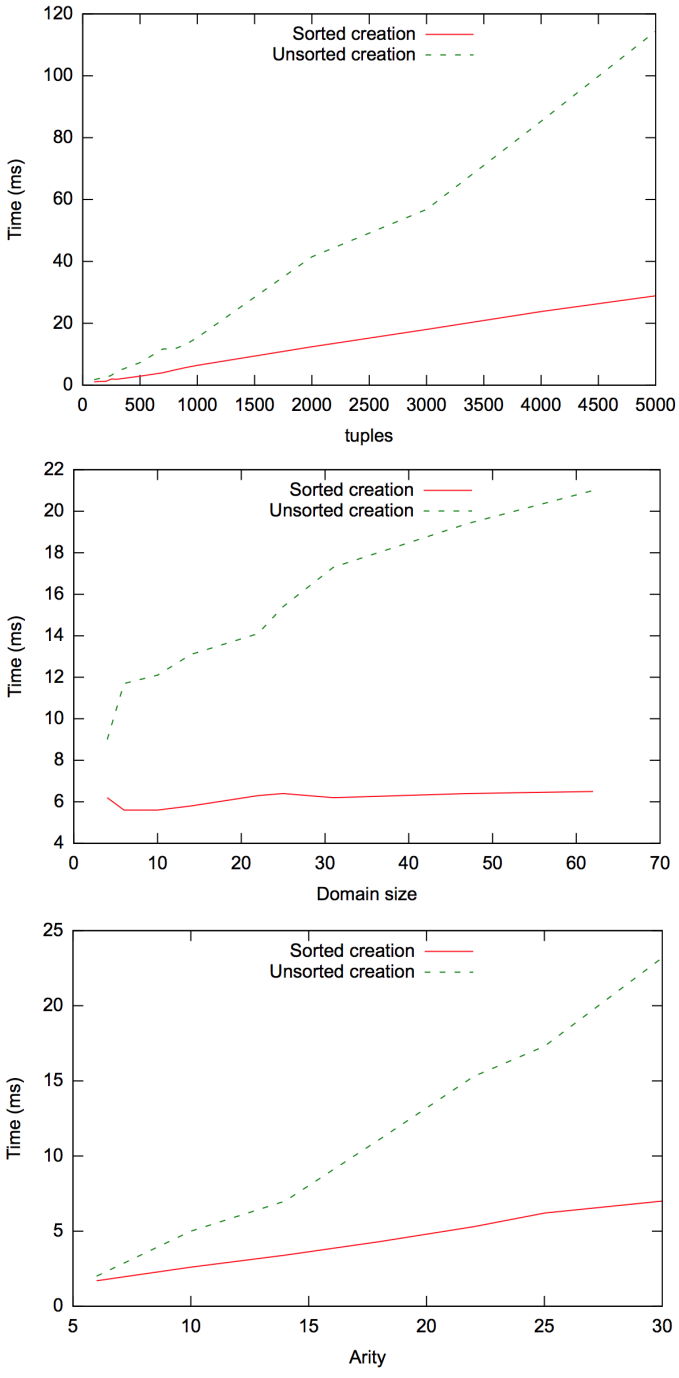

Fig. 6. Sorted vs unsorted creation

6 Acknowledgments.

We would like to thank very much Laurent Perron and Christophe Lecoutre for their useful comments which helped to improve the paper

7 Conclusion

We have given an algorithm for transforming tuple sets, GCS and tuple sequences into an MDD. Then, we have described efficient in-place algorithms for adding or deleting tuples from an MDD. These algorithms are based on the idea of path isolation. Furthermore, we have introduced a simple modification of the PREDUCE algorithm for improving the reduction of an MDD when it is used after an in place operation. We have experimentally shown on a real life application, on a set of benchmarks and on random problems that the algorithms we propose outperform the existing ones.

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. D. Bergman, A. Cire, and W-J. van Hoeve. Mdd propagation for sequence constraints. *Journal of Artificial Intelligence Research*, 50:697–722, 2014.
3. David Bergman, Willem Jan van Hoeve, and John N. Hooker. Manipulating mdd relaxations for combinatorial optimization. In *CPAIOR*, pages 20–35, 2011.
4. Karl S Brace, Richard L Rudell, and Randal E Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE design automation conference*, pages 40–45. ACM, 1991.
5. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
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*, 15:265–304, 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. André A. Ciré and John N. Hooker. The separation problem for binary decision diagrams. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2014, Fort Lauderdale, FL, USA, January 6-8, 2014*, 2014.
10. F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proc. CP'01*, pages 77–92, Paphos, Cyprus, 2001.
11. G. Gange, P. Stuckey, and Radoslaw Szymanek. Mdd propagators with explanation. *Constraints*, 16:407–429, 2011.
12. 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.
13. 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.
14. Samid Hoda, Willem Jan van Hoeve, and John N. Hooker. A systematic approach to mdd-based constraint programming. In *CP*, pages 266–280, 2010.

15. Olivier Lhomme. Practical reformulations with table constraints. In *ECAI*, pages 911–912, 2012.
16. A. Papadopoulos, P. Roy, and F. Pachet. Avoiding plagiarism in markov sequence generation. In *Proceeding of the Twenty-Eight AAAI Conference on Artificial Intelligence*, pages 2731–2737, 2014.
17. G. Perez and J-C. Régin. Improving GAC-4 for table and MDD constraints. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 606–621, 2014.
18. G. Perez and J-C. Régin. Efficient operations on mdds for building constraint programming models. In *International Joint Conference on Artificial Intelligence, IJCAI-15*, pages 374–380, Argentina, 2015.
19. J-C. Régin. Improving the expressiveness of table constraints. In *CP'11, proceedings workshop ModRef'11*, 2011.
20. Arvind Srinivasan, Timothy Ham, Sharad Malik, and Robert K Brayton. Algorithms for discrete function manipulation. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 92–95. IEEE, 1990.