# On a Polytime Factorization Algorithm for Multilinear Polynomials over $\mathbb{F}_2$

Pavel Emelyanov[1,2](✉) and Denis Ponomaryov[1,2]

[1] A.P. Ershov Institute of Informatics Systems,
Lavrentiev av. 6, 630090 Novosibirsk, Russia
{emelyanov,ponom}@iis.nsk.su
[2] Novosibirsk State University, Pirogov st. 1, 630090 Novosibirsk, Russia

**Abstract.** In 2010, Shpilka and Volkovich established a prominent result on the equivalence of polynomial factorization and identity testing. It follows from their result that a multilinear polynomial over the finite field of order 2 can be factored in time cubic in the size of the polynomial given as a string. Later, we have rediscovered this result and provided a simple factorization algorithm based on computations over derivatives of multilinear polynomials. The algorithm has been applied to solve problems of compact representation of various combinatorial structures, including Boolean functions and relational data tables. In this paper, we describe an improvement of this factorization algorithm and report on preliminary experimental analysis.

## 1 Introduction

Polynomial factorization is a classic algorithmic problem in algebra [14], whose importance stems from numerous applications. The computer era has stimulated interest to polynomial factorization over finite fields. For a long period of time, Theorem 1.4 in [8] (see also [12, Theorem 1.6]) has been the main source of information on the complexity of this problem: a (densely represented) polynomial $F_{p^r}(x_1, \ldots, x_m)$ of the total degree $n > 1$ over all its variables can be factored in time that is polynomial in $n^m$, $r$, and $p$. In addition, practical probabilistic factorization algorithms have been known.

In 2010, Shpilka and Volkovich [13] established a connection between polynomial factorization and polynomial identity testing. The result has been formulated in terms of the arithmetic circuit representation of polynomials. It follows from these results that a multilinear polynomial over $\mathbb{F}_2$ (the finite field of the order 2) can be factored in the time that is cubic in the size of the polynomial given as a symbol sequence.

Multilinear polynomials over $\mathbb{F}_2$ are well known in the scope of mathematical logic (as Zhegalkine polynomials [15] or Algebraic Normal Form) and in circuit synthesis (Canonical Reed-Muller Form [10]). Factorization of multilinear

This work is supported by the Ministry of Science and Education of the Russian Federation under the 5-100 Excellence Program and the grant of Russian Foundation for Basic Research No. 17-51-45125.

polynomials is a particular case of decomposition (so-called conjunctive or AND-decomposition) of logic formulas and Boolean functions. By the idempotence law in the algebra of logic, multilinearity (all variables occur in degree 1) is a natural property of these polynomials, which makes the factors have disjoint sets of variables $F(X,Y) = F_1(X)F_2(Y)$, $X \cap Y = \varnothing$. In practice, this property allows for obtaining a factorization algorithm by variable partitioning (see below).

Among other application domains, such as game and graph theory, the most attention has been given to decomposition of Boolean functions in logic circuit synthesis, which is related to the algorithmic complexity and practical issues of electronic circuits implementation, their size, time delay, and power consumption (see [9,11], for example). One may note the renewed interest in this topic, which is due to the novel technological achievements in circuit design.

The logic interpretation of multilinear polynomials over $\mathbb{F}_2$ admits another notion of factorization, which is commonly called Boolean factorization (finding Boolean divisors). For example, there are Boolean polynomials, which have decomposition components sharing some common variables. Their product/conjunction does not produce original polynomials in the algebraic sense but it gives the same functions/formulas in the logic sense. In general, logic-based approaches to decomposition are more powerful than algebraic ones: a Boolean function can be decomposable logically, but not algebraically [9, Chap. 4].

In 2013, the authors have rediscovered the result of Shpilka and Volkovich under simpler settings and in a simpler way [5,7]. A straightforward treatment of sparsely represented multilinear polynomials over $\mathbb{F}_2$ gave the same worst-case cubic complexity of the factorization algorithm. Namely, the authors provided two factorization algorithms based, respectively, on computing the greatest common divisor (GCD) and formal derivatives (FD) for polynomials obtained from the input one.

The algorithms have been used to obtain a solution to the following problems of compact representation of different combinatorial structures (below we provide examples, which intuitively explain their relation to the factorization problem).

– Conjunctive disjoint decomposition of monotone Boolean functions given in positive DNF [5,7]. For example, the following DNF

$$\varphi = (x \wedge u) \;\vee\; (x \wedge v) \;\vee\; (y \wedge u) \;\vee\; (y \wedge v) \vee (x \wedge u \wedge v) \tag{1}$$

is equivalent to

$$\psi = (x \wedge u) \;\vee\; (x \wedge v) \;\vee\; (y \wedge u) \;\vee\; (y \wedge v), \tag{2}$$

since the last term in $\varphi$ is redundant, and we have

$$\psi \equiv (x \vee y) \wedge (u \vee v) \tag{3}$$

and the decomposition components $x \vee y$ and $u \vee v$ can be recovered from the factors of the polynomial

$$F_\psi = xu + xv + yu + yv = (x + y) \cdot (u + v) \tag{4}$$

constructed for $\psi$.

- Conjunctive disjoint decomposition of Boolean functions given in full DNF [5,7]. For example, the following full DNF

$$\varphi = (x \wedge \neg y \wedge u \wedge \neg v) \vee (x \wedge \neg y \wedge \neg u \wedge v) \vee \\ \vee (\neg x \wedge y \wedge u \wedge \neg v) \vee (\neg x \wedge y \wedge \neg u \wedge v)$$

  is equivalent to

$$(x \wedge \neg y) \vee (\neg x \wedge y) \bigwedge (u \wedge \neg v) \vee (\neg u \wedge v), \tag{5}$$

  and the decomposition components of $\varphi$ can be recovered from the factors of the polynomial

$$F_\varphi = x\bar{y}u\bar{v} + x\bar{y}\bar{u}v + \bar{x}yu\bar{v} + \bar{x}y\bar{u}v = (x\bar{y} + \bar{x}y) \cdot (u\bar{v} + \bar{u}v) \tag{6}$$

  constructed for $\varphi$.
- Non-disjoint conjunctive decomposition of multilinear polynomials over $\mathbb{F}_2$, in which components can have common variables from a given set. In [3], a fixed-parameter polytime decomposition algorithm has been proposed, for the parameter being the number of the shared variables between components.
- Cartesian decomposition of data tables (i.e., finding tables such that their unordered Cartesian product gives the source table) [4,6] and generalizations thereof for the case of a non-empty subset of shared attributes between the tables. For example, the following table has a decomposition of the form:

$$\begin{array}{|c|c|c|c|c|} \hline B & E & D & A & C \\ \hline z & q & u & x & y \\ \hline y & q & u & x & y \\ \hline y & r & v & x & z \\ \hline z & r & v & x & z \\ \hline y & p & u & x & x \\ \hline z & p & u & x & x \\ \hline \end{array} = \begin{array}{|c|c|} \hline A & B \\ \hline x & y \\ \hline x & z \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline C & D & E \\ \hline x & u & p \\ \hline y & u & q \\ \hline z & v & r \\ \hline \end{array}$$

  which can be obtained from the factors of the polynomial

$$\begin{aligned} & z_B \cdot q \cdot u \cdot x_A \cdot y_C + y_B \cdot q \cdot u \cdot x_A \cdot y_C + \\ & y_B \cdot r \cdot v \cdot x_A \cdot z_C + z_B \cdot r \cdot v \cdot x_A \cdot z_C + \\ & y_B \cdot p \cdot u \cdot x_A \cdot x_C + z_B \cdot p \cdot u \cdot x_A \cdot x_C \\ & \qquad = (x_A \cdot y_B + x_A \cdot z_B) \cdot (q \cdot u \cdot y_C + r \cdot v \cdot z_C + p \cdot u \cdot x_C) \end{aligned}$$

  constructed for the table's content.
  In terms of SQL, Cartesian decomposition means reversing the first operator and the second operator represents some feasible generalization of the problem:

```
T1 CROSS JOIN T2                SELECT T1.*, T2.* EXCEPT(Attr2)
                                  FROM T1 INNER JOIN T2
                                    ON T1.Attr1 = T2.Attr2
```

  where `EXCEPT(list)` is an informal extension of SQL used to exclude `list` from the resulting attributes. This approach can be applied to other table-based structures (for example, decision tables or datasets appearing in the K&DM domain, as well as the truth tables of Boolean functions).

Shpilka and Volkovich did not address the problems of practical implementations of the factorization algorithm. However, the applications above require a factorization algorithm to be efficient enough on large polynomials. In this paper, we propose an improvement of the factorization algorithm from [4,6], which potentially allows for working with larger inputs. An implementation of this version of the algorithm in Maple 17 outperforms the native Maple's `Factor(poly) mod 2` factorization, which in our experiments failed to terminate on input polynomials having $10^3$ variables and $10^5$ monomials.

## 2 Definitions and Notations

A polynomial $F \in \mathbb{F}_2[x_1, \dots, x_n]$ is called *factorable* if $F = F_1 \cdot \ldots \cdot F_k$, where $k \geq 2$ and $F_1, \dots, F_k$ are non-constant polynomials. The polynomials $F_1, \dots, F_k$ are called *factors* of $F$. It is important to realize that since we consider multilinear polynomials (every variable can occur only in the power of $\leq 1$), the factors are polynomials *over disjoint sets of variables*. In the following sections, we assume that the polynomial $F$ does not have *trivial divisors*, i.e., neither $x$, nor $x+1$ divides $F$. Clearly, trivial divisors can easily be recognized.

For a polynomial $F$, a variable $x$ from the set of variables $Var(F)$ of $F$, and a value $a \in \{0, 1\}$, we denote by $F_{x=a}$ the polynomial obtained from $F$ by substituting $x$ with $a$. For multilinear polynomials over $\mathbb{F}_2$, we define a *formal derivative* as $\frac{\partial F}{\partial x} = F_{x=0} + F_{x=1}$, but for non-linear ones, we use the definition of a "standard" formal derivative for polynomials. Given a variable $z$, we write $z|F$ if $z$ divides $F$, i.e., $z$ is present in every monomial of $F$ (note that this is equivalent to the condition $\frac{\partial F}{\partial z} = F_{z=1}$).

Given a set of variables $\Sigma$ and a monomial $m$, the *projection* of $m$ onto $\Sigma$ is 1 if $m$ does not contain any variable from $\Sigma$, or is equal to the monomial obtained from $m$ by removing all the variables not contained in $\Sigma$, otherwise. The *projection* of a polynomial $F$ onto $\Sigma$, denoted as $F|_{\Sigma}$, is the polynomial obtained as sum of monomials from the set $S$, where $S$ is the set of the monomials of $F$ projected onto $\Sigma$.

$|F|$ is the *length* of the polynomial $F$ given as a symbol sequence, i.e., if the polynomial over $n$ variables has $M$ monomials of lengths $m_1, \dots, m_M$ then $|F| = \sum_{i=1}^{M} m_i = O(nM)$.

We note that the correctness proofs for the algorithms presented below can be found in [5,7].

## 3 GCD-Algorithm

Conceptually, this algorithm is the simplest one. It outputs factors of an input polynomial whenever they exist.

```
1. Take an arbitrary variable $x$ from $Var(F)$
2. $G := \gcd(F_{x=0}, \frac{\partial F}{\partial x})$
3. If $G = 1$ then stop
```

```
4. Output factor $\frac{F}{G}$
5. $F := G$
6. Go to 1
```

Here the complexity of factorization is hidden in the algorithm for finding the greatest common divisor of polynomials.

Computing GCD is known as a classic algorithmic problem in algebra [14], which involves computational difficulties. For example, if the field is not too rich ($\mathbb{F}_2$ is an example) then intermediate values vanish quite often, which essentially affects the computation performance. In [2], Wittkopf et al. developed the LINZIP algorithm for the GCD-problem. Its complexity is $O(|F|^3)$, i.e., the complexity of the GCD-algorithm is asymptotically the same as for Shpilka and Volkovich's result for the case of multilinear polynomials (given as strings).

## 4 FD-Algorithm

In the following, we assume that the input polynomial $F$ contains at least two variables. The basic idea of FD-Algorithm is to partition a variable set into two sets with respect to a selected variable:

– the first set $\Sigma_{same}$ contains the selected variable and corresponds to an irreducible polynomial;
– the second set $\Sigma_{other}$ corresponds to the second polynomial that can admit further factorization.

As soon as $\Sigma_{same}$ and $\Sigma_{other}$ are computed (and $\Sigma_{other} \neq \varnothing$), the corresponding factors can be easily obtained as projections of the input polynomial onto these sets.

```
1. Take an arbitrary variable $x$ occurring in $F$
2. Let $\Sigma_{same} := \{x\}, \Sigma_{other} := \varnothing, F_{same} := 0, F_{other} := 0$
3. Compute $G := F_{x=0} \cdot \frac{\partial F}{\partial x}$
4. For each variable $y \in Var(F) \setminus \{x\}$:
     If $\frac{\partial G}{\partial y} = 0$ then  $\Sigma_{other} := \Sigma_{other} \cup \{y\}$
                        else $\Sigma_{same} := \Sigma_{same} \cup \{y\}$
5. If $\Sigma_{other} = \varnothing$ then report "$F$ is non-factorable" and stop
6. Return polynomials $F_{same}$ and $F_{other}$ obtained as projections
   onto $\Sigma_{same}$ and $\Sigma_{other}$, respectively
```

The factors $F_{same}$ and $F_{other}$ have the property mentioned above and hence, the algorithm can be applied to obtain factors for $F_{other}$.

Note that FD-algorithm takes $O(|F|^2)$ steps to compute the polynomial $G = F_{x=0} \cdot \frac{\partial F}{\partial x}$ and $O(|G|)$ time to test whether the derivative $\frac{\partial G}{\partial y}$ equals zero. As we have to verify this for every variable $y \neq x$, we have a procedure that computes a variable partition in $O(|F|^3)$ steps. The algorithm allows for a straightforward parallelization on the selected variable $y$: the loop over the variable $y$ (selected in line 4) can be performed in parallel for all the variables.

One can readily see that the complexity of factorization is hidden in the computation of the product $G$ of two polynomials and testing whether a derivative of this product is equal to zero. In the worst case, the length of $G = F_{x=0} \cdot \frac{\partial F}{\partial x}$ equals $\Omega(|F|^2)$, which makes computing this product expensive for large input polynomials. In the next section, we describe a modification of the FD-algorithm, which implements the test above in a more efficient recursive fashion, without the need to compute the product of polynomials explicitly.

## 5 Modification of FD-Algorithm

Assume the polynomials $A = \frac{\partial F}{\partial x}$ and $B = F_{x=0}$ are computed. By taking a derivative of $A \cdot B$ on $y$ (a variable different from $x$) we have $D = \frac{\partial F_{x=0}}{\partial y}$ and $C = \frac{\partial^2 F}{\partial x \partial y}$. We need to test whether $AD + BC = 0$, or equivalently, $AD = BC$. The main idea is to reduce this test to four tests involving polynomials of smaller sizes. Proceeding recursively in this way, we obtain smaller, or even constant, polynomials for which identity testing is simpler. Yet again, the polynomial identity testing demonstrates its importance, as Shpilka and Volkovich have readily established.
Steps 3–4 of FD-algorithm are modified as follows:

```
Let $A = \frac{\partial F}{\partial x}$, $B = F_{x=0}$
For each variable $y \in Var(F) \setminus \{x\}$:
  Let $D = \frac{\partial B}{\partial y}$, $C = \frac{\partial A}{\partial y}$
  If IsEqual($A$,$D$,$B$,$C$) then  $\Sigma_{other} := \Sigma_{other} \cup \{y\}$,
                              else  $\Sigma_{same} := \Sigma_{same} \cup \{y\}$
```

where (all the above mentioned variables are chosen from the set of variables of the corresponding polynomials).

```
Define IsEqual($A$,$D$,$B$,$C$) returning Boolean

 1. If $A = 0$ or $D = 0$ then return ($B = 0$ or $C = 0$)
 2. If $B = 0$ or $C = 0$ then return FALSE
 3. For all variables $z$ occurring in at least one of $A, B, C, D$ :
 4.   If ($z|A$ or $z|D$) xor ($z|B$ or $z|C$) then return FALSE
 5.   Replace every $X \in \{A, B, C, D\}$ with $X := \frac{\partial X}{\partial z}$, provided $z|X$
 6. If $A = 1$ and $D = 1$ then return ($B = 1$ and $C = 1$)
 7. If $B = 1$ and $C = 1$ then return FALSE
 8. If $A = 1$ and $B = 1$ then return ($D = C$)
 9. If $D = 1$ and $C = 1$ then return ($A = B$)
10. Pick a variable $z$
11. If not IsEqual($A_{z=0}$,$D_{z=0}$,$B_{z=0}$,$C_{z=0}$) then return FALSE
12. If not IsEqual( $\frac{\partial A}{\partial z}$, $\frac{\partial D}{\partial z}$, $\frac{\partial B}{\partial z}$, $\frac{\partial C}{\partial z}$) then return FALSE
13. If IsEqual( $\frac{\partial A}{\partial z}$, $B_{z=0}$, $A_{z=0}$, $\frac{\partial B}{\partial z}$) then return TRUE
 14 Return IsEqual( $\frac{\partial A}{\partial z}$, $C_{z=0}$, $A_{z=0}$, $\frac{\partial C}{\partial z}$)

End Definition
```

Several comments on `IsEqual` are in order:

- Lines 1–9 implement processing of trivial cases, when the condition $AD = BC$ can easily be verified without recursion. For example, when line 2 is executed, it is already known that neither $A$ nor $D$ equals zero and hence, $AD$ can not be equal to $BC$. Similar tests are implemented in lines 6–9.
- At line 5 it is known that $z$ divides both, $AD$ and $BC$ and thus, the problem $AD = BC$ can be reduced to the polynomials obtained by eliminating $z$.
- Finally, lines 11–14 implement recursive calls to `IsEqual`. Observe that the parameter polynomials are obtained from the original ones by evaluating a variable $z$ to zero or by computing a derivative. Both of the operations yield polynomials of a smaller size than the original ones and can give constant polynomials in the limit. To determine the parameters of `IsEqual` we resort to a trick that transforms one identity into two smaller ones. This transformation uses a multiplier, which is not unique. Namely, we can select 16 variants among 28 possible ones (see comments in Sect. 5.1 below) and this gives 16 variants of lines 13–14.

## 5.1 Complete List of Possible Parameters

If $A$, $D$, $B$, $C$ are the parameters of `IsEqual`, we denote for a $Q \in \{A, D, B, C\}$ the derivative on a variable $z$ and evaluation $z = 0$ as $Q_1$ and $Q_2$, respectively.

$$AD = BC \quad \text{iff} \quad (A_1 z + A_2)(D_1 z + D_2) = (B_1 z + B_2)(C_1 z + C_2),$$

$$A_1 D_1 z^2 + (A_1 D_2 + A_2 D_1) z + A_2 D_2 = B_1 C_1 z^2 + (B_1 C_2 + B_2 C_1) z + B_2 C_2.$$

The equality holds iff the corresponding coefficients are equal:

$$\begin{cases} A_1 D_1 = B_1 C_1 & (1) \\ A_2 D_2 = B_2 C_2 & (2) \\ A_1 D_2 + A_2 D_1 = B_1 C_2 + B_2 C_1 & (3) \end{cases}$$

If at least one of the identities (1), (2) does not hold then $AD \neq BC$. Otherwise, we can use these identities to verify (3) in the following way.

By the rule of choosing $z$, we can assume $A_1, A_2 \neq 0$. Multiplying both sides of (3) by $A_1 A_2$ gives

$$A_1^2 A_2 D_2 + A_1 A_2^2 D_1 = A_1 A_2 B_1 C_2 + A_1 A_2 B_2 C_1.$$

Next, by using the identities (1) and (2),

$$A_1^2 B_2 C_2 + A_1 A_2 B_2 C_1 = A_2^2 B_1 C_1 + A_1 A_2 B_1 C_2,$$

$$A_1 B_2 (A_1 C_2 + A_2 C_1) = A_2 B_1 (A_2 C_1 + A_1 C_2).$$

Hence, it suffices to check $(A_1 B_2 + A_2 B_1)(A_1 C_2 + A_2 C_1) = 0$, i.e., at least one of these factors equals zero. It turns out that we need to test at most 4 polynomial identities, and each of them is smaller than the original identity $AD = BC$.

Notice that the multiplier $A_1A_2$ is used to construct the version of `IsEqual` given above.

By the rule of choosing $z$, we can take different multiplier's combinations of the pairs of 8 elements. Only 16 out of 28 pairs are appropriate:

$$\begin{aligned}
A_1A_2 &\to A_1C_2 = A_2C_1, \quad A_1B_2 = A_2B_1\\
A_1B_2 &\to A_1D_2 = B_2C_1, \quad A_1B_2 = A_2B_1\\
A_1C_2 &\to A_1D_2 = B_1C_2, \quad A_1C_2 = A_2C_1\\
A_1D_2 &\to A_1D_2 = B_2C_1, \quad A_1D_2 = B_1C_2\\
A_2B_1 &\to A_2D_1 = B_1C_2, \quad A_1B_2 = A_2B_1\\
A_2C_1 &\to A_2D_1 = B_2C_1, \quad A_1C_2 = A_2C_1\\
A_2D_1 &\to A_2D_1 = B_2C_1, \quad A_2D_1 = B_1C_2\\
B_1B_2 &\to B_1D_2 = B_2D_1, \quad A_1B_2 = A_2B_1\\
B_1C_2 &\to A_2D_1 = B_1C_2, \quad A_1D_2 = B_1C_2\\
B_1D_2 &\to B_1D_2 = B_2D_1, \quad A_1D_2 = B_1C_2\\
B_2C_1 &\to A_2D_1 = B_2C_1, \quad A_1D_2 = B_2C_1\\
B_2D_1 &\to B_1D_2 = B_2D_1, \quad A_2D_1 = B_2C_1\\
C_1C_2 &\to C_1D_2 = C_2D_1, \quad A_1C_2 = A_2C_1\\
C_1D_2 &\to C_1D_2 = C_2D_1, \quad A_1D_2 = B_2C_1\\
C_2D_1 &\to C_1D_2 = C_2D_1, \quad A_2D_1 = B_1C_2\\
D_1D_2 &\to C_1D_2 = C_2D_1, \quad B_1D_2 = B_2D_1
\end{aligned}$$

## 5.2 Analysis of ModFD-Algorithm for Random Polynomials

We now provide a theoretical analysis of ModFD-algorithm. The complexity estimations we describe here are conservative and, therefore, they give an upper bound greater than $O(|F|^3)$ of the original FD-algorithm. However, the approach presented here could serve as a basis to obtain a more precise upper bound, which would explain the gain in performance in practice; we report on a preliminary experimental evaluation in Sect. 6.

Our estimation is based on

**Theorem 1 (Akra and Bazzi, [1]).** *Let the recurrence*

$$T(x) = g(x) + \sum_{i=1}^{k} \lambda_i T(\omega_i x + h_i(x)) \quad \textit{for} \quad x \geq C$$

*satisfy the following conditions:*

1. *$T(x)$ is appropriately defined for $x < C$;*
2. *$\lambda_i > 0$ and $0 < \omega_i < 1$ are constants for all $i$;*
3. *$|g(x)| = O(x^c)$; and*
4. *$|h_i(x)| = O\left(\frac{x}{(\log x)^2}\right)$ for all $i$.*

*Then*

$$T(x) = \Theta\left(x^p\left(1 + \int_1^x \frac{g(t)}{t^{p+1}}dt\right)\right),$$

*where $p$ is determined by the characteristic equation $\sum_{i=1}^{k} \lambda_i \omega_i^p = 1$.*

For the complexity estimations, we assume that polynomials are represented by alphabetically sorted lists of bitscales corresponding to indicator vectors for the variables of monomials. Hence, to represent a polynomial $F$ over $n$ variables with $M$ monomials $|F| = nM + cM$ bits are required, where $c$ is a constant overhead to maintain the list structure. This guarantees the linear time complexity for the following operations:

– computing a derivative with respect to a variable (the derived polynomial also remains sorted);
– evaluating to zero for a variable with removing the empty bitscale representing the constant 1 if it occurs (the derived polynomial also remains sorted);
– identity testing for polynomials derived from the original sorted polynomial by the two previous operations.

For `IsEqual` we have

1. $x = |A| + |B| + |C| + |D|$. By taking into account the employed representation of monomials (the bitscale is not shortened when a variable is removed), we may also assume that $|Q| = |Q_1| + |Q_2|$.
2. $\forall i, \lambda_i = 1$.
3. $\forall i, h_i(x) = 0$.
4. $g(x) = O(nx)$. Therefore, the total time for lines 1–10 consists of the constant numbers of linear (with respect to the input of `IsEqual`) operations executed at most $n$ times. Apparently, $n$ is quite a conservative assumption, because at a single recursion step, at least one variable is removed from the input set of variables.
5. We need to estimate $\omega_1$, $\omega_2$, $\omega_3$, $\omega_4$.
   Among all the possible choices of the multipliers mentioned in Sect. 5.1, let us consider those of the form $Q_1Q_2$. They induce two equations that do not contain one of the input parameters of `IsEqual`: $A$, $B$, $C$, $D$ result in the absence of the parts of $D$, $C$, $B$, $A$, respectively, among the parameters of `IsEqual` in lines 13 and 14. Hence, the largest parameter can be excluded by taking an appropriate $Q$; lines 13–14 of ModFD-algorithm are to be rewritten with the help of this observation.
   Without loss of generality, we may assume that the largest parameter is $D$ and thus, we can take $Q$ equal to $A$. In this case, $\omega_1$, $\omega_2$, $\omega_3$, $\omega_4$ represent the relative lengths of the parameters $|A_1| + |B_1| + |C_1| + |D_1|$, $|A_2| + |B_2| + |C_2| + |D_2|$, $|A| + |B|$, $|A| + |C|$ for the recursive calls to `IsEqual` with respect to $|A| + |B| + |C| + |D|$.
   Since $|A|, |B|, |C| \leq |D|$, we obtain $|A| + |B|$, $|A| + |C|$, $|B| + |C| \leq 2|D|$. Then the lengths $|A| + |B|$ and $|A| + |C|$, respectively, can be estimated in the following way:

$$|A| + |B| = x - |C| - |D| \leq x - 0 - \frac{|A| + |B|}{2},$$

   hence, $|A| + |B|$, $|A| + |C| \leq \frac{2}{3}$.

Let $F$ be a multilinear polynomial over $n$ variables with $M$ monomials such that no variable divides $F$. A random polynomial consists of monomials randomly chosen from the set of all monomials over $n$ variables. Variables appear in monomials independently. For each variable $x$ from $\texttt{var}(F)$, we can consider the following quantity $\mu_x = \frac{\partial F}{\partial x}$ (i.e. the part of monomials containing this variable). We want to estimate the probability that among $\mu_x$ there exist at least one, which is (approximately) equal to $\frac{M}{2}$. Hence

$$\begin{aligned}\mathbb{P}\left[\text{there exists } x \text{ such that } \mu_x \text{ is a median}\right] &= 1 - \mathbb{P}\left[\textstyle\bigwedge_x \mu_x \text{ is not median}\right]\\ &= 1 - \mathbb{P}\left[\mu_1 \text{ is not median}\right]^n\\ &= 1 - \left(1 - \mathbb{P}\left[\mu_1 \text{ is a median}\right]\right)^n\\ &= 1 - \left(1 - \tfrac{1}{2}\right)^n\\ &= 1 - \tfrac{1}{2^n}\end{aligned}$$

Thus, with a high probability one can pick from a large polynomial (in our case, from $D$) a variable such that $|D_1| \approx |D_2|$.
Let us consider the following multicriteria linear program:

$$\text{maximize} \begin{Bmatrix} a_1 + b_1 + c_1 + d_1 \\ a_2 + b_2 + c_2 + d_2 \\ a + b \\ a + c \end{Bmatrix} \text{ subject to } \begin{cases} a + b + c + d = 1 \\ d_1 = d_2 \\ a \le d,\ b \le d,\ c \le d \\ a + b \le \frac{2}{3} \\ a + c \le \frac{2}{3} \\ \text{all nonnegative} \end{cases}.$$

Since the objective functions and constraints are linear and the optimization domain is bounded, we can enumerate all the extreme points of the problem and select those points that give the maximum solution of the characteristic equation of Theorem 1. By taking into account the symmetries between the first and the second objective functions and between the third and fourth ones, we obtain that

$$\omega_1 = \frac{3}{4}, \quad \omega_2 = \frac{1}{4}, \quad \omega_3 = \frac{1}{2}, \quad \omega_4 = \frac{1}{2}. \qquad (*)$$

Hence, the characteristic equation is

$$\left(\frac{3}{4}\right)^p + \left(\frac{1}{4}\right)^p + \left(\frac{1}{2}\right)^p + \left(\frac{1}{2}\right)^p = 1.$$

Its unique real solution is $p \approx 2.226552$. Finally, the total time for the ModFD-algorithm obtained this way is

$$T = O(n^2 |F|^{2.226552}).$$

## 6 Preliminary Experiments and Discussion

For a computational evaluation of the developed factorization algorithms, we used Maple 17 for Windows run on 3.0 GHz PC with 8 GB RAM. The factorization algorithm implemented in Maple `Factor(poly) mod 2` can process multilinear polynomials over $\mathbb{F}_2$ with hundreds of variables and several thousands of monomials in several hours. But many attempts of factorization of polynomials with $10^3$ variables and $10^5$ monomials were terminated by the time limit of roughly one week of execution. In general, a disadvantage of all Maple implementations is that they are memory consuming. For example, the algorithm that requires computing products of polynomials fails to work even for rather small examples (about $10^2$ variables and $10^3$ monomials). Although GCD-algorithm is conceptually simple, it involves computing the greatest common divisor for polynomials over the "poor" finite field $\mathbb{F}_2$. A practical implementation of LINZIP is not that simple. An older version of Maple reports on some inputs that "LINZIP/not implemented yet". We did not observe this issue in Maple 17. It would be important to conduct an extensive comparison of the performance of GCD- and FD-algorithm implemented under similar conditions. The factorization algorithm (FD-based) for sparsely represented multilinear polynomials over $\mathbb{F}_2$ demonstrates reasonable performance. BDD/ZDD can be considered as some kind of the black box representation. We are going to implement factorization based on this representation and to compare these approaches.

A careful study of the solution (*) given at the end of Sect. 5.2 shows that it describes the case when $|A| \approx |D| \approx \frac{x}{2}$ and $|B| \approx |C| \approx 0$. This means that at the next steps the maximal parameter is $A$: $|A| \approx \frac{x}{2}$, while the remaining parameters are smaller. Thus, one can see that the lengths of the inputs to the recursive calls of `IsEqual` are reduced at least twice in at most two levels of the recursion. This allows for obtaining a more precise complexity bound, which will be further studied.

Yet another property is quite important for the performance of the algorithm. Evaluating the predicate `IsEqual` for the variables from the same factor requires significantly less time compared with evaluation for other variables. For polynomials with 50 variables and 100 monomials in the both components, the speed-up achieves 10–15 times. The reason is evident and it again confirms the importance of (Zero) Polynomial Identity Testing, as shown by Shplika and Volkovich. Testing that the polynomial $AD + BC$ is not zero requires less reduction steps in contrast with the case when it does equal zero. The latter requires reduction to the constant polynomials. Therefore, we used the following approach: if the polynomials $A$, $D$, $B$, $C$ are "small" enough then the polynomial $AD + BC$ was checked to be zero directly via multiplication. For the polynomials with the above mentioned properties, this allows to save about 3–5% of the execution time. The first practical conclusion is that in general, the algorithm works faster for non-factorable polynomials than for factorable ones. The second is that we need to investigate new methods to detect variables from the "opposite" component (factor). Below we give an idea of a possible approach.

It is useful to detect cases of irreducibility before launching the factorization procedure. Using simple necessary conditions for irreducibility, as well as testing simple cases of variable classification for variable partition algorithms, can substantially improve performance. Let $F$ be a multilinear polynomial over $n$ variables with $M$ monomials such that no variable divides $F$. For each variable $x$, recall that the value $\mu_x$ corresponds to the number of monomials containing

$x$, i.e. the number of monomials in $\frac{\partial F}{\partial x}$. Then a necessary condition for $F$ to be factorable is

$$\forall x \quad \gcd(\mu_x,\ M) > 1.$$

In addition, we have deduced several properties, which are based on analyzing occurrences of pairs of variables in the given polynomial (for example, if there is no monomial in which two variables occur simultaneously then these variables can not belong to different factors). Of course, the practical usability of these properties depends on how easily they can be tested.

Finally, we note an important generalization of the factorization problem, which calls for efficient implementations of the factorization algorithm. To achieve a deeper optimization of logic circuits we asked in [5,7] how to find a representation of a polynomial in the form $F(X,Y) = G(X)H(Y) + D(X,Y)$, where a "relatively small" defect" $D(X,Y)$ extends or shrinks the pure disjoint factors. Yet another problem is to find a representation of the polynomial in the form

$$F(X,Y) = \sum_k G_k(X)H_k(Y), \qquad X \cap Y = \varnothing,$$

i.e., a complete decomposition without any "defect", which (along with the previous one) has quite interesting applications in the knowledge and data mining domain. Clearly, such decompositions (for example, the trivial one, where each monomial is treated separately) always exist, but not all of them are meaningful from the K&DM point of view. For example, one might want to put a restriction on the size of the "factorable part" of the input polynomial (e.g., by requiring the size to be maximal), which opens a perspective into a variety of optimization problems. Formulating additional constraints targeting factorization is an interesting research topic. One immediately finds a variety of the known computationally hard problems in this direction and it is yet to be realized how the computer algebra and theory of algorithms can mutually benefit from each other along this way.

## References

1. Akra, M., Bazzi, L.: On the solution of linear recurrence equations. Comput. Optim. Appl. **10**(2), 195–210 (1998). https://doi.org/10.1023/A:1018373005182
2. de Kleine, J., Monagan, M.B., Wittkopf, A.D.: Algorithms for the non-monic case of the sparse modular GCD algorithm. In: Proceedings of 2005 International Symposium on Symbolic and Algebraic Computation (ISSAC 2005), pp. 124–131. ACM, New York (2005). https://doi.org/10.1145/1073884.1073903
3. Emelyanov, P.: AND–decomposition of boolean polynomials with prescribed shared variables. In: Govindarajan, S., Maheshwari, A. (eds.) CALDAM 2016. LNCS, vol. 9602, pp. 164–175. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29221-2_14
4. Emelyanov, P.: On two kinds of dataset decomposition. In: Shi, Y., et al. (eds.) ICCS 2018. LNCS, vol. 10861, pp. 171–183. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93701-4_13

5. Emelyanov, P., Ponomaryov, D.: Algorithmic issues of AND-decomposition of Boolean formulas. Program. Comput. Softw. **41**(3), 162–169 (2015). https://doi.org/10.1134/S0361768815030032. Trans. by: Programmirovanie **41**(3), 62–72 (2015)
6. Emelyanov, P., Ponomaryov, D.: Cartesian decomposition in data analysis. In: Proceedings of Siberian Symposium on Data Science and Engineering (SSDSE 2017), Novosibirsk, Russia, pp. 55–60 (2017). https://doi.org/10.1109/SSDSE.2017.8071964
7. Emelyanov, P., Ponomaryov, D.: On tractability of disjoint AND-decomposition of Boolean formulas. In: Voronkov, A., Virbitskaite, I. (eds.) PSI 2014. LNCS, vol. 8974, pp. 92–101. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46823-4_8
8. Grigoriev, D.: Theory of Complexity of Computations. II. Notes of Scientific Seminars of LOMI (Zapiski Nauchn. Semin. Leningr. Otdel. Matem. Inst. Acad. Sci. USSR). Nauka, Leningrad, vol. 137, pp. 20–79 (1984). (in Russian)
9. Khatri, S.P., Gulati, K. (eds.): Advanced Techniques in Logic Synthesis, Optimizations and Applications. Springer, New York (2011). https://doi.org/10.1007/978-1-4419-7518-8
10. Muller, D.E.: Application of Boolean algebra to switching circuit design and to error detection. IRE Trans. Electron. Comput. **EC-3**, 6–12 (1954)
11. Perkowski, M.A., Grygiel, S.: A survey of literature on function decomposition, Version IV. PSU Electrical Engineering Department Report, Portland State University, Portland, Oregon, USA, November 1995
12. Shparlinski, I.E.: Computational and Algorithmic Problems in Finite Fields. Springer, New York (1992). https://doi.org/10.1007/978-94-011-1806-4
13. Shpilka, A., Volkovich, I.: On the relation between polynomial identity testing and finding variable disjoint factors. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 408–419. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14165-2_35
14. von zur Gathen, J., Gerhard, J.: Modern Computer Algebra, 3rd edn. Cambridge University Press, New York (2013). https://doi.org/10.1017/CBO9781139856065
15. Zhegalkin, I.: Arithmetization of symbolic logics. Sbornik Mathematics **35**(1), 311–377 (1928). (in Russian)