A BDD Domain for Constraint Programming

Mathieu Vavrille
M2 internship
Supervised by Charlotte Truchet
TASC team, LS2N (UMR 6004), Université de Nantes

May 31, 2019

Contents

1 Introduction

Constraint Programming is a research field aiming at solving combinatorial problems. It inherits from Artificial Intelligence for its declarative framework, and is also tied to Operations Research for its efficiency at solving hard problems. Problems are specified as a set of constraints between variables that are the unknown of the problem. Variables are given a domain, *i.e.* a set (most often finite) of values it can take. The goal of a constraint solver is to find an instantiation of the variables to a value in its domain such that the constraints are satisfied. CP has been successfully applied to many different fields, such as biology, music [10], transportation, task scheduling [12], computer-assisted design, etc.

Depending on the application, the constraints and the variables can be very different. Constraints are first-order predicates in a language that typically includes equalities, inequalities, classical functions, and specific relations expressed with a mathematical definition (global constraints), for instance to count values in a set of variables. The domains of the variables can have many different types: real intervals, finite sets of integers, sets of vectors of bits, or even graphs. Each domain comes with its own set of constraints: reals feature continuous functions (*e.g.* exponential or logarithms), integers have all the arithmetic operations and many global constraints, vectors of bits have bitwise boolean operations. Several domains representations exist in the literature, and they can have an huge impact both on the expressivity and efficiency of the solver.

In another field, binary decision diagrams (BDDs) are a data structure representing efficiently sets of vectors of bits (bit-vectors). Representing sets of bit-vectors allows a user to represent many other data structures, such as set of integers or boolean functions. BDDs are well suited to many computations possible on bit-vectors, and some of these computations can't be done natively on constraint solvers yet. This is the example of bitwise operations on bit-vectors, or divisions and modulos on integers. BDDs have already been proven useful as an internal tool on some global constraints in [2]. A generalization of BDDs, called MDDs, have also been studied as a domain store for variables in [13].

The goal of this internship was to study the use of BDDs in a constraint solver as a representation of vectors of bits, and as a way to express new constraints. It leads to the following contributions:

- Design and implementation of a library for the computations on BDDs,
- Analysis of limited-width-BDDs and design of specific algorithms for them,
- Design of propagators for constraint solving: bitwise operations, div/mod operations, table constraints propagators (constraints defined in extension).

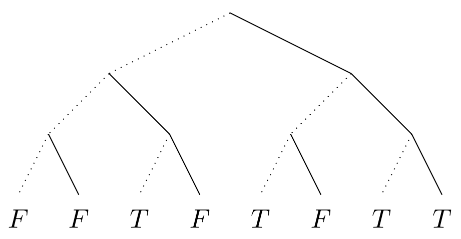
The implementation has been done in `OCaml`, and has been tested on a cryptographic problem.

2 State of the art

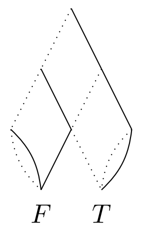
In this section, we define the core notions that we need afterwards.

2.1 The BDD data structure

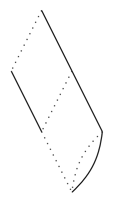
Binary decision diagrams (BDDs) are a data structure allowing representation of boolean functions, and efficient computations on them. It was first introduced by [7] and [1]. It has been formalized to the representation that is now commonly used by [3]. Since the notion of BDDs strongly relies on the way it is implemented, we have chosen to first give a formal definition which is immediately followed by a description of its implementation.



(a) An example of BDD



(b) The same BDD after reduction of equivalent subtrees


(c) The same BDD but fully reduced

2.1.1 Definition

Definition 1 (Labelled binary tree)**.** A *labelled binary tree* is a binary tree such that:

- the leaves are labelled T or F
- the edges are labelled 0 or 1 (and called 0-edges or 1-edges), and lead to the 0-child or 1-child,
- each node has at most one 0-edge and at most one 1-edge
- all the leaves T are at depth equal to the depth of the tree.

To represent labelled binary trees, we draw a tree where the root is at the top of the figure, the leaves at the bottom, 0-edges are drawn with dotted lines, and the 1-edges are drawn with plain lines. There is an example in Figure 1a. This representation of a tree grows exponentially as the depth grows, and therefore is not really suited for efficient computations.

To prevent the representation to grow too large, we introduce the notion of BDD that is a structure based on labelled binary trees, but with important properties that help to reduce the size of the representation. BDDs rely on the notion of equivalence of nodes, so that all equivalent subtrees are shared.

Definition 2 (Equivalence of subtrees)**.** Two nodes are equivalent if:

- they are both leaves and have the same value (F or T),
- they are both internal nodes, their 0-children are equivalent and their 1-children are equivalent.

To give a definition of BDDs, we first have to define a layer.

Definition 3 (Layer)**.** A *layer* is a set containing nodes or leaves. The *width* of the layer is the number of nodes it contains.

In the following, we are interested in lists of layers $(L_i)_{i\in\{0,\dots,n\}}$ such that for all $i \in \{0,\dots,n\}$, the children of each node in L_i is in L_{i+1}. Each node in the layered list has to be the child of a node in some previous layer (except the root that is the only element of L_0). That enforces the last layer L_n to contains only leaves. Such lists of layers represent labelled binary trees in a compact way. Layers represent each depth of the BDD.

Definition 4 (BDD)**.** A list of layers (with layers $L_i, i \in \{0,\dots,n\}$) is a *BDD* if:

(i) $|L_0| = 1$, and its node is called the root of the BDD,

(ii) each node $t \in L_i$ has one 0-edge and one 1-edge that link to x and y in L_{i+1}. We note $t = Node(x, y)$ to say that t is a node whose 0-child is x and 1-child is y,

(iii) $T \in L_n$, and $\forall i \in \{0, \ldots, n-1\}, T \notin L_i$: the terminal leaf T is only on the last layer,

(iv) $\forall x, y \in L_i$ x and y are not equivalent (ensuring maximal sharing of subtrees).

The *width* of the BDD is the maximum width of the layers, and is noted $\omega(B)$ (when the BDD is B).

The *size* of the BDD is the number of nodes and leaves it contains, and is noted $|B|$.

The *depth* of the BDD is the number of layers minus one.

Remark. *The definition enforces the BDD to be reduced, in the literature, these BDDs are called ROBDDs (for reduced and ordered BDDs), and in the following we simply use the notation BDD because they will always be reduced. The fact that these BDDs are ordered is important when representing boolean functions or sets of integers, but is not relevant for the general definition.*

This definition is more restrictive than the definition of labelled binary tree because the T leaf is imposed to be on the last layer. The last point of the definition ensures maximal sharing of the subtrees. Note that a BDD represents a labelled binary tree as defined above, yet it is not a tree anymore. It can be seen as a directed acyclic graph (DAG). An example can be seen on Figure 1b where the equivalent subtrees have been merged (some nodes points to the same subtree).

Reduction of paths to F One last rule that we add to the representation is the reduction of paths to F. Intuitively, paths from the root to T represent the elements that we want to store, whereas paths to F represent elements that we do not want to store. The idea is to reduce paths that are sure to lead to an F leaf. We add a reduction rule that transforms nodes $t = Node(F, F)$ into the the F leaf. Doing this recursively removes edges and gives a smaller representation of the BDD. This is shown in Figure 1c, where we do not write the edges that lead to an F leaf anymore. Thus, we can omit to write the T leaf, because all the paths represented lead to T.

The fact that a BDD is reduced is not tied to its representation, but more to its implementation. To construct reduced BDDs, we specify the two 0 and 1 children, and give them to a function that ensures the sharing of equivalent subtrees. This function is defined in next section.

2.1.2 Implementation

Despite the definition that involves layers and labels, the implementation of a BDD package can be done quite easily.

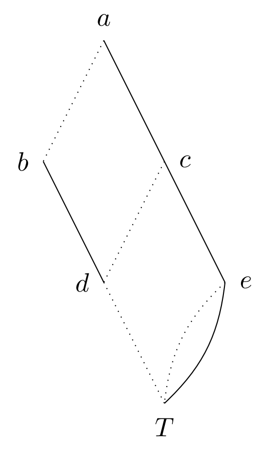
The main type of BDDs is an enumerated type where the BDD can be either a leaf (T or F) or a node with two parameters that are the 0-child and the 1-child, marked $N(a, b)$ where a is the 0-child, and b is the 1-child.

The way to deal with the reduction of equivalent subtrees is to have a global hash table that stores all the subtrees that we encountered (not representing equivalent subtrees) [1]. The idea is the following: every time we create a new BDD (or simply a subtree), we first check if the BDD is already in the hash table, in this case we return it, otherwise we create it, add it to the table, and return it. The key in the table for each BDD is the couple (a, b) when the BDD is $Node(a, b)$. This hash table is implemented through a function BDD_OF(a, b) that has to be used instead of the constructor $N(a, b)$. This function is shown in Algorithm 1 and is the core of the implementation of a BDD library. Equivalence is now tested through structural equality, because at each time, there is only one reference stored in memory for the subtrees.

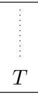
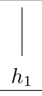
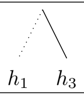
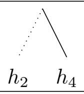
[1] A video showing this representation can be seen at `https://www.college-de-france.fr/site/gerard-berry/course-2016-03-09-16h00.htm`

```
CREATE($global\_hash$)
Function BDD_OF($a, b$)
    Data: The two children $a$ and $b$ of a node
    Result: The BDD $Node(a, b)$, with sharing of equivalent subtrees
    if $(a, b) \in global\_hash$ then
        return FIND($global\_hash, (a, b)$)
    else
        $new\_ref \leftarrow Node(a, b)$
        ADD_TO_HASH($global\_hash, (a, b), new\_ref$)
        return $new\_ref$
```

Algorithm 1: Ensuring sharing of equivalent subtrees



(a) Example of BDD with names for each nodes

name in the tree	reference in the table	key in the table	graphical view
-	h_0	F, F	F
d	h_1	T, F	
b	h_2	F, h_1	
e	h_3	T, T	
c	h_4	h_1, h_3	
a	h_5	h_2, h_4	

(b) Values stored in the hashtable

Table 1: Example of reduced BDD with the associated hashtable

Reduction of paths to F To reduce nodes that only lead to F, we simply add an entry in the hash table: we add the value F with the key (F,F). This way, every time we want to create a BDD from the children F and F, the hash table finds this entry, and return F.

This implementation of BDDs ensures maximal sharing of their subtrees. This has a great influence on all the algorithms we develop in the following sections, as they rely on the function BDD_OF. This function guarantees the maximal sharing of subtrees property, a property that is needed to ensure a reasonable complexity, and even, in some cases, the correctness of the algorithms.

On Table 1 is an example of what is stored in the hash table when creating a BDD. The BDD is traversed by going from the root to the leaves, and building the hash table from the leaves. In practice the hash table contains only the two middle columns of Table 1b, the first and last columns are here for a better understanding of the objects. The important data of this hash table is the key and the associated BDD.

2.1.3 BDDs as abstractions

Now that we have a reduced way to represent BDDs, we use them as an abstraction of objects. BDDs were first used to represent boolean functions in [3]. The way to represent objects is by

```
1 Function BDD_OF_BITVECTSET($B$)
      Data: A set of bit-vectors $B$
      Result: The BDD representing the set of bit-vectors
2     if $B = \emptyset$ then
3         return $F$
4     else if $B = \{\epsilon\}$ then
5         return $T$
6     else
7         Let $B^0$, $B^1$ such that $B = 0 \cdot B^0 \cup 1 \cdot B^1$
8         return BDD_OF(BDD_OF_BITVECTSET($B^0$), BDD_OF_BITVECTSET($B^1$))
```

Algorithm 2: Generating the BDD representing a set of bit-vectors

using the paths that lead to the T leaf. A path is indexed by a list of bits because each edge taken is indexed by 0 or 1. These lists of bits, called bit-vectors, are the core objects that are represented by BDDs.

Definition 5 (Bit-vectors and operations)**.** A *bit-vector* on $n > 0$ bits is an element of $\{0,1\}^n$. We denote by b_i the i-th bit of b a bit-vector for $i \in \{0,\ldots,n-1\}$. The *length* of a bit-vector b on n bits is n and is noted $|b|$. We denote by $\cdot$ the concatenation: if b^1 and b^2 are bit-vectors on n and m bits, then $b' = b^1 \cdot b^2$ is the bit-vector on $n+m$ bits defined by

$$\forall i \in \{0,\ldots,n+m-1\}, b'_i = \begin{cases} b^1_i & \text{if } i \leq n \\ b^2_{i-n} & \text{otherwise} \end{cases}$$

The concatenation between a bit-vector and a set of bit-vectors is defined by $b \cdot B = \{b \cdot b' | b' \in B\}$. The concatenation between two sets of bit-vectors is defined by $B \cdot B' = \{b \cdot b' | b \in B, b' \in B'\}$

Remark.

- *We always consider sets of bit-vectors with the same length: if B is a set of bit-vectors, then $\forall b^1, b^2 \in B, |b^1| = |b^2|$. We call the length of the set the length of its elements.*
- *We can extend the definition of bit-vectors to represent empty vector. The empty bit-vector is noted ϵ.*

The representation of sets of bit-vectors relies on the following property.

Property 1. *Let B be a set of non empty bit-vectors of length n. Then there exist a unique couple of sets of bit-vectors (B^0, B^1) of length $n-1$ such that*

$$B = 0 \cdot B^0 \cup 1 \cdot B^1.$$

With this property in our hands, we can give a simple procedure to generate a BDD from a set of bit-vectors. Algorithm 2 shows this function.

The BDD output by this function can be seen as follows. To get the set of bit-vectors associated to a BDD, one simply have to look at all the paths from the root to the T leaf. A bit-vector is represented by a path from the root to the T leaf in the BDD. For each path, mark down the indices of the edges traversed, this gives a bit-vector, and doing this for each path of the BDD gives the set of bit-vectors. The function that generates the set of bit-vectors associated to a BDD is called the concretization function and is defined as follow:

Definition 6 (Concretization function)**.** The *concretization function* of the BDDs is the function $\gamma : BDDs \rightarrow \mathcal{P}(\{0,1\}^n)$ that takes as input a BDD and returns the set of bit-vectors it represents. It is defined by the following recursive definition:

- $\gamma(F) = \emptyset$

- $\gamma(T) = \{\epsilon\}$
- $\gamma(Node(a, b)) = 0 \cdot \gamma(a) \cup 1 \cdot \gamma(b)$

This concretization function and the function BDD_OF_BITVECTSET are inverse of each other.

Example. *We consider the example given on figure 1a. It is the example B we already saw, where the nodes are named from a to e (and the leaf T). Recall that the nodes that only have one outgoing edge actually have two edges, but one leads to F, so we do not represent it.*

Let's compute the sets represented by each node to find the set represented by the BDD:

- $\gamma(d) = 0 \cdot \{\epsilon\} \cup 1 \cdot \emptyset = \{0\}$
- $\gamma(e) = 0 \cdot \{\epsilon\} \cup 1 \cdot \{\epsilon\} = \{0\}$
- $\gamma(b) = 0 \cdot \emptyset \cup \gamma(d) = \{10\}$
- $\gamma(c) = 0 \cdot \gamma(d) \cup 0 \cdot \gamma(e) = \{00, 10, 11\}$
- $\gamma(a) = 0 \cdot \gamma(b) \cup 1 \cdot \gamma(c) = \{010, 100, 110, 111\}$

Finally, the set represented is $\gamma(B) = \gamma(a) = \{010, 100, 110, 111\}$

The following theorem states an important property of the BDDs.

Theorem 1. *[3] The representation of a set of bit-vectors as a BDD is unique.*

Thus, there is a unique BDD corresponding to a given set of bit-vectors. Due to their nature of decision trees, BDDs are really a structure representing bit-vector sets. There is a one-to-one equivalence between a path in the BDD and a bit-vector.

Set of integers To represent integers as bit-vectors, we can simply use the binary representation. The only choice we have to make is the ordering of the bits (big endian, little endian, or a custom ordering). In the following the integers are represented with the high weight bits first.

Remark. *The choice of the ordering is up to the user of the BDD. Recall that we said that our BDDs are in facts reduced-ordered-BDDs. The ordering comes from here, because we have to choose an ordering of the bits representing the integer.*

The representation as a set of bit-vectors can be used to generate a BDD. The user has to remember what ordering was used because it is not stored in the BDD, neither in the bit-vector set.

2.2 Constraint programming

Constraint programming is a paradigm where the problem is stated using constraints defined with variables. The framework is the following:

Definition 7 (Constraint Satisfaction Problem)**.** A *constraint satisfaction problem* (CSP) is given by $(X, \mathcal{D}, C)$ where X is a set of variables, $\mathcal{D}$ is a function associating each variable to its domain (the accepted values for this variable), and C is the set of constraints.

Constraints can be defined in intention (with formulas, operations, equalities, logical operators) with a high level language, or in extension by enumerating allowed solutions (in this case, they are called *table* constraints).

The goal of constraint solvers is to find an instantiation (assignment of a value to each variable), or all the instantiations, such that the constraints are all satisfied. This is often done through a "propagate and search" method. These algorithms have two main components:

- Propagation: the goal of this step is to remove from the domains of the variables the values that are not part of the solutions of the problem. This is done with the notion of consistency. By looking at each constraint, one can find if some values are never in any solution of the problem, and therefore are not consistent and should be removed.
- Search: as propagation is in general not sufficient to find a solution/all the solutions, we have to search in the domain for solution. This is often done by splitting the domain of our variables in two (or more), and searching recursively for the solutions.

The propagation techniques are based on the notion of consistency. The main consistency property is arc-consistency.

Definition 8. Arc-consistency A constraint $C(X_1, \ldots, X_n)$ where $X_i \in \mathcal{D}_i$ is *generalized arc consistent* if $\forall i \leq n, \forall x \in \mathcal{D}_i$, $\exists v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \in \mathcal{D}_1 \times \ldots \times \mathcal{D}_{i-1} \times \mathcal{D}_{i+1} \times \ldots, \mathcal{D}_n$ such that $(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_n)$ satisfies the constraint C

There are other weaker and other stronger notions of consistency (such as bound consistency or path consistency).

These two steps of propagation and search are interleaved such that we try to eliminate inconsistent values when it is possible during the search.

A contribution of this internship is the design of the BDD domain, with specific propagation and the search algorithms, allowing us to integrate this new domain into a constraint solver.

3 A new type of BDDs: Limited-width-BDDs

Even though we share a lot of information by using reduction of equivalent subtrees, the number of nodes in a BDD can still be exponential (depending on the depth). It has been shown in [3] that the BDD representing some boolean functions on n variables contains at least $2^{n/8}$ vertices, and we can extend this result to sets of bit-vector.

This exponential representation might give computational issues, so we have to come up with a new notion of BDD that has a polynomial representation. This notion is the one of limited-width-BDDs.

Definition 9 (Set of limited-width-BDD)**.** The set of limited-width-BDDs of width w is the set of BDDs of width smaller or equal to w. Such a BDD is calld a w-limited-width-BDD

The number of nodes in a w-limited-width-BDD of depth n is bounded by nw, and therefore is polynomial in the depth.

As a corollary to Theorem 1, the width of the BDD that represents a given set is determined by the set and the ordering. Hence, it cannot be bounded by some parameter. In order to deal with BDDs with a bounded width, we have to over-approximate the set that we want to represent. Given a set of bit-vectors B, and a maximum width w, the goal is to find a w-limited-width-BDD that represents a super-set of B. Knowing the width of the BDDs that we are using can give us a better knowledge of the complexity of the functions in our library.

3.1 Generating a limited-width-BDD

Limited-width-BDDs are a great way to reduce the size of the BDDs. The counter-part is that we cannot represent all the sets of bit-vectors. The goal here is to generate a limited-width-BDD that over-approximates a given BDD (or a set of bit-vectors), with the smallest possible over-approximation. This is not possible in practice, but we can use clever heuristics to try to minimize the over-approximation.

```
1  Function LIMITED_OF_BDD($B, w$)
       Data: A BDD $B$ (of depth $n$) and a desired width $w$
       Result: A $w$-limited-width-BDD over-approximating $B$
2      Let $r$ be the root of $B$
3      $Q \leftarrow \{r\}$
4      for $i \in \{0, \dots, n-1\}$ do
5          while $|Q| > w$ do
6              $nodes\_to\_merge \leftarrow$ CHOOSE($Q$)
7              $t \leftarrow$ MERGE($nodes\_to\_merge$)
8              $Q \leftarrow (Q \setminus nodes\_to\_merge) \cup \{t\}$
9          $Q' \leftarrow \varnothing$
10         for $t \in Q$ do
11             Let $a, b$ such that $t = Node(a, b)$
12             $Q \leftarrow Q \cup \{a, b\}$
13         $Q \leftarrow Q'$
14     return The BDD after merging the nodes
```

Algorithm 3: Generating a limited-width-BDD over-approximating the given BDD

Here we are given a BDD, and we want to generate a BDD that over-approximates it and that has a width bounded by a given constant. Algorithm 3 shows how to generate a limited-width-BDD from a BDD.

The algorithm relies on a MERGE function that merges a list of nodes of the same layer into one. We suppose here that this function also updates the links with the parents of the nodes (the new node should have the same parents as the initial nodes in the list). The algorithm also relies on a function CHOOSE that chooses at least two nodes to merge from a set of nodes. There might be many different heuristics to choose the nodes, and they are implemented in this function. The algorithm simply goes through each layer (from the top to the bottom), and when a layer is too wide, it merges nodes until the layer has the desired width.

3.2 Computation of the over-approximation

When using the function to generate a limited-width-BDD, the returned BDD is an over-approximation of the initial BDD. The goal of the heuristics is to get the smallest over-approximation possible. A contribution of the internship is the introduction of the notion of *merge value* that is useful to evaluate the cost of merging two nodes.

Definition 10 (Merge value)**.** Let B be a BDD, and u, v be two nodes in the same layer in B. The *merge value* (noted $mv(u,v)$) is the number of bit-vectors that are added in B when merging the two nodes u and v.

The merge value is exactly the notion of cost of merging two nodes that we want. Now we want a way to compute it. We note by $\cup$ the merge of two nodes (because in fact we do an union of the associated nodes).

Theorem 2. *Let B be a BDD of root r, and u and v two nodes of the same layer in B. Let p_u (resp. p_v) the number of paths that go from r to u (resp. v). The merge value is then*

$$\begin{aligned} mv(u,v) &= p_u|\gamma(v) \setminus \gamma(u)| + p_v|\gamma(u) \setminus \gamma(v)| \\ &= p_u|\gamma(v)| + p_v|\gamma(u)| - (p_u + p_v)|\gamma(u) \cap \gamma(v)| \end{aligned}$$

The proof of this theorem is given in Appendix A.1. This merge value can be used in the heuristics choosing the nodes to merge when limiting a BDD to a given width. The natural simple heuristic that comes to mind is to merge the two BDDs that have the smallest merge value, this way we try to minimize the over-approximation we are doing. This method is still a heuristic because we are only looking locally to minimize the over-approximation, and not globally.

4 A library for a BDD domain in CP

The goal of this internship was to study the use of BDDs in a generic way in a constraint solver. As a first step, we defined and implemented a library allowing us to represent domains (of variables inside a CP solver) as BDDs. This section describes the basic operations that can be performed within this library. They are useful to implement the constraints primitives needed in a solver presented in section 5. All the algorithms are written in pseudo-code in Appendix B.

4.1 Set operations

As BDDs are used to represent sets of values (integers or bit-vectors), set operations are needed to perform many computations. The basic set operations are intersection, union, and difference. The algorithm to compute these three operations are almost the same: when the two considered BDDs are nodes $Node(a_1, b_1)$ and $Node(a_2, b_2)$ there are recursive calls on (a_1, a_2) and an other on (b_1, b_2) (like in an usual traversal of two trees). The only cases that change are the initial cases when one or two of the BDDs are leaves. The operations are straightforward to implement using recursion.

For these algorithms, we always suppose that the BDDs have the same depth. This could be checked by the algorithm, but for sake of simplicity we do not do it as it will be ensured by our framework.

Remark. *These functions do not preserve the width of the BDDs. It is possible that the width of $B_1 \cap B_2$ is bigger than the width of B_1 and B_2. This is problematic because these operations are used a lot during other computations. Section 5.1 shows algorithms that preserve width.*

4.2 Bitwise boolean operations

4.2.1 Direct operations

In [3], the BDDs were used to represent boolean functions. The goal of using BDDs was to be able to compute combinations of these functions using logical operators, *e.g.* given the BDD representation of f and g, compute the representation of $h = f \wedge g$. The goal here is different, we want to compute *bitwise* operations. Bitwise operations are the operations on boolean numbers extend to bit-vectors.

Definition 11 (Extension of boolean operations to bit-vector and sets of bit-vector)**.** Let $\Box$ be a boolean operation (*e.g.* $\vee, \wedge$, or $\oplus$). Let b^1 and b^2 be two bit-vectors of size n. Then we extend the definition of $\Box$ to bit-vectors as:

$$b^1 \Box b^2 = b^1_0 \Box b^2_0 \cdot \ldots \cdot b^1_{n-1} \Box b^2_{n-1}$$

This definition can also be extended to sets. Let B_1 and B_2 be two sets of bit-vectors of size n. Then

$$B_1 \Box B_2 = \{b^1 \Box b^2 | b^1 \in B_1, b^2 \in B_2\}$$

Algorithm 4 presents the bitwise `xor` $\oplus$ computation. The `or` $\vee$, `and` $\wedge$ and `not` $\neg$ are presented in Appendix 9.

These algorithms are different from the set operations because all the combinations between the children have to be explored. Given an operator $\Box$, the i-th child of the first BDD and the

```
1  Function XOR($B_1, B_2$) notation $B_1 \oplus B_2$
       Data: Two BDDs $B_1$ and $B_2$ of same depth
       Result: The xor $B_1 \oplus B_2$
2      if $B_1 = T$ and $B_2 = T$ then
3          return $T$
4      else if $B_1 = F$ or $B_2 = F$ then
5          return $F$
6      else
7          Let $a_1, b_1$ such that $B_1 = Node(a_1, b_1)$
8          Let $a_2, b_2$ such that $B_2 = Node(a_2, b_2)$
9          $a \leftarrow$ XOR$(a_1, a_2) \cup$ XOR$(b_1, b_2)$
10         $b \leftarrow$ XOR$(a_1, b_2) \cup$ XOR$(b_1, a_2)$
11         return BDD_OF$(a, b)$
```

Algorithm 4: Computing the bitwise `xor` $\oplus$ of two BDDs

$X \wedge Y$	Y	X
0	0	$\{0,1\}$
0	1	0
1	0	$\perp$
1	1	1

(a) Inverse boolean `and` $\wedge$

$X \wedge Y$	Y	X
0	0	0
0	1	$\perp$
1	0	1
1	1	$\{0,1\}$

(b) Inverse boolean `or` $\vee$

Table 2: Inverse boolean operations truth tables'

j-th child of the second ($i, j \in \{0,1\}$) add values in the ($i\Box j$)-th child of the result. Depending on the boolean operator, we may have to do the union of the recursive results to get the final child.

4.2.2 Inverse bitwise operations

This section presents how to modify the previous algorithm to compute inverse bitwise operations. Inverse bitwise operations are the computations of the possible values of X given Y and $X \Box Y$. For the `xor`, it is simple because we have the property that $X \oplus Y = Z \Leftrightarrow X \oplus Y \oplus Z = 0 \Leftrightarrow Y \oplus Z = X$. The computation is harder for the `and` $\wedge$ and the `or` $\vee$ operations. In Table 2 are the tables that show the values that the input can take given the output (and the other input). The symbol $\perp$ in the table means that there is no possibility for Y to get the desired output. Using these (modified) truth tables, it is possible to give an algorithm just like the other bitwise algorithms that give all the bit-vectors that can be output of the inverse bitwise operations.

4.3 Concatenation

Keeping up with all the computations that are possible on sets of bit-vectors, we now present how to compute the concatenation of two BDDs. With the representation we have, computing the concatenation of B_1 and B_2 is easy because we simply have to replace all the T leaves in B_1 with B_2. The algorithm is a traversal of B_1 where we reconstruct it by replacing T with B_2. It is presented in Appendix 6.

4.4 Extraction of sub-strings

The last computation possible on sets of bit-vectors that we want to extend to BDDs are extraction of sub-bit-vectors(prefixes or suffixes of some bit-vectors). Extraction is the opposite of concatenation, and can be seen as its inverse operation.

Definition 12 (Prefixes and suffixes of bit-vectors)**.** Let b be a bit-vector. The *prefix* $Pref_k(b)$ of length $k < n$ of b is $b_0 \dots b_{k-1}$ and its *suffix* $Suff_k(b)$ of length k is $b_{n-k} \dots b_{n-1}$. This definition can be extend to sets of bit-vectors.

We want to extend the operations of computing the set $Pref(B)$ or $Suff(B)$ to BDDs. These operations come handy when computing modulo on integers, because with the binary representation, computing the result modulo 2^k is exactly computing the suffix of the bit-vector representing the integer (if it is represented with the highest bits first).

The algorithms are done by simply replacing the nodes at a certain depth by the leaf T(for the prefixes), or extracting all the nodes at a given depth (for the suffixes). For the prefixes, the output is a BDD, but for the suffixes the output is a set of BDDs whose union is the set of suffixes. This union is not computed here, and in practice it is not necessary to compute it because we are able to use the set of BDDs representing the suffixes as it is. The algorithms are presented in Appendix 8.

4.5 Splitting

The operation of splitting is not an usual computations on sets of bit-vectors, but is necessary for the purpose of constraint solving. It can be seen as partitioning the BDD B into two disjoints and non empty BDDs B_1 and B_2 whose union is the original BDD. There are many ways to split a BDD, the easiest one is to choose a depth, and generate the BDD that contains only the 0-edges at this depth, and the one that contains only the 1-edges at this depth. A simple function splitting the BDD at a given depth is shown in Appendix 10. There are many possible variations to this algorithm. The first question is to know the depth at which we want to split. One simple heuristic is to choose the smallest depth (the highest in the BDD). One other is to choose the depth that minimizes the widths of the resulting BDDs.

4.6 Implementation

All the algorithms presented rely on the idea of traversal of tree (and maybe simultaneous traversal of two trees). Recall that the BDDs we are representing are sharing the equivalent subtrees to reduce memory usage. If we trace the execution of a function (for example the intersection), we remark that we compute many times the same intersection of nodes (that always results in the same output).

The solution to use the fact that subtrees are shared in the BDDs is to have a way to have a cache to store the result (to do memoization). All the function have a local cache to store the results that have already been computed. Algorithm 5 shows how to implement such a function that uses caching of already computed results.

All the algorithms that uses caching define a hash table that stores the results computed by this algorithm. There are many ways to implement these cache. The first would be to decorate the functions: implement the functions without caching, and create a function (called decorator) transforming them into cached functions. The other way (the one that I did) was to simply implement the cache system in each function. This was due to the fact that in `OCaml` the way to decorate functions is not simple (as it is in `Python` for example), and it would have required syntax extensions [2].

5 Propagation algorithms

The following subsections shows how to compute the propagation on BDDs. The first subsection shows how to modify the BDD representing the domain of a variable when we know what are

[2] An example of implementation is given in `https://github.com/ghilesZ/finding-memo`

```
1 CREATE($f\_hash$)
2 Function CACHED_F($B_1, B_2$)
      Data: Two BDDs $B_1$ and $B_2$
      Result: The computation of the function $f$ on $B_1$ and $B_2$ with caching
3     if $(B_1, B_2) \in f\_hash$ then
4         return FIND($f\_hash, (B_1, B_2)$)
5     else
6         $result \leftarrow$ F($B_1, B_2$)
7         ADD_TO_HASH($f\_hash, (B_1, B_2), result$)
8         return $result$
```

Algorithm 5: Caching of the computations

the allowed values, and the second subsection shows all the algorithms for specific constraints to remove inconsistent values.

5.1 Consistency

The goal of consistency is that given a constraint, we want to find all the values of a variable that are not in a solution of the constraint. When we have all these values, we can remove them from the domain of the variable, because they are inconsistent (they are never part of a solution, recall the definition of arc consistency in Definition 8).

Example. *Let's take a `xor` constraint C of the form $X \oplus Y = Z$ and the domains $\mathcal{D}(X) = \{000, 001, 110\}, \mathcal{D}(Y) = \{011, 100, 110\}, \mathcal{D}(Z) = \{000, 010, 011, 111\}$. We want to find all the values of the domain of Z that are consistent with respect to the constraint C. To do so, we compute the possible values of Z with respect to the constraint, meaning the possible values of the xor of X and Y. This set is $\mathcal{D}(X) \oplus \mathcal{D}(Y) = \{010, 011, 100, 101\}$. We find that the values* 000 *and* 111 *of the domain of Z is never the `xor` of values of X and Y, and therefore are inconsistent with respect to the constraint. To remove the inconsistent values from the domain of Z, we simply have to compute $\mathcal{D}(Z) \leftarrow \mathcal{D}(Z) \cap (\mathcal{X} \oplus \mathcal{D}(Y)$ to update the domain.*

This method of deleting values in domains of variables would work well on BDDs, but is not suited for limited-width-BDDs. This is due to the fact that the intersection of two BDDs may increase their width. We then have to come up with a method to delete values in the domain of variables (represented by limited-width-BDDs) that does not increase the width above the bound that was previously fixed.

5.1.1 From MDD-consistency

The notion that solves the issue of the increase of the width during consistency was first introduced on MDDs in [13]. When doing the consistency of a domain represented by a BDD B with respect to an other BDD B', we simply delete edges in B. To choose the edges that can be deleted, we have to find the edges in B that are not part of some paths in B', meaning that they are never involved in consistent paths (for consistent values).

The algorithm to impose consistency of B with respect to B' works as follow: traverse the two BDDs simultaneously (like in the computation of the intersection) and mark the edges of B that are in paths in B'. Then traverse once more B to delete the edges that have not been marked. The algorithm is presented in Appendix 10.

This algorithm ensures that the width is conserved (the width can only decrease). In facts this may be an issue because the more we decrease the width, the less expressive is the BDD. This issue is solved with the notion of refining.

Consistency with respect to a set An improvement to this algorithm that was found during the internship is the consistency with respect to a set. Here we are given a BDD M and a set S of BDDs, and we want to compute the consistency of M with respect to $\bigcup_{B \in S} B$. Instead of computing the union, we remark that the consistency with respect to a set can be formulated as: we want that each edge in the BDD M is in a path of at least one BDD $B \in S$. This way, we simply have to apply the TRAVERSE function for each BDD in the set (to mark all the possible edges), and then remove the edges that are not marked.

The advantage of this method is that if the operations are cached, we may win a lot of time by not doing twice the same computations. If the BDDs in the set come from the same original BDD (as it would often be the case), because of sharing in this BDD many operations are redundant (so do not do them thanks to caching).

5.1.2 Refining

Refining has also been introduced in [13] to increase the expressiveness of the BDD we are using. If we choose a bound on the width at the beginning of the program, we want this bound to be achieved, because we will be able to have a smaller over-approximation of the sets we are representing. The issue is that the more we apply the consistency, the more the width decreases.

To solve this issue the notion of refining is introduced. The idea is to choose a node, and split it so that the paths leading to this node are now leading to different nodes. Doing the consistency after refining may lead to a better consistency, because the two nodes are now distinct and can be treated separately.

This idea is great in theory but does not work well with the implementation I have, because splitting a node creates two equivalent nodes in the BDD, which is forbidden in reduced BDDs. Even more than forbidden, it is impossible to have equivalent nodes in my implementation because the global hash table ensures maximum sharing of equivalent subtrees, and is done automatically.

An other issue with refining is that choosing the node to split depends on the consistency that we want to do. In [13], the authors use some meta-data given by the `all_different` constraint to choose on which node to use refining. Also, one have to choose when to apply refining and when to apply consistency, maybe interleaving the two.

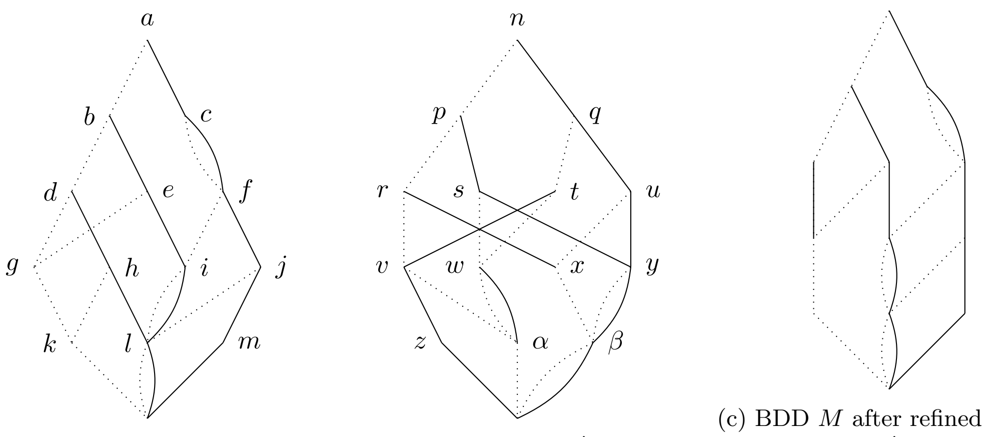
All these issues lead to the creation of a new algorithm that does the consistency and the refining at the same time. This algorithm always does refining when possible (when the width becomes smaller), and as it is done simultaneously with the consistency, it can use meta-data from the constraint to choose on which node to use refining.

5.1.3 Refined consistency

Here we present our solution to the issue we raised. The goal was to have an algorithm that takes the advantages of the algorithms for consistency that we presented, without having the disadvantages:

- intersection: computed the optimal consistency (removes all the inconsistent values), but increase the width,
- MDD-consistency: only decreases the width, but this is also its disadvantage because we loose expressiveness,
- refining: allows to increase the width as we want, but hard to choose the node to use, have to be interleaved with consistency, and can't be implemented easily with sharing of subtrees.

For this algorithm, there is one strict requirement that we have to ensure: the consistency algorithm has to be monotonous. This is needed for the termination of the algorithm of propagate



(a) BDD M (b) BDD M' (c) BDD M after refined consistency w.r.t. $\{M'\}$

Figure 2: Differences between the different consistency algorithms

and search: if the propagation adds values in the domain of the variable, we may create an infinite loop.

The algorithm we present is more powerful than a simple algorithm imposing the consistency of a BDD with respect to an other. It imposes the consistency with respect to a set of BDDs. We only present the algorithm imposing the consistency with respect to a set of BDDs, because it can be applied with a singleton to get back the previous consistency.

The algorithm works as follow: given the BDD M and a set S of BDDs, the algorithm generates the intersection layer by layer. When the layers are too big, it chooses two nodes to merge, merge them, and repeat while the width of the layer is too big. There are conditions on the nodes to merge, and we will see this condition on an example.

Algorithm on an example On Figure 2 are the BDDs M and M' with nodes marked. We compute step by step the refined consistency of width 4 of M with respect to $\{M'\}$. The computation starts with the first layer containing the node $(a, \{n\})$. Here the nodes contain the original node in M, and a set of nodes of M' (with which we do the consistency). The computation of the second layer is done by dividing each node into the 0-children and the 1-children. In this case, the 0-child of a is b and the 0-child of n is p. For the 1-children, we have c and q. This gives the second layer $\{(b, \{p\}), (c, \{q\})\}$. This layer still has the desired width (less than 4), so we compute the next layer. The next layer is $\{(d, \{r\}), (e, \{s\}), (f, \{t\}), (f, \{u\})\}$. This layer has width less than 4 so we keep on computing the next layer. The next layer is $\{(g, \{v\}), (h, \{x\}), (g, \{w\}), (i, \{y\}), (i, \{w\}), (j, \{v\}), (i, \{x\}), (j, \{y\})\}$. This set is too big, we have to merge nodes in it.

The condition on the merge of two nodes is that when merging the nodes (δ_1, Δ_1) with (δ_2, Δ_2), we need $\delta_1 = \delta_2$. This is due to the fact that we want to have the monotonicity of the consistency algorithm. If we merge nodes coming from the same original node in M, we cannot add values in the BDD that were not in it beforehand. Here, we iteratively choose two nodes in the constructed layer to merge. We start by choosing $(g, \{v\})$ and $(g, \{w\})$, and merging these two nodes give $(g, \{v, w\})$. We keep on with $(i, \{y\})$ and $(i, \{w\})$ to get $(i, \{y, w\})$. We then choose $(j, \{v\})$ and $(j, \{y\})$ to get $(j, \{v, y\})$, and finally $(i, \{y, w\})$ (that was just constructed) and $(i, \{x\})$ to get $(i, \{x, y, w\})$. Finally, the reduced layer is $\{(g, \{v, w\}), (h, \{x\}), (i, \{w, x, y\}), (j, \{v, y\})\}$. The next layers are computed using the same method. To give an example of children with these kind of nodes, we choose $(j, \{v, y\})$. The 0-child of j is l, and the 0-children of $\{v, y\}$ are $\{\alpha, \beta\}$. The 1-child of j is m, and the 1-children of $\{v, y\}$ are $\{z, \beta\}$. This give the associated nodes $(l, \{\alpha, \beta\})$ and $(m, \{z, \beta\})$. The final output of the algorithm after all the computations is given in Figure 2c

This algorithm really computes the intersection layer by layer, but in practice, with the functional implementations, many details are hard to deal with. The hardest part is to remember the links between the nodes. This algorithm helps to keep the width that has been chosen for the BDD. It is an improvement over the previous method because the nodes that are refined are chosen during the consistency, and not with an external algorithm.

5.2 Propagation

After the section about consistency, it is time to focus on constraints. In this subsection, we present how to reduce the domains of the variables that are involved in some constraints. The functions that reduce the domains of variables given a constraint are called propagators. Here we suppose that the domains $\mathcal{D}(X)$ of the variables (X in this case) is a BDD.

5.2.1 General propagators

Here we present the propagator to impose the consistency on constraints with bitwise operations. This is done on the example of an and constraint $X \wedge Y = Z$. The first step is to compute the allowed values of X, Y and Z with respect to the constraint. To do this, we use the bitwise (and inverse bitwise operators):

- $allowed_Z = \text{AND}(X, Y)$
- $allowed_X = \text{INVERSE\_AND}(Z, Y)$
- $allowed_Y = \text{INVERSE\_AND}(Z, X)$

Then, we update the domains of X, Y and Z: $\mathcal{D}(X) \leftarrow \textsc{Consistency}(\mathcal{D}(X), allowed_X)$ (and equivalently for Y and Z). The consistency function can be either the function implementing the improved consistency, or a simple intersection (if we don not look at the width).

5.2.2 Modulo and division propagators

The modulo and division propagators work differently. Defined on integers represented with the highest weighted bits first, getting the $X \mod 2^k$ is extracting the last k bits, and getting the $X//2^k$ is extracting the first $n - k$ bits. On BDDs of depth n, the constraints can be expressed as $X \equiv Y \mod 2^k, k \in \{1, \ldots, n\}$ and $X//2^k = Y$ (where $depth(Y) = n - k$). The propagators associated to these constraints use the extraction of sub-bit-vectors and concatenation. More precisely, the `mod` propagator uses suffixes and the `div` propagator uses prefixes.

`mod` propagator: Let the constraint $X \equiv Y \mod 2^k$, we show the propagation on X (this is symmetric for Y). We first start to compute all the possible values for $X \mod 2^k$. These values can be computed using suffixes: $mod_X = \textsc{Suff}(Y, k)$. This gives a set of BDDs whose union represent the possible values for $X \mod 2^k$. Then to do the consistency, we need BDDs of the same depth as X. We first compute the BDD M representing $\{0, 1\}^{n-k}$ (this can be done faster by remarking that this BDD is a chain of 0 and 1-edges linking to the same next node). Then we concatenate this BDD with all the BDDs in the set mod_X: $allowed_X = \{M \cdot M' | M' \in mod_X\}$. We finally do the consistency of X with respect to the set of BDDs $allowed_X$.

div propagator: The div propagator works the same way, except that we use the prefixes instead of the suffixes of the BDDs.

5.2.3 Table constraints

Table constraints are constraints defined in extension by the list of allowed tuples. Here we present how to do the consistency on table constraints using BDDs. We suppose that the table constraint is defined on two variables X and Y of both depth n. Let the constraint $Table(X,Y)$, where we have access to a set T containing all the tuples (x,y) such that the instantiation $X=x$ and $Y=y$ is allowed by the constraint. We construct the set of bit-vectors B by $\{x{\cdot}y|(x,y) \in T\}$. The BDD M representing this set is the one representing the table constraint. To do propagation on X and Y, we simply remark that the allowed values for X are the prefixes of M and the allowed values of Y are the suffixes of M. Then, we can express the propagator with a new constraint $X \cdot Y = M$, and use the concatenation, prefixes and suffixes to do the propagation.

6 Practical test case: AES cryptanalysis

To test the implementation, we needed problems on bit-vectors that has constraints using bitwise operations. There are recent papers ([9, 5, 6, 4]) that found a way to express a cryptographic problem under the formalism of constraint programming. This framework is really great for our tests because it involves fixed size bit-vectors, and different constraints defined sometimes bitwise, and sometimes by harder functions. It allowed me to test the implementation on real instances, and trying to optimize the computations helped to find new ideas of improvements.

The background behind the problem is omitted because due to space requirements. It comes from the differential analysis of the AES algorithm.

The constraints appearing in this problem are the following:

- Xor constraints: bitwise `xor` between three variables: $X \oplus Y = Z$
- Mix column [3]: operation on 4 bit-vectors, outputting 4 bit-vectors applying xor and logical shifts
- S-box [4]: a known function $SB : \{0,1\}^8 \to \mathcal{P}\left(\{0,1\}^8\right)$, and the constraints associated are $Y \in SB(X)$
- Equality or difference to 0^8

The Xor and Mix column constraints were implemented using the usual bitwise operations. The S-box constraint was implemented using table constraints.

The results are that we are able to find the solutions to the problems for small instances, but for bigger instances, the solver takes too long to find the solutions. In [4], the solver Choco [11] was used and was able to return solutions in a reasonable time. The goal of testing our implementation was not really to compare the times to solvers that have been existing for a long time, but more to show that the implementation was able to raise solutions.

One interesting fact that I saw during the experiments is that on the examples, it was faster to simply use general BDDs instead of limited-width-BDDs. This is probably due to the fact that in the problem, the BDDs had always depth 8, so the cost of keeping the small width of the BDDs was higher than the cost of doing computations on BDDs with big width. This makes us want to test our implementation on BDDs of higher depth to see the behaviour of the algorithms on bigger BDDs.

7 Properties of BDDs

In this section, I detail some properties that I found useful on BDDs. These are contributions of my internship.

[3] `https://en.wikipedia.org/wiki/Rijndael_MixColumns`
[4] `https://en.wikipedia.org/wiki/Rijndael_S-box`

7.1 Equivalences

The simplest class of limited-width-BDDs possible is the one of 1-limited-width-BDDs. This class might seems too restrictive, but it is still interesting and has already been studied, but not under the formalism of BDDs.

Here we introduce the notion of bit-vector domain as a representation of sets of bit-vectors. This representation was first introduced by [8] as a way to represent a set of bit-vector that supports bit operations that can be transformed into propagation algorithms to solve constraint problems on bit-vectors.

Definition 13 (Bit-vector domain)**.** A *bit-vector domain* of size n is a couple of bit-vector (l, u) of size n. The couple (l, u) represents the set of bit-vector

$$B = \{b | l_i \leq b_i \leq u_i, i \in \{0, \dots, n-1\}\}$$

Remark.

- *The bit-vector domain can be seen as the equivalent of interval domains on integers. It contains all the bit-vectors that are between two bit-vectors, where the order is defined bitwise. The definition of intervals is really equivalent to the one of bit-vector domains:* $[a, b] = \{x | a \leq x \leq b\}$
- *This definition does not prevent from having* $l_i > u_i$*. In this case, the domain is inconsistent and represent the set of bit-vector* $\emptyset$*.*

The second structure that is already known is the boolean abstract domain.

Definition 14 (Boolean domain)**.** The set of boolean domains is noted $\mathcal{B}^{\#} = \{\bot, 0, 1, \top\}$. A set $S \subseteq \{0, 1\}$ can be transformed into a boolean domain $s^{\#}$ and reciprocally with the abstraction $\alpha : \mathcal{P}(\{0,1\}) \to \mathcal{B}^{\#}$ and concretization $\gamma : \mathcal{B}^{\#} \to \mathcal{P}(\{0,1\})$ functions:

$$\alpha(S) = \begin{cases} \bot & \text{if } S = \emptyset \\ 0 & \text{if } S = \{0\} \\ 1 & \text{if } S = \{1\} \\ \top & \text{if } S = \{0,1\} \end{cases}, \gamma(s^{\#}) = \begin{cases} \emptyset & \text{if } s^{\#} = \bot \\ \{0\} & \text{if } s^{\#} = 0 \\ \{1\} & \text{if } s^{\#} = 1 \\ \{0,1\} & \text{if } s^{\#} = \top \end{cases}$$

These two functions can be extended to cartesian products, with $\widetilde{\alpha} : \mathcal{P}(\{0,1\})^n \to \left(\mathcal{B}^{\#}\right)^n$ and $\widetilde{\gamma} : \left(\mathcal{B}^{\#}\right)^n \to \mathcal{P}(\{0,1\})^n$

The definitions of bit-vector domain, boolean domain and limited-width-BDDs may seem unrelated, and in facts they were introduced to solve different problems, but they represent the same object.

Theorem 3. *[Equivalence between domains] The following domains represent the same sets of bit-vectors of size n:*

i $(\mathcal{P}(\{0,1\}))^n$

ii the bit-vector domain of size n

iii cartesian product of boolean domains: $\left(B^{\#}\right)^n$

iv 1-limited-width-BDDs of depth n (and F).

The proof of this theorem is given in Appendix A.2

Remark. *The set of bit-vectors represented by an element* $(B_1, \dots, B_n) \in (\mathcal{P}(\{0,1\}))^n$ *is* $\{b_0 \dots b_{n-1} | b_i \in B_i, i \in \{0, \dots, n-1\}\}$

7.2 Enumeration of BDDs

Knowing the number of limited-width-BDDs of certain width may give an insight on the power of expressiveness of the given class of limited-width-BDDs. BDDs of unbounded width can represent any boolean function (or equivalently any set of bounded integers), but limited-width-BDDs cannot. The more BDDs of given width exist, the more this class of limited-width-BDDs is able to represent a set of integers without a big over-approximation.

Unbounded width BDDs can represent any set of integers on n bits. There are 2^n integers on n bits, then there are 2^{2^n} set of integers on n bits, so there are 2^{2^n} BDDs of depth n.

Width at most 1 The class of 1-limited-width-BDDs is the smallest class of limited-width-BDDs. To compute the number of BDDs of width exactly one, we remark that there is only three possibilities for the first node (the root): either it has only one 0-edge, or only one 1-edge,or a $0-1$-edge. Then we are reduced to the case of the depth $n-1$. The recurrence formula with a_n the number of BDDs of width exactly one is $a_n = 3a_{n-1}$. The number of 1-limited-width-BDDs is 3^n (because F has depth zero).

Width at most 2 To compute the number of 2-limited-width-BDDs, we have to introduce some notations. Let a_n be the number of 2-limited-width-BDDs of depth n, and b_n the number of ways to start a 2-limited-width-BDD such that at depth n there are two nodes. Then we have a recurrence relation to compute these two sequences (and knowing $a_0 = 1$ and $b_0 = 0$:

$$\forall n \geq 1, \begin{pmatrix} a_n \\ b_n \end{pmatrix} = \begin{pmatrix} 3 & 3 \\ 2 & 22 \end{pmatrix} = \begin{pmatrix} 3 & 3 \\ 2 & 22 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

The computations that led to this matrix are shown in appendix A.3.

The general term of the sequence a_n can be computed by diagonalizing the matrix. The eigenvalues of the matrix are $\lambda_1 = \frac{25+\sqrt{(385)}}{2} \approx 21.7$ and $\lambda_2 = \frac{25-\sqrt{(385)}}{2}$, so the general term is $a_n = \mathcal{O}\left(\lambda_1^n\right)$.

The fact that the formula for the number of 2-limited-width-BDDs was hard to compute makes us think that computing a closed formula (or even an equivalent) for the number of k-limited-width-BDDs depending on the depth would be too hard. The method used for width two can be generalized to every other width, but there is too much work to be done by hand.

7.3 Maximum width

The width of the BDDs we are using during our computations is an important factor of the complexity of the program, so it is interesting to know what can be its value in the worst case.

Obviously at depth k there can be at most 2^k nodes because each node has at most 2 children. Moreover, because all the subtrees are not equivalent, there can not be more than $2^{2^{n-k}} - 1$ BDDs at depth k(this is the number of BDDs of depth $n-k$). Then we know two bounds on the number of nodes at a certain depth, we can simply compute the depth at which these bounds are maximized together. The formula to compute the maximum width (depending on the depth n) is:

$$max\_width(n) = \max_k \min\left(2^k, 2^{2^{n-k}} - 1\right).$$

There no simple formula to this value, but we can give a table showing this value for different depths. Table 3 shows these values for depths multiple of 8.

depth	max width
8	2^5
16	2^{12}
24	2^{19}
32	2^{27}
40	2^{34}

Table 3: Maximum width depending on the depth of the BDD

8 Conclusion

During this internship, I studied the BDD data structure as a way to represent sets of bit-vectorsin a CP solver. This data structure can be modified to have a polynomial representation (in the depth). With this representation, I was able to design algorithms to extend the operations that are possible on sets of bit-vectors to BDDs. For constraint solving, we had to come up with new notions to impose the consistency on limited-width-BDDs. Propagators have been defined to be able to propagate constraints defined on bit-vectors, or integers (such as bitwise operations, or div/mod constraints). The implementation has been done in `OCaml` and encapsulates the data structure with all the operations that can be done on it. Moreover, with the use of hash tables, the creation and operations on BDDs are as simple as doing tree traversal. The implementation has been tested a problem coming from cryptography.

There are a lot of computations that are possible on BDDs and that have not yet been theorized. The use of BDDs to represent integer domains can be extended to all the operations on integers. We already have division and modulos, and there are still many operations to study (*e.g.* arithmetic operations, comparison) on integers. We compared the BDDs to the bit-vector domain, showing that we can express more sets of bit-vectors with the BDDs. A comparison of these two domains on benchmarks could help us to quantify their differences. With the creation of a new domain, one have to give new algorithms for global constraints. Some global constraints like `regular` or cardinality constraints seem really suited to the BDD domain. For the applications, we applied our solver on the cryptanalysis of the AES protocol. The same method can be applied to other analyses of protocols.

References

[1] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on computers*, 6:509–516, 1978.

[2] David Bergman. *New techniques for discrete optimization.* PhD thesis, PhD thesis, Tepper School of Business, Carnegie Mellon University, 2013.

[3] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[4] David Gérault, Pascal Lafourcade, Marine Minier, and Christine Solnon. Revisiting aes related-key differential attacks with constraint programming. *Information Processing Letters*, 139:24–29, 2018.

[5] David Gerault, Marine Minier, and Christine Solnon. Constraint programming models for chosen key differential cryptanalysis. In *International Conference on Principles and Practice of Constraint Programming*, pages 584–601. Springer, 2016.

[6] David Gerault, Marine Minier, and Christine Solnon. Using constraint programming to solve a cryptanalytic problem. In *IJCAI 2017-International Joint Conference on Artificial Intelligence-Sister Conference Best Paper Track*, page 5, 2017.

[7] Chang-Yeong Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959.

[8] Laurent D Michel and Pascal Van Hentenryck. Constraint satisfaction over bit-vectors. In *Principles and Practice of Constraint Programming*, pages 527–543. Springer, 2012.

[9] Marine Minier, Christine Solnon, and Julia Reboul. Solving a symmetric key cryptographic problem with constraint programming. In *13th International Workshop on Constraint Modelling and Reformulation (ModRef), in conjunction with CP*, volume 14, pages 1–13, 2014.

[10] François Pachet and Pierre Roy. Musical harmonization with constraints : a survey. *Constraints*, 2001.

[11] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation.* TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.

[12] Gilles Simonin, Christian Artigues, Emmanuel Hebrard, and Pierre Lopez. Scheduling scientific experiments for comet exploration. *Constraints*, 20(1):77–99, 2015.

[13] Julien Vion and Sylvain Piechowiak. From mdd to bdd and arc consistency. *Constraints*, 23(4):451–480, 2018.

A Proof of the theorems

A.1 Merge value theorem

Theorem 2. *Let B be a BDD of root r, and u and v two nodes of the same layer in B. Let p_u (resp. p_v) the number of paths that go from r to u (resp. v). The merge value is then*

$$\begin{aligned} mv(u,v) &= p_u|\gamma(v)\backslash\gamma(u)| + p_v|\gamma(u)\backslash\gamma(v)| \\ &= p_u|\gamma(v)| + p_v|\gamma(u)| - (p_u+p_v)|\gamma(u)\cap\gamma(v)| \end{aligned}$$

Proof. Let P_u (resp. P_v) the set of bit-vectors representing all the paths from the root to node u (resp. v). The set represented by the paths passing by u and v separately is $A = P_u\cdot\gamma(u)\cup P_v\cdot\gamma(v)$. The one represented by the nodes merged is $B = (P_u \cup P_v)\cdot(\gamma(u)\cup\gamma(v))$

We can decompose B:

$$\begin{aligned} B =&(P_u\cup P_v)\cdot(\gamma(u)\cup\gamma(v)) \\ =&P_u\cdot\gamma(u)\cup P_u\cdot\gamma(v)\cup P_v\cdot\gamma(u)\cup P_v\cdot\gamma(v) \\ =&A\cup P_u\cdot\gamma(v)\cup P_v\cdot\gamma(u) \\ =&A\cup P_u\cdot(\gamma(v)\backslash\gamma(u)\cup\gamma(v)\cap\gamma(u))\cup P_v\cdot(\gamma(u)\backslash\gamma(v)\cup\gamma(u)\cap\gamma(v)) \\ =&A\cup P_u\cdot(\gamma(v)\backslash\gamma(u))\cup P_v\cdot(\gamma(u)\backslash\gamma(v)) \end{aligned}$$

$$\begin{aligned} mv(u,v) =&|B|-|A| \\ =&|P_u\cdot(\gamma(v)\backslash\gamma(u))\cup P_v\cdot(\gamma(u)\backslash\gamma(v)| \\ =&p_u|\gamma(v)\backslash\gamma(u)|+p_v|\gamma(u)\backslash\gamma(v)| \\ &= p_u|\gamma(v)|+p_v|\gamma(u)|-(p_u+p_v)|\gamma(u)\cap\gamma(v)| \end{aligned}$$

□

A.2 Equivalence theorem

Theorem 3. *[Equivalence between domains] The following domains represent the same sets of bit-vectors of size n:*

i $(\mathcal{P}(\{0,1\}))^n$

ii *the bit-vector domain of size n*

iii *cartesian product of boolean domains:* $\left(B^{\#}\right)^n$

iv *1-limited-width-BDDs of depth n (and F).*

Proof. We are going to prove equivalences one at a time. Here, we identify the cartesian product and the concatenation (such that we can say that $(\mathcal{P}(\{0,1\}))^n \subseteq \mathcal{P}(\{0,1\}^n)$).

i $\Leftrightarrow$ ii:

- Let $(B_1,\dots,B_n) \in (\mathcal{P}(\{0,1\}))^n$, we construct the bit-vector domain (l,u) in the following way:

$$\forall i \in \{1,\dots,n\}, l_i = \begin{cases} 0 & \text{if } 0 \in B_i \\ 1 & \text{otherwise} \end{cases}, u_i = \begin{cases} 1 & \text{if } 1 \in B_i \\ 0 & \text{otherwise} \end{cases}$$

- Let (l,u) be a bit-vector domain. Then we construct the cartesian product by:

$$\forall i \in \{1,\dots,n\}, B_i = \begin{cases} \emptyset & \text{if } l_i > u_i \\ \{0\} & \text{if } l_i = u_i = 0 \\ \{1\} & \text{if } l_i = u_i = 1 \\ \{0,1\} & \text{if } l_i < u_i \end{cases}$$

i $\Leftrightarrow$ iii: We simply have to show that $B^{\#}$ is equivalent to $\mathcal{P}(\{0,1\})$ (then it is extended to the cartesian product). The way to go from one to another is simply a rewriting of the elements: $\bot \leftrightarrow \emptyset, 0 \leftrightarrow \{0\}, 1 \leftrightarrow \{1\}, \top \leftrightarrow \{0,1\}$.

i $\Leftrightarrow$ iv . The each layer L_i of a 1-limited-width-BDD contain only one node t_i (because it had width one).

We first deal with the case of the inconsistent domain. If one of the set of the cartesian product is empty, then the associated BDD is F. Conversely, if the considered BDD is F then we construct the empty domain by giving only empty sets.

Now we can suppose that the domain is not empty (so the BDD is not F and the sets are not empty).

- Let $(B_1,\dots,B_n) \in (\mathcal{P}(\{0,1\}) \backslash \{\emptyset\})^n$. We construct the BDD by telling how each node $t_i, i \in \{0,\dots,n\}$ is connected to its next layer. We have that $t_n = T$. Then, for $i \in \{0,\dots,n-1\}$:
  - if $B_{i+1} = \{0\}$ then $t_i = (t_{i+1}, F)$
  - if $B_{i+1} = \{1\}$ then $t_i = (F, t_{i+1})$
  - if $B_{i+1} = \{0,1\}$ then $t_i = (t_{i+1}, t_{i+1})$
- We can do the inverse construction (simply by reversing the previous cases) to generate the sets from the BDD.

□

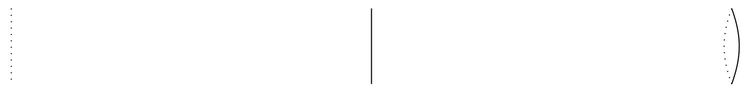


Figure 3: Cases when $|L_{n-1}| = |L_n| = 1$

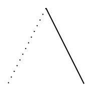


Figure 4: Case when $|L_{n-1}| = 1, |L_n| = 2$

A.3 Enumeration of 2-limited-width-BDDs

To enumerate the 2-limited-width-BDDs we do a distinction on the second to last layer (before the ending T leaf). We define a_{n-1} (resp. b_{n-1}) as the number of ways to have the first $n-1$ layers of a BDD of depth n where the second to last layer has width 1 (resp. 2). We have that a_n is the number of 2-limited-width-BDDs.

Then to get recurrence relations, we need to know how to put 0 and 1-edges between the second to last layer and the last layer. To compute b_n we also have to know how to put the edges supposing that the last layer have two nodes.

All the cases are presented in Figures 3, 4, 5, and 6. The first three figures are simple enumeration of all the possible cases.

The last case is a little harder to enumerate because we have to ensure that all the combinations are found. The nine first cases have their symmetric counterpart but not the last four ones. This means that in total there are $9 \times 2 + 4 = 22$ cases.

We finally get our recurrence relation as

$$\forall n \geq 0, \begin{pmatrix} a_n \\ b_n \end{pmatrix} = \begin{pmatrix} 3 & 3 \\ 2 & 22 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

.

To compute the general term of the sequence a_n, the eigenvalues of the matrix have to be computed. They can be computed using the characteristic polynomial of the matrix, and finding its roots (a polynomial of degree 2). The eigenvalues are $\lambda_1 = \frac{29+\sqrt(553)}{2}$ and $\lambda_2 = \frac{29-\sqrt(553)}{2}$.

B Pseudo-code for algorithms

Set operations The algorithms for set operations are presented here. There is the intersection in Algorithm 6, the union in Algorithm 7 and the set difference in Algorithm 8.

Bitwise operations The algorithms for bitwise operations on BDDs are presented here. There is the logical or $\vee$ operator in Algorithm 9, the logical and $\wedge$ in Algorithm 10, and the logical not $\neg$ in Algorithm 11.

Concatenation The algorithm for the concatenation of two BDDs is presented in Algorithm 12

Extraction The algorithms for extraction of prefixes and suffixes are presented in Algorithms 13 and 14.

Splitting The algorithm for splitting a BDD is presented in Algorithm 15.

Consistency The algorithm for MDD-consistency is presented in Algorithm 16.

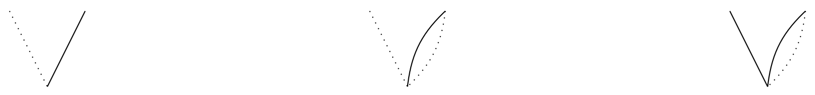


Figure 5: Case when $|L_{n-1}| = 2, |L_n| = 1$

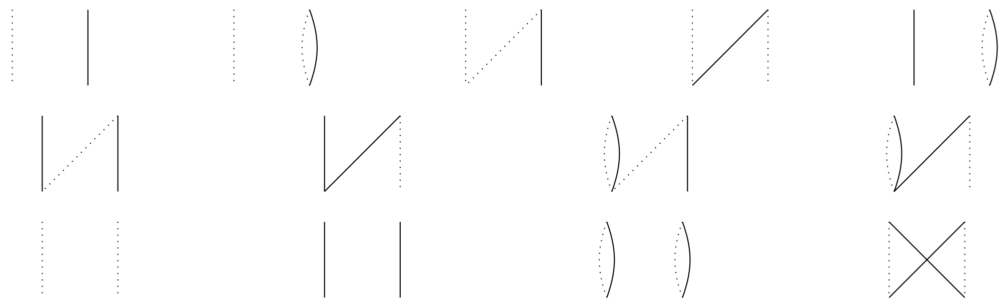


Figure 6: Case when $|L_{n-1}| = 2, |L_n| = 2$ considered up to symmetry, except the last four ones

```
1 Function INTER($B_1, B_2$) notation $B_1 \cap B_2$
      Data: Two BDDs $B_1$ and $B_2$ of same depth
      Result: The intersection $B_1 \cap B_2$
2     if $B_1 = F$ or $B_2 = F$ then
3         return $F$
4     else if $B_1 = T$ or $B_2 = T$ then
5         return $T$
6     else
7         Let $a_1, b_1$ such that $B_1 = Node(a_1, b_1)$
8         Let $a_2, b_2$ such that $B_2 = Node(a_2, b_2)$
9         return BDD_OF(INTER($a_1, a_2$), INTER($b_1, b_2$))
```

Algorithm 6: Computing the intersection of two BDDs

```
1 Function UNION($B_1, B_2$) notation $B_1 \cup B_2$
      Data: Two BDDs $B_1$ and $B_2$ of same depth
      Result: The union $B_1 \cup B_2$
2     if $B_1 = F$ then
3         return $B_2$
4     else if $B_2 = F$ then
5         return $B_1$
6     else if $B_1 = T$ or $B_2 = T$ then
7         return $T$
8     else
9         Let $a_1, b_1$ such that $B_1 = Node(a_1, b_1)$
10        Let $a_2, b_2$ such that $B_2 = Node(a_2, b_2)$
11        return BDD_OF(UNION($a_1, a_2$), UNION($b_1, b_2$))
```

Algorithm 7: Computing the union of two BDDs

```
1 Function DIFF($B_1, B_2$) notation $B_1 \backslash B_2$
      Data: Two BDDs $B_1$ and $B_2$ of same depth
      Result: The difference $B_1 \backslash B_2$
2     if $B_1 = F$ or $B_2 = T$ then
3         return $F$
4     else if $B_2 = F$ then
5         return $B_1$
6     else
7         Let $a_1, b_1$ such that $B_1 = Node(a_1, b_1)$
8         Let $a_2, b_2$ such that $B_2 = Node(a_2, b_2)$
9         return BDD_OF(DIFF($a_1, a_2$), DIFF($b_1, b_2$))
```

Algorithm 8: Computing the set difference of two BDDs

```
1 Function OR($B_1, B_2$) notation $B_1 \vee B_2$
      Data: Two BDDs $B_1$ and $B_2$ of same depth
      Result: The or $B_1 \vee B_2$
2     if $B_1 = T$ and $B_2 = T$ then
3         return $T$
4     else if $B_1 = F$ or $B_2 = F$ then
5         return $F$
6     else
7         Let $a_1, b_1$ such that $B_1 = Node(a_1, b_1)$
8         Let $a_2, b_2$ such that $B_2 = Node(a_2, b_2)$
9         $a \leftarrow$ OR($a_1, a_2$)
10        $b \leftarrow$ OR($b_1, b_2$) $\cup$ OR($a_1, b_2$) $\cup$ OR($b_1, a_2$)
11        return BDD_OF($a, b$)
```

Algorithm 9: Computing the bitwise `or` $\vee$ of two BDDs

```
1 Function AND($B_1, B_2$) notation $B_1 \wedge B_2$
      Data: Two BDDs $B_1$ and $B_2$ of same depth
      Result: The and $B_1 \wedge B_2$
2     if $B_1 = T$ and $B_2 = T$ then
3         return $T$
4     else if $B_1 = F$ or $B_2 = F$ then
5         return $F$
6     else
7         Let $a_1, b_1$ such that $B_1 = Node(a_1, b_1)$
8         Let $a_2, b_2$ such that $B_2 = Node(a_2, b_2)$
9         $a \leftarrow$ XOR($a_1, a_2$) $\cup$ XOR($a_1, b_2$) $\cup$ XOR($b_1, a_2$)
10        $b \leftarrow$ XOR($b_1, b_2$)
11        return BDD_OF($a, b$)
```

Algorithm 10: Computing the bitwise `and` $\wedge$ of two BDDs

```
1 Function NOT($B$) notation $\neg B$
      Data: A BDD $B$
      Result: The not $\neg B$
2     if $B = T$ or $B = F$ then
3         return $B$
4     else
5         Let $a, b$ such that $B = Node(a, b)$
6         return BDD_OF($b, a$)
```

Algorithm 11: Computing the bitwise `not` $\neg$ of a BDD

```
1 Function CONCAT($B_1, B_2$) notation $B_1 \cdot B_2$
      Data: Two BDDs $B_1$ and $B_2$
      Result: The concatenation $B_1 \cdot B_2$
2     if $B_1 = F$ then
3         return $F$
4     else if $B_1 = T$ then
5         return $B_2$
6     else
7         Let $a, b$ such that $B_1 = Node(a, b)$
8         return BDD_OF(CONCAT($a, B_2$), CONCAT($b, B_2$))
```

Algorithm 12: Computing the concatenation of two BDDs

```
1 Function PREF($B, k$)
      Data: A BDD $B$ and a length of the wanted prefixes $k$
      Result: A BDD representing the prefixes of length $k$ of $B$
2     if $B = F$ then
3         return $F$
4     else if $k = 0$ then
5         return $T$
6     else if $B = T$ then
7         return ERROR("The length is too big")
8     else
9         Let $a, b$ such that $B = Node(a, b)$
10        return BDD_OF(PREF($a, k - 1$), PREF($b, k - 1$))
```

Algorithm 13: Computing the prefixes of a BDD

```
1 Function SUFF($B, k$)
      Data: A BDD $B$ and a length of the wanted suffixes $k$
      Result: A set of BDDs representing the suffixes of length $k$ of $B$
2     if $B = F$ then
3         return $F$
4     else if $depth(B) = k$ then
5         return $\{B\}$
6     else if $B = T$ then
7         return ERROR("The length is too big")
8     else
9         Let $a, b$ such that $B = Node(a, b)$
10        return SUFF($a, k$) $\cup$ SUFF($b, k$)
```

Algorithm 14: Computing the suffixes of a BDD

```
Function SPLIT_AT_DEPTH($B, d$)
    Data: A BDD $B$ and a depth d
    Result: Two non empty BDDs whose union is $B$
    if $B = T$ then
        return ERROR("The depth is too high")
    else if $B = F$ then
        return $F$
    else
        Let $a, b$ such that $B = Node(a, b)$ if $d = 0$ then
            return BDD_OF($a, F$), BDD_OF($F, b$)
        else
            Let $a_1, a_2$ = SPLIT_AT_DEPTH($a, d - 1$) Let $b_1, b_2$ = SPLIT_AT_DEPTH($b, d - 1$)
             return (BDD_OF($a_1, b_1$), BDD_OF($a_2, b_2$))
```

Algorithm 15: Splitting the BDD at given depth

```
Function TRAVERSE($B_1, B_2$)
    Data: Two BDDs $B_1$ and $B_2$ of same depth
    Result: Works in place, mark the edges of $B_1$ that are in paths in $B_2$
    if $B_1 \notin \{T, F\}$ and $B_2 \notin \{T, F\}$ then
        Let $a_1, b_1$ such that $B_1 = Node(a_1, b_1)$ Let $a_2, b_2$ such that $B_2 = Node(a_2, b_2)$ if
         $a_1 \neq F$ and $a_2 \neq F$ then
            MARK_EDGE($B_1, a_1$) TRAVERSE($a_1, a_2$)
        if $b_1 \neq F$ and $b_2 \neq F$ then
            MARK_EDGE($B_1, b_1$) TRAVERSE($b_1, b_2$)
Function ERASE($B$)
    Data: A BDD with edges marked or not
    Result: The same BDD where we deleted edges not marked
    if $B \in \{F, T\}$ then
        return $B$
    else
        Let $a, b$ such that $B = Node(a, b)$
        if IS_MARKED($B, a$) then
            $a' \leftarrow$ ERASE($a$)
        else
            $a' \leftarrow F$
        if IS_MARKED($B, b$) then
            $b' \leftarrow$ ERASE($b$)
        else
            $b' \leftarrow F$
        return BDD_OF($a', b'$)
Function CONSISTENCY($B_1, B_2$)
    Data: Two BDDs $B_1$ and $B_2$ of same depth
    Result: The BDD $B_1$ consistent with respect to $B_2$
    TRAVERSE($B_1, B_2$)
    return ERASE($B_1$)
```

Algorithm 16: Computing the MDD-consistency of a BDD with respect to an other