Algorithm xxx: Evaluating a Boolean Polynomial on All Possible Inputs

Charles Bouillaguet

Algorithm xxx: Evaluating a Boolean Polynomial on All Possible Inputs

CHARLES BOUILLAGUET, Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

Evaluating a Boolean polynomial on all possible inputs (i.e. building the truth table of the corresponding Boolean function) is a simple computational problem that sometimes appears inside broader applications, for instance in cryptanalysis or in the implementation of more sophisticated algorithms to solve Boolean polynomial systems.

Two techniques share the crown to perform this task : the "Fast Exhaustive Search" (FES) algorithm from 2010 (which is based on Gray Codes) and the space-efficient Moebius transform from 2021 (which is reminiscent of the FFT). Both require $O(d2^n)$ operations for a degree-d Boolean polynomial on n variables and operate mostly in-place, but have other slightly different characteristics. They both provide an efficient iterator over the full truth table.

This article describes BeanPolE (BoolEAN POLynomial Evaluation), a concise and flexible C library that implements both algorithms, as well as many other functions to deal with Boolean multivariate polynomials in dense representation.

CCS Concepts: • **Mathematics of computing** → *Mathematical software*; • **Computing methodologies** → **Representation of polynomials**; **Boolean algebra algorithms**.

Additional Key Words and Phrases: Boolean polynomials, exhaustive search, Moebius transform, software implementation

ACM Reference Format:
Charles Bouillaguet. 2018. Algorithm xxx: Evaluating a Boolean Polynomial on All Possible Inputs. *J. ACM* 37, 4, Article 111 (August 2018), 36 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

We consider the problem of efficiently evaluating a Boolean polynomial, given by its coefficients, on all possible values of the input variables. This provides a way to build the truth table of the corresponding Boolean function. Any Boolean function

$$\begin{array}{rccc} f: & \{0,1\}^n & \to & \{0,1\} \\ & (x_0,\ldots,x_{n-1}) & \mapsto & f\,(x_0,\ldots,x_{n-1}) \end{array}$$

on n variables can be completely described by providing its *truth table*, namely the array of 2^n bits that contain its value on each of the possible values of the n input variables.

Low-degree Boolean polynomials admit a much more compact representation. For instance, a Boolean quadratic polynomial

$$f(x_0,\ldots,x_{n-1}) = \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} a_{ij}x_ix_j + \sum_{i=0}^{n-1} b_ix_i + c$$

Author's address: Charles Bouillaguet, charles.bouillaguet@lip6.fr, Sorbonne Université, CNRS, LIP6, F-75005 Paris, 4 place Jussieu, Paris, France, 75252.

0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

is entirely described by the values of its $n(n+1)/2+1$ coefficients. In general, a degree-d Boolean polynomial has $O(n^d)$ coefficients. As such, low-degree Boolean polynomials are easier to manipulate than generic Boolean functions, even with a high number of variables.

Our focus is on the case of dense Boolean polynomials with a potentially high number of variables (n) but a relatively low degree (d). Typical sizes that can be practically relevant are $n \approx 80$ and $d = 5$. A Boolean polynomial of this size can be represented in about 3MB in memory, yet storing its full truth table would require 2^{80} bits, and that is strictly impossible.

This article describes the BeanPolE library (BoolEAN POLynomial Evaluation), that offers a toolbox to deal with such Boolean polynomials. It is written in plain C for maximum portability. Besides a handful of useful low-level functions to manipulate dense multivariate Boolean polynomials, it implements two different algorithms to *visit* all entries of the truth table of a degree-d polynomial f in n variables given by its coefficients. In other terms, it offers efficient ways to iterate over pairs $(x, f(x))$ for all bit strings $x \in \{0,1\}^n$. These two procedures are not only efficient but quite frugal: they require only $O(n)$ words of extra memory in addition to the space needed to store the input polynomial f.

The first of these procedures is a slight variation of the "Fast Exhaustive Search" (FES) algorithm [BCC+10]. The second is a slight variation of the "Space-Efficient Moebius Transform" [Din21a]. Both require $O(d2^n)$ operations to enumerate the full truth table of a degree-d polynomial in n variables (2^n is an obvious lower-bound on the running-time, as it is the size of the output). Three main applications of these algorithms are known to the author:

(1) Solving systems of Boolean polynomial equations by exhaustive search. The FES algorithm was designed for this purpose. This NP-complete problem is practically relevant because the security of several digital signature schemes rely on the assumption that it is intractable [CHR+16, DS05, CFMR+17, BP17]. One of these schemes was practically broken by a cryptographic attack that reduced the problem to a smaller subsystem; it could be then be solved using the FES algorithm [DDVY21].
(2) Use as a sub-component inside more sophisticated Boolean solvers. Many improved algorithm have been proposed to solve systems of Boolean polynomial equations (mostly of degree two) [BFSS13, LPT+17, JV17, BKW19, Din21b, Din21a, BDT22]. All of them combine exhaustive search with other techniques. Specifically, they need to either completely evaluate some polynomials over all possible inputs, or to partially evaluate them (*i.e.* obtain $\mathbf{x} \mapsto f(\mathbf{x}, \mathbf{y})$ for all possible values of the variables in $\mathbf{y}$).
(3) Direct computation of the truth table. This is required for instance in some cryptographic attacks against symmetric primitives. A relevant example is provided by the recent and practical attack of [BDL+21] against the GPRS Encryption Algorithms GEA-1 and GEA-2, used in "2G cellphones". This repeatedly builds the truth table of several degree-4 polynomials in 33 variables. Other examples include [DS11] (degree-6 polynomials on 32 variables) and [DRS20] (degree-4 polynomials on $\approx$ 128 variables).

2 RELATED SOFTWARE PACKAGES

To the best of our knowledge, there is no software library devoted to the manipulation of dense Boolean Polynomials.

There are software libraries devoted to the analysis of (arbitrary) Boolean functions such as VBF [ACZ16]. It allows in particular the conversion of the Algebraic Normal Form of a Boolean function (its representation by the coefficients of a polynomial) into the truth table and vice-versa. These libraries are usually limited to a small number of variables, if only because the truth table has

to fit entirely in memory. Their typical use case is $n = 8$ variables, which matches sizes commonly used in cryptographic constructions.

The MQSoft library [FPR19] implements several operations over $\mathbb{F}_{2^n}$ that are required for the implementation of some "post-quantum" digital signature schemes bases on the hardness of solving systems of Boolean polynomial equations. In particular, it implements an efficient procedure to evaluate a collection of quadratic Boolean polynomials on a single arbitrary input. This is designed to handle a few hundred variables.

The GF2X library [BGTZ08] manipulates univariate polynomials over $\mathbb{F}_2$, in dense representation (one bit per coefficient). It implements asymptotically fast multiplication algorithms in particular.

The PolyBori library [BD09] is devoted to the algebraic manipulation of sparse Boolean polynomials in a large number of variables. It is used for this purpose inside the SageMath computer algebra system [The23]. PolyBori is also used as a foundation in some Boolean solvers such as [ZZL+21].

The FES algorithm was implemented several times by the author with the objective of maximum speed. The libfes-lite[1] library is the latest iteration. It is restricted to *quadratic* Boolean systems (the most relevant case in cryptology). In terms of usability, it provides a standalone multi-threaded program that reads a Boolean polynomial system from a text file in a simple format and prints its solutions. It is the fastest CPU-only implementation of exhaustive search for Boolean quadratic polynomials at the time of this writing. However, because it is restricted to quadratic polynomials only, it is not suited to most of the applications discussed in the introduction.

It must be noted that although exhaustive search used to be the most practical method to solve unstructured (random) dense Boolean polynomial systems, that is no longer the case. The "Crossbred" algorithm [JV17], combining exhaustive search and algebraic techniques, has been demonstrated to be significantly faster in practice. It requires the evaluation of a large number of Boolean polynomials on all possible inputs and therefore can be implemented on top of the algorithms presented in this article.

The BeanPolE library does not aim for maximum speed, but for simplicity, flexibility and ease of reuse in more complex software packages.

3 PRELIMINARIES

In the sequel, we will often omit the word "Boolean". Polynomials, monomials, variables, etc. are all Boolean. Given an array A, we occasionally use the "slice notation" $\mathsf{A}[i:j]$, as found in Python, to denote the sub-array $\mathsf{A}[i], \mathsf{A}[i+1], \ldots, \mathsf{A}[j-1]$.

3.1 Bit strings and Integers

Bit strings are elements of $\{0,1\}^*$. The i-th symbol of a bit string a is denoted as a_i. We simply denote by ab the concatenation of two bit strings a and b. Similarly, if a is a bit string, we write $a^k = aaa\ldots a$ where a is repeated k times.

Any non-negative integer can be written in base two as $i = (\ldots a_3 a_2 a_1 a_0)_2 = \sum_k a_k 2^k$, where a is a bit string. Therefore we often identify bit strings and integers. The right shift operator is defined by $i \gg k = \lfloor i/2^k \rfloor$. Clearly, if $i = (\ldots a_2 a_1 a_0)_2$, then $i \gg k = (\ldots a_{k+2} a_{k+1} a_k)_2$. In the same vein, the left-shift operator is defined by $i \ll k = i2^k = \left(\ldots a_2 a_1 a_0 0^k\right)_2$.

We denote by E_i the n-bit string which is zero everywhere except on the i-th coordinate — the value of n is usually clearly given by the context. Note that $(E_i)_2$ represents the integer 2^i. The sequence $(E_0, E_1, \ldots, E_{n-2})$ forms the canonical basis of the vector space ${\mathbb{F}_2}^n$.

We denote by $\oplus$ the exclusive-or operation (XOR) applied indistinctively to integers and to bit strings. This is also the addition in the field with two elements or in vector spaces over $\mathbb{F}_2$. We

[1] https://gitlab.lip6.fr/almasty/libfes-lite

use $-$ to denote the usual subtraction over $\mathbb{Z}$ and $\boxminus$ to denote *saturating subtraction*: if $a \geq b$, then $a \boxminus b = a - b$, otherwise $a \boxminus b = 0$.

Let $\rho(i)$ denote the greatest integer k such that 2^k divides i, with $\rho(0) = +\infty$. This is also known as the "ruler function" or the 2-adic valuation of i. This locates the rightmost "1" bit in the binary representation of the integer i. We will later need the following

Lemma 3.1. *For all integers i, we have: $i \oplus (i+1) = 2^{\rho(i+1)+1} - 1$.*

Proof. Take an integer i and write it in binary as $i = (x01^a)_2$ for some bit string x and some integer $a \geq 0$. Incrementing i clears the least significant run of ones and we get that $i+1 = (x10^a)_2$. This shows in passing that a is simply $\rho(i+1)$. It follows that $i \oplus (i+1) = \left(1^{a+1}\right)_2$, hence the announced result. □

Finally, define the "generalized" function $\rho^* : (\ldots a_2 a_1 a_0)_2 \mapsto \{j \in \mathbb{N} \ : \ a_j = 1\}$. This tracks the location of all "1" bits in the binary representation of its argument. For instance, $42 = (101010)_2$, therefore $\rho^*(42) = \{1, 3, 5\}$. Also, $1337 = (10100111001)_2$, and $\rho^*(1337) = \{0, 3, 4, 5, 8, 10\}$. It follows that $\rho(i) = \min \rho^*(i)$. Also define $\rho_i(x)$ to be the i-th smallest element of $\rho^*(x)$ — this is the position of the i-th rightmost "1" bit in x. This means that $\rho(x) = \rho_1(x)$.

3.2 Boolean Monomials and Boolean Polynomials

The ring of Boolean polynomials in n variables $x = (x_0, \ldots, x_{n-1})$, hereafter denoted by $\mathcal{R}$, is the quotient of the polynomial ring $\mathbb{F}_2[x_0, \ldots, x_{n-1}]$ by the ideal spanned by the so-called "field equations" $\left\langle x_0^2 - x_0, \ldots, x_{n-1}^2 - x_{n-1} \right\rangle$. Therefore, if f is a Boolean polynomial, then the exponent of any variable in all monomials is either 0 or 1. Note that we use the symbol + to denote addition in $\mathcal{R}$.

A Boolean monomial ${x_0}^{e_0} {x_1}^{e_1} \ldots {x_{n-1}}^{e_{n-1}}$ is completely described by the bit string $e_0 \ldots e_{n-1}$ (the "exponent vector"). It is also completely determined by the set $\{0 \leq i < n \ : \ e_i = 1\}$. Therefore, we happily identify monomials with n-bit strings and with subsets of $\{0, \ldots, n-1\}$. Beware that the constant monomial 1 is identified with the all-zero bit string (this can be confusing).

The degree of a monomial is the Hamming weight of the exponent bit string. The degree of a Boolean polynomial f, denoted by $\deg f$, is the largest degree of its monomials.

A Boolean polynomial of degree d in n variables has at most $\binom{n}{\downarrow d} = \sum_{k=0}^{d} \binom{n}{k}$ terms — this convenient notation for the partial sum is borrowed from [Din21a]. Indeed, there are $\binom{n}{k}$ Boolean monomials of degree k: they are the product of k distinct variables.

3.3 Monomial Orders

Monomials can be ordered in a variety of ways — the reader may consult [CLO07] for a gentle introduction to the subject. It turns out that different algorithms favor different orders.

The usual lexicographic order over monomials is obtained by ordering the exponent vectors in lexicographic order. This yields for example:

$$1 < x_7 < x_4 x_5 < x_1 < x_0 < x_0 x_7 < x_0 x_1$$

All monomials that contain x_0 (the first variable) are greater than monomials that do not contain it.

The *colexicographic* order (often abbreviated colex) is obtained by ordering the exponent vectors in lexicographic order while reading them from the least significant bit to the most significant. This is the same thing as taking the usual lexicographic order with variables in reversed order $(\ldots, x_2, x_1, x_0)$. The previous example now becomes:

$$1 < x_0 < x_1 < x_0 x_1 < x_4 x_5 < x_7 < x_0 x_7$$

All monomials that contain x_7 (the last variable) are greater than monomials that do not. More formally, $s < t$ iff there exists $0 \leq i < n$ such that the last $i-1$ exponents of s and t are equal but the i-th exponent of s is less than the i-th exponent of t.

The colex order is called `invlex` in SageMath and `rp` in Singular. Using the colex order greatly simplifies the implementation of some functions without a significant loss of generality, since the variables just have to be reversed. This fact is pointed out in [Rus03].

The Moebius transform usually requires the lexicographic or colexicographic order. In contrast, our implementation of FES uses the *graded colexicographic* order: monomials are ordered by increasing degree, and ties are broken using the colex order. This yields

$$1 < x_0 < x_1 < x_7 < x_0x_1 < x_4x_5 < x_0x_7 < \dots.$$

In this order, all monomials of degree k come before monomials of degree $k+1$. Note that this is not the same as the more common "graded reverse lexicographic order" that is often favored when computing Gröbner bases.

Each monomial has a *rank* in a given order, which is simply its position in the totally ordered sequence. *Ranking* is the operation of computing the rank of a monomial, while *unranking* is the operation of building a monomial given its rank. The ability to do both efficiently is crucial to the performance of BeanPolE.

3.4 Derivatives

By analogy with the corresponding notion from calculus, define the "differential operator" $D_k : \mathcal{R} \mapsto \mathcal{R}$ that differentiates with respect to the k-th variable as

$$D_k : f \mapsto f(x + E_k) - f(x)$$

Note that subtraction and addition coincide in characteristic two. D_k is easily seen to be a linear operator on $\mathcal{R}$. If m denotes a monomial that is not a multiple of x_k, then it is easy to check that $D_k(m) = 0$ and $D_k(mx_k) = m$. These properties make it straightforward to evaluate the derivatives of any polynomial. In fact, this shows that $D_k(f)$ contains all the monomials of f that are divisible by x_k, divided by x_k. It follows that $D_k(f)$ does not depend on x_k and that $\deg D_k(f) \leq \deg f - 1$. More precisely, we can write

$$f(x) = f(x_0, \dots, x_{k-1}, 0, x_{k+1}, \dots, x_{n-1}) + x_k \cdot D_k(f)(x)$$

Higher-order derivatives are simply the derivatives of the derivatives. For instance

$$D_\ell \circ D_k : f \mapsto (f(x + E_k + E_\ell) - f(x + E_\ell)) - (f(x + E_k) - f(x)).$$

Looking at this definition, it is easy to conclude that the differentiation operators commute: $D_k \circ D_\ell = D_\ell \circ D_k$. This implies that differentiating with respect to a subset of the variables (*i.e.* a monomial) makes sense. We thus write $D_{k,\ell}$ to denote the second-order differentiation operator with respect to the two variables x_k and x_ℓ, and D_m for differentiation with respect to the variables contained in an arbitrary monomial m. The special case $m = 1$ corresponds to the derivation with respect to nothing, and thus we set $D_1(f) = f$. If f has degree d, then its d-th order derivatives are constant. The polynomial $D_m(f)$ contains all monomials of f that are multiples of m, divided by m. It follows that evaluating $D_m(f)$ on zero yields the coefficient of the monomial m in f.

The derivative of a polynomial f with respect to m is denoted as $D_m(f)$, and its evaluation on a vector x is logically denoted as $D_m(f)(x)$. To alleviate notations, we will often omit f, because it is usually fixed and obvious from the context. The result of the evaluation of the polynomial $D_m(f)$ on x will thus be denoted as $D_m(x)$.

3.5 Computational Model

The computational complexities of algorithms described in this article are given in the *transdichotomous model* [FW93, FW94]. It is the familiar model taught in most algorithmic classes and described in the introduction of the well-known textbook [CLRS09].

It assumes a machine with a finite number of w-bit registers and an unbounded memory where each memory location holds a w-bit word. Arithmetic or logical operations between registers have unit cost. Reading or writing the memory location whose address is in a register is also an elementary operation with unit cost. On an input of size n, the machine word size w is assumed to be larger than $c \cdot \log n$ for some constant c. In other terms, the machine is large enough to deal with the input.

For practical input sizes, the stated complexities correspond, up to a constant factor, to the number of instructions executed by a RISC-V processor running the compiled C code. The algorithms given in this paper do not require multiplication, so the minimal RV64I ISA is actually sufficient.

4 THE BEANPOLE LIBRARY

BeanPolE is a concise C library totalling 840 lines of code and headers. It is bundled with a comprehensive user manual in the form of a texinfo document and a test suite. The test suite contains demonstration programs that illustrate how the library could be used.

BeanPolE has been developed with the explicit goal of providing the necessary foundation upon which to build state-of-the art algorithms to solve quadratic Boolean systems. It has notably been used in a complete implementation of the Joux-Vitse algorithm [JV17] that currently holds computational records. This makes uses of nearly all the functionalities of the library.

4.1 Algorithmic Improvements

The two main algorithms implemented in the BeanPolE library are slightly better than their previous presentations. First of all, both are *nearly in-place*: they alter the input polynomial, but restore it to its original state once the full truth table has been visited. They require $O(n)$ extra words of storage in addition to the space needed for the input polynomial[2].

The version of the FES algorithm implemented in BeanPolE also improves over [BCC+10] in two other ways: it visits the next entry of the truth table in $O(d)$ operations in the worst case —as opposed to $O(d)$ amortized operations— and its setup phase is simpler.

The space-efficient Moebius transform also improves upon the presentation given in [Din21a] in that it runs in time $O(d2^n)$ on usual computers. The original presentation of the algorithm claims this many bit operations in the abstract computational model of straight-line programs. These programs have no loops and the measure of computational complexity is just the size of the program, *i.e.* the number of statements. The programs are specific to the size of the input, have exponential size (because their running time is exponential), and therefore may contain an exponential amount of precomputed information that depends on the size of the input.

In the case of the Moebius transform, these straight-line programs perform an exponential number of memory accesses at addresses that are "hard-coded" — this is an exponential amount of information embedded in the code.

In BeanPolE, there is a single, fixed, program written in the C programming language that accepts inputs of any size. This implies that the exponentially many memory addresses that are accessed are computed "on the fly", in constant time.

[2]The commonly accepted meaning of "in-place" requires the extra space requirements to be logarithmic in the input size and it is not the case here. However, the extra space requirements are practically negligible compared to the input size.

4.2 Data Structures

BeanPolE uses a simple dense representation: all coefficients of a polynomial f are stored contiguously in a single array (denoted by `A`). It follows that `A` has $\binom{n}{\downarrow d}$ entries if f is a degree-d polynomial in n variables. Once a monomial order has been fixed, the coefficient of the i-th monomial in this order is stored in `A[i]`.

The use of a simple array allows constant-time access to the coefficients of a monomial, once the corresponding rank in the array has been determined. This is a decisive advantage to obtain asymptotically efficient algorithms. However, determining the index positions to access the array concentrates all the technical difficulties.

It is the user's responsibility to prepare the array containing the coefficients of the polynomial before feeding it to BeanPolE. Helper functions are provided to determine its size and to compute the rank of monomials in the relevant monomial orders. The companion demonstration programs illustrate how to parse text representations of polynomials and populate this dense representation.

Functions of the BeanPolE library are *data-oblivious*, *i.e.* they perform the same sequence of operations regardless of the input values. Moreover, they only access the coefficients through two simple "update" operations: $\mathtt{A}[i] \leftarrow \mathtt{A}[i] \oplus \mathtt{B}[j]$ or $\mathtt{A}[i] \leftarrow \mathtt{A}[i] \oplus (\mathtt{B}[j] \,\&\, \mathtt{C}[k])$, where `A`, `B` and `C` are arrays of coefficients. This makes the library quite generic: it only requires the two update functions described above to be provided, and is oblivious to the actual machine representation of the coefficients. This genericity is implemented in the C programming language using function pointers. It has several advantages. The same unmodified library code is capable of:

- Working with a simple representation that uses a value of type `bool` to store a coefficient.
- Working with a packed representation that uses a single bit per coefficient.
- Dealing with m polynomials simultaneously by seeing them as a single polynomial whose coefficients are m-bit strings.

The companion demonstration programs illustrate these capabilities.

The main drawback of using a dense representation is that the number of variables and the degree of the polynomial usually have to be known beforehand, at least to allocate the array. This is slightly inconvenient, for instance when reading a file containing the description of a polynomial of unknown degree.

4.3 Interface

BeanPolE is written in plain C for maximum portability and ease of reuse. Its main features are efficient *iterators* over the truth table of a Boolean polynomial, with slightly different characteristics.

- The FES algorithm takes as input a degree-d polynomial in n variables, in graded colex order. It iterates over the entries of the truth table in the order specified by the binary reflected Gray code. It moves to the next entry in $O(d)$ operations. It requires a setup phase whose asymptotic complexity is not easy to analyze precisely, besides the obvious claim that it is quadratic in the size of the polynomial. In some specific scenarios (e.g., degree-$n/3$ polynomials in n variables), this setup phase asymptotically dominates the cost of building the full truth table — these scenarios are mostly of theoretical interest; our practical experience with the algorithm is that the setup phase is completely negligible.
- The in-place Moebius transform takes as input the coefficients of a polynomial in colex order. It iterates over the truth table in lexicographic order, by producing chunks of size 2^d at a time. This has to be repeated 2^{n-d} times to obtain the full truth table. Obtaining the next chunk has amortized complexity $O(d2^d)$ but its worst-case execution time may be almost linear in the size of the input polynomial.

Operating on polynomials stored in dense representation, *i.e.* as a linear array of coefficients, makes crucial use of the ranking operation for the (graded) colex order. It also requires the ability to efficiently iterate over monomials of degree k (resp. at most k) in n variables. Functions performing these tasks form the infrastructure of the library.

In order to iterate over a sequence of objects (entries of the truth table, monomials, etc.), the library exposes *iterators* made of three functions:

- **void** xxx_prepare (..., **struct** beanpole_iterator * it): initializes the iteration.
- **void** xxx_advance(**struct** beanpole_iterator * it): move to the next element.
- **bool** xxx_finished (**const struct** beanpole_iterator * it): indicates if there is a next element.

The iterators maintain their state in an *ad hoc* variable, the **struct** beanpole_iterator object. The user accesses the current state of the iteration by reading some fields of this iterator variable.

5 CORE INFRASTRUCTURE

This section describes the low-level functions that underlie all operations on multivariate polynomials in dense representation. This includes functions to iterate over the set of all monomials, as well as function to rank and unrank monomials. The definitions of these functions are shown in Figure 1.

Most of these function operate on the set of all monomials of degree at most d in n variables in a specific order. They usually take arguments named n, d and *order* for this purpose. A monomial is described by its degree and an array containing the indices of the variables it contains, in increasing order.

5.1 Enumerating Monomials

Enumerating all degree-k monomials in colex order is strictly equivalent to the well-known problem of enumerating all k-subsets of $\{0, 1, 2, \ldots, n-1\}$ in colex order. For this, we use the venerable algorithm first described in [Mif63], along with the optimization described in [Dvo90]. All of this is summarized in algorithm T from [Knu14, §7.2.1.3]. It moves to the next monomial in amortized constant time. Repeating this for $k = 0, \ldots, d$ allows the enumeration of all monomials of degree at most d in the graded colex order.

We could not trace a reference to an algorithm that enumerates all subsets of $\{0, 1, 2, \ldots, n-1\}$ of size *at most d* in colex order, therefore we now describe the one we use, even though we suspect that it belongs to the folklore.

The iteration maintains an integer k and an array S of size $d+1$. At all times, $S[d-k:d]$ contains the elements of the current subset in ascending order. We set a "sentinel" value $S[d] \leftarrow 1$. The enumeration starts with $k = 0$, and thus with the empty set. Advancing to the next subset proceeds as shown in Figure 2.

Lemma 5.1. *The procedure shown in Figure 2 transforms S into the next monomial of degree at most d in n variables in colex order.*

Proof. It is convenient to consider that the current subset $S \subseteq \{0, \ldots, n-1\}$ is described by a bit string $s = s_0 s_1 \ldots s_{n-1}$, where $i \in S \Leftrightarrow s_i = 1$. In particular, we write $s = 0^a 1^b 0 x$ for some integers $a, b \geq 0$ and some suffix $x \in \{0,1\}^*$. This means that $S = \{a, a+1, \ldots, a+b-1, \ldots\}$.

First, if $0 \notin S$ and $|S| < d$, then $\{0\} \cup S$ is the next subset in lexicographic order and satisfies the size constraint, therefore it is acceptable.

Otherwise, we have $|S| = d$ or $0 \in S$. In both cases, $S \neq \emptyset$ and therefore $b > 0$ in the bit string representation.

If $|S| = d$, then the next $2^a - 1$ subsets in lexicographic order have size strictly greater than d because they correspond to the bit strings $y 1^b 0 x$ where $1^b x$ has weight exactly d, $y \in \{0,1\}^a$ and

```
typedef enum {BEANPOLE_COLEX, BEANPOLE_GRCOLEX} beanpole_order;

struct beanpole_iterator {
        int k;                          /* degree of the current monomial */
        int m[];                        /* variables of the current monomial */
        size_t rank;                    /* rank of the current monomial */
         ...
};

/* Return the total number of monomials in n variables of degree at most d */
size_t beanpole_size(int n, int d);

/* Iterator over all monomials of degree <= d in n variables in the given order */
void beanpole_monomials_prepare(int n, int d, beanpole_order order, struct beanpole_iterator *it);
void beanpole_monomials_advance(struct beanpole_iterator *it);
bool beanpole_monomials_finished(const struct beanpole_iterator *it);

/* c <--- a*b (monomials product). Return the degree of c */
int beanpole_monomials_product(int adeg, const int ax[], int bdeg, const int bx[], int cx[]);

/* Return the rank of the given monomial */
size_t beanpole_rank(int n, int d, beanpole_order order, int k, const int m[]);

/* Write the monomial of the given rank in m and return its degree */
int beanpole_unrank(int n, int d, beanpole_order order, size_t rank, int m[]);
```

Fig. 1. Declarations of the "infrastructure" functions in BeanPolE.

```
1: if k ≠ d and S[d − k] ≠ 0 then
2:     Increment k                                   ▷ Add zero
3:     set S[d − k] ← 0
4: else
5:     while k > 1 and S[d − k] + 1 = S[d − k + 1] do
6:         Decrement k                               ▷ Erase smallest "run"
7:     Increment S[d − k]                            ▷ Bump smallest digit
```

Fig. 2. Advancing to the next subset of $\{0, \ldots, n-1\}$ of size at most d in lexicographic order.

$y \neq 00\ldots0$. The following subset in lexicographic order is 0^a0^b1x. It has size less than or equal to d, hence it is acceptable.

If $0 \in S$, then $a = 0$ and the next subset in lexicographic order is again 0^a0^b1x, which is acceptable.

It follows that either it is possible to add zero to the current subset, or we need to transform $\{a, a+1, \ldots, a+b-1, \ldots\}$ into $\{a+b, \ldots\}$. This is precisely what the algorithm does. □

Lemma 5.2. *The procedure shown in Figure 2 runs in constant amortized time.*

Proof. If the **while** loop does t iterations (*i.e.* if line 6 is executed t times), then the total number of statement executed is $3 + 2t$.

By the accounting method, assume that each invocation of the procedure requires a deposit of 5 credits and that executing a single statement costs one credit. When the condition of the **if**

statement is true, then one item is added to the current subset, the procedure terminates quickly and two extra credits are left in the account.

Otherwise, t iterations of the **while** loop take place, t items are removed from the current subset and the balance of the account decreases by $2t - 2$.

If follows that the balance of the account is always greater than two times the size of the current subset, and as such it cannot become negative. A sequence of N invocations of the procedure therefore executes less than $5N$ statements in total, hence the constant amortized cost. □

5.2 Ranking

It is not very difficult to compute the rank of a given degree-k monomial among all monomials of degree at most d in n variables, in the orders that matter to us. All of this is well-known, see for instance the textbook [Rus03]. The rank of m is the number of strictly smaller monomials. A ranking function assigns a distinct positive integer to each monomial.

Let us begin with the colex order restricted to degree-k monomials. Let μ_k denote the corresponding ranking function, which assigns an integer less than $\binom{n}{k}$ to each monomial of degree k in n variables. If $m = \{i_0, \dots, i_{k-1}\}$, then m is greater than all degree-k monomials only containing variables strictly smaller than i_{k-1}. There are $\binom{i_{k-1}}{k}$ such monomials (recall that variable numbering starts at zero). Then, the rank of m among all degree-k monomials whose greatest variable is i_{k-1} is precisely $\mu_{k-1}(i_0, \dots, i_{k-2})$. This leads to the recursive definition:

$$\mu_0(\emptyset) \coloneqq 0$$
$$\mu_k(\{i_0, \dots, i_{k-1}\}) \coloneqq \binom{i_{k-1}}{k} + \mu_{k-1}(i_0, \dots, i_{k-2})$$

Unfolding the recursive definitions leads to the sum:

$$\mu_k(\{i_0, \dots, i_{k-1}\}) = \sum_{j=0}^{k-1} \binom{i_j}{j+1} \tag{1}$$

Let ν denote the ranking function for the graded colexicographic order on n variables. Because monomial are ordered by degree, then $\nu(m) < \nu(m')$ if $\deg m < \deg m'$. If a monomial m has degree k, there are $\binom{n}{\downarrow k-1}$ monomials of strictly smaller degree. This shows that:

$$\nu(m) \coloneqq \binom{n}{\downarrow \deg m - 1} + \mu_{\deg m}(m).$$

Evaluating μ_k and ν requires k operations, assuming that the binomial coefficients and their partial sums are precomputed.

Finally, let $\mu_{\downarrow d}$ denote the the ranking function for all monomials of degree at most d in n variables in the colex order. It assigns integer less than $\binom{n}{\downarrow d}$ to each of them. A similar reasoning that what has already been done shows that:

$$\mu_{\downarrow 0}(\emptyset) \coloneqq 0$$
$$\mu_{\downarrow d}(\{i_0, \dots, i_{k-1}\}) \coloneqq \binom{i_{k-1}}{\downarrow d} + \mu_{\downarrow d-1}(i_0, \dots, i_{k-2})$$

Unfolding this recursive definitions leads to:

$$\mu_{\downarrow d}(\{i_0, \dots, i_{k-1}\}) = \sum_{j=0}^{k-1} \binom{i_j}{\downarrow d - k + j + 1} \tag{2}$$

6 SUPPORT FUNCTIONS

The "core infrastructure" functions described in the previous section make it easy to implement many higher-level operations on dense Boolean polynomials.

6.1 Input / Output

One of the first relevant use is the ability to load and store polynomial to and from text files. The companion demonstration programs load a polynomial from a file as follows:

(1) Find the degree d and the number of variables n; choose a monomial order.
(2) Set $N \leftarrow$ `beanpole_size(n, d)`; allocate an array `A` of size N and initialize it with zeros.
(3) Read the next monomial from the file; sort its variables; compute its rank i using the `beanpole_rank()` function and set `A`$[i] \leftarrow 1$.
(4) Repeat step 3 until all the monomials have been read from the file.

Writing a polynomial represented by an array `A` to a text file can be done like this:

(1) Initialize an iterator over all monomials in n variables of degree at most d in the correct order.
(2) While there is a next monomial, find the rank i of the current monomial in the iterator variable; if `A`$[i] = 1$, then write the description of the monomial accessible in the iterator to the text file.
(3) Advance the iterator to the next monomial and return to step 2.

6.2 Change of Order

If necessary, copying a degree-d polynomial and changing its monomial order is simple:

(1) Allocate an array `B` of the right size; initialize an iterator over all degree-d monomials in n variables in the input order.
(2) While there is a next monomial, find the rank i of the current monomial m in the iterator variable; compute the rank j of m among all monomials in n variables of degree at most d in the desired output order using `beanpole_rank()`; set `B`$[j] \leftarrow$ `A`$[i]$.
(3) Advance the iterator to the next monomial and return to step 2.

This requires at most d operation per coefficient of the input polynomial.

6.3 Multiplication

The `beanpole_product_pxp` function naively computes the product ($C \leftarrow C + A \times B$) of two polynomials (in any order, with the output in any desired order). It works as follows:

(1) Initialize an "outer" iterator over all monomials of A in the correct order. Set $d' \leftarrow \deg A + \deg B$. Fill the output array C with zeros.
(2) If there is no next monomial in A, stop the algorithm. Otherwise, look the rank i of the current monomial m in A; initialize an "inner" iterator over all monomials of B in the correct order.
(3) If there is no next monomial in B, go to step 5. Otherwise, look the rank j of the current monomial m' in B; compute the monomial product $m \times m'$ using `beanpole_monomials_product()`; compute the rank k of $m \times m'$ among all monomials in n variables of degree at most d' using the `beanpole_rank()` function; set $C[k] \leftarrow C[k] \oplus (A[i] \,\&\, B[j])$.
(4) Advance the inner iterator to the next monomial of B and return to step 3.
(5) Advance the outer iterator to the next monomial of A and return to step 2.

If S_A and S_B denote the respective sizes of the two input polynomials, then this function requires at most $(\deg A + \deg B)S_A S_B$ operation.

6.4 Last Variable Substitution

The `beanpole_subslastvar_colex()` function takes as argument the coefficients of a polynomial f, an array of Booleans `b` and overwrite the input polynomial with the coefficients of $f(x_0, x_1, \ldots, x_{n-2}, x_{n-1}+b_0x_0+\cdots+b_{n-2}x_{n-2})$. In other terms, it adds an arbitrary linear combination of the $n-1$ first variables to the last one. Note that this operation is involutive. Subsequently setting the last variable to zero makes it possible to perform the substitution $x_{n-1} \leftarrow b_0x_0 + \cdots + b_{n-2}x_{n-2}$. This function only works with polynomials in colex order. Note that adding a constant term to the last variable is also possible and it is more efficient; this is discussed in section 8.1.

Let m denote a monomial of degree strictly less than d that does not contain x_{n-1}. The substitution turns mx_{n-1} into $mx_{n-1} + b_0mx_0 + \cdots + b_{n-2}mx_{n-2}$. Let c denote the coefficient of mx_{n-1}; the substitution can be realized in-place by adding $c \times b_i$ to the coefficient of mx_i, for all m and i.

The procedure operates as follows:

(1) Initialize an iterator over all monomials of degree at most $d-1$ in $n-1$ variables in colex order; set $u \leftarrow \binom{n-1}{\downarrow d}$ [u is then the rank of mx_{n-1} where m is the monomial in the iterator].
(2) If there is no next monomial, stop the algorithm. Otherwise, set $i \leftarrow 0$.
(3) If $i = n-1$, then advance the iterator to the next monomial, increment u and return to step 2.
(4) If $\mathtt{b}[i] = 0$, then advance to step 5. Otherwise, compute the monomial product mx_i using `beanpole_monomials_product()`; compute the rank v of mx_i among monomials in n variables of degree at most d using `beanpole_rank()`; set $A[v] \leftarrow A[v] \oplus A[u]$.
(5) Increment i and return to step 3.

Up to a constant multiplicative factor, the total number of operation is $nd\binom{n-1}{\downarrow d-1}$. This is essentially d times the size of the polynomial.

6.5 Evaluation

The `beanpole_eval()` functions evaluates a degree-d polynomial on an arbitrary input a given as an array of Booleans. More precisely, given y, it computes $y \oplus f(a_0, \ldots, a_{n-1})$. A monomial evaluates to 1 on input a when it only contains variables x_i such that $a_i = 1$. Therefore we enumerate the subsets of size at most d of $\{0 \le i < n : a_i = 1\}$ and sum the corresponding coefficients of the polynomial. The evaluation function thus works as follows:

(1) Initialize two arrays `xx` and `mm` of size n and set $h \leftarrow 0$.
(2) For $0 \le i < n$, do: if $a_i = 1$, then set $\mathtt{xx}[h] \leftarrow i$ and increment h. [after this h is the Hamming weight of a].
(3) Set $D \leftarrow \min(h, d)$; initialize an iterator on all monomials of degree at most D in h variables.
(4) If there is no next monomial, stop the algorithm. Otherwise, look the degree k of the current monomial m in the iterator; for $0 \le i < k$, do: $\mathtt{mm}[i] \leftarrow \mathtt{xx}[m[i]]$. [`mm` is the next monomial that evaluates to 1 on input a].
(5) Compute the rank j of `mm` among all monomials in n variables of degree at most d in the input order using `beanpole_rank()`; set $y \leftarrow y \oplus A[j]$.
(6) Advance the iterator to the next monomial and return to step 4.

Up to a constant multiplicative factor, the running time of this function is upper-bounded by $d\binom{h}{\downarrow d}$, where h denotes the Hamming weight of the evaluation point x. On random inputs, where on expectation $h = n/2$, this is approximately $d2^{-d}$ times the size of the input polynomial (when n is large compared to d).

6.6 Computation of Derivatives

The derivative of a degree-d polynomial f with respect to a degree-d' monomial m (with $d' \leq d$) contains all monomials of f that are multiples of m, divided by m. We compute it by enumerating all monomials of degree at most d of the form $m \times m'$ where m' does not contain any variable of m. The procedure works as follows:

(1) Initialize arrays `mprime` and `xbar` of size n and set $u \leftarrow 0, v \leftarrow 0$.
(2) Initialize an array `B` of size $\binom{n-d'}{\downarrow d-d'}$ and fill it with zeros.
(3) For $0 \leq i < n$, do: if $m[u] = i$, then increment u; otherwise set `xbar`$[v] \leftarrow i$ and increment v. (at this point, `xbar` contains variables not in m)
(4) Initialize an iterator on all monomials of degree at most $d - d'$ in $n - d'$ variables in the desired output order.
(5) If there is no next monomial, stop the algorithm. Otherwise, look the degree k of the current monomial m in the iterator; for $0 \leq i < k$, do: `mprime`$[i] \leftarrow$ `xbar`$[m[i]]$. (m' is the next monomial on variables not in m).
(6) Compute the monomial product $m \times m'$ using `beanpole_monomials_product()`; Let u denote the rank of m accessible in the iterator; compute the rank v of $m \times m'$ among all monomials in n variables of degree at most d in the input order using `beanpole_rank()`; set $B[u] \leftarrow A[v]$.
(7) Advance the iterator to the next monomial and return to step 5.

6.7 Computation of Macaulay Matrices

We conclude with one last direct application of the "core infrastructure" functions: the generation of a Macaulay matrix in sparse representation. This is for instance required when implementing the "Crossbred" algorithm to solve Boolean polynomial systems [JV17]. The set of all polynomials of degree at most D form a vector space of dimension $\binom{n}{\downarrow D}$. Let $f_0, \ldots, f_{\ell-1}$ denote degree-d polynomials and consider the subspace U spanned by all the $m \times f_i$, where $0 \leq i < \ell$ and m ranges across all degree-$(D-d)$ monomials. We consider the problem of assembling the corresponding Macaulay matrix M, namely the matrix of dimension $\ell\binom{n}{D-d} \times \binom{n}{\downarrow D}$ whose rows describe the generators of U. Computing a Gröbner basis of the polynomial ideal spanned by the f_i's can be done by putting such matrices into reduced row echelon form, if D is sufficiently large. These matrices are very sparse, and only the non-zero coefficients need to be stored. The procedure shown in Figure 3 generates the list of non-zero entries of the matrix row-by-row, which is convenient when dealing with sparse matrices in Compressed Sparse Row (CSR) representation.

The next two sections describe the *pièce de résistance* of the BeanPolE library, namely the two algorithms that iterate over the truth table of a Boolean polynomial.

7 THE FAST EXHAUSTIVE SEARCH (FES) ALGORITHM

This section presents a version of the FES algorithm that is slightly enhanced compared to its original presentation in [BCC+10].

7.1 Main Ideas

The FES algorithm walks through the truth table by flipping a single variable at each step, using the binary reflected Gray code. Flipping variables $0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, \ldots$ will eventually visit all n-bit strings exactly once. More precisely, the enumeration process evaluates f on a sequence of inputs $g^{(j)}$ for $j = 0, 1, 2, \ldots$. The initial value is $g^{(0)} = 000\ldots000$ (the all-zero n-bit string). Jumping to the next input is done by setting $g^{(j+1)} = g^{(j)} + E_k$ (flipping the k-th

```
1:  procedure MACAULAY(n, d, D, A)
2:      Set k ← 0                                              ▷ current row of the matrix
3:      for all monomials m of degree at most D − d do         ▷ use iterator
4:          for 0 ≤ i < ℓ do
5:              # emit m × f_i
6:              for all monomial m′ of degree less than d do   ▷ use iterator
7:                  u ← rank of m′                             ▷ accessible in the iterator
8:                  if A[i, u] = 1 then                        ▷ coefficients of f_i are available in A[i, ·]
9:                      Compute the monomial m × m′            ▷ monomial product
10:                     v ← rank of m × m′ (n variables, degree ≤ D)   ▷ ranking function
11:                     Set M[k, v] ← 1                        ▷ append new non-zero entry to M
12:             Increment k                                    ▷ advance to the next row
```

Fig. 3. Creating a Macaulay matrix in sparse representation

variable), where $k = \rho(j+1)$. It is well-known that using such a Gray code, the j-th visited bit string is given by

$$g^{(j)} = j \oplus (j \gg 1). \tag{3}$$

We also introduce the notation $[\![j]\!]$ as a shorthand for $g^{(j)} = j \oplus (j \gg 1)$, namely the j-th entry of the binary reflected Gray code. We will use several times the following simple

LEMMA 7.1. $$\left[\!\left[(2^{k+1} - 1) \ll \ell \right]\!\right] = \begin{cases} E_k & \text{if } \ell = 0 \\ E_{k+\ell} \oplus E_{\ell-1} & \text{if } \ell > 0. \end{cases}$$

PROOF. If $\ell = 0$, then

$$\left[\!\left[2^{k+1} - 1 \right]\!\right] = 1^{k+1} \oplus 1^k = 10^k = E_k.$$

Otherwise, if $\ell > 0$, then

$$\left[\!\left[(2^{k+1} - 1) \ll \ell \right]\!\right] = 1^{k+1}0^\ell \oplus 1^{k+1}0^{\ell-1} = 10^k 10^{\ell-1} = E_{k+\ell} \oplus E_{\ell-1}$$

□

The main idea behind the algorithm is the following. Suppose that $y^{(j)} = f([\![j]\!])$. Moving on to the next input $[\![j+1]\!]$ requires flipping the k-th variable with $k = \rho(j+1)$. By definition of the derivative of f with respect to the k-th variable (cf. section 3.4), this yields $y^{(j+1)} = y^{(j)} \oplus D_k([\![j]\!])$. In order to move on to the next entry of the truth table, it is sufficient to evaluate D_k on $[\![j]\!]$ and to XOR the result onto $y^{(j)}$. This is easier than the original problem because D_k has degree $d-1$, whereas f has degree d.

The idea of using a Gray code to evaluate a Boolean polynomial on all inputs appears to belong to the folklore — we could not trace a precise reference to the first time it has been used. The FES algorithm goes further by using it recursively on the derivatives. The end result is that a single XOR operation allows to decrease by one the degree of the derivative that remains to be evaluated, until it drops to zero. Because f has degree d, a total of d XOR operations is required to "update" y each time a variable is flipped, plus some bookkeeping that also requires $O(d)$ operations.

In order to make this possible, the key observation is that in the procedure described above, the derivative D_k with respect to the k-th variable only needs to be evaluated on $[\![j]\!]$ when $\rho(j+1) = k$. This happens when $j + 1 = 2^k + i2^{k+1}$ for $i = 0, 1, 2, \ldots$. In other terms, it happens very regularly, every 2^{k+1} iterations.

The main idea behind the FES algorithm consists in storing the values of all the derivatives on their last evaluation point, and "update" these when needed using higher-order derivatives. These updated derivatives must be stored somewhere, and the only available space is the array holding the coefficients of f. Let us denote by $\mathcal{A}$ an array that can be directly indexed by monomials. Then a natural dense representation of f consists in storing in $\mathcal{A}[m]$ the coefficient of m in f. It follows from the discussion of section 3.4 that $\mathcal{A}[m] = D_m(0)$. We make use of this fact by considering that $\mathcal{A}$ initially contains all the derivatives of f evaluated on zero. In the sequel, if m is any monomial, then we mostly deal with "the derivative of f with respect to m" instead of "the coefficients of m in f". When all the derivatives are evaluated on zero, the two concepts coincide. However, as the algorithm progresses, the evaluation points of the derivatives change.

In the procedure outlined above, the first evaluation of D_0 happens when $j = 0$ and D_0 must be evaluated on $g^{(0)} = [\![0]\!] = 0$. There is nothing to do, as $D_0(0)$ is already available in $\mathcal{A}[x_0]$. The first evaluation of D_k (with $k > 0$) happens when $j + 1 = 2^k$ and lemma 7.1 tells us that $[\![j]\!] = E_{k-1}$. From the definition of D_{k-1}, we see that $D_k\,(E_{k-1})$ can be obtained as

$$D_k\,(E_{k-1}) = D_k(0) \oplus D_{k-1,k}(0). \tag{4}$$

Note that $D_k(0)$ and $D_{k-1,k}(0)$ are the coefficients of the monomials x_k and $x_{k-1}x_k$ in f, respectively.

Let us now consider the subsequent evaluations of the first-order derivatives. Suppose that we have in store the result of the evaluation of D_k on $u := [\![2^k + i2^{k+1}]\!]$ and we wish to "update" this to the result of the next evaluation on $v := [\![2^k + (i+1)2^{k+1}]\!]$. What is to be done? To obtain the answer, consider the XOR-difference between the two successive evaluation points. It is an easy consequence of (3) that $[\![s]\!] \oplus [\![t]\!] = [\![s \oplus t]\!]$. In addition, the integer additions in u and v are in fact XOR operations because the two summands only have bits in disjoint locations. This implies that $u \oplus v = [\![(i \oplus (i+1)) \ll (k+1)]\!]$. Lemma 3.1 tells us that $i \oplus (i+1) = 2^{\rho(i+1)+1} - 1$. Finally, lemma 7.1 yields

$$\underbrace{\left[\!\left[2^k + (i+1)2^{k+1}\right]\!\right]}_{v} = \underbrace{\left[\!\left[2^k + i2^{k+1}\right]\!\right]}_{u} + E_{\rho(i+1)+k+1} + E_k \tag{5}$$

In other terms, each time D_k has to be evaluated, the new evaluation point only differ from the previous one by two bits; one of these bits is the k-th, and D_k does not depend on the k-th variable. The other bit has index $\ell := k + 1 + \rho(i+1) = \rho_2(j+1)$. Recall from section 3.1 that $\rho_2(j+1)$ denotes the position of the second rightmost bit in $j+1$. It follows from (5) that:

$$D_k(v) = D_k(u) + D_{k,\ell}(u) \tag{6}$$

Therefore, updating the derivative with respect to the k-th variable from its evaluation on u to its evaluation on v requires the evaluation of the second-order derivative on u.

In the specific case of quadratic polynomials, where second-order derivatives are constant functions, we can stop there. As an introduction to the full-blown FES algorithm, the pseudo-code in Figure 4 shows the FES algorithm specialized for quadratic polynomials. This procedure evaluates a quadratic polynomial f on the 2^n possible inputs using a constant number of operations per iteration. For the sake of hiding nasty technical details, f is represented by a dictionary $\mathcal{A}$ that maps all Boolean monomials to their coefficients in f. Turning this pseudo-code into an actual program requires some additional work, notably to locate the rightmost bit and second-rightmost bit in a counter efficiently, and to implement the $\mathcal{A}$ data structure efficiently.

The libfes-lite library essentially implements the algorithm of Figure 4, along with several optimizations:

- The main loop is deeply unrolled (256 or 512 times). This practically removes the need to evaluate $\rho_1(j+1)$ and $\rho_2(j+1)$, as their value is known at compile-time in most iterations.

```
1:  procedure QuadraticFES(𝒜, n)
2:      # prepare initial values of first-order derivatives
3:      for 1 ≤ k < n do
4:          𝒜[x_k] ← 𝒜[x_k] + 𝒜[x_{k-1}x_k]          ▷ evaluate D_k on E_{k-1} using (4)
5:      # main loop
6:      g ← (0, 0, ..., 0)                            ▷ size n
7:      for 0 ≤ j < 2^n do
8:          emit (g, 𝒜[1])                            ▷ 𝒜[1] = f(g)
9:          k ← ρ(j + 1)                              ▷ locate rightmost set bit
10:         ℓ ← ρ_2(j + 1)                            ▷ locate second rightmost set bit
11:         g_k ← g_k + 1                             ▷ bump g to the next n-bit string
12:         if ℓ < +∞ then
13:             𝒜[x_k] ← 𝒜[x_k] + 𝒜[x_k x_ℓ]          ▷ update D_k with D_{k,ℓ} using (6)
14:         𝒜[1] ← 𝒜[1] + 𝒜[x_k]                      ▷ update f(g) using D_k
```

Fig. 4. A simplified FES algorithm restricted to quadratic polynomials.

- The input polynomial f is extended with 2 "fictitious" variables x_n and x_{n+1} that it does not depend upon. The main loop starts with $j = (100^n)_2$ and runs while $j < (110^n)_2$ — this makes 2^n iterations. The point is that $\rho_2(j+1)$ is then always defined. Setting $D_n = 0$ and $D_{n+1} = 0$ removes the need for the conditional statement of line 12.

The rest of this section presents the full FES algorithm and demonstrates its correctness.

7.2 Tracking Bits in a Counter

We describe a (worst-case) constant-time algorithm to evaluate ρ^* on consecutive integers, starting from zero. In other terms, we track the positions of all set bits in a counter j as it is repeatedly incremented. This can be seen as a way to increment an n-bit counter in worst-case constant time, as opposed to the classic solution of propagating carries that requires amortized constant time. This is required to implement the FES algorithm efficiently, as it provides the evaluation of $\rho(j+1), \rho_2(j+1), \ldots$

We represent the counter using a stack of height at most n. It contains, from bottom to top, the locations of all non-zero bits of the counter, from the most significant (at the bottom of the stack) to least significant (at the top). If the counter is zero, the stack is empty. If the counter is, say, $(1010010)_2$, then the stack is `[6, 4, 1]`. If the counter is incremented to $(1010011)_2$, the stack becomes `[6, 4, 1, 0]`. Note that the actual binary value of the counter j is not stored, because it is represented implicitly by the stack and it would not be possible to update it in constant time anyway.

In any case, $\rho^*(j)$ can be directly read off the stack. If h denotes its height, we find that $\rho_k(j)$ is $+\infty$ if $k > h$ and that it is the $(k-1)$-th item from the top of the stack otherwise.

Here is how to update the stack when the counter is incremented. The counter can always be written $j = \left(a01^k\right)_2$ for some $k \geq 0$ and some bit string a. The next value is $j+1 = \left(a10^k\right)_2$. Note that $k = \rho(j+1)$ is in fact the position of the rightmost non-zero bit of $j+1$. To maintain the stack while j is incremented, we need to 1) pop the top k entries and 2) push k.

Maintaining the stack while the counter is incremented thus boils down to evaluating ρ on consecutive integers, starting from zero. Some CPUs have a hardware instruction to evaluate ρ. For instance, the x86-compatible CPUs have the `bsf` instruction that does just this. The Gnu C Compiler

exposes the `__builtin_ffs()` pseudo-function that invokes the corresponding instruction if the target architecture has it. On the other hand, trying to evaluate ρ efficiently without relying on *ad hoc* hardware instructions leads to a wealth of algorithmic tricks.

A particularly elegant solution has been proposed in [Ehr73, BER76] and summarized in [Knu14, §7.2.1.1]. Using an array of $(n+1)$ "focus pointers" (hereafter denoted by p), it enables the evaluation of ρ on the next value of the counter in a constant number of operations without special hardware support.

Using the technique based on "focus pointers", we now provide a procedure that evaluates ρ^* on all consecutive integers. It first needs to be initialized as follows:

1: Allocate an array p of size $n+1$ ▷ p holds the "focus pointers"
2: Allocate an array `stack` of size n
3: $h \leftarrow 0$ ▷ Height of the stack
4: **for** $0 \leq i < n+1$ **do**
5: $\quad p[i] \leftarrow i$ ▷ Initial values of the focus pointers

The constant-time procedure based on "focus pointers" that computes $\rho(j+1)$ is as follows. Its correctness is proved in the original articles that describe it.

7: $k \leftarrow p[0]$ ▷ $k = \rho(j+1)$
8: $p[0] \leftarrow 0$ ▷ update focus pointers
9: $p[k] = p[k+1]$
10: $p[k+1] = k+1$

Once k is known, updating the stack is straightforward, and the correctness of the procedure follows from the above discussion.

11: $h = h - k$ ▷ Pop top k elements from the stack
12: $\texttt{stack}[h] \leftarrow k$ ▷ Push k onto the stack
13: $h \leftarrow h+1$

7.3 Enumeration in Any Degree

The quadratic FES algorithm shown in Figure 4 works by updating first-order derivatives using second-order derivatives, and updates the current value of the polynomial (*i.e.* the zero-th order derivative) using the first-order derivatives. The second-order derivative are constant and thus do not require updating. The full FES algorithm works for any degree; it uses the constant d-th degree derivatives to update the $(d-1)$-th order derivatives, then use these to update the $(d-2)$-th order derivatives, etc.

Figure 5 shows full FES the algorithm implemented in BeanPolE. All the low-level details have been dealt with and this pseudo-code is easy to translate to the C language. Our C implementation is 128 line long, in a slightly verbose style. By inspection, it is straightforward that each iteration of the main loop requires $O(d)$ operations.

In the j-th iteration, an *update sequence* is determined. It is fully described by a monomial $m^{(j)} = x_{i_1} x_{i_2} \dots x_{i_t}$ of degree at most d ($1 \leq t \leq d$). The indices $i_1, \dots, i_t$ are given by $\rho^*(j+1)$. In other terms, they correspond to the locations of the rightmost bits of $j+1$. They are determined by lines 19–26 of the pseudo-code, in constant time, using the technique described in section 7.2. Note that the quadratic version presented in Figure 4 does a special case of this by computing $k = i_1 = \rho(i+1)$ and $\ell = i_2 = \rho_2(i+1)$.

```
1:  procedure FES(A, n, d)
2:      g ← (0, 0, ..., 0)                          ▷ current evaluation point — size n
3:      r ← (0, 0, ..., 0)                          ▷ r[0] = ν(1) — size d + 1
4:      # Initialize the stack and the focus pointers (cf. section 7.2)
5:      Allocate an array p of size n + 1           ▷ p holds the "focus pointers"
6:      Allocate an array stack of size n
7:      h ← 0                                       ▷ height of the stack
8:      for 0 ≤ i < n + 1 do
9:          p[i] ← i                                ▷ initial values of the focus pointers
10:     # Prepare initial values of derivatives
11:     for all monomials m of degree ≤ d − 1 do    ▷ using iterator
12:         i ← Rank of m                           ▷ accessible in the iterator
13:         if m ≠ 1 then
14:             A[i] ← D_m(m ⊕ (m ≫ 1))             ▷ using evaluation function
15:     # Main loop
16:     loop
17:         emit (g, A[0])                          ▷ A[0] = f(g)
18:         # Update the stack and the focus pointers (cf. section 7.2)
19:         k ← p[0]                                ▷ k = ρ(j + 1)
20:         p[0] ← 0                                ▷ update focus pointers
21:         p[k] = p[k + 1]
22:         p[k + 1] = k + 1
23:         h = h − k                               ▷ pop top k elements from the stack
24:         stack[h] ← k                            ▷ push k onto the stack
25:         h ← h + 1
26:         t ← min(h, d)                           ▷ write i_t, ..., i_2, i_1 = stack[h − t : h]
27:         if k = n then
28:             stop the algorithm
29:         # Compute the ranks of the relevant derivatives (cf. section 5.2)
30:         a ← 0                                   ▷ accumulator
31:         for ℓ = 1, 2, ..., t do
32:             i_ℓ ← stack[h − ℓ]                  ▷ write m_ℓ = x_{i_1} x_{i_2} ... x_{i_ℓ}
33:             a ← a + C(i_ℓ, ℓ)                   ▷ a = μ_k(m_ℓ)
34:             r[k] ← a + C(n, ↓ℓ−1)               ▷ r[i] = ν(m_ℓ)
35:         # Advance to the next entry of the truth table
36:         g_k ← g_k ⊕ 1                           ▷ Update g
37:         for ℓ = t − 1, t − 2, ..., 0 do
38:             A[r[ℓ]] ← A[r[ℓ]] ⊕ A[r[ℓ + 1]]     ▷ Update derivatives
```

Fig. 5. An efficient implementation of the FES algorithm for any degree. If emits $(x, f(x))$ for all $x \in \{0,1\}^n$, where f is a degree-d polynomial in n variables described by the dense array of coefficients A. The coefficients must be in graded colex order.

The t derivatives that are updated in the j-th iteration are given by the set of *prefixes* of $m^{(j)}$, namely

$$\begin{aligned} m_0 &= 1, \\ m_1 &= x_{i_1}, \\ m_2 &= x_{i_1} x_{i_2}, \\ &\vdots \\ m_t &= x_{i_1} x_{i_2} \dots x_{i_t} = m. \end{aligned}$$

D_{m_i} is updated using $D_{m_{i+1}}$ for $i = t-1, \dots, 0$. This is what happens on line 38 in Figure 5.

Computing the ranks of the t derivatives with respect to m_i is necessary to access them in the array A. It follows from eq. (1) that computing $\mu_k(m_t)$ yields in passing the values of

$$\mu_0(m_0), \mu_1(m_1), \mu_2(m_2), \dots, \mu_{k-1}(m_{k-1}).$$

This naturally extends to v. In other terms, it is possible to compute the ranks (in the graded colex order) of all prefixes of a given degree-t monomial in $O(t)$ operations. This is done in lines 30–34 of the pseudo-code. This is the reason why the FES algorithms favors the graded colex order.

The only extra memory that is needed beyond the input polynomial is for storing the stack and the focus pointers; this makes $O(n)$ words — this is always less than the size of the polynomial if it is non-linear. Once the enumeration is terminated, the original array is left in a modified state. However, it is possible to restore the input state; this requires approximately the same time as the setup phase.

The "setup phase" (lines 11–14) evaluates all non-constant derivatives, and there are $O(n^{d-1})$ of them. A very crude upper-bound on the time complexity of the setup phase is thus given by $O(n^{2d-1})$. It is possible to obtain refined complexity bounds but this is largely irrelevant, because the cost of the setup phase is negligible in front of the $d2^n$ operations required by the enumeration itself in all practical scenarios.

7.4 Correctness

We now prove that the algorithm shown in Figure 5 is correct. Following what has implicitly been done previously, we denote by $\texttt{foo}^{(j)}$ the value of variable foo at the beginning of the j-th iteration of the main loop (the **while** loop of line 16). The first iteration corresponds to $j = 0$.

For the sake of lighter notations, given a monomial m, we use the shorthand $\mathcal{A}[m] = \texttt{A}[\textsc{Rank}(m)]$. In other terms, while A is the regular array accessed in the pseudo-code of Figure 5, $\mathcal{A}$ is a version that can be directly indexed by monomials. For instance, line 38 of the algorithm could be rewritten as

$$\mathcal{A}[m_\ell] \leftarrow \mathcal{A}[m_\ell] \oplus \mathcal{A}[m_{\ell+1}]. \tag{7}$$

Let $\mathrm{LV}(m)$ denotes the index of the *leading variable* of the monomial m, *i.e.* the greatest index of any variable that occurs in m. For instance, $\mathrm{LV}(x_2 x_6) = 6$. We define $\mathrm{LV}(1) = -1$. In fact, $\mathrm{LV}(m)$ is the index of the most significant set bit in the exponent vector of m. Finally, define

$$\psi(m, j) = [\![m]\!] \oplus \big([\![(j \boxminus m) \gg (\mathrm{LV}(m) + 1)]\!] \ll (\mathrm{LV}(m) + 1)\big).$$

The following result fully describes the progress of the computation as the main loop of algorithm of 5 runs.

Lemma 7.2. *At the beginning of each iteration of the loop of line 16, and for all monomial m of degree at most d, we have:*

$$\mathcal{A}^{(j)}[m] = D_m(\psi(m, j)). \tag{$\star$}$$

j	$\psi(m_0, j)$	m_1	$\psi(m_1, j)$	m_2	$\psi(m_2, j)$	m_3
00000	00000	x_0	00001			
00001	00001	x_1	00011			
00010	00011	x_0	00001 $\longrightarrow$ 00011	x_0x_1	00010	
00011	00010	x_2	00110			
00100	00110	x_0	00011 $\longrightarrow$ 00111	x_0x_2	00111	
00101	00111	x_1	00011 $\longrightarrow$ 00111	x_1x_2	00101	
00110	00101	x_0	00111 $\longrightarrow$ 00101	x_0x_1	00010 $\longrightarrow$ 00110	
00111	00100	x_3	01100			
01000	01100	x_0	00101 $\longrightarrow$ 01101	x_0x_3	01101	
01001	01101	x_1	00111 $\longrightarrow$ 01111	x_1x_3	01111	
01010	01111	x_0	01101 $\longrightarrow$ 01111	x_0x_1	00110 $\longrightarrow$ 01110	$x_0x_1x_3$
01011	01110	x_2	00110 $\longrightarrow$ 01110	x_2x_3	01010	
01100	01010	x_0	01111 $\longrightarrow$ 01011	x_0x_2	00111 $\longrightarrow$ 01111	$x_0x_2x_3$
01101	01011	x_1	01111 $\longrightarrow$ 01011	x_1x_2	00101 $\longrightarrow$ 01101	$x_1x_2x_3$
01110	01001	x_0	01011 $\longrightarrow$ 01001	x_0x_1	01110 $\longrightarrow$ 01010	$x_0x_1x_2$
01111	01000	x_4	11000			
10000	11000	x_0	01001 $\longrightarrow$ 11001	x_0x_4	11001	
10001	11001	x_1	01011 $\longrightarrow$ 11011	x_1x_4	11011	
10010	11011	x_0	11001 $\longrightarrow$ 11011	x_0x_1	01010 $\longrightarrow$ 11010	$x_0x_1x_4$
10011	11010	x_2	01110 $\longrightarrow$ 11110	x_2x_4	11110	
10100	11110	x_0	11011 $\longrightarrow$ 11111	x_0x_2	01111 $\longrightarrow$ 11111	$x_0x_2x_4$
10101	11111	x_1	11011 $\longrightarrow$ 11111	x_1x_2	01101 $\longrightarrow$ 11101	$x_1x_2x_4$
10110	11101	x_0	11111 $\longrightarrow$ 11101	x_0x_1	11010 $\longrightarrow$ 11110	$x_0x_1x_2$
10111	11100	x_3	01100 $\longrightarrow$ 11100	x_3x_4	10100	
11000	10100	x_0	11101 $\longrightarrow$ 10101	x_0x_3	01101 $\longrightarrow$ 11101	$x_0x_3x_4$
11001	10101	x_1	11111 $\longrightarrow$ 10111	x_1x_3	01111 $\longrightarrow$ 11111	$x_1x_3x_4$
11010	10111	x_0	10101 $\longrightarrow$ 10111	x_0x_1	11110 $\longrightarrow$ 10110	$x_0x_1x_3$
11011	10110	x_2	11110 $\longrightarrow$ 10110	x_2x_3	01010 $\longrightarrow$ 11010	$x_2x_3x_4$
11100	10010	x_0	10111 $\longrightarrow$ 10011	x_0x_2	11111 $\longrightarrow$ 10111	$x_0x_2x_3$
11101	10011	x_1	10111 $\longrightarrow$ 10011	x_1x_2	11101 $\longrightarrow$ 10101	$x_1x_2x_3$
11110	10001	x_0	10011 $\longrightarrow$ 10001	x_0x_1	10110 $\longrightarrow$ 10010	$x_0x_1x_2$
11111	10000	stop				

Fig. 6. Execution trace of the algorithm with $n = 5$ and $d = 3$.

Lemma 7.2 with $m = 1$ states that the sequence of evaluation points of the polynomial is actually $\psi(1, j) = [\![j]\!]$. In other terms, the algorithm evaluates f on all successive n-bit strings in the order given by the binary reflected gray code. Proving lemma 7.2 thus establishes the correctness of the algorithm.

The table in Figure 6 illustrate the progress of the algorithm on a small example. This shows for instance that $\mathcal{A}[x_1x_2]$ keeps its initial value until the 14-th iteration. The 15-th iteration (with $j = 14$), the 23-th (with $j = 22$) and the 31-th (with $j = 30$) begin with a "new" value that was not present at the beginning of the previous iteration.

We now characterize precisely what happens in each iteration of the main loop. Define $\overline{\rho}(j) = \rho^*(j) - \max \rho^*(j)$. For instance, $\rho^*(1337) = \{0, 3, 4, 5, 8, 10\}$ and $\overline{\rho}(1337) = \{0, 3, 4, 5, 8\}$. Let

$\textsc{Prefix}(m)$ denote the set of all prefixes of the monomial m. For instance $\textsc{Prefix}(x_0x_3x_4) = \{1, x_0, x_0x_3, x_0x_3x_4\}$.

Lemma 7.3. *For all $j \geq 0$ and all monomial m. Then:*

$$\psi(m, j+1) = \begin{cases} \psi(m,j) \oplus E_{\rho_{1+\deg m}(j+1)} & \text{if } m \in \textsc{Prefix}\left(\overline{\rho}(j+1)\right), \\ \psi(m,j) & \text{otherwise} \end{cases}$$

Proof. Set $k = \mathrm{LV}(m) + 1$ and write:

$$j \boxminus m = (x\ a_{k-1} \dots a_1 a_0)_2,$$
$$(j+1) \boxminus m = (x' b_{k-1} \dots b_1 b_0)_2,$$

where x, x' are some bit strings. Rearranging these expressions, we obtain $(j \boxminus m) \gg k = (x)_2$ and $((j+1) \boxminus m) \gg k = (x')_2$.

Suppose that $\psi(m, j) \neq \psi(m, j+1)$. Looking at the definition of ψ, it is clear that $(j \boxminus m) \gg k \neq ((j+1) \boxminus m) \gg k$. This is equivalent to $x \neq x'$ with the above notations. By inspection, the only possibility for this to happen is $j \boxminus m = \left(x1^k\right)_2$ and $(j+1) \boxminus m = \left(x'0^k\right)_2$ with $(x')_2 = (x)_2 + 1$.

It follows that $(j+1) \boxminus m = (x+1) \ll k$. Because the right-hand side is non-zero, we can get rid of the saturating subtraction and obtain

$$j + 1 = m + (x+1) \ll k. \tag{8}$$

Unfolding the definition of ψ then yields:

$$\psi(m, j) = [\![m]\!] \oplus ([\![x]\!] \ll k),$$
$$\psi(m, j+1) = [\![m]\!] \oplus ([\![x+1]\!] \ll k),$$

and therefore $\psi(m,j) \oplus \psi(m, j+1) = [\![x \oplus (x+1)]\!] \ll k$. Lemma 3.1 tells us that $x \oplus (x+1) = \left(1^{\rho(x+1)+1}\right)_2$ and we conclude from from lemma 7.1 that $[\![x \oplus (x+1)]\!] = E_{\rho(x+1)}$.

Eq. (8) implies that the lower-significant bits of $j+1$ coincide with those of the exponent vector of m. More precisely, we have:

$$\rho_i(j+1) = \begin{cases} \rho_i(m) & 1 \leq i \leq \deg m \\ \rho_{i-\deg m}(x+1) + k & \text{otherwise} \end{cases}$$

Therefore, $\rho(x+1) + k = \rho_{1+\deg m}(j+1)$. This enables us to conclude that if $\psi(m,j) \neq \psi(m, j+1)$, then $\psi(m, j+1) = \psi(m,j) \oplus E_{\rho_{\deg m+1}(j+1)}$.

The above reasoning in fact shows that eq. (8) is a necessary and sufficient for condition for $\psi(m,j) \neq \psi(m, j+1)$. Therefore we now seek to characterize the set of monomials m that satisfy (8). It is clear that the constant monomial $m = 1$ does so, therefore we concentrate our attention on non-constant monomials.

Write again $j+1$ and m in binary, as $j+1 = (c_{n-1} \dots c_1 c_0)_2$ and $m = (1 d_{k-2} \dots d_1 d_0)_2$. Then (8) is equivalent to $c_i = d_i$ for $0 \leq i < k$ (this implies that $c_{k-1} = 1$) and $(c_n \dots c_k)_2 \geq 1$.

It appears that each 1 bit in the binary writing of $j+1$, except the leftmost one, gives rise to a monomial m that satisfies (8). Write $\rho^*(j+1) = \{i_0, i_1, \dots, i_\ell\}$. Then $1, x_{i_0}, x_{i_0}x_{i_1}, \dots, x_{i_0} \dots x_{i_{\ell-1}}$ are the monomials that satisfy (8), and this is precisely $\textsc{Prefix}\left(\overline{\rho}(j+1)\right)$. □

We now prove the main result.

Proof of lemma 7.2. Observe first that ($\star$) always hold for degree-d monomials m, because D_m is constant and $\mathcal{A}[m]$ is never modified.

We prove that ($\star$) always holds for derivatives of lesser order by induction on the number of iterations of the main loop. First, it clearly does at the beginning of the first iteration of the

main loop ($j = 0$). Indeed, the "setup phase" of lines 11–14 serves precisely this purpose. It sets $\mathcal{A}[m] = D_m(\llbracket m \rrbracket)$ for all monomials m, and $\psi(m, 0) = \llbracket m \rrbracket$.

Next, suppose that ($\star$) holds at the beginning of the j-th iteration of the main loop for all m of degree at most d, and let us show that it is still the case at the beginning of the $(j+1)$-th iteration.

Write $\rho^*(j+1) = \{i_1, i_2, \ldots, i_t, \ldots\}$. Looking at the pseudo-code, we see that the sequence of modified derivatives $m_0, m_1, \ldots, m_{t-1}$ is precisely $\textsc{Prefix}(\{i_1, \ldots, i_{t-1}\})$. All other derivatives are left untouched, and according to lemma 7.3, they are not supposed to change, so ($\star$) still holds at the beginning of the next iteration for the derivatives that are not modified.

It remains to show that those that *are* modified still satisfy ($\star$) at the beginning of the next iteration. In the sequel, t denotes the value computed on line 26 of the pseudo-code. The algorithm executes the following sequence of $t-1$ updates:

$$\begin{aligned}
\mathcal{A}^{(j+1)}\left[m_{t-1}\right] &\leftarrow \mathcal{A}^{(j)}\left[m_{t-1}\right] \oplus \mathcal{A}^{(j)}\left[m_t\right], \\
\mathcal{A}^{(j+1)}\left[m_{t-2}\right] &\leftarrow \mathcal{A}^{(j)}\left[m_{t-2}\right] \oplus \mathcal{A}^{(j+1)}\left[m_{t-1}\right], \\
&\vdots \\
\mathcal{A}^{(j+1)}\left[m_0\right] &\leftarrow \mathcal{A}^{(j)}\left[m_0\right] \oplus \mathcal{A}^{(j+1)}\left[m_1\right].
\end{aligned}$$

We show by descending induction on $\ell = t-1, t-2, \ldots, 0$, that $\mathcal{A}^{(j+1)}\left[m_\ell\right]$ satisfies ($\star$). This will establish the result of the lemma. The first "updater" $\mathcal{A}^{(j)}\left[m_t\right]$ satisfies ($\star$) because either $t = d$ and it is constant (always correct), or $t < d$ and it is not modified, so it satisfies ($\star$) by induction hypothesis on j.

Next, consider the update:

$$\mathcal{A}^{(j+1)}\left[m_\ell\right] \leftarrow \mathcal{A}^{(j)}\left[m_\ell\right] \oplus \mathcal{A}^{(j+1)}\left[m_{\ell+1}\right]$$

and suppose that $\mathcal{A}^{(j+1)}\left[m_{\ell+1}\right]$ satisfies ($\star$). The initial value $\mathcal{A}^{(j)}\left[m_\ell\right]$ of the "updatee" also satisfies ($\star$) by induction hypothesis on j. The final value $\mathcal{A}^{(j+1)}\left[m_\ell\right]$ of the "updatee" is correct if and only if:

$$D_{m_\ell}\left(\psi(m_\ell, j+1)\right) = D_{m_\ell}\left(\psi(m_\ell, j)\right) \oplus D_{m_{\ell+1}}\left(\psi(m_{\ell+1}, j+1)\right). \tag{9}$$

We conclude the proof by showing that (9) is true. By lemma 7.3, we have

$$\psi(m_\ell, j+1) = \psi(m_\ell, j) \oplus E_{i_{\ell+1}}. \tag{10}$$

Eq. (9) is therefore equivalent to

$$D_{m_\ell}\left(\psi(m_\ell, j) \oplus E_{i_{\ell+1}}\right) = D_{m_\ell}\left(\psi(m_\ell, j)\right) \oplus D_{m_{\ell+1}}\left(\psi(m_{\ell+1}, j+1)\right). \tag{11}$$

Note that $m_{\ell+1} = m_\ell x_{i_{\ell+1}}$. By definition of the derivative, we have that

$$D_{m_\ell}\left(\psi(m_\ell, j) \oplus E_{i_{\ell+1}}\right) = D_{m_\ell}\left(\psi(m_\ell, j)\right) \oplus D_{m_{\ell+1}}\left(\psi(m_\ell, j)\right).$$

Therefore (9) is equivalent to

$$D_{m_{\ell+1}}\left(\psi(m_\ell, j)\right) = D_{m_{\ell+1}}\left(\psi(m_{\ell+1}, j+1)\right).$$

Using (10) again, this becomes

$$D_{m_{\ell+1}}\left(\psi(m_\ell, j)\right) = D_{m_{\ell+1}}\left(\psi(m_\ell, j) \oplus E_{i_{\ell+1}}\right).$$

This last equation is always satisfied, because $D_{m_{\ell+1}}$ does not depend on $x_{i_{\ell+1}}$. □

```
/*
 * A[0:2**n] contains the truth table of f;
 * this overwrites it with its Moebius transform
 */
void Moebius(bool *A, int n)
{
        int S = 1;
        int N = 1 << n;
        for (int i = 0; i < n; i++) {
                for (int p = 0; p < N; p += 2 * S)
                        for (int j = 0; j < S; j++)
                                A[p + S + j] ^= A[p + j];
                S += S;
        }
}
```

Fig. 7. C code of the classic Moebius transform, adapted from [Jou09].

8 NEARLY IN-PLACE MOEBIUS TRANSFORM FOR LOW-DEGREE POLYNOMIALS

The "classic", usual Moebius transform converts the truth table of a boolean function to its algebraic normal form, *i.e.* the coefficients of its polynomial representation, and vice-versa (it is involutive). It can be computed in-place by a few lines of C code, as shown in Figure 7. The interested reader can consult [Jou09, §9.2] for more details. This well-known procedure requires $O(n2^n)$ operations. It applies to arbitrary Boolean functions and therefore operates on an array of size 2^n.

Designing an *in-place* Moebius transform that converts the coefficients of a *low-degree* polynomial to its truth table presents an obvious and seemingly insurmountable challenge: the input is much smaller than the output. Indeed, on input we have an array of size $O(n^d)$ holding the coefficients of a degree-d polynomial, while on output we should provide an array of size 2^n containing its truth table.

The only way such a procedure could use less than 2^n bits of memory is by incrementally producing the truth table, one chunk at a time, without ever holding its 2^n entries in memory all at once. The algorithm implemented in BeanPolE is adapted from [Din21a], which is not in-place. It relies on two sub-algorithms:

- The classic in-place Moebius transform that operates on arrays of size 2^n.
- An in-place procedure to "flip" the last variable in sub-linear time.

It assumes that $n \geq d$ and works as follows:

1. If there are exactly d variables:
   a. Run the "classic" in-place Moebius transform described in Figure 7. [this turn the 2^d coefficients of the polynomial into its truth table of size 2^d]
   b. Emit the corresponding chunk of 2^d entries of the truth table.
   c. Run the classic in-place Moebius again. [this reverts the polynomial to its original state]
2. Otherwise:
   a. Proceed recursively on $f(x_0, \ldots, x_{n-2}, 0)$ — one less variable.
   b. Proceed recursively on $f(x_0, \ldots, x_{n-2}, 1)$ — one less variable.

This procedure uses the colexicographic order for two reasons. Firstly, it is required by the classic Moebius transform. Secondly, it enables the last variable to be fixed in-place, and the operation to be efficiently reverted.

Let A denote the array representing the coefficients of a degree-d polynomial f in n variable in colex order. A has size $\binom{n}{\downarrow d}$. Denote by f_b the polynomial in one less variable given by $f(x_0, \ldots, x_{n-2}, b)$.

The description of f_0 is in fact readily available in A: all the coefficients of monomials containing the last variable are located at the end of A in the colex order. Therefore the first $\binom{n-1}{\downarrow d}$ elements of A are precisely the coefficients of f_0.

Obtaining f_1 requires a bit more work. To this end, our main tool is an efficient procedure that flips the last variable in-place: it overwrites the coefficients of f with those of $f(x_0, \ldots, x_{n-2}, x_{n-1} + 1)$. This operation is clearly involutive. Working with the coefficients of f_1 is done as follows:

- Flip the last variable in-place. [the beginning of A then contains the coefficients of f_1].
- Do whatever is needed with f_1.
- Flip the last variable in-place once more. [this restores A to its original state].

8.1 Flipping the Last Variable In-Place

Let m denote a monomials that does not contain x_{n-1}. The transformation $x_{n-1} \mapsto x_{n-1} + 1$ leaves m invariant and turns mx_{n-1} into $m + mx_{n-1}$. Let B denote the array containing coefficients of $f(x_0, \ldots, x_{n-2}, x_{n-1} + 1)$. Assuming that A and B can be indexed with monomials, we have:

$$B[mx_{n-1}] = A[mx_{n-1}]$$
$$B[m] = A[m] \oplus A[mx_{n-1}]$$

We now describe an in-place procedure that overwrites the content of A with that of B. For each monomial m of degree at most d in $n-1$ variables: let i (resp. j) denote the colex rank of m (resp. mx_{n-1}) among all degree-d monomials, then do $A[i] \leftarrow A[i] \oplus A[j]$. The only technical difficulty is to compute the ranks i and j efficiently.

Suppose that $r = \mu_{\downarrow d}(\{i_0, \ldots, i_{k-1}\})$ is the colex rank of a monomial among all degree-d monomials. It follows from (2) that is easy to update r if the smallest variable is removed, or if another even smaller variable $j < i_0$ is added:

$$\mu_{\downarrow d}(\{i_1, \ldots, i_{k-1}\}) = r - \binom{i_0}{\downarrow d - k + 1} \tag{12}$$

$$\mu_{\downarrow d}(\{j, i_0, \ldots, i_{k-1}\}) = r + \binom{j}{\downarrow d - k} \tag{13}$$

This ability to "update" the rank of a monomial in constant time plays a crucial role in the procedure that flips the last variable of a Boolean polynomial described in Figure 8.

Each iteration of the **for** loop uses the algorithm of Figure 2 to advance to the next monomial m in $n-1$ variables of degree at most $(d-1)$. It also maintains the colex rank of m among all monomials of degree at most d using equations (12) and (13). During this enumeration, the rank of mx_{n-1} increases monotonically from $\binom{n-1}{\downarrow d}$ to $\binom{n}{\downarrow d}$.

The correctness of this procedure follows from lemma 5.1 and from the discussion above about incrementally updating the rank of a monomial. The proof of lemma 5.2 also shows that it runs in amortized constant time per iteration. It follows that flipping the last variable takes time proportional to $\binom{n-1}{\downarrow d-1}$, namely the number of updated coefficients.

8.2 An Iterative Nearly In-place Moebius Transform

The full pseudo-code of the in-place Moebius transform implemented in BeanPolE is shown in Figure 9. It remains to determine its complexity. The classic Moebius transform is invoked 2^{n-d+1} times on arrays of size 2^d, so the total number of operations this represents is $O(d2^n)$. We now

```
1:  procedure FlipLastVariable(A, n, d)
2:      k ← 0                                                    ▷ k = deg m
3:      m[d − 1] ← 0
4:      m[d] ← 1                          ▷ variables of m appear in m[d − k + 1 : d]
5:      i ← 0                                                    ▷ i = μ_{↓d−1}(m)
6:      for (n−1 choose ↓d) <= j < (n choose ↓d) do   ▷ j = μ_{↓d}(m x_{n−1}) with deg m ≤ d − 1
7:          A[i] ← A[i] ⊕ A[j]                                   ▷ Update A
            # The sequel advances to the next m using the algorithm of Figure 2.
            # It maintains the corresponding rank i using (12) and (13).
8:          if k ≠ d − 1 and m[d − k] ≠ 0 then
9:              k ← k + 1                                        ▷ Add x_0 to m
10:             m[d − k] ← 0
11:             i ← i + 1                          ▷ add smallest variable using eq. (13)
12:         else
13:             while k > 1 and m[d − k] + 1 = m[d − k + 1] do    ▷ Erase smallest "run"
14:                 i = i − (m[d−k] choose ↓d−k+1)          ▷ remove smallest variable using eq. (12)
15:                 k ← k − 1
16:             i = i − (m[d−k] choose ↓d−k+1)              ▷ remove smallest variable using eq. (12)
17:             m[d − k] = m[d − k] + 1          ▷ Replace smallest variable by the next one
18:             i = i + (m[d−k] choose ↓d−k+1)              ▷ add smallest variable using eq. (13)
```

Fig. 8. Flipping the last variable of a Boolean polynomial in colex order, in-place.

```
1:  procedure InPlaceMoebius(A, n, d)
2:      Allocate an array x of n + 1 bits
3:      Set x_i ← 0 for all 0 ≤ i ≤ n.
4:      Moebius(A, d)                                  ▷ coefficients → truth table
5:      while x[n] = 0 do
            # At this point A[0 : 2^d] contains the next chunk of 2^d entries of the truth table
6:          emit A[0 : 2^d]
7:          Moebius(A, d)                              ▷ truth table → coefficients
8:          Set i ← d                                  ▷ unwind the recursion stack
9:          while i < n and x[i] = 1 do
10:             Set x[i] ← 0
11:             FlipLastVariable(A, i + 1, d)
12:             Increment i
13:         Set x[i] ← 1                               ▷ flip the last variable
14:         if i < n then
15:             FlipLastVariable(A, i + 1, d)
16:             Moebius(A, d)                          ▷ coefficients → truth table
```

Fig. 9. The nearly in-place Moebius transform for low-degree polynomials.

claim that the total time spent flipping variables is also $O(d2^n)$. The $(d+i)$-th variable is flipped $2^{n-d-i-1}$ times. The total time spent in variable flipping is then (up to a constant factor)

$$T = \sum_{i=d}^{n-1} 2^{n-i-1} \binom{i-1}{\downarrow d-1}$$

Suppose that $d \leq n/2$. Under this assumption, the trivial bound $\binom{n}{\downarrow k} \leq k\binom{n}{k}$ shows that the time spent flipping variables is upper-bounded by

$$T \leq d2^{n-1} \sum_{i=d}^{n-1} 2^{-i} \binom{i-1}{d-1}$$

Because $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$, it follows that:

$$T \leq d^2 2^{n-1} \sum_{i=d}^{n-1} \frac{2^{-i}}{i} \binom{i}{d} \tag{14}$$

But we then have the following

Lemma 8.1. *For any $d \in \mathbb{N}$,* $\displaystyle\sum_{i=0}^{+\infty} 2^{-i} \binom{i}{d} = 2.$

Proof. To begin with, it is clear that the series converges by the ratio test. We establish the result using the method of "creative telescoping" [PWZ96]. Define $F(d,i) = 2^{-i}\binom{i}{d}$ and $G(d,i) = 2\frac{d-i}{d+1}F(d,i)$. Then we have

$$F(d+1,i) - F(d,i) = G(d,i+1) - G(d,i).$$

Summing on $0 \leq i \leq n$ yields

$$\sum_{i=0}^{n} F(d+1,i) - \sum_{i=0}^{n} F(d,i) = G(d,n+1) - G(d,0).$$

In all cases, $G(d,0) = 0$. Then passing to the limit with $n \longrightarrow +\infty$, we find that:

$$\sum_{i=0}^{+\infty} F(d+1,i) - \sum_{i=0}^{+\infty} F(d,i) = \lim_{n\to+\infty} G(d,n+1) = 0.$$

This shows that the sum of the series is independent of d. Then with $d = 0$ we quickly find that

$$\sum_{i=0}^{+\infty} F(0,i) = \sum_{i=0}^{+\infty} 2^{-i} = 2.$$

□

It then follows from lemma 8.1 that

$$\sum_{i=d}^{n-1} \frac{2^{-i}}{i} \binom{i}{d} \leq \frac{1}{d} \sum_{i=d}^{n-1} 2^{-i} \binom{i}{d} \leq \frac{2}{d}$$

Combining this with (14) finally shows that the total time spent flipping variable is $O(d2^n)$. This establishes the announced complexity.

9 PRACTICAL RESULTS

To the best of our knowledge, there is no other software package devoted to handling dense multivariate Boolean polynomials. Speed comparisons are thus of little interest. The few experiments reported below were performed on a machine equipped with a pair of Intel Xeon Gold 6130 (*Skylake*) CPUs running at 2.1GHz. To ensure a frequency of exactly 2.1GHz, "Turbo Boost" was disabled. Only a single core was used, with a single pinned process. A single core has a 32KB L1 cache, a 1MB L2 cache and a 22MB L3 cache is shared with all the (otherwise idle) cores of the CPU.

Note that when used to measure size in bytes or in bits, the suffixes G, M and K mean 2^{30}, 2^{20} and 2^{10}, respectively, but in any other context they mean 10^9, 10^6 and 10^3.

The most closely related software package is without doubt the PolyBoRi library, but it does not target the same applications. PolyBori uses Binary Decision Diagrams to represent the set of Boolean monomials of each polynomial. This is particularly well-suited for sparse polynomials in a high number of variables. This makes PolyBori capable of handling sparse, high-degree polynomials in a large number of variable, something that BeanPolE cannot do because their dense representation would use too much space. The downside is that PolyBori is not so efficient with dense polynomials.

Multiplication. Before writing BeanPolE, the author was facing the following practical problem: compute $R \leftarrow R + P \times Q$, where P and Q are random polynomials of degree 2 and 3, respectively, in 80 variables. Polynomial multiplication is not the main feature of BeanPolE, but this example illustrates that, by design, it is better suited at dealing with dense, low-degree polynomials than existing software.

The naive quadratic multiplication algorithm described in section 6.3 represents 20 lines of C code in BeanPolE. It computes the product ($R \leftarrow R + P \times Q$) in 10.8s, using the most compact dense representation with a single bit per coefficient. The result R requires 3MB of memory. Using a simpler memory representation with an array of `bool` (one per coefficient) yields essentially the same performance but uses 8 times more memory, as `gcc` uses 8-bit integers to represent values of type `bool`. Finally, performing 32 parallel, "vectorized", multiplications using 32-bit coefficients takes exactly the same amount of time as doing just one.

The SageMath v9.5 open-source computer algebra system uses PolyBoRi to handle multivariate Boolean Polynomials. A direct performance comparison is complicated by the fact that PolyBoRi maintains a global state; the performance of the next operation is affected by the history of past operations and tends to degrade over time. This concerns both the computation of the product ($R \leftarrow R + P \times Q$) and the generation of random P, Q. We therefore repeated 10 times the operation, starting with $R = 0$ and drawing random P and Q in each iteration. Figure 10 shows the result. The memory footprint increases during the first iterations then stabilizes. This requires 581MB of memory to store R, about 24 bytes per coefficient. With $n = 128$ variables, BeanPolE does a single multiplication of a bit-packed polynomial in 112s; the result R requires 33MB. SageMath does it in 195s; the result requires 7620MB.

The MAGMA v2.26-8 closed-source computer algebra system [BCP97] also offers facilities for dealing with Boolean polynomials. Its documentation does not explicitly specify what kind of data structure it uses to represent them in memory, but the same experiment suggests the use of sparse polynomials. The time taken to do $R \leftarrow R + P \times Q$ depends on the previous value of R. On a machine[3] equipped with Intel Xeon CPU E5-2680 v4 CPUs, MAGMA takes 365s to compute When $R \leftarrow P \times Q$. The memory usage increasd from 64MB to 4GB, which makes about 160 bytes per coefficient of R. We guess that each exponent is represented using a 16-bit integer. After that, subsequent updates $R \leftarrow R + P \times Q$, with fresh P and Q, take 1065s.

[3]We unfortunately do not own a MAGMA licence so we had to rely on a third party to run the experiment for us.

n	Operation	Time (1st iteration)	Time (10th iteration)	Memory footprint
80	random P	0.05	1	581MB
	random Q	4.5	60	
	$R \leftarrow R + P \times Q$	15	14	
128	random P	0.15	25	7.6GB
	random Q	26	3047	
	$R \leftarrow R + P \times Q$	195	182	

Fig. 10. Performance of the polynomial product using SageMath 9.5, that itself uses PolyBori. Times are given in seconds. P has degree 2 and Q has degree 3.

Boolean System Solving. The BeanPolE library is bundled with a few demonstration programs. Two of them solve systems of 64 polynomial equations in n variables by exhaustive search, using either the FES algorithm or the in-place Moebius transform. They evaluate the 64 input polynomials on all the 2^n input values and check if all the polynomials vanish simultaneously. Figures 11 and 12 show a speed comparison between the FES algorithm and the in-place Moebius transform.

Here are a few comments. The main loop of the FES algorithm always seem to be faster than that of the Moebius transform by a factor of about two. It seems that this gap narrows when the degree increase. Both algorithm require $O(d)$ amortized operations to produce the next entry of the truth table. Figure 13 shows the number of CPU cycles per entry of the truth table, divided by d — this exposes the constant factor inside the $O(\ldots)$. It confirms that the dependence to n is very weak. Such a dependence, which is not visible in the theoretical analysis in a simplified abstract model, could be exposed by architectural details of the target machine: the algorithms randomly access an array that gets larger with n, so that cache misses, TLB faults, etc. could become more frequent when n increases. We observe that it is not the case, and that the memory access pattern of the algorithms is dealt with efficiently by the memory subsystem. In any case, the slight superiority of the main loop of the FES algorithm is manifest.

However, the FES algorithm has a setup phase before its main loop, and our experiments show that in some circumstances its complexity cannot be neglected. This is in particular the case when d is not negligible compared to n. Figure 14 shows the running time of the setup phase. In the case of $n = 32$ variables for instance, the Moebius transform gets faster than the FES algorithm as soon as $d \geq 12$, because the setup phase dominates. Numerical estimates suggest that it is always the case when $d = n/3$. However, when n increases and d stays fixed, the complexity of the setup phase tends to become negligible compared to that of the main loop. When it is the case, gaining a factor of two in the main loop by switching from the Moebius transform to the FES algorithm yields a global improvement in running time.

For the sake of comparison, the libfes-lite library, which contains a very optimized version of the FES algorithm restricted to quadratic polynomials does this exhaustive search in 0.15s, about 200× faster than the BeanPolE implementation. However, BeanPolE is written in plain C, does not use bits of hand-tuned CPU-specific assembler code, and works for any degree.

As reported in the introduction, several cryptographic attacks require the computation of the truth table of low-degree Boolean polynomials. The time BeanPolE would take to perform these tasks can be inferred from the tables shown in Figure 11 and 12. Producing the truth table of degree-4 polynomials in 33 variables (as in [BDL+21]) would take 222s on a core of the target machine, while for degree-6 polynomials in 32 variables (as in [DS11]) it would take 151s.

Fig. 11. Using BeanPolE to solve Boolean polynomial systems by exhaustive search. Random systems of 64 polynomials in $n = 32$ variables of increasing degrees are solved. The "size" column indicates the total size of the polynomial system, using a single bit to store each coefficient. The "FES" and "Moebius" columns indicate the running time of the full enumeration, as well as the number of CPU cycles per entry of the truth table. For the FES algorithm, the running of the setup has to be added to the running time of the main loop. The running time of the main loop was extrapolated from the first 2^{30} iterations.

n	d	Size (B)	FES			Moebius	
			setup	main loop	cycles / it	main loop	cycles / it
32	2	4.1K	0.0s	78.0s	38	122s	60
	3	42.9K	0.0s	96.4s	47	166s	81
	4	324K	0.0s	111s	54	213s	104
	5	1.9M	0.0s	125s	61	252s	123
	6	8.8M	0.1s	151s	74	299s	146
	7	34.4M	0.4s	175s	86	348s	170
	8	115M	2.1s	191s	93	397s	194
	9	329M	9.0s	213s	104	437s	214
	10	821M	33.8s	229s	112	486s	238
	11	1.8G	114s	251s	123	527s	258
	12	3.4G	351s	303s	148	564s	276
	13	6.0G	996s	356s	174	611s	299
	14	9.5G	2544s	407s	199	651s	318
	15	13.8G	1.7h	421s	206	670s	327
	16	18.2G	3.5h	452s	221	668s	327
36	2	5.2K	0.0s	1248s	38	1952s	60
	3	61.0K	0.0s	1606s	49	2714s	83
	4	521K	0.0s	1798s	55	3366s	103
	5	3.4M	0.0s	2010s	61	1.1h	124
	6	18.2M	0.2s	2394s	73	1.3h	145
	7	81.9M	1.0s	2714s	83	1.6h	172
	8	313M	5.3s	3002s	92	1.7h	192
	9	1.0G	25.0s	3379s	103	2.0h	218
	10	2.9G	108s	1.1h	116	2.2h	241
	11	7.4G	410s	1.1h	125	2.3h	256
	12	16.7G	1429s	1.4h	149	2.5h	280
	13	33.9G	1.3h	1.6h	174	2.7h	292
40	2	6.4K	0.0s	5.5h	38	9.0h	62
	3	83.6K	0.0s	7.0h	48	12.1h	83
	4	798K	0.0s	8.0h	55	15.2h	104
	5	5.8M	0.0s	9.5h	65	18.1h	124
	6	35.1M	0.3s	10.6h	73	21.7h	149
	7	177M	2.0s	12.5h	86	25.4h	175
	8	764M	11.7s	13.6h	93	28.6h	196
	9	2.8G	62.4s	15.4h	106	32.1h	221
	10	9.1G	294s	16.6h	114	34.2h	235
	11	26.3G	1263s	17.9h	123	37.3h	257
	12	67.9G	1.4h	22.1h	152	41.5h	285

Fig. 12. Figure 11, continued.

n	d	Size (B)	FES			Moebius	
			setup (s)	main loop (s)	cycles / it	main loop (s)	cycles / it
44	2	7.7K	0.0s	90.1h	39	141.5h	61
	3	111K	0.0s	106.5h	46	195.7h	84
	4	1.1M	0.0s	125.2h	54	241.2h	104
	5	9.4M	0.1s	149.3h	64	294.9h	127
	6	63.3M	0.5s	166.6h	72	351.3h	151
	7	356M	3.7s	197.1h	85	405.0h	174
	8	1.7G	24.4s	215.7h	93	455.1h	196
	9	6.9G	140s	243.9h	105	509.3h	219
	10	25.4G	734s	263.1h	113	549.8h	236
	11	82.6G	3473s	292.6h	126	600.3h	258
48	2	9.2K	0.0s	1.4Kh	39	2.3Kh	61
	3	144K	0.0s	1.7Kh	46	3.1Kh	84
	4	1.6M	0.0s	2.0Kh	54	3.8Kh	103
	5	14.7M	0.1s	2.4Kh	64	4.6Kh	125
	6	108M	0.8s	2.6Kh	71	5.5Kh	147
	7	670M	6.7s	3.1Kh	83	6.5Kh	174
	8	3.5G	47.1s	3.4Kh	92	7.3Kh	197
	9	16.0G	297s	3.9Kh	104	8.2Kh	220
	10	64.7G	1688s	4.3Kh	116	9.0Kh	243

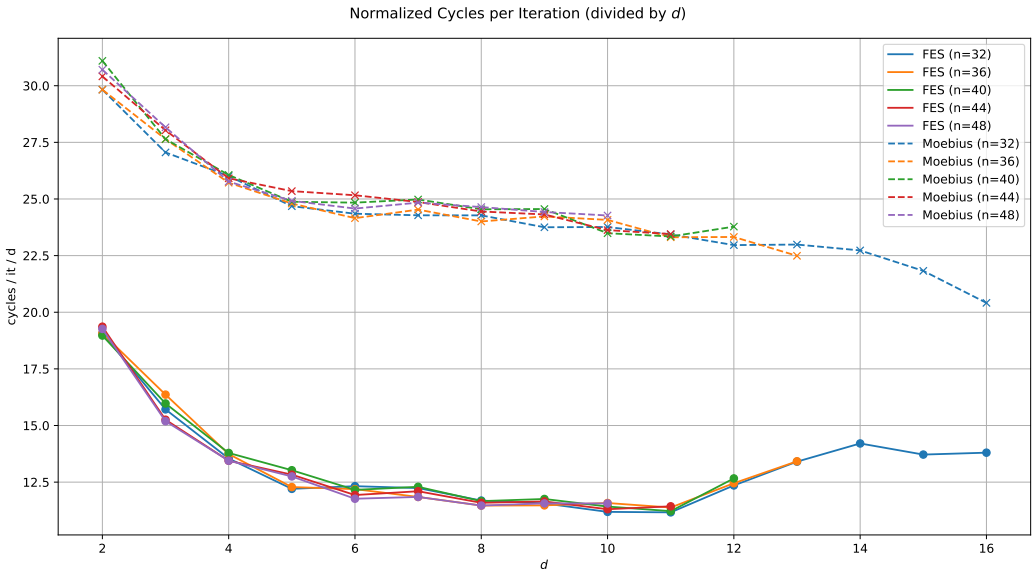

Fig. 13. Comparison of the main loops of both algorithms.

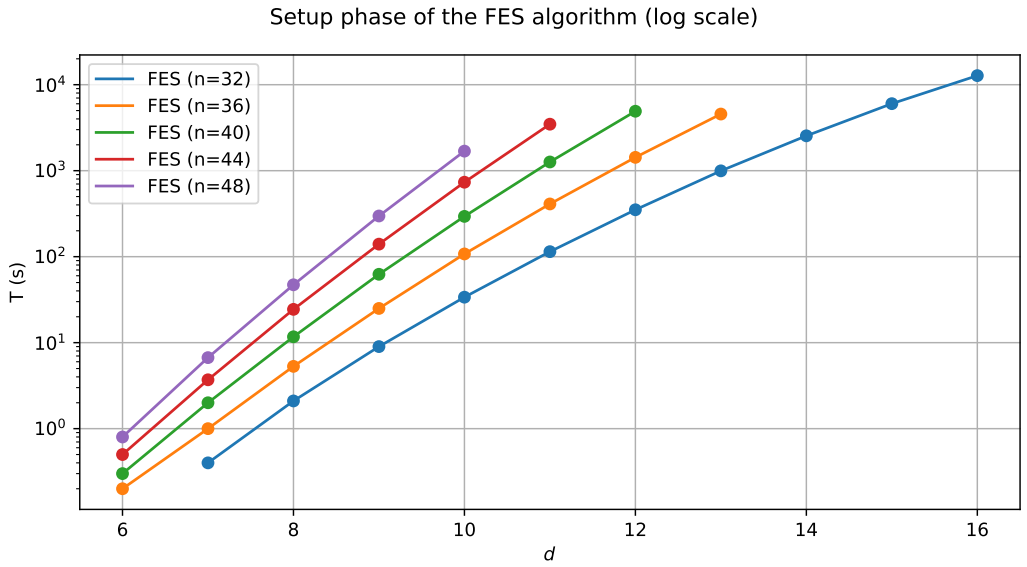

Fig. 14. Running time of the setup phase of the FES algorithm.

Use in the "Crossbred" algorithm. We conclude with the most relevant use of the library. The "Crossbred" algorithm [JV17] is presently the most efficient solution to solve quadratic Boolean systems. Its most computationally heavy phase requires the evaluation of a large collection of degree-D polynomials on all possible inputs, with $D \in \{3, 4, 5\}$.

A record-size system of 186 random quadratic polynomials in 83 variables has recently been solved using the Crossbred algorithm implemented on top of BeanPolE[4]. If AVX512 instructions are available (as it is the case on the "*Skylake*" Xeon CPUs), this can be done by evaluating 532K degree-4 polynomials and 20.5K degree-5 polynomials on 48 variables. The FES algorithm was used, because it is slightly faster. These large collections of w polynomials are represented as a single polynomial with w-bit coefficients.

The coefficients are so large that the "derivative update" step of line 38 in Figure 5 completely dominates the running time of the FES algorithm. Let us for instance consider the case of 532480 degree-4 polynomials, that was executed in practice. The critical "derivative update" step boils down to executing the sequence of operations:

$$d \leftarrow d \oplus e$$
$$c \leftarrow c \oplus d$$
$$b \leftarrow b \oplus c$$
$$a \leftarrow a \oplus b$$

where a, b, c, d and e denote long bit strings of size 532480. Each iteration needs to read a, b, c, d and e ($5 \times 532Kb = 325KB$) from memory and writes back a, b, c and d ($4 \times 532Kb = 260KB$). It is the understanding of the author that a "*skylake*" core has two 512-bit load units, two 512-bit integer Arithmetic and Logical Units as well as a single 512-bit store unit. Figure 15 summarizes this. Performing all the XORs operation requires at least 2080 cycles, but writing the updated values to memory requires at least 4160 cycles (because there is only a single store unit).

[4]See https://www.mqchallenge.org/details/details_I_20230916.html

operation	Maximum instructions / cycle	
	AVX2 (256-bit vectors)	AVX-512 (512-bit vectors)
register ⟵ register ⊕ register	3	2
register ⟵ memory	2	2
memory ⟵ register	1	1

Fig. 15. Instruction-level parallelism in a "*Skylake*" CPU core.

Access every ... iteration	1	2	4	8	16
Coefficient	1	x_0	x_1	x_2	x_3
			x_0x_1	x_0x_2	x_0x_3
				x_1x_2	x_1x_3
				$x_0x_1x_2$	x_2x_3
					$x_0x_1x_3$
					$x_0x_2x_3$
					$x_1x_2x_3$
					$x_0x_1x_2x_3$

Fig. 16. Most frequently accessed coefficients in the FES algorithm.

In addition, a single coefficient is about twice the size of the L1 cache. We also understand that the bandwidth between the L1 and L2 cache is only 512 bits per cycle in either direction. Only one operation can be done each cycle: the L1 cache can either send or receive data from the L2 cache, but not both. This creates an extra bottleneck and leads to a more stringent theoretical lower-bound of 9360 cycles per iteration.

Finally, the bandwidth between the L2 and the L3 cache is at most 256 bits per cycle in either direction. The coefficient are so large that the 1MB L2 cache can only store about 15.75 of them. It follows that L2 cache misses are bound to be frequent, and the L2 ↔ L3 bandwidth can also be a limiting factor.

In the best of all possible worlds, the 16 most frequently accessed coefficients (shown in Figure 16) would always stay in the L2 cache and never be evicted. In this ideal scenario, there would be on expectation 1/16 L2 cache misses for degree-1 coefficients per iteration, plus 5/16 misses for degree-2 coefficients, plus 11/16 for degree-3 coefficients, plus 15/16 for degree-4 coefficients. Any higher-order coefficient would create a cache miss. Therefore, exactly $D - 2$ coefficients would be transferred from the L3 cache to the L2 cache on expectation, and $D - 3$ would be transferred back to the L3 cache per iteration.

However, the L2 cache unfortunately most likely implements a variant of the Least Recently Used replacement strategy, so that the situation is both worse and messier in practice. In this case, a simple simulation shows that when $D = 4$, only the four most frequently accessed coefficients ($1, x_0, x_1$ and x_0x_1) always stay in the L2 cache; all the other are almost systematically evicted. Under this assumption, 3 coefficients would have to be read from the L3 cache on expectation per iteration, and 2 would have to be written back. This means that 5 full coefficients would have to be transferred to and from the L3 cache per iteration, and its bandwidth limitation would then impose a lower-bound of 10400 cycles per iteration.

Running the actual code, we observe using actual hardware performance counters that 3.2 coefficients are loaded on average from the L3 cache per iteration. This confirms the result of the simulation. Our code actually runs at 24850 cycles / iteration. About 13.7 bytes are transferred to or

from the L3 cache per cycle. This is less than the theoretical maximum of 32. However, [Int23, Table 2.9] suggests a "Sustained bandwidth" of 15 bytes per cycle, and [VSIH22] empirically measured it at 11.3 bytes / cycle, so we are somewhere in the middle. This leads us to conclude that, with a large number of polynomials, the FES algorithm is in fact memory-bound and runs close to the peak performance of the hardware.

Another simple experiment confirms this observation: an AVX2 version was assembled and benchmarked. It runs at exactly the same speed as the AVX512 version. Looking at the table of Figure 15, we see that when restricted to AVX2 instructions, in a single clock cycle a CPU core can only:

- Load 512 bits from the L1 cache (vs. 1024 with AVX512)
- XOR 768 bits (vs 1024)
- Store 256 bits in the L1 cache (vs 512)

We conclude that if the AVX2 version was bottlenecked by the speed at which the CPU can execute instructions without stalls, then it would be slower than the AVX512 version by a factor between 1.33 and 2. The fact that it runs at the same speed then strongly suggest that the process is memory-bound.

On the target machine, the full evaluation of 532K degree-4 polynomials on 48 variables would have taken about 9.25M CPU·hours.

10 EXTENSIONS AND FUTURE WORK

This concluding section discusses how the BeanPolE library could be extended, and pinpoints some research perspectives.

The in-place Moebius transform emits the truth table in chunks of size 2^d. It would not be difficult to modify the code to obtain it in chunks of size 2^k, with $k \neq d$. If $k > d$, then it would need to be out-of-place, as in the original presentation. This could potentially be more practical for some use cases. The runtime would increase to $O(k2^n)$.

While the FES algorithm is intrinsically sequential, the Moebius transform offers potential for parallelization both inside the "classic" Moebius transform and inside the "last variable flip" sub-algorithm. It could also be possible to "backport" the idea to use a Gray code in the Moebius transform. It could lead to a constant speed-up, at the expense of not visiting the truth table in lexicographic order.

With Boolean polynomials, evaluation and interpolation are very similar, and sometimes they coincide: the classic Moebius transform does both. Adapting the in-place Moebius transform to interpolate a degree-d polynomial seems relatively straightforward. Turning the FES algorithm into an interpolation algorithm is less direct, and thus more interesting.

This also opens up an interesting algorithmic perspective: a degree-d polynomial can be interpolated from $\binom{n}{\downarrow d}$ evaluations, for instance with its value on all monomials of degree at most d. Designing a fast procedure to convert these $\binom{n}{\downarrow d}$ evaluations to the $\binom{n}{\downarrow d}$ coefficients of the polynomial would be interesting.

In the reverse direction, evaluating a degree-d polynomial on all bit strings of Hamming weight at most d would be relevant. In [Din21a], Dinur suggests to use the FES algorithm for this purpose, on the basis that there exist "monotonic" Gray codes that enumerate all bit strings by increasing Hamming weight, while flipping one variable at a time.

Acknowledgments

Many thanks to Itai Dinur, Hiroki Furue and Quentin Hammerer for thought-provoking discussions. The author is indebted to the two anonymous reviewers who provided accurate and in-depth comments that helped improve the presentation of this work.

We acknowledge financial support from the French *Agence Nationale de la Recherche* under project "GORILLA" (ANR-20-CE39-0002) and "PostCryptum" (ANR20-ASTR-0011).

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). This work also benefited from access to the HPC resources of the "MatriCS Platform" at the Université de Picardie Jules Verne.

REFERENCES

[ACZ16] José Antonio Álvarez Cubero and Pedro J. Zufiria. Algorithm 959: Vbf: A library of c++ classes for vector boolean functions in cryptography. *ACM Trans. Math. Softw.*, 42(2), may 2016.

[BCC+10] Charles Bouillaguet, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir, and Bo-Yin Yang. Fast exhaustive search for polynomial systems in $\mathbb{F}_2$. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2010.

[BCP97] Wieb Bosma, John J. Cannon, and Catherine Playoust. The Magma Algebra System I: The User Language. *J. Symb. Comput.*, 24(3/4):235–265, 1997.

[BD09] Michael Brickenstein and Alexander Dreyer. Polybori: A framework for gröbner-basis computations with boolean polynomials. *Journal of Symbolic Computation*, 44(9):1326 – 1345, 2009. Effective Methods in Algebraic Geometry.

[BDL+21] Christof Beierle, Patrick Derbez, Gregor Leander, Gaëtan Leurent, Håvard Raddum, Yann Rotella, David Rupprecht, and Lukas Stennes. Cryptanalysis of the GPRS encryption algorithms GEA-1 and GEA-2. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II*, volume 12697 of *Lecture Notes in Computer Science*, pages 155–183. Springer, 2021.

[BDT22] Charles Bouillaguet, Claire Delaplace, and Monika Trimoska. A simple deterministic algorithm for systems of quadratic polynomials over gf(2). In Karl Bringmann and Timothy Chan, editors, *5th Symposium on Simplicity in Algorithms, SOSA@SODA 2022, Virtual Conference, January 10-11, 2022*, pages 285–296. SIAM, 2022.

[BER76] James R. Bitner, Gideon Ehrlich, and Edward M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Commun. ACM*, 19(9):517–521, 1976.

[BFSS13] Magali Bardet, Jean-Charles Faugère, Bruno Salvy, and Pierre-Jean Spaenlehauer. On the complexity of solving quadratic boolean systems. *J. Complexity*, 29(1):53–75, 2013.

[BGTZ08] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster multiplication in gf(2)[x]. In Alfred J. van der Poorten and Andreas Stein, editors, *Algorithmic Number Theory, 8th International Symposium, ANTS-VIII, Banff, Canada, May 17-22, 2008, Proceedings*, volume 5011 of *Lecture Notes in Computer Science*, pages 153–166. Springer, 2008.

[BKW19] Andreas Björklund, Petteri Kaski, and Ryan Williams. Solving systems of polynomial equations over GF(2) by a parity-counting self-reduction. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 26:1–26:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[BP17] Ward Beullens and Bart Preneel. Field lifting for smaller UOV public keys. In Arpita Patra and Nigel P. Smart, editors, *Progress in Cryptology - INDOCRYPT 2017 - 18th International Conference on Cryptology in India, Chennai, India, December 10-13, 2017, Proceedings*, volume 10698 of *Lecture Notes in Computer Science*, pages 227–246. Springer, 2017.

[CFMR+17] Antoine Casanova, Jean-Charles Faugère, Gilles Macario-Rat, Jacques Patarin, Ludovic Perret, and Jocelyn Ryckeghem. GeMSS: A Great Multivariate Short Signature. Research report, UPMC - Paris 6 Sorbonne Universités ; INRIA Paris Research Centre, MAMBA Team, F-75012, Paris, France ; LIP6 - Laboratoire d'Informatique de Paris 6, December 2017.

[CHR+16] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. From 5-pass *MQ*-based identification to *MQ*-based signatures. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, volume 10032 of *Lecture Notes in Computer Science*, pages 135–165, 2016.

[CLO07] David A. Cox, John Little, and Donal O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, 3/e (Undergraduate Texts in Mathematics)*. Springer-Verlag, Berlin, Heidelberg, 2007.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[DDVY21] Jintai Ding, Joshua Deaton, Vishakha, and Bo-Yin Yang. The nested subset differential attack - A practical direct attack against LUOV which forges a signature within 210 minutes. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 329–347. Springer, 2021.

[Din21a] Itai Dinur. Cryptanalytic applications of the polynomial method for solving multivariate equation systems over GF(2). In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 374–403. Springer, 2021.

[Din21b] Itai Dinur. Improved algorithms for solving polynomial systems over GF(2) by multiple parity-counting. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2550–2564. SIAM, 2021.

[DRS20] Christoph Dobraunig, Yann Rotella, and Jan Schoone. Algebraic and higher-order differential cryptanalysis of pyjamask-96. *IACR Trans. Symmetric Cryptol.*, 2020(1):289–312, 2020.

[DS05] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In John Ioannidis, Angelos D. Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security, Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, Proceedings*, volume 3531 of *Lecture Notes in Computer Science*, pages 164–175, 2005.

[DS11] Itai Dinur and Adi Shamir. An improved algebraic attack on hamsi-256. In Antoine Joux, editor, *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, volume 6733 of *Lecture Notes in Computer Science*, pages 88–106. Springer, 2011.

[Dvo90] S. Dvořák. Correspondance. *Comput. J.*, 33(2):188, 1990.

[Ehr73] Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. ACM*, 20(3):500–513, 1973.

[FPR19] Jean-Charles Faugère, Ludovic Perret, and Jocelyn Ryckeghem. Software toolkit for hfe-based multivariate schemes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):257–304, 2019.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[FW94] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.

[Int23] *Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1*, 2023.

[Jou09] Antoine Joux. *Algorithmic cryptanalysis*. CRC Press, 2009.

[JV17] Antoine Joux and Vanessa Vitse. A Crossbred Algorithm for Solving Boolean Polynomial Systems. In *NuTMiC*, volume 10737 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2017. https://eprint.iacr.org/2017/372.pdf.

[Knu14] Donald Ervin Knuth. *The art of computer programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley, 2014.

[LPT+17] Daniel Lokshtanov, Ramamohan Paturi, Suguru Tamaki, R. Ryan Williams, and Huacheng Yu. Beating brute force for systems of polynomial equations over finite fields. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2190–2202. SIAM, 2017.

[Mif63] Charles J. Mifsud. Algorithm 154: combination in lexicographical order. *Commun. ACM*, 6(3):103, 1963.

[PWZ96] Marko Petkovsek, Herbert S Wilf, and Doron Zeilberger. $A = B$. A K Peters/CRC Press, 1996.

[Rus03] Frank Ruskey. Combinatorial generation, 2003. unpublished book, available online.

[The23] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.5)*, 2023. `https://www.sagemath.org`.

[VSIH22] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. Memory performance of AMD EPYC rome and intel cascade lake SP server processors. In Dan Feng, Steffen Becker, Nikolas Herbst, and Philipp Leitner, editors, *ICPE '22: ACM/SPEC International Conference on Performance Engineering, Bejing, China, April 9 - 13, 2022*, pages 165–175. ACM, 2022.

[ZZL+21] Juan Zhao, Min Zhu, Xiaoyong Li, Zhenyu Huang, Jincai Li, and Junqiang Song. Solving boolean polynomial systems by parallelizing characteristic set method for cyber-physical systems. *Software: Practice and Experience*, 51(11):2143–2167, 2021.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009