



Efficient Operations Between MDDs and Constraints

Victor Jung^(✉) and Jean-Charles Régin^{ID}

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

Abstract. Many problems can be solved by performing operations between Multi-valued Decision Diagrams (MDDs), for example in music or text generation. Often these operations involve an MDD that represents the result of past operations and a new constraint. This approach is efficient, but it is very difficult to implement with some constraints such as ALLDIFFERENT or cardinality constraints because it is often impossible to represent them by an MDD because of their size (e.g. a permutation constraint involving n variables requires 2^n nodes).

In this paper, we propose to build on-the-fly MDDs of structured constraints as the operator needs them. For example, we show how to realise the intersection between an MDD and an ALLDIFFERENT constraint by never constructing more than the parts of the ALLDIFFERENT constraint that will be used to perform the intersection. In addition we show that we can anticipate some reductions (i.e. merge of MDD nodes) that normally occur after the end of the operation.

We prove that our method can be exponentially better than building the whole MDD beforehand and we present a direct application of our method to construct constraint MDDs without having to construct some intermediate states that will be removed by the reduction process.

At last, we give some experimental results confirming the gains of our approach in practice.

1 Introduction

Multi-valued and binary decision diagrams (MDDs/BDDs) took an important place in modern optimisation techniques. From the theory to the applications, MDDs have shown a large interest in operational research and optimisation [1, 2, 7, 10, 15, 18]. They offer a broad range of modeling and solving possibilities, from being a basic block of constraint solvers [6, 13], to the development of MDD-based solvers [3, 9, 19].

MDDs are a very efficient graph-based data structure to represent a set of solutions in a compressed way. The fundamental reasons of their use is their exponential compression power. A polynomial size MDD have the capacity to represent an exponential size set of tuples. For example in a music scheduling problem [16], an MDD having 14,000 nodes and 600,000 arcs stored 10^{90} meaningful tuples of size one hundred. Unlike trees, where each leaf represent a solution, an MDD can have much fewer nodes and arcs than solutions.

MDDs are also often reduced, that is a reduction operation is applied to them. The reduction operation of an MDD merges equivalent nodes until a fix point is reached. It may reduce the size of an MDD by an exponential factor.

Several other operators are available to combine MDDs without decompressing them. The most important are the intersection, which corresponds to a conjunction of two constraints, and the union, which corresponds to a disjunction of two constraints and the negation.

Some problems can be solved by a succession of operations applied on MDDs [8, 14, 16]. In other words, there is no search procedure that is used. To do so, the different constraints are represented by MDDs, and they are combined by applying operators between these different MDDs. In this way, all solutions can be computed at once. However, even if the final solution can fit into memory, it is possible that the memory explodes during the intermediate computation steps - worse, it is even quite frequent that the MDD of the constraints are themselves too big to fit in memory because they are exponential, as for most cardinality constraints (e.g. ALLDIFFERENT and global cardinality constraints). One solution to be able to represent such constraints is to relax them [4, 10]: we gain memory in exchange for the loss of information. Usually, a relaxed MDD is an MDD representing a super set of the solutions of an exact MDD. Preferably, these relaxed MDDs are smaller than their exact versions. In general in such techniques, the total size is a fixed given parameter, which has a strong impact on the quality of the relaxation. Even if this approach is efficient, it can still be unsatisfactory to a certain extent. The ideal would be to be able to perform the computations while remaining exact. This is what interests us in this article: to be exact. In particular, we are interested in being able to perform operations without having to represent the whole constraint's MDD in order to avoid the problem of the intermediate representation.

More precisely the question we consider in this paper is: how can we compute an operation between a given MDD and the MDD of a constraint without consuming too much memory? We need to answer this question even if the MDD of the constraint cannot fit into memory. A simple example is the intersection between an MDD involving fifty variables and the MDD representing an ALLDIFFERENT constraints on these variables. This latter MDD will have 2^{50} nodes (i.e. 1,000,000 Giga nodes) and so is too big to fit in memory.

The main ideas of this paper are to avoid building the MDD of the constraint before applying the operation and to anticipate the reduction of the obtained MDD.

The first idea can be implemented by using operators that proceed by layer [14] because we can avoid building in advance the MDD of the constraint. During the operation, if a node can hold all the information necessary to build its children in a way that satisfies the constraint, then we do not need to retain all previous nodes. Thus, we can perform the construction having only at most two layers in memory (the current layer and the next one being built). Furthermore, if an arc does not exist in the first MDD, then it does not need to exist in the

constraint's MDD: building the MDD during the operation allows us to have more gain by only representing what is necessary to be represented.

The second idea is based on the remark that a constraint is useless when we can make sure that it will always be satisfied. For instance if 3 variables remains and if we know that they have disjoint domains then an ALLDIFFERENT constraints between these variables is useless. Thus, we can avoid defining the constraint's MDD and we can immediately merge some nodes that would have been merged by the reduction process and so gaining some space in memory.

The advantage of this approach is that it can provably gain an exponential factor in space. It has also a direct application to construct constraint MDDs without having to construct all intermediate states. This allows either to build constraint MDDs that cannot be built otherwise, or to build them much faster.

We also show that processing several constraints simultaneously is not advantageous compared to doing the operations successively for each constraint. This is due to the lack of reduction which is normally performed after each operation and which can strongly reduce the resulting.

The paper is organised as follows. First, we enrich the classical internal data of an MDD with several notions: we introduce the notion of node states allowing to represent the information associated with the node with respect to a certain constraint, the notion of transition function $\delta_C(s, v)$ and the notion verification function $V_C(s)$ allowing respectively to make the state of a node evolve by performing a transition of value v and to verify if a state is satisfying the constraint (absence of violation). Then, we elaborate on the importance of performing merges to have some control over the growing behavior of some constraints, by giving for each constraint described in the article the conditions to perform a merge. In addition, we try to convey some intuition of the potential gain behind these merges, which might greatly depend on the constraint's parameters. Next, we give for each of these constraints a possible implementation of the state and functions $\delta_C(s, v)$ and $V_C(s)$, as well as a generic algorithm allowing to perform an on-the-fly intersection operation based on the notions described. We also give the size of the MDD representing each constraint. Afterwards, we prove that building the constraint's MDD on the fly can be exponentially better than building the whole MDD beforehand. Finally, we present a direct application to compute constraint MDDs and we give some experimental results, notably for the car sequencing problem and we study the generalisation of our method for a set of constraints. At last, we conclude.

2 Preliminaries

2.1 Constraint Programming

A finite constraint network \mathcal{N} is defined as a set of n variables $X = \{x_1, \dots, x_n\}$, a set of current domains $\mathcal{D} = \{D(x_1), \dots, D(x_n)\}$ 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 $\mathcal{D}_0 = \{D_0(x_1), \dots, D_0(x_n)\}$ 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 $X(C) = (x_{i_1}, \dots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D_0(x_{i_1}) \times \dots \times D_0(x_{i_r})$ that specifies the allowed combinations of values for the variables x_{i_1}, \dots, x_{i_r} . An element of $D_0(x_{i_1}) \times \dots \times D_0(x_{i_r})$ 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 occurrences of the value a in a tuple τ .

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

Definition 1. *Given X a set of variables and $[l, u]$ a range, the **sum constraint** ensures that the sum of all variables $x \in X$ is at least l and at most u .*

$\text{SUM}(l, u) = \{\tau \mid \tau \text{ is a tuple on } X(C) \text{ and } l \leq \sum_{i=0} \tau_i \leq u\}$.

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 ALLDIFFERENT constraint corresponds to a GCC in which every value can be taken at most once.

Definition 2. *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*

$\text{GCC}(X, l, u) = \{\tau \mid \tau \text{ is a tuple on } X(C) \text{ and } \forall a_i \in D(X(C)) : l_i \leq \#(a_i, \tau) \leq u_i\}$.

2.2 Multi-valued Decision Diagram

The decision diagrams considered in this paper are reduced, ordered multi-valued decision diagrams (MDD) [2, 12, 17], which are a generalisation 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 $f : \{0 \dots d - 1\}^n \rightarrow \text{true}, \text{false}$. 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 $f(a_1, \dots, a_n) = \text{true}$ and the existence of a path from the root node to the tt whose arcs are labeled a_1, \dots, 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 children associated with the same labels. This means that only nodes of the same layer can be merged.

3 Generalisation of the Construction Process

Perez and Régis [14] have explained how an MDD can be built directly from functions defining an automaton or a pattern. The general principle is to define states and to link them by a transition function, which can be defined globally or for each level. In order to be more general and less dependent on automata theory, we propose to generalize the previous concepts and to introduce a verification function. This is similar to what is done in [9, 11].

By doing so, the notion of *state*, the function of transition $\delta_C(s, v)$ as well as the verification function $V_C(s)$, form a lightweight and general scheme allowing an efficient on the fly construction of a constraint's MDD.

3.1 State, Transition and Verification

The notion of *state* holds some information in an MDD node about the constraint representation, giving it an actual meaning: for the SUM constraint, a state will hold the value of the sum for the current node. Given that piece of information, we will be able to build all the valid successors of a node.

In order to build the MDD layer by layer, i.e. to build the children of a node, we define a transition function on the nodes: this transition function takes into account the current state s of a node and the constraint C . Given a certain state, we need to build all successors of a node such that the state of each successor satisfies the constraint C .

Let $\delta_C(s, v)$ be the *transition* function that builds a new state from a state s and a label v and $V_C(s)$ the *verification* function that checks whether a state s satisfies the constraint or not.

Thanks to these notions we can define the following property:

Property 1. *Nodes of the same layer with the exact same state can be merged.*

Proof. The transition function δ_C takes into account the constraint C and the current state s to build the successors of a node. The constraint C being invariant during the construction process, it means that for a given layer if $s_1 = s_2$, then $\delta_C(s_1) = \delta_C(s_2)$. In an MDD, two nodes can be merged if they have the same successors: therefore, we can merge two nodes having the same state. \square

Immediate Merges. If during the construction of the MDD, we can clearly see that some nodes will be merged during the reduce operation, then we can immediately merge some states. However, merging states will result in the loss of precision concerning the information represented. For instance, if two nodes representing a sum s_1 and s_2 are merged, the result of this merge is the set $\{s_1, s_2\}$. The node holding this state could be considered to be in both state s_1 and s_2 at the same time. Now, let's imagine that we must have a sum satisfying the range $[10, 15]$, and that we have a merged state of $[14, 15]$ for a given node: can we add 1? The answer is yes when considering 14, and no when considering 15. This shows that, in order not to lose information about the constraint (i.e.

solutions), we have to be careful about what we merge - we cannot do it blindfolded. In a general way, a node holding a merged state would mean that the state of this node represents less information than the initial constraint (which is the case for the GCC constraint), or in extreme cases that the state of the node does not matter anymore (which is the case for the SUM constraint).

For each constraint described in this article, we will detail: the total number of states for the constraint, the conditions to perform a merge and its consequences for the size of the MDD. Please note that the impact of the merges on the size highly depends on the constraint and its parameters.

We will now present a possible implementation for the SUM, GCC, and ALLDIFFERENT constraints. Henceforth, we will refer to the transition function δ as CREATESTATE and the verification function V_C as ISVALID.

3.2 Sum

Representation. The SUM constraint is simply represented by the minimum min and maximum max value of the sum. For convenience purposes, we also add the minimum v_{min} and maximum v_{max} value of D , the union of the domains of the variables involved in the constraint.

The state is represented by a single integer sum (Fig. 1).

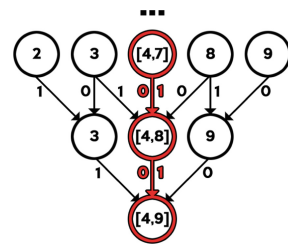


Fig. 1. Merged states in the SUM, with $min = 4$ and $max = 9$. The merged path (red) necessarily lead to tt, whatever the value taken.

Transition and Validity. Creating a state means to add the label's value to the current sum, and a transition is valid if the obtained sum belongs in the constraint's bounds (See Algorithm 1).

Number of States. The number of states at layer i is at most $i \times (v_{max} - v_{min} + 1)$. The total number of states in the MDD is therefore at most:

$$2 + \sum_{i=1}^{n-1} i \times (v_{max} - v_{min} + 1) = 2 + \frac{(n-1)^2 + (n-1)}{2} \times (v_{max} - v_{min} + 1)$$

This upper bound is achieved when the set of values D is an integer range. This number can also be bounded by the upper and lower bounds of the sum.

Merging Condition. Let s be the state representing a sum. If $min \leq s + (n - layer) \times v_{min}$ and $s + (n - layer) \times v_{max} \leq max$, then we are certain to satisfy the constraint no matter what values we assign next. Therefore, we can drop the state of the node. Table 1 gives some experimental results of immediate merges.

Algorithm 1. Sum State

```

CREATESTATE(constraint, state, label, layer, n): state
┌   nextState.sum ← state.sum + label;
└   return nextState;

ISVALID(constraint, state, label, layer, n): boolean
┌   minReach ← state.sum + label + (n - layer) × constraint.vmin;
┌   maxReach ← state.sum + label + (n - layer) × constraint.vmax;
┌   // We cannot reach the minimum sum or go below the maximum
┌   if (maxReach < constraint.min ∨ minReach > constraint.max) then
┌   └   return false;
└   return true;

```

Table 1. Impact of immediate merges on the number of nodes created and memory consumption for the SUM constraint

				With merges		Without merges	
min	max	n	V	#nodes	memory (MB)	#nodes	memory (MB)
20	200	50	10	6204	29	7914	35
124	480	200	14	82349	448	87047	472
500	1000	200	10	117864	456	131504	506
500	1000	200	20	161720	1192	168051	1242

3.3 GCC

Representation. To represent the GCC constraint, we need: the set V of constrained values, and the minimum lb_v and maximum ub_v occurrences of each specific value $v \in V$. To represent the state, we only need a array *count* that contains the number of occurrences of each value $v \in V$. We also define another variable named *minimum* that counts the minimum number of layers required to reach all the lower bounds of values.

Transition and Validity. Creating a new state means adding 1 to the counter of the value of the label we take if this value is constrained. A transition is valid if we can reach the lower bounds with the remaining layers, and if the added value of the label is not greater than its upper bound (always true if the value is not constrained, of course). Algorithm 2 is a possible implementation.

Notation 1

- n is number of variables and $layer$ the index of the current layer.
- $\forall v \in V: c_v$ is the number of times v is assigned, lb_v the lower bound of v and ub_v the upper bound of v .

Number of States. The number of states in a GCC is at most:

$$\prod_{v \in V} ub_v$$

because this is the number of count tuples that can be represented by the numeral system defined by the GCC.

Merging Condition

Property 2. We can remove the count c_v of value v from the state iff:

$$(n - layer \leq ub_v - c_v) \wedge (lb_v \leq c_v + \max(0, (n - layer) - \sum_{i \neq v} (u_i - c_i)))$$

If the number of variables left to assign is less than the number of times we can assign the value v , and if the lower bound lb_v is reached, then it means that, no matter how many times we assign the value v to the future variables, we are certain to be in the range $[lb_v, ub_v]$.

Example: Let the bound $[lb_v, ub_v] = [10, 20]$, $c_v = 10$ and $n - layer = 10$. Then, we can have a merged state $[c_v, c_v + i] = [10, 10 + i]$ up to $i = n - layer = 10$. In that case, the value v can be ignored (i.e. deleted) by the state because, no matter what choices we make, we are assured to satisfy the constraint: there is therefore no need to take into account v .

Table 2 gives some experimental results of immediate merges.

Table 2. Impact of immediate merges on the number of nodes created and memory consumption for the GCC constraint

With merges		Without merges	
#nodes	memory (MB)	#nodes	memory (MB)
405081	3308	543196	4300
58385	450	100341	715
5064	44	40558	266
430076	3584	470801	3849

3.4 AllDifferent

Representation. The ALLDIFFERENT constraint is simply a GCC constraint for which the set of values V contains all values and each value can only be assigned once. We represent the state by the set of previously assigned values.

Transition and Validity. A transition is valid if the *label* is not already assigned. To create a new state, we simply copy the current state (i.e. the set of assigned values) and add to it the new *label* (See Algorithm 3).

Algorithm 2. GCC State

```

CREATESTATE(constraint, state, label, layer, n): state
  count ← ∅ ;
  min ← state.minimum;
  potential ← n − layer − 1;
  for each value v ∈ state.count do
    if state.count[v] < constraint.min[v] then count[v] ← state.count[v]
    if state.count[v] + potential > constraint.max[v] then
      | count[v] ← state.count[v]
  if label ∈ count then
    if state.count[label] < constraint.min[label] then min ← min − 1;
    // If we are sure to satisfy the constraint for the label
    if (constraint.min[label] ≤ count[label] + 1) ∧ (count[label] + n − layer ≤
      constraint.max[label]) then count.remove(label);
    else count[label] ← count[label] + 1
  nextState.count ← count;
  nextState.minimum ← min;
  return nextState;

ISVALID(constraint, state, label, layer, n): boolean
  potential ← n − layer − 1;
  min ← state.minimum;
  if label ∉ state.count.values then return min ≤ potential;
  value ← state.count[label];
  if value < constraint.min[label] then min ← min − 1;
  return (min ≤ potential) ∧ (value + 1 ≤ constraint.max[label]);

```

Number of States. The number of states in the i^{th} layer for a given set of constrained values V is $\binom{|V|}{i}$. Therefore, the total number of nodes in the MDD is:

$$2^n \leq \sum_{i=0}^n \binom{|V|}{i} \leq 2^{|V|}$$

Merging Condition. The ALLDIFFERENT constraint can be seen as a GCC constraint where all values $v \in V$ are associated with the range $[0, 1]$. Thus, the merging conditions for the ALLDIFFERENT constraint is the same as the GCC under the described parameters: it means that we can only merge during the last layer, which is negligible.

3.5 Generic Constraint Intersection Function

This function is a possible implementation of the generic constraint intersection function. It is based on Functions defined in [14]. It shows that we do not need to store the full MDD_c and that we build the next layer only knowing the current layer.

Algorithm 3. ALLDIFFERENT State

```

CREATESTATE(constraint, state, label, layer, n): state
  values  $\leftarrow \emptyset$ ;
  for each  $v \in state.values$  do values.APPEND( $v$ );
  if  $label \in constraint.values$  then values.APPEND( $label$ );
  nextState.values  $\leftarrow values$ ;
  return nextState;

ISVALID(constraint, state, label, layer, n): boolean
  return  $label \notin state.values$ 

```

Algorithm 4. Generic Constraint Intersection Function.

```

APPLYINTER(mdd1, constraint, rootc): MDD
  // When creating a node, we associate it with one node from each MDD
  root  $\leftarrow$  CREATENODE(root(mdd1), rootc) ;
  L[0]  $\leftarrow$  {root} // L[i] is the set of nodes in layer i. ;
  C[0]  $\leftarrow$  {rootc};
  for each  $i \in 1..n$  do
    L[i]  $\leftarrow \emptyset$ ; C[i]  $\leftarrow \emptyset$  ;
    for each node  $x \in L[i - 1]$  do
      get  $x_1$  and  $x_2$  from  $x = (x_1, x_2)$  ;
      for each  $v \in \text{CHILD OF}(x_1)$  do
        if  $v \notin \text{CHILD OF}(x_2) \wedge \text{ISVALID}(constraint, x_2.state, v, i - 1, r)$ 
        then
           $y_2 \leftarrow$  CREATENODE();
           $y_2.state \leftarrow$  CREATESTATE(constraint, x_2.state, v, i - 1, r);
          ADDCHILD( $x_2, v, y_2$ );
          C[i].APPEND( $y_2$ );
          // Add the arc between  $x$  and the node defined by  $y = (y_1, y_2)$ 
          // The node  $y$  will be added to the MDD if it is not yet in it.
          ADDARCANDNODE(L, i, x, v,  $y_1, y_2$ ) ;
    DESTROY(C[i - 1]) // Remove previous constraint layer from memory ;
  merge all nodes of L[n] into  $t$ ;
  PREDUCE(L) ;
  return root ;

```

4 Exponential Gain

Theorem 1. *Building the MDD of a constraint on the fly can be exponentially better in terms of space and time than building the whole MDD beforehand.*

Proof. We show how to perform an intersection between MDD_x , an MDD, and MDD_{AD} the MDD of an ALLDIFFERENT constraint. Let x be a number of sets, MDD_x is built as follows (See Fig. 2 for $x = 3$ and $|X| = 3$):

- **Step 1** - Generate $MDD_U(X)$ a universal MDD with domain X . This means that the variables can take any value in X .

- **Step 2** - Copy the MDD from step 1 for x sets having a cardinality equal to $|X|$ and make the union of them. We denote by MDD_V the obtained MDD.
- **Step 3** - Copy x times MDD_V , and concatenate them.

The size of MDD_{AD} is exponential (i.e. at least 2^n). So, when n is large, it is not possible to build it. However, the size of $MDD_x \cap MDD_{AD}$ is exponentially smaller than the size of MDD_{AD} , and our method is able to compute this intersection as shown by the following process:

1. The number of nodes in the MDD_{AD} involving all the variables is $2^{|X| \times x}$, being $(2^{|X|})^x$.
2. MDD_V is the union of x universal MDDs. The intersection between a universal MDD and MDD_C (i.e. the MDD of a constraint C) is MDD_C . So, for any set X , the intersection between $MDD_U(X)$ and MDD_{AD} is equal to MDD_{AD} which has $2^{|X|}$ nodes.
3. We can simplify by stating that each $MDD_U(X)$ is an arc. The shape of our MDD is therefore the one of a universal MDD. Thus, the same observation that in 2. applies: the simplified MDD intersecting with the ALLDIFFERENT constraint is the MDD of the ALLDIFFERENT constraint. It is denoted by $metaMDD_{AD}$.
4. If we know the number of arcs in our $metaMDD_{AD}$, we can deduce the number of nodes created during the intersection.
5. The layer i of $metaMDD_{AD}$ contains $\binom{x}{i}$ nodes.
6. Each node in the layer i has $(x - i)$ out-going arcs (because we already chose i values of the x possible).
7. By combining (5) and (6), the total number of arcs in our $metaMDD_{AD}$ is $\sum_{i=0}^x \binom{x}{i} \times (x - i) = x \times 2^{x-1}$
8. By combining (2) and (7), the total number of nodes in our $metaMDD_{AD}$ is $x \times 2^{x-1} \times 2^{|X|} = x \times 2^{|X|+x-1}$, because each arc of $metaMDD_{AD}$ is an $MDD_{AD}(X)$ involving $|X|$ variables.
9. The difference between (8) $x \times 2^{|X|+x-1}$ and (1) $(2^{|X|})^x$ is exponential. □

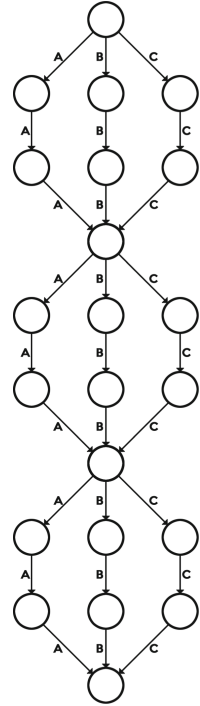


Fig. 2. Final step. For convenience purposes, we represent only one arc for a whole set of values.

We perform some benchmarks to experimentally confirm this gain.

4.1 Building the AllDifferent's MDD

Table 3 shows that the ALLDIFFERENT constraint quickly becomes impossible to construct: after barely 24 values it is impossible to represent the constraint.

Table 3. Construction of the ALLDIFFERENT MDD with size variation

$ V $	Memory (MB)	Time (s)	Layer
20	8840	12.363	20
21	18789	27.476	21
22	39675	62.850	22
23	83619	134.696	23
24	Out of memory (≥ 100 GB)	160.894	13

4.2 Performing the Construction on the Fly

The results of Table 4 show that by constructing the ALLDIFFERENT’s MDD on the fly, it is possible to compute intersections with a lot of values (here between $|V| = 25$ and $|V| = 100$) very efficiently. We notice that, when we increase the number of sets, the intersection becomes more and more difficult to compute: this testifies to the exponential behaviour of the constraint.

Table 4. Evolution of Time (ms) and Memory (MB) consumption for MDD_{AD} intersection according to the variation of the number and size of sets (A, B, C in Fig. 2). The number of variables is equal to $Number \times Size$.

		Time (ms)						Memory (MB)					
		5	6	7	8	9	10	5	6	7	8	9	10
Number	Size												
	5	8	16	36	84	192	424	50	22	53	128	284	665
	6	19	36	132	212	492	1096	53	58	167	348	852	2042
	7	44	96	224	524	1232	2748	101	151	374	930	2312	5613
	8	101	257	548	1273	2993	8129	216	403	984	2456	6116	15433
	9	241	564	1397	3384	7814	15957	543	1010	3042	6605	16012	50836
	10	556	1348	3144	7116	16649	39073	1211	3079	6600	15308	37723	137917

The second test (Table 5) is a variant of the first one (Table 4). Arcs are added randomly between several sets, which as a consequence drastically increases the number of states in the MDD. The result is that intersection becomes impossible very quickly: for 6 sets and 6 values by set, we have a factor of 5 200 in time.

Table 5. Evolution of Time (ms) and Memory (MB) consumption for the ALLDIFFERENT MDD intersection according to the variation of the number and size of sets with random arcs added between sets

		5			6		
		Time	Memory	#Arcs	Time	Memory	#Arcs
Number	Size						
	5	322	107	6	2753	505	6
6	7059	2513	6	187061	53981	7	

5 Application: Construction of the MDD of Constraints

In this section, we show that our method can be useful to build the MDD of some constraints and not only to perform some intersections.

Consider C a constraint. Suppose that the construction of MDD_C , the MDD of C , is problematic because a very large number of intermediate states are generated but do not appear in the reduced MDD_C . As the reduction can gain an exponential factor this case is quite conceivable. It occurs, for example, with a bounded sum of variables that can take very different values. The number of states created is therefore huge, but it is quite possible that the reduction induced by the bounds on the sum removes a large part of them.

This kind of constraint can either prevent us from building the MDD due to lack of memory to store all the intermediate states, or require a lot of time to compute. To remedy these problems we propose to use successively our method on relaxations of MDD_C allowing to deal with smaller MDDs.

This approach assumes that it is possible to define different relaxations of MDD_C more or less strong. We recall that an MDD is a relaxation of an MDD if it represents a super set of the solutions of the exact MDD. In addition, we assume that the relaxation has fewer nodes. This is achievable by merging nodes for relaxing the MDD, which is quite usual. Thus, we suppose that we have MDDs noted $Relax(MDD_C, p)$ which are relaxations of MDD_C according to a parameter p such that $p < q$ implies $Relax(MDD_C, p)$ is smaller than or equal to $Relax(MDD_C, q)$. The value of p can be ad hoc. For example, for a sum constraint, a relaxation is simply to consider the numbers up to a given p precision. Thus we can merge many more states and the greater the precision the less the MDD is relaxed.

For convenience we will consider that $Relax(MDD_C, n)$ is MDD_C . Then, we can compute MDD_C by applying the following process named OTF Inc:

1. Let $M \leftarrow Relax(MDD_C, p)$
2. Compute M' by performing the intersection on the fly between M and $Relax(MDD_C, p + 1)$
3. Set $p \leftarrow p + 1$, and $M \leftarrow M'$
4. If $p < n$ then goto 2 else return M

6 Experiments

6.1 Constraint Building

We consider the following stochastic problem: there are n variables with domains having the same size. The values represent the chance for an event to appear. A solution is a combination of events such that their chance to happen simultaneously is above a certain threshold, for instance 75%. This problem is equivalent to a bounded product of variables. The goal is to build the MDD containing all the solutions. It is equivalent to building the MDD of C_{Π} the constraint defining

a bounded product of variables. The difficulty is that values are quite different (because computed from other elements) leading to the lack of collision.

We propose to compute the MDD of this problem by using the process OTF Inc defined in Sect. 5. We define $\text{Relax}(\text{MDD}_{C_{II}}, p)$ by the MDD of C_{II} for which the variables have been rounded to a precision p (i.e. the number of decimal places after the decimal point).

We test the method on different sets of data (available upon request). Each set involves values between 0.95 and 1 with at least 4 digits. The combined probability must be greater than 0.9. We compare the time and memory needed to compute the MDD using OTF Inc and the MDD directly computed (Base), both for a final precision $p = 8$, starting from $p = 0$. Table 6 shows that we obtain a factor of at least 9 both in time and in memory. We achieve up to a factor of 77.55 in time (181.5 s vs 2.6 s) and 19 in memory (405 MB vs 7 617 MB) for the hardest dataset (set 5).

Table 6. Time (ms) and memory (MB) needed to compute the exact MDD.

Set	Time (ms)			Memory (MB)		
	OTF Inc	Base	Ratio	OTF Inc	Base	Ratio
Set 1	1 131	10 235	9.05	136	1 423	10.46
Set 2	1 877	17 970	9.57	266	2 584	9.71
Set 3	1 405	25 528	18.17	189	1 869	9.89
Set 4	1 974	61 904	31.36	289	3 854	13.34
Set 5	2 642	181 468	77.55	405	7 617	18.81

6.2 The Car Sequencing Problem

A number of cars are to be produced. There are different options available to customise a car, and it is possible that a car has to be built with several options (paint job, sunroof, ABS, etc.). Each option is installed in a station that has a maximum handling capacity: if, for example, a station installing an option A can only handle one car in any two, then the assembly line must be designed in such a way that there are never two cars in a row requiring the option A . This constraint must be satisfied for each station. This problem is NP-complete.

All instances used in this article are available on csplib: <https://www.csplib.org/Problems/prob001/data/data.txt.html>

We will use the following methods:

- OTF: On The Fly, the method presented in this article.
- OTF_×: OTF performing operations with multiple constraints at once.
- Classic: The method that builds the MDD then performs the operation.

For the car sequencing problem it means that the final MDD is built as follows: for the classic method, we build the MDD containing all SEQUENCE constraints,

then we build the MDD of the GCC, and we perform the intersection between these two MDDs. For the OTF method, we do the same as the classic method, but the intersection with the GCC is done on the fly (we do not build the GCC MDD). Finally, for the OTF_×, we directly compute the intersection between the SEQUENCE and GCC, without building them explicitly.

Table 7. Problem 4/72 (Regin & Puget #1), Problem 19/71 (Regin & Puget #4) and Problem 60-02 from CSPLib. Time measured in ms and Memory in MB.

#options	Method	Problem 4/72			Problem 19/71		
		Time	Memory	Layer	Time	Memory	Layer
2	OTF	1042	187	100	915	127	100
	OTF _×	3435	712	100	5858	1274	100
	Classic	5177	1684	100	4845	1356	100
3	OTF	665 628	90 614	100	73 019	15 724	100
	OTF _×	2 736 002	224 931	69 ^{1*}	847 006	114 448	100
	Classic	957 063	231 213	38 ^{2*}	819 905	214 445	38 ^{2*}
Problem 60-02							
2	OTF	41 129	9 043	100			
	OTF _×	315 825	33 986	100			
	Classic	51 975	11 059	100			
3	OTF	1 960 854	212 669	41 ^{3*}			
	OTF _×	2 763 521	172 543	31 ^{1*}			
	Classic	957 063	231 213	38 ^{2*}			

^{1*}: MO during intersection of SEQUENCE + GCC

^{2*}: MO during the construction of the GCC, after the SEQUENCE intersection

^{3*}: MO during the GCC intersection

Table 7 shows that it is possible to build an MDD that is **not** possible to build otherwise (because the GCC explodes in memory). Thus, we can conclude that an instance has no solution (*Problem 19/71*), which we could not do before. In the case of larger problems, we still manage to observe a strong progression, even if it remains insufficient: where we could only build 38 layers of the GCC, we manage to carry out the intersection with it up to layer 41 (*Problem 60-02*). These results clearly show the advantage of this intersection method.

We notice that OTF_× is systematically worse than the method doing them one by one (*Problem 4/72*, *Problem 60-02*), and even worse in some cases than Classic (*Problem 60-02*). This can be explained by the fact that the complexity is proportional to the number of states constructed for each constraint. However, by making successive intersections, we observe a reduction in the number of solutions, which can imply (and does imply in a general case) a reduction of states. Performing several operations at the same time is therefore not interesting, especially if the MDDs are easy to construct, i.e. they are not exponential in memory like would be ALLDIFFERENT or GCC.

7 Conclusion

This article shows that building constraints' MDD during an operation is more advantageous in every way than building the complete constraint's MDD first, even if it does not prevent an explosion of memory. Moreover, this method shows a major impact in performance for solving some well known problems or building MDDs of constraints. However, doing multiple constraints at once is not necessarily better, and is shown to be worse most of times.

References

1. Andersen, H.R.: An Introduction to Binary Decision Diagrams (1999)
2. Bergman, D., Ciré, A.A., van Hoeve, W., Hooker, J.N.: Decision Diagrams for Optimization. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, Heidelberg (2016). <https://doi.org/10.1007/978-3-319-42849-9>
3. Bergman, D., Cire, A.A., Van Hoeve, W.J., Hooker, J.N.: Discrete optimization with decision diagrams. *INFORMS J. Comput.* **28**(1), 47–66 (2016)
4. Bergman, D., van Hoeve, W.-J., Hooker, J.N.: Manipulating MDD relaxations for combinatorial optimization. In: Achterberg, T., Beck, J.C. (eds.) CPAIOR 2011. LNCS, vol. 6697, pp. 20–35. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21311-3_5
5. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.* **35**(8), 677–691 (1986). <https://doi.org/10.1109/TC.1986.1676819>
6. Cheng, K.C.K., Yap, R.H.C.: An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints* **15**(2), 265–304 (2010). <https://doi.org/10.1007/s10601-009-9087-y>
7. Davarnia, D., van Hoeve, W.: Outer approximation for integer nonlinear programs via decision diagrams. *Math. Program.* **187**(1), 111–150 (2021). <https://doi.org/10.1007/s10107-020-01475-4>
8. Demasse, S.: Compositions and hybridizations for applied combinatorial optimization. Habilitation à Diriger des Recherches (2017)
9. Gentzel, R., Michel, L., van Hoeve, W.-J.: HADDOCK: a language and architecture for decision diagram compilation. In: Simonis, H. (ed.) CP 2020. LNCS, vol. 12333, pp. 531–547. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58475-7_31
10. Hadzic, T., Hooker, J.N., O'Sullivan, B., Tiedemann, P.: Approximate compilation of constraints into multivalued decision diagrams. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 448–462. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85958-1_30
11. Hoda, S., van Hoeve, W.-J., Hooker, J.N.: A systematic approach to MDD-based constraint programming. In: Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 266–280. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15396-9_23
12. Kam, T., Brayton, R.K.: Multi-valued decision diagrams. Technical report. UCB/ERL M90/125, EECS Department, University of California, Berkeley. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/1990/1671.html>
13. Perez, G., Régin, J.-C.: Improving GAC-4 for table and MDD constraints. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 606–621. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_44

14. Perez, G., Régin, J.C.: Efficient operations on MDDs for building constraint programming models. In: International Joint Conference on Artificial Intelligence, IJCAI 2015, Argentina, pp. 374–380 (2015)
15. Perez, G., Régin, J.C.: Soft and cost MDD propagators. In: The Thirty-First AAAI Conference on Artificial Intelligence (AAAI 2017) (2017)
16. Roy, P., Perez, G., Régin, J.-C., Papadopoulos, A., Pache, F., Marchini, M.: Enforcing structure on temporal sequences: the Allen constraint. In: Rueher, M. (ed.) CP 2016. LNCS, vol. 9892, pp. 786–801. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_49
17. 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>
18. Tjandraatmadja, C., van Hoeve, W.-J.: Incorporating bounds from decision diagrams into integer programming. *Math. Program. Comput.* **13**(2), 225–256 (2020). <https://doi.org/10.1007/s12532-020-00191-6>
19. Verhaeghe, H., Lecoutre, C., Schaus, P.: Compact-MDD: efficiently filtering (s) MDD constraints with reversible sparse bit-sets. In: IJCAI, pp. 1383–1389 (2018)