Monomials in arithmetic circuits: Complete problems in the counting hierarchy

Hervé Fournier and Guillaume Malod
Univ Paris Diderot, Sorbonne Paris Cité,
Institut de Mathématiques de Jussieu, UMR 7586 CNRS,
F-75205 Paris, France
{fournier,malod}@math.univ-paris-diderot.fr

Stefan Mengel*
Institute of Mathematics
University of Paderborn
D-33098 Paderborn, Germany
smengel@mail.uni-paderborn.de

November 16, 2021

We consider the complexity of two questions on polynomials given by arithmetic circuits: testing whether a monomial is present and counting the number of monomials. We show that these problems are complete for subclasses of the counting hierarchy which had few or no known natural complete problems before. We also study these questions for circuits computing multilinear polynomials.

1. Introduction

Several recent papers in arithmetic circuit complexity refer to a family of classes called the counting hierarchy consisting of the classes $\mathsf{PP} \cup \mathsf{PP}^{\mathsf{PP}} \cup \mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}} \cup \ldots$. For example, Bürgisser [6] uses these classes to connect computing integers to computing polynomials, while Jansen and Santhanam [13] — building on results by Koiran and Perifel [17] — use them to derive lower bounds from derandomization. This hierarchy was originally introduced by Wagner [31] to classify the complexity of combinatorial problems. Curiously, after Wagner's paper and another by Torán [26], this original motivation of the counting hierarchy has to the best of our knowledge not been pursued for more than twenty years. Instead, research focused on structural properties and the connection to threshold circuits [3]. As a result, there are very few natural complete problems for classes in the counting hierarchy: for instance, Kwisthout

*Partially supported by DFG grants BU 1371/2-2 and BU 1371/3-1.

et al. give in [19] "the first problem with a practical application that is shown to be $\mathsf{FP}^{\mathsf{PP}^{\mathsf{PP}}}$-complete". The related class $\mathsf{C_=P}$ appears to have no natural complete problems at all (see [12, p. 293]). It is however possible to define generic complete problems by starting with a #P-complete problem and considering the variant where an instance and a positive integer are provided and the question is to decide whether the number of solutions for this instance is equal to the integer. We consider these problems to be counting problems disguised as decision problems and thus not as natural complete problems for $\mathsf{C_=P}$, in contrast to the questions studied here. Note that the corresponding logspace counting class $\mathsf{C_=L}$ is known to have interesting complete problems from linear algebra [1].

In this paper we follow Wagner's original idea and show that the counting hierarchy is a helpful tool to classify the complexity of several natural problems on arithmetic circuits by showing complete problems for the classes $\mathsf{PP}^{\mathsf{PP}}$, $\mathsf{PP}^{\mathsf{NP}}$ and $\mathsf{C_=P}$.[1] The common setting of these problems is the use of circuits or straight-line programs to represent polynomials. Such a representation can be much more efficient than giving the list of monomials, but common operations on polynomials may become more difficult. An important example is the question of determining whether the given polynomial is identically zero. This is easy to do when given a list of monomials. When the polynomial is given as a circuit, the problem, called ACIT for *arithmetic circuit identity testing*, is solvable in coRP but is not known to be in P. In fact, derandomizing this problem would imply circuit lower bounds, as shown in [14]. This question thus plays a crucial part in complexity and it is natural to consider other problems on polynomials represented as circuits. In this article we consider mainly two questions.

The first question, called ZMC for *zero monomial coefficient*, is to decide whether a given monomial in a circuit has coefficient 0 or not. This problem has already been studied by Koiran and Perifel [16]. They showed that when the formal degree of the circuit is polynomially bounded the problem is complete for $\mathsf{P}^{\#\mathsf{P}}$. Unfortunately this result is not fully convincing, because it is formulated with the rather obscure notion of strong nondeterministic Turing reductions. We remedy this situation by proving a completeness result for the class $\mathsf{C_=P}$ under more traditional logarithmic space many-one reductions. This provides a natural complete problem for this class. Koiran and Perifel also considered the general case of ZMC, where the formal degree of the circuits is not bounded. They showed that ZMC is in CH. We provide a better upper bound by proving that ZMC is in $\mathsf{coRP}^{\mathsf{PP}}$. We finally study the case of monotone circuits and show that the problem is then coNP-complete.

The second problem is to count the number of monomials in the polynomial computed by a circuit. This seems like a natural question whose solution should not be too hard, but in the general case it turns out to be $\mathsf{PP}^{\mathsf{PP}}$-complete, and the hardness holds even for weak circuits. We thus obtain another natural complete problem, in this case for the second level of the counting hierarchy.

Finally, we study the two above problems in the case of circuits computing multilinear polynomials. We show that our first problem becomes equivalent to the fundamental problem ACIT and that counting monomials becomes PP-complete.

[1] Observe that Hemaspaandra and Ogihara [12, p. 293] state that Mundhenk et al. [23] provide natural complete problems for $\mathsf{PP}^{\mathsf{NP}}$. This appears to be a typo as Mundhenk et al. in fact present complete problems not for $\mathsf{PP}^{\mathsf{NP}}$ but for the class $\mathsf{NP}^{\mathsf{PP}}$ which indeed appears to have several interesting complete problems in the AI/planning literature.

2. Preliminaries

Complexity classes We assume the reader to be familiar with basic concepts of computational complexity theory (see e.g. [4]). All reductions in this paper will be logspace many-one unless stated otherwise.

We consider different counting decision classes in the counting hierarchy [31]. These classes are defined analogously to the quantifier definition of the polynomial hierarchy but, in addition to the quantifiers $\exists$ and $\forall$, the quantifiers C, $\mathsf{C}_=$ and $\mathsf{C}_{\neq}$ are used.

Definition 2.1. Let $\mathcal{C}$ be a complexity class.

- $A \in \mathsf{C}\mathcal{C}$ if and only if there is $B \in \mathcal{C}$, $f \in \mathsf{FP}$ and a polynomial p such that

$$x \in A \Leftrightarrow \left|\left\{y \in \{0,1\}^{p(|x|)} \mid (x,y) \in B\right\}\right| \geq f(x),$$

- $A \in \mathsf{C}_=\mathcal{C}$ if and only if there is $B \in \mathcal{C}$, $f \in \mathsf{FP}$ and a polynomial p such that

$$x \in A \Leftrightarrow \left|\left\{y \in \{0,1\}^{p(|x|)} \mid (x,y) \in B\right\}\right| = f(x),$$

- $A \in \mathsf{C}_{\neq}\mathcal{C}$ if and only if there is $B \in \mathcal{C}$, $f \in \mathsf{FP}$ and a polynomial p such that

$$x \in A \Leftrightarrow \left|\left\{y \in \{0,1\}^{p(|x|)} \mid (x,y) \in B\right\}\right| \neq f(x).$$

Observe that $\mathsf{C}_{\neq}\mathcal{C} = \mathsf{coC}_=\mathcal{C}$ with the usual definition $\mathsf{co}\mathcal{C} = \{L^c \mid L \in \mathcal{C}\}$, where L^c is the complement of L. That is why the quantifier $\mathsf{C}_{\neq}$ is often also written as $\mathsf{coC}_=$, so $\mathsf{C}_{\neq}\mathsf{P}$ is sometimes called $\mathsf{coC}_=\mathsf{P}$.

The counting hierarchy CH consists of the languages from all classes that we can get from P by applying the quantifiers $\exists$, $\forall$, C, $\mathsf{C}_=$ and $\mathsf{C}_{\neq}$ a constant number of times. Observe that with the definition above $\mathsf{PP} = \mathsf{CP}$. Torán [27] proved that this connection between PP and the counting hierarchy can be extended and that there is a characterization of CH by oracles similar to that of the polynomial hierarchy. We state some such characterizations which we will need later on, followed by other technical lemmas.

Lemma 2.2. *[27]* $\mathsf{PP}^{\mathsf{NP}} = \mathsf{C}\exists\mathsf{P}$.

Lemma 2.3. $\mathsf{PP}^{\mathsf{PP}} = \mathsf{CC}_{\neq}\mathsf{P}$.

Proof of Lemma 2.3. This is not stated in [27] nor is it a direct consequence, because Torán does not consider the $\mathsf{C}_{\neq}$-operator. It can be shown with similar techniques and we give a proof for completeness. We show that $\mathsf{CC}_{\neq}\mathsf{P} = \mathsf{CCP}$, the claim then follows, because $\mathsf{CCP} = \mathsf{PP}^{\mathsf{PP}}$ by [27].

The direction from left to right is straightforward: From the definition we have $\mathsf{CC}_{\neq}\mathsf{P} \subseteq \mathsf{PP}^{\#\mathsf{P}}$. By binary search we have $\mathsf{PP}^{\#\mathsf{P}} = \mathsf{PP}^{\mathsf{PP}} = \mathsf{CCP}$. The other direction is a little more work: Let $L \in \mathsf{CCP}$. There are $A \in \mathsf{P}$,$f, g \in \mathsf{FP}$ and a polynomial p such that

$$\begin{aligned} x \in L \quad &\Leftrightarrow \quad \text{there are more than } f(x) \text{ values } y \in \{0,1\}^{p(|x|)} \text{ such that} \\ &\qquad\quad \left|\left\{z \in \{0,1\}^{p(|x|)} \mid (x,y,z) \in A\right\}\right| \geq g(x,y) \\ &\Leftrightarrow \quad \text{there are more than } f(x) \text{ values } y \in \{0,1\}^{p(|x|)} \text{ such that} \\ &\qquad\quad \forall v \in \{1,\ldots,2^{p(|x|)}\}\colon \left|\left\{z \in \{0,1\}^{p(|x|)} \mid (x,y,z) \in A\right\}\right| \neq g(x,y) - v \qquad (1) \\ &\Leftrightarrow \quad \text{there are more than } 2^{p(|x|)}(2^{p(|x|)}-1) + f(x) \text{ pairs } (x,v) \text{ with} \\ &\qquad\quad y \in \{0,1\}^{p(|x|)} \text{ and } v \in \{1,\ldots,2^{p(|x|)}]\} \text{ such that} \\ &\qquad\quad \left|\left\{z \in \{0,1\}^{p(|x|)} \mid (x,y,z) \in A\right\}\right| \neq g(x,y) - v. \qquad (2) \end{aligned}$$

From statement (2) we directly get $L \in \mathsf{CC}_{\neq}\mathsf{P}$ and thus the claim. To see the last equivalence we define $r(x,y) := \left|\left\{z \in \{0,1\}^{p(|x|)} \mid (x,y,z) \in A\right\}\right|$. Fix x, y, then obviously $r(x,y) \neq g(x,y) - v$ for all but at most one v. It follows that of the pairs (y,v) in the last statement $2^{p(|x|)}(2^{p(|x|)}-1)$ always lead to inequality. So statement (2) boils down to the question how many y there are such that there is no v with $r(x,y) = g(x,y) - v$. We want these to be at least $f(x)$, so we want at least $2^{p(|x|)}(2^{p(|x|)}-1) + f(x)$ pairs such that $r(x,y) \neq g(x,y) - v$. □

Lemma 2.4. *[10]* $\exists\mathsf{C}_{\neq}\mathsf{P} = \mathsf{C}_{\neq}\mathsf{P}$.

Lemma 2.5. *[24] For a large enough constant $c > 0$, it holds that for any integers n and x with $|x| \leqslant 2^{2^n}$ and $x \neq 0$, the number of primes p smaller than 2^{cn} such that $x \not\equiv 0 \mod p$ is at least $2^{cn}/cn$.*

Lemma 2.6. *[12, p. 81] For every oracle X we have $\mathsf{PP}^{\mathsf{BPP}^X} = \mathsf{PP}^X$.*

Arithmetic circuits An *arithmetic circuit* is a labeled directed acyclic graph (DAG) consisting of vertices or gates with indegree or fanin 0 or 2. The gates with fanin 0 are called input gates and are labeled with -1 or variables $X_1, X_2, \ldots, X_n$. The gates with fanin 2 are called computation gates and are labeled with $\times$ or $+$. We can also consider circuits where computation gates may receive more than two edges, in which case we say that they have *unbounded fanin.* The polynomial computed by an arithmetic circuit is defined in the obvious way: an input gate computes the value of its label, a computation gate computes the product or the sum of its children's values, respectively. We assume that a circuit has only one sink which we call the output gate. We say that the polynomial computed by the circuit is the polynomial computed by the output gate. The *size* of an arithmetic circuit is the number of gates. The *depth* of a circuit is the length of the longest path from an input gate to the output gate in the circuit. A formula is an arithmetic circuit whose underlying graph is a tree. Finally, a circuit or formula is called *monotone* if, instead of the constant -1, only the constant 1 is allowed.

It is common to consider so-called *degree-bounded* arithmetic circuits, for which the degree of the computed polynomial is bounded polynomially in the number of gates of the circuit. In our opinion this kind of degree bound has two problems. One is that computing the degree of a polynomial represented by a circuit is suspected to be hard (see [2, 16, 15]), so problems defined with this degree bound must often be promise problems. The other problem is that the bound on the degree does not bound the size of computed constants, which by iterative squaring can have exponential bitsize. Thus even evaluating circuits on a Turing machine becomes intractable. The paper by Allender et al. [2] discusses problems that result from this. To avoid all these complications, instead of bounding the degree of the computed polynomial, we choose to bound the formal degree of the circuit or equivalently to consider multiplicatively disjoint circuits. A circuit is called *multiplicatively disjoint* if, for each ×-gate, its two input subcircuits are disjoint from one another. See [22] for a discussion of degree, formal degree and multiplicative disjointness and how they relate.

3. Zero monomial coefficient

We first consider the question of deciding if a single specified monomial occurs in a polynomial. In this problem and others regarding monomials, a monomial is encoded by giving the variable powers in binary.

ZMC
Input: Arithmetic circuit C, monomial m.
Problem: Decide if m has the coefficient 0 in the polynomial computed by C.

Theorem 3.1. ZMC *is* $\mathsf{C_=P}$*-complete for both multiplicatively disjoint circuits and formulas.*

Proof. Using standard reduction techniques from the #P-completeness of the permanent (see for example [4]), one define the following generic $\mathsf{C_=P}$-complete problem, as mentioned in the introduction.

$\text{PER}_=$
Input: Matrix $A \in \{0, 1, -1\}^n$, $d \in \mathbb{N}$.
Problem: Decide if $\text{PER}(A) = d$.

Therefore, for the hardness of ZMC it is sufficient to show a reduction from $\text{PER}_=$. We use the following classical argument. On input $A = (a_{ij})$ and d we compute the formula $Q := \prod_{i=1}^{n} \left(\sum_{j=1}^{n} a_{ij} Y_j \right)$. It is a classical observation by Valiant [28][2] that the monomial $Y_1 Y_2 \dots Y_n$ has the coefficient $\text{PER}(A)$. Thus the coefficient of the monomial $Y_1 Y_2 \dots Y_n$ in $Q - dY_1 Y_2 \dots Y_n$ is 0 if and only if $\text{PER}(A) = d$.

We now show that ZMC for multiplicatively disjoint circuits is in $\mathsf{C_=P}$. The proof is based on the use of parse trees, which can be seen as objects tracking the formation of monomials during the computation [22] and are the algebraic analog of proof trees [29]. A brief description is given in Appendix A.

Consider a multiplicatively disjoint circuit C and a monomial m, where the input gates of C are labeled either by a variable or by -1. A parse tree T contributes to the monomial m in the output polynomial if, when computing the value of the tree, we get exactly the powers in m; this contribution has coefficient $+1$ if the number of gates labeled -1 in T is even and it has coefficient -1 if this number is odd. The coefficient of m is thus equal to 0 if and only if the number of trees contributing positively is equal to the number of trees contributing negatively.

[2] According to [30] this observation even goes back to [11].

Let us represent a parse tree by a boolean word $\bar{\epsilon}$, by indicating which edges of C appear in the parse tree (the length N of the words is therefore the number of edges in C). Some of these words will not represent a valid parse tree, but this can be tested in polynomial time. Consider the following language L composed of triples $(C, m, \epsilon_0\bar{\epsilon})$ such that:

1. $\epsilon_0 = 0$ and $\bar{\epsilon}$ encodes a valid parse tree of C which contribute positively to m,

2. or $\epsilon_0 = 1$ and $\bar{\epsilon}$ does not encode a valid parse tree contributing negatively to m.

Then the number of $\bar{\epsilon}$ such that $(C, m, 0\bar{\epsilon})$ belongs to L is the number of parse trees contributing positively to m and the number of $\bar{\epsilon}$ such that $(C, m, 1\bar{\epsilon})$ belongs to L is equal to 2^N minus the number of parse trees contributing negatively to m. Thus, the number of $\epsilon_0\bar{\epsilon}$ such that $(C, m, \epsilon_0\bar{\epsilon}) \in L$ is equal to 2^N if and only if the number of trees contributing positively is equal to the number of trees contributing negatively, if and only if the coefficient of m is equal to 0 in C. Because L is in P, ZMC for multiplicatively disjoint circuits is in $\mathsf{C_{=}P}$. □

Theorem 3.2. ZMC *belongs to* $\mathsf{coRP}^{\mathsf{PP}}$.

Proof. Given a circuit C, a monomial m and a prime number p written in binary, CoeffSLP is the problem of computing modulo p the coefficient of the monomial m in the polynomial computed by C. It is shown in [15] that CoeffSLP belongs to $\mathsf{FP}^{\#\mathsf{P}}$.

We now describe a randomized algorithm to decide ZMC. Let c be the constant given in Lemma 2.5. Consider the following algorithm to decide ZMC given a circuit C of size n and a monomial m, using CoeffSLP as an oracle. First choose uniformly at random an integer p smaller than 2^{cn}. If p is not prime, accept. Otherwise, compute the coefficient a of the monomial m in C with the help of the oracle and accept if $a \equiv 0 \mod p$. Since $|a| \leq 2^{2^n}$, Lemma 2.5 ensures that the above is a correct one-sided error probabilistic algorithm for ZMC. This yields ZMC $\in \mathsf{coRP}^{\text{CoeffSLP}}$. Hence ZMC $\in \mathsf{coRP}^{\mathsf{PP}}$. □

Theorem 3.3. ZMC *is* coNP*-complete both for monotone formulas and monotone circuits.*

Proof. For hardness, we reduce the NP-complete problem Exact-3-Cover [9] to the complement of ZMC on monotone formulas, as done in [25, Chapter 3] (we reproduce the argument here for completeness).

Exact-3-Cover
Input: Integer n and $C_1, \dots, C_m$ some 3-subsets of $\{1, \dots, n\}$.
Problem: Decide if there exists $I \subseteq \{1, \dots, m\}$ such that $\{C_i \mid i \in I\}$ is a partition of $\{1, \dots, n\}$.

Consider the formula $F = \prod_{i=1}^{m}(1 + \prod_{j \in C_i} X_j)$. The monotone formula F has the monomial $\prod_{i=1}^{n} X_i$ if and only if $(n, C_1, \dots, C_m)$ is a positive instance of Exact-3-Cover.

Let us now show that ZMC for monotone circuits is in coNP. This proof will use the notion of *parse tree types*, which are inspired by the generic polynomial introduced in [21] to compute coefficient functions. We give here a sketch of the argument, more details are provided in Appendix A. The parse trees of a circuit which is not necessarily multiplicatively disjoint may be of a much bigger size than the circuit itself, because they can be seen as parse trees of the formula associated to the circuit and obtained by duplicating gates and edges. Define the *type* of a parse tree by giving, for each edge in the original circuit, the number of copies of this edge in the parse tree. There can be many different parse trees for a given parse tree type but they will all contribute to the same monomial, which is easy to obtain from the type: the power of a variable in the monomial is the sum, taken over all input gates labeled by this variable, of the number of edges leaving from this gate. In the case of a monotone circuit, computing the exact number of parse trees for a given type is thus not necessary, as a monomial will have a non-zero coefficient if and only if there exists a valid parse tree type producing this monomial.

Parse tree types, much like parse trees in the proof of Theorem 3.1, can be represented by Boolean tuples which must satisfy some easy-to-check conditions to be valid. Thus the coefficient of a monomial is 0 if and only if there are no valid parse tree types producing this monomial, which is a coNP condition. □

4. Counting monomials

We now turn to the problem of counting the monomials of a polynomial represented by a circuit.

CountMon
Input: Arithmetic circuit C, $d \in \mathbb{N}$.
Problem: Decide if the polynomial computed by C has at least d monomials.

To study the complexity of CountMon we will look at what we call extending polynomials. Given two monomials M and m, we say that M is m-extending if $M = mm'$ and m and m' have no common variable. We start by studying the problem of deciding the existence of an extending monomial.

ExistExtMon
Input: Arithmetic circuit C, monomial m.
Problem: Decide if the polynomial computed by C contains an m-extending monomial.

Proposition 4.1. ExistExtMon *is in* $\mathsf{RP}^{\mathsf{PP}}$. *For multiplicatively disjoint circuits it is* $\mathsf{C}_{\neq}\mathsf{P}$*-complete.*

Proof. We first show the first upper bound. So let (C, m) be an input for ExistExtMon where C is a circuit in the variables $X_1, \ldots, X_n$. Without loss of generality, suppose that $X_1, \ldots, X_r$ are the variables appearing in m. Let $d = 2^{|C|}$: d is a bound on the degree of the polynomial computed by C. We define $C' = \prod_{i=r+1}^{n}(1 + Y_iX_i)^d$ for new variables Y_i. We have that C has an m-extending monomial if and only if in the product CC' the polynomial $P(Y_{r+1}, \ldots, Y_n)$, which is the coefficient of $m \prod_{i=r+1}^{n} X_i^d$, is not identically 0. Observe that P is not given explicitly but can be evaluated modulo a random prime with an oracle for CoeffSLP. Thus it can be checked if P is identically 0 with the classical Schwartz-Zippel-DeMillo-Lipton lemma (see for example [4]). It follows that ExistExtMon $\in \mathsf{RP}^{\mathsf{PP}}$.

The upper bound in the multiplicatively disjoint setting is easier: we can guess an m-extending monomial M and then output the answer of an oracle for the complement of ZMC, to check whether M appears in the computed polynomial. This establishes containment in $\exists\mathsf{C}_{\neq}\mathsf{P}$ which by Lemma 2.4 is $\mathsf{C}_{\neq}\mathsf{P}$.

For hardness we reduce to ExistExtMon the $\mathsf{C}_{\neq}\mathsf{P}$-complete problem $\textsc{per}_{\neq}$, i.e., the complement of the $\textsc{per}_{=}$ problem introduced for the proof of Theorem 3.1. We use essentially the same reduction constructing a circuit $Q := \prod_{i=1}^{n}\left(\sum_{j=1}^{n} a_{ij}Y_j\right)$. Observe that the only potential extension of $m := Y_1Y_2\dots Y_n$ is m itself and has the coefficient $\textsc{per}(A)$. Thus $Q - dY_1Y_2\dots Y_n$ has an m-extension if and only if $\textsc{per}(A) \neq d$. □

CountExtMon
Input: Arithmetic circuit C, $d \in \mathbb{N}$, monomial m.
Problem: Decide if the polynomial computed by C has at least d m-extending monomials.

Proposition 4.2. CountExtMon *is* $\mathsf{PP}^{\mathsf{PP}}$*-complete.*

Proof. Clearly CountExtMon belongs to $\mathsf{PP}^{\mathrm{ZMC}}$ and thus with Theorem 3.2 it is in $\mathsf{PP}^{\mathsf{coRP}^{\mathsf{PP}}}$. Using Lemma 2.6 we get membership in $\mathsf{PP}^{\mathsf{PP}}$. To show hardness, we reduce the canonical $\mathsf{CC}_{\neq}\mathsf{P}$-complete problem $\mathsf{CC}_{\neq}$3SAT to CountExtMon. With Lemma 2.3 the hardness for $\mathsf{PP}^{\mathsf{PP}}$ follows.

$\mathsf{CC}_{\neq}$3SAT
Input: 3SAT-formula $F(\bar{x}, \bar{y})$, $k, \ell \in \mathbb{N}$.
Problem: Decide if there are at least k assignments to $\bar{x}$ such that there are not exactly ℓ assignments to $\bar{y}$ such that F is satisfied.

Let $(F(\bar{x}, \bar{y}), k, \ell)$ be an instance for $\mathsf{CC}_{\neq}$3SAT. Without loss of generality we may assume that $\bar{x} = (x_1, \dots, x_n)$ and $\bar{y} = (y_1, \dots, y_n)$ and that no clause contains a variable in both negated and unnegated form. Let $\Gamma_1, \dots, \Gamma_c$ be the clauses of F.

For each literal u of the variables in $\bar{x}$ and $\bar{y}$ we define a monomial $I(u)$ in the variables $X_1, \dots, X_n, Z_1, \dots, Z_c$ in the following way:

$$I(x_i) = X_i \prod_{\{j \mid x_i \in \Gamma_j\}} Z_j, \qquad I(\neg x_i) = \prod_{\{j \mid \neg x_i \in \Gamma_j\}} Z_j,$$

$$I(y_i) = \prod_{\{j \mid y_i \in \Gamma_j\}} Z_j, \qquad I(\neg y_i) = \prod_{\{j \mid \neg y_i \in \Gamma_j\}} Z_j.$$

From these monomials we compute a formula C by

$$C := \prod_{i=1}^{n} \left(I(x_i) + I(\neg x_i)\right) \prod_{i=1}^{n} \left(I(y_i) + I(\neg y_i)\right). \tag{3}$$

We fix a mapping *mon* from the assignments of F to the monomials computed by C: Let $\bar{\alpha}$ be an assignment to $\bar{x}$ and $\bar{\beta}$ be an assignment to $\bar{y}$. We define $mon(\bar{\alpha}\bar{\beta})$ as the monomial obtained in the expansion of C by choosing the following terms. If $\alpha_i = 0$, choose $I(\neg x_i)$, otherwise choose $I(x_i)$. Similarly, if $\beta_i = 0$, choose $I(\neg y_i)$, otherwise choose $I(y_i)$.

The monomial $mon(\bar{\alpha}\bar{\beta})$ has the form $\prod_{i=1}^{n} X_i^{\alpha_i} \prod_{j=1}^{c} Z_j^{\gamma_j}$, where γ_j is the number of true literals in Γ_j under the assignment $\bar{\alpha}\bar{\beta}$. Then F is true under $\bar{\alpha}\bar{\beta}$ if and only if $mon(\bar{\alpha}\bar{\beta})$ has the factor $\prod_{j=1}^{c} Z_j$. Thus F is true under $\bar{\alpha}\bar{\beta}$ if and only if $mon(\bar{\alpha}\bar{\beta}) \prod_{j=1}^{c} \left(1 + Z_j + Z_j^2\right)$ has the factor $\prod_{i=1}^{n} X_i^{\alpha_i} \prod_{j=1}^{c} Z_j^3$. We set $C' = C \prod_{j=1}^{c} \left(1 + Z_j + Z_j^2\right)$.

Consider an assignment $\bar{\alpha}$ to $\bar{x}$. The coefficient of the monomial $\prod_{i=1}^{n} X_i^{\alpha_i} \prod_{j=1}^{c} Z_j^3$ in C' is the number of assignments $\bar{\beta}$ such that $\bar{\alpha}\bar{\beta}$ satisfies F. Thus we get

$$
\begin{aligned}
& (F(\bar{x}, \bar{y}), k, \ell) \in \mathsf{CC}_{\neq}3\mathrm{SAT} \\
\Leftrightarrow\ & \text{there are at least } k \text{ assignments } \bar{\alpha} \text{ to } \bar{x} \text{ such that the monomial } \prod_{i=1}^{n} X_i^{\alpha_i} \prod_{j=1}^{c} Z_j^3 \\
& \text{does not have coefficient } \ell \text{ in } C' \\
\Leftrightarrow\ & \text{there are at least } k \text{ assignments } \bar{\alpha} \text{ to } \bar{x} \text{ such that the monomial } \prod_{i=1}^{n} X_i^{\alpha_i} \prod_{j=1}^{c} Z_j^3 \\
& \text{occurs in } C'' := C' - \ell \prod_{i=1}^{n} (1 + X_i) \prod_{j=1}^{c} Z_j^3 \\
\Leftrightarrow\ & \text{there are at least } k \text{ tuples } \bar{\alpha} \text{ such that } C'' \text{ contains the monomial } \prod_{i=1}^{n} X_i^{\alpha_i} \prod_{j=1}^{c} Z_j^3 \\
\Leftrightarrow\ & C'' \text{ has at least } k\ (\prod_{j=1}^{c} Z_j^3)\text{-extending monomials.}
\end{aligned}
$$

□

Theorem 4.3. CountMon *is* $\mathsf{PP}^{\mathsf{PP}}$*-complete. It is* $\mathsf{PP}^{\mathsf{PP}}$*-hard even for unbounded fan-in formulas of depth* 4.

Proof. CountMon can be easily reduced to CountExtMon since the number of monomials of a polynomial is the number of 1-extending monomials. Therefore CountMon belongs to $\mathsf{PP}^{\mathsf{PP}}$.

To show hardness, it is enough to prove that instances of CountExtMon constructed in Proposition 4.2 can be reduced to CountMon in logarithmic space. The idea of the proof is that we make sure that the polynomial for which we count all monomials contains all monomials that are not m-extending. Thus we know how many non-m-extending monomials it contains and we can compute the number of m-extending monomials from the number of all monomials. We could use the same strategy to show in general that CountExtMon reduces to CountMon but by considering the instance obtained in the proof of Proposition 4.2 and analyzing the extra calculations below we get hardness for unbounded fanin formulas of depth 4.

So let (C'', k, m) be the instance of CountExtMon constructed in the proof of Proposition 4.2, with $m = \prod_{j=1}^{c} Z_j^3$. We therefore need to count the monomials computed by C'' which are of the form $f(X_1, \ldots, X_n) \prod_{j=1}^{c} Z_j^3$. The circuit C'' is multilinear in X, and the Z_j can only appear with powers in $\{0, 1, 2, 3, 4, 5\}$. So the non-m-extending monomials computed by C'' are all products of a multilinear monomial in the X_i and a monomial in the Z_j where at least one Z_j has a power in $\{0, 1, 2, 4, 5\}$. Fix j, then all monomials that are not m-extending because of Z_j are computed by the formula

$$\tilde{C}_j := \left(\prod_{i=1}^{n}(X_i+1)\right)\left(\prod_{j'\neq j}\sum_{p=0}^{5} Z_{j'}^{p}\right)\left(1+Z_j+Z_j^2+Z_j^4+Z_j^5\right). \qquad (4)$$

Thus the formula $\tilde{C} := \sum_j \tilde{C}_j$ computes all non-m-extending monomials that C'' can compute. The coefficients of monomials in C'' cannot be smaller than $-\ell$ where ℓ is part of the instance of $\mathsf{CC}_{\neq}$3SAT from which we constructed (C'', k, m) before. So the formula $C^* := C'' + (\ell+1)\tilde{C}$ contains all non-m-extending monomials that C'' can compute and it contains the same extending monomials. There are $2^n 6^c$ monomials of the form that C'' can compute, only 2^n of which are m-extending, which means that there are $2^n(6^c-1)$ monomials computed by C^* that are not m-extending. As a consequence, C'' has at least k m-extending monomials if and only if C^* has at least $2^n(6^c-1)+k$ monomials. □

Theorem 4.4. CountMon *is* $\mathsf{PP}^{\mathsf{NP}}$*-complete both for monotone formulas and monotone circuits.*

Proof. We first show hardness for monotone formulas. The argument is very similar to the proof of Theorem 4.3. Consider the following canonical $\mathsf{C}\exists\mathsf{P}$-complete problem $\mathsf{C}\exists$3SAT.

$\mathsf{C}\exists$3SAT
Input: 3SAT-formula $F(\bar{x},\bar{y})$, $k \in \mathbb{N}$.
Problem: Decide if there are at least k assignments $\bar{\alpha}$ to $\bar{x}$ such that $F(\bar{\alpha},\bar{y})$ is satisfiable.

We reduce $\mathsf{C}\exists$3SAT to CountMon. With Lemma 2.2 the hardness for $\mathsf{PP}^{\mathsf{NP}}$ follows. Consider a 3SAT-formula $F(\bar{x},\bar{y})$. Let $n = |\bar{x}| = |\bar{y}|$ and let c be the number of clauses of F. Define the polynomial $C^* = C + \sum_{j=1}^{c}\tilde{C}_j$ where C is defined by Equation 3 and $\tilde{C}_j$ by Equation 4. The analysis is similar to the proof of Theorem 4.3. The polynomial C^* is computed by a monotone arithmetic formula and has at least $2^n(6^c-1)+k$ monomials if and only if (F,k) is a positive instance of $\mathsf{C}\exists$3SAT.

We now prove the upper bound. Recall that CountMon $\in \mathsf{PP}^{\mathrm{ZMC}}$. From Theorem 3.3, it follows that CountMon on monotone circuits belongs to $\mathsf{PP}^{\mathsf{NP}}$. □

5. Multilinearity

In this section we consider the effect of multilinearity on our problems. We will not consider promise problems and therefore the multilinear variants of our problems must first check if the computed polynomial is multilinear. We start by showing that this step is not difficult.

CheckML
Input: Arithmetic circuit C.
Problem: Decide if the polynomial computed by C is multilinear.

Proposition 5.1. CheckML *is equivalent to* ACIT.

Proof. Reducing ACIT to CheckML is easy: Simply multiply the input with X^2 for an arbitrary variable X. The resulting circuit is multilinear if and only if the original circuit was 0.

For the other direction the idea is to compute the second derivatives of the polynomial computed by the input circuit and check if they are 0.

So let C be a circuit in the variables $X_1, \ldots, X_n$ that is to be checked for multilinearity. For each i we inductively compute a circuit C_i that computes the second derivative with respect to X_i. To do so for each gate v in C the circuit C' has three gates v_i, v_i' and v_i''. The polynomial in v_i is that of v, v_i' computes the first derivative and v_i'' the second. For the input gates the construction is obvious. If v is a $+$-gate with children u and w we have $v_i = u_i + w_i$, $v_i' = u_i' + w_i'$ and $v_i'' = u_i'' + w_i''$. If v is a $\times$-gate with children u and w we have $v_i = u_i w_i$, $v_i' = u_i' w_i + u_i w_i'$ and $v_i'' = u_i'' w_i + 2u_i' w_i' + u_i w_i''$. It is easy to see that the constructed circuit computes indeed the second derivative with respect to X_i.

Next we compute $C' := \sum_{i=1}^n Y_i C_i$ for new variables Y_i. We have that C' is identically zero if and only if C is multilinear. Also C' can easily be constructed in logarithmic space. □

Next we show that the problem gets much harder if, instead of asking whether *all* the monomials in the polynomial computed by a circuit are multilinear, we ask whether at least *one* of the monomials is multilinear.

MonML **Input:** Arithmetic circuit C. **Problem:** Decide if the polynomial computed by C contains a multilinear monomial.

The problem monML lies at the heart of fast exact algorithms for deciding k-paths by Koutis and Williams [18, 32] (although in these papers the polynomials are in characteristic 2 which changes the problem a little). This motivated Chen and Fu [7, 8] to consider monML, show that it is #P-hard and give algorithms for the bounded depth version. We provide further information on the complexity of this problem.

Proposition 5.2. MonML *is in* $\mathsf{RP}^{\mathsf{PP}}$. *It is* $\mathsf{C}_{\neq}\mathsf{P}$*-complete for multiplicatively disjoint circuits.*

Proof. For the first upper bound, let C be the input in variables $X_1, \ldots, X_n$. We set $C' = \prod_{i=1}^n (1 + X_i Y_i)$. Then C computes a multilinear monomial if and only if in the product CC' the coefficient polynomial $P(Y_1, \ldots, Y_n)$ of $\prod_{i=1}^n X_i$ is not identically 0. This can be tested as in the proof of Proposition 4.1, thus establishing MonML $\in \mathsf{RP}^{\mathsf{PP}}$.

The $\mathsf{C}_{\neq}\mathsf{P}$-completeness in the multiplicatively disjoint case can be proved in the same way as in Proposition 4.1. □

We now turn to our first problem, namely deciding whether a monomial appears in the polynomial computed by a circuit, in the multilinear setting.

ML-ZMC **Input:** Arithmetic circuit C, monomial m. **Problem:** Decide if C computes a multilinear polynomial in which the monomial m has coefficient 0.

Proposition 5.3. ML-ZMC *is equivalent to* ACIT.

Proof. We first show that ACIT reduces to ML-ZMC. So let C be an input for ACIT. Allender et al. [2] have shown that ACIT reduces to a restricted version of ACIT in which all inputs are -1 and thus the circuit computes a constant. Let C_1 be the result of this reduction. Then C computes identically 0 if and only if the constant coefficient of C_1 is 0. This establishes the first direction.

For the other direction let (C, m) be the input, where C is an arithmetic circuit and m is a monomial. First check if m is multilinear, if not output 1 or any other nonzero polynomial. Next we construct a circuit C_1 that computes the homogeneous component of degree $\deg(m)$ of C with the classical method (see for example [5, Lemma 2.14]). Observe that if C computes a multilinear polynomial, so does C_1. We now plug in 1 for the variables that appear in m and 0 for all other variables, call the resulting (constant) circuit C_2. If C_1 computes a multilinear polynomial, then C_2 is zero if and only if m has coefficient 0 in C_1. The end result of the reduction is $C^* := C_2 + ZC_3$ where Z is a new variable and C_3 is a circuit which is identically 0 iff C computes a multilinear polynomial (obtained via Proposition 5.1). C computes a multilinear polynomial and does not contain the monomial m if and only if both C_2 and ZC_3 are identically 0, which happens if and only if their sum is identically 0. □

In the case of our second problem, counting the number of monomials, the complexity falls to PP.

ML-CountMon
Input: Arithmetic circuit C, $d \in \mathbb{N}$.
Problem: Decide if the polynomial computed by C is multilinear and has at least d monomials.

Proposition 5.4. ML-CountMon *is* PP*-complete (for Turing reductions).*

Proof. We first show ML-CountMon $\in$ PP. To do so we use CheckML to check that the polynomial computed by C is multilinear. Then counting monomials can be done in $\mathsf{PP}^{\text{ML-ZMC}}$, and ML-ZMC is in coRP. By Lemma 2.6 the class $\mathsf{PP}^{\mathsf{coRP}}$ is simply PP.

For hardness we reduce the computation of the $\{0, 1\}$-permanent to ML-CountMon. The proposition follows, because the $\{0, 1\}$-permanent is #P-complete for Turing reductions. So let A be a 0-1-matrix and $d \in \mathbb{N}$ and we have to decide if $\text{PER}(A) \geq d$. We get a matrix B from A by setting $b_{ij} := a_{ij}X_{ij}$. Because every entry of B is either 0 or a distinct variable, we have that, when we compute the permanent of B, every permutation that yields a non-zero summand yields a unique monomial. This means that there are no cancellations, so that $\text{PER}(A)$ is the number of monomials in $\text{PER}(B)$.

The problem is now that no small circuits for the permanent are known and thus $\text{PER}(B)$ is not a good input for ML-CountMon. But because there are no cancellations, we have that $\text{DET}(B)$ and $\text{PER}(B)$ have the same number of monomials. So take a small circuit for the determinant (for instance the one given in [20]) and substitute its inputs by the entries of B. The result is a circuit C which computes a polynomial whose number of monomials is $\text{PER}(A)$. Observing that the determinant, and thus the polynomial computed by C, is multilinear completes the proof. □

Acknowledgements We would like to thank Sylvain Perifel for helpful discussions. The results of this paper were conceived while the third author was visiting the Équipe de Logique Mathématique at Université Paris Diderot Paris 7. He would like to thank Arnaud Durand for making this stay possible, thanks to funding from ANR ENUM (ANR-07-BLAN-0327). The third author would also like to thank his supervisor Peter Bürgisser for helpful advice.

References

[1] E. Allender, R. Beals, and M. Ogihara. The complexity of matrix rank and feasible systems of linear equations. *Computational Complexity*, 8(2):99–126, 1999.

[2] E. Allender, P. Bürgisser, J. Kjeldgaard-Pedersen, and P. B. Miltersen. On the Complexity of Numerical Analysis. *SIAM J. Comput.*, 38(5):1987–2006, 2009.

[3] E.W. Allender and K.W. Wagner. Counting hierarchies: polynomial time and constant depth circuits. *Current trends in theoretical computer science: essays and Tutorials*, 40:469, 1993.

[4] S. Arora and B. Barak. *Computational complexity: a modern approach.* Cambridge University Press, 2009.

[5] P. Bürgisser. *Completeness and reduction in algebraic complexity theory*, volume 7. Springer Verlag, 2000.

[6] P. Bürgisser. On Defining Integers And Proving Arithmetic Circuit Lower Bounds. *Computational Complexity*, 18(1):81–103, 2009.

[7] Zhixiang Chen and Bin Fu. Approximating multilinear monomial coefficients and maximum multilinear monomials in multivariate polynomials. In Weili Wu and Ovidiu Daescu, editors, *COCOA (1)*, volume 6508 of *Lecture Notes in Computer Science*, pages 309–323. Springer, 2010.

[8] Zhixiang Chen and Bin Fu. The Complexity of Testing Monomials in Multivariate Polynomials. In Weifan Wang, Xuding Zhu, and Ding-Zhu Du, editors, *COCOA*, volume 6831 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2011.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[10] F. Green. On the Power of Deterministic Reductions to $C_=P$. *Theory of Computing Systems*, 26(2):215–233, 1993.

[11] J. Hammond. Question 6001. *Educ. Times*, 32:179, 1879.

[12] L.A. Hemaspaandra and M. Ogihara. *The complexity theory companion.* Springer Verlag, 2002.

[13] M. Jansen and R. Santhanam. Permanent Does Not Have Succinct Polynomial Size Arithmetic Circuits of Constant Depth. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP*, volume 6755 of *Lecture Notes in Computer Science*, pages 724–735. Springer, 2011.

[14] V. Kabanets and R. Impagliazzo. Derandomizing Polynomial Identity Tests Means Proving Circuit Lower Bounds. *Computational Complexity*, 13:1–46, 2004.

[15] N. Kayal and C. Saha. On the Sum of Square Roots of Polynomials and related problems. In *IEEE Conference on Computational Complexity*, pages 292–299, 2011.

[16] P. Koiran and S. Perifel. The complexity of two problems on arithmetic circuits. *Theor. Comput. Sci.*, 389(1-2):172–181, 2007.

[17] P. Koiran and S. Perifel. Interpolation in Valiant's Theory. *Computational Complexity*, pages 1–20, 2011.

[18] I. Koutis. Faster Algebraic Algorithms for Path and Packing Problems. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP*, volume 5125 of *Lecture Notes in Computer Science*, pages 575–586. Springer, 2008.

[19] J. H. P. Kwisthout, H. L. Bodlaender, and L. C. Van Der Gaag. The complexity of finding kth most probable explanations in probabilistic networks. In *Proceedings of the 37th international conference on Current trends in theory and practice of computer science*, SOFSEM'11, pages 356–367, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] M. Mahajan and V. Vinay. A combinatorial algorithm for the determinant. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 730–738. Society for Industrial and Applied Mathematics, 1997.

[21] G. Malod. The Complexity of Polynomials and Their Coefficient Functions. In *IEEE Conference on Computational Complexity*, pages 193–204. IEEE Computer Society, 2007.

[22] G. Malod and N. Portier. Characterizing Valiant's algebraic complexity classes. *J. Complexity*, 24(1):16–38, 2008.

[23] M. Mundhenk, J. Goldsmith, C. Lusena, and E. Allender. Complexity of Finite-Horizon Markov Decision Process Problems. *Journal of the ACM (JACM)*, 47(4):681–720, 2000.

[24] A. Schönhage. On the Power of Random Access Machines. In Hermann A. Maurer, editor, *ICALP*, volume 71 of *Lecture Notes in Computer Science*, pages 520–529. Springer, 1979.

[25] Y. Strozecki. *Enumeration complexity and matroid decomposition.* PhD thesis, Université Paris Diderot - Paris 7, 2010.

[26] J. Torán. Succinct Representations of Counting Problems. In Teo Mora, editor, *AAECC*, volume 357 of *Lecture Notes in Computer Science*, pages 415–426. Springer, 1988.

[27] J. Torán. Complexity Classes Defined by Counting Quantifiers. *J. ACM*, 38(3):753–774, 1991.

[28] L.G. Valiant. Completeness classes in algebra. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 249–261. ACM, 1979.

[29] H. Venkateswaran and M. Tompa. A New Pebble Game that Characterizes Parallel Complexity Classes. *SIAM J. Comput.*, 18(3):533–549, 1989.

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

[31] K. W. Wagner. The Complexity of Combinatorial Problems with Succinct Input Representation. *Acta Informatica*, 23(3):325–356, 1986.

[32] R. Williams. Finding paths of length k in $O^*(2^k)$ time. *Information Processing Letters*, 109(6):315–318, 2009.

A. Parse trees and coefficients

Define inductively the parse trees of a circuit C in the following manner:

1. the only parse tree of an input gate is the gate itself,

2. the parse trees of an addition gate α with argument gates β and γ are obtained by taking either a parse tree of β and adding the edge from β to α or by taking a parse tree of γ and adding the edge from γ to α,

3. the parse trees of a multiplication gate α with argument gates β and γ are obtained by taking a parse tree of β and a parse tree of γ and adding the edge from β to α and the edge from γ to α, renaming vertices so that the chosen parse trees of β and γ are disjoint.

The value of a parse tree is defined as the product of the labels of each input gate in the parse tree (note that in the parse tree there may be several copies of a given input gate of the circuit, so that the corresponding label will have as power the number of copies of the gate). It is easy to see that the polynomial computed by a circuit is the sum of the values of its parse trees:

$$C(\bar{x}) = \sum_{T \text{ parse tree of } C} \text{value}(T).$$

In the case of a multiplicatively disjoint circuit, any parse tree is a subgraph of the circuit. In this case, a parse tree can be equivalently seen as a subgraph defined by a subset of T of the edges satisfying the following properties:

1. it contains the output gate,

2. for any addition gate α, if T contains an edge with origin α, then T contains exactly one edge with destination α,

3. for any multiplication gate α, if T contains an edge with origin α, then T contains both (all) edges with destination α,

4. for any gate α, if T contains an edge with destination α, then T contains an edge with origin α.

In the case of an arbitrary circuit, the set of parse trees of a circuit C is equal to the set of parse trees of the associated arithmetic formula (or of the associated multiplicatively disjoint circuit). But a parse tree of a circuit C need not be a subgraph, in particular the size of a parse tree may be exponential in the size of the circuit, because of the duplication involved in the definition of parse trees. This means that for a given edge e in the circuit C, there may be several copies of e in a parse tree. We will call the number of copies the *multiplicity* of e.

The type τ of a parse tree T for a circuit C is the function which associates to each edge e of C its multiplicity $\tau(e)$ in T. One could give an inductive characterization of parse tree types similar to the definition of parse trees above, but the following characterization is more useful. Add a unique artificial edge e_{out} whose origin is the output gate of the circuit. A parse tree type τ must satisfy the properties below:

1. the multiplicity of e_{out} is 1,
2. for any addition gate α with arguments β and γ, the sum of multiplicities of the edges with origin α is equal to the sum of the multiplicity of (β, α) and the multiplicity of (γ, α),
3. for any multiplication gate α with arguments β and γ, the sum of multiplicities of the edges with origin α is equal both to the multiplicity of (β, α) and to the multiplicity of (γ, α),
4. for any gate α, if the multiplicity of an edge with destination α is strictly positive, then there must be an edge with origin α whose multiplicity is strictly positive.

Note that if the circuit is multiplicatively disjoint, since all parse trees are subgraphs of the circuit, the multiplicity of an edge in a parse tree is always at most 1. In this case a parse tree type contains exactly one parse tree and we could identify parse tree types and parse trees, the characterization above becomes identical to the one given for parse trees.

Define the value of a parse tree type as the product, for all input gate α, of the label of α raised to a power equal to the sum of the multiplicities of edges with origin α. The sum of the values of the parse trees can now be partitioned by parse tree types to express the value computed by C:

$$C(\bar{x}) = \sum_{\tau \text{ a type for } C} |\{T \text{ parse tree of type } \tau\}| \cdot \text{value}(\tau).$$