arXiv:1907.06743v1 [cs.DS] 15 Jul 2019

BINARY DECISION DIAGRAMS: FROM TREE COMPACTION TO SAMPLING

JULIEN CLÉMENT AND ANTOINE GENITRINI

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 (DAG) 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. For efficiency our algorithms rely on a precomputation step. Finally, we give some key ideas to extend the approach to other strategies of compaction, in relation with variants of BDDs (namely QBDDs and ZBDDs).

Keywords: Binary decision diagram (BDD); Data structure analysis; Unranking; Sampling; Uniform random generation; Tree compaction.

Date: July 17, 2019. This research was partially supported by the ANR MetACOnc project ANR-15-CE40-0014.

1. Introduction

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 trees [3]. Its takeoff has been so spectacular that many variants of compacted structures have been developed, and called through many acronyms like ROBDDs [4], OKFBDDs [7], QOBDDs [23], ZBDDs [18], and many others. While most of the data structures are central in the context of verification [24], they also appear, for example, in the context of cryptography [15]. Some specific classes are also relevant to strategies for the resolution of combinatorial problems (cf. [14, vol. 4]) like the classical satisfiability count problem.

One way to represent the different diagrams consists in their embedding as directed acyclic graphs 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 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 in the same order along any path from the source to any 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, and thus, once a structure is found for a fixed Boolean function, the problem of improving the structure has no sense anymore.

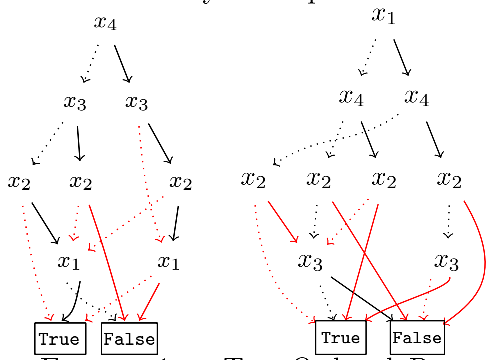

Figure 1. Two Ordered Reduced Binary Decision Diagrams associated to the same Boolean function. Nodes are labeled with Boolean variables; left dotted edges (resp. right solid edges) are `False` links (resp. `True` links)

In order to get smaller DAGs the constraint of the fixed variable ordering along any path can be relaxed as, for example, in the structures known as FREE-BDDs [12]. But in 1996, it was proved in the context of computational complexity that finding the optimal ordering is a NP-hard problem [1]. While several heuristics have been exhibited, e.g. [20] to find smaller structures, either by permuting variables or by using other varieties of BDDs [5, 8], the results have been only a qualified success. A possible way to improve the global understanding of the compacted structures consists in exhibiting their combinatorial properties. In his book [14] Knuth proves or recalls combinatorial results, like properties for the profile of a ROBDD, or the way to combine two structures to represent a more complex function. However, one notes an unseemly fact. To the best of our knowledge there are no results about the distribution of the Boolean functions according to their ROBDD size.

In this paper we study the compaction process that takes a binary decision tree and outputs its compacted form, the ROBDD. This combinatorial study allows then to exhibit the key points of the process and gives a way of construction for ROBDDs of a given size 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. Obviously due to the understanding of the total order for the structures, it can be biased it in order to generate structures with specific parameters (for example the left substructure of the BDD satisfying a given truth table).

Due to the state space in the context of Boolean functions, 2^{2^k} functions in k variables, the analysis of the distribution functions according to k and their size n, becomes rapidly unfeasible.

Our C++ implementation fully manages the case of 9 variables, that corresponds to approximately 10^{154} elements. In particular we obtain that for 9 Boolean variables, the number ROBDDs of size $n = 132$ equals (the number is split on two lines for convenience)

$$19328494569039846815842904086105185268201912361073744504659259374033691887255775 2359//$$

$$8884862388912050663541703532265322183008414844006128680960000000000000 \approx 10^{153},$$

that corresponds to approximately one sixth of all ROBDDs of 9 variables (the possible sizes range from 3 to 143). Furthermore, ROBDDs of size between 125 and 143 represent more than 99.8% of all ROBDDs.

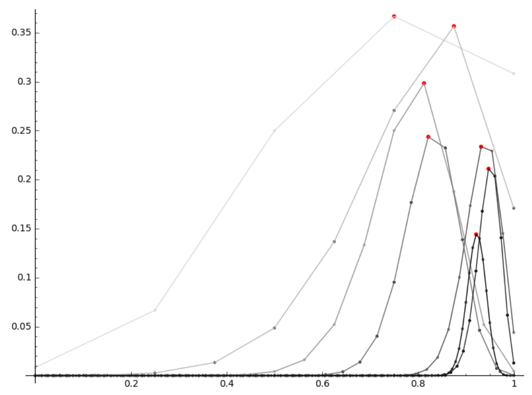

FIGURE 2. Normalized distributions of ROBDDs according to their number of variables and size

Some combinatorial studies on related questions on compression of trees have been investigated. For the case of binary trees (and other tree families representing for example arithmetic expressions), the average ratio of compaction has been first computed in the paper by Flajolet *et al.* [9]. The results have been completed later in [2, 19]. We also have started a combinatorial study devoted to the compacted binary tree structures in the paper [11].

In the following, Section 2 introduces the combinatorics underlying the decision tree compaction. Then, by using these results, Section 3 introduces a way to unambiguously specify the structure of reduced ordered binary decision diagrams. We apply this strategy in Section 4 to obtain an unranking algorithm for ROBDDS. Finally, in Section 5 we explain how to adapt the approach first to Boolean functions containing only essential variables, and then to other strategies of compaction which are of interest for applications (QBDDS or ZBDDS).

2. DECISION DIAGRAMS AS COMPACTED TREES

This section defines more precisely the context of the combinatorial structures we are interested in. Almost all definitions and facts are detailed in the dedicated volume [14] of Knuth.
Boolean functions. A Boolean function in k variables is a mapping from $\{\texttt{True}, \texttt{False}\}^k$ to $\{\texttt{True}, \texttt{False}\}$. We recall the classical result concerning the number of Boolean functions.

Fact 1. *Let $\mathcal{F}_k$ be the number of Boolean functions in k variables: $\mathcal{F}_k = 2^{2^k}$. The first terms of the sequence are* $2, 4, 16, 256, 65536, 4294967296, 18446744073709551616, \dots$

There are functions in k variables that do not really depend on the k variables, think e.g. to the projection $(x_1, x_2, \dots, x_k) \mapsto x_1$, whose evaluation is not modified if the truth value of some variable x_i $(i > 1)$ is modified. More formally, we define the notion of essential variable.

Definition 2. Let f be a Boolean function in k variables and x one of the variables. If the two functions in $(k-1)$ variables defined when the partial assignments for x are different, then x is an *essential variable* for f. Otherwise x is an *unessential variable* for f.

Through the literature we observe many possibilities to represent Boolean functions, like Boolean formulas in specific normal forms, Boolean circuits, truth tables, depending on the problem under consideration. Each representation induces an efficient way to solve a given problem, but usually it is not adapted for others. For example, to compute the Hamming weight of a function, i.e.

the number of assignments such that the functions computes `True`, the truth table is much more adequate than a representation as a circuit.

A typical operation is to evaluate a function for a given assignment of the variables. An efficient way regarding time complexity consists in using a full binary decision tree, however this solution is not space efficient. In the Figure 3 we have depicted, on the left hand-side, a decision tree of

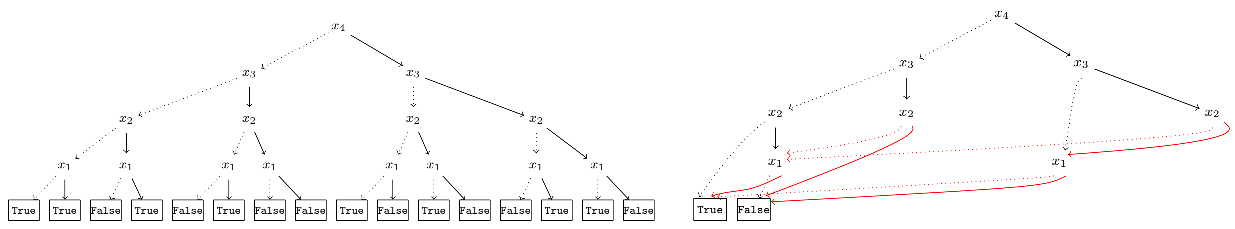

FIGURE 3. A decision tree and its compacted version

a Boolean function on 4 variables. For the assignment $x_1 = $ `True`$, x_2 = $ `False`$, x_3 = $ `False` and $x_4 = $ `True`, we follow the path starting in the root x_4, going to the right (using the solid edge – meaning that $x_4 = $ `True`) then leaving the node x_3 with the dotted edge (meaning $x_3 = $ `False`) and so on. Thus, for the latter assignment the function computes `False`. Remark that the sequence of labels of the leaves (from left to right) of the decision trees corresponds to the truth table of the function for a given order of the variables.
Compaction and decision diagrams. Given the full binary decision tree, the structure can be compacted by merging isomorphic substructures. More precisely, let us introduce the following compaction rules.

Definition 3. Let start with a binary decision tree, and let apply the following rules iteratively. The nodes M and N are distinct nodes in the structure under consideration:

- **M-rule:** if M and N are roots of isomorphic subgraphs, then all the incoming edges of N are replaced by pointers to M and N is deleted;
- **R-rule:** if N has both children pointing to the same node M, then all the incoming edges of N are replaced by pointers to M and N is deleted;

Fact 4. *The compaction rules* ***M*** *and* ***R*** *are confluent and the process of compaction terminates.*

Definition 5. A *reduced ordered binary decision diagram* (ROBDD) is the unique structure, induced by a given decision tree, applying to the extent possible the compaction rules **M** and **R**.

The ROBDD can be built from the decision tree with a single traversal of the tree. By applying this compaction process on the decision tree of Figure 3, we obtain the compacted structure corresponding to the ROBDD presented on the left handside of Figure 1.

In other contexts, this compaction procedure is classically called the *common subexpression recognition* and is used in particular through the compilation processes. As combinatorial studies in this context, for the case of binary trees (and other tree families representing for example arithmetic expressions), the average ratio of compaction has been first computed in the paper by Flajolet *et al.* [9]. Finally, we have started a combinatorial study devoted to the compacted binary tree structures in the paper [11].

Our goal in the rest of the section is to exhibit a strategy of compaction defined as a deterministic way of applying rules **M** and **R** in order to derive some combinatorial properties of the compacted structure. But first we need some further details.

Fact 6. *An ordering of the variables being fixed, each Boolean function is represented by exactly one single reduced ordered binary decision diagram.*

The result is almost direct, since for each ordering there is exactly one binary decision tree whose compaction (introduced in Definition 5) defines a single binary decision diagram.

In the rest of the paper, we define the variable ordering to be $x_k, x_{k-1}, \dots, x_1$. A reduced ordered binary decision diagram according to this ordering will from now be denoted by BDD. Due to this ordering, a BDD whose root is labeled by x_k is representing a Boolean function in k variables and for which the variable x_k is essential. Analogously to tree structures, we define the *root* of a BDD to be the single node of in-degree 0 and the two *leaves* to be the nodes of out-degree 0. They are labeled by the constants `True` and `False`.

Definition 7. For each BDD, its *size* is the number of nodes (including both leaves) it contains.

Note, for a given Boolean function, by using two distinct variable orderings, the two reduced ordered binary decision diagrams could be of different sizes. Figure 1 contains such an example.

Definition 8. Let B be a BDD with root-label x_k. For every internal node, labeled by x_i, we define its *index* to be the integer i and its *depth* to be the integer $k - i$. For each leaf (labeled by a constant), its index and depth are respectively 0 and k.

Abusively we define the index (resp. the depth) of a rooted substructure inside a BDD to be the index (resp. the depth) of its root. Remark that neither the index, nor the depth of a node of a BDD are necessarily equal to the longest path from that node to any leaf or to the root. Our definitions are such that the for each node, the sum of its index and its depth is always k. In Figure 1 the leftmost BDD has size 10, index 4, and it contains 3 distinct substructures of depth 2 and index 2 (all of them with a root labeled by x_2).

Our first goal is to determine a constructive recurrence to enumerate the BDDs combinatorial class. In order to get a non ambiguous construction of the BDD, we introduce a deterministic way for the compaction procedure: its major advantage is that it breaks the symmetry inside the BDD structure, and thus the building procedure is easier to describe.

We start with the set of decision trees in k variables, based on the ordering $x_k, x_{k-1}, \dots, x_1$, (thus supposing that x_k is an essential variable, i.e. both child-substructures of the root are distinct). Otherwise we focus on the subtree rooted at x_ℓ such that ℓ is the greatest index satisfying x_ℓ is an essential variable for the function. We construct the set of BDDs in the following way.

> `Each decision tree is traversed by using the` *`post-order traversal`*`. At the node` ν `under visit, its children have already been compacted as BDDs. If possible we apply rule R to the roots of the children of` ν `(thus` ν `disappears), otherwise if another node` μ `(already seen during the traversal), such that` ν `and` μ `are satisfying the rule M, we apply it and` ν `is removed. Finally we continue the traversal.`

Obviously at the end of the procedure, we obtain the BDD corresponding to the input decision tree. By applying this compaction process on the decision tree of Figure 3, we obtain the compacted structure presented on the right hand-side of the decision tree. This compacted structure is exactly the BDD presented in the left of Figure 1 with a (slightly) different formatting. In Figure 3 the emphasis is put on the compaction process through the post-order traversal: only the first occurrence of a BDD-substructure is kept in the final compacted structure. In the Figure 1, the formatting is the classical one for BDDs. The *forward pointers* in black are going to the children that have been first visited during the post-order traversal. The *backward pointers* in red are going to the first (and single-kept) occurrence of the child already seen during the traversal. Due to the post-order traversal, all pointers are going to the left, inside the structure. By definition of BDDs an edge starting at index i goes to a substructure of index j with $j < i$.

In Figure 4 we have depicted each step (building a new structure) in the compaction process.

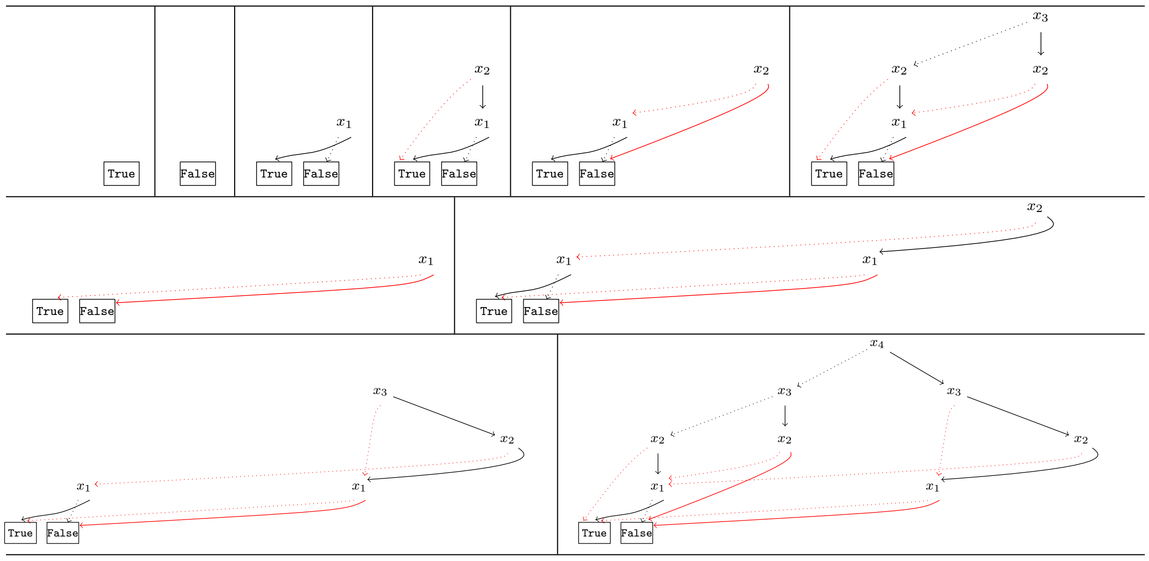

FIGURE 4. The building set associated to the decision tree of Figure 3

Definition 9. In a BDD, the binary tree containing all the nodes and induced by the forward (black) edges is called the *spine* of the BDD.

Remark that the spine of a BDD is a planar binary tree: each node has either 0, 1 or 2 children, where left (dotted) edges and right (solid) edges are distinguished.

Our first goal aims at giving an effective method to enumerate BDDs with a chosen number k of variables and size n. The naive algorithm to do this is the following:

- enumerate all the 2^{2^k} Boolean functions;
- apply the compaction procedure;
- finally filter the BDDs of size equal to the target size n.

We propose to give a combinatorial description of BDDs which translates to a recurrence, providing the basis for an enumeration algorithm avoiding the heavy enumeration of all Boolean functions on k variables.

3. Recursive decomposition

This section introduces a canonical and unambiguous decomposition of the BDDs in order to prove a recursive formula for their enumeration. Although we have introduced our process as a compaction of decision trees, this is just used as an artifact to easily visualize and understand the compacted structures and the new type of substructures interacting with the BDDs. But later on, to count or to sample BDDs, no decision tree will be used anymore.

In order to recall some classical properties but also to introduce new ones for the BDDs seen as combinatorial structures, we highlight some results in the Appendix 6.1. However these results are not necessary to follow the core of the paper, thus the reader can decide to skip them.

We are now ready to introduce the notion of profile of a BDD that corresponds informally to its size distribution.

Definition 10. The *profile* of a BDD of index $k \geqslant 1$ is an integer vector such that the value of the i-th component, $i \in \{0, 1, \dots, k-1\}$, is the number of nodes labeled by x_{k-i} in the BDD; furthermore a last component is appended and is always 2 because both constants do appear in any BDD.

In our running example depicted in left hand-side of Figure 1 the profile is $(1, 2, 3, 2, 2)$.

Let us now highlight an important fact in the BDD structure that makes the combinatorial class of BDDs hard to enumerate. In the context of classical recursive structures, let us say binary trees for example, the structures can be decomposed as a root with two substructures that are themselves binary trees. It is not the case for BDDs.

Let us see a BDD as a compacted structure induced by a decision tree with the post-order traversal compaction. Obviously by looking to our example in the right hand-side of Figure 3, the BDD has its root labeled by x_4, its left substructure is a BDD (all pointers are going to substructures that are lying inside the BDD); but its right substructure is not closed in the following sense: some (red) pointers starting in the right substructure (at nodes labeled either by x_1 or x_2) are going to the left substructure.

This problematic recursive decomposition in mind we will decompose a BDD by focusing on the post-order traversal. During the compaction process we construct a set of BDDs. This set is called the *building set* (it can be seen as a dynamical set during the compaction process). At the beginning, the building set is the empty set. Due to the traversal, the two first BDDs that are traversed and that appear in the final compacted structure are the constants `True` and `False` (they can appear in different orders according to the decision tree). Remark the special case in our example: the leftmost structure in the decision tree rooted at x_1 does not appear in the BDD because both its leaves are equal to `True`), thus it is not added to the building set. During the traversal, each time a new BDD is created, i.e. it does not yet belong to the building set, it is added to it. Note the following important fact: once the two constants have been traversed, then *each time a new* BDD *is added to the building set, it is obtained by a combination of two* BDD*s already belonging to the building set.* Eventually, some substructures of these two BDDs are in common, and thus, obviously only the one of the BDD used as the left substructure is kept. And in the right substructure a red pointer (pointing to the relevant substructure that is kept) is added. See for example the sixth BDD (at the end of the first raw) added in the building set in Figure 4, to observe the new red pointers in the right substructure.

Note, once the compaction process is finished, the number of BDDs in the building set is exactly the size of the complete BDD: each element of the building set contributes for 1 to the BDD-size, in fact in each BDD of the building set, the single new node appearing is its root. Finally, the compaction process being finished, each BDD defines an unique building set: the single BDD with the largest index is the complete BDD.

In order to distinguish sub-BDDs that use red pointers, we define the following dedicated notion.

Definition 11. Let B be a BDD of index $k \geqslant 1$ and ν one of its nodes of index h. The *pool* associated to ν is the following set of BDDs. Let Δ_ν be the (fringe) subtree rooted at ν in the spine of B. For all nodes μ with index at most h and visited before every nodes of Δ_ν during the post-order traversal of the spine of B, the BDD, rooted at μ from the building set, is added to the pool of ν. Furthermore, as an extreme case, both constants `True` and `False` are supposed to always belong to the pool.
The vector $\mathbf{p}$ of integers, called the *pool-vector*, with $\mathbf{p} = (p_h, p_{h+1}, \dots, p_{k-1}, 2)$, is such that p_{h+i} is the number of BDDs of index $(h + i)$ in the pool.

For example, in the BDD in the left hand-side of Figure 1, the root x_4 is such that its subtree Δ_{x_4} contains the whole spine and thus it is associated to the pool containing only the two constants, thus its pool-vector is $(0, 0, 0, 0, 2)$. The rightmost node x_3 is such that its subtree contains the three rightmost internal nodes and thus is associated to a pool whose vector is $(1, 2, 1, 2)$. And the second node x_2 (in the middle) is associated to $(1, 1, 2)$.

Obviously, the pool associated to a node ν can be computed during the compaction process when ν is visited (thanks to the post-order traversal). The (complete) spine only simplifies the formalization of the pool definition, but it is not necessary.

Our goal consists in enumerating and building BDDs avoiding the compaction process, thus we now take a constructive point of view while still focusing on the left/right dissymmetry induced by the post-order traversal of the BDD.

Proposition 12. *The left substructure of the root of a* BDD *contains all the information related to the pool of the root of the right substructure of the* BDD*.*

The constructive approach is obtained through an adaptation of the notion of a pool, disconnecting it from the compaction process. In fact, note that the building set can be directly deduced from a BDD, without considering its associated decision tree. We define an extension for the pools.

Definition 13. A pool $\mathcal{P}$ is a set of BDDs corresponding to the union of the building sets of some BDDs.

A pool cannot be empty, it contains at least both constants. Furthermore, remark that a constraint could be relaxed and $\mathcal{P}$ could simply be a set of BDDs, but to achieve our goal, the constrained definition is sufficient.

Definition 14. Let k be a positive integer and $\mathcal{P}$ a pool. A *pool*-BDD is a BDD such that some of its pointers (drawn in red) can point to BDDs belonging to $\mathcal{P}$.
If a substructure of the pool-BDD has an occurrence in the pool, it must necessarily be replaced by a pointer to the relevant BDD from the pool.
The *size* of a pool-BDD is the number of nodes in its spine (linked through black edges, in particular, its spine does not contain any constant).

The notion of colored edges could be avoided, it does not bring any information in the enumeration problem. However it facilitates the understanding of the structure composition. We are now ready to define the unambiguous structural decomposition of a pool-BDD relying on the post-order traversal.

Theorem 15. *Let B be a pool-*BDD *of index $k \geqslant 1$ associated to a pool $\mathcal{P}$. Let us denote by ν the root of B and by n its size (with $n > 0$). The left substructure B_ℓ, attached to ν, of index $\tilde{k}$ and of size ℓ, is a pool-*BDD *with pool $\tilde{\mathcal{P}}$ containing all the* BDD*s of $\mathcal{P}$ of index at most $\tilde{k}$, and the right substructure attached to ν, of index $\bar{k}$ is a pool-*BDD *of size $(n-1-\ell)$ with pool containing all* BDD*s of index at most $\bar{k}$ in the union of $\mathcal{P}$ and the building set of B_ℓ. In the special cases when ℓ is 0, then B_ℓ is just a pointer from ν to a* BDD *in $\mathcal{P}$ and when $\ell = n-1$, then B_r is a pointer.*

key-ideas. The fundamental idea is that a BDD present in the pool cannot be duplicated. Furthermore, when a red pointer is used to point an already seen BDD, or one present in the pool, the pointer goes to the left (by supposing the pool to be pictured in the left of the pool-BDD) and the destination node is deeper than the source node. □

Note that a BDD is nothing else than a specific pool-BDD.

Corollary 16. *A* BDD *of size $(n+2)$ and index k is a pool-*BDD *of size n, with index k and relying on the pool $\{\mathtt{True}, \mathtt{False}\}$.*

From now we are interested in the enumeration problem related to the set of BDDs: we aim at determining how many BDDs of index k and size $(n+2)$ there are.

Proposition 17. *Let $\mathcal{P}$ and $\mathcal{Q}$ be two pools whose pool-vectors are both equal (denoted by $\boldsymbol{p}$). The number of pool-*BDD*s based on $\mathcal{P}$, of size $n \geqslant 1$, and index $k \geqslant 1$, is equal to the one of pool-*BDD*s based on $\mathcal{Q}$ (of size n and index k).*

The basic idea is that the combinatorics in the context of BDDs is induced by the spine structure and the pointers starting in unary nodes and leaves.

key-ideas. Let S be the spine of a pool-BDD that is built by using $\mathcal{P}$. The number of pool-BDDs built by using $\mathcal{P}$ and having the spine S is only dependent on the numbers of structures encoded in $\mathbf{p}$, thus the latter is equal to the number of pool-BDDs built with $\mathcal{Q}$ that have the same spine S. By summing over all possible spines we prove the result. □

We thus are able to partition pool-BDDs according to the set of pool-vectors.

Notation 3.1. *We define $B_{n,\boldsymbol{p}}$ to be the number of pool-BDDs, of size $n \geqslant 1$ and index k, based on all pools $\mathcal{P}$ whose pool-vectors are equal to $\boldsymbol{p}$.*

Obviously the number $B_{n,(0,\dots,0,2)}$, (the pool-profile $(0,\dots,0,2)$ containing $(k+1)$ components) is equal to the number of BDDs of size $(n+2)$ and index k.

Proposition 18. *Summing the numbers of BDD of all possible sizes for indices up to k, we get*

$$\mathcal{F}_k - 2 = \sum_{i=1}^{k} \sum_{n=1}^{m_i} B_{n,(0,\dots,0,2)},$$

the upper bounds, denoted by m_i, are defined in Fact 24 and the pool-profiles $(0,\dots,0,2)$ contain $(i+1)$ components.

Proof. The expression $\mathcal{F}_k - 2$ is the number of Boolean functions on k variables with at least one of them being essential (the term -2 corresponds to the forbidden constant functions). Let f be one of this function, we define i to be the maximal index such that x_i is essential for f. Then f is represented by a BDD of index exactly i. The function is counted (exactly once) in $\sum_{n=1}^{m_i} B_{n,\mathbf{p}}$ with the pool-vectors $\mathbf{p}$ being $(0,\dots,0,2)$ and containing $(i+1)$ components. □

Definition 19. Let $k \geqslant \tilde{k} \geqslant 1$ be two integers, and a pool $\mathcal{P}$ whose pool-vector is $\mathbf{p} = (p_0,\dots,p_{k-1},2)$. A pool-BDD of index $\tilde{k}$ and pool $\tilde{\mathcal{P}}$ (containing all BDDs of $\mathcal{P}$ of index at most $\tilde{k}$) has the following profile $(p_{\tilde{k}} + \alpha_{\tilde{k}},\dots,p_{k-1}+\alpha_{k-1},2)$ where the α_i's are the numbers of nodes labeled by x_{k-i} in the spine of the pool-BDD.

The latter notion is the extension to pool-BDDs of the profile of a BDD (introduced in the Definition 10). Remark that eventually not all elements of the pool are pointed by the nodes from the spine of the pool-BDD. And recall that the size of the pool-BDD is $\alpha_{\tilde{k}} + \cdots + \alpha_{k-1}$.

We are now ready to enter the last stage to reach our goal consisting in the enumeration problem. But first let us introduce some useful notations.

Notations 3.1. *Let $\boldsymbol{p}$ and $\boldsymbol{q}$ be two integer vectors respectively equal to $(p_0,\dots,p_r)$ and $(q_0,\dots,q_s)$. We define the* concatenation *of the vectors $(\boldsymbol{p},\boldsymbol{q})$ to be the vector $(p_0,\dots,p_r,q_0,\dots,q_s)$. For two vectors of the same size (here suppose $r = s$) we define the* sum *$\boldsymbol{p}+\boldsymbol{q}$ as $(p_0+q_0,\dots,p_r+q_r)$. The* prefix *of length $(h+1)$ of the vector $\boldsymbol{p}$ is denoted by $\boldsymbol{p}^h$ and defined as $(p_0,\dots,p_h)$. The* suffix *of length $(h+1)$ and denoted by $\boldsymbol{p}_h$ and $(p_{r-h},\dots,p_r)$. Finally, the* norm *of $\boldsymbol{p}$ is denoted by $|\boldsymbol{p}|$ and is equal to $p_0+\cdots+p_r$.*

Let us now partition the set of pool-BDDs according to their profiles. Let $\mathbf{p}$ and $\mathbf{q}$ be two $(k+1)$-components vectors. The number of size-n pool-BDDs with pools $\mathcal{P}$ (whose pool-vectors are equal to $\mathbf{p}$) and profiles $\mathbf{q}$ is denoted by $B^{\mathbf{q}}_{n,\mathbf{p}}$. In particular the number of BDDs of size $(n+2)$, index k and profile $\mathbf{q}$ is $B^{\mathbf{q}}_{n,(0,\dots,0,2)}$.

Proposition 20. *Let $n \geqslant 1$ and $k \geqslant 1$ be two integers and a pool-vector $\boldsymbol{p} = (p_0,\dots,p_{k-1},2)$. The set of profiles of size-n pool-BDDs of index k, based on a pool with pool-vector $\boldsymbol{p}$, is denoted by $\mathcal{P}_{n,\boldsymbol{p}}$, and is partitioned as*

$$\mathcal{P}_{n,\boldsymbol{p}} = \Big\{(p_0+1, p_1+n_1,\dots,p_{k-1}+n_{k-1},2) \mid \quad \text{with } |\boldsymbol{n}| = n \quad \text{and} \quad \forall i, \quad 1 \leqslant i \leqslant k-1,$$
$$0 \leqslant n_i \leqslant \min\left\{2^i, r_i\cdot(r_i-1)-p_i, \tfrac{n-s_i+1}{2}\right\}\Big\},$$

where we denote $\boldsymbol{n} = (1, n_1, \dots, n_{k-1}, 0)$ *and for all* i, $1 \leqslant i \leqslant k-1$, *we define* $r_i = |\boldsymbol{p}_{k-i-1} + \boldsymbol{n}_{k-i-1}|$ *and* $s_i = |\boldsymbol{n}_{k-i-1}|$.

The detailed proof is given in Appendix 6.1. We are now ready to state the main theorem that exhibits a combinatorial recurrence satisfied by the enumeration of pool-BDDs.

Theorem 21. *Let* $k \geqslant 1$ *be a positive integer and* $\boldsymbol{p} = (p_0, \dots, p_{k-1}, 2)$ *a vector containing* $(k+1)$ *non-negative integers. For* $n = 1$, $1 \leqslant h \leqslant k$ *and* $\boldsymbol{q} = (q_{k-h}, \dots, q_{k-1}, 2)$, *we get*

$$B_{1,\boldsymbol{q}} = |\boldsymbol{q}_{h-1}| \cdot \big(|\boldsymbol{q}_{h-1}| - 1\big) - q_{k-h}.$$

For $n > 1$,

$$\begin{aligned} B_{n,\boldsymbol{p}} =& |\boldsymbol{p}_{k-1}| \cdot \sum_{r=\lceil \log_2(n) \rceil}^{k-1} B_{n-1,\boldsymbol{p}_r} + \sum_{i=1}^{n-2} \sum_{\ell=\lceil \log_2(i+1) \rceil}^{k-1} \sum_{\boldsymbol{q} \in \mathcal{P}_{i,\boldsymbol{p}_\ell}} B^{\boldsymbol{q}}_{i,\boldsymbol{p}_\ell} \cdot \sum_{r=\lceil \log_2(n-i) \rceil}^{k-1} B_{n-1-i,(\boldsymbol{p}^{k-\ell-1},\boldsymbol{q})_r} \\ &+ (n-2+|\boldsymbol{p}_{k-1}|) \cdot \sum_{\ell=\lceil \log_2(n) \rceil}^{k-1} B_{n-1,\boldsymbol{p}_\ell}. \end{aligned}$$

Proof. The number $B_{1,\mathbf{q}}$ enumerates the pool-BDDs of size 1 and index h. Its value corresponds to all possibilities of pointing a pair of BDDs of index less than h from the pool minus the pairs that already appear in the pool and that are counted by q_{k-h}.

When the size n is greater than 1, we are interested in $B_{n,\mathbf{p}}$, the number of size-n pool-BDDs of index k. We decompose the size n in i and $(n-1-i)$ to be respectively the size for the left substructure and the one for the right one.
First when $i = 0$ the left substructure is reduced to a pointer (we have $|\mathbf{p}_{k-1}|$ possibilities) and the size of the right substructure is positive, thus we cannot build a pool-BDD already present in the pool. The right substructure is of size $(n-1)$ and of index r which varies between $\lceil \log_2(n) \rceil$ corresponding to the full binary spine (as deep as possible) and $(k-1)$ the maximal possible index. When $1 \leqslant i < n-1$, then both substructures have a positive size. The index ℓ describes all possibilities for the index of the left substructure (from the full binary spine, as deep as possible, to the maximal possible index). In order to calculate the new available pointers possibilities for the right substructure, we pay attention to the profile $\mathbf{q}$ of the left subtructure: thus we use the numbers $B^{\mathbf{q}}_{i,\mathbf{p}_\ell}$. The index r describes the range of indices for the right substructure. The pool $(\mathbf{p}^{k-\ell-1}, \mathbf{q})_r$ corresponds to the $r+1$ last components because the right substructure is of index r. Finally, the case when $i = n-1$ is such that the right substructure is reduced to a pointer. The left substructure is like in the previous case a pool-BDD, here of size $(n-1)$ and we do not care about its profile, since the right pointer can point (almost) everywhere in the left substructure or in the pool, thus only the whole size is important $(n-2+|\mathbf{p}_{k-1}|)$: the pointer cannot point to the left-structure attached to the root. $\square$

In Figure 2 we represent the normalized distributions of these sequences for $3 \leqslant k \leqslant 9$. The color is darker when k becomes larger, and the red point is the highest value for each k. In particular, when $k = 9$ more than 14% of the BDDs have size 132 (the maximal one being 143), and the proportion of functions with BDD-size larger than 124 is greater than 99.8%. Thus even for small k, the proportions of functions with a big BDD are in accordance with the asymptotics [21].

4. Uniform random sampling through the unranking method

Using the classical recursive method for the generation of structures [25] we base our generation approach on the recurrences exhibited in Theorem 21. Since the class of objects under study in not decomposable in the Analytic Combinatorics's way, we cannot not directly apply the advanced techniques presented in [10] nor the approaches by Martínez and Molinero [16, 17]. We are interested in the unranking generation of BDDs and as by-products we obtain algorithms for the uniform random sampling and the exhaustive generation.

Algorithm 1 Unranking the BDD of a given rank

```
    function GENERATE(rank, n, avail_nodes, current_node_id, forbid_pairs,
                      target_profile)
 1: bdd_index := length(avail_nodes)                ▷ maximal index of the already built nodes
 2: pool_profile := GET_PROFILE(forbid_pairs)                           ▷ cf. fct GET_PROFILE
 3: max_ind := length(pool_profile) - 1                   ▷ index of the node under construction
                          ▷ Step 1: Managing both extreme cases
 4: if n = 0 then
 5:     return (node_id, forbid_pairs, [avail_nodes[max_ind+1][rank]])
                                                                   ▷ builds a pointer to the
                                                                    ▷ pool-BDD with rank rank
 6: if n = 1 then
 7:     (F, x, y) := UNRANK_PAIR(rank, avail_nodes[max_ind], forbid_pairs[max_ind])  ▷ cf. fct UNRANK_PAIR
 8:     forbid_pairs[max_ind] := F                                 ▷ updates the forbidden pairs
 9:     for i from max_ind +1 to bdd_index-1 do                    ▷ updates the available nodes
10:         avail_nodes[i].append(node_id)                         ▷ for greater indices
11:     return (node_id+1, forbid_pairs, [node_id, [x], [y], max_ind])
                                                           ▷ builds the new node with label
                                                           ▷ node_id and pointers to x and y
                          ▷ Step 2: Managing the classical case: n > 1
12: (r, i, h_left, profile_left, h_right)
             := DECOMPOSE(rank, n, pool_profile, target_profile)             ▷ cf. DECOMPOSE
13: max_left := ENUM_BDD(i, pool_profile[:h_left+1])[profile_left]            ▷ cf. ENUM_BDD
14: r_left := r % max_left                       ▷ determines the rank of the left and right
15: r_right := r // max_left                     ▷ substructure by using an Euclidean division
                          ▷ Step 3: Building of the left subBDD
16: (new_node_id, new_forbid_pairs, subBDD_left)
      := GENERATE(r_left, i, avail_nodes, node_id, forbid_pairs[:h_left+1], profile_left)
17: new_forbid_pairs := EXTEND(new_forbid_pairs, forbid_pairs)
                                                                   ▷ updates the forbidden pairs
                          ▷ Step 4: Building of the right subBDD
18: (new_node_id, new_forbid_pairs_right, subBDD_right)
      := GENERATE(r_right, n-1-i, avail_nodes, new_node_id,
             new_forbid_pairs[:h_right+1], target_profile[:h_right+1])
19: new_forbid_pairs := EXTEND(new_forbid_pairs_right, new_forbid_pairs)
                                                                   ▷ updates the forbidden pairs
                          ▷ Step 5: Building of the result of the function
20: BDD := [new_node_id, subBDD_left, subBDD_right, max_ind]
21: for h from length(pool_profile) to length(avail_nodes)-1 do            ▷ updates the
22:     avail_nodes[h].append(new_node_id)                                 ▷ available nodes
23:     avail_nodes[h] := sort(avail_nodes[h])
24: new_forbid_pairs[max_ind] := INSERT_PAIR((subBDD_left[0], subBDD_right[0]),
                                   new_forbid_pairs[max_ind])
25: return (new_node_id+1, new_forbid_pairs, BDD)
```

The unranking method needs the definition of a total order over the class of objects under consideration: here we consider the BDDs of given index and size. Once the order is set, it remains to choose a rank (i.e. an integer between 0 and the total number of objects), and then to built the single object associated to this rank. Obviously, when we are interested in the sampling, we cannot afford the building of all the objects from rank 0 to $r-1$ in order to build the one of rank r. Thus we must find a method to construct the object with rank r directly by using the combinatorics associated to the total order and arithmetic properties of r. More precisely, once the unranking algorithm is given, to obtain the exhaustive generation, it remains to build successively the objects of all ranks. Due to the memoization during the unranking of a BDD, the exhaustive generation is efficient. Also to obtain an uniform random sampler, we just need to sample uniformly the rank of an object, and then to build it.

In order to associate a rank to each object (with index k and size n) of the combinatorial class under consideration we must know how many objects there are. Thus unranking algorithms need a first step of pre-computations, done only once. Furthermore here we will use a recursive approach, thus, as usually (see [10]) we must compute the whole distribution of the numbers of objects up to index k and size n. But, since our objects are not decomposable in the sense of [10], these numbers do not contain sufficiently information for the sampling. The study of Theorem 21 reveals the necessity of a complete calculations of the numbers $B_{\nu,\mathbf{p}}$, for all $\nu = 1, \ldots, (n-1)$ and all possible profiles $\mathbf{p}$ (with $(k+1)$ components).

In Theorem 21, the way of presentation of the recursive formula for $B_{\nu,\mathbf{p}}$ is adapted to the reader, however technically it is not the best way for driving all the calculations. Thus the algorithmic framework we will describe is not a direct implementation of the previous theorems.

First a memoization of the intermediate results is necessary to avoid the same computations to be done a huge number of times. Furthermore, in order to compute all the $B_{\nu,\mathbf{p}}$ requested for a given n and k, the direct application of the formula of Theorem 21 needs to filter all profiles $\mathbf{q}$ valid for the left substructure and that can be combined with a right substructure to obtain the whole structure with pool $\mathbf{p}$. But finally we are enumerating all positive $B_{\nu,\mathbf{p}}$, thus the filtering operation is not necessary: for any profile $\mathbf{q}$ we just have to increment the suitable value of $B_{\nu,\tilde{\mathbf{p}}}$ for the right profile $\tilde{\mathbf{p}}$. Thus it is much more efficient to enumerate all left structure profiles of any size (and partition them according to their pool) and then to enumerate all right structure profiles of any size according to the possible pools obtained through the enumeration of the left structures. Finally it remains to partition the results according to the complete pool of whole pool-BDDs.

Proposition 22. *The complexity (in the number of arithmetic operations) of the pre-computations necessary to evaluate $B_{n,\boldsymbol{p}}$ is $O\left(2^{\frac{3}{2}k^2+k}/k\right)$, where k is the index of $\boldsymbol{p}$.*

First, note that the numbers we are computing with are (very) big numbers (as seen before, of order 2^{2^k}). Furthermore, if we are interested in the BDDs for Boolean functions in k variables, we thus call an algorithm of exponential growth in $2^{3/2\,k^2}$, but recall that the state space of Boolean functions is 2^{2^k} thus our computation is much better than an exhaustive construction of all possibilities.

key-ideas. We first compute the number P of profiles for all possible pools up to k variables and size m_k defined in Fact 24 as the maximal size for BDDs of size k. Then we must compute the number of pool-BDDs with given pool and profile, thus we obtain P^2 as an upper bound (but all possibilities are not often possible). Finally, for each size and profile we traverse all profiles and for each possibility one multiplication is done: thus the total complexity is $O(m_k \cdot P^3)$.

A first crude upper bound for P is the following. For each size n, we compute a weak composition of n to insert in all possibilities n among a profile of $k+1$ components. This gives $P = 2^{k^2}$. In order to obtain the stated value, we are using the following upper bound: for depth i the numbers of nodes in the profile is between 0 and 2^i (this is too much for the greatest values of i as we have proved in Proposition 20). This is related to both sequences `OEIS`[1] `A028361, A131791` whose asymptotic behavior closes the proof (with $P = 2^{\frac{k^2}{2}}$). □

Proposition 23. *Once the pre-computations are done, the unranking (or uniform random sampling) algorithm needs $O\left(n \cdot |\mathcal{P}_{n,(0,\ldots,0,2)}|\right)$ arithmetic operations to build a* BDD[2] *of index k and size n.*

[1]`OEIS` is the Online Encyclopedia of Integer Sequences: `http://oeis.org`.
[2] The vector $(0,\ldots,0,2)$ contains $(k+1)$-components.

First of all, remark that the worst case is when n is of order $m_k = O\left(2^{\frac{k}{2}}/k\right)$ and this is the case most of the time according to Figure 2. Furthermore is this context, the number of profiles if of order $2^{\frac{k^2}{2}}$. The key-idea is the following, for each of the n recursive calls, we must traverse at most all profiles in order to decompose the substructures in its left and right part. In the Appendix 6.2 the input and output of the function Generate are described and also the sub-function that are called by Generate.

5. Conclusion

As a conclusion a special highlight of the method we used must be given. First, by adapting the combinatorial specification, thus the unranking algorithm, we manage to select the subclass of Boolean functions whose variables are all essential. Their study is exhibited in Appendix 6.3.1. Furthermore, our combinatorial analysis of the compaction process underlying BDDs can be generalized to study other kind of structures resulting of some compaction process. In the last part of the Appendices (see 6.3.2 and 6.3.3) we underline this fact by exhibiting the same kind of results for other kinds of decision diagrams, usually called QBDDs or ZBDDs.

Finally in the same vein of study, a combinatorially analysis of the compaction of tries would be a possible further work. Recall that tries, or digital trees are very common data structures, and their tree-structure is now well known, cf. e.g. [6].

6. Appendix

Let us start this Appendix section with the representation of two BDDs, uniformly sampled at random, among BDDs of index 8 and size, respectively 62 et 79. The latter is of maximal size, thus each level is full.

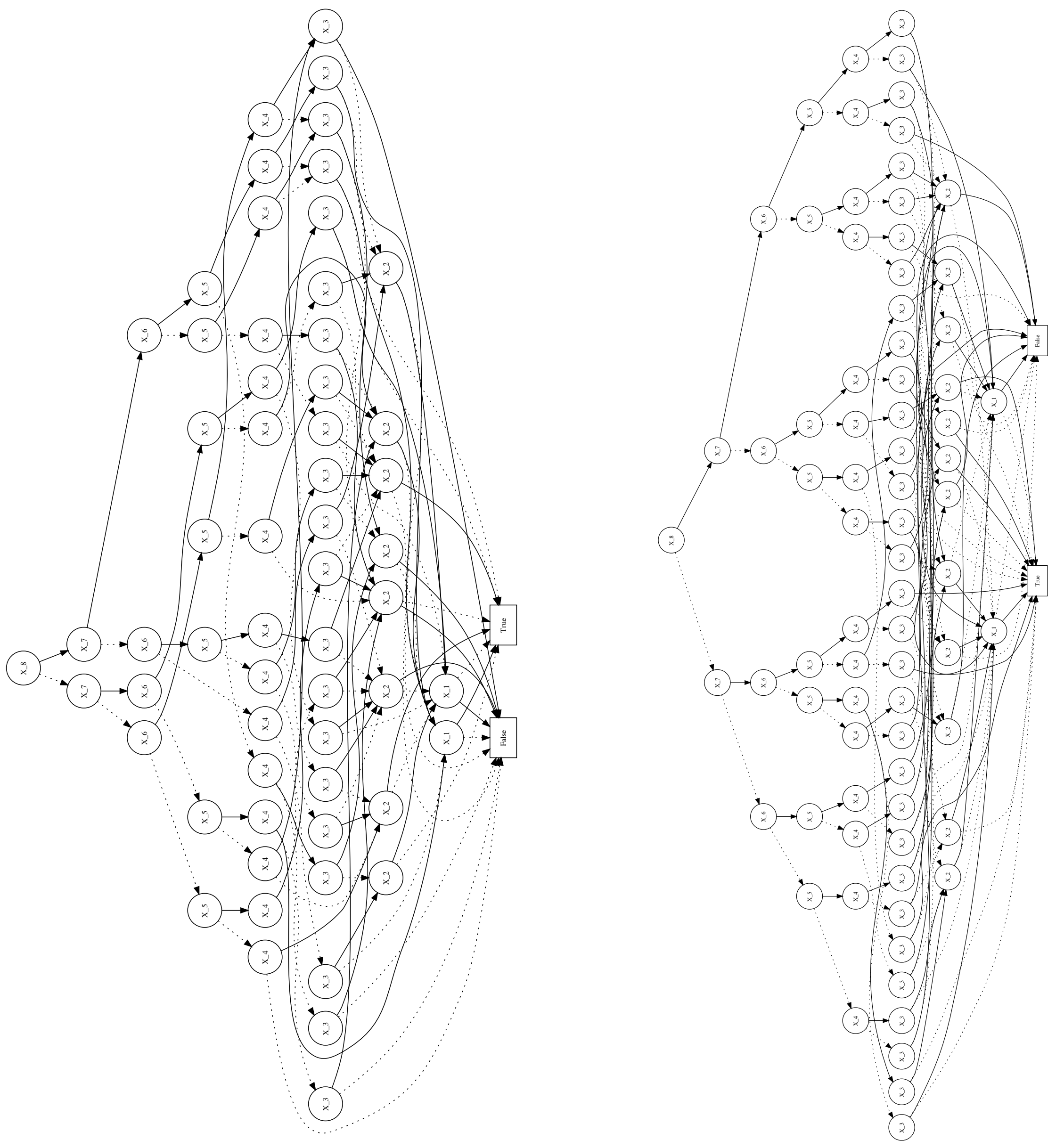

Figure 5. Two uniformly sampled BDDs of index 8

6.1. **Appendix related to Section 3: Recursive decomposition.** We first describe some combinatorial results, most of them are already known (the references are given below) but some are new results.

For small values of k, one can easily compute the maximal sizes of the BDDs of index k. In particular, we get the maximal sizes of the BDDs for index $1 \leqslant k \leqslant 10$:

$$(3, 5, 7, 11, 19, 31, 47, 79, 143, 271, \dots)$$

Remark that this sequence has not yet been stored in `OEIS`. The following fact has been proved by Knuth in [14, p. 234].

Fact 24. *Let $k \geqslant 1$. The maximal size of a BDD of index k, denoted by m_k, is*

$$m_k = 2^{2^{\lfloor \rho_k \rfloor}} + 2^{k - \lfloor \rho_k \rfloor} - 1,$$

with ρ_k the solution of $2^{2^{x-1}} \left(2^{2^{x-1}} - 1\right) = 2^{k-x}$.

In order to give a taste of the rest of the section, here we present the key-ideas of the proof even it is detailed in [14]. The main idea here consists in putting in each depth the maximum number of nodes as possible (i.e. such that the main constraints of a BDD are not violated). Then it remains to prove that such a profile does exist for some BDD.

Proof. We are interested in calculating the maximal size of a BDD of index k. In order to get a structure as large as possible, each level contains as much as possible of nodes. For each index i, we must take into account two structural constraints. The nodes under consideration must be attached as leaves in the binary tree (ie. the spine) obtained by cutting the spine of the BDD at the previous level. Thus from the top point of view, the maximal number of nodes of index i is 2^{k-i}. From the bottom point of view we cannot create more nodes than the numbers of pairs of distinct BDD built deeper in the structure. Thus we introduce the following sequence $u_0 = 2$ and $u_{n+1} = u_n \cdot (u_n - 1)$ that enumerates the possible pairs, in particular, at index i the important term is u_{k-i}. Thus we obtain the following cumulating value: $m_k = \sum_{i=1}^{k} \ell_i$, ,with $\ell_1 = 3$ and for $i > 1$, $\ell_i = \min\left\{ \left(2 + \sum_{j=1}^{i-1} \ell_j\right) \cdot \left(1 + \sum_{j=1}^{i-1} \ell_j\right), 2^{k-i} \right\}$. Finally, ρ_k is the threshold where the minimal value determining ℓ_i goes from one expression to the other. □

From the latter result we deduce the behavior of the maximal size of a BDD on k variables, when k tends to infinity. This asymptotic result has already been settled in [22, 21], but with distinct proofs.

Corollary 25. *Asymptotically, when k is large the threshold ρ_k satisfies*

$$\rho_k \underset{k\to\infty}{=} \sum_{\ell=0}^{L} \frac{P_\ell(\ln k)}{k^\ell \ \ln^{\ell+1} 2} + O\left(\frac{\ln^L k}{k^L}\right),$$

where the P_ℓ's are computable polynomials of degree ℓ. In particular

$$\rho_k \underset{k\to\infty}{=} \frac{\ln k}{\ln 2} - \frac{\ln k}{k \ln^2 2} - \frac{\ln^2 k}{2k^2 \ \ln^3 2} + \frac{\ln k}{k^2 \ \ln^3 2} - \frac{\ln^3 k}{3k^3 \ \ln^4 2} + \frac{3 \ln^2 k}{2k^3 \ \ln^4 2} - \frac{\ln k}{k^3 \ \ln^4 2} + O\left(\frac{\ln^4 k}{k^4}\right).$$

Thus we get $m_k = \Theta\left(\frac{2^{k+1}}{k}\right)$.

The asymptotic behavior of m_k was already known (cf. [21]), but we present a new proof of this result in the Appendix 6.1.

Proof. First we prove the asymptotic development of ρ_k, which is the key-result for the behavior of the numbers m_k.

Studying the equation satisfied by ρ_k we first exhibit the first order of its asymptotic behavior, when k tends to infinity:

$$\rho_k \underset{k\to\infty}{=} \frac{\ln k}{\ln 2} + o(1).$$

We then obtain the next term by bootstrapping in the equation, $2^{2^{x-1}}\left(2^{2^{x-1}}-1\right) = 2^{k-x}$, and obtaining each time a more precise result for ρ_k. Finally, due to the shape of the equation, the shape of the asymptotic development of ρ_k is direct.

Using the behavior of ρ_k we deduce the one of m_k. But, because of the floor and ceiling function appearing in the expression of m_k, the sequence $(m_k)_k$ has no limit when k tends to infinity. □

We are now interested in the whole profile of BDDs and pool-BDDs.

of Proposition 20. There are two distinct results to prove in this statement. Let us first prove the upper bound for each n_i and in a second time that all described profiles do appear among pool-BDD of index k and pool $\mathbf{p}$.

Let i be an integer in $\{1, \dots, k-1\}$. The number n_i corresponds to i-th depth of the pool-BDD, i.e. the depth containing the labels x_{k-i}.
The value 2^i is the maximal number of nodes at level i in a complete binary tree (thus in a pool-BDD also).
We define r_i to be $2 + \sum_{j=i+1}^{k-1}(p_j + n_j)$, that corresponds to the total number of pool-BDD either belonging to the pool or constructed earlier in the post-order traversal (with index smaller than i). Obviously the expression $r_i \cdot (r_i - 1)$ is the total number of possible pairs of BDDs of index at most i. To that value we must remove the number of pairs already built in the pool.
The upper bound $\frac{n-s_i+1}{2}$ is related to the fact that the new BDDs, outside the pool, must be connected at the end of the post-order traversal, i.e. in the global pool-BDD. Let us define $s_i = \sum_{j=i+1}^{k-1} n_j$, it corresponds to the new BDDs built in the greater levels (for smaller depths). When focusing on structures in the level i we get the following constraint: $n_i \leqslant n - s_i - n_i + 1$. In fact by building n_i BDDs it must remains enough nodes, i.e. $(n - s_i - n_i)$, for the levels above i to connect these nodes of level i. Here, BDDs in level i can be seen as the leaves of a binary tree designed by the levels above. This justifies the bound $n_i \leqslant n - s_i - n_i + 1$.
Thus, the minimum of the three latter bounds is an upper bound for value n_i.

It remains to prove that for each profile in $\mathcal{P}_{n,\mathbf{p}}$, is the profile of at least one pool-BDD. A proof by induction on the index of the poll-BDD is direct. □

2											
2	8	2									
2	16	60	88	74							
2	24	174	872	3174	8928	17666	23280	11160			
2	32	344	2960	21692	139904	801608	4054160	18027338	69381840	223877520	…
						…572592240	1074728520	1281360960	806420160	223534080	19958400

FIGURE 6. Distribution of BDDs according to their number of variables ($\geqslant 1$) and size ($\geqslant 3$)

In Figure 6 we give the first distribution of BDDs according to their index and size. In the latter, the value in the row $1 \leqslant k \leqslant 5$ (the row $k = 5$ is divided on two lines) and column $1 \leqslant n \leqslant 17$ corresponds to the number of BDDs of index k and size $(n + 2)$.

6.2. **Appendix related to Section 4: Uniform random sampling through the unranking method.** In order to unrank a pool-BDD, we call the function GENERATE that need the 6 following inputs.

(1) *rank*: the rank of the pool-BDD we target to build;
(2) n: the size of the pool-BDD we are building (in fact, the real size of the built object is $n+2$ because we do not count the two constants in the algorithm);
(3) *avail_nodes*: a list of lists of the identifiers; at index h the list contains all identifiers of BDDs whose index is larger or equal to h (for efficiency reason we could only save in the h-th list the identifiers of BDDs whose index is exactly h);
(4) *current_node_id*: the identifier for the next node (it corresponds to a fresh positive integer, the smaller not already used);
(5) $forbid\_pairs$: the list of list of pairs of identifiers, the h-th element of the list is a the list of pairs of identifiers that already are the left and right substructures of a BDD from the pool with index h;
(6) $target\_profile$: the profile of the pool-BDD under construction.

The output of the function GENERATE is the triple of the following values:

(1) *new_node_id*: the next fresh positive integer (no yet used);
(2) new_forbid_pairs: the update of the list $forbid\_pairs$;
(3) BDD: an encoding of the pool-BDD that has been built through that call to the function GENERATE

Let us introduce the following example of a call to the function GENERATE.
The call GENERATE$(14781, 8, [[0,1],[0,1],[0,1],[0,1],[0,1]], 2, [[],[],[],[],[]], (1,2,3,2,2))$ produces the following tuple:

$$\begin{aligned}&(10,\\&[[(5,8)],[(3,4),(6,7)],[(1,2),(2,0),(2,6)],[(0,1),(1,0)],[]],\\&[9,[5,[3,[1],[2,[0],[1],1],2],[4,[2],[0],2],3],[8,[6,[1],[0],1],[7,[2],[6],2],3],4])\end{aligned}$$

The two first elements of the tuple are necessary for the recursive calls. The last element is an encoding a the spine of the BDD, with some extra information. The later encoding wan be represented as the following DAG. In the encoding, each node is represented as a list either of 4 elements or of a single one. In the case of 4 elements, the first one is the identifier of the node, the two following elements are respectively the root-node of the left substructure and the right one. The last element is the index of the current node. If the list associated to the node contains a single element it is the identifier of the root-node of a BDD that has already been construct during the prefix traversal.

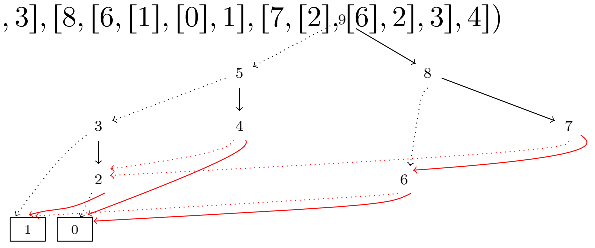

FIGURE 7. Encoding of the BDD from Figure 3

Let us finally introduce the sub-function that are called trough the function GENERATE.

- GET_PROFILE: function that builds the profile of a BDD based on the list of pairs that have already been built;
- UNRANK_PAIR: function that builds the pair (x,y) of a given rank, avoiding already existing pairs and updating the latter list;
- DECOMPOSE: function that exhibits information of the substructures of a BDD of given $rank, pool\_profile$ and $target\_profile$;
- EXTEND: function that prepends a list by the prefix of the second one in order that both have the same length.

6.3. **Extensions to other strategies of compaction.** By adapting our approach, we can easily adapt our approach in order to cover subclasses of Boolean functions. Here for example, we exhibit how to consider only Boolean functions on k variables such that all of them are essentials. In a second time we extent our results to other classes of decision diagrams, in particular quasi reduced BDD (QRBDD), or a zero suppressed BDD (ZBDD).

6.3.1. *Functions with only essential variables.* We can completely adapt Section 3 and in particular obtain extensions for Theorem 21. This section is devoted to function based only on essential variables. Recall the Definition 2.

Fact 26. *Let $\mathcal{E}_k$ be respectively the number of Boolean functions in k variables such that all k variables are essential, then:*

$$\mathcal{E}_k = \sum_{i=0}^{k} (-1)^i \binom{k}{i} 2^{2^{k-i}}.$$

The first terms of the sequence are $2, 2, 10, 218, 64594, 4294642034, 18446744047940725978, \dots$

The fact can be proved by using the inclusion-exclusion principle (see [13, p. 69]).

Fact 27. *Let f be a Boolean function in k variables, x one of the variables and T a decision tree representing f. The variable x is essential for f if and only if there exists a subtree in T, rooted at a node ν labeled by x, such that both children of ν are roots of distinct subtrees.*

In the following we are interested in BDDs of index k where all k variables are essential. Obviously such pool-BDDs have no component equal to 0 in their profile.

Proposition 28. *The number of size-$(n+2)$ BDD of index k, with k essential variables is $E_{n,(0,\dots,0,2)}$ and*

$$\mathcal{E}_k = \sum_{n=k}^{m_k} E_{n,(0,\dots,0,2)},$$

the pool vector $(0, \dots, 0, 2)$ containing $(k+1)$ components.

In order to adapt Theorem 21 to the case of pool-BDDs with only essential variables, we introduce the following notation.

Notation 6.1. *The number of size-n pool-BDD of index k with k essential variables, with pool vector $\boldsymbol{p}$ and profile $\boldsymbol{q}$ is denoted by $E^{\boldsymbol{q}}_{n,\boldsymbol{p}}$.*

Theorem 29. *Let $k \geqslant 1$ be a positive integer and $\boldsymbol{p} = (p_0, \dots, p_{k-1}, 2)$ a vector containing $(k+1)$ non-negative integers. For $n = 1$, $1 \leqslant h \leqslant k$ and $\boldsymbol{q} = (q_{k-h}, \dots, q_{k-1}, 2)$, we get*

$$E_{1,\boldsymbol{q}} = |\boldsymbol{q}_{h-1}| \cdot \big(|\boldsymbol{q}_{h-1}| - 1\big) - q_{k-h}, \quad \text{if } \forall i \in \{1, \dots, h-1\},\ q_{k-i} > 0, \qquad \text{and otherwise } E_{1,\boldsymbol{q}} = 0.$$

For $n > 1$,

$$E_{n,\boldsymbol{p}} = |\boldsymbol{p}_{k-1}| \cdot \sum_{r=\max\{h_{\boldsymbol{p}}, \lceil \log_2(n) \rceil\}}^{k-1} E_{n-1,\boldsymbol{p}_r} + \sum_{i=1}^{n-2} \sum_{\ell=\lceil \log_2(i+1) \rceil}^{k-1} \sum_{\boldsymbol{q} \in \mathcal{P}_{i,\boldsymbol{p}_\ell}} B^{\boldsymbol{q}}_{i,\boldsymbol{p}_\ell} \cdot \sum_{r=\max\left\{ \substack{h_{(\boldsymbol{p}^{k-\ell-2},\boldsymbol{q})}, \\ \lceil \log_2(n-i) \rceil} \right\}}^{k-1} E_{n-1-i,(\boldsymbol{p}^{k-\ell-1},\boldsymbol{q})_r}$$
$$+ (n - 2 + |\boldsymbol{p}_{k-1}|) \cdot \sum_{\ell=\max\{h_{\boldsymbol{p}}, \lceil \log_2(n) \rceil\}}^{k-1} E_{n-1,\boldsymbol{p}_\ell}.$$

where $h_{\boldsymbol{q}}$ is the difference between k and the smallest positive index of the components equal to 0 in $\boldsymbol{q}$, if it exists, otherwise $h_{\boldsymbol{q}} = 0$.

Key-ideas. The difference between Theorems 21 and 29 is the fact that in the latter case the profiles of the whole structures have only non-zero components. This is ensured by mixing substructures that are either classical pool-BDDs and other ones that are enforced to have positive coefficients for the indices where that are not used in the left substructure. Furthermore, we avoid to construct substructures deep in the whole BDD and whose result would necessarily omit some variables. □

6.3.2. *Quasi-Reduced BDDs.* Let start with a binary decision tree, and let apply some of the following rules iteratively. The nodes M and N are distinct nodes in the structure under consideration:

- **M-rule:** if M and N are roots of isomorphic subgraphs, then all the incoming edges of N are replaced by pointers to M and N is deleted;
- **R-rule:** if N has both children pointing to the same node M, then all the incoming edges of N are replaced by pointers to M and N is deleted;
- **Z-rule:** if N has its right outgoing edge pointing to `False`, then all its incoming edges are replaced by pointers to the left child of N and N is deleted.

Recall, by taking a binary decision diagram and applying as long as possible both **M-rule** or **R-rule**, we obtain the ROBDD. Otherwise, by applying to the extent possible only the **M-rule** we obtain tte QRBDD. Or, by applying to the extent possible both **M-rule** or **Z-rule**, we obtain the ZBDD.

Abusively in the following we use the same notations as before, even we are studying new structures. In particular the number of Quasi-Reduced BDDs of size $(n+2)$, index k and profile $\mathbf{q}$ is $B^{\mathbf{q}}_{n,(0,\dots,0,2)}$, and the total number of Quasi-Reduced BDDs of size $(n+2)$ and index k is $B_{n,(0,\dots,0,2)}$.

Proposition 30. *Let $n \geqslant 1$ and $k \geqslant 1$ be two integers and the pool-vector $\boldsymbol{p} = (p_0, \dots, p_{k-1}, 2)$. The set of profiles of size-n pool-BDDs of index k, based on a pools with pool-vectors $\boldsymbol{p}$, is denoted by $\mathcal{P}_{n,\boldsymbol{p}}$, and is partitioned as*

$$\mathcal{P}_{n,\boldsymbol{p}} = \Bigg\{ (p_0+1, p_1+n_1, \dots, p_{k-1}+n_{k-1}, 2) \mid \qquad \textit{with } |\boldsymbol{n}| = n \quad \textit{and} \quad \forall i, \quad 1 \leqslant i \leqslant k-1,$$

$$\left\lceil \frac{n_{i+1}}{2} \right\rceil \leqslant n_i \leqslant \min\left\{2^i, (p_{i+1}+n_{i+1})^2 - p_i, \frac{n - s_i + 1}{2}\right\} \Bigg\},$$

where we denote $\boldsymbol{n} = (1, n_1, \dots, n_{k-1}, 0)$ and for all i, $s_i = |\boldsymbol{n}_{k-i-1}|$.

Theorem 31. *Let $k \geqslant 1$ be a positive integer and $\boldsymbol{p} = (p_0, \dots, p_{k-1}, 2)$ a vector containing $(k+1)$ non-negative integers.*
For $n = 1$, $1 \leqslant h \leqslant k$ and $\boldsymbol{q} = (q_{k-h}, \dots, q_{k-1}, 2)$, we get

$$B_{1,\boldsymbol{q}} = q^2_{k-1-h} - q_{k-h}.$$

For $n > 1$,

$$B_{n,\boldsymbol{p}} = p_1 \cdot B_{n-1,\boldsymbol{p}_{k-1}} + \sum_{i=1}^{n-2} \sum_{\boldsymbol{q} \in \mathcal{P}_{i,\boldsymbol{p}_{k-1}}} B^{\boldsymbol{q}}_{i,\boldsymbol{p}_{k-1}} \cdot B_{n-1-i,((p_0),\boldsymbol{q})} + (p_1+1) \cdot B_{n-1,\boldsymbol{p}_{k-1}}.$$

Remark: here, for each index k we get exactly one structure for each Boolean function of at most k variables.

Corollary 32. *A Quasi-Reduced* BDD *of size $(n+2)$ and index k is pool-Quasi-Reduced-*BDD *of size n, index k and relying on the pool reduced to $\{$**`True`**$,$**`False`**$\}$, with pool-vector $(0, \dots, 0, 2)$.*

4										
0	4	12								
0	0	4	12	72	144	24				
0	0	0	4	12	72	576	1752	10656	20736	31728

FIGURE 8. Distribution of Quasi-Reduced BDDs according to their number of variables ($\geqslant 1$) and size ($\geqslant 3$)

6.3.3. *Zero-suppressed BDDs.* The combinatorial meaning of the graph is distinct from the one of BDDs. Here we can have only one constant in the representation of non constant functions. Also the notion of profile is a bit technical because of the fact that either only the `True` constant does appear. Thus the size itself must be study with care. But in order to calculate the distribution of the ZBDDs, we can take as un upper bound of the set of profiles the profiles of binary trees. Then y recurrence, if a profile is not valid, automatically the number of ZBDDs with such profile will be equal to zero. Thus the application of the following results is completely possible as it is also explained in Section 4 for BDDs.

Proposition 33. *Let $k \geqslant 1$ be a positive integer and $\boldsymbol{p} = (p_0, \dots, p_{k-1}, p_k)$ a vector containing $(k+1)$ non-negative integers.*
For $n = 1$, $1 \leqslant h \leqslant k$ and $\boldsymbol{q} = (q_{k-h}, \dots, q_{k-1}, q_k)$, we get

$$\tilde{B}_{1,\boldsymbol{q}} = \begin{cases} |\boldsymbol{q}_{h-1}| \cdot \left(|\boldsymbol{q}_{h-1}| - 1\right) - q_{k-h} & \textit{if } q_k = 2 \\ |\boldsymbol{q}_{h-1}|^2 - q_{k-h} & \textit{if } q_k = 1. \end{cases}$$

For $n > 1$,

$$\tilde{B}_{n,\boldsymbol{p}} = |\boldsymbol{p}_{k-1}| \cdot \sum_{r=\lceil \log_2(n) \rceil}^{k-1} \tilde{B}_{n-1,\boldsymbol{p}_r} + \sum_{i=1}^{n-2} \sum_{\ell=\lceil \log_2(i+1) \rceil}^{k-1} \sum_{\boldsymbol{q} \in \mathcal{P}_{i,\boldsymbol{p}_\ell}} \tilde{B}^{\boldsymbol{q}}_{i,\boldsymbol{p}_\ell} \cdot \sum_{r=\lceil \log_2(n-i) \rceil}^{k-1} \tilde{B}_{n-1-i,(\boldsymbol{p}^{k-\ell-1},\boldsymbol{q})_r}$$
$$+ (n - p_k + |\boldsymbol{p}_{k-1}|) \cdot \sum_{\ell=\lceil \log_2(n) \rceil}^{k-1} \tilde{B}_{n-1,\boldsymbol{p}_\ell}.$$

Theorem 34. *Let $n \geqslant 2$ and $k \geqslant 1$. The number of Zero-suppressed-BDD of size n and level k are as follows. The Z-BDD that do not contain the **`False`** constant are counted by*

$$B_{n,(0,\dots,0,1)} = \tilde{B}_{n-1,(0,\dots,0,1)}.$$

The Z-BDD that do contain both constants are counted by

$$B_{n,(0,\dots,0,2)} = \tilde{B}_{n-2,(0,\dots,0,2)} - \tilde{B}_{n-2,(0,\dots,0,1)}.$$

1	1								
1	4	5	2						
1	7	27	55	76	74				
1	10	66	314	1137	3414	8568	17354	23256	11160

FIGURE 9. Distribution of Zero-suppressed BDDs according to their number of variables ($\geqslant 1$) and size ($\geqslant 2$)

REFERENCES

[1] B. Bollig and I. Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[2] M. Bousquet-Mélou, M. Lohrey, S. Maneth, and E. Noeth. Xml compression via directed acyclic graphs. *Theory of Computing Systems*, 57(4):1322–1371, 2015.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[4] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[5] R. E. Bryant. Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '95, pages 236–243. IEEE Computer Society, 1995.

[6] J. Clément, P. Flajolet, and B. Vallée. Dynamical Sources in Information Theory: A General Analysis of Trie Structures. *Algorithmica*, 29(1):307–369, 2001.

[7] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski. Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decision Diagrams. In *31st Design Automation Conference*, pages 415–419, 1994.

[8] R. Ebendt and R. Drechsler. Lower bounds for dynamic bdd reordering. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 579–582. ACM, 2005.

[9] P. Flajolet, P. Sipala, and J.-M. Steyaert. Analytic variations on the common subexpression problem. In *Automata, languages and programming (Coventry, 1990)*, volume 443 of *Lecture Notes in Comput. Sci.*, pages 220–234. Springer, New York, 1990.

[10] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theor. Comput. Sci.*, 132(2):1–35, 1994.

[11] A. Genitrini, B. Gittenberger, M. Kauers, and M. Wallner. Asymptotic enumeration of compacted binary trees, 2016, under submission.

[12] W. Gunther and R. Drechsler. Minimization of free bdds. In *Proceedings of the ASP-DAC '99 Asia and South Pacific Design Automation Conference 1999 (Cat. No.99EX198)*, volume 1, pages 323–326, 1999.

[13] A. Kheyfits. *A Primer in Combinatorics.* De Gruyter Textbook. De Gruyter, 2010.

[14] D. E. Knuth. *The Art of Computer Programming, Volume 4A, Combinatorial Algorithms.* Addison-Wesley Professional, 2011.

[15] L. Kruger, S. Jha, E.-J. Goh, and D. Boneh. Secure Function Evaluation with Ordered Binary Decision Diagrams. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 410–420, New York, NY, USA, 2006. ACM.

[16] C. Martínez and X. Molinero. A generic approach for the unranking of labeled combinatorial classes. *Random Structures & Algorithms*, 19(3-4):472–497, 2001.

[17] C. Martínez and X. Molinero. Generic algorithms for the generation of combinatorial objects. In *MFCS'03*, pages 572–581. Springer Berlin Heidelberg, 2003.

[18] S.-I. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. *30th ACM/IEEE Design Automation Conference*, pages 272–277, 1993.

[19] D. Ralaivaosaona and S. Wagner. Repeated fringe subtrees in random rooted trees. In *2015 Proceedings of the Twelfth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 78–88. SIAM, Philadelphia, PA, 2015.

[20] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '93, pages 42–47. IEEE Computer Society Press, 1993.

[21] J. Vuillemin and Fr. Béal. On the bdd of a random boolean function. In *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, pages 483–493. Springer Berlin Heidelberg, 2005.

[22] I. Wegener. *The Complexity of Boolean Functions.* John Wiley & Sons, Inc., 1987.

[23] I. Wegener. The size of reduced obdds and optimal read-once branching programs for almost all boolean functions. In *Graph-Theoretic Concepts in Computer Science*, pages 252–263. Springer Berlin Heidelberg, 1994.

[24] I. Wegener. *Branching Programs and Binary Decision Diagrams.* Society for Industrial and Applied Mathematics, 2000.

[25] H. S. Wilf and A. Nijenhuis. *Combinatorial algorithms: An update.* CBMS-NSF. Philadelphia, Pa. Society for Industrial and Applied Mathematics, 1989.

GREYC, CNRS, UMR 6072, Université de Caen, 14032 Caen, France.
E-mail address: `Julien.Clement@unicaen.fr`

Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6 -LIP6- UMR 7606, F-75005 Paris, France.
E-mail address: `Antoine.Genitrini@lip6.fr`