Checking Constraint Satisfaction

Victor Jung, Jean-Charles Régin

Checking Constraint Satisfaction

Victor Jung[0000−1111−2222−3333] and Jean-Charles Régin[0000−0001−6204−5894]

Université Côte d'Azur, CNRS, I3S, France
{victor.jung,jean-charles.regin}@univ-cotedazur.fr

Abstract. We address the problem of verifying a constraint by a set of solutions S. This problem is present in almost all systems aiming at learning or acquiring constraints or constraint parameters. We propose an original approach based on MDDs. Indeed, the set of solutions can be represented by the MDD denoted by MDD_S . Checking whether S satisfies a given constraint C can be done using $MDD(C)$, the MDD that contains the set of solutions of C, and by searching if the intersection between $MDD(S)$ and $MDD(C)$ is equal to $MDD(S)$. This step is equivalent to searching whether $MDD(S)$ is included in $MDD(C)$. Thus, we give an inclusion algorithm to speed up these calculations. Then, we generalize this approach for the computation of global constraint parameters satisfying C. Next, we introduce the notion of properties on the MDD nodes and define a new algorithm allowing to compute in only one step the set of parameters we are looking for. Finally, we present experimental results showing the interest of our approach.

Keywords: Multi-valued Decision Diagram · Inclusion · Constraint Learning.

1 Introduction

Many works in Constraint Programming try to improve the quality of a model by adding new implicit constraints[11], redundant constraints [4] or global constraints [8]. All these works face a common problem: the verification of the satisfaction of constraints by a given set of solutions. Some choose a brute force approach [8,11], others prefer a more specific but ad-hoc approach [2]. In all cases, these methods go through the solutions to test if they satisfy constraints. Constraints are not necessarily tested individually, but the solutions can be considered one after the other.

In this paper, we propose a more global and efficient method to test whether a set of solutions verifies one or a set of constraints. Multi-valued decision diagrams (MDDs) are a very efficient data structure to represent a set of solutions in a compressed way and for which many operators are available to combine MDDs without decompressing them. We therefore propose to use $MDD(S)$ the MDD which corresponds to the set of solutions S. We show that we can simply test if S satisfies a constraint C, by using $MDD(C)$, the MDD that represents the solutions of C, and then by performing the intersection between $MDD(S)$

and $MDD(C)$. This method is simple to implement since it only requires constructing the two MDDs and performing their intersection, for which efficient algorithms are available, and then testing whether the resulting MDD is similar to $MDD(S)$. However it has an important flaw: it will create and calculate a MDD even if the intersection will not be equal to $MDD(S)$. It therefore risks doing many operations unnecessarily. To avoid this we introduce an inclusion operator between MDDs since this is what we want to test: is $MDD(S)$ included in $MDD(C)$?

This operator is efficient when it is a question of verifying a precise and unique constraint, but not very efficient when it is a question of searching for the parameters of a constraint such that the resulting constraint is satisfied by a set of solutions. Finding the parameters of a global constraint so that it is satisfied by a set of solutions is a recurrent problem at present ([11], [12], [8], [2], [4]). To solve this problem we propose to work with $MDD(S)$ which we enrich by introducing the notion of node properties. Then, a process called "the parent-child propagation of the parameters" is performed through the MDD. More precisely, the global constraint is expressed by properties including the parameters and these properties are propagated in $MDD(S)$ in order to compute for each sub-tree of the MDD those which are compatible with the constraint. Thus, we determine the most restrictive parameters of the constraint that are satisfied by the MDD.

This article is organized as follows. First, we give some basic definitions. Then, we present a general scheme to check the satisfaction of a constraint and improve it by defining a new operation between MDDs. Next, we address the problem of finding parameters of global constraints by introducing the notion of node properties. Finally, we provide benchmarks and results testing the different approaches described in this article, and we conclude.

2 Preliminaries

2.1 Constraint Programming

A finite constraint network $\mathcal{N}$. is defined as a set of n variables $X = \{x_1, \ldots, x_n\}$, a set of current 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. We introduce the particular notation $\mathcal{D}_0 = \{D_0(x_1), \ldots, D_0(x_n)\}$ to represent the set of initial domains of $\mathcal{N}$ on which constraint definitions were stated. 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 $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ that specifies the allowed combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An element of $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ is called a tuple on $X(C)$. A value a for a variable x is often denoted by (x, a). Let C be a constraint. A tuple τ on $X(C)$ is valid if $\forall (x, a) \in \tau, a \in D(x)$. C is consistent iff there exists a tuple τ of $T(C)$ which is valid. A value $a \in D(x)$ is consistent with C iff $x \notin X(C)$ or there exists a valid tuple τ of $T(C)$ with $(x, a) \in \tau$. We denote by $\#(a, \tau)$ the number of occurences of the value a in a tuple τ.

We present some constraints that we will use in the rest of this paper.

A global cardinality constraint (GCC) constrains the number of times every value can be taken by a set of variables. This is certainly one of the most useful constraints in practice. Note that the ALLDIFF constraint corresponds to a GCC in which every value can be taken at most once.

Definition 1 *A* **global cardinality constraint** *is a constraint C in which each value $a_i \in D(X(C))$ is associated with two positive integers l_i and u_i with $l_i \leq u_i$ defined by*
$\text{GCC}(X, l, u) = \{\tau | \tau \text{ is a tuple on } X(C) \text{ and } \forall a_i \in D(X(C)) : l_i \leq \#(a_i, \tau) \leq u_i\}$

Definition 2 *Given X a set of variables, l and u two integers with $l \leq u$ and V a set of values. The* **among** *constraint ensures that at least l variables of X and at most u will take a value in V, that is*
$\text{AMONG}(X, V, l, u) = \{\tau \mid \tau \text{ is a tuple on } X(C) \text{ and } l \leq \sum_{a \in V} \#(a, \tau) \leq u\}$

This constraint has been introduced in CHIP [1].

The SEQUENCE constraint [1] is a conjunction of sliding AMONG constraints.

Definition 3 *Given X a set of variables, q, l and u three integers with $l \leq u$ and V a set of values. The* **sequence** *constraint holds if and only if for $1 \leq i \leq n_q+1$ $\text{AMONG}(\{x_i, ..., x_{i+q-1}\}, V, l, u)$ holds. More precisely*
$\text{SEQUENCE}(X, V, q, l, u) = \{ \tau \mid \tau \text{ is a tuple on } X(C) \text{ and for each sequence } S$
of q consecutive variables: $l \leq \sum_{v \in V} \#(v, \tau, S) \leq u\}$

2.2 Multi-valued Decision Diagram

The decision diagrams considered in this paper are reduced, ordered multi-valued decision diagrams (MDD) [7,13,3], which are a generalization of binary decision diagrams [5]. They use a fixed variable ordering for canonical representation and shared sub-graphs for compression obtained by means of a reduction operation. An MDD is a rooted directed acyclic graph (DAG) used to represent some multi-valued functions $f : \{0...d-1\}^n \rightarrow true, false$. Given the n input variables, the DAG contains $n+1$ 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. Each arc is labeled by its corresponding integer. The arc (u, a, v) is from node u to node v and labeled by a. Sometimes it is convenient to say that v is a child of u. All outgoing arcs of the layer n reach tt, the true terminal node (the false terminal node is typically omitted). There is an equivalence between $f(a_1, ..., a_n) = true$ and the existence of a path from the root node to the tt whose arcs are labeled $a_1, ..., a_n$.

The reduction of an MDD is an important operation that may reduce the MDD size by an exponential factor. It consists in removing nodes that have no successor and merging equivalent nodes, i.e. nodes having the same set of neighbors associated with the same labels. This means that only nodes of the same layer can be merged.

MDD of a constraint. Let C be a constraint defined on $X(C)$. The MDD associated with C, denoted by $MDD(C)$, is an MDD modeling the set of tuples of C. More precisely, $MDD(C)$ is defined on $X(C)$, such that the arc labels of the layer of the variable x correspond to values of x, and a path of $MDD(C)$ (that is a path from the root node to the tt node) where a_i is the label of layer i corresponds to a tuple $(a_1, ..., a_n)$ on $X(C)$.

Operators We reproduce here the description of the generic Function APPLY [9,10] because we will see that the inclusion can be easily modelled thanks to it[1]. From the MDDs mdd_1 and mdd_2 it computes $mdd_r = mdd_1 \oplus mdd_2$, where $\oplus$ is union, intersection, difference, symmetric difference, complementary of union and complementary of intersection. Function APPLY is mainly based on the possible combinations of labeled arcs. It proceeds by associating nodes of the two MDDs operands. Each node x of the resulting MDD is associated with a node x_1 of the first MDD and a node x_2 of the second MDD represented by a pair (x_1, x_2). First, the root is created from the two roots. Then, the layers are successively built. From the nodes of layer $i-1$ the nodes of layer i are built as follows. For each node $x = (x_1, x_2)$ of layer $i-1$, the arcs outgoing from nodes x_1 and x_2 and labeled by the same value v are considered. We recall that there is only one arc leaving a node x with a given label. Thus, there are four possibilities depending on whether there are y_1 and y_2 such that (x_1, v, y_1) and (x_2, v, y_2) exist or not. The action that is performed for each of these possibilities will define the operation performed for the given layer. For instance, a union is defined by creating a node $y = (y_1, y_2)$ and an arc (x, v, y) each time one of the arcs (x_1, v, y_1) or (x_2, v, y_2) exists. An intersection is defined by creating a node $y = (y_1, y_2)$ and an arc (x, v, y) when both arcs (x_1, v, y_1) and (x_2, v, y_2) exist. Thus, these operations can be simply defined by expressing the condition for creating a node and an arc.

Function APPLY, given in Algorithm 1 takes as parameters the two MDDs, two arrays *op*, V having as many elements as layers, and *typeOp* the operation type (i.e. intersection, union...). For each layer i, $op[i]$ contains 4 entries, each one representing the fact that we create an arc or not for a combination of arc existence in the two MDDs and $V[i]$ represents the set of values needed by the complementary set. If it is equal to nil then $V[i]$ will be equal to the union of the values of the neighbors of the considered nodes. At the end the resulting MDD is reduced by calling pREDUCE algorithm [9].

The values of $op[i]$ defining the binary operations are defined as follows for the different combinations:

[1] Unlike Perez and Régin [9], the complementary of an MDD M is computed by making the difference between the universal MDD and M. This avoids the need of a dedicated algorithm.

	op[0] $\neg a1 \wedge \neg a2$		op[1] $\neg a1 \wedge a2$		op[2] $a1 \wedge \neg a2$		op[3] $a1 \wedge a2$	
layer	[1..r-1]	r	[1..r-1]	r	[1..r-1]	r	[1..r-1]	r
$A \cap B$	F	F	F	F	F	F	T	T
$A \cup B$	F	F	T	T	T	T	T	T
$A - B$	F	F	F	F	T	T	T	F

Algorithm 1 Generic Apply Function.

```
APPLY(mdd_1, mdd_2, op, V, typeOp): (MDD, bool)
    // L[i] is the set of nodes in layer i.
    root ← createNode(root(mdd_1), root(mdd_2))
    L[0] ← {root}
    for each i ∈ 1..r do
        L[i] ← ∅
        for each node x ∈ L[i − 1] do
            get x_1 and x_2 from x = (x_1, x_2)
            if V[i] = nil then V[i] ← values(ω+(x_1) ∪ ω+(x_2))
            for each v ∈ V[i] do
                if ∄(x_1, v, y_1) ∈ ω+(x_1) then
                    if ∄(x_2, v, y_2) ∈ ω+(x_2) ∧ op[0] then createArc(L, i, x, v, w[i])
                    if ∃(x_2, v, y_2) ∈ ω+(x_2) ∧ op[1] then addArcAndNode(L, i, x, v, nil, y_2)
                else
                    if ∄(x_2, v, y_2) ∈ ω+(x_2) ∧ op[2] then addArcAndNode(L, i, x, v, y_1, nil)
                    if ∃(x_2, v, y_2) ∈ ω+(x_2) ∧ op[3] then addArcAndNode(L, i, x, v, y_1, y_2)
                if typeOp = Inclusion then
                    if ∃(x_1, v, y_1) ∈ ω+(x_1) ∧ ∄(x_2, v, y_2) ∈ ω+(x_2) then return (nil, false)
    if typeOp = Inclusion then return (nil, true)
    merge all nodes of L[r] into t
    pReduce(L)
    return (root, true)

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(L, i, x, v, y)
```

3 Checking constraint satisfaction

A first solution to test whether a set of solutions S satisfies a constraint C is to represent S by an MDD, denoted $MDD(S)$, then use $MDD(C)$ the MDD of the constraint C and calculate the intersection between $MDD(S)$ and $MDD(C)$. If this intersection is equal to $MDD(S)$ then this means that all solutions of S satisfy the constraint C. The proof of the soundness of this approach is quite immediate : an MDD is a set of solutions, so if the intersection does not modify $MDD(S)$ then it means that any solution of $MDD(S)$ is also a solution of $MDD(C)$ and therefore this solution satisfies the constraint.

This approach is not particularly efficient, because it systematically requires the intermediate calculation of an intersection between MDDs. However, we

are not interested in this intersection[2]. What matters is to know whether this intersection is similar to the initial MDD. We can reduce what we are trying to do in a single step: we check a relation of inclusion. Indeed, answering the question $MDD(S) \cap MDD(C) = MDD(S)$? is equivalent to answering the following question: $MDD(S) \subseteq MDD(C)$? As this operator does not exist in the literature, we propose to create it.

3.1 Operator of Inclusion

The inclusion operator between MDDs is easily done using the generic function APPLY. Let's consider that we want to know if MDD_1 is included in MDD_2. We use the same rules as the intersection operator with a notable exception: if an edge a is in MDD_1 but not in MDD_2, then we end the algorithm by returning false. In this case there is at least one solution in MDD_1 which is not in MDD_2 so MDD_1 cannot be included in MDD_2. For the three other cases we can easily find those of the intersection. Using the terminology of the preliminaries, we have clearly: $\neg a_1$ implies that no arc is created and $a_1 \wedge a_2$ implies an arc creation, since the solutions are common.

We notice that it is not necessary to keep in memory the MDD that is built. Indeed, we just need to know if we can create each new level. To do this, only the last level that has just been built is useful and must be kept in memory, the others being no longer useful can be destroyed. The reduction of the built MDD is no longer useful either since we are only interested in the ability to build an MDD from the root to tt. Function APPLY must therefore return true instead of performing the reduction at the end. This allows us to save time compared to the previous method.

4 Inferring parameters of global constraints

The inclusion operator allows to answer in an efficient way to the question of the satisfaction of a constraint by a set of solutions S. However, in practice, the question that is often asked is more general: given a global constraint C involving a set of parameters, for which parameters S satisfies this constraint?

Let us consider P a set of parameters and $C(P)$ a constraint defined using these parameters. Formally, we can present the problem in the following way: What are the sets P such that $\forall s \in S$, s satisfies $C(P)$?

A first way to proceed is to check for each set of parameters P if we have $MDD_S \subseteq MDD_{C(P)}$.

We propose to add additional information, called properties, to each node of an MDD. This idea has similarities with the scheme introduced by J. Hooker et al.[6]. This information is used to memorize the valid parameters from the root to the node in relation to the constraint under consideration. When we

[2] We could also perform the intersection between $MDD(S)$ and the negation of $MDD(C)$ and check whether it is empty or not. However the computation of the negation is required so it does not improve the classical intersection.

will reach tt, we will know the parameters that are checked by all the solutions. In addition, retaining all the parameter sets is superfluous and we can be satisfied with retaining the more restrictive parameters. Other acceptable parameters may be derived from these restrictive parameters. For example, if the constraint SEQUENCE$(X, V, q = 3, l = 1, u = 2)$ is satisfied then the constraints SEQUENCE$(X, V, q = 3, l = 0, u = 2)$, SEQUENCE$(X, V, q = 3, l = 1, u = 3,)$ and SEQUENCE$(X, V, q = 3, l = 0, u = 3)$ are also satisfied. So, the more restrictive parameters are $(q = 3, l = 1, u = 2)$.

We present on an example the ideas of our algorithm. We will use the binary representation of a sequence constraint. Indeed, for a sequence constraint, we can abstract any X and V into a binary problem with $V = \{1\}$. If $x_i = 1$ then it means that x_i takes its value in V ($x_i \in V$) otherwise we have x_i=0. So we are in the presence of only binary variable and we are looking for the parameter values (q, u, l) which are satisfied by S.

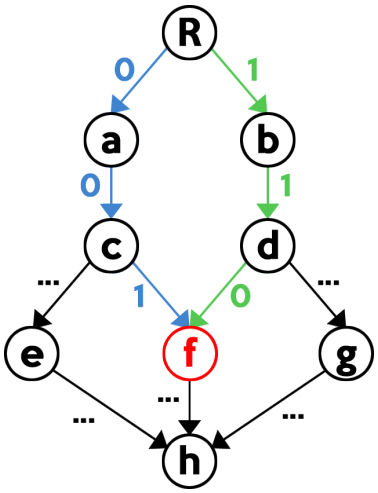

Fig. 1: Sequences for $MDD(S)$

Consider the red node f in $MDD(S)$ (Fig. 1). There are two paths to access this node from the root: take 0, 0 then 1 (which corresponds to the blue path -left- on the diagram) or take 1, 1 then 0 (which corresponds to the path in green -right-). In total, there are 5 sub-MDDs (i.e. smaller MDDs contained in the main MDD) having this red node f as the terminal node: 1 having as starting point the general root ({0-0-1, 1-1-0}), then 2 having as root the two nodes of the first layer ({0-1}, {1-0}) and finally 2 having as root the two nodes of the second layer ({1}, {0}). Strictly speaking, there are actually 6 sub-MDDs, since the MDD consisting only of the red node f exists.

Properties (i.e. satisfied sequences) are added to nodes:

- **Node** R**.** This node contains only the basic information, i.e. $(q = 0, l = 0, u = 0)$, since the only way to reach this node is to start from it and take no edge. This is the basic case, viable for all nodes.
- **Node** a **(blue).** In addition to the basic case, it is possible to reach this node starting from the root and taking the value 0. As we take 0 times the value 1, the satisfied SEQUENCE is $(q = 1, l = 0, u = 0)$.

- **Node b (green).** Same as for the previous node, except that we take once the value 1. The satisfied SEQUENCE is therefore $(q = 1, l = 1, u = 1)$.
- **Node c (blue).** We retrieve the information from the parent node (there is only one here). So we have $(q = 0, l = 0, u = 0)$ and $(q = 1, l = 0, u = 0)$. There is only one edge that can be traversed, with a value of 0. If we add 0 to the preceding satisfied sequences, we obtain $(q = 1, l = 0, u = 0)$ and $(q = 2, l = 0, u = 0)$. We retain these sequences.
- **Node d (green).** By the same reasoning, we obtain $(q = 1, l = 1, u = 1)$ and $(q = 2, l = 2, u = 2)$.
- **Node f (red).** We start by looking at parent Node c (blue). By adding the fact that we can reach the red node f by taking the value 1, we have $(q = 1, l = 0, u = 1)$, $(q = 2, l = 0, u = 1)$ and $(q = 3, l = 1, u = 1)$. In the same way, looking at the side of parent Node d (green), we obtain $(q = 1, l = 0, u = 1)$, $(q = 2, l = 1, u = 2)$ and $(q = 3, l = 2, u = 2)$. We notice that SEQUENCE constraints of size 2 and size 3 are not compatible. In this case, the union of the two SEQUENCE constraints is performed (since both are satisfied). We thus obtain $(q = 1, l = 0, u = 1), (q = 2, l = 0, u = 2)$ and $(q = 3, l = 1, u = 2)$. We can then check that for each path leading to the red node f, we take between 0 and 2 times the value 1 for a path of size 2, and between 1 and 2 times the value 1 for a path of size 3.

Fig.2 shows a slightly more complete example.

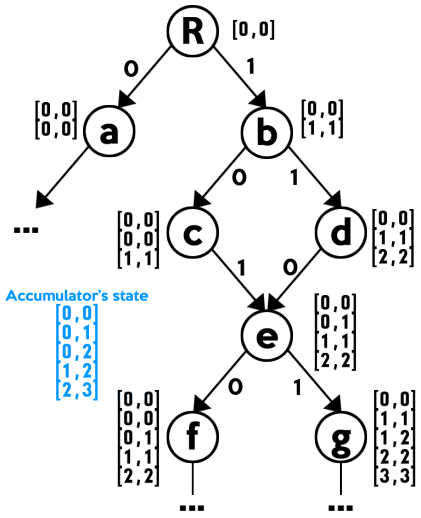

Fig. 2: Satisfied sequence constraints for each node. Value q corresponds to the index in the array associated with a node.

However, one thing is noticeable: it is possible to lose information (Fig. 3). For example, the node b contains the information $(q = 0, l = 0, u = 0)$, $(q =$

$1, l = 0, u = 0)$ and $(q = 2, l = 1, u = 1)$, but we can see that for paths of size 1 belonging to the MDD it is possible to take between 0 and 1 times the value 1. To solve this problem, we make the union of all the nodes of a layer (for each layer) that we store in an accumulator. This operation can be performed at the same time as the information is constructed.

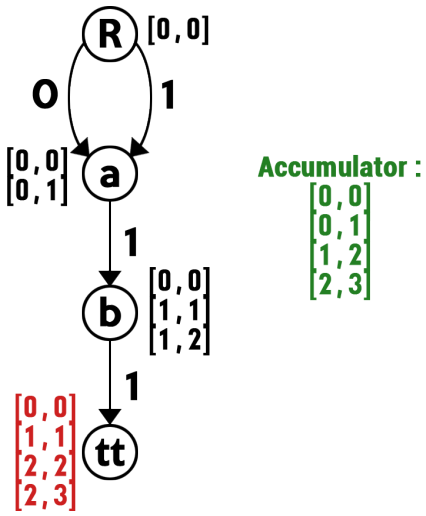

Fig. 3: Simple example to show information loss. Value q correspond to the index in the array associated with a node.

We remind that it is not necessary to retain *all* the information as represented on the diagram: we can simply keep the last two layers since the construction is done in a sliding way.

4.1 Implementation

The information is represented in the form of a property associated with each node. Each constraint will define its own property. The propagation of information from the MDD nodes is performed using a breadth-first approach, because we need all parent nodes to be correctly defined before propagating to the children. A depth-first approach would be strictly speaking impossible because all children would have to be re-explored each time the parents are updated (which is highly inefficient). During propagation, one looks to see if the child is already carrying a property or not. Two cases are possible.

– The child's property is already defined. In this case, the information already present must be merged with the new information provided by the parent. Function MERGEWITHPROPERTY of a property is in charge of this.

– The child property is not defined. In this case we simply create new information based only on the parents' information. Function createProperty of the property performs this operation.

Algorithm 2 is a possible implementation of this mechanism. The important Functions are mergeWithProperty and createProperty. They depend on the type of constraint that the property represents, so it is difficult to define a general way to represent the information. Technically, Function createProperty is quite simple. We believe that the real difficulty lies in the definition of Function mergeWithProperty, because the information must be complete and valid, i.e. it must represent the state of the constraint in a correct way for the node that contains it at any moment of the propagation.

Algorithm 2 Propagation

```
propagatePropertyMDD, property
    MDD.root.addProperty(property)
    for each layer L in MDD do
        for each node in L do
            transferProperty(node)
    return MDD.tt.getProperty().getResult()
transferProperty(node) for each (label, child) in node.children do
    if child.hasProperty() then
        child.getProperty().mergeWithProperty(node.getProperty(), label)
    else child.addProperty(node.getProperty().createProperty(label))
```

Time complexity: in O((|A| − |N|)× O(mergeWithProperty) + |N|× O(createProperty)), where |A| is the number of edges and |N| the number of nodes. Function createProperty is called only once for each node, that is |N| times globally. Function mergeWithProperty is globally called |A| − |N| times. O(mergeWithProperty) and O(createProperty) are the time complexity of the functions mergeWithProperty and createProperty. As these functions depend on the constraint and the implementation, we can't give any further details.

Space complexity. We retain (i+1) information for each node of layer i. The space complexity thus depends both on the layer where we are located (the last two layers in reality) and on the number of nodes present in the layer. As it is not possible to predict the number of nodes in a layer for any MDD, the complexity remains rather vague.
Let L_i be the number of nodes in the layer i and L the number of layers. The space complexity is in: O(Max$(i \times |L_i| + (i+1) \times |L_{i+1}|)$), $\forall i$ s.t. $0 \leq i < L$

4.2 Properties definitions

We provide the definition of properties for sum ($\sum_{x \in X} x = [min, max]$), sequence, and cardinality constraints as examples. These constraints have been chosen in relation to the problems we are working on, and not in relation to the difficulty of implementation. These are also quite common constraints.

Sum Constraint. The sum property is quite simple to model (Algorithm 3). As the result is an interval, we just need to use a pair (min, max) to represent the data. Switching from a parent to a child is simply the addition of an interval with an integer (the value of the arc), and the merge operation is a simple union between two intervals. Each node contains the minimum and maximum value that can be obtained by a path from the root to that node. The end node tt therefore contains the minimum and maximum value that can be obtained by taking any path through the MDD.

Algorithm 3 Sum Property

```
createProperty(label)
    (min, max) ← (this_min + label, this_max + label)
    return (min, max)
mergeWithProperty(property, label)
    this_min ← min(this_min, property_min + label)
    this_max ← max(this_max, property_max + label)
getResult()
    return this
```

Cardinality Constraint We represent the property of a global cardinality constraint as a *values* matrix of size $|V| \times 2$. Each value considered in the constraint is associated with a pair (min, max) representing the minimum and maximum number of times the value is taken. Moving from a parent to a child through an arc labeled by a, amounts to incrementing by 1 the number of times the value a is taken, which is a simple addition operation on the intervals. On the other hand, performing the merge amounts, as for the sum, to performing a union operation. Algorithm 4 is a possible implementation.

Sequence Constraint. The sequence property is the most complex of the three (Algorithm 5). In itself, Function createProperty and mergeWithProperty correspond, as for the cardinality constraint, to perform respectively an addition on intervals and a union operation. The difference comes from the fact that, for the sequence, it is necessary to have separate information, represented here by *accumulator*. This peculiarity comes from the sliding and global aspect of the sequence constraint: each node contains information about a local sequence, i.e. sequences that contain it as the final value. However, it is possible

Algorithm 4 Cardinality Property

```
createProperty(label)
    values ← ∅
    for each value v in the GCC do
        values[v]_min ← this.values[v]_min
        values[v]_max ← this.values[v]_max
    if label ∈ values then
        values[label]_min ← values[label]_min + 1
        values[label]_max ← values[label]_max + 1
    property.values ← values
    return property
mergeWithProperty(property, label)
    for each value v in the GCC do
        add ← 0
        if v = label then
            add ← 1
        values[v]_min ← min(property.values[v]_min + add, values[v]_min)
        values[v]_max ← min(property.values[v]_max + add, values[v]_max)
getResult
    return this
```

Algorithm 5 Sequence Property

```
createProperty(label)
    values ← ∅
    values[0] ← (0, 0)
    add ← (label is in the sequence values)? 1 : 0
    for each i from 1 to depth + 1 do
        values[i]_min ← this.values[i − 1]_min + add
        values[i]_max ← this.values[i − 1]_max + add
    property.values ← values
    property.depth ← depth + 1
    accumulate(property)
    return property
mergeWithProperty(property, label)
    add ← (label is in the sequence values)? 1 : 0
    for each i from 1 to depth do
        values[i]_min ← min(property.values[i − 1]_min + add, values[i]_min)
        values[i]_max ← max(property.values[i − 1]_max + add, values[i]_max)
    accumulate(this)
accumulate(property)
    for each i from 1 to property.depth do
        accumulator[i]_min ← min(property.values[i]_min, accumulator[i]_min)
        accumulator[i]_max ← max(property.values[i]_max, accumulator[i]_max)
getResult()
    return accumulator
```

in this case to lose information - as shown in Fig.3. This accumulator can be implemented in different ways: either by building it on the fly (memory gain but time loss), in which case it is not necessary to retain the information on more than two layers (the last two in a sliding manner), or by building it once the propagation of the property is completed, but all the information of all the nodes must be in memory (time gain but memory loss). Here, the accumulator is built on the fly.

5 Experiments

5.1 Testing environment

The algorithms have been implemented in Java 12. The experiments were performed on a Windows 10 machine using a Ryzen 2600 AMD CPU and 32 GB of RAM for the car sequencing problem and on a machine having four E7- 4870 Intel processors, each having 10 cores with 256 GB of memory and running under Scientific Linux for the nurse rostering problem.

The different tests comparing the methods presented in this paper were performed using solutions from instances of Car Sequencing and Nurse Rostering (represented as a MDD). The details of this instances can be found in appendix. However, we give some important information:

Car Sequencing The Car Sequencing MDD contains 25942 nodes, 53985 arcs and represents 2.6×10^{14} solutions. There are two options with capacity 1/2 and 2/3, 4 car classes (configurations) and a total of 100 cars.

Nurse Rostering The Nurse Rostering MDD contains 128325 nodes, 220600 arcs and represents 1.2×10^{28} solutions. There are 6 nurses, 28 days and 3 shifts. The scheduling has some predefined data : 1 means that the nurse is working that day, - means that the nurse is not working that day and 0 means that it is yet to be determined. Each day, each shift has a minimum number required of nurse working that shift (we can have more, but never less). We have various constraints, such that a nurse cannot work more than 7 days straight and must have at least 2 free days in a row in a 2-weeks window.

Testing each raw solution individually requires testing 2.6×10^{14} and 1.2×10^{28} solutions (respectively for Car Sequencing and Nurse Rostering problems). Doing such benchmark in a reasonable amount of time is out of the question.

5.2 Comparison between inclusion and intersection based inclusion

We will compare the two methods to compute the inclusion presented in this paper. We do not take into account the time and space needed to create and store the MDDs representing the constraints - we only focus on time and space needed to compute the inclusion between two MDDs.

We can see from results in Table 1 that the inclusion method is better in every way than the intersection based inclusion. We observe improvements by a

	Car Sequencing				Nurse Rostering			
Constraints	Intersection		Inclusion		Intersection		Inclusion	
	time	memory	time	memory	time	memory	time	memory
GCC	95	48	50	29	7 052	1 716	2 820	1 186
Sum	85	47	46	30	-	-	-	-
Sequence	187	98	38	27	15 320	3 244	5 039	1 402

Table 1: Intersection vs Inclusion (Average time in ms, memory in MB).

factor between 2 and 3 in time and between 1.5 and 3 in memory for GCC and Sum constraints, and almost 5 in time for the sequence constraint.
This result was clearly expected, as the inclusion operation is at worst an intersection, without the reduce and compare part of the intersection based inclusion.

In the Car Sequencing problem, the GCC constraint expresses the number of time a car has to be produced and the Sequence constraint represents the maximum capacity of each option (that is, for any subsequence of q consecutive cars, the maximum number of cars that can have this option). In the Nurse Rostering problem, the GCC constraint expresses the demands for a shift (that is the minimum number of nurses required for a given shift) and the Sequence constraint expresses the fact that a nurse must have at least a certain number of breaks in a sliding time window.
The Sum constraint does not have any powerful meaning in these problems, but we decided to test it on at least one problem. We did not test the sum constraint for the Nurse Rostering problem, hence the dash symbol.

5.3 Learning parameters of a global constraint

We compare the different methods (inclusion, intersection based inclusion and properties) in order to determine the parameters of a global constraint. The tested constraints are the sequence, sum and GCC constraints. To determine the parameters with the inclusion and intersection based methods, a dichotomous search on the parameters is performed. For example, if the constraint SUM(a, b) is satisfied, we test if the constraint SUM$(a, b/2)$ is satisfied: if it is, we test SUM$(a, b/4)$, otherwise we test SUM$(a, b * 3/4)$. We modify the parameters one by one until they are fixed. However, for both methods, we do not compute the sequence parameters for all possible values of q because it would take too much time (we stop at $q = 11$). The time and memory used when building the MDDs are integrated into the results.

As we can see from these results, using properties to compute the parameters of a global constraint is better than performing successive inclusion checks by at least a factor 65 in time (1 391ms vs 22ms for the sum constraint for the Car Sequencing, in Table 2) and at most a factor 145 in time (283 994ms vs 1 952ms for the GCC constraint for the Nurse Rostering, in Table 3). For the sequence constraint, we reach a factor of 75 with the properties even if we do not

	GCC		Sum		Sequence	
Methods	Time	Memory	Time	Memory	Time	Memory
Intersection	6 080	3 240	2 137	1 328	11 306	6 493
Inclusion	3 371	2 204	1 391	1 021	6 114	3 867
Properties	48	10	22	4	82	37

Table 2: Car Sequencing Problem (Time in ms, memory in MB).

	GCC		Sequence	
Methods	Time	Memory	Time	Memory
Intersection	663 647	170 281	704 405	137 625
Inclusion	283 994	137 625	235 996	65 768
Properties	1 952	348	5 677	1 436

Table 3: Nurse Rostering Problem (Average time in ms, memory in MB).

compute all sequences with the other methods. Furthermore, a very big part of the process is to build all the MDDs of the constraints, resulting in an increase in both time and memory consumption, as expected. Once again we find the factor 2 that we had in our previous comparison between the two methods of inclusion.

5.4 Conclusion

This article sheds light on a new aspect of the interest of using MDDs in the context of constraint programming. We have introduced a new inclusion operator that allows to answer the question of the satisfaction of a constraint more efficiently than by using the classical sequence of operations on MDDs (intersection, reduction, comparison). In addition, we have shown that adding properties to the nodes of a MDD allowing to represent locally the state of a constraint is a very efficient way to obtain the parameters of a global constraint (we presented GCC, sum and sequence), provided that this constraint can be formalized as a node property. This method is much more advantageous than a succession of inclusion operations, both from a temporal and spatial point of view, because it does not require any constraint construction in the form of an MDD. The use of inclusion is nevertheless of interest when the constraint is particularly complex, unique, or very difficult to formalize in the form of a property. The question of the use of properties for other constraints (other than global) seems to be the next step in order to answer in more detail the problem of extracting the parameters of a constraint from a set of solutions.

5.5 Acknowledgments

This work has been supported by the French government, through the 3IA Côte d'Azur Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-19-P3IA-0002.

References

1. Beldiceanu, N., Contejean, E.: Introducing global constraints in CHIP. Journal of Mathematical and Computer Modelling **20**(12), 97–123 (1994)
2. Beldiceanu, N., Simonis, H.: A constraint seeker: Finding and ranking global constraints from examples. In: Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings. pp. 12–26 (2011)
3. Bergman, D., Ciré, A.A., van Hoeve, W., Hooker, J.N.: Decision Diagrams for Optimization. Artificial Intelligence: Foundations, Theory, and Algorithms, Springer (2016). https://doi.org/10.1007/978-3-319-42849-9, `https://doi.org/10.1007/978-3-319-42849-9`
4. Bessière, C., Coletta, R., Petit, T.: Learning implied global constraints. In: IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007. pp. 44–49 (2007)
5. Bryant, R.E.: Graph-based algorithms for boolean function manipulation **35**(8), 677–691 (1986)
6. Hoda, S., van Hoeve, W.J., Hooker, J.N.: A systematic approach to MDD-based constraint programming. In: CP. pp. 266–280 (2010)
7. Kam, T., Brayton, R.K.: Multi-valued decision diagrams. Tech. Rep. UCB/ERL M90/125, EECS Department, University of California, Berkeley (Dec 1990), `http://www2.eecs.berkeley.edu/Pubs/TechRpts/1990/1671.html`
8. Leo, K., Mears, C., Tack, G., de la Banda, M.G.: Globalizing constraint models. In: Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings. pp. 432–447 (2013)
9. Perez, G., Régin, J.C.: Efficient operations on MDDs for building constraint programming models. In: International Joint Conference on Artificial Intelligence, IJCAI-15. pp. 374–380. Argentina (2015)
10. Perez, G.: Decision Diagrams : Constraints and Algorithms. Ph.D. thesis, Université Nice Sophia Antipolis (2017)
11. Picard-Cantin, É., Bouchard, M., Quimper, C., Sweeney, J.: Learning parameters for the sequence constraint from solutions. In: Rueher, M. (ed.) Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9892, pp. 405–420. Springer (2016)
12. Picard-Cantin, É., Bouchard, M., Quimper, C., Sweeney, J.: Learning the parameters of global constraints using branch-and-bound. In: Beck, J.C. (ed.) Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10416, pp. 512–528. Springer (2017)
13. Srinivasan, A., Ham, T., Malik, S., Brayton, R.K.: Algorithms for discrete function manipulation. In: 1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers. pp. 92–95 (1990). https://doi.org/10.1109/ICCAD.1990.129849

5.6 Appendix

Car Sequencing The Car Sequencing MDD contains 25942 nodes, 53985 arcs and represents 2.6×10^{14} solutions. There are two options with capacity 1/2 and 2/3, 4 car classes (configurations) and a total of 100 cars, defined as follow :

Table 4: Input of the Car Sequencing Problem

Option	Capacity	Car Class			
Name	m/N	1	2	3	4
Option 1	1/2	0	0	1	1
Option 2	2/3	0	1	0	1
Number of cars		15	37	21	27

Nurse Rostering The Nurse Rostering MDD contains 128325 nodes, 220600 arcs and represents 1.2×10^{28} solutions. There are 6 nurses, 28 days and 3 shifts. The scheduling has some predefined data : 1 means that the nurse is working that day, - means that the nurse is not working that day and 0 means that it is yet to be determined. Each day, each shift has a minimum number required of nurse working that shift (we can have more, but never less). We have various constraints, such that a nurse can't work more than 7 days straight and must have at least 2 free days in a row in a 2-weeks window.

Table 5: Input of the Nurse Rostering Problem

Nurses	Week 1							Week 2						
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	-	1	1	1	-	-	1	1	-	1	1	1	1
2	-	-	-	-	-	-	-	-	0	0	0	0	0	0
3	0	0	0	0	-	0	0	0	0	-	0	0	-	-
4	0	0	0	-	0	-	-	-	-	-	-	-	-	-
5	0	0	0	-	0	1	0	0	0	0	0	0	0	0
6	0	0	0	1	0	-	0	0	0	0	0	0	0	0
Demands per shift														
Shift 1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Shift 2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Shift 3	1	1	1	1	1	0	0	1	1	1	1	1	0	0

Nurses	Week 3							Week 4						
	15	16	17	18	19	20	21	22	23	24	25	26	27	28
1	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	0	0	0	0	0	0	0	0	0	-	0	0	0	0
3	-	-	-	-	-	-	-	-	0	0	0	0	0	0
4	-	0	-	0	0	-	-	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	-	0	0	0	0
6	0	0	0	0	0	0	0	0	0	-	-	-	-	-
Demands per shift														
Shift 1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Shift 2	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Shift 3	1	1	0	1	1	0	0	1	1	1	1	1	0	0