Binary Decision Diagrams: from Tree Compaction to Sampling*

Julien Clément[1] and Antoine Genitrini[2]

[1] Normandie Univ, UNICAEN, ENSICAEN, CNRS, GREYC, 14000 Caen, France
Julien.Clement@unicaen.fr
[2] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France.
Antoine.Genitrini@lip6.fr

Abstract. Any Boolean function corresponds with a complete full binary decision tree. This tree can in turn be represented in a maximally compact form as a direct acyclic graph where common subtrees are factored and shared, keeping only one copy of each unique subtree. This yields the celebrated and widely used structure called reduced ordered binary decision diagram (ROBDD). We propose to revisit the classical compaction process to give a new way of enumerating ROBDDs of a given size without considering fully expanded trees and the compaction step. Our method also provides an unranking procedure for the set of ROBDDs. As a by-product we get a random uniform and exhaustive sampler for ROBDDs for a given number of variables and size.

1 Introduction

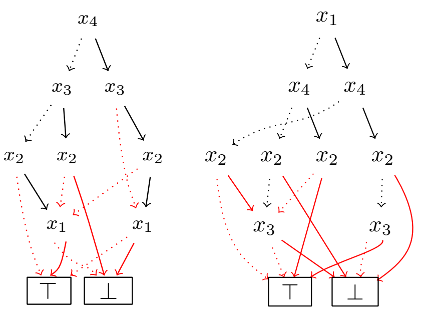

Fig. 1. Two Reduced Ordered Binary Decision Diagrams associated to the same Boolean function. Nodes are labeled with Boolean variables; left dotted edges (resp. right solid edges) are `0` links (resp. `1` links).

The representation of a Boolean function as a binary decision tree has been used for decades. Its main benefit, compared to other representations like a truth table or a Boolean circuit, comes from the underlying *divide-and-conquer* paradigm. Thirty years ago a new data structure emerged, based on the compaction of binary decision tree, and hereafter denoted as Binary Decision Diagrams (or BDDs) [1]. Its take-off has been so spectacular that many variants of compacted structures have been developed, and called through many acronyms as presented in [14]. One way to represent the different diagrams consists in their embedding as directed acyclic graphs (or DAGs). One reason for the existence of all these variants of diagrams is due to the fact that each DAG correspondence has its own internal

* This work was partially supported by the ANR projects Metaconc ANR-15-CE40-0014 and Ping/Ack ANR-18-CE40-0011. An implementation of the results is provided at `https://github.com/agenitrini/BDDgen`.

agency of the nodes and thus each representation is oriented towards a specific constraint. For example, the case of Reduced Ordered Binary Decision Diagrams (ROBDDs) is such that the variables do appear at most once and in the same order along any path from the source to a sink of the DAG, and furthermore, no two occurrences of the same subgraph do appear in the structure. For such structures and others, like QOBDDs or ZBDDs for example, there is a canonical representation of each Boolean function.

In his book [9] Knuth proves or recalls combinatorial results, like properties for the profile of a BDD, or the way to combine two structures to represent a more complex function. However, one notes an unseemly fact. There are no results about the distribution of the Boolean functions according to their ROBDD size. In fact in contrast to (e.g.) binary trees where there is a recursive characterization that allows to well specify the trees, we have no local-constraint here for ROBDDs ans thus a similar recurrence is unexpected. Very recently, there is a first study exploring experimentally, numerically, and theoretically the typical and worst-case ROBDD sizes in [12]. We aim at obtaining the same kind of combinatorial results but here we design a partition of the decision diagrams that allows us to go much further in terms of size. In particular we obtain an exhaustive enumeration of the diagrams according to their size up to 9 variables. This was unreachable through the exhaustive approach proposed in [12] due to the double exponential complexity of the problem: there are 2^{2^k} Boolean functions with k variables. Our C++ implementation fully manages the case of 9 variables (see Fig 2) that corresponds to $2^{512} \approx 10^{154}$ functions. In particular for 9 Boolean variables, our implementation shows one seventh of all ROBDDs are of size 132 (the possible sizes range from 3 to 143). Furthermore, ROBDDs of size between 125 and 143 represents more than 99.8% of all ROBDDs, in accordance with theoretical results from [13,8].

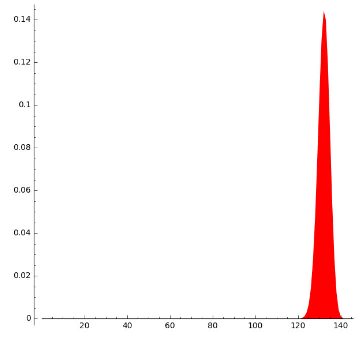

Fig. 2. Proportion of BDDs over 9 variables according to their size

Starting from the well-known compaction process (that takes a binary decision tree and outputs its compacted form, the ROBDD), our combinatorial study gives a way of construction for ROBDDs of a given size, but without the compaction step. We further define a total order over the set of ROBDDs and we propose both an unranking and an exhaustive generation algorithm. The first one gives as a by-product a uniform random sampler for ROBDDs of a given number of variables and size. One strength of our approach is that it allows to sample *uniformly* ROBDDs of "small" size, for instance of linear size w.r.t the number k of variables, very efficiently in contrast to a naive rejection algorithm. The usual uniform distribution on Boolean functions [13] yields with high probability ROBDDs of near maximal size of order $2^k/k$, although ROBDDs encountered in applications, when tractable, are smaller. As a perspective, once the unranking method is well understood, and in particular the poset underlying the ROBDDs, then we might be able to bias the distribution to sample only in a specific subclass, e.g. ROBDDs corresponding to a particular class of formulas (e.g. read-once formulas).

Our results have practical applications in several contexts, in particular for testing structures and algorithms. The study given in [3] executes tests for an algorithm whose parameter is a binary decision diagram. It is based on QuickCheck [2], the famous software, taking as an entry a random generator and generating test cases for test suites. Using our uniform generator, we aim at obtaining statistical testing, in the sense that the underlying distribution of the samples is uniform, thus allowing to extract statistics thanks to the tests. Another application of our approach allows to derive exhaustive testing for small structures, like the study in [10], that we also can conduct inside QuickCheck.

In this paper, we focus exclusively on ROBDDs which is one of the first and simplest variants. Section 2 introduces the combinatorics underlying the decision tree compaction, leading in Section 3 to a way to unambiguously specify the structure of reduced ordered binary decision diagrams. We apply this strategy in Section 4 and obtain an unranking algorithm for ROBDDs.

2 Decision diagrams as compacted trees

This section defines precisely our combinatorial context. Many definitions are detailed in the monograph of Wegener [14] and in the dedicated volume [9] of Knuth.

In this section we first recall a one-to-one correspondence between the representation of a Boolean function as a binary decision tree (built on a specific variable ordering and seen as a *plane tree*, i.e. the children of an internal nodes are ordered) and a reduced ordered binary decision diagram (ROBDDs) also seen as a plane structure. This approach is *non-classical* in the context of BDDs, but it allows the formalization of an equivalence relationship on ROBDDs that is the key of our enumeration: in fact our approach foundation relies on breaking down the symmetry in ROBDDs. We consider Boolean function on k variables. We recall there are 2^{2^k} such Boolean functions. The compaction process is now formalized.

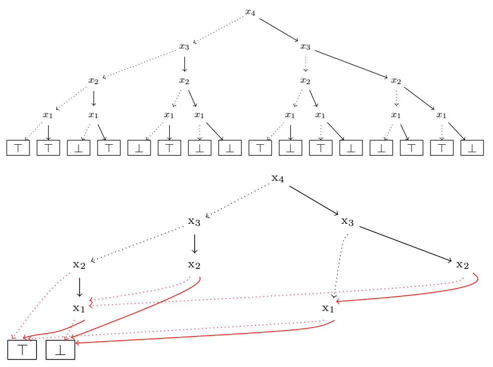

Fig. 3. A decision tree and its postorder compaction

Compaction and plane decision diagrams A Boolean function can be represented thanks to a binary decision diagram, which is a rooted, directed, acyclic graph, which consists of decision nodes and terminal nodes. There are two types of terminal nodes $\top$ and $\bot$ corresponding to truth values (resp. 1 and 0). Each decision node ν is labeled by a Boolean variable x_ν and has two child nodes (called *low child* and *high child*). The edge from node ν to a low (or high) child represents an assignment of x_ν to 0 and is represented as a dotted line (respectively 1, represented as a solid line). In the following we represent ROBDDs and decision trees (or BDDs in general) as *plane structures*, i.e. for a node we consider its low child to be its left child and the high child to be its right child.

In a (plane) full binary decision tree, no subtree is shared. By contrast we may decrease the number of decision nodes by factoring and sharing common substructures. Representing a function with its full decision tree is not space efficient. In Fig. 3 we depict, on top, a decision tree of a Boolean function on 4 variables. In the bottom of the figure we represent the compaction of the latter decision tree by using the classical common subexpression recognition notion (cf. e.g. [4,7]) based on a postorder traversal of the tree.

Definition 1 (Compaction). *Let T be the (plane) binary decision tree of a function f. The* DAG *T is modified through a postorder traversal. When the node ν is under visit, ν being a child of a node ρ. If an identical subtree than T_ν, the one rooted in ν, has already be seen during the traversal, rooted in a node μ, then T_ν is removed from T and the node ρ gets a pointer to μ (replacing the edge to ν). Once T has been traversed, the resulting* DAG *is the plane* ROBDD *of f.*

In our figures of ROBDDs we draw the pointers in red (there is an exception for the edges to the terminal nodes as we remark in Fig. 3 also drawn in red).

In a classical setting, ROBDDs are obtained by applying repetitively reduction rules (well detailed in [14]) to OBDDs, and the process is confluent. Our approach conceptually takes as a starting point a full decision tree with a given ordering on variables (meaning all nodes at the same level are labeled by the same variable) and applies the compaction rules by examining nodes of the tree in postorder.

For example the plane ROBDD in Fig. 3 corresponds to the leftmost ROBDD depicted (in the classical way) in Fig.1. Note that for a given Boolean function, using two distinct variable orderings can lead to two ROBDDs of different sizes (see Fig. 1 for such a situation). Nonetheless, an ordering of the variables being fixed, each Boolean function is represented by exactly one single ROBDD obtained through the compaction of its decision tree for this order.

In the rest of the paper, we consider only plane ROBDD*s. From now we thus call them* BDD*s. We also assume the set of variables $X = \{\ldots > x_k > \ldots > x_1\}$ is totally ordered.*

Our first goal aims at giving an effective method to enumerate BDDs with a chosen number k of variables and size n. A first naive approach is: (1) enumerate all the 2^{2^k} Boolean functions by construction of the decision trees; (2) apply the compaction procedure; and (3) finally filter the BDDs of size equal to the target size n. This algorithm ceases to be practical for k larger than 4 (see [12]).

In this paper, we propose a new combinatorial description of BDDs providing the basis for an enumeration algorithm avoiding the enumeration of all Boolean functions on k variables.

3 Recursive decomposition

This section introduces a canonical and unambiguous decomposition of the BDDs yielding a recursive algorithm for their enumeration.

Automaton point of view Let us introduce an equivalent representation for a BDD. A BDD can indeed be described as a deterministic finite automaton with additional constraints and properties. This point of view gives a convenient formal characterization of the decomposition of BDDs used in our algorithms.

Definition 2 (BDD as an automaton). *A BDD B of index k is a tuple (Q, I, r, δ) where*

- *Q is the set of nodes of the BDD. Q contains two special sink nodes $\bot$ and $\top$.*
- *$I : Q \to \{0, \dots, k\}$ is the index function which associates with every node its index. By convention the index of both sink nodes is 0.*
- *$r \in Q$ is the root and has index $I(r) = k$.*
- *$\delta : Q \setminus \{\bot, \top\} \times \{\mathtt{0}, \mathtt{1}\} \to Q$ is the full transition function.*

There are constraints on δ translating the classical ones of the BDDs:

- *for any node $\nu \in Q \setminus \{\bot, \top\}$, $\delta(\nu, \mathtt{0}) \neq \delta(\nu, \mathtt{1})$.*
- *for any distinct nodes μ and ν with the same index, we have $\delta(\mu, \mathtt{0}) \neq \delta(\nu, \mathtt{0})$ or $\delta(\mu, \mathtt{1}) \neq \delta(\nu, \mathtt{1})$.*
- *the graph underlying δ forms a DAG with a unique node of in-degree 0, the root r.*
- *if $\tau = \delta(\nu, \alpha)$ for some $\alpha \in \{\mathtt{0}, \mathtt{1}\}$ then $I(\tau) < I(\nu)$.*

We say τ is the low child of ν (respectively high child of ν) if $\delta(\nu, \mathtt{0}) = \tau$ (resp. $\delta(\nu, \mathtt{1}) = \tau$).

Definition 3 (Spine of a BDD, tree and non-tree edges). *Let a BDD $B = (Q, I, r, \delta)$ of root-index k. The* spine *of B is the spanning tree obtained by a* depth-first search *of the (plane) BDD (where low child is accessed before the high one), and omitting the sinks $\bot$ and $\top$. For a BDD B, the edges of the spine forms the set of* tree edges *(drawn in black). The other edges form the set of* non-tree edges *(drawn in red). We describe the spine T as a tuple $T = (Q', I, r, \delta')$ with set of nodes $Q' = Q \setminus \{\bot, \top\}$ (with the same index function I as for B). The edges of the spine are described using a partial transition function $\delta' : Q' \times \{\mathtt{0}, \mathtt{1}\} \to Q' \cup \{\text{NIL}\}$ where NIL is a special symbol designating an undefined transition.*

Using standard terminology for depth-first search, *non-tree* edges are either *forward or cross* edges. We remark that, by definition, a DAG admits no cycles and still in the standard notation, it has no *backward* edges.

Undefined values of the transition function δ' can conveniently be seen as half edges. Since in a BDD every non-sink node has two children, the spine of a BDD of size n has $(n - 2)$ nodes and $(n - 1)$ half edges (drawn in red in Fig. 1). The four possible types of a node are depicted, as the roots in Fig. 4.

Definition 4 (Valid tree). *A binary tree is said to be* valid *if it is the spine of some BDD. The set of spines of size n is denoted as $\mathcal{T}_n$.*

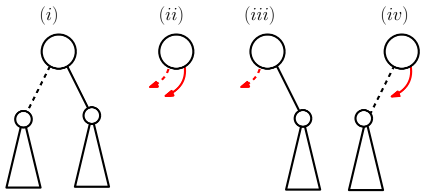

Fig. 4. The four cases for a node of a spine. From left to right: an internal node with both transitions defined, two half edges, one low (left) half edge, one high (right) half edge (cf. Proposition 1)

See Fig. 5 for examples of valid and invalid trees. To the best of our knowledge, there is no way to characterize valid trees, apart from exhibiting a ROBDD admitting this tree as a spine. We will discuss this point later.

For enumerating BDDs it will prove convenient to introduce the profile list of a set of nodes and some other useful notation for lists manipulation.

Definition 5. *The* profile *of $\mathcal{N}$, denoted by* profile($\mathcal{N}$), *is a list with* $(k+1)$ *components* $\boldsymbol{p} = (p_0, \dots, p_k)$ *where* $k = \max_{\nu \in \mathcal{N}} I(\nu)$ *is the maximal index and* p_i *is the number of nodes of index* i *in* $\mathcal{N}$.

This definition extends naturally to trees, graphs, etc. We also equip the set of lists with a '+' operation: let two lists $\boldsymbol{v} = (v_0, \dots, v_m)$ and $\boldsymbol{v'} = (v'_0, \dots, v'_n)$ with $n \geq m$ (w.l.o.g.), the sum $\boldsymbol{v} + \boldsymbol{v'}$ is equal to $\boldsymbol{w} = (w_0, \dots, w_n)$ where for all $0 \leq i \leq m$, $w_i = v_i + v'_i$ and otherwise, when $m < i \leq n$, $w_i = v'_i$.

In the following, we will use two orderings on the nodes of a plane BDD induced by depth-first search, and called postordering and preordering. Since the structure is plane these orderings correspond exactly with the classical postorder traversal and the preorder traversal of its spanning tree. In a tree, for a node ν with low child ν_0 and high child ν_1, the postorder traversal visits the subtree rooted at ν_0 then, the one rooted at ν_1 and finally ν. The preorder traversal first visits the node ν, then the subtree rooted at ν_0 and finally the subtree rooted at ν_1. We use the notation $\mu \prec_{\text{post}} \nu$ (resp. $\mu \prec_{\text{pre}} \nu$) if the node μ is visited before ν using the postorder (resp. preorder) traversal.

We characterize now how the partial transition function of the spine is related to the full transition function of the BDD. Introducing the pool and level set of a node, we describe the valid choices for non-tree edges to yield a BDD.

Definition 6 (Pool and level set). *Let T be the spine of a* BDD*. The pool of a node $\nu \in T$ is*

$$\mathcal{P}_T(\nu) = \{\tau \in T \mid \tau \prec_{\text{pre}} \nu \text{ and } I(\tau) < I(\nu)\} \cup \{\bot, \top\}\,.$$

The pool profile $p_T(\nu)$ of a node ν in a spine T is $p_T(\nu) = \text{profile}(\mathcal{P}_T(\nu))$.
The level set *of ν is $\mathcal{S}_T(\nu) = \{\tau \in T \mid \tau \prec_{\text{pre}} \nu$ and $I(\tau) = I(\nu)\}$, and the* level rank $s_T(\nu) = |\mathcal{S}_T(\nu)|$ *of a node ν is the rank of ν among the set of nodes with the same index.*

Informally the pool of a node ν of a tree T is the set of nodes we could choose as a low child for ν without invalidating the spine. The first component of a pool profile is always 2 since both sinks $\bot$ or $\top$ are present in the pool of any node of the spine (providing the underlying BDD is not reduced to $\mathtt{1}$ or $\mathtt{0}$).

Proposition 1. *Let $T = (Q', I, r, \delta')$ be a valid spine with set of nodes Q', root r and partial transition function $\delta' : Q' \times \{\mathtt{0}, \mathtt{1}\} \to Q' \cup \{\text{NIL}\}$. The full transition function $\delta : Q' \times \{\mathtt{0}, \mathtt{1}\} \to Q' \cup \{\bot, \top\}$ is the transition function of a BDD with spine T if and only for any node $\nu \in Q'$, noting $\nu_{\mathtt{0}} = \delta(\nu, \mathtt{0})$ and $\nu_{\mathtt{1}} = \delta(\nu, \mathtt{1})$, the pair $(\nu_{\mathtt{0}}, \nu_{\mathtt{1}})$ satisfies*

(i) if $\delta'(\nu, \mathtt{0}) \neq \text{NIL}$ and $\delta'(\nu, \mathtt{1}) \neq \text{NIL}$ then $\nu_\alpha = \delta'(\nu, \alpha)$ for $\alpha \in \{\mathtt{0}, \mathtt{1}\}$.

(ii) if $\delta'(\nu, \mathtt{0}) = \delta'(\nu, \mathtt{1}) = \text{NIL}$, then

$$\nu_\alpha \prec_{\text{pre}} \nu \text{ and } I(\nu_\alpha) < I(\nu) \text{ for } \alpha \in \{\mathtt{0}, \mathtt{1}\} \text{ and } \nu_{\mathtt{0}} \neq \nu_{\mathtt{1}},$$

and there is no node $\tau \neq \nu$ with the same index as ν such that $\delta(\tau, \cdot) = \delta(\nu, \cdot)$.

(iii) if $\delta'(\nu, \mathtt{0}) = \text{NIL}$ and $\delta'(\nu, \mathtt{1}) \neq \text{NIL}$, then

$$\nu_{\mathtt{0}} \prec_{\text{pre}} \nu,\ I(\nu_{\mathtt{0}}) < I(\nu) \text{ and } \nu_{\mathtt{1}} = \delta'(\nu, \mathtt{1}).$$

(iv) if $\delta'(\nu, \mathtt{0}) \neq \text{NIL}$ and $\delta'(\nu, \mathtt{1}) = \text{NIL}$, then $\nu_{\mathtt{0}} = \delta'(\nu, \mathtt{0})$ and

$$\nu_{\mathtt{1}} \prec_{\text{post}} \nu,\ \nu_{\mathtt{1}} \neq \nu_{\mathtt{0}} \text{ and } I(\nu_{\mathtt{1}}) < I(\nu).$$

Proof. Since $\delta(\cdot, \cdot)$ must extend $\delta'(\cdot, \cdot)$, case (i) is trivial since we must only extend the transition function where $\delta(\nu, \alpha) = \text{NIL}$. In case (ii), we have to choose for $(\nu_{\mathtt{0}}, \nu_{\mathtt{1}})$ two nodes in the pool of ν (ν is an external node of the spine). We use the preorder traversal (but since ν is an external node, the postorder would also be fine). Moreover $\nu_{\mathtt{0}} \neq \nu_{\mathtt{1}}$ and no node with the same index as ν can have the exact same descendants $(\nu_{\mathtt{0}}, \nu_{\mathtt{1}})$ in accordance with Definition 2. In case (iii), the low child must be chosen in the pool of ν since we preserve the spine. In case (iv), the high child of ν is also chosen in the pool ν or in the pool of $\nu_{\mathtt{0}}$ (and must still be different from $\nu_{\mathtt{0}}$ by Definition 2). □

4 Counting and Generating BDDs

In this section, we sketch algorithms in order to count and sample BDDs of a given size n and given number k of variables.

Counting BDDs Given a spine T, we can compute the number of BDDs corresponding with this spine. Thus counting BDDs of a certain size n will consists in building all valid spines of size $(n-2)$ and completing the transition function of the spine in all possible ways according to Proposition 1.

Definition 7 (Weight). *Let $T = (Q', I, \delta', r)$ be a spine, the* weight $w_T(\nu)$ *of a node $\nu \in Q'$ is the number of possibilities for completing the transition function $\delta'(\mu, \cdot)$ and yielding a BDD with spine T. The* cumulated weight *of a subtree T_ν rooted at $\nu \in T$ is $W_T(\nu) = \prod_{\tau \in T_\nu} w_T(\tau)$. We write $W(T) = W_T(r)$ to denote the cumulated weight of the whole spine T rooted at r.*

Note that the number of choices for the missing transitions out of a node ν are the ones remaining after previous choices have been made for other nodes of the spine.

Proposition 2 (Weight of a node). *Let T be a spine T, the* weight *of a node $\nu \in T$ is*

$$w_T(\nu) = \begin{cases} 1 & \text{if } \delta'(\nu,0) \neq \textsc{nil} \text{ and } \delta'(\nu,1) \neq \textsc{nil} \\ \|p_T(\nu)\|\big(\|p_T(\nu)\| - 1\big) - s_T(\nu) & \text{if } \delta'(\nu,0) = \delta'(\nu,1) = \textsc{nil} \\ \|p_T(\nu) + \text{profile}(T')\| & \text{if } \delta'(\nu,0) \neq \textsc{nil} \text{ and } \delta'(\nu,1) = \textsc{nil} \\ \|p_T(\nu)\| & \text{if } \delta'(\nu,0) = \textsc{nil} \text{ and } \delta'(\nu,1) \neq \textsc{nil} \end{cases}$$

where $p_T(\nu)$ is the pool profile of node ν, $T' = T_{\nu_0}$ is the subtree (when defined) rooted at $\nu_0 = \delta'(\nu, \mathsf{0})$, and, for a list $\boldsymbol{p} = (p_0, \ldots, p_k)$, we denote $\|\boldsymbol{p}\| = \sum_{i=0}^{k} p_i$.

In the third case, by $p_T(\nu) + \text{profile}(T')$, we mean the profile of the set of nodes visited before ν with the postorder traversal of T and of index strictly smaller than $I(\nu)$.

Proof. This is a direct application of Proposition 1. □

This formula allows to detect if a tree is a valid spine. Indeed as soon as the weight of a node is zero or negative, there is no way to define a total transition function δ for a BDD. Note that this situation can only happen for external nodes having two half edges, since for any node $\nu \in Q'$ and any spine T, $\|p_T(\nu)\| \geq 2$. In Fig. 5, the binary tree on the left is invalid and cannot be the spine of any BDD. The two other trees on the right have weights 4 and 24, i.e., are resp. the spines of exactly 4 and 24 BDDs. It is an open problem to characterize the set of valid trees (apart from exhibiting corresponding BDDs).

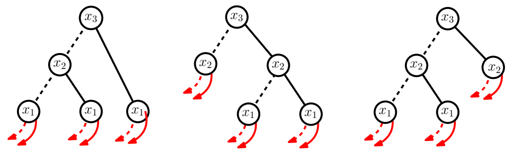

Fig. 5. three examples of binary trees (first one is invalid, the two other have respective weights 4 and 24).

Proposition 2 gives access to the total weight of the spine $W(T)$ using a recursive procedure. A natural way to proceed algorithmically is to use a recursive postorder traversal of the tree maintaining at each node the weight in a multiplicative manner. To do so we need to keep track in the traversal of the pool profile and level rank of the current node.

Initially the pool of the root is reduced to the set $\{\bot, \top\}$. Thus the initial pool profile of the root of index k is initialized to $(2, 0, \ldots, 0)$ of length k. The level rank of the root of the spine is 0.

Proposition 3. *Let $N(n,k)$ be the number of BDDs of index k and size n*

$$N(n,k) = \textstyle\sum_{T \in \mathcal{T}_{n-2,k}} W(T),$$

where $\mathcal{T}_{m,k}$ is the set of valid spines with m nodes for BDDs of index k.

Proof. The weight of a spine is the number of ways of extending the transition function of T (Proposition 1), hence the number of BDDs for this given spine. □

Combinatorial description of spines The set of spines is not straightforward to characterize in a combinatorial way. Indeed we need context to decide if the weight of a particular node in a tree is 0 or less, which in turn yields that the tree is not valid . To enumerate spines, we build recursively binary trees, and, while computing weights for its nodes, as soon we can decide the (partially built) tree is not valid, the tree is discarded.

To decompose (or count) spines of any size or index, $\mathcal{T} = \bigcup_{n\geq 1}\bigcup_{k\geq 1}\mathcal{T}_{n,k}$, we introduce a partition over subtrees which can occur in a spine $T \in \mathcal{T}$. The goal is to identify identical subtrees occurring within different spines and with the same weight to avoid redundant computations.

The combinatorial description we are about to present originates from the following observation: let us fix a spine T and a node $\nu \in T$. From Proposition 2, to compute the cumulated weight of the subtree T_ν rooted at ν, the sole knowledge of the pool profile $p_T(\nu)$ and the level rank $s_T(\nu)$ is sufficient.

Let S and S' be two subtrees with respective roots ν and ν' in some spines T and T', we denote $S \equiv S'$ if the following three conditions are satisfied:
– both trees have the same size: $|S| = |S'|$;
– the roots of both trees have the same pool profile: $p_T(\nu) = p_{T'}(\nu')$;
– the roots of both trees have the same level rank: $s_T(\nu) = s_{T'}(\nu')$.

The set $\mathcal{T}_{m,\boldsymbol{p},s}$ is the class equivalence for the relation '$\equiv$' and gathers trees (as a set, without multiplicities) which are possible subtrees of size m in any spine, knowing only the pool profile $\boldsymbol{p}$ and level rank s of the root of the subtree. More formally:

$$\mathcal{T}_{m,\boldsymbol{p},s} = \{T_\nu \mid (\exists T \in \mathcal{T})\ (\exists \nu \in T)\ p_T(\nu) = \boldsymbol{p} \text{ and } s_T(\nu) = s\}.$$

Note that we have $\mathcal{T}_{n,k} = \mathcal{T}_{n,(2,0,\dots,0),0}$, where $(2, 0, \dots , 0)$ has k components.

Proposition 4. *The set $\mathcal{T}_{m,\boldsymbol{p},s}$ of subtrees of size m rooted at a node having pool profile $\boldsymbol{p} = (p_0, \dots , p_{k-1})$ and level rank s occurring in the set of spines $\mathcal{T}$ is decomposed without any ambiguity. We decompose a subtree $T \in \mathcal{T}_{m,\boldsymbol{p},s}$ as a tuple (ν, T', T'') where the root ν has index k and T' and T'' are its left and right (possibly empty) subtrees of respective sizes i and $m-1-i$, with $0 \leq i \leq m-1$, and verifying (when non empty)*

(i) $T' \in \displaystyle\bigcup_{k_0\in\{1,\dots,k-1\}} \mathcal{T}_{i,(p_0,\dots,p_{k_0-1}),p_{k_0}}$

(ii) $T'' \in \displaystyle\bigcup_{k_1\in\{1,\dots,k-1\}} \mathcal{T}_{m-i-1,(p'_0,\dots,p'_{k_1-1}),p'_{k_1}}$, *with* $\boldsymbol{p}' = \boldsymbol{p} + \operatorname{profile}(T')$

(iii) if $m = 1$ *then* $\left(\sum_{i=0}^k p_i\right) \cdot \left(-1 + \sum_{i=0}^k p_i\right) - s > 0$.

This proposition ensures that we can decompose unambiguously subtrees occurring in spines in accordance with the equivalence relation '$\equiv$'. Practically this means that instead of considering all possible subtrees for all possible spines, we can compute cumulative weights for each representative of the equivalence relation (which are fewer although still of exponential cardinality).

Algorithm $\textsc{count}(n, \boldsymbol{p}, s)$ in Algorithm 1 enumerates spines of BDDs and, at the same time, computes their cumulated weights. It takes as arguments a size n for considering all subtrees of size n, assuming an initial pool profile $\boldsymbol{p} = (p_0, \dots, p_{k-1})$, level rank s and index k for the root of these trees. It returns in an associative array a list of pairs $(\boldsymbol{t}, w)$ where

- $\boldsymbol{t} = (t_0, t_1, \dots, t_{k-1}, t_k)$ ranges over the set of profiles of trees in $\mathcal{T}_{m,\boldsymbol{p},s}$, i.e., $\boldsymbol{t} \in \{\text{profile}(T) \mid T \in \mathcal{T}_{m,\boldsymbol{p}=(p_0,\dots,p_{k-1}),s}\}$.
- w is the sum over all equivalent trees of size m with profile $\boldsymbol{t}$ of their cumulated weights when the root has pool profile $\boldsymbol{p}$ and level rank s (which gives enough information to compute the cumulated weight for each tree using Proposition 2).

Note that any subtree T with a root of index i has a profile $\boldsymbol{t} = (t_0, \dots, t_i)$ with $t_0 = 0$ and $t_i = 1$.

Proposition 5. *The number $N(n, k)$ of* BDD*s of size n and of index k is computed thanks to Algorithm* COUNT() *and is equal to*

$$N(n, k) = \sum_{(\boldsymbol{t},w) \in \textsc{count}(n-2,\,(2,0,\dots,0),\,0)} w,$$

where $(2, 0, \dots, 0)$ has k components and corresponds with a pool reduced to the two sink nodes $\bot$ and $\top$ of index 0.

Proof. Indeed $\boldsymbol{t}$ ranges over all possible profiles for spines of size $(n-2)$ and we sum the weights of all spines for these profiles. Hence we compute exactly the number of BDDs of size n. □

An important refinement for this algorithm is to remark when summing over all spines, we consider subtrees of the same size whose root shares the same pool profile and same level rank, hence the same context. In order to avoid performing the same exact computations twice (or more) we can use *memoization* technique (that is storing intermediary results). It is an important trick to reduce the time complexity, although at the cost of some memory consumption.

Complexity of the counting algorithm First, we remark the numbers involved in the computations are (very) big numbers (as seen before, of order 2^{2^k}).

Proposition 6. *The complexity (in the number of arithmetic operations) of the computations of the Algorithm 1 to evaluate $N(n, k)$ is $O\left(\frac{1}{k} 2^{3k^2/2+k}\right)$.*

For Boolean functions in k variables, although the time complexity of our algorithm is of exponential growth $2^{3k^2/2}$. However the state space of Boolean functions is 2^{2^k} thus our computation is still much better than the exhaustive construction.

Algorithm 1 Algorithm COUNT(). The initial pool profile of the root (of index k) is $(2, 0, \ldots, 0)$ of length k.

```
function COUNT($n, \boldsymbol{p} = (p_0, \ldots, p_{k-1}), s$)
    $d \leftarrow \{\,\}$                                                        ▷ Empty dictionary
    if $n = 0$ then
        $S \leftarrow \left(\sum_{j=0}^{k-1} p_j\right) \cdot \left(\sum_{j=0}^{k-1} p_j - 1\right)$
        if $S > 0$ then $d \leftarrow \{\boldsymbol{e}^{(k)} : S\}$                                ▷ See *
    else
        for $i \leftarrow 0$ to $n-1$ do                         ▷ Left/right subtrees of size $i/n-i-1$
            $d_0 \leftarrow \{\,\}$
            if $i = 0$ then $d_0 \leftarrow \left\{\boldsymbol{\epsilon} : \sum_{i=0}^{k-1} p_i\right\}$          ▷ left subtree is empty, see *
            else
                for $k_0 \leftarrow 1$ to $k-1$ do                          ▷ left node has index $k_0$
                    $d_0 \leftarrow d_0 \cup$ COUNT($i, (p_0, \ldots, p_{k_0-2}), p_{k_0-1}$)
            for $(\boldsymbol{\ell}, w_0) \leftarrow d_0$ do
                $d_1 \leftarrow \{\,\}$
                $\boldsymbol{p'} \leftarrow \boldsymbol{p} + \boldsymbol{\ell}$
                if $n-1-i = 0$ then $d_1 \leftarrow d_1 \cup \left\{\boldsymbol{\epsilon} : -1 + \sum_{i=0}^{k-1} p'_i\right\}$     ▷ right subtree is empty
                else
                    for $k_1 \leftarrow 1$ to $k-1$ do                     ▷ right node has index $k_1$
                        $d_1 \leftarrow d_1 \cup$ COUNT($n-1-i, (p'_0, \ldots, p'_{k_1-2}), p'_{k_1-1}$)
                for $(\boldsymbol{r}, w_1) \leftarrow d_1$ do
                    $w \leftarrow w_0 \cdot w_1$
                    $\boldsymbol{t} \leftarrow \boldsymbol{\ell} + \boldsymbol{r} + \boldsymbol{e}^{(k)}$                          ▷ index profile of the subtree
                    if $\boldsymbol{t} \in d$ then $d[\boldsymbol{t}] \leftarrow d[\boldsymbol{t}] + w$          ▷ update if $\boldsymbol{t}$ is already a key in $d$
                    else $d \leftarrow d \cup \{\boldsymbol{t} : w\}$                            ▷ $\boldsymbol{t}$ is a new key in $d$
    return $d$
```

* For an integer $k \geq 0$, the list $\boldsymbol{e}^{(k)} = (0, \ldots, 0, 1)$ is the list with $(k+1)$ components where the last entry is 1 and all others are 0. The empty list of size 0 is denoted $\boldsymbol{\epsilon}$.

Unranking BDDs Using the classical recursive method for the generation of structures [15] we base our generation approach on the combinatorial counting approach. Since the class of objects under study seems not admissible in the sense given in Analytic Combinatorics [6], we cannot directly apply the advanced techniques presented in [5] nor the approaches by Martínez and Molinero [11]. Thus we devise an unranking algorithm for BDDs and get as by-products algorithms for uniform random sampling and exhaustive generation.

The ranking/unranking techniques for objects of a combinatorial class $\mathcal{C}$ of size N consists in building a bijection between any $c \in \mathcal{C}$ and an integer (its *rank*) in the interval $[0\,..\,N-1]$ (if we starts from 0). This leads trivially to a uniform sampling algorithm by drawing uniformly first an integer and then building the corresponding object.

Proposition 7. *Once the pre-computations are done, the unranking (or uniform random sampling) algorithm needs $O\left(n \cdot |\mathcal{T}_{n,k}|\right)$ arithmetic operations to build a* BDD *of index k and size n.*

First, remark that the worst case happens when n is of order the largest possible size of a BDD over k variables $O(\frac{2^{k/2}}{k})$ (cf. [9, p. 102]) which corresponds to the generic case according to Fig. 2. Furthermore, the number of profiles is of order $2^{\frac{k^2}{2}}$. To generate a BDD given its rank, we first identify the correct profile of its spine (by enumeration). Then according to this target profile, recursively, for each node, we traverse at most all spines with this profile, in order to decompose the substructures in its left and right part, yielding the upper bound.

As a conclusion, note the process of enumerating, counting and sampling we introduced can be adapted to subclasses of functions (for instance those for which all variables are essential), but also to other strategies of compaction, like those used for Quasi-Reduced BDD*s and Zero-suppressed* BDD*s. A natural question is also to provide an algorithm enumerating valid spines and not all invalid ones as well to get more efficient enumeration and unranking algorithms for* ROBDD*s. These questions will be addressed in future work.*

Acknowledgement. We thank the anonymous reviewers whose comments and suggestions helped improve and clarify this manuscript.

References

1. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986)
2. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. SIGPLAN Not. **35**(9), 268–279 (2000)
3. Dybjer, P., Haiyan, Q., Takeyama, M.: Verifying Haskell Programs by Combining Testing and Proving. In: QSIC'03. pp. 272–279 (2003)
4. Flajolet, P., Sipala, P., Steyaert, J.M.: Analytic variations on the common subexpression problem. In: Automata, languages and programming (Coventry, 1990). Lecture Notes in Comput. Sci., vol. 443, pp. 220–234 (1990)
5. Flajolet, P., Zimmermann, P., Cutsem, B.V.: A calculus for the random generation of labelled combinatorial structures. Theor. Comput. Sci. **132**(2), 1–35 (1994)
6. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press, New York, NY, USA, 1 edn. (2009)
7. Genitrini, A., Gittenberger, B., Kauers, M., Wallner, M.: Asymptotic Enumeration of Compacted Binary Trees of Bounded Right Height. to appear in Journal of Combinatorial Theory, Series A (2020)
8. Gröpl, C., Prömel, H.J., Srivastav, A.: Ordered binary decision diagrams and the shannon effect. Discrete Applied Mathematics **142**(1), 67 – 85 (2004)
9. Knuth, D.E.: The Art of Computer Programming, Volume 4A, Combinatorial Algorithms. Addison-Wesley Professional (2011)
10. Marinov, D., Andoni, A., Daniliuc, D., Khurshid, S., Rinard, M.: An evaluation of exhaustive testing for data structures. Tech. rep., MIT -LCS-TR-921 (2003)
11. Martínez, C., Molinero, X.: A generic approach for the unranking of labeled combinatorial classes. Random Structures & Algorithms **19**(3-4), 472–497 (2001)
12. Newton, J., Verna, D.: A theoretical and numerical analysis of the worst-case size of reduced ordered binary decision diagrams. ACM TCL **20**(1), 6:1–6:36 (2019)
13. Vuillemin, J., Béal, F.: On the BDD of a Random Boolean Function. In: ASIAN'04. pp. 483–493 (2004)
14. Wegener, I.: Branching Programs and Binary Decision Diagrams. SIAM (2000)
15. Wilf, H.S., Nijenhuis, A.: Combinatorial algorithms: An update. SIAM (1989)