Équipe de logique mathématique, Université Paris 7 - Denis Diderot
strozecki@gmail.com

On enumerating monomials and other combinatorial structures by polynomial interpolation

Yann Strozecki

December 11, 2012

Abstract

We study the problem of generating the monomials of a black box polynomial in the context of enumeration complexity. We present three new randomized algorithms for restricted classes of polynomials with a polynomial or incremental delay, and the same global running time as the classical ones. We introduce **TotalBPP**, **IncBPP** and **DelayBPP**, which are probabilistic counterparts of the most common classes for enumeration problems. Our interpolation algorithms are applied to algebraic representations of several combinatorial enumeration problems, which are so proved to belong to the introduced complexity classes. In particular, the spanning hypertrees of a 3-uniform hypergraph can be enumerated with a polynomial delay. Finally, we study polynomials given by circuits and prove that we can derandomize the interpolation algorithms on classes of bounded-depth circuits. We also prove the hardness of some problems on polynomials of low degree and small circuit complexity, which suggests that our good interpolation algorithm for multilinear polynomials cannot be generalized to degree 2 polynomials. This article is an improved and extended version of [42] and of the third chapter of the author's phd thesis [41].

1 Introduction

1.1 Motivations and connection to previous works

Enumeration, the task of generating all solutions of a given problem, is an interesting generalization of decision and counting. Since a problem typically has an exponential number of solutions, the time to compute them does not seem to be a relevant complexity measure. A first solution is to relate the computing time of an enumeration algorithm to its input *and* output. For instance, there is an algorithm generating the extensions of a partial order in a time linear in the number of extensions [37]. If we consider that an enumeration algorithm

produces solutions in a dynamic fashion, the most interesting complexity measure is the *delay* between two consecutive solutions. One should then design enumeration algorithm with a good – polynomial in the input – delay. There is such an algorithm to enumerate the maximal independent sets of a graph in lexicographic order while strangely there is none for the reverse lexicographic order unless P = NP [23].

One way to obtain an algorithm for an enumeration problem is to represent it by a formula in some well-chosen logic. For instance, most of the time we know how to enumerate the graphs of a given size which satisfy a first-order property [20]. One can also enumerate all satisfying assignments of an existential first order formula with second order free variables with a polynomial delay [14]. There are several other "meta-algorithms" for enumeration of this kind [13, 11] and our aim is here to obtain one based on algebra instead of logic.

The method we propose in this article is to represent an enumeration problem by a family of polynomials rather than by a formula. This method had already a lot of success for decision and search problems. One can associate a well chosen determinant to the perfect matchings of a graph, and thus decide if there is one in randomized parallel logarithmic time [35]. The beautiful polynomial time algorithm to decide if a number is prime also relies on a polynomial representation of the number [1]. In parametrized complexity, there is also a method to solve several combinatorial problems [30] which relies on detecting multilinear monomials of total degree k in polynomials represented by circuits.

This later approach is quite similar to what we present here, since we will also be interested in monomials in multilinear polynomials as representation of the solutions of an enumeration problem. More precisely, we associate to an instance of a problem a multivariate polynomial whose monomials are in bijection with its solutions. We prove that we are able to enumerate the monomials of a polynomial given by some implicit representation with a good delay. Therefore we obtain an enumeration algorithm for all problems which can be represented by a family of polynomials enjoying some structural properties and which can be evaluated in polynomial time.

In other words, we want to solve the problem of *interpolating* a polynomial, with an emphasis on the delay between the production of two monomials. The polynomial is represented by a black box, that is an oracle that can be queried for the value of the polynomial on any point. The black box model is chosen because we do not want to make assumptions on the representation of the polynomial. The interpolation of a dense polynomial of a fixed degree, i.e. with all possible monomials, can be done efficiently through inversion of a Vandermonde matrix. This method is always exponential in the number of variables, but better algorithms (both deterministic and probabilistic) have been designed for a polynomial represented either by a black box or a circuit [8, 49, 25, 19]. Their complexity polynomially depends on the number of monomials of the polynomial which can be much smaller than the potential number of monomials. The deterministic methods rely on evaluating the polynomial on specific prime numbers, with the drawback that these numbers may be very large. The probabilistic methods use the Schwarz-Zippel lemma which describes how to

solve the Polynomial Identity Testing problem, that is deciding whether a polynomial is identically zero, in randomized polynomial time. This lemma is also essential in this article.

It turns out than most interpolation algorithms have a bad delay, in fact they produce all the monomials at the same time after a potentially exponential computation. There is one exception: as a consequence of a result about random efficient identity testing [29], Klivans and Spielman give an interpolation algorithm, which happens to have an incremental delay. In this article, we try to further improve this delay by proposing new interpolation algorithms on restricted classes of polynomials such as multilinear polynomials. Similar restrictions have been studied in other works about Polynomial Identity testing for instance depth 3 circuits [39] and multilinear polynomials defined by a depth 4 circuit [26, 38]. Moreover, the result applies to a lot of interesting polynomials which happen to be multilinear like the Determinant, the Pfaffian, the Permanent, the elementary symmetric polynomials, and others.

1.2 Main results

The main contributions of the paper are as follows:

- We present a randomized algorithm which enumerates the monomials of a black box multilinear polynomial with a *polynomial delay*. Moreover, this algorithm is easily parallelizable and works over fields of small characteristic.
- We give a randomized algorithm with an incremental delay which is better than existing algorithms for polynomials of degree less than 10.
- We introduce the use of polynomials representation and randomization in the study of enumeration problems. It yields algorithms for several combinatorial problems, amongst them the enumeration of the spanning hypertrees of a 3-uniform hypergraph for which, to our knowledge, no algorithm was known.
- We prove a few hardness results for the problem to decide if there is a given monomial (or to compute its coefficient) in a low degree polynomial given by a low depth formula. This shows that our polynomial delay algorithm for multilinear polynomials cannot be generalized to non-multilinear polynomials.

1.3 Organization of the paper

Section 2 is an introduction to enumeration problems and to the basics of polynomial identity testing and interpolation.

In Section 3 we present a simple randomized algorithm which produces a monomial of polynomials satisfying the following property : no two of their monomials use the same set of variables. We then outline a deep result which

enables to recover the monomial of any polynomial [29]. We also explain how we can turn both monomial search procedures into an interpolation algorithm with incremental delay.

In Section 4, the notion of partial degree is defined. We explain how to compute it in randomized polynomial time. From that result, we design an interpolation algorithm for multilinear polynomials with a delay polynomial in the number of variables.

In Section 5, the techniques of the two previous sections are generalized to build an incremental interpolation algorithm for fixed degree polynomials.

We explain in Section 6 how to turn our interpolation algorithms into efficient enumeration procedures for several combinatorial problems. These different procedures serve as an illustration of three probabilistic complexity classes that we introduce, namely **TotalBPP**, **IncBPP** and **DelayBPP**.

In Section 7 we propose several methods to improve the complexity of the previous algorithms. Then using some of these tricks, we explain how to enumerate the spanning hypertrees of a 3-uniform hypergraph with a delay as small as possible.

Finally, in Section 8 we study a different model: polynomials given by a circuit rather than a black box. For those polynomials, we give an alternate method to compute their partial degree. As a consequence we prove that our algorithm interpolating multilinear polynomials can be derandomized on the same class of circuits as the polynomial identity test can. To explain why this algorithm cannot be extended to higher degree polynomials, we prove that several related decision or counting problems are hard even on restricted circuits computing polynomials of degree 2 or 3.

2 Introduction to Enumeration Problems and Interpolation

In this section, we recall basic definitions about enumeration problems and their complexity measures (for further details and examples see [41, 7]). Then we introduce the central problem of this article and a few useful tools.

2.1 Basics of enumeration

The computational model is the random access machine model (RAM) with addition, subtraction and multiplication as its basic arithmetic operations. It has read-only input registers I_1 , I_2 , ... (containing the input x), read-write work registers R_1 , R_2 , ... and output registers O_1 , O_2 , ... Our model is equipped with an additional instruction OUTPUT which, when executed, returns the contents of the non empty output registers. The result of a computation of such a RAM machine is the sequence of words $(w_1, \ldots, w_n)$, where w_i is encoded in the output registers when the i^{th} OUTPUT instruction is executed.

Let M be such a machine and x a word, we write $M(x)$ the result of the computation of M on x. Sometimes we will use $M(x)$ to denote the set of out-

puts although it is a sequence. We note $|M(x)|$ the size of the output sequence, which is equal to the number of OUTPUT instructions executed.

We prefer a RAM machine to a Turing machine since it may be useful to deal with an exponential amount of memory in polynomial time, see for instance the enumeration of the maximal independent sets of a graph [23].

We deal with randomized algorithm in the following, therefore we need to make our RAM machine probabilistic. To this aim we extend the machine with a RAND instruction which writes with probability one half 0 or 1 in a special register. All outcomes of this instruction during a run of a RAM machine are independent. We say that a probabilistic RAM machine computes the word w on the instance x with probability p if p is the number of runs on x for which the machine computes w divided by the total number of runs.

[Enumeration Problem] Let A be a polynomially balanced binary predicate, i.e. $A(x, y) \Rightarrow |y| \leq Q(|x|)$, for a polynomial Q. We write $A(x)$ for the set of y such that $A(x, y)$. We say that a RAM machine M solves the enumeration problem associated to A, ENUM·A for short, if $M(x) = A(x)$ and there is no repetition of solutions in the computation.

Given a RAM machine M and an input x, we note $T(M, x, i)$ the number of instructions executed before the i^{th} OUTPUT. This function is defined for $1 \leq i \leq |M(x)|$ and we extend it at $i = 0$ by 0 and at $i = |M(x)| + 1$ by the number of instructions before M stops. Remark that all arithmetic operations are assumed to be of unit cost regardless to the size of the integers they operate on. The more realistic bit complexity model takes into account the complexity of such operations. However, the complexity of the algorithms presented in this article is very similar in both models since we are cautious to never work with large integers.

We are interested in two complexity measures, the first one is very similar to what is used for decision problems: Let M be a RAM machine, and let x be a word. The *total time* taken by M on x is $T(M, x, |M(x)| + 1)$. We say that the machine M has a total time $f(n)$ if $\max_{|x|=n} T(M, x, |M(x)| + 1) = O(f(n))$.

The next measure is specific to enumeration and its study is the main goal of enumeration complexity.

Let M be a RAM machine, let x be a word and i be an integer. The *delay* between the i^{th} and the $i + 1^{\text{th}}$ OUTPUT is the quantity $T(M, x, i + 1) - T(M, x, i)$, when it is defined. We say that the machine M has a delay $f(n)$ if $\max_{|x|=n} \max_{i \leq |M(x)|} T(M, x, i + 1) - T(M, x, i) = O(f(n))$.

The delay between i^{th} and the $i + 1^{\text{th}}$ solution may depend on both $|x|$, the size of the input, and i, the number of already generated solutions. When the delay is bounded by a polynomial in $|x|$ and i, we say that the algorithm has an *incremental* delay. If the delay is bounded by a polynomial in $|x|$ only, we say that the algorithm has a *polynomial delay*. Polynomial delay is often hard to obtain, but better –though not yet proved to be strictly better– than incremental delay.

In this article we interpret the problem of interpolating a polynomial given by a black box as an enumeration problem. It means that we try to list all the

monomials of a polynomial given by the number of its variables and an oracle. We denote this problem by Enum·Poly.

The oracle allows to evaluate the polynomial on any point in unit time. To formalize computation with an oracle, let us introduce the *RAM machine with oracle*. Such a machine has a sequence of new registers $\mathrm{O}(0), \mathrm{O}(1), \ldots$ and a new instruction Oracle. The semantic of the machine depends on the oracle, which is a function f from $\{0,1\}^*$ to $\{0,1\}^*$. We call w the word which is encoded in the register $\mathrm{O}(0), \mathrm{O}(1), \ldots$. When the instruction Oracle is executed, the machine writes in the work register the value of $f(w)$. In our settings, the function f is the input polynomial.

The complexity measures of this particular model, in addition to the total time and delay are the number of calls to the oracle and the size of the words which are given as arguments to the oracle. One usually uses this model to work with multivariate polynomials since it helps give lower bounds and it abstracts away the operation of evaluating the polynomial.

2.2 Black box Polynomial Identity Testing and Dense Interpolation

Let introduce all the basic tools and notations we need to build interpolation algorithms. We study multivariate polynomials in $\mathbb{Q}[X_1, \ldots, X_n]$. A *term* $\vec{X}^{\vec{e}} = X_1^{e_1} X_2^{e_2} \ldots X_n^{e_n}$ is a product of variables, it is defined by the sequence of n positive integers $\vec{e} = (e_1, \ldots, e_n)$. A *monomial* is a term $\vec{X}^{\vec{e}}$ multiplied by some constant λ, it is defined by the pair $(\lambda, \vec{e})$. We call t the number of monomials of a polynomial P written $P(\vec{X}) = \sum_{1 \leq j \leq t} \lambda_j \vec{X}^{\vec{e_j}}$.

The degree of a monomial is the maximum of the degrees of its variables and the total degree is the sum of the degrees of its variables. Let d (respectively D) denote the degree (respectively the total degree) of the polynomial we consider, that is to say the maximum of the degrees (respectively total degrees) of its monomials. In Section 4 we assume that the polynomial is multilinear, i.e. $d = 1$ and D is thus bounded by n. Since we are interested in algorithms for low degree polynomials, we always assume that the total degree D is at most polynomial in n. In fact, most of the time d is considered to be a fixed constant and therefore D is in $O(n)$.

When analyzing the delay of an algorithm solving Enum·Poly, we are interested in both the number of calls to the black box and the time spent between two generated monomials. We are also interested in the size of the integers used in the calls to the oracle, since in some applications the complexity of the evaluation depends on it.

The support of a monomial is the set of variables which appear in the monomial.

Let L be a set of variables, then f_L is the homomorphism of $\mathbb{Q}[X_1, \ldots, X_n]$ defined by:

$$\begin{cases} f_L(X_i) = X_i & \text{if } X_i \in L \\ f_L(X_i) = 0 & \text{otherwise} \end{cases}$$

From now on, we denote $f_L(P)$ by P_L. It is the polynomial obtained by substituting 0 to every variable of index not in L, that is to say the sum of the monomials of P which have their support in L. Let $\vec{X}^L$ be the multilinear term of support L, it is the product of all X_i in L.

Let P be the polynomial $2X_1^2 - X_4X_5X_6^3 + 3X_1X_3X_7 + X_2^2X_1$. The support of $3X_1X_3X_7$ is $L = \{X_1, X_3, X_7\}$ and $P_L = 2X_1^2 + 3X_1X_3X_7$.

In this article, we often need to decide whether a polynomial given by a black box is the zero polynomial. This problem is called Polynomial Identity Testing or PIT for short. Remark that PIT can be seen as the decision version of the interpolation problem we try to solve. For any t evaluation points, one may build a polynomial with t monomials which vanishes at every point (see [49]). Therefore, if we do not have any a priori bound on t, we must evaluate the polynomial on at least $(d+1)^n$ n-tuples of integers to decide PIT. Since such an exponential complexity is not satisfying, we use a probabilistic algorithm, which has a good complexity and an error bound which can be made exponentially small.

[Schwarz-Zippel [40, 12, 48]] Let P be a non zero polynomial with n variables of total degree D, if $x_1, \ldots, x_n$ are randomly chosen values in a set of integers S of size $\frac{D}{\epsilon}$ then the probability that $P(x_1, \ldots, x_n) = 0$ is bounded by ϵ.

The following classical algorithm to decide PIT over polynomials of known total degree is derived from this lemma. We denote by $[x]$ the set of integers $\{1, \ldots, \lceil x \rceil\}$.

Algorithm 1: The procedure *not_zero*(P, ϵ)

Data: A polynomial P with n variables, its total degree D and the error bound ϵ

begin
  pick $x_1, \ldots, x_n$ randomly in $[\frac{D}{\epsilon}]$
  if $P(x_1, \ldots, x_n) = 0$ **then**
    return False
  else
    return True

Remark that the algorithm never gives a false answer when the polynomial is zero. The probability of error when the polynomial is not zero is bounded by ϵ thanks to Lemma 1. in fact the problem PIT for polynomials given by circuits is in coRP [21].

Algorithm 1 makes exactly one call to the black box on integers of size $\log(D\epsilon^{-1})$. The error rate may then be made exponentially smaller by increasing the size of the integers. There is an other way to achieve the same reduction of error. Repeat the previous algorithm k times for $\epsilon = \frac{1}{2}$, that is to say the integers are randomly chosen in $[2D]$. If all runs return zero, then the algorithm decides that the polynomial is zero else it decides it is not zero. Since the random choices are independent in each run, the probability of error of this algorithm is bounded by 2^{-k}. Hence, to achieve an error bound of ϵ, we have to

set $k = \log(\epsilon^{-1})$. The procedure that we use in this article and that we denote by $not\_zero(P, \epsilon)$ is this latter variant. It uses slightly more random bits but it only involves numbers less than $2D$ which is especially convenient if we want to use the algorithm on a field with a small number of elements.

Dense interpolation It is well known that a univariate polynomial P of total degree less than D is totally defined by its values in D points. There are a lot of methods to interpolate P from D points, the simplest being Lagrange interpolation:

$$P(X) = \sum_{i=1}^{D} P(i) \prod_{j \leq D,\, j \neq i} \frac{X - j}{i - j}$$

The naive expansion of this formula to write P as a sum of monomials requires D^3 operations. It uses only integers as evaluation points, while more efficient methods relying on Fast Fourier transform use unit roots as evaluation points.

In fact interpolating P is equivalent to solving a linear system where the variables are the coefficients of P. Each linear equation is given by the evaluation of the polynomial on a different point. The system forms a Vandermonde matrix, therefore if the evaluation points have been chosen different (say we choose $1, \ldots, D$) it is invertible. Moreover, the structure of the matrix enables us to find its inverse in $O(D^2)$ operations.

[Section 3 of [49]] Let P be a black box univariate polynomial of total degree less than D. It can be interpolated in D calls to the black box on integers of size $O(\log(D))$ and $O(D^2)$ arithmetic operations.

The same method can be used to interpolate multivariate polynomials, see also [49]. The idea is to set the variables of the polynomial to different powers of the first prime numbers. If the interpolated polynomial is of degree d and has n variables, it has at most $(d+1)^n$ monomial. The polynomial can be recovered by inverting a $(d+1)^n \times (d+1)^n$ Vandermonde matrix.

All the mentioned interpolation methods have a complexity which depends on the maximal number of monomials of the polynomial. These methods are said to be dense, since they work well for polynomials with all monomials.

3 Incremental delay algorithm to interpolate a polynomial

In this section, we explain how to produce one monomial of a black box polynomial. We give an efficient algorithm of our own for a restricted class of polynomials and we recall the algorithm of [29] which works for any polynomial. We then explain how an algorithm which produces one monomial can be turned into an interpolation algorithm with an incremental delay.

3.1 Monomials with distinct supports

We define a natural extension of the multilinear polynomials which is exactly the class of polynomials on which the next algorithm is correct.

Let $\mathcal{DS}$ be the set of polynomials with no constant term and whose monomials have distinct supports.

Remark that being without constant term is not restrictive. We can indeed compute the constant term of a polynomial P by a single oracle call to $P(0,\ldots,0)$. Then we can simulate an oracle call to the polynomial $P - P(0,\ldots,0)$, which is equal to P without its constant term, with an overhead of only one arithmetic operation.

Let P be a polynomial in $\mathcal{DS}$ and let L be a set of variables minimal for inclusion such that P_L is not identically zero. Then P_L is a single monomial of support L.

Using Lemma 2 and the procedure *not_zero* which solves the PIT problem, we give an algorithm which finds, in randomized polynomial time, a monomial of a polynomial P. Let L be a set of variables and let i be an integer used to denote the index of the current variable. In the first step of the next algorithm we build a set of variables L satisfying the hypothesis of Lemma 2 by trying to set each X_i to 0.

Then once we have found L such that P_L is a monomial $\lambda\vec{X}^{\vec{e}}$, we compute λ and $\vec{e}$. The evaluation of P_L on $(1,\ldots,1)$ returns λ. For each $X_i \in L$, the evaluation of P_L on $X_i = 2$ and for $j \neq i$, $X_j = 1$ returns $\lambda 2^{e_i}$. From these n calls to the black box, one can find $\vec{e}$ in linear time and thus output $\lambda\vec{X}^{\vec{e}}$.

Algorithm 2: The procedure *find_monomial*(P,ϵ)

```
Data: A polynomial P with n variables, its total degree D and the error
      bound ε
Result: A monomial of P
begin
    L ⟵ {1,...,n}
    if not_zero(P, ε/(n+1)) then
        for i = 1 to n do
            if not_zero(P_{L\{i}}, ε/(n+1)) then
                L ⟵ L \ {i}
        return The monomial of support L
    else
        return "Zero"
```

We give the complexity of this algorithm in the next proposition.

Given a black box polynomial P in $\mathcal{DS}$, Algorithm 2 finds, with probability $1-\epsilon$, a monomial of P by making $O(n\log(\frac{n}{\epsilon}))$ calls to the black box on entries of size $\log(2D)$.

We analyze this algorithm, assuming first that the procedure *not_zero* never

returns a wrong answer and that P is not the zero polynomial. In this case, the algorithm does not answer "Zero" at the beginning. Therefore P_L is not zero at the end of the algorithm, because an element is removed from L only if this condition is respected. Since removing any other element from L would make P_L zero by construction, the set L is minimal for the property of P_L being non zero. Then by Lemma 2 we know that P_L is a monomial of P, which allows us to output it as previously explained.

The only source of error is the subroutine *not_zero* which fails with probability $\frac{\epsilon}{n+1}$. Since this subroutine is called $n+1$ times we can bound the total probability of error by ϵ. The total complexity of this algorithm is $O(n \log(\frac{n}{\epsilon}))$ since each of the n invocations of *not_zero* makes $O(\log(\frac{n}{\epsilon}))$ calls to the oracle in time $O(1)$.

3.2 Klivans and Spielman algorithm

Here we briefly explain an elaborate result to solve PIT on black box polynomials with few random bits [29], which can be used to interpolate sparse polynomials. The key of the method is a clever transformation of a multivariate polynomial into a univariate polynomial.

[Theorem 5 of [29]] There exists a randomized polynomial time algorithm which maps the zero polynomial to itself and any non zero polynomial P with n variables and total degree at most D to a non zero univariate polynomial P' of degree at most $D^4 n^6$ with probability $\frac{2}{3}$.

The idea of the proof is to map a variable X_i to h^{z_i} where h is a new variable and the z_i's are well chosen linear forms. Each monomial of P is thus mapped in P' to h to the power of a sum of the linear forms z_i. A generalized isolation lemma (Lemma 4 in [29]) proves that among these sums of linear forms evaluated at random points, there is one which is minimum with high probability.

Therefore the monomial of lowest degree in P' comes with probability $\frac{2}{3}$ from a unique monomial of P, denoted by $\lambda \vec{X}^{\vec{e}}$. The polynomial P' is interpolated in polynomial time in order to find λh^l the lowest degree monomial. This gives λ but one still needs to recover $\vec{e}$.

Let P'' be a new polynomial, where $p_i h^{z_i}$ is substituted to X_i and p_i is the i^{th} prime. The linear form z_i is evaluated at the same points as in the previous step, therefore the lowest degree monomial of P'' is $\lambda \prod p_i^{e_i} h^{z_i}$. Finally P'' is interpolated to recover $\vec{e}$ and the monomial $\lambda \vec{X}^{\vec{e}}$ is returned.

[Adapted from Theorem 12 of [29]] There is a randomized polynomial time algorithm which given a black box access to a non zero polynomial P with n variables and total degree D returns a monomial of P with probability $\frac{2}{3}$ in a time polynomial in n, D and with $O(n^6 D^4)$ calls to the black box on integers of size $\log(\frac{nD}{\epsilon})$.

3.3 From a monomial to the polynomial: an incremental delay algorithm

We build an algorithm which enumerates the monomials of a polynomial incrementally once we know how to produce one of its monomials in polynomial time. The idea is to subtract the monomial to the polynomial and recurse. The procedure *find_monomial* defined in Proposition 1 is used to find the monomial.

We also need a procedure *subtract*(P, Q) which acts as a black box for the polynomial $P - Q$ when P is given as a black box and Q as an explicit set of monomials. Let D be the total degree of Q and i is the number of its monomials. One evaluates the polynomial *subtract*(P, Q) on an evaluation point as follows:

1. compute the value of each monomial of Q in $O(D)$ arithmetic operations
2. add the value of each monomial to an accumulator, the sum is obtained in $O(iD)$ operations
3. call the black box to compute P on the same points and return this value minus the one we have computed for Q

The integers used in the computation of Q may be large, which would make the cost of a call to *subtract* even worse in a bit complexity model. However, the complexity of *subtract* could be improved by using a less naive way to evaluate the explicit polynomial Q, like fast or iterated multiplication. One other possible improvement would be to do all computations modulo some small random prime. This idea is developed in Section 7.

Algorithm 3: Incremental computation of the monomials of P

Data: A polynomial P with n variables and the error bound ϵ
Result: The set of monomials of P
begin
  $Q \longleftarrow 0$
  while not_zero(subtract*(P,Q)*, $\frac{\epsilon}{2^{n+1}}$) **do**
    $M \longleftarrow$ find_monomial(subtract(P,Q), $\frac{\epsilon}{2^{n+1}}$)
    Output(M)
    $Q \longleftarrow Q + M$

Let P be a polynomial in $\mathcal{DS}$ with n variables, t monomials and a total degree D. Algorithm 3 computes the set of monomials of P with probability $1 - \epsilon$. The delay between the i^{th} and $i + 1^{\text{th}}$ produced monomial is bounded by $O(iDn(n + \log(\epsilon^{-1})))$ in time and $O(n(n + \log(\epsilon^{-1})))$ calls to the oracle. The whole algorithm performs $O(tn(n + \log(\epsilon^{-1})))$ calls to the oracle on integers of size $\log(2D)$.

Correctness. We analyze Algorithm 3 under the assumption that the procedures *not_zero* and *find_monomial* never return a wrong answer.

The following is an invariant of the while loop: Q is a subset of the monomials of P. It is true at the beginning because Q is the zero polynomial. Assume that Q satisfies this property at a certain point of the while loop. Since *not_zero*(*subtract*(P,Q)) is true, the polynomial $P - Q$ is not zero and is then a non empty subset of the monomials of P. The outcome of *find_monomial*(*subtract*(P,Q)) is thus a monomial of P which is not in Q, therefore Q plus this monomial still satisfies the invariant. Remark that we have also proved that the number of monomials of Q is increasing by one at each step of the while loop. The algorithm must then terminate after t steps and when it does *not_zero*(*subtract*(P,Q)) gives a negative answer meaning that $Q = P$.

Probability of error. The probability of failure is bounded by the sum of the probabilities of error coming from *not_zero* and *find_monomial*. These procedures are both called t times with an error bounded by $\frac{\epsilon}{2^{n+1}}$. There are 2^n different supports, hence there is at most 2^n monomials in P. The total probability of error is bounded by $\frac{2t}{2^{n+1}}\epsilon \leq \epsilon$.

Complexity. The subroutine *not_zero* is called t times and it itself calls the oracle $n + \log(\epsilon^{-1})$ times. The subroutine *find_monomial* is called t times and calls the black box $n(n+\log(\epsilon^{-1}))$ times. In total, we have $t(n+1)(n+\log(\epsilon^{-1}))$ calls to the the black box on evaluation points of size $\log(2D)$.

The delay between two solutions is the time to execute *find_monomial*. It is dominated by the execution of the subroutine *subtract*(P, Q) at each oracle call. By Proposition 1 *find_monomial* does $n(n + \log(\epsilon^{-1}))$ calls to the oracle. The delay between the i^{th} and $i + 1^{\text{th}}$ produced monomial is $O(iDn(n + \log(\epsilon^{-1})))$ since *subtract*(P, Q) does $O(iD)$ arithmetic operations.

Remark that Theorem 3 can be used to implement *find_monomial* in Algorithm 4. It yields an incremental interpolation algorithm for any polynomial but with a worse delay over polynomials in $\mathcal{DS}$.

4 A Polynomial Delay Algorithm for Multilinear Polynomials

In this section we first solve the problem of finding the degree of a polynomial with regard to a set of variables.

Let n be an integer and $L \subseteq \{X_1, \ldots, X_n\}$. The degree of the term $\vec{X}^{\vec{e}}$ with regard to L is the integer $\sum_{X_i \in L} e_i$ and it is denoted by $d_L(\vec{X}^{\vec{e}})$. We write $d_L(P)$ for the degree of a polynomial P with regard to S, it is the maximum of the degree of its monomials with regard to L.

Remark that $d_{\{X_i\}}(P)$ is the degree of X_i in P, while $d_{\{X_1,\ldots,X_n\}}(P)$ is the total degree of P. We present a method to efficiently compute the degree of a polynomial with regard to any set of variables. It transforms the multivariate polynomial into a univariate one, which is then interpolated. To achieve this, one uses a polynomial number of calls to the black box on small integers. As a corollary, an efficient algorithm for the following problem is given, when the polynomial is multilinear.

MONOMIAL-FACTOR
Input: a polynomial given as a black box and a term $\vec{X}^{\vec{e}}$
Output: accept if $\vec{X}^{\vec{e}}$ divides a monomial in the polynomial

We also give a second algorithm, which solves this problem with only one call to the black box but using exponentially larger integers. We then design an algorithm which enumerates the monomials of a multilinear polynomial with a polynomial delay. This algorithm has the interesting property of being easily parallelizable, which is obviously not the case of the incremental one.

4.1 Small evaluation points

Let $P(\vec{X})$ be a n variables polynomial over $\mathbb{Q}$, and let (L_1, L_2) be a partition of $\{X_1, \ldots, X_n\}$. We can see P as a polynomial over the ring $\mathbb{Q}[\{X_i \in L_2\}]$ with variables $X_j \in L_1$. In fact, the total degree of P as a polynomial over this ring is equal to $d_{L_1}(P)$. We introduce a new variable Y, the polynomial $\tilde{P}$ is the polynomial P where YX_i is substituted to X_i when $X_i \in L_1$. We have the equality $d_{\{Y\}}(\tilde{P}) = d_{L_1}(P)$.

Let $P(\vec{X})$ be a non zero polynomial with n variables, a total degree D and let L be a subset of $\{X_1, \ldots, X_n\}$. There is an algorithm which computes $d_L(P)$ with probability greater than $\frac{2}{3}$ in time polynomial in n and D.

We define the polynomial $\tilde{P}$ with $n+1$ variables from P and L, as explained previously. We write it $\sum_{i=0}^{d} Y^i Q_i(\vec{X})$ where d is the degree of $\tilde{P}$ seen as a univariate polynomial over $\mathbb{Q}[X_1, \ldots, X_n]$. Recall that $d = d_L(P)$, thus we want to compute the degree of $\tilde{P}$.

Now choose randomly in $[3D]$ a value x_i for each X_i. The polynomial $\tilde{P}(x_1, \ldots, x_n, Y)$ is a univariate polynomial, and the coefficient of Y^d is $Q_d(\vec{x})$. By Lemma 1, the probability that $Q_d(\vec{x})$ is zero is bounded by $\frac{1}{3}$ since Q_d is a non zero polynomial.

By Theorem 1, the polynomial $\tilde{P}(x_1, \ldots, x_n, Y)$ can be interpolated from its value on the integers $1, \ldots, D$ with $O(D^2)$ arithmetic operations, because it is of degree less or equal to $d_L(P) \leq D$. The value of $\tilde{P}(x_1, \ldots, x_n, y)$ can be computed from an oracle call to P, since it is equal to $P(x'_1, \ldots, x'_n)$ where $x'_i = yx_i$ if $X_i \in L$ and x_i otherwise. From the interpolated polynomial we obtain its degree which is equal to $d_L(P)$ with probability greater than $\frac{2}{3}$.

We now give a solution to the problem MONOMIAL-FACTOR for terms of the form $\vec{X}^L$ in a multilinear polynomial as a corollary. In fact to obtain a better complexity, we do not use directly Proposition 2, but rather the idea of its proof.

Let $P(\vec{X})$ be a multilinear polynomial with n variables, a total degree D and let L be a subset of $\{X_1, \ldots, X_n\}$. There is an algorithm which solves the problem MONOMIAL-FACTOR on the polynomial P and the term $\vec{X}^L$ with probability $1 - \epsilon$. It does $|L| \log(\epsilon)$ calls to the black box on points of size $\log(|L|)$ and $O(\log(\epsilon)|L|^2)$ arithmetic operations. The polynomial $P(\vec{X})$ can

be written $\vec{X}^L P_1(\vec{X}) + P_2(\vec{X})$, where $\vec{X}^L$ does not divide $P_2(\vec{X})$. Since P is multilinear, one of its monomials is divided by $\vec{X}^L$ if and only if $d_L(P) = |L|$.

Let substitute in P the new variable Y to all $X_i \in L$. Remark that P_1 does not depend on variables in L, since $\vec{X}^L P_1$ is multilinear. Therefore P_1 is not zero after the substitution if and only if it was not zero before substitution.

Now let substitute a random value in $[2D]$ to each $X_i \notin L$ and interpolate the obtained univariate polynomial. The bound on the total degree of the polynomial is $|L|$, therefore we need $|L|$ calls to the black box on points of size $\log(|L|)$ and $O(|L|^2)$ arithmetic operations.

Finally to bring the probability of error from $\frac{1}{2}$ down to ϵ the procedure is repeated $\log(\epsilon)$ times.

4.2 One large evaluation point

We prove here a proposition similar to Corollary 1. If needed, we could adapt it to give an algorithm which decides the degree of a polynomial with regard to a set.

Note that in this case we need an a priori bound on the coefficients of the polynomial, but it is not so demanding since in most applications those coefficients are bounded by a constant. We also assume that the coefficients are in $\mathbb{Z}$ to simplify the proof.

Let $P(\vec{X})$ be a multilinear polynomial with n variables, a total degree D and let L be a subset of $\{X_1, \ldots, X_n\}$. There is an algorithm which solves the problem Monomial-Factor on the polynomial P and the term $\vec{X}^L$ with probability $1 - \epsilon$. It does one call to the black box on a point of size $O(n + D\log(\frac{D}{\epsilon}))$.

We write $P = \vec{X}^L P_1(\vec{X}) + P_2(\vec{X})$ where $\vec{X}^L$ does not divide $P_2(\vec{X})$ and we want to decide if $P_1(\vec{X})$ is the zero polynomial. Let C be a bound on the size of the coefficients of P. We let α be the integer $2^{2(C+n+D\log(\frac{2D}{\epsilon}))}$. The black box is called to compute $P(x_1, \ldots, x_n)$ where $(x_i)_{i\in[n]}$ are defined as follows:

$$\begin{cases} x_i = \alpha & \text{if } X_i \in L \\ x_i \text{ is randomly chosen in } [\frac{2D}{\epsilon}] & \text{otherwise} \end{cases}$$

The value of a variable not in L is bounded by $\frac{2D}{\epsilon}$, therefore a monomial of P_2 (which contains at most $|L| - 1$ variables in L) has its contribution to $P(x_1, \ldots, x_n)$ bounded by $2^C(\frac{2D}{\epsilon})^D\alpha^{|L|-1}$. Since P_2 has at most 2^n monomials, its total contribution is bounded in absolute value by $2^{n+C+D\log(\frac{2D}{\epsilon})}\alpha^{|L|-1}$ which is equal to $\alpha^{|L|-\frac{1}{2}}$. If $P_1(x_1, \ldots, x_n)$ is zero, this also bounds the absolute value of $P(x_1, \ldots, x_n)$.

Assume now that $P_1(x_1, \ldots, x_n)$ is not zero, it is at least 1 since it is defined on $\mathbb{Z}$ and evaluated on integers. Moreover $\vec{x}^L$ is equal to $\alpha^{|L|}$, thus the absolute value of $\vec{x}^L P_1(x_1, \ldots, x_n)$ has $\alpha^{|L|}$ for lower bound. By the triangle inequality

$$\begin{aligned} |P(x_1, \ldots, x_n)| &> \big||\vec{x}^L P_1(x_1, \ldots, x_n)| - |P_2(x_1, \ldots, x_n)|\big| \\ |P(x_1, \ldots, x_n)| &> \alpha^{|L|} - \alpha^{|L|-\frac{1}{2}} > \alpha^{|L|-\frac{1}{2}} \end{aligned}$$

We can then decide if $P_1(x_1, \ldots, x_n)$ is zero by comparison of $P(x_1, \ldots, x_n)$ to $\alpha^{|L|-\frac{1}{2}}$. Remark that $P_1(x_1, \ldots, x_n)$ may be zero even if P_1 is not zero. Nonetheless P_1 only depends on variables which are not in L and are thus randomly taken in $[\frac{2D}{\epsilon}]$. By Lemma 1, the probability that the polynomial P_1 is not zero although $P_1(x_1, \ldots, x_n)$ has value zero is bounded by ϵ.

4.3 The algorithm

Let P be a multilinear polynomial with n variables and a total degree D. Let L_1 and L_2 be two disjoint sets of variables, we want to determine if there is a monomial of P, whose support contains L_2 and is contained in $L_1 \cup L_2$.

Let us consider the polynomial $P_{L_1 \cup L_2}$, its monomials are the monomials of P such that their supports are included in $L_1 \cup L_2$. Obviously P has a monomial whose support contains L_2 and is contained in $L_1 \cup L_2$ if and only if $(P_{L_1 \cup L_2}, \vec{X}^{L_2}) \in$ Monomial-Factor. Let us call *linear_factor*(L_1, L_2, P, ϵ) the algorithm given by Corollary 1, which solves this question in polynomial time with probability $1 - \epsilon$.

We now describe a binary tree such that there is a bijection between the leaves of the tree and the monomials of P. The nodes of this tree are pairs of sets (L_1, L_2) such that there exists a monomial of support L in P with $L_2 \subseteq L \subseteq L_1 \cup L_2$. Consider a node labeled by (L_1, L_2), we note i the smallest element of L_1, it has for left child $(L_1 \setminus \{X_i\}, L_2)$ and for right child $(L_1 \setminus \{X_i\}, L_2 \cup \{X_i\})$ if they exist. The root of this tree is $(\{X_1, \ldots, X_n\}, \emptyset)$ and the leaves are of the form $(\emptyset, L_2)$. A leaf $(\emptyset, L_2)$ is in bijection with the monomial of support L_2.

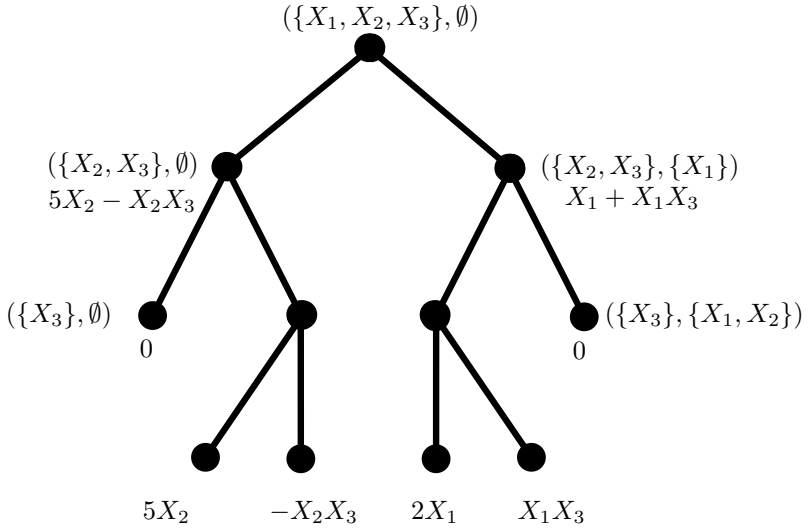

Figure 1: The tree for $P = 5X_3 - X_2X_3 + 2X_1 + X_1X_3$

To enumerate the monomials of P, Algorithm 4 does a depth first search in

this tree using *linear_factor*. When it visits a leaf, it outputs the corresponding monomial thanks to the procedure *coefficient*(P, L) that we now describe. Assume we have found L the support of a monomial of P, we want to find its coefficient. The same procedure as in Corollary 1 is followed (substitution, interpolation) and the coefficient of the monomial of the highest degree is outputed. Indeed, after the substitution of Corollary 1, the obtained univariate polynomial $\tilde{P_L}$ has a monomial of degree $|L|$ which is the image of the monomial of support L in P and has thus the same coefficient.

Algorithm 4: A depth first search of the support of monomials of P (recursive description)

Data: A multilinear polynomial P with n variables and the error bound ϵ
Result: All monomials of P
begin
  Monomial$(L_1, L_2, i) =$
  if $i = n + 1$ **then**
    Output(coefficient(P, L_2),L_2)
  else
    if linear_factor$(L_1 \setminus \{X_i\}, L_2, P, \frac{\epsilon}{2^n n})$ **then**
      monomial$(L_1 \setminus \{X_i\}, L_2, i + 1)$
    if linear_factor$(L_1 \setminus \{X_i\}, L_2 \cup \{X_i\}, P, \frac{\epsilon}{2^n n})$ **then**
      monomial$(L_1 \setminus \{X_i\}, L_2 \cup \{X_i\}, i + 1)$
  in monomial$(\{X_1, \ldots, X_n\}, \varnothing, 1)$

Let P be a multilinear black box polynomial with n variables, t monomials and a total degree D. Algorithm 4 computes the set of monomials of P with probability $1 - \epsilon$. The delay between the i^{th} and $i + 1^{\text{th}}$ monomials is bounded by $O(D^2 n(n + \log(\epsilon^{-1})))$ in time and by $O(nD(n + \log(\epsilon^{-1})))$ oracle calls. The whole algorithm performs $O(tnD(n + \log(\epsilon^{-1})))$ calls to the oracle on points of size $O(\log(D))$.

Between the visit of two leaves, the subroutine *linear_factor* is called at most n times. The procedure *coefficient* is called only once and has the same complexity than *linear_factor* thus we neglect it. By Corollary 1, one call to *linear_factor* on a term of support of size at most D with an error parameter $\frac{\epsilon}{n2^n}$ has a $O(D^2(n + \log(\epsilon^{-1})))$ time complexity and does $O(D(n + \log(\epsilon^{-1})))$ calls to the oracle on points of size $\log(D)$.

Since we call the procedures *linear_factor* and *coefficient* less than nt times during the whole algorithm, the error is bounded by $nt\frac{\epsilon}{n2^n} \leq \epsilon$.

Recall that *linear_factor* and *coefficient* can be implemented thanks to Proposition 3 when a bound on the size of the coefficients of the polynomial is known. The number of calls in the algorithm is then less than tn which is close to the optimal $2t$. We have a trade-off: we can either do several calls to the black box on small points or one on a large point. There are potentially more efficient ways to implement *linear_factor* when the polynomial is given by a circuit, see

Section 8.

Algorithm 4 has two features which makes it appropriate for real implementation. First it uses a space quadratic in the *number of variables* (and linear when the field is of fixed size), while the other algorithms we present need a space linear in the *output*. Second it happens to be paralellizable. Indeed, it is easy to distribute the computation to different RAM machines while doing the traversal of the tree. It is a noteworthy feature in enumeration algorithms, since their total running time is potentially exponential and can thus be distributed to as many RAM machines.

A polynomial is said to be monotone when its coefficients are either all positive or all negative. One can solve PIT without randomness for these polynomials: one oracle call on strictly positive points returns a non zero result if and only if the polynomial is not identically zero. Algorithms 3 and 4 may then be modified to work deterministically for monotone polynomials. The term $(n + \log(\epsilon^{-1}))$ in the time complexity and number of calls of both algorithms disappears, since there are no more repetitions of the procedures *not_zero* or *linear_factor* to exponentially decrease the error. Derandomization of the polynomial delay algorithm on classes of polynomials represented by circuits can be obtained, see Section 8.

5 Interpolating polynomials of small degree

In this section is presented yet another algorithm which produces one monomial of a polynomial. It runs on any polynomial and performs $O(nD^d)$ calls to the oracle. Since its dependency in the degree is exponential it is useful to interpolate a family of polynomials of fixed degree only. Nonetheless, over polynomials of degree $d < 10$, the algorithm performs less than $O(n^6D^4)$ calls which is the number needed by the method of Klivans and Spielman outlined in Section 3. The algorithm is based on a generalization of the ideas used to design *linear_factor* and Algorithm 3.

Let P be a polynomial with n variables of degree d and of total degree D. There is an algorithm, which returns a set L of cardinal l, maximal such that $(\vec{X}^L)^d$ divides a monomial of P with probability greater than $1 - \epsilon$. It uses $ldn \log(n\epsilon^{-1})$ calls to the oracle on points of size less than $\log(D)$. By the method of Corollary 1, we can solve MONOMIAL-FACTOR for the polynomial P and the term $(\vec{X}^L)^d$: We substitute to all variables in L a new variable Y and to the other variables random values in $[2D]$. To solve the problem, we test wether the degree of the obtained univariate polynomial is ld. The interpolation procedure calls the black box on integers in $[ld]$ and $[2D]$, that is of size $O(\log(D))$ since $ld \leq D$. To bring the probability of error down to ϵ, we repeat the preceding steps $\log(\epsilon^{-1})$ times and return the degree computed in most of the runs. We call the procedure solving this problem *exist_monomial*(P,L,ϵ).

Algorithm 5 finds a set L maximal for the property that $(\vec{X}^L)^d$ divides a monomial of P. It works in the same way Algorithm 2 does. In total, it does $ldn \log(n\epsilon^{-1})$ calls to the oracle and the randomly chosen points are of size

Algorithm 5: The procedure $max\_monomial(P, \epsilon)$

Data: A polynomial P with n variables, degree d and the error bound ϵ
Result: A list of variables L
begin
  $L \longleftarrow \varnothing$
  for $i = 1$ **to** n **do**
    if exist_monomial$(P, L \cup \{X_i\}, \frac{\epsilon}{n+1})$ **then**
      $L \longleftarrow L \cup \{X_i\}$
  return L

$\log(D)$.

One run of *max_monomial* returns the set L such that $P = (X^L)^d P_1 + P_2$ and $(X^L)^d$ does not divide any monomial of P_2. Since P is of degree d and that L is maximal for the property that $(X^L)^d$ divides a monomial of P, P_1 is of degree $d-1$ at most. We give a way to evaluate P_1 in the following proposition. We can then apply *max_monomial* recursively and eventually find a monomial of P.

Let P be a polynomial with n variables of degree d. Assume that $P = (X^L)^d P_1 + P_2$ where P_1 is not zero and P_2 is of degree less than d. There is an algorithm, denoted by $restriction(P, L)$, which acts as a black box computing P_1. To do one evaluation of P_1 on points of size less than s, it does ld calls to P on points of size less than $\max(s, \log(ld))$. W.l.o.g. say that L is the set $[l]$, the polynomial P_1 depends only on the variables $X_{l+1}, \ldots, X_n$. We have to compute $P_1(x_{l+1}, \ldots, x_n)$. Let $H(Y)$ be the polynomial P where Y is substituted to X_i if $i \in [l]$, otherwise $X_i = x_i$. We have $H(Y) = Y^{ld} P_1(\vec{x}) + P_2(\vec{x}, Y)$ where $P_2(\vec{x}, Y)$ is a univariate polynomial of degree less than ld. The coefficient of Y^{ld} in $H(Y)$ is equal to the evaluation of P_1 on the desired values. To compute it, one has only to interpolate $H(Y)$, with ld calls to the oracle on points of size bounded by $\log(ld)$ and s.

Algorithm 6: A recursive algorithm finding one monomial of P

Data: A polynomial P with n variables, a degree d and the error bound ϵ
Result: A monomial of P
begin
  Monomial(Q, i) =
  if $i = 0$ **then**
    Return$(Q(0))$
  else
    $L \leftarrow max\_monomial(Q, \frac{\epsilon}{n})$;
    Return L;
    Monomial$(restriction(Q, L), i-1)$;
  in Monomial(P, d) ;

Let P be a polynomial with n variables of degree d and of total degree D. Algorithm 6 returns the sets $L_d, \ldots, L_1$ and the integer λ such that $\lambda \prod_{i=1}^{d} (\vec{X}^{L_i})^i$ is a monomial of P, with probability $1 - \epsilon$. It performs $O(nD^d)$ calls to the oracle on points of size $\log(\frac{n^2 D}{\epsilon})$.

Let $Q_d, Q_{d-1} \ldots, Q_1$ be the sequence of polynomials on which the procedure Monomial of Algorithm 6 is recursively called. We denote by l_i the size of L_i and by n_i the number of variables of Q_i for all $i \leq d$. Since Proposition 4 proves that L_i is maximal, Q_i is of degree one less than Q_{i+1}. By a simple induction, we have that Q_i is of degree i. The correction of the algorithm derives from the construction of procedures *max_monomial* and *restriction* given in Propositions 4 and 5.

We now bound the number of oracle calls this algorithm performs. One evaluation of Q_i done through the procedure *restriction* requires $O((i+1)l_{i+1})$ calls to Q_{i+1} by Proposition 5. By induction we have that one evaluation of Q_i requires $\prod_{j=i+1}^{d} jl_j$ calls to P. The points on which the oracle is called are of size less than $\log(D)$ for all i.

In Algorithm 6, the procedure *max_monomial* computes L_i in $in_i l_i \log(n^2\epsilon^{-1})$ calls to Q_i. By the previous remark, these calls to Q_i are in fact $n_i \log(n^2\epsilon^{-1}) \prod_{j=i}^{d} jl_j$ calls to P.

It holds that $\sum_{j=i}^{d} jl_j \leq D$ because the term $\prod_{j=i}^{d} (\vec{X}^{L_j})^j$ divides a monomial of P and thus its total degree is less than the total degree of P. Therefore $\prod_{j=i}^{d} jl_j \leq D^{d-i+1}$. Since $n \leq n_i$, the number of calls to P when executing *max_monomial* on Q_i is bounded by $n \log(n^2\epsilon^{-1}) D^{d-i+1}$.

The total number of calls to P in the algorithm is the sum of calls done by *max_monomial* on each polynomial Q_i for $1 \leq i \leq d$. Hence it is bounded by $\sum_{i=1}^{d} n \log(n^2\epsilon^{-1}) D^{d-i+1} = O(nD^d \log(n^2\epsilon^{-1}))$.

The complexity of Algorithm 6 depends on the number of degrees at which the variables of the polynomial appear rather than on the degree itself. It means that, if we want to find a monomial of a polynomial whose variables are either at the power one or forty-two, the previous algorithm does $O(nD^2)$ calls and not $O(nD^{42})$.

Algorithm 6 produces a monomial of a polynomial of any degree with an arbitrary small probability of error. It can thus be used to implement *find_monomial* in Algorithm 3, which yields an incremental delay interpolation algorithm for fixed d.

6 Complexity Classes for Randomized Enumeration

In this section the results about interpolation in the black box formalism are transposed into complexity results on more classical combinatorial problems. Let first define how we represent efficiently an enumeration problem by a family of polynomial.

[Polynomial representation] Let $A(x, y)$ be a polynomially balanced predicate decidable in polynomial time. The family of polynomial $(P_x)_{x\in\Sigma^*}$ represents the problem Enum·A if:

- There is a bivariate polynomial time function f such that for all $x \in \Sigma^*$, $f(x, .)$ is a bijection between the monomials of P_x and the set of solutions $\{y \mid A(x, y)\}$.
- There is an algorithm which computes $P_x(\vec{v})$ on any values $\vec{v}$ in a time polynomial in the size of $\vec{v}$ and x.

This definition is tailored so that any interpolation algorithms for the family $(P_x)_{x\in\Sigma^*}$ gives an algorithm of the same complexity for the problem Enum·A.

Let Enum·Cycle-cover be the problem of listing the cycle covers of a graph. To each graph G is associated a polynomial P_G: the determinant of its adjacency matrix. The monomials of P_G are in bijection with the cycle covers of G. Moreover the determinant of a matrix can be evaluated in a polynomial time in the matrix, therefore the family (P_G) represents the problem Enum·Cycle-cover.

We recall a few complexity classes for enumeration and introduce their probabilistic counterparts. We show that different enumeration algorithms enable us to prove that several problems are in these classes through the polynomial representation we have just defined.

There are three main enumeration complexity classes (for further details on their properties and relationships, see Chapter 2 of [41]):

1. the problems that can be solved in polynomial total time, **TotalP**
2. the problems that can be solved with incremental delay, **IncP**
3. the problems that can be solved with polynomial delay, **DelayP**

We now give the probabilistic version of **TotalP**.

A problem Enum·A is computable in probabilistic polynomial total time, written **TotalBPP**, if there is a polynomial $Q(x, y)$ and a machine M which solves Enum·A with probability greater than $\frac{2}{3}$ and satisfies for all x, $T(x, |M(x)|) < Q(|x|, |M(x)|)$.

The class **TotalBPP** is very similar to the class **BPP** for decision problems. For both classes, the choice of $\frac{2}{3}$ is arbitrary, everything larger than $\frac{1}{2}$ would do. Indeed, one can increase the probability of success exponentially in a polynomial time as soon as it is strictly larger than $\frac{1}{2}$. To achieve this, repeat a polynomial

number of times the algorithm working in total polynomial time and return the set of solutions generated in the majority of runs. To prove that the probability of success has increased properly, one has to use Chernoff's bound, which is stated and proved in [5]. Note that this method requires a space proportional to the number of solutions.

A refinement [25] of Zippel's algorithm [49] solves Enum·Poly in a time polynomial in the number of monomials. The polynomial P_G of Example 2 can be interpolated by this algorithm. Each produced monomials is then translated into a cycle cover of G and outputted. This proves that Enum·Cycle-cover $\in$ **TotalBPP**. We now explain how the interpolation algorithms we have presented in this article help improve this result.

A problem Enum·A is computable in probabilistic incremental time, written **IncBPP**, if there is a polynomial $Q(x, y)$ and a machine M which solves Enum·A with probability greater than $\frac{2}{3}$ and satisfies for all x, $T(x, i+1) - T(x, i) \leq Q(|x|, i)$.

Again, it is possible to improve the error bound in the definition if we are willing to use a lot of space. However, the proof is more technical than in the **TotalBPP** case.

[Proposition 3.17 of [41]] If Enum·A is in **IncBPP** then there is a polynomial Q and a machine M which for all ϵ computes the solutions of Enum·A with probability $1-\epsilon$ and satisfies for all x, $T(x, i+1) - T(x, i) \leq Q(|x|, i)\log(\epsilon^{-1})$.

The class **IncBPP** may be related to the following search problem:

AnotherSolution$_A$
Input: an instance x of A and a subset S of $A(x)$
Output: an element of $A(x) \setminus S$ and a special value if $A(x) = S$

[Proposition 3.19 of [41]] There is an algorithm which computes with probability $\frac{2}{3}$ a solution of AnotherSolution$_A$ in polynomial time if and only if Enum·A $\in$ **IncBPP**.

The last proposition roughly states that if there is an algorithm which produces one solution of a problem, then there is an incremental delay algorithm which solves the enumeration problem. It is exactly the method we use in Section 3 to obtain incremental delay interpolation algorithms.

We want to solve the problem Enum·Perfect-Matching, that is to enumerate the perfect matchings of a graph. To a graph G, we associate the polynomial $\mathsf{PerfMatch}(G)$, whose monomials represent the perfect matchings of G. We write $\mathcal{C}$ the set of perfect matching of G.

$$\mathsf{PerfMatch}(G) = \sum_{C \in \mathcal{C}} \prod_{(i,j) \in C} X_{i,j}$$

For graphs with a "Pfaffian" orientation, such as the planar graphs, this polynomial is related to a Pfaffian and can be evaluated in polynomial time. Thus $(\mathsf{PerfMatch}(G))$ is a representation of Enum·Perfect-Matching restricted to the planar graphs. We can use Algorithm 3 to interpolate any polynomial of $(\mathsf{PerfMatch}(G))$ therefore Enum·Perfect-Matching $\in$ **IncBPP**. Moreover

all the coefficients of $\mathsf{PerfMatch}(G)$ are positive therefore the interpolation algorithm is deterministic and Enum·Perfect-Matching $\in$ **IncP**. Note though that there already exists a very efficient algorithm to list all perfect matchings [44].

Our two previous examples of enumeration problems are represented by *multilinear* polynomials and are thus in the following class, thanks to Algorithm 4.

A problem Enum·A is computable in probabilistic polynomial delay **DelayBPP** if there is a polynomial $Q(x, y)$ and a machine M which solves Enum·A with probability greater than $\frac{2}{3}$ and satisfies for all x, $T(x, i+1) - T(x, i) \leq Q(|x|)$.

We consider the problem Enum·Spanning-Tree, that is to enumerate all spanning trees of a graph. Let G be a graph, the Kirchhoff matrix $K(G)$ is defined by: for $i \neq j$, $K(G)_{i,j} = -X_{i,j}$ and $K(G)_{i,i} = \sum_{(i,j)\in E(G)} X_{i,j}$. The Matrix-Tree theorem (see [22]) is the following equality where $\mathcal{T}$ is the set of spanning trees of G:

$$\det(K(G)) = \sum_{T\in\mathcal{T}} \prod_{(i,j)\in T} X_{i,j}$$

The family $(\det(K(G)))$ represents the problem Enum·Spanning-Tree. Moreover for all G, $\det(K(G))$ is multilinear and monotone therefore using Algorithm 4 we have Enum·Spanning-Tree $\in$ **DelayP**.

Let A be a probabilistic automaton with n states. For each word $w = \sigma_1\sigma_2\ldots\sigma_k$ we denote by $A(w)$ its probability to be accepted. Associate with A the polynomial

$$P_A(x) = \sum_{k=0\ldots n} \sum_{w\in\Sigma^k} A(w) x_{\sigma_1,1} x_{\sigma_2,2} \ldots x_{\sigma_k,k}$$

to the probabilistic automaton. This polynomial is multilinear, monotone and can be computed in polynomial time in n a proved in [28].

We want to enumerate the worlds of A along with their probabilities (but not the words of probability zero), a problem we denote by Enum·Automaton. Since (P_A) represents Enum·Automaton, we have Enum·Automaton $\in$ **DelayP**. In [28], the problem of deciding whether two automata A and B have the same language is solved by testing if $P_A - P_B$. Remark that $P_A - P_B$ is not monotone but still multilinear. Therefore the problem of enumerating all the words of a given size in the symmetric difference of the languages of A and B is in **DelayBPP**.

7 A few improvements to interpolation algorithms and an application

In this section we give several methods to improve the complexity of the algorithms of this article. We denote here by ω the matrix multiplication exponent,

that is the smallest number such that there is an algorithm to multiply two n by n matrices in time $O(n^\omega)$. The bound on ω has been recently improved to $w < 2,3727$ [10, 47].

7.1 Finite fields

We first show that the algorithms we have presented works over small finite fields. We then explain how it can help to speed-up interpolation of polynomials with coefficients in $\mathbb{Z}$.

Let P be a polynomial of total degree D with coefficient in F a finite field of cardinality larger than $2D$. Algorithm 3 uses evaluation points in $[2D]$ to interpolate polynomials of total degree D. Since it relies on the Schwarz-Zippel lemma which holds over a finite field, it works without modification to interpolate P. Algorithm 4 in addition to Schwarz-Zippel lemma relies on a dense interpolation of a univariate polynomial to compute a partial degree. Since the univariate polynomial is of degree less than D, it is uniquely defined by its monomials over F and the subroutine to compute the partial degree return a correct answer. Therefore Algorithm 4 also works over fields with $2D$ elements or more. This behavior is good in comparison with other classical algorithms, for instance the one of [49], which needs exponentially larger evaluation points and thus cannot be adapted to small finite fields.

If we want to minimize the bit cost of our interpolation algorithms, we have to use integers as small as possible. Another option is to do the computation on a finite field of small size. Moreover, if we replace a black box call by an actual computation, like in Section 6, it may be more efficient to do this computation over a finite field. For instance, the computation of a determinant can be done in $O(n^\omega)$ arithmetic operations [10], which have a time complexity of $O(\log(p))$ over $\mathbb{F}_p$.

Assume now that we want to interpolate a polynomial P with small integer coefficients, less than its total degree D. Remark that it is the case in most examples given in Section 6: the polynomials have coefficients 1 or -1. There is a prime p between D and $2D$. Let us consider P modulo p, it has the same monomials as P. All computations are then done in the finite field $\mathbb{F}_p$. It is especially useful to speed up the computation of *substract* in Algorithms 3 and 6. Indeed, one needs to evaluate a polynomial given explicitly, and arithmetic operations are computed faster in a small finite field. In particular the result of the evaluation of a monomial in $\mathbb{F}_p$ is always of size $O(\log(D))$, while its value in $\mathbb{Z}$ is of size $O(D \log(D))$.

Finally, assume that P is a polynomial with large coefficients. For complexity reason, one may want to evaluate P only modulo some small number. It is possible to adapt Algorithms 3 and 4: at each of their step choose a random prime $p > D$ and do all computations modulo p. With high probability, the algorithms will work in the same way. The only problem is, once a support L is found so that P_L is a monomial, to compute the coefficient of P_L. We can compute the coefficient, that is $P_L(1, \ldots, 1)$, modulo several primes. Thanks to the Chinese remainder theorem, we derive from these values the value over $\mathbb{Z}$ of

the coefficient.

7.2 A method to decrease the degree

We have seen that the complexity of Algorithm 6 is highly dependent on the degree of the interpolated polynomial. We propose here a simple technique to decrease the degree of the polynomial by one, which makes Algorithm 6 competitive for degree 10 polynomials.

Let P be a polynomial of degree d. A procedure similar to the one of Algorithm 2 is run on P. It finds, with probability $1-\epsilon$, a minimal set L such that P_L is not zero, with $n\log(\epsilon^{-1})$ calls to the oracle. In this case, it does not give a monomial but we can write $P_L(\vec{X}) = \vec{X}^L Q(\vec{X})$ and Q is of degree $d-1$.

We may simulate an oracle call to $Q(\vec{X})$ by a call to the oracle giving P_L and a division by the value of $\vec{X}^L$ as long as no X_i is chosen to be 0. Moreover the monomials of Q are in bijection with those of P_L by multiplication by $\vec{X}^L$. Therefore to find a monomial of P we only have to find a monomial of Q.

Algorithm 6 does not evaluate the polynomial to 0, hence we can run it on Q, that we simulate with a low overhead in time, and one call to P for each evaluation. Since the polynomial Q have degree $d-1$, the computation of one of its monomials requires only nD^{d-1} calls to the oracle plus the n calls used to find L.

7.3 Adaptive error bound

In all interpolation algorithms the subroutines are called with an error bound exponentially small in the number of variables. However, we only need to set the error bound to be in $O(\frac{1}{t})$, where t is the number of monomials. It is better when the polynomial is sparse, but t is usually hard to compute. We now explain how to reduce the error bound without knowing t in Algorithm 3. Note that the exact same technique works for the other interpolation algorithms.

We add a variable i to count the number of calls to *find_monomial* in Algorithm 3. The idea is to decrease the error bound at each new monomial, quickly enough so that the probability of error of the whole algorithm is still small. We replace the call to *find_monomial*($subtract(P,Q), \frac{\epsilon}{2^{n+1}}$) by *find_monomial*($subtract(P,Q), \frac{\epsilon}{2i^2}$).

There are t calls to *find_monomial*, therefore the error bound of the whole algorithm is bounded by $\sum_{i=1}^{t} \frac{\epsilon}{2i^2}$. Since $\sum_{i=0}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6} < 2$, the total probability of error is less than ϵ. In the end, the modified algorithm still have a probability $1-\epsilon$ to correctly interpolate the polynomial.

The first advantage is obviously that the factor $n + \log(\epsilon^{-1})$ in the delay, the total time and the number of calls to the oracle of Algorithm 3 becomes $\log(t) + \log(\epsilon^{-1})$. It is an improvement when $\log(t)$ is in $o(n)$.

Moreover it could help broaden the class of polynomials which are interpolable with an incremental delay. To do that, another implementation of the procedure *find_monomial*(P, ϵ) is required. If we use this new way to set the

error bound in Algorithm 3, the complexity of *find_monomial* has only to be polynomial in ϵ^{-1} rather than in $\log(\epsilon^{-1})$.

7.4 Application: listing the spanning hypertrees

In Section 6, we have seen how to enumerate the spanning trees of a graph thanks to their representation by a determinant. The generalization of a spanning tree in a hypergraph is a connected acyclic subhypergraph or hypertree. The most well-known notions of acyclicity are Berge, γ, β and α-acyclicity. Let denote by Spanning Hypertree the problem of deciding whether there is a spanning hypertree in a hypergraph and by Enum·Spanning Hypertree the corresponding enumeration problem.

For the three notions γ, β and α-acyclicity and 3-uniform hypergraphs, Spanning Hypertree is NP-complete [15]. On the other hand, Spanning Hypertree is NP-complete for Berge-acyclicity and 4-uniform hypergraphs but not when restricted to 3-uniform hypergraphs. Indeed, one can adapt Lovász matching algorithm in linear polymatroids [31] to solve Spanning Hypertree for 3-uniform hypergraphs. This algorithm is very complicated and not designed for this particular case hence it seems hard to extend it into an efficient enumeration algorithm of the spanning hypertrees. However it is easy to give a randomized enumeration algorithm by using Algorithm 4.

To this aim, we must first give a representation of Enum·Spanning Hypertree by a family of polynomials. Let H be a 3-uniform hypergraph, we denote by $\mathcal{T}(H)$ the set of its spanning hypertrees. We now define a polynomial Z_H, such that its monomials are in bijection with the spanning hypertrees of H.

$$Z_H = \sum_{T \in \mathcal{T}(H)} \epsilon(T) \prod_{e \in E(T)} w_e$$

where $\epsilon(T) \in \{-1, 1\}$.

The function $\epsilon(T)$ has a precise definition, see [34], but it is not needed here. This polynomial has exactly one variable w_e for each hyperedge e of H.

Let H be a 3-uniform hypergraph, $\Lambda(H)$ is the Laplacian matrix defined by $\Lambda(H)_{i,j} = \sum_{i \neq k,j} \epsilon_{ijk} w_{\{i,j,k\}}$.

$\epsilon_{i,j,k}$ is 0 when $\{ijk\} \notin E(H)$, otherwise $\epsilon_{ijk} \in \{-1, 1\}$.

The coefficient ϵ_{ijk} is equal to 1 when $i < j < k$ or any other cyclic permutation and is equal to -1 when $i < k < j$ or any other cyclic permutation. Thus ϵ_{ijk} is computable in polynomial time in the size of i, j and k. We may relate to Z_H, the Pfaffian of the Laplacian matrix which is of interest since it is computable in polynomial time. The following theorem is a generalization of the Matrix-Tree theorem for graphs.

[Pfaffian-Hypertree (cf. [34])] Let $\Lambda(i)$ be the minor of $\Lambda(G)$ where the column and the line of index i have been removed.

$$Z_H = (-1)^{i-1} Pf(\Lambda(i))$$

For H a hypergraph with n vertices and m hyperedges, Z_H is a multilinear polynomial with m variables, and the size of its coefficients is one. Moreover, it is of total degree $\frac{n-1}{2}$ because a spanning hypertree of a 3-uniform hypergraph has $\frac{n-1}{2}$ hyperedges. Finally by Theorem 7, Z_H is computable in polynomial time, therefore Z_H represents the problem Enum·Spanning Hypertree.

The problem Enum·Spanning Hypertree for 3-uniform hypergraphs is in **DelayBPP**. More precisely, there is an algorithm solving the problem with delay $O(mn^{1+\omega}(n\log(n)+\log(\epsilon^{-1})))$ (in bit complexity) where m is the number of hyperedges, n is the number of vertices and ϵ is a bound on the probability of error. The fact that Enum·Spanning Hypertree for 3-uniform hypergraphs is in **DelayBPP** is clear since it is represented by a multilinear polynomial family.

Let now compute precisely the delay. Let H be a hypergraph with n vertices, the degree of Z_H is $\frac{n-1}{2}$ (or 0). Let p be a prime number between $\frac{n-1}{2}$ and n. We now compute Z_H over $\mathbb{F}_p$ as explained in a previous section.

To solve Enum·Spanning Hypertree on the instance H, Algorithm 4 is run on the polynomial Z_H. We first evaluate the time taken by the the calls to the oracle, here the computation of the polynomial Z_H. The first step to compute Z_H is to build the $(n-1)*(n-1)$ matrix $\Lambda(i)$ and it is negligible with regard to the time to compute its Pfaffian. The evaluation of a Pfaffian can be reduced to the evaluation of a Determinant which itself requires as many arithmetic operations as a matrix product. Since an arithmetic operation over the field $\mathbb{F}_p$ has a complexity $O(\log(n))$, the evaluation of Z_H on any points of $\mathbb{F}_p$ has a complexity $(n^\omega)^{1+o(1)}$.

By Theorem 5 there are $O(mn(m+\log(\epsilon^{-1})))$ oracle calls between two solutions. By using an adaptive error bound as explained in Section 7, we can decrease it to $O(mn(\log(t)+\log(\epsilon^{-1})))$ where t is the number of spanning hypertrees, which is bounded by n^n. In conclusion, the contribution to the delay of the evaluations of Z_H is a $O((mn^{\omega+1}(n+\log(\epsilon^{-1})))^{1+o(1)})$.

One must take into account the cost of the univariate interpolation that the procedure *not_zero_improved* realizes. Since Z_H is defined over a finite field, the size of the evaluations of the polynomial are $O(\log(n))$. Furthermore, its total degree is $\frac{n-1}{2}$, hence the time to do the interpolation is $O(n^2\log(n))$. The contribution to the delay is negligible, since for one interpolation, there are n evaluations of Z_H, each of them taking more time than the interpolation.

Since the size of a 3-uniform hypergraph is typically $m=\Theta(n^3)$, the delay is quite good, less than cubic. As one of the anonymous reviewer pointed out, this algorithm could be derandomized if we can decide in deterministic polynomial time whether a hypergraph has a spanning hypertree, with some edges forbidden and some others forced. Note that we can forbid an edge by removing it from the hypergraph. However, contracting an edge is not equivalent to forcing it to be in any hypertree. For instance the hypergraph $\{\{v_1,v_2,v_3\},\{v_1,v_2,v_4\}\}$ has no spanning hypertree, but when the edge $\{v_1,v_2,v_3\}$ is contracted into a vertex, the resulting hypergraph is a single edge and thus a hypertree.

8 Polynomials represented by circuits

In this last section, we consider a polynomial represented by an *arithmetic circuit* instead of a black box. An arithmetic circuit is a directed acyclic graph, whose nodes of indegree zero are called input gates and are labeled by either a variable or an integer. All the other nodes are of indegree two or more and are labeled by either $+$ or $\times$. There is one node of outdegree zero which is called the output gate. The arithmetic circuit defines a polynomial by computing the value of each node using the appropriate operator and the value of its ancestors. An *arithmetic formula* is a circuit whose underlying graph is a directed tree. The depth of a circuit is the length of its longest directed path.

The representation of a polynomial by a circuit is more explicit than by a black box. That is why some problems are easier to solve in this setting. For instance there is an interpolation algorithm which is polynomial in the number of monomials and in the logarithm of the degree of the polynomial [19]. Moreover PIT can be solved by deterministic polynomial time algorithms for several classes of circuits of small depth. In this section, we transfer a derandomization result to enumeration. We also prove that some problems related to interpolation are NP-hard or #P-hard over formulas of small depth.

8.1 Derandomization

We give here a solution to the problem Monomial-Factor in a different model. We say that a circuit is multilinear when each of its gates compute a multilinear polynomial. We want to solve Monomial-Factor with input a multilinear circuit C and a set of variable S representing the term $\vec{X}^S$. Thanks to a transformation of C into its homogeneous components with regard to S, we obtain an arithmetic circuit which represents a polynomial different from zero if and only if the answer to Monomial-Factor is positive.

Let C be an arithmetic circuit of size c, let P be the polynomial it computes and S a set of variables of cardinal s. There is a polynomial time algorithm which produces another circuit C' which computes the zero polynomial if and only if $(P, S) \in$ Monomial-Factor. Moreover if C is a multilinear circuit of depth l and of top fan-in k, C'' is a multilinear circuit of depth at most l and of top fan-in at most k.

Let consider that every variable X in S is replaced in the circuit C by XY where Y is a new variable. The degree of Y is at most s in the computed polynomial, that we call Q. By a classical construction, see for instance Lemma 2.14 of [9], we can build a new circuit of size $O(s^2c)$ which computes each homogeneous component in Y of Q. Let C' be this circuit where the output node is the one computing the homogeneous component of degree s of Q and where Y is replaced by 1.

Remark now that C' does not compute the zero polynomial if and only if Q is of degree s in Y or said otherwise if the partial degree of P in S is s. Therefore, to decide whether $(P, S) \in$ Monomial-Factor is equivalent to solve the problem PIT on the circuit C'.

The claims on depth, fanin and multinearity are clear from a careful examination of the homogenization procedure. Each gate of the circuit is turned into $|S|+1$ gates, which compute the homogeneous components of degree 0 to $|S|$. The main problem is that multiplication gates are replaced by a sum of multiplication gates. When inside the circuit, the new addition gate can be merged with an addition gate of the next layer. But a multiplication gate in the last or the next to last layer increases either the depth of 1 or the arity of the top gate. However, since we want to compute only the homogeneous component of degree $|S|$, we can remove most of the added gates and we obtain a circuit of the same depth and top arity.

If we want to solve Monomial-Factor with the best possible complexity, it is possible not to build explicitly C' but to evaluate its value on inputs directly from the circuit C. A very efficient parallel algorithm for this kind of problem can be found in [6].

We recall that if the interpolated polynomials are monotone then both Algorithms 3 and 4 can be made deterministic. We want here to derandomize our interpolation algorithms for classes of polynomials represented by circuits. First remark that if we are able to solve Enum·Poly in deterministic polynomial total time over a class of polynomials, then we can also solve PIT in deterministic polynomial time. Therefore we cannot hope to derandomize algorithms for Enum·Poly on larger classes of polynomials than for PIT.

The good news is that Algorithms 3 and 4 can be made deterministic on any class of circuits on which there is a deterministic algorithm for PIT. In Algorithm 3, the only randomized step is the call to the procedure *not_zero* which solves PIT on a restriction of the polynomial we want to interpolate. Therefore a deterministic solution for PIT makes Algorithm 3 deterministic.

The source of randomness in Algorithm 4 is the call to the randomized subroutine Monomial-Factor. Since Proposition 9 reduces the problem Monomial-Factor on a multilinear polynomial given by a circuit to the problem PIT on a circuit of the same kind, we can also derandomize Algorithm 4.

The largest class of multilinear polynomials for which PIT is known to be in P is given in the following theorem, which is also implied by some recent results on algebraically independent polynomials [2].

[In [38]] Let k, n, s be integers. There is an explicit set of integers $\mathcal{H}$ of size polynomial in n, s and exponential in k that can be constructed in a time linear in its size such that the following holds. Let P be a non-zero polynomial on n variables computed by a multilinear circuit of size s, depth 4 and top fanin k. Then there is some $\alpha \in \mathcal{H}$ such that $P(\alpha) = 0$.

The problem Enum·Poly restricted to polynomials represented by multilinear circuits of depth 4 and bounded top fanin is in **DelayP**.

Theorem 8 gives a black box algorithm for PIT: it only requires to be able to evaluate the polynomial and does not need its representation by a circuit. But the algorithm used to solve the problem Monomial-Factor need the circuit itself for the homogenization. However our algorithm can be made black box thanks to the following proposition inspired by Proposition 2.

Let P be a polynomial given by a black box and D its total degree. Let

$k \leq D$, it is possible to simulate a black box for P_k the homogeneous component of P of degree k with D calls to P. Let $\vec{x}$ be the point on which we want to compute P_k. The polynomial $\tilde{P}$ is obtained by substituting formally to each variable X_i of P the product $X_i X_{n+1}$. The univariate polynomial $\tilde{P}(\vec{x}, X_{n+1})$ is interpolated with D calls to the oracle. The coefficient of the term X_{n+1}^k is $P_k(\vec{x})$ the value we had to compute.

Remark that the structural homogenization method is still needed to prove that, when P is representable by a circuit of low depth, P_k is representable by a circuit of the same depth. This method does not prove that a polynomial represented by a formula of unbounded depth has its homogeneous components representable by formulas of size polynomial in the original one. By a different technique involving univariate interpolation, this result holds (see the proof of Theorem 2 in [24]). But this cannot be used to deal with bounded-read formulas since this property is not preserved by the homogenization. Though, we can work on the formula itself to solve in deterministic polynomial time, using the deterministic algorithm for PIT [4] as in [32]. In conclusion, we can derandomize Algorithm 4 on any class of multilinear bounded-read formulas.

8.2 Hard problems for low degree low depth polynomials

We describe four (families of) polynomials, representable by formulas or circuits. Their sizes and the degree of the polynomials computed at each of their nodes are polynomial in the number of their variables therefore the represented polynomials can be evaluated in polynomial time. We prove that we can encode hard combinatorial questions in these polynomials such as the Monomial Factor problem or one of the two following problems:

Non-Zero-Monomial
Input: a polynomial given as a circuit and a term $\vec{X}^{\vec{e}}$
Output: accept if $\vec{X}^{\vec{e}}$ has a coefficient different from zero in the polynomial

Monomial-Coefficient
Input: a polynomial given as a circuit and a term $\vec{X}^{\vec{e}}$
Output: return the coefficient of $\vec{X}^{\vec{e}}$ in the polynomial

We now explain how these three problems are related.

If the problem Monomial-Factor can be solved in probabilistic polynomial time (in the number of variables and the total degree of the polynomial) for a class of polynomials, then Non-Zero-Monomial can be solved in probabilistic polynomial time for the same class. Let P be a polynomial of total degree D with n variables and $\vec{X}^{\vec{e}}$ a term. Let k be the total degree of $\vec{X}^{\vec{e}}$, we denote by P_k the sum of monomials of P of total degree k. Let us remark that $(P, \vec{X}^{\vec{e}}) \in$ Non-Zero-Monomial if and only if $(P_k, \vec{X}^{\vec{e}}) \in$ Monomial-Factor.

By Proposition 10 it is possible to evaluate P_k in polynomial time since P itself can be evaluated in polynomial time. Therefore we can test whether $(P_k, \vec{X}^{\vec{e}}) \in$ Monomial-Factor in polynomial time.

The converse of the proposition does not seems to hold, that is Monomial-Factor may be harder than Non-Zero-Monomial. The problem Monomial-Coefficient is the search version of Non-Zero-Monomial and is thus also harder. Therefore, the best result we can achieve is to prove that Non-Zero-Monomial is a hard problem on a family of polynomials.

Note that Monomial-Factor is exactly the problem solved for multilinear polynomials by the procedure *linear_factor* and Monomial-Coefficient the problem solved by *coefficient*. In what follows, we prove that these problems are not likely to be solvable in probabilistic polynomial time, even restricted to polynomials of small degree representable by formulas of small depth. As a consequence, Algorithm 4, which is based on repeatedly solving Monomial-Factor, cannot be generalized to polynomials of degree 2 unless RP = NP. Therefore a new method must be devised to find a polynomial delay algorithm for polynomials of degree two and more.

The results of the next sections are recapped in the following table. Finer completeness results which uses some of the polynomial families introduced in the next sections can be found in [17] but these results do not take into account the degree of the polynomials.

	Unbounded degree Depth-2	Degree 3 Depth-3	Degree 2 Circuit	Degree 2 Depth-4
Non-Zero-Monomial	P	NP-hard	NP-hard	?
Monomial-Coefficient	#P-hard	#P-hard	#P-hard	?
Monomial-Factor	P	NP-hard	NP-hard	NP-hard

8.3 Depth-3 formula, unbounded degree

As a warm-up we present a simple hardness result obtained thanks to a classical polynomial introduced by Valiant (see [46]). The polynomial Q has $n^2 + n$ variables, degree n and is defined by:

$$Q(X, Y) = \prod_{i=1}^{n} (\sum_{j=1}^{n} X_{i,j} Y_j)$$

If we see Q as a polynomial in the variables Y_j only, the term $T = \prod_{j=1}^{n} Y_j$ has $\sum_{\sigma \in \Sigma_n} \prod_{i=1}^{n} X_{i,\sigma(i)}$ for coefficient, which is the Permanent in the variables $X_{i,j}$.

The problem Monomial-Coefficient is #P-hard over polynomials given by depth-3 formulas. One can reduce the problem Monomial-Coefficient in polynomial time to the computation of the Permanent. Assume one wants to compute the Permanent of the n^2 values $x_{i,j}$. Let us consider the polynomial $Q(\vec{x}, \vec{Y})$ where $x_{i,j}$ has been substituted to $X_{i,j}$. It is represented by a depth-3

formula of size polynomial in n. The coefficient of T in $Q(\vec{x}, \vec{Y})$ is the Permanent of the $x_{i,j}$'s and it is also the solution to MONOMIAL-COEFFICIENT when given $Q(\vec{x}, \vec{Y})$ and T as input. Since the computation of the Permanent is #P-complete, the problem MONOMIAL-COEFFICIENT is #P-hard over polynomials given by depth-3 formulas.

8.4 Depth-3 formula, degree 3

We now prove a hardness result for the problem NON-ZERO-MONOMIAL over depth-3 formulas of degree at most 3. In the next subsection, we prove a similar result where the degree bound is improved to 2, but the polynomials are represented by circuits of any depth.

The problem NON-ZERO-MONOMIAL restricted to degree 3 polynomials represented by a depth-3 formula is NP-hard.

Let C be a collection of three-elements subsets of $[n]$ (3-uniform hypergraphs). We construct a polynomial from C, on the variables $(X_i)_{i\in[n]}$, as follows. To each subset C' of C we associate the monomial $\chi(C') = \prod_{\{i,j,k\}\in C'} X_i X_j X_k$. Let Q_C be the polynomial:

$$\sum_{C'\subseteq C} \chi(C')$$

It can be represented by a depth-3 formula polynomial in the size of C, since it is equal to:

$$\prod_{\{i,j,k\}\in C} (X_i X_j X_k + 1)$$

The degree of Q_C is the maximal number of occurrences of an integer in elements of C. If each integer of $[n]$ appears in at most three elements of C, Q_C is of degree 3 and the problem of finding an exact cover of C is still NP-complete [18].

By definition of χ, a subset C' is an exact cover of $[n]$ if and only if $\chi(C') = \prod_{i\in[n]} X_i$. Therefore to decide if C' has an exact cover, we only have to decide if $\prod_{i\in[n]} X_i$ has a coefficient different from zero. It proves that NON-ZERO-MONOMIAL is NP-hard over circuits representing degree 3 polynomials.

8.5 Circuit, degree 2

To obtain the hardness result of this subsection we have to drop all assumption on the structure of the circuit. Indeed, we use a determinant in the proof which can "efficiently" simulate a large class of polynomials representable by a polynomial size circuit (see [43, 33]).

The problem NON-ZERO-MONOMIAL restricted to degree 2 polynomials given by circuits is NP-hard.

Let G be a directed graphs on n vertices, the Laplace matrix $L(G)$ is defined by $L(G)_{i,j} = -X_{i,j}$ when $(i,j) \in E(G)$, $L(G)_{i,i} = \sum_{(i,j)\in E(G)} X_{i,j}$ and 0

otherwise. Let $\mathcal{T}_s$ be the set of spanning trees of G, rooted in s and such that all edges of a spanning tree is oriented away from s. Let $L(G)_{s,t}$ be the minor of $L(G)$ where the row s and the column t have been deleted.

The Matrix-Tree theorem (see [3] for more details) is the following equality:

$$\det(L(G)_{s,t})(-1)^{s+t} = \sum_{T \in \mathcal{T}_s} \prod_{(i,j) \in T} X_{i,j}$$

We substitute to $X_{i,j}$ the product of variables $Y_i Z_j$ in $\det(L(G)_{s,t})$ which makes it a polynomial in $2n$ variables. This polynomial is derived from a Determinant and can thus be represented by a polynomial size circuit. Every monomial is in bijection with a spanning tree whose maximum outdegree is the degree of the monomial. We assume that every vertex of G has indegree and outdegree less or equal to 2 therefore $\det(L(G)_{s,t})$ is of degree 2.

Remark now that a spanning tree, all of whose vertices have outdegree and indegree less or equal to 1 is an Hamiltonian path. Therefore G has an Hamiltonian path beginning by s and finishing by a vertex v if and only if $\det(L(G)_{s,t})$ contains the monomial $Y_s Z_v \prod_{i \notin \{s,v\}} Y_i Z_i$. To decide whether G has an Hamiltonian path, one has only to solve Non-Zero-Monomial on the polynomial $\det(L(G)_{s,t})$ and the term $Y_s Z_v \prod_{i \notin \{s,v\}} Y_i Z_i$, for all pairs (s, v) which are in polynomial number. The Hamiltonian path problem restricted to directed graphs of outdegree and indegree at most 2 is NP-complete [36]. Therefore Non-Zero-Monomial is NP-hard over degree 2 polynomials.

8.6 Depth-4 formula, degree 2

Here we give a hardness result for the problem Monomial-Factor over degree 2 polynomials. Since Non-Zero-Monomial reduces by Proposition 11 to Monomial-Factor it should be obvious, but we also restrict the input to be a depth-4 formula.

The problem Monomial-Factor restricted to degree 2 polynomials represented by a depth-4 formula is NP-hard.

Let ϕ be a 2-CNF formula, it is a conjunction of n clauses C_i and each of them is the disjunction of two literals, which are either a variable or the negation of a variable. We note V the set of variables of ϕ and $v \in C_i$ if v is one of the literal of C_i. We build the polynomial Q_ϕ from ϕ. It has n variables X_i which represent the clauses C_i and one special variable Y.

$$Q_\phi(\vec{X}, Y) = \prod_{v \in V} ((Y \prod_{\neg v \in C_j} X_j) + (\prod_{v \in C_i} X_i)) \tag{1}$$

Any empty product in the equation is 1.

Remark that each clause has at most two literals, thus any variable X_i appears in at most two factors of the outermost product. Therefore the polynomial is of degree 2 in the variables X_i. We rewrite Q_ϕ by expanding the product over V. In the next equation, the function d can be seen as a distribution of truth values or as a choice in each factor of Q_ϕ of the left or right part of the sum.

The integer $\alpha(d)$ is the number of j such that $d(j) = 0$ and $\beta(d, i)$ is the number of literals in C_i made true by d.

$$Q_\phi(\vec{X}, Y) = \sum_{d \in 2^{|V|}} Y^{\alpha(d)} \prod_i X_i^{\beta(d,i)}$$

We write $Q_\phi(\vec{X}, Y) = \sum_{k=1}^{|V|} Y^k Q_k(\vec{X})$. Equation 1 allows us to build a circuit of size and formal degree polynomial in ϕ, which represents Q_ϕ. By homogenization, as in Proposition 9, one builds from the formula representing Q_ϕ, a formula which represents $Q_k(\vec{X})$. Its depth is one more than the original formula, that is 4.

Finally, a monomial comes from a truth value assignment which satisfies ϕ if and only if $T = \prod_{i=1}^{n} X_i$ divides the monomial. In addition, a monomial represents an assignment of Hamming weight k if and only if it is in Q_k. The problem Monomial-Factor for the polynomial Q_k and the term T is hence equivalent to the problem of deciding if ϕ has a satisfying assignment of Hamming weight equal to k. This latter problem is NP-complete over 2-CNF formulas [16]. Therefore, Monomial-Factor is NP-hard over degree 2 polynomials.

Remark that Monomial-Coefficient is solved in polynomial time for multilinear polynomials by Corollary 1, while it is NP-hard for degree 2 polynomials, by the last proposition. Since Algorithm 4 relies on a subroutine solving Monomial-Coefficient it cannot be generalized to degree 2 polynomials unless RP = NP or equivalently P = NP.

8.7 Depth-2 formula, unbounded degree

Here we consider depth-2 formulas but we drop all restrictions on the degree of the computed polynomials. We try to determine if the considered problems are still hard on this class of polynomials on which for instance PIT is solvable in deterministic polynomial time.

A polynomial represented by a depth-2 formula whose top gate is labeled by a + has only a number of monomials linear in the size of the formula. Therefore it can be interpolated in polynomial time and from its explicit representation by monomials it is easy to solve Non-Zero-Monomial, Monomial-Coefficient and Monomial-Factor. Therefore in the next proofs we always assume that the top gate is labeled by $\times$.

The problem Non-Zero-Monomial is in P over depth-2 formulas.

Let $P = \prod_{i=1}^{k} T_i$ where T_i is a sum of variables in $\{X_1, \ldots, X_n\}$. We reduce Non-Zero-Monomial to the problem of deciding if there is a perfect matching in a graph. To P and the term $\vec{X}^{\vec{e}}$, we associate the bipartite graph $G = (V, E)$ defined by:

1. $V = \{u_1, \ldots, u_k\} \cup \{v_j^l \mid j \in [n] \text{ and } l \leq e_j\}$
2. $E = \{(u_i, v_j^l) \mid X_j \text{ has a non-zero coefficient in } T_i\}$

A vertex u_i represents the linear form T_i, while the vertices $v_j^1 \ldots, v_j^{e_j}$ represent the variable X_j (e_j is its degree in $\vec{X}^{\vec{e}}$). There is an edge (u_i, v_j^l) if X_j has a coefficient different from zero in T_i. A perfect matching M of G corresponds to the choice of one variable $X_{\alpha(i)}$ for each T_i. If we expand the product in the definition of P, we obtain the term $\prod_i X_{\alpha(i)}$. Since all terms obtained by expansion have positive coefficients, they do not cancel out and $\prod_i X_{\alpha(i)}$ has a coefficient different from zero in P. All vertices v_j^l are saturated by M, because it is a perfect matching. It means that $\prod_i X_{\alpha(i)} = \vec{X}^{\vec{e}}$.

Conversely, and for the same reasons, the term $\vec{X}^{\vec{e}}$ has a non zero coefficient in P, if there is a perfect matching in G. This proves that Non-Zero-Monomial is reducible to the problem of finding a perfect matching in a bipartite graph, which is in P.

By a slight modification of the reduction, we could prove that Monomial-Factor is in P over depth-2 formulas. We now show that a slight generalization of the class of considered polynomials makes the problem Non-Zero-Monomial hard.

The problem Non-Zero-Monomial is NP-hard over the polynomials represented by depth-2 formulas where the input gates may be labeled by a variable times a constant in $\{-1, 0, 1, 2, 3\}$.

Let M be a $n \times n$ matrix with coefficients in $-1, 0, 1, 2, 3$. We substitute $M_{i,j}$ to $X_{i,j}$ in the polynomial Q introduced at Section 8.3. Its monomial $\prod_{j=1}^{n} Y_j$ has for coefficients the permanent of the matrix M. Deciding whether a matrix with coefficients in $\{-1, 0, 1, 2, 3\}$ has a permanent 0 is NP-hard [45]. Thus Non-Zero-Monomial is also NP-hard.

Finally, the evaluation of the Permanent is #P-complete for Turing reduction, even for matrices with coefficients in $\{0, 1\}$, which yields the next proposition.

The problem Monomial-Coefficient is #P-complete over depth 2 circuits.

9 Conclusion

As a conclusion, we recall the complexity of our three interpolation algorithms and we compare them to three methods of the literature. In Ben-Or/Tiwari and Zippel algorithms a bound on t, the number of monomials, must be explicitly given, we denote it by T. In the row labeled Enumeration is written the enumeration complexity of the interpolation method when applied to a polynomial computable in polynomial time.

Remark that Algorithm 6 given by Theorem 6 has a better total complexity and delay than KS algorithm when the degree is less or equal to 10.

	Ben-Or/Tiwari [8]	Zippel [49]	KS [29]
Algorithm type	Deterministic	Probabilistic	Probabilistic
Number of calls	$2T$	tnD	tn^7D^4
Total time	Quadratic in T	Quadratic in t	Quadratic in t
Enumeration	Exponential	Polynomial total time	Incremental delay
Size of points	$T\log(n)$	$\log(nT^2\epsilon^{-1})$	$\log(nD\epsilon^{-1})$

	Theorem 4	Theorem 5	Theorem 6
Algorithm type	Probabilistic	Probabilistic	Probabilistic
Number of calls	$tn(n+\log(\epsilon^{-1}))$	$tnD(n+\log(\epsilon^{-1}))$	$tnD^{d-1}(n+\log(\epsilon^{-1}))$
Total time	Quadratic in t	Linear in t	Quadratic in t
Enumeration	Incremental delay	Polynomial delay	Incremental delay
Size of points	$\log(D)$	$\log(D)$	$\log(D)$
Restriction	$\mathcal{DS}$	Multilinear	Fixed degree

Figure 2: Classical and new interpolation algorithms

Open question: is it possible to turn Algorithm 6 into a fixed parameter algorithm? That is to reduce the number of calls to $O(n^aD^bf(d))$ with a and b small but f increasing exponentially fast or worse. In this case, the interpolation algorithm obtained would be better than KS algorithm for any fixed d.
Open question: is there an interesting class of *non-multilinear* polynomials which can be interpolated with polynomial delay? What about the class of (degree 2) polynomials computed by a read-twice formula ?

Finally, note that the speed of the different interpolation algorithms often depends on the degree of the interpolated polynomial. Therefore we should compute it before choosing the interpolation algorithm. When the total degree is polynomially bounded it can be easily done: using the Algorithm of Proposition 2 we compute the degree in each variable with high probability.

However the complexity of finding the degree of a polynomial of unbounded total degree is an open question even for univariate polynomials. The best upper bound for polynomials given by circuits is $\mathrm{coRP}^{\mathrm{PP}}$ [27] but the problem could well be in P. Remark that the problem to find a lexicographically maximal monomial in a multivariate polynomial of degree d can be reduced to the problem of finding the degree by mapping X_i to X^{d^i}. Thus if there is a polynomial time algorithm to compute the degree, we could enumerate the monomials in lexicographic order with an incremental delay. We would then obtain an enumeration algorithm in a given order, in the spirit of [23] which gives an enumeration algorithm for the maximal independent sets in lexicographic order.

Thanks to Hervé Fournier "l'astucieux", Guillaume Malod, Sylvain Perifel and Arnaud Durand for their helpful comments about this article. The conversations I had with Meena Mahajan led to improvements of the last section of this article. I would also like to thank an anonymous referee of the conference version of this article for his good suggestions.

References

[1] Agrawal, M., Kayal, N., Saxena, N.: PRIMES is in P. Annals of Mathematics **160**(2), 781–793 (2004)

[2] Agrawal, M., Saha, C., Saptharishi, R., Saxena, N.: Jacobian hits circuits: hitting-sets, lower bounds for depth-d occur-k formulas & depth-3 transcendence degree-k circuits. In: STOC, pp. 599–614 (2012)

[3] Aigner, M.: A course in enumeration. Springer Verlag (2007)

[4] Anderson, M., van Melkebeek, D., Volkovich, I.: Derandomizing polynomial identity testing for multilinear constant-read formulae. In: Proceedings of the 26rd Annual IEEE Conference on Computational Complexity, CCC (2011)

[5] Arora, S., Barak, B.: Computational Complexity: A Modern Approach. Cambridge University Press (2009)

[6] Arvind, V., Joglekar, P.S.: Arithmetic circuit size, identity testing, and finite automata. Electronic Colloquium on Computational Complexity (ECCC) **16**, 26 (2009)

[7] Bagan, G.: Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques. Ph.D. thesis, Université de Caen (2009)

[8] Ben-Or, M.: A deterministic algorithm for sparse multivariate polynomial interpolation. In: Proceedings of the 20th annual ACM symposium on Theory of computing, pp. 301–309. ACM New York, NY, USA (1988)

[9] Bürgisser, P.: Completeness and reduction in algebraic complexity theory, vol. 7. Springer Verlag (2000)

[10] Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. Journal of symbolic computation **9**(3), 251–280 (1990)

[11] Courcelle, B.: Linear delay enumeration and monadic second-order logic. Discrete Applied Mathematics **157**(12), 2675–2700 (2009)

[12] DeMillo, R.: A probabilistic remark on algebraic program testing. Tech. rep., DTIC Document (1977)

[13] Durand, A., Grandjean, E.: First-order queries on structures of bounded degree are computable with constant delay. ACM Transactions on Computational Logic (TOCL) **8**(4), 21–es (2007)

[14] Durand, A., Strozecki, Y.: Enumeration complexity of logical query problems with second-order variables. In: Proceedings of the 20th Conference on Computer Science Logic, pp. 189–202 (2011)

[15] Duris, D., Strozecki, Y.: The complexity of acyclic subhypergraph problems. Workshop on Algorithms and Computation pp. 45–56 (2011)

[16] Flum, J., Grohe, M.: Parameterized complexity theory. Springer-Verlag New York Inc (2006)

[17] Fournier, H., Malod, G., Mengel, S.: Monomials in arithmetic circuits: Complete problems in the counting hierarchy. In: STACS, pp. 362–373 (2012)

[18] Garey, M., Johnson, D.: Computers and intractability: a guide to NP-completeness. WH Freeman and Company, San Francisco (1979)

[19] Garg, S., Schost, É.: Interpolation of polynomials given by straight-line programs. Theoretical Computer Science **410**(27-29), 2659–2662 (2009)

[20] Goldberg, L.: Listing graphs that satisfy first-order sentences. Journal of Computer and System Sciences **49**(2), 408–424 (1994)

[21] Ibarra, O., Moran, S.: Probabilistic algorithms for deciding equivalence of straight-line programs. Journal of the ACM (JACM) **30**(1), 217–228 (1983)

[22] Jerrum, M.: Counting, sampling and integrating: algorithms and complexity. Birkhäuser (2003)

[23] Johnson, D.S., Papadimitriou, C.H., Yannakakis, M.: On generating all maximal independent sets. Information Processing Letters **27**(3), 119–123 (1988)

[24] Kaltofen, E., Koiran, P.: Expressing a fraction of two determinants as a determinant. In: Proceedings of the 21st international symposium on Symbolic and algebraic computation, pp. 141–146. ACM (2008)

[25] Kaltofen, E., Lee, W., Lobo, A.: Early termination in Ben-Or/Tiwari sparse interpolation and a hybrid of Zippel's algorithm. In: Proceedings of the 2000 international symposium on Symbolic and algebraic computation, pp. 192–201. ACM New York, NY, USA (2000)

[26] Karnin, Z., Mukhopadhyay, P., Shpilka, A., Volkovich, I.: Deterministic identity testing of depth-4 multilinear circuits with bounded top fan-in. In: Proceedings of the 42nd ACM symposium on Theory of computing, pp. 649–658. ACM (2010)

[27] Kayal, N., Saha, C.: On the sum of square roots of polynomials and related problems. In: IEEE Conference on Computational Complexity, pp. 292–299 (2011)

[28] Kiefer, S., Murawski, A., Ouaknine, J., Wachter, B., Worrell, J.: Language equivalence for probabilistic automata. In: Computer Aided Verification, pp. 526–540. Springer (2011)

[29] Klivans, A., Spielman, D.: Randomness efficient identity testing of multivariate polynomials. In: Proceedings of the 33rd annual ACM symposium on Theory of computing, pp. 216–223. ACM New York, NY, USA (2001)

[30] Koutis, I., Williams, R.: Limits and applications of group algebras for parameterized problems. Automata, Languages and Programming pp. 653–664 (2009)

[31] Lovász, L.: Matroid matching and some applications. J. Combin. Theory Ser. B **28**(2), 208–236 (1980)

[32] Mahajan, M., Rao, B.V.R., Sreenivasaiah, K.: Identity testing, multilinearity testing, and monomials in read-once/twice formulas and branching programs. In: MFCS, pp. 655–667 (2012)

[33] Malod, G., Portier, N.: Characterizing Valiant's algebraic complexity classes. Journal of complexity **24**(1), 16–38 (2008)

[34] Masbaum, G., Vaintrob, A.: A new matrix-tree theorem. International Mathematics Research Notices **2002**(27), 1397 (2002)

[35] Mulmuley, K., Vazirani, U., Vazirani, V.: Matching is as easy as matrix inversion. In: Proceedings of the 19th annual ACM symposium on Theory of computing, pp. 345–354. ACM (1987)

[36] Plesník, J.: The NP-completeness of the hamiltonian cycle problem in planar digraphs with degree bound two. Information Processing Letters **8**(4), 199–201 (1979)

[37] Pruesse, G., Ruskey, F.: Generating linear extensions fast. SIAM Journal on Computing **23**(2), 373–386 (1994)

[38] Saraf, S., Volkovich, I.: Black-box identity testing of depth-4 multilinear circuits. In: Proceedings of the 43rd ACM Symposium on Theory of Computing (STOC), pp. 421–430 (2011)

[39] Saxena, N., Seshadhri, C.: An almost optimal rank bound for depth-3 identities. SIAM journal on computing **40**(1), 200–224 (2011)

[40] Schwartz, J.: Fast probabilistic algorithms for verification of polynomial identities. Journal of the ACM (JACM) **27**(4), 717 (1980)

[41] Strozecki, Y.: Enumeration complexity and matroid decomposition. Ph.D. thesis, Université Paris Diderot - Paris 7 (2010)

[42] Strozecki, Y.: Enumeration of the monomials of a polynomial and related complexity classes. Mathematical Foundations of Computer Science pp. 629–640 (2010)

[43] Toda, S.: Classes of arithmetic circuits capturing the complexity of computing the determinant. IEICE Transactions on Information and Systems **75**(1), 116–124 (1992)

[44] Uno, T.: Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. Algorithms and Computation pp. 92–101 (1997)

[45] Valiant, L.: The complexity of computing the permanent. Theoretical computer science **8**(2), 189–201 (1979)

[46] Von Zur Gathen, J.: Feasible arithmetic computations: Valiant's hypothesis. Journal of Symbolic Computation **4**(2), 137–172 (1987)

[47] Williams, V.: Multiplying matrices faster than coppersmith-winograd. In: Proceedings of the 44th symposium on Theory of Computing, pp. 887–898. ACM (2012)

[48] Zippel, R.: Probabilistic algorithms for sparse polynomials. Symbolic and algebraic computation pp. 216–226 (1979)

[49] Zippel, R.: Interpolating polynomials from their values. Journal of Symbolic Computation **9**(3), 375–403 (1990)