Dealing with the product constraint

Steve Malalel, Victor Jung, Jean-Charles Régin, Marie Pelleau

Dealing with the product constraint

Steve Malalel, Victor Jung, Jean-Charles Régin, and Marie Pelleau

Université Côte d'Azur, CNRS, I3S, France
{firstname.lastname}@univ-cotedazur.fr

Abstract. The product constraint ensures that the product of some variables will be greater than a given value, that is $\Pi_{i=1}^{n} x_i \geq w$. With the emergence of stochastic problems, this constraint appears more and more frequently in practice. The variables are most often probability variables that represent the probability that an event will occur and the minimum bound is the minimum probability that must be satisfied. This is often done to guarantee a certain level of security or a certain quality of service. To deal with this constraint, it is tempting as proposed by many authors to take the logarithm of the sum and the bound in order to transform the product into a sum. In this article we show that this idea creates many problems and forbids an exact calculation. We propose and compare different representations allowing to compute the set of solutions of this problem exactly or up to a certain precision. We also give an efficient method to represent that constraint by a Multi-valued Decision Diagram (MDD) in order to combine this constraint with some others MDDs.

1 Introduction

More and more problems involve uncertain data associated with probabilities. For quality of service or security reasons, it is frequently imposed that any solution must be associated with a minimum probability. This kind of problem is naturally modeled by defining for each variable x representing uncertain values, a variable p_x which represents the probabilities of these values. Then, the variables x and p_x are linked together (i.e. $x = a \Leftrightarrow p_x = p(a)$) and the constraint $\Pi_{i=1}^{n} p_{x_i} \geq w$ is added to the model in order to guarantee that each solution will be associated with a probability higher than a given value.

We are mainly interested in defining the multi-valued decision diagram (MDD) of this constraint as it is classically made for a sum constraint. We assume that the values of variables are decimal with a given precision.

Usually, the product constraint is modeled by taking the logarithm of both terms, and so by transforming it into the sum $\sum_{i=1}^{n} \log(p_{x_i}) \geq \log(w)$. It seems more convenient because the product of variables is not easy to manage in constraint programming solvers due to overflows. However, using a logarithm has a major drawback: we lose the possibility to make exact calculations because the logarithm function cannot be represented exactly in a computer as it can return a transcendental number. Thus, floating-point numbers have to be used and errors in the representation have to be managed.

In this paper we study several methods for defining the MDD of this constraint. The first one is based on the sum of the logarithm. The second one computes the exact MDD of the product of variables. Unfortunately, this method may need a lot of memory. Therefore we propose to relax the previous MDD up to a certain precision. On the other hand, we present a method that builds the MDDs by successive iterations and further compresses it, taking into account the bound imposed on the constraint. Each iteration corresponds to a precision that is higher than the previous one. The main idea is to stop considering the parts of the MDD that will always be satisfied when the precision of the computation is increased. For instance, no matter the precision, we will always have $(0.95... \times 0.95...) > 0.9$.

The paper is organised as follows. First, we recall some definitions. Then, we present different methods to compute the MDD of the product constraint. Next, we experiment with these methods. At last, we conclude.

2 Preliminaries

2.1 Constraint Programming

A finite constraint network $\mathcal{N}$ is defined as a set of n variables $X = \{x_1, \ldots, x_n\}$, 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 $X(C)$ denoted by τ. In a tuple τ, the assignment of the ith variable is denoted by τ_i.

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

Definition 1 *Given X a set of variables and w a value, the* sum constraint *ensures that the sum of all variables $x \in X$ is greater than or equal to w.*
$\text{SUM}(l, u) = \{\tau \mid \tau \text{ is a tuple on } X(C) \text{ and } \sum_{i=0} \tau_i \geq w\}$

Definition 2 *Given X a set of variables and w a value, the* product constraint *ensures that the product of all variables $x \in X$ is greater than or equal to w.*
$\text{PRODUCT}(l, u) = \{\tau \mid \tau \text{ is a tuple on } X(C) \text{ and } \Pi_{i=0} \tau_i \geq w\}$

Use of logarithm function. Using mathematical functions in CP imposes to have some guarantees on the values that are computed. In this case, two important concepts are considered: the correctness and the completeness.

Consider M a model of a problem P, that can use different types of numbers like floating-point and real numbers. We say that a model M satisfies the correctness condition iff all solutions found by the solver are solutions of P. In other words, there is no solution of M that is not a solution of P. The completeness

condition is satisfied iff all solutions of P are solutions of M. If both conditions are met then we say that P is exactly solved.

Unfortunately, the log function may be very complex to calculate exactly, mainly because the discrete logarithm problem (given real numbers a and b, the logarithm $\log_b(a)$ is a number x such that $b^x = a$) is considered to be computationally intractable. Note that computing the log function with a fixed precision is equivalent to the discrete logarithm problem. Therefore, we need to compute an approximation of log. Consider $\underline{\log}$ (resp. $\overline{\log}$) a lower bound (resp. upper bound) of the log function. If the product constraint is modeled by $\sum_{i=1}^{n} \underline{\log}(p_{x_i}) \geq \overline{\log}(w)$ then the model is correct, because $\sum_{i=1}^{n} \log(p_{x_i}) \geq \sum_{i=1}^{n} \underline{\log}(p_{x_i}) \geq \overline{\log}(w) \geq \log(w)$. If the product constraint is modeled by $\sum_{i=1}^{n} \overline{\log}(p_{x_i}) \geq \underline{\log}(w)$ then the model is complete because $\sum_{i=1}^{n} \overline{\log}(p_{x_i}) \geq \sum_{i=1}^{n} \log(p_{x_i})$ and $\overline{\log}(w) \geq \underline{\log}(w)$.

As the filtering algorithms in CP are based on the deletion of values that do not belong to a solution, it is fundamental to guarantee that at least all solutions of the problem are considered and thus that the model is complete. This means that we need to be able to compute both a lower and an upper bound of the log. Unfortunately, the properties of the log functions available in a language, like Java, or in a library is not always provided. However, modern implementation of elementary functions are at least faithful [7], i.e., they return one of the two floating-point numbers surrounding the exact value $\log(x)$. Thus, with floating-point representation, a lower bound can be obtained by subtracting one machine epsilon to the value returned by the log function and an upper bound can be obtained by adding one machine epsilon to the value returned by the log function.

Use of decimal variables. Decimal variables impose a certain precision in their representation and in the calculations involving them. If they are represented by floating-point variables IEEE754 rounding modes can be managed for ensuring the safeness of some operations (in order to guarantee the completeness of the model). Note that some programming languages, like Java, do not offer this possibility.

We will use the following notations:

Notation 1

- δ *is the precision of a decimal variable, i.e. the number of decimal digits that are taking in account (after the decimal separator).*
- ϵ *is the computational precision between decimal numbers.*

2.2 Multi-valued Decision Diagram

The decision diagrams considered in this paper are reduced ordered multi-valued decision diagrams (MDD) [6,8,1], which are a generalisation of binary decision diagrams [2]. 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. The set of outgoing arcs from node u is denoted by $\omega^+(u)$. 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.

Construction of MDDs. The classical approach to build MDD(C), the MDD of a constraint C, is to use states. When building MDD(C), we assign an information representing the current state $s(x)$ of the constraint C to each node x. Given (u, a, v) an arc, $s(u)$ the state of the node u and a transition function, we are able to produce $s(v)$ the state of the node v and to know if this state satisfies the constraint C or not. If two different nodes a and b have the same state (i.e. $s(a) = s(b)$), they can be merged into one node ab with $s(ab) = s(a) = s(b)$ during the building process. As we try to build the MDD of a constraint we add a validity function that anticipates the fact that a state cannot lead to at least one solution. This function, noted isValid returns false if we can guarantee that a state will never satisfy the constraint. This function is called before creating a state. For a given layer i it computes the maximum value from the next layer to tt denoted by $vMax[i+1]$.

In the case of the sum (resp. product) constraint, the state represents the current sum (resp. product) of the current node. The creation of a new state is given by function createState. The transition function is simply the addition (resp. multiplication) between the current sum (resp. product) and the label, therefore we will not explicit them.

3 Product constraint as a sum of logarithms

There are two possible ways to build the MDD of the constraint depending on the representation of the decimal/floating-point variable: either by floating-point or by integers. In any case, a faithful log function has to be used and the rounding errors of the sum have to be managed. In the first case, the MDD of the constraint is just the MDD of the sum constraint on floating-point variables. The latter case deserves more attention.

The logarithm is computed up to a certain precision λ, that is to say the number of digits taken into account after the decimal point [3]. This value can be different than δ (the precision of decimal variables). It is therefore important

Algorithm 1 Integer Logarithm Representation

```
createState(state, label) : State
    newState ← createState()
    newState.sum ← state.sum + label
    return newState
isValid(C, state, label, layer) : Boolean
    newSum ← state.sum + label
    maxPotential ← newSum + vMax[layer + 1]
    return maxPotential ≥ C.min
signature(C, state, label, layer, size) : Integer
    return ⌈(state.sum + label)/10^(δ−ϵ)⌉
merge(C, state1, state2)
    if state2.sum < state1.sum then state1.sum ← state2.sum
```

to take a precision value such that the integer representation of the sum does not cause an overflow: $n \times 10^{\lambda+d} \leq 2^b$, with d the maximum number of digits required to represent the integer value of the logarithm, n the number of variables and b the number of bits used to represent the integer (32 for an int, 64 for a long). We will denote this representation `Integer Logarithm Representation`. Algorithm 1 gives a possible implementation of the required functions to build the MDD of the product constraint, with $vMax[i] = \sum_{j=i}^{n} max(D_j)$. Algorithm 1 also gives a possible implementation of the merging conditions. To perform merges during the construction process, we introduce a SIGNATURE function that will behave as a hash : if two nodes have the same signature, then we merge them according to the MERGE function. This allows us to keep a slightly more precise *sum* variable, while still being able to merge nodes and save some space.

4 Exact representation of the product constraint

This method aims to be as accurate as possible by representing the result of consecutive multiplications without loss. First, the decimal variables are transformed into integer variables. We choose a value for δ (i.e., the number of decimal we want to take into account for representing the values of variables) and then we turn decimals into their corresponding integers (for example with the decimal 0.98 and $\delta = 4$, we obtain 9800). Afterward, knowing w, the minimum value of the product of variables, and n the number of variables in the scope of the constraint, we calculate the minimal threshold $min = \lfloor w \times 10^{(n \times \delta)} \rfloor$.

The state of a node contains the value *prod* which is the product of the labels of the arcs from the *root* node to the current node. The *prod* value of the *root* is 1. Algorithm 2 gives a possible implementation for the createState and isValid functions. isValid is defined using $vMax[i] = \prod_{j=i}^{n} max(D_j)$.

Unfortunately, the use of integers is limited when representing big numbers: multiplying all the integers together can quickly cause an overflow. In order to address this issue, a specific data structure that can represent really big numbers

Algorithm 2 Exact Product Implementation

```
createState(state, label)
    newState ← createState()
    newState.prod ← state.prod × label
    return newState
isValid(C, state, label, layer) : Boolean
    newProd ← state.prod × label
    maxPotential ← newProd × vMax[layer + 1]
    return maxPotential ≥ C.min
```

must be used. Most programming languages offer such data structures (e.g. BigInteger in Java). However, these data structures have two important drawbacks that can prevent their use in practice: they quickly run out of memory during the computation, and the computation of the multiplication takes more time as the product value increases. It is therefore necessary to study a more relaxed representation.

5 Relaxed Product constraint

In order to decrease the memory consumption and the computation time, we can deliberately lose accuracy during the computation by truncating and rounding the result correctly, depending on a given precision ϵ. Therefore, instead of computing the exact MDD by performing basic multiplications, *relaxed* multiplications are performed, which will as a result compute a relaxed MDD. This means that the MDD can either be complete, correct, or both (but both cannot be guaranteed). We note this multiplication $\times^\epsilon$ with ϵ the precision such that $\lceil x \times^\epsilon y \rceil = \lceil (x \times y)/10^\epsilon \rceil$ and $\lfloor x \times^\epsilon y \rfloor = \lfloor (x \times y)/10^\epsilon \rfloor$. For example $\lceil 9800 \times^4 9780 \rceil = 9585$. Similarly we note $\prod^\epsilon$ the product using $\times^\epsilon$. Note that for the following part the rounding is done to guarantee only the completeness, but it can also be done to guarantee only the correctness by doing a switch between the floor rounding and the ceiling rounding. The MDD is built following the same first steps defined for the Exact Product: we choose the precision ϵ and turn floating-point numbers into their corresponding integers.

However, the minimal threshold is different: it is now defined as $min = \lfloor w \times 10^\epsilon \rfloor$. The way the value $vMax[i]$ is calculated for each layer is modified to take this change into account i, $vMax[i] = \lceil \prod^{\epsilon n}_{j=i} max(D_j) \rceil$.

The state of a node x still holds the value *prod*, but it now represents the consecutive relaxed multiplications between the labels of the arc from the node *root* to the node x. We initialise the root value as 10^ϵ, which is the equivalent of 1 in ϵ precision.

Algorithm 3 gives a possible implementation of the createState and isValid functions.

Even if this method cannot ensure both the completeness and the correctness at the same time, it allows us to balance between performance and accuracy

Algorithm 3 Relaxed Product Implementation

```
createState(state, label) : State
    newState ← createState()
    newState.prod ← ⌈state.prod ×^ϵ label⌉
    return newState
isValid(C, state, label, layer) : Boolean
    newProd ← ⌈state.prod ×^ϵ label⌉
    maxPotential ← ⌈newProd ×^ϵ vMax[layer + 1]⌉
    if maxPotential < C.min then return false
    return true
```

of the solutions. Indeed the smaller ϵ is, the more equivalent states we have. Therefore, we obtain more merges resulting in less computational resources to build the MDD.

6 Incremental Precision Refinement

In this section we present an Incremental Precision Refinement of the set of solutions (IPR). This method aims at computing the MDD of the constraint for a precision ϵ by computing successive MDDs of the constraint having a lower precision. Let MDD_ϵ be the MDD for the precision ϵ. The idea is to start by building MDD_1 and then build MDD_k from MDD_{k-1}. The advantage of this approach is that it avoids creating intermediate states that are not in the final reduced MDD.

We can classify the solutions computed in a relaxed MDD in two categories, the "sure solutions" and the "relaxed solutions". The "sure solutions" are the solutions that no matter the precision are valid. For instance $(0.95... \times 0.95...)$ is greater than 0.9 no matter the precision $k > 2$. The "relaxed solutions" are the solutions for which a higher precision is required in order to determine if they are solutions. For instance the fact that $(0.94... \times 0.95...)$ is greater than 0.9 is uncertain and requires a higher precision ($0.948 \times 0.955 > 0.9$ and $0.940 \times 0.950 < 0.9$).

This method is based on the fact that "sure solutions" in MDD_k are solutions in all the following iterations. Thus the method focuses only on the part of the MDD containing "uncertain solutions" to improve the precision at a lower cost.

6.1 Extracting suspicious arcs

The first step of the algorithm is to identify and extract arcs and nodes that are part of at least one "relaxed solution". In order to do so, we use the scheme introduced in [4] to propagate the bounds of the product constraint to each node of the MDD. In our case, the property associated with a node is the current possible sum interval for the SUM constraint and the current possible product interval for the PRODUCT constraint. A bottom-up propagation is performed (instead of the

Algorithm 4 Extraction

```
EXTRACTION(mdd, w) : MDD
    mdd_M ← CREATEMDD()
    mdd.root.value ← 0
    mdd.root.x_1 ← mdd_M.root
    L[0] ← {mdd_M.root}
    foreach i ∈ 0..r − 1 do
        foreach Node x ∈ L[i] do
            foreach label ∈ ω⁺(x) do
                y ← x.GETCHILD(label)
                v ← x.value + label
                if y.property[0] + v < w then
                    x_1 ← x.x_1
                    y_1 ← y.x_1
                    if y_1 is nil then
                        y_1 ← CREATENODE()
                        y.x_1 ← y_1
                        ADDNODE(L, y_1, i + 1)
                    ADDARC(x_1, label, y_1)
                    y.value ← MIN(y.value, v)
    merge all nodes of L[r] into t
    PREDUCE(L)
    return mdd_M
```

top-down propagation presented in the cited paper). Starting from the *tt* node the propagation computes for each node the interval of the minimal values, called property, needed to be part of a "sure solution". After the bottom-up propagation, a top-down propagation of properties is performed. Starting from the root node, each outgoing arcs $(source, v, destination)$ is checked. If the value v of the arc combined with the property p of the *destination* is below the threshold w, then we cannot be certain that this arc only leads to "sure solutions". This arc is thus added to the marked MDD_M containing all arcs and nodes marked "suspicious". When such an arc is marked, the value of the node *destination* is updated to the lowest value between its current value and the value of *source* combined with the value of the arc v. At the end of the algorithm, we obtain the MDD_M containing all arcs and nodes appearing in at least one "relaxed solution".

The implementation of the algorithm is given Algorithm 4. The algorithm takes as input an MDD *mdd* and the lower bound w.

Note that the algorithm only deals with a lower bound w, but can easily be adapted to deal with an upper bound, or both a lower and upper bound.

Proposition 1 *After executing Algorithm 4 MDD_M contains all the "suspicious" arcs and nodes.*

Proof. Let the *root* node with property $p < w$. This means that at least one "relaxed solution" pass by the root node, which means that at least one "relaxed solution" pass by one of its children. Let p_c be the value of the property of the child c, and v be the value of the arc from c to *root*. If $p_c + v < w$, it means that at least one "relaxed solution" pass by the arc $(root, v, c)$, because taking it makes the property go below the threshold. Now, suppose that each node holds the value of the lowest path from the root. Consider the arc $(x, label, y)$ where x has a value v_x below the threshold w, and y has a property p_y. $p_y + label + v_x$ correspond to the lowest possible path taking the arc $(x, label, y)$. If this value is below the threshold w, then it means that at least one "relaxed solution" pass by this arc.

6.2 On the fly intersection

After extracting MDD_M containing all "relaxed solutions", we need to improve the precision in order to only have solutions. To do so, we perform the intersection between the MDD with higher precision and MDD_M, but without computing the entire constraint: we only need the parts in common with MDD_M. In order to do so, we perform the on the fly intersection [5]. This allows us to compute higher precision only for the parts of the MDD that need it, without wasting time and memory to recompute a large amount of solutions.

6.3 IPR Algorithm

We give a possible implementation of the global algorithm (Algorithm 5) that computes the MDD of the product constraint using the different schemes presented in this section. The algorithm will stop once it reaches an equilibrium, or when the maximum precision allowed is reached.

Algorithm 5 IPR Algorithm

```
IPR(w, ϵ, D) : MDD
    mdd_c ← createMDD(0, w, D)
    mdd_M ← extraction(mdd_c, w)
    mdd_S ← mdd_c − mdd_M
    foreach e in 1..ϵ do
        mdd_c ← performIntersection(mdd_M, C, e)
        mdd_M ← extraction(mdd_c, w)
        if mdd_M is empty then return mdd_S
        mdd_S ← mdd_S ∪ (mdd_c − mdd_M)
    return mdd_S ∪ mdd_c
```

The first step of the algorithm is to compute the initial MDD_c (with lowest precision). Once this MDD is created, we execute the extract function (Algorithm 4) and retrieve the associated MDD_M. Then, we compute the difference between

the MDD_c and MDD_M, basically filtering the initial MDD from all uncertain solutions. We store all good solutions in an accumulator MDD_S. We then repeat the same steps over MDD_M: we intersect it with the MDD of the constraint C with higher precision e (PERFORMINTERSECTION), then extract, filter, and add solutions to MDD_S. The algorithm stops when MDD_M is empty or when the maximum precision allowed is reached.

Note that, even if very unlikely, it is possible that $\text{MDD}_c = \text{MDD}_M$. This cannot be a stop criterion for the algorithm as it would still be possible that all solutions of MDD_c require a higher precision to decide if they belong to MDD_S or not.

7 Experiments

The algorithms presented in this paper have been implemented in Java 11. The experiments were performed on a machine having four E7-4870 Intel processors, each having 10 cores with 256 GB of memory and running under Scientific Linux. All the experiments were run in sequential.

First we use fixed data sets, then we study the impact of varying some parameters (number of variables, domain size, w value). The fixed data sets, denoted by data1... data10, involve 10 variables with a domain of size 10. Each value represents a probability between 95% and 100% to ensure that there exist solutions for the instances. Each resolution was made with our minimum threshold w representing 90%. All the data used in this paper are available upon request.

7.1 Exact product method

Data set	**#Solutions**	**Time (ms)**	**Memory (MB)**
data1	341 051	7 403	2 268
data2	902 485	14 134	3 383
data3	1 819 820	24 629	6 508
data4	489 297	5 469	1 807
data5	104 506	2 373	879
data6	882 970	13 567	3 394
data7	4 049 230	98 740	14 072
data8	510 291	23 077	4 121
data9	5 473 625	389 801	25 273
data10	797 484	37 497	4 689

Table 1: Time (ms) and Memory (MB) needed to compute the exact MDD using the Exact Product method.

The Exact Product method behaves exactly as expected: we obtain the exact MDD at a high cost both in term of memory consumption and time (Table 1).

This method is nonetheless interesting because it serves as a proof of the total number of solutions, which will be helpful to compare the relative accuracy of other methods (Table 2).

7.2 Logarithm and Relaxed Methods

Comparison of the methods. Concerning the number of solutions produced by the Logarithm method (both Integer and Library) and Relaxed Product method, we notice an interesting phenomenon: the more we increase the precision ϵ, the less we improve the lower bound and thus the accuracy of the solutions at each iteration (Table 2). Even worse, the time needed to compute the MDD does not scale at all with the number of solutions (Table 3). For instance, the time needed to compute 5 473 669 solutions is about 13s (Table 2 and Table 3) for $\epsilon = 7$, while the time needed to compute 5 473 625 solutions is about 103s for $\epsilon = 8$. On this data, the computation time is 8 times slower when $\epsilon = 8$ than when $\epsilon = 7$ for a difference of 44 solutions. It means that the trade-off between precision and computational resources is not worth it. It is nevertheless complicated to determine the correct ϵ such that we do not spend too much time on computation for a relatively good approximation of the exact MDD. Furthermore, the "relaxed" solutions introduced by the relaxation are very close to the defined threshold w (as shown by the lower bounds in Table 2). However, we notice that it is faster to build the MDD using the logarithm than using the Exact Method; we obtain the exact MDD at $\epsilon = 8$ for a time of 103s (compared to almost 400s). This difference is explained by the heaviness of the exact representation.

ϵ	Logarithm	Lower Bound	Relaxed Product	Lower Bound
1	**9 800 000 000**	≈ 0.628	10 000 000 000	≈ 0.628
2	261 007 356	≈ 0.762	**213 455 660**	≈ 0.821
3	**9 193 737**	≈ 0.886	9 222 380	≈ 0.891
4	**5 731 323**	≈ 0.899	5 770 914	≈ 0.8992
5	**5 493 720**	≈ 0.8999	5 498 953	≈ 0.8999
6	**5 474 681**	≈ 0.89999	5 476 117	≈ 0.89999
7	**5 473 669**	≈ 0.899999	5 473 844	≈ 0.899999
8	**5 473 625**	≈ 0.90	5 473 649	≈ 0.8999999
9	**5 473 625**	≈ 0.90	5 473 627	≈ 0.89999999

Table 2: Comparison of the number of solutions generated depending on the precision ϵ for data9. Number of exact solutions: 5 473 625.

Table 3 also shows that the method using a sum of logarithm based on a log function call from a library and using floats (Library Log), the method using a log sum represented as an integer and whose precision is controlled (Integer Log) and the method of relaxing the MDD of the exact product of variables constraint (RelaxProd) give very close results as soon as one chooses a

computational precision higher than 4 decimals. However, the IntegerLog method seems to be faster and to consume a little less memory than the other two methods.

	Time (ms)			**Memory (MB)**		
ϵ	Library Log	Integer Log	RelaxProd	Library Log	Integer Log	RelaxProd
1	49	49	**46**	4	**3**	**3**
2	60	**56**	68	**4**	**4**	**4**
3	92	**78**	109	6	**4**	7
4	197	**174**	240	19	**17**	31
5	566	**509**	801	110	**106**	166
6	2 752	**2 546**	4 558	609	**598**	830
7	13 604	**13 165**	25 848	2 556	**2 377**	2 915
8	103 460	**102 079**	181 015	6 852	**6 756**	7 251
9	324 239	**313 094**	337 803	9 929	9 838	**7 854**

Table 3: Time (ms) and memory (MB) needed to compute the MDD of data9 for a given ϵ depending on the representation used.

7.3 Incremental Precision Refinement (IPR)

Tables 4 and 5 clearly show the advantages of the IPR routine. IPR L is the application of the IPR routine to the Library Log model, IPR IL is the application of the IPR routine to the Integer Log model and IPR RP is the application of the IPR routine to the RelaxProd model.

	Time (ms)					**Memory (MB)**				
Data set	Lib Log	IPR L	IPR IL	IPR RP	Exact Prod	Lib Log	IPR L	IPR IL	IPR RP	Exact Prod
data1	5 708	**872**	904	1 145	7 403	808	**113**	**113**	173	2 268
data2	12 067	**1 075**	1 095	1 628	14 134	1 496	154	**153**	229	3 383
data3	23 069	1 883	**1 869**	2 420	24 629	2 062	258	**257**	397	6 508
data4	4 629	859	**858**	1 127	5 469	811	109	**105**	173	1 807
data5	1 930	616	**609**	825	2 373	378	**65**	**65**	105	879
data6	10 359	**980**	1 018	1 424	13 567	1 494	142	**137**	221	3 394
data7	81 257	**2 357**	2 526	2 945	98 740	5 775	**345**	**345**	495	14 072
data8	21 990	**954**	974	1 343	23 077	1 642	133	**129**	205	4 121
data9	324 239	**2 962**	3 085	3 715	389 801	9 929	438	**437**	619	25 273
data10	31 717	**1 182**	1 200	1 614	37 497	1 552	154	**153**	250	4 689

Table 4: Time (ms) and Memory (MB) comparison between the Exact Product method, the Library Log method with $\epsilon = 9$ and the methods with the IPR routine in order to compute the exact MDD.

The IPR routine improves the computation up to a factor 131 in time and 60 in memory (data9 in Table 4). Furthermore, contrary to the direct computation

of the MDD at a given precision ϵ, it is possible to guarantee the exactitude of the MDD if the IPR routine stopped before reaching the maximum allowed precision. Moreover, we can see that the differences between all IPR methods and the exact method or the Library Logarithm method are in the same order of magnitude. This shows that the IPR routine is generalisable to any form of relaxed representation.

Parameters			**Time (ms)**				**Memory (MB)**			
n	$\|D\|$	w	IPR L	IPR IL	IPR RP	Exact	IPR L	IPR IL	IPR RP	Exact
10	10	0.77	80 932	**74 697**	91 505	-	**7 996**	8 640	11 357	MO
10	15	0.65	**8 779**	9 091	9 186	-	**1 262**	1 267	1 351	MO
10	15	0.90	9 090	**8 857**	13 773	-	1 421	**1 420**	1 893	MO
15	10	0.85	**248 596**	256 005	276 118	-	24 672	**24 557**	24 572	MO
15	10	0.90	**1 615**	1 684	2 222	49 986	**217**	218	277	7 176
15	15	0.92	12 139	**11 329**	16 273	-	1 647	**1 643**	2 136	MO
15	15	0.90	**178 285**	185 549	205 604	-	20 050	**20 046**	22 914	MO
20	5	0.9	**32 363**	32 598	44 197	-	4 270	**4 263**	5 108	MO

Table 5: Time (ms) and Memory (MB) comparison between the different methods depending on the variations of the parameters, with n the number of variables, $|D|$ the size of each domain and w the threshold. The first line corresponds to data10 in other benchmarks. MO = 30GB. All solving methods have the same number of solutions.

Variations of data set parameters Table 5 shows the behaviour of the building process depending on the different parameters such as the number of variables n, the size of the domains $|D|$ or the threshold w. The results seem to show that, the more we increase n and $|D|$, the more difficult the problem is to resolve, which is an expected result. The Exact Product method is only able to close one instance, which is the easiest one ($n = 15$, $|D| = 10$, $w = 0.90$). Nonetheless, the results are confirming yet again that the IPR method dominates the classical building approach. When comparing the efficiency of the methods, we find the same results as in Table 3 : the logarithm approach is better than the relaxed product in terms of time and memory. Even though very close, the Library Logarithm (L) seems to be better at solving these instances than the Integer Logarithm (IL). However, the variation of the effect of w seems interesting : for some instances, lowering it makes the problem more difficult (1 615ms for $n = 10$ $|D| = 15$ $w = 0.90$ compared to 248 596 for $w = 0.85$) while it makes it easier for others ($n = 10$, $|D| = 15$).

When focusing particularly on the variation of w, we in fact observe a bell-shaped curve evolution for the time and memory (Figures 1 and 2). The top of the curve seems to be achieved for the value w such that it cuts the set of all possible combinations in half (Table 3). For instance, the first line ($n = 10, |D| =$

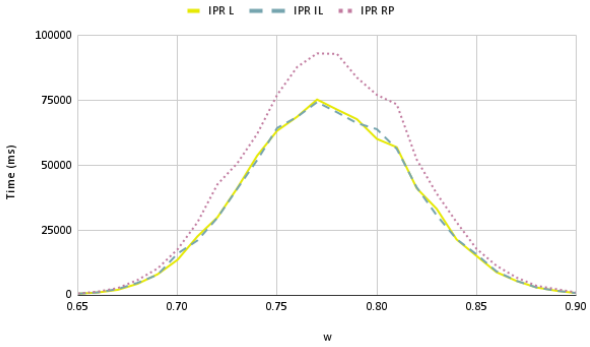

Fig. 1: Evolution of the time (ms) needed to compute the MDD for data1 depending on the parameter w.

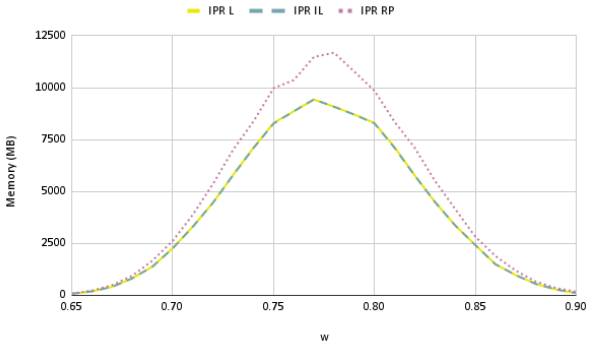

Fig. 2: Evolution of the memory (MB) needed to compute the MDD for data1 depending on the parameter w

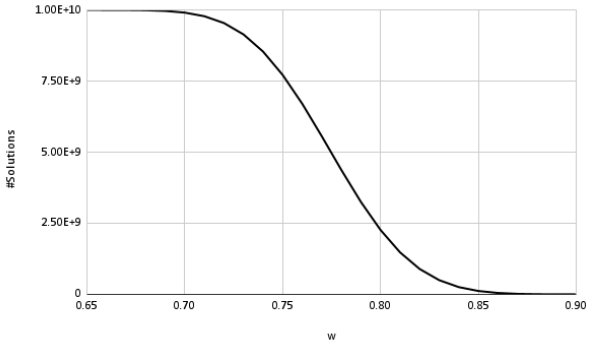

Fig. 3: Evolution of the number of solutions for data1 depending on the parameter w.

$10, w = 0.77$) of Table 5 is very close to the top of the curve (Figure 1), resulting in a factor 60 in time when comparing with $w = 0.90$ for the same data set (data10 in Table 4).

8 Conclusion

In this paper we studied several methods for defining the MDD of the PRODUCT constraint of decimal variables. The first and most popular one is based on the sum of the logarithm (using either floating point or integer numbers) with a given precision. The second one computes the exact MDD of the product of variables. The last one relaxes the previous MDD up to a certain precision.

We showed that an exact representation is not that expensive in terms of computational resources. More importantly we showed that models based on a precision can be accurate when using at least 5 decimals.

We also presented an incremental precision refinement method that efficiently computes an MDD for a given precision. It relies on the fact that if a solution is correct (a solution of the constraint) at a given precision it is also a correct solution at a higher precision. Thus this method only refines the precision on the uncertain parts of the MDD. In addition, when this method stops before reaching the fixed precision, it guarantees that the resulting MDD is exact. We showed that this method is very efficient both in terms of computational time and memory consumption no matter the method used to compute the logarithm.

In a future work it would be interesting to see if the obtained results on the logarithm are still the same when values are represented as interval of probabilities. Another very interesting development would be to study what are the necessary conditions for the use of the incremental precision refinement method, and what kind of constraints meet them.

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. 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
2. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819, https://doi.org/10.1109/TC.1986.1676819

3. Goldberg, M.: Computing logarithms digit-by-digit. International Journal of Mathematical Education in Science and Technology **37**(1), 109–114 (2006)
4. Jung, V., Régin, J.C.: Checking constraint satisfaction. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 332–347. Springer (2021)
5. Jung, V., Régin, J.C.: Efficient operations between mdds and constraints. Tech. rep., Submitted to the International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (2022)
6. Kam, T., Brayton, R.K.: Multi-valued decision diagrams. Tech. Rep. UCB/ERL M90/125, EECS Department, University of California, Berkeley (1990), `http://www2.eecs.berkeley.edu/Pubs/TechRpts/1990/1671.html`
7. Muller, J.M.: Elementary Functions, Algorithms and Implementation. Birkhäuser, 2nd edn. (2006)
8. 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