Universiteit Leiden

Exploring finite field multiplication

Daniël de Haas (s4080041)

Supervisors:
Nusa Zidaric & Lars van der Kuil

Bachelor Thesis – Informatica

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

August 24, 2026

Abstract

We look at the implementation of multiplication in binary finite fields. Such multiplications can be represented as matrix-vector products over $\mathbb{F}_2$, allowing the use of techniques from linear circuit optimization. This can reduce the number of XOR gates required, that is the XOR-count, to do multiplication.

For multiplication by a fixed element, two metrics for estimating the XOR-count of a matrix-vector product are discussed: the direct XOR-count and sequential XOR-count. Then different Gaussian-elimination-based algorithms to find the sequential XOR-count are discussed and implemented . We also consider Boyar-Peralta's algorithm for finding linear circuits with low XOR-counts. These algorithms are compared for runtime and XOR-count, which shows that Boyar-Peralta's algorithm yields lower XOR-counts than the Gaussian-elimination-based algorithms at the cost of higher runtime.

Next, we consider multiplication by an unknown element. Here the resulting matrix has expressions as entries, which are optimized with a new algorithm based on Boyar-Peralta's algorithm. Then the XOR-count and runtime are experimentally evaluated. Finally, it is shown that with the aforementioned algorithm, matrices obtained using the normal basis have a higher average XOR-count and higher average runtime compared to matrices obtained using the polynomial basis.

Contents

1 Introduction

In the modern world, a significant number of devices are connected to the Internet. Many of these devices, particularly within the Internet of Things (IoT), have strict performance, power consumption and size constraints, but are still required to encrypt sensitive data. This motivated the development of lightweight cryptographic schemes.

For illustrative purposes, the Advanced Encryption Standard (AES) may be considered. Although it was not specifically designed to be lightweight, it is very efficient on modern CPUs. Many modern CPUs have special instructions to perform encryption [1]. IoT devices require similar instructions to have efficient encryption capabilities. However, implementation proves to be difficult, because of the resource constraints. This has caused some devices to communicate without any form of encryption in an attempt to increase lifetime [2].

AES, like many other encryption standards, uses linear layers which can be represented as an $\mathbb{F}_{2^m}$-linear mapping $\mathbb{F}_{2^m} \to \mathbb{F}_{2^m}$. Many encryption algorithms operate in $\mathbb{F}_{2^m}$ because, with current technology, binary is a natural choice. Elements in binary finite fields can be represented as bit strings internally and allow for bitwise operations, which are efficient to work with in circuits.

We measure the efficiency of a whole circuit using the number of XOR gates, because in the circuits we will look at the number of AND gates is consistent. Calculating the minimum number of XOR gates required for a computation in general is difficult. Indeed, finding the minimum number of XOR operations required is called the Shortest Linear Program (SLP) problem, which has been shown to be NP-hard [3].

1.1 Our contribution

For multiplication in binary finite fields where one variable is a fixed constant, we will represent the fixed element as a matrix and the second unknown input as a vector. We provide an explanation for 2 metrics used to estimate the implementation cost of this matrix-vector multiplication, followed by discussing several algorithms to actually compute the XOR-count of the multiplication. Multiple modern optimization techniques based on Gaussian elimination and Boyar-Peralta's algorithm are implemented and compared based on runtime and XOR-count. This will show that Boyar-Peralta has lower average XOR-count, but takes longer to finish.

Then we extend this to multiplication in a binary finite field where both variables are unknown. One of the variables is represented as a matrix where the entries are expressions, then we apply Boyar-Peralta's algorithm to optimize the matrix construction.

We perform an experimental evaluation of the considered algorithms and compare their effectiveness and runtime on randomly generated instances. Together, these results provide a practical framework for applying linear circuit optimization techniques to a broader class of finite-field multiplication problems.

2 Background and Related Work

2.1 Mathematical Background

We begin with the mathematical background needed to understand what binary finite fields are. For a more comprehensive study, we suggest reading [4].

2.1.1 Groups

Definition 1 [Group]. A group is a non-empty set G with a closed binary operation $*$, such that it satisfies

1. All elements $a, b, c \in G$ are associative
$$(a * b) * c = a * (b * c). \tag{1}$$
2. G contains an identity element $e \in G$ such that $e * g = g = g * e$ for all $g \in G$.
3. For all elements $g \in G$, there exists an inverse g^{-1} such that $g^{-1} * g = e = g * g^{-1}$.

An operation $*$ is a map $* : G \times G \to G$. If $*$ is commutative, that is $g * h = h * g$ for all $g, h \in G$, then it is usually denoted by $+$ and the group is called *abelian*. Examples of groups are the integers $\mathbb{Z}$ under standard addition, real numbers $\mathbb{R}$ under standard addition and $\mathbb{R} \setminus \{0\}$ under multiplication. The natural numbers $\mathbb{N}$ is not a group under addition or multiplication since the elements lack an inverse in both cases.

2.1.2 Homomorphisms

Let G, H be groups.

Definition 2 [homomorphism]. A mapping $f : G \to H$ is called a homomorphism if for every pair $g, h \in G$
$$f(g) * f(h) = f(g * h). \tag{2}$$

Homomorphisms are often said to preserve algebraic structure. A homomorphism is called an isomorphism if it is also bijective. If an isomorphism between G and H exists, then G and H are called isomorphic. Isomorphic groups are the "same" group with a different representation and that is why we denote isomorphisms using $G \cong H$. We make this clear with an example.

Example 1. Let $G = \{0, 1\}$ with standard addition modulo 2 and $H = \{\checkmark, \smile\}$ with the following operations:
$$\begin{aligned} \checkmark + \checkmark &= \checkmark \\ \smile + \smile &= \checkmark \\ \checkmark + \smile &= \smile . \end{aligned} \tag{3}$$

Clearly H and G have no elements in common. The elements of H do not even have any concrete meaning. If you replace $\checkmark$ with 0 and $\smile$ with 1, you will see that G and H represent the exact same thing. That is, H and G are isomorphic with the isomorphism $f : H \to G$ given by
$$\begin{aligned} f(\checkmark) &= 0 \\ f(\smile) &= 1. \end{aligned} \tag{4}$$

This is also visualized in Figure 1.

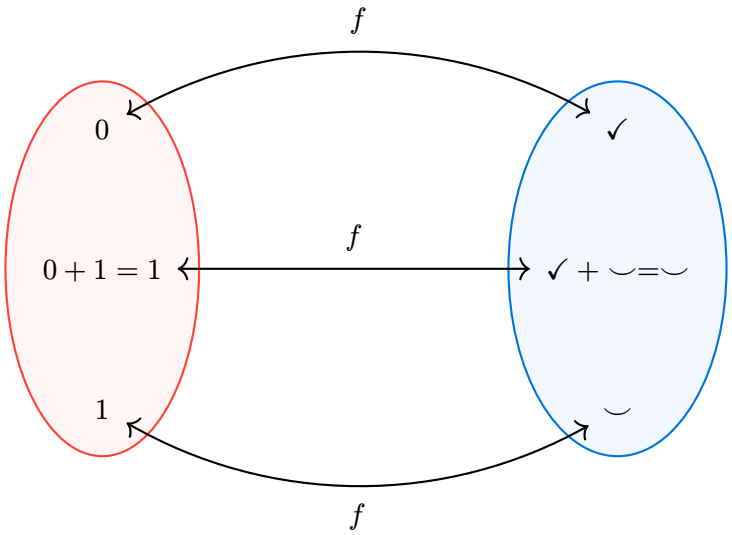

Figure 1: Visualisation for isomorphism f

2.1.3 Rings

Definition 3 [ring]. A ring is a commutative group R under $+$ that also has a second operation $*$ that satisfies

1. All elements $x, y, z \in R$ have the associative property
$$(x * y) * z = x * (y * z). \tag{5}$$
2. All elements $x, y, z \in R$ have the distributive property
$$x(y + z) = xy + xz \quad \text{and} \quad (x + y)z = xz + yz. \tag{6}$$
3. R contains an identity element $1 \in R$ such that $1 * x = x = x * 1$ for all $x \in R$.

The identity element for addition is naturally denoted with 0. If $*$ is commutative, then we call the ring commutative. Notice that $*$ does not require an inverse, but $+$ does.

2.1.4 Polynomial rings

Every ring R can form a polynomial ring $R[X]$ in the variable X with coefficients from the base ring R, such that every element in $R[X]$ is of the form

$$\sum_{k=0}^{n} r_k X^k \tag{7}$$

with coefficients $r_k \in R$ and $n \in \mathbb{N}$. The highest power above the X is called the degree of the polynomial. Constants, including 0, have degree 0. Note that polynomials with an infinite number of terms are **not** part of $R[X]$.

The operations of polynomials are standard: Addition is termwise and multiplication is done with the distributive property.

2.1.5 Fields

Definition 4 [field]. A field is a commutative ring, where every element, except for 0, has an inverse.

Examples of fields are the real numbers $\mathbb{R}$ and the reational numbers $\mathbb{Q}$ with standard addition and multiplication.

Finite fields are fields with a finite number of elements. It can be proven that all finite fields with the same number of elements are isomorphic. That is why we refer to a finite field with n elements as if it was unique. We will use $\mathbb{F}_n$ to mean the field with n elements.

The field $\mathbb{F}_n$ does not exist for all $n \in \mathbb{N}$. It only exists when n is of the form $n = p^m$ where p is a prime number and $m \in \mathbb{N}$.

In the case that $m = 1$ we have $\mathbb{F}_p$. This field is defined as the set $\{0, 1, ..., p-1\}$ with standard operators $+$ and $*$ modulo p. Integers modulo n are denoted as $\mathbb{Z}/n\mathbb{Z}$, so $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$. For general n this would **not** form a field, because not every element has an inverse. For example $3 \in \mathbb{Z}/6\mathbb{Z}$ has no multiplicative inverse, since 3 times an odd number is 3 modulo 6 and 3 times an even number is 0 modulo 6.

A binary finite field is a finite field of the form $\mathbb{F}_{2^m}$.

2.1.6 Vector spaces

Definition 5 [vector space]. A vector space over a field F is a non-empty set V with a binary operation written as $v + w$ for $\boldsymbol{v}, \boldsymbol{w} \in V$, called vector addition and a binary operation written as $a\boldsymbol{v}$ for $\boldsymbol{v} \in V$ and $a \in F$, called scalar multiplication that satisfies the following seven axioms:

1. Vector addition is associative.
2. Vector addition is commutative.
3. There is an identity element $\mathbf{0} \in V$.
4. Every $\boldsymbol{v} \in V$ has an inverse $-\boldsymbol{v} \in V$, such that $\boldsymbol{v} + (-\boldsymbol{v}) = \mathbf{0}$.
5. Comaptibility with scalar multiplication $a(b\boldsymbol{v}) = (ab)\boldsymbol{v}$.
6. The identity element $1 \in F$ is also an identity for scalar multiplication $1 \cdot \boldsymbol{v} = \boldsymbol{v}$.
7. Scalar multiplication is distributive with respect to vector addition $a(\boldsymbol{v} + \boldsymbol{w}) = a\boldsymbol{v} + a\boldsymbol{w}$.
8. Scalar multiplication is distributive with respect to field addition $(a + b)\boldsymbol{v} = a\boldsymbol{v} + b\boldsymbol{v}$.

Every field is trivially a vector space over itself. A binary finite field may also be viewed as a vector space over $\mathbb{F}_2$.

Definition 6 [linear map]. For two vector spaces V, W a mapping $f : V \to W$ is called linear if $f(\boldsymbol{v} + \boldsymbol{w}) = f(\boldsymbol{v}) + f(\boldsymbol{w})$ and $f(a\boldsymbol{v}) = af(\boldsymbol{v})$ for all $\boldsymbol{v}, \boldsymbol{w} \in V$ and $a \in F$.

Definition 7 [basis]. A basis of a vector space V is a non-empty linearly independent set B such that every element of V is a linear combination of elements in B.

If B has n elements, we call V an n-dimensional vector space. Let f be a linear map and $\boldsymbol{v} \in V$, then we can write $v = a_1\boldsymbol{b_1} + ... + a_n\boldsymbol{b_n}$ for $a_i \in F$ and $\boldsymbol{b_i} \in B$. We find that

$$\begin{aligned} f(\boldsymbol{v}) &= f(a_1\boldsymbol{b_1} + ... + a_n\boldsymbol{b_n}) \\ &= a_1 f(\boldsymbol{b_1}) + ... + a_n f(\boldsymbol{b_n}). \end{aligned} \tag{8}$$

The result of a linear mapping only depends on the image of the basis elements, so we can define a linear mapping using these images.

Definition 8 [matrix]. The matrix of a linear function f with respect to a basis $B = \{\boldsymbol{b_1}, ..., \boldsymbol{b_n}\}$ is defined as

$$[f]_B = \begin{pmatrix} | & & | \\ f(\boldsymbol{b_1}) & ... & f(\boldsymbol{b_n}) \\ | & & | \end{pmatrix}. \tag{9}$$

There are many ways to define a matrix, but this one makes the dependence on a basis clear. The dimension of a matrix is $n \times m$ where n is the number of rows and m is the number of columns. This matrix would then represent a linear map from an m-dimensional vector space to an n-dimensional vector space.

Multiplication with a fixed element in F is linear, because a field is distributive and commutative. This means we can represent multiplication with a fixed element as matrix-vector multiplication with a fixed matrix.

2.1.7 Field extensions

A field extension of a field K is a field L such that $K \subset L$, so $\mathbb{R}$ is a field extension of $\mathbb{Q}$ and $\mathbb{C}$ is a field extension of $\mathbb{R}$ and $\mathbb{Q}$.

Definition 9 [degree]. The degree $[L : K]$ of a field extension L over K is the dimension of L as vector space over K.

For example $\mathbb{C}$ as vector space over $\mathbb{R}$ has degree two because it has basis $\{1, i\}$.

There is an easy way to construct a field extension of a field K using a process called *adjoining* a root of a *monic* and *irreducible* polynomial $f \in K[X]$ of degree n. A monic polynomial has leading coefficient 1 and irreducible means that it cannot be written as the product of two other polynomials of degree 1 or higher. This means polynomials of degree 1 and 0 are always irreducible. Suppose α is a root of f, that is $f(\alpha) = 0$, then we can construct the field $F = K(\alpha)$, where $K(\alpha)$ is the smallest field that contains α and K. That is a field where elements are polynomials in the variable α. Since f has degree n every α^m can be expressed as a linear combination of $\{1, \alpha, ..., \alpha^{n-1}\}$ and this set is linearly independent, so the set forms a basis. This is called the *polynomial basis*. It follows that $|F| = |K|^n$ if $|K|$ is finite.

Example 2. Consider $\mathbb{F}_2$ and we want to find $\mathbb{F}_{2^4}$. First we find a monic irreducible polynomial with coefficients in $\mathbb{F}_2$ of degree 4, so for example $f(x) = x^4 + x^3 + x^2 + x + 1$. Let α be a root of this polynomial, then $\mathbb{F}_2(\alpha) = \mathbb{F}_{2^4}$. We know that

$$\begin{aligned} f(\alpha) &= 0 \\ \alpha^4 + \alpha^3 + \alpha^2 + \alpha + 1 &= 0 \\ \alpha^4 &= \alpha^3 + \alpha^2 + \alpha + 1 \text{ , because } -1 \bmod 2 = 1 \bmod 2. \end{aligned} \tag{10}$$

This means we can rewrite every α^n as a sum of $1, \alpha, \alpha^2, \alpha^3$.

We will also show how to construct a matrix for the multiplication with the element $A = 1 + \alpha + \alpha^3$ with the polynomial basis. Multiply each basis element by A, so

$$
\begin{aligned}
A &= 1 + \alpha + \alpha^3 \\
\alpha A &= \alpha + \alpha^2 + \alpha^4 \\
&= \alpha + \alpha^2 + (\alpha^3 + \alpha^2 + \alpha + 1) \\
&= 1 + \alpha^3 \\
\alpha^2 A &= \alpha + \alpha^4 \\
&= \alpha + (\alpha^3 + \alpha^2 + \alpha + 1) \\
&= 1 + \alpha^2 + \alpha^3 \\
\alpha^3 A &= \alpha + \alpha^3 + \alpha^4 \\
&= \alpha + \alpha^3 + (\alpha^3 + \alpha^2 + \alpha + 1) \\
&= 1 + \alpha^2.
\end{aligned} \tag{11}
$$

The vector representation of the polynomials would simply be the coordinate vector of each polynomial, so the matrix is

$$
\begin{pmatrix}
1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 \\
1 & 1 & 1 & 0
\end{pmatrix}. \tag{12}
$$

2.2 Implementation Efficiency

2.2.1 XOR and similar metrics

To estimate how efficient a matrix over $\mathbb{F}_2$ is, different metrics have been introduced: *direct XOR-count* (d-XOR), *sequential XOR-count* (s-XOR) [5] and *general XOR-count* (g-XOR) [6].

Definition 10 [direct XOR-count]. The direct XOR-count of an invertible $n \times n$ matrix M over $\mathbb{F}_2$ is $\omega(M) - n$, where ω counts the number of non-zeros in a matrix.

One would expect the XOR-count to always be at least zero, since you cannot use a negative number of XOR gates. This is indeed the case, because an invertible matrix requires at least one non-zero element in every column and every row. The d-XOR-count is straightforward to compute and gives an upper bound for the number of XOR gates required, but it is not capable of subexpression elimination.

Example 3.

$$
\begin{pmatrix}
1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1
\end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} (x_1 + x_2) \\ (x_3 + x_4) \\ (x_1 + x_2) + x_3 \\ x_2 + (x_3 + x_4) \end{pmatrix}. \tag{13}
$$

This can be computed with 4 XOR gates by reusing the intermediate results, which we call subexpressions, but the direct XOR-count is $10 - 4 = 6$.

The number of ones in a vector is called the Hamming weight. Similarly the Hamming weight of a matrix is the sum of the Hamming weights of its rows. This means the d-XOR-count can also be written as

$\mathrm{HW}(M) - n$ for an $n \times n$ matrix M. The sequential XOR-count give better estimates most of the time, because it is capable of using subexpressions.

Definition 11 [sequential XOR-count]. The sequential XOR-count of an invertible $n \times n$ matrix M is the minimal number t such that

$$M = \prod_{k=1}^{t} A_{i_k,j_k} \cdot P \tag{14}$$

where P is a permutation matrix and A_{i_k,j_k} is an addition matrix.

Equation 14 is called the s-XOR representation. A permutation matrix is a matrix with exactly one '1' in every column and every row. As the name implies, a permutation matrix permutes the rows under left multiplication and permutes the columns under right multiplication. Define $E_{i,j}$ as the matrix with exactly one '1' in the i-th row and j-th column. Then define the addition matrix $A_{i,j}$ as $A_{i,j} = I + E_{i,j}$ where I is the identity matrix. The addition matrix $A_{i,j}$ adds the j-th row to the i-th row under left multiplication and adds the j-th column to the i-th column under right multiplication.

If we return to Example 3 we will find

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} = A_{4,2}A_{3,1}A_{2,3}A_{1,4} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \tag{15}$$

so the s-XOR-count is at most 4. None of the rows have exactly one 1, so every row needs atleast one addition. Therefore the s-XOR-count is at least 4. It follows that the s-XOR-count is indeed 4. The same as the number of XOR gates required we found by manually looking. Performing this decomposition manually becomes impossible for large matrices, so we want an algorithm that finds the s-XOR-count for us.

It is important to remember that despite the fact that s-XOR-count can take subexpressions into consideration, it is not *always* lower than the d-XOR-count, see [5] for an example.

Finding the true lowest number of XOR gates required is called the shortest linear program (SLP) problem. The solution to this problem is called the general XOR-count. This has no formula.

2.2.2 Related work

For estimating the g-XOR-count there are many strategies, like computing the s-XOR-count. There are many ways to find the s-XOR-count, like graph-based searches that reduce the problem to a shortest-path problem [7], [8]. Research is also being done into the theoretical properties of s-XOR-count [5]. The s-XOR-count is similar to finding the optimal pivots for Gaussian elimination, which has its own research. Usually this is done with heuristics like Markowitz [9], but there is also research into using machine learning [10]. Additionally, there is research into making Gaussian elimination run faster by parallelizing the algorithm [11], [12]

In general most research into SLP is related to Boyar-Peralta [13]. This is with respect to improving the result of the algorithm and modifying the algorithm to different needs. Improvements can be made using different heuristics [14] and by changing what to do during ties [15]. There is also a lot of research into reducing latency or depth of the circuit [16], [17]. Adapting Boyar-Peralta to solve SLP with 3-input XOR gates is also possible [18]. Some of these will be explained more in Section 3.1.7

Before Boyar-Peralta, a common algorithm was Paar [19]. This algorithm came out before Boyar-Peralta and is generally faster, but with a worse result. It works by finding the most common pair of elements and replacing it with a new intermediate variable. Do this until all functions are solved.

2.3 GAP

To implement the algorithms in this thesis, we will use the system GAP. GAP, which stands for Groups, Algorithms and Programming, is an open-source system for computational discrete algebra. In addition to providing a high-level programming language, GAP includes an extensive library of algebraic algorithms and data structures for objects such as groups, rings, vector spaces, finite fields and combinatorial structures. These capabilities make GAP good for implementing algorithms that involve finite field arithmetic and symbolic algebraic manipulation.

Algorithms are presented throughout this thesis using pseudocode, but the actual implementations are written in GAP. In addition to the standard GAP library, the implementation relies on the FFCSA package [20], which depends on the FSR package [21]. The FFCSA package provides functionality for finite field constructions, searches and algorithms. We will mainly be using it to create matrices that represent multiplication by an element in a given binary finite field.

3 Algorithms

3.1 Multiplication with one fixed input

3.1.1 The meaning of XOR-counts

For fixed inputs in $\mathbb{F}_{2^n}$ we precompute the matrix M over $\mathbb{F}_2$ associated with the fixed input. The XOR counts of this matrix provide an upper bound on the minimum number of XOR gates required to implement the circuit that will perform multiplication. For example, the d-XOR-count gives the number of XOR-gates required for the naive implementation. The d-XOR-count is very easy to compute compared to the s-XOR-count. See Algorithm 1.

Algorithm 1: d-XOR-count

Input : An invertible $n \times n$ matrix M
Output: The d-XOR-count
1: **return** HW(M) - n

However, it seems the s-XOR-count is smaller most of the time. Therefore, if we want to compute the minimum number of XOR gates algorithmically, it makes sense to consider this representation. Our goal is to compute the s-XOR-count for a given input matrix M.

In this section, an s-XOR representation is any product $\prod_{k=1}^{t} A_{i_k,j_k} \cdot P$ that equals M. It is not required that t is minimal.

3.1.2 From s-XOR-representation to subexpressions

We first describe how to interpret a given s-XOR-representation

$$M = \prod_{k=1}^{t} A_{i_k,j_k} \cdot P. \tag{16}$$

We evaluate the product from right to left. The rightmost matrix is P, a permutation matrix. Applying P requires no XOR gates; it simply reorders the inputs. Each row represents an intermediate value, obtained by XOR-ing all input variables corresponding to non-zero entries in that row. Left multiplication by $A_{i,j}$ adds row j to row i. This corresponds to computing the XOR of the intermediate values in positions i and j and storing the result in position i. This algorithm is described in Algorithm 2.

Example 4. We will show this algorithm on the matrix from Example 3

$$M = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} = A_{4,2}A_{3,1}A_{2,3}A_{1,4}P, \text{ where } P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}. \tag{17}$$

We start with the permuted input vector defined by P,

$$y_0 = P \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_4 \\ x_3 \\ x_2 \end{pmatrix}. \tag{18}$$

Applying $A_{1,4}$:

$$y_1 = A_{1,4} \cdot y_0 = \begin{pmatrix} x_1 + x_2 \\ x_4 \\ x_3 \\ x_2 \end{pmatrix}. \tag{19}$$

Applying $A_{2,3}$:

$$y_2 = A_{2,3} \cdot y_1 = \begin{pmatrix} x_1 + x_2 \\ x_3 + x_4 \\ x_3 \\ x_2 \end{pmatrix}. \tag{20}$$

Applying $A_{4,2}$:

$$y_3 = A_{4,2} \cdot y_2 = \begin{pmatrix} x_1 + x_2 \\ x_3 + x_4 \\ x_3 \\ x_2 + (x_3 + x_4) \end{pmatrix}. \tag{21}$$

Applying $A_{3,1}$:

$$y_4 = A_{3,1} \cdot y_3 = \begin{pmatrix} x_1 + x_2 \\ x_3 + x_4 \\ (x_1 + x_2) + x_3 \\ x_2 + (x_3 + x_4) \end{pmatrix}. \tag{22}$$

After applying all matrices, we obtain the output defined by M, with all reused subexpressions shown in brackets.

Algorithm 2: s-XOR representation to circuit

Input : An invertible $n \times n$ matrix M
Output: A vector y containing functions corresponding to every row of M

Find s-XOR representation $M = \prod_{k=1}^{t} A_{i_k,j_k} \cdot P$ # Implemented later
$y \leftarrow P \cdot [x_1, ..., x_n]$ # x_i is symbolic input
for k **in** $[t, ..., 1]$ **do**
  $y[i_k] \leftarrow y[i_k] \oplus y[j_k]$
end
return y

Finding the s-XOR representation is similar to Gaussian elimination. For an invertible square matrix, reduced row echelon form is the identity matrix. In our case, however, it is enough to reduce the matrix to a permutation matrix, which allows for additional flexibility and optimizations.

3.1.3 Naive Gaussian

We begin with a standard Gaussian elimination strategy without row permutations. For each column, we select a pivot row with a non-zero entry and eliminate all other non-zero entries in that column by adding the pivot row as described in Algorithm 3.

Example 5. We illustrate Algorithm 3 with the same matrix from the previous example and will represent the locked rows with an $*$:

$$M = \left(\begin{array}{cccc|c} 1 & 1 & 0 & 0 & \\ 0 & 0 & 1 & 1 & \\ 1 & 1 & 1 & 0 & \\ 0 & 1 & 1 & 1 & \end{array}\right). \tag{23}$$

We start in the first column where the first row has a non-zero entry, so that will be our pivot row, which will be added to row 3:

$$\left(\begin{array}{cccc|c} 1 & 1 & 0 & 0 & * \\ 0 & 0 & 1 & 1 & \\ 0 & 0 & 1 & 0 & \\ 0 & 1 & 1 & 1 & \end{array}\right). \tag{24}$$

In the second column the first row with a non-zero entry is row 1, but that row is locked so we move on to the next row with a non-zero entry: row 4. We add this to row 1 and lock row 4:

$$\left(\begin{array}{cccc|c} 1 & 0 & 1 & 1 & * \\ 0 & 0 & 1 & 1 & \\ 0 & 0 & 1 & 0 & \\ 0 & 1 & 1 & 1 & * \end{array}\right). \tag{25}$$

In the third column, row 1 again has a non-zero entry, but it is locked, so the algorithms selects row 2, which will be added to every other row:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 0 & * \\ 0 & 0 & 1 & 1 & * \\ 0 & 0 & 0 & 1 & \\ 0 & 1 & 0 & 0 & * \end{array}\right). \tag{26}$$

There is only one row not locked, so row 3 is selected and added to row 2:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 0 & * \\ 0 & 0 & 1 & 0 & * \\ 0 & 0 & 0 & 1 & * \\ 0 & 1 & 0 & 0 & * \end{array}\right). \tag{27}$$

The number of row additions required was $1 + 1 + 3 + 1 = 6$.

Algorithm 3: Naive Gaussian elimination

Input : An invertible $n \times n$ matrix M
Output: An array *Operations* with pairs (i, j) that represent $A_{i,j}$ and the resulting permutation matrix

```
1:  LockedRows ← [];
2:  Operations ← [];
3:  for PivotColumn in [1, ..., n] do
4:      for PivotRow in [1, ..., n] do
5:          if M[PivotRow][PivotColumn] = 0 or PivotRow in LockedRows then
6:              continue
7:          fi
8:          LockedRows.push( pivotRow )
9:          for Row in [1, ..., n] do
10:             if M[Row][PivotColumn] ≠ 0 and Row ≠ PivotRow then
11:                 M[Row] ← M[Row] ⊕ M[PivotRow]
12:                 Operations.push( (Row, PivotRow) )
13:             fi
14:         end
15:     end
16: end
17: return Operations, M
```

3.1.4 Exhaustive search

The easiest way to find the least number of row operations is simply to try every possible pivot row and pivot column. See Algorithm 4

Algorithm 4: Gaussian elimination with Exhaustive search

Input : An invertible $n \times n$ matrix M
Output: An array *bestOps* with row additions and the resulting permutation matrix P

```
1:  BestOps ← None
2:  P ← None
3:  Search ← function(M, Ops, LockedRows, LockedColumns)
4:      if BestOps and Length(Ops) >= Length(BestOps) then
5:          return Fail
6:      fi
7:      if Length(LockedRows) = n then
8:          if BestOps = None or Length(Ops) < Length(BestOps) then
9:              BestOps ← copy(Ops)
10:             P ← copy(M)
11:         fi
12:         return Fail
13:     fi
14:     for PivotRow in [1, ..., n] \ LockedRows do
15:         for PivotColumn in [1, ..., n] \ LockedColumns do
16:             if M[PivotRow][PivotColumn] = 0 then
17:                 continue
18:             fi
19:             NewM ← copy(M)
20:             NewOps ← copy(Ops)
21:             for Row in [1, ..., n] do
22:                 if NewM[Row][PivotColumn] ≠ 0 and Row ≠ PivotRow then
23:                     NewM[Row] ← NewM[Row] ⊕ NewM[PivotRow]
24:                     NewOps.push( (Row, PivotRow) )
25:                 fi
26:             end
27:             Search(
28:                 NewM,
29:                 NewOps,
30:                 LockedRows ∪ [PivotRow])),
31:                 LockedColumns ∪ [PivotColumn]))
32:             )
33:         end
34:     end
35: end
36: Search(M, [], [], [])
37: return BestOps, P
```

Running this algorithm shows that 4 is indeed the lowest number of row additions required for the example. In this case we could also prove it using the fact that every row needs at least one row addition, but in general this algorithm can be used to compare other algorithms. Exhaustive search works for only very small n. A more realistic version would use a heuristic to select both pivot rows and pivot columns.

3.1.5 Heuristic

A common strategy is partial pivoting, which selects the row with the largest absolute value in a column in order to improve numerical stability [22].

Since we work over $\mathbb{F}_2$, numerical stability is not a concern. Instead, we aim to choose pivots that lead to better s-XOR representations. Gaussian elimination with a heuristic is given by Algorithm 5.

Algorithm 5: Gaussian elimination with Heuristic

Input : An invertible $n \times n$ matrix M
Output: An array *Operations* with pairs (i, j) that represent $A_{i,j}$ and the resulting permutation matrix

```
1:  LockedRows ← []
2:  LockedColumns ← []
3:  Operations ← []
4:  repeat n times
5:      (PivotRow, PivotColumn) ← pickPivot(M, n, LockedRows, LockedColumns) # pickPivot implementation depends on the heuristic.
6:      LockedRows.push(PivotRow)
7:      LockedColumns.push(PivotColumn)
8:      for Row in [1, ..., n] do
9:          if Row ≠ PivotRow and M[Row][PivotColumn] ≠ 0 then
10:             M[Row] ← M[Row] ⊕ M[PivotRow]
11:             Operations.push( (Row, PivotRow) )
12:         fi
13:     end
14: end
15: return Operations, M
```

In the example of the Naive implementation, we saw some undesirable situations. After the first step, we already obtained a row with a single non-zero entry, but this structure was destroyed in a later step. Additionally, some row additions increased the number of non-zero entries in certain rows, as seen for row 1 in Equation 25.

We estimate how much choosing a particular pivot will increase the number of non-zero entries in the matrix. Let the pivot row have Hamming weight w, and let k be the number of rows with a non-zero entry in the candidate column. Eliminating this column requires $k-1$ row additions, each of which can introduce up to $w-1$ new non-zero entries. This gives the cost estimate

$$(k-1) \cdot (w-1). \tag{28}$$

We select the pivot column and pivot row that minimize this cost. This heuristic is called the Markowitz heuristic [10]. This is implemented in Algorithm 6. In [10] it is shown that for small sparse matrices up to 8×8 matrices a strategy created with machine learning is better than Markowitz. We also want to use bigger matrices up to 16×16 matrices, so we will not use this strategy.

Algorithm 6: `pickPivot()` for Markowitz heuristic

Input : An $n \times n$ matrix M and an array *lockedRows*
Output: *PivotRow* and *PivotColumn* with best pivot according to Markowitz heuristic

```
1:  PivotRow ← None
2:  PivotColumn ← None
3:  BestHeuristic ← ∞
4:  for Row in [1, ..., n] \ LockedRows do
5:      HammingWeight ← sum(M[Row])
6:      for Column in Columns do
7:          if M[Row][Column] = 0 then
8:              continue
9:          fi
10:         ColumnWeight ← sum(Column M)
11:         Heuristic ← (ColumnWeight - 1) · (HammingWeight - 1)
12:         if Heuristic < BestHeuristic then
13:             PivotColumn ← Column
14:             PivotRow ← Row
15:             BestHeuristic ← Heuristic
16:         fi
17:     end
18: end
19: return PivotRow, PivotColumn
```

Example 6. We will illustracte Algorithm 5 with the Markowitz heuristic Algorithm 6 on our example to see if it is an improvement:

$$M = \left(\begin{array}{cccc|} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{array}\right). \tag{29}$$

The lowest Hamming weight is 2 and the lowest column weight is also 2, so the lowest heuristic possible is 3, so the algorithms selects row 1 and column 1:

$$\left(\begin{array}{cccc|c} 1 & 1 & 0 & 0 & * \\ 0 & 0 & 1 & 1 & \\ 0 & 0 & 1 & 0 & \\ 0 & 1 & 1 & 1 & \end{array}\right). \tag{30}$$

Now the algorithms selects row 3, since it has a single non-zero entry which means the heuristic is 0. We can only choose one column, so this yields

$$\left(\begin{array}{cccc|c} 1 & 1 & 0 & 0 & * \\ 0 & 0 & 0 & 1 & \\ 0 & 0 & 1 & 0 & * \\ 0 & 1 & 0 & 1 & \end{array}\right). \tag{31}$$

Now the algorithms selects row 2, since it has a single non-zero entry. We can again only choose one column, so this yields

$$\left(\begin{array}{cccc|c} 1 & 1 & 0 & 0 & * \\ 0 & 0 & 0 & 1 & * \\ 0 & 0 & 1 & 0 & * \\ 0 & 1 & 0 & 0 & \end{array}\right). \tag{32}$$

and now there is only one option left:

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & 0 & * \\ 0 & 0 & 0 & 1 & * \\ 0 & 0 & 1 & 0 & * \\ 0 & 1 & 0 & 0 & * \end{array}\right). \tag{33}$$

The number of row additions is now $1 + 2 + 1 + 1 = 5$, which is closer to what we manually came up with.

3.1.6 Lookahead

Instead of estimating how many non-zero entries will be added by picking a pivot row and pivot column, we can simply calculate it. This means we add the candidate pivot row to every row with a non-zero entry in the pivot column and calculate how many ones were added, which we will call *Cost*. The *Cost* can be negative.

The goal is to reduce the number of row additions, so after computing the *Cost* we divide it by the number of row additions. This final number is used as the heuristic that we want to minimize. With Markowitz we only look at one column and one row, but here we have to look at many rows, making the algorithm generally more computationally expensive, but possibly giving better pivots. This is implemented in Algorithm 7.

Algorithm 7: `pickPivot()` for Lookahead heuristic

Input : An $n \times n$ matrix M and arrays *lockedRows* and *lockedColumns*
Output: *BestRow* and *BestColumn* with best pivot according to look ahead heuristic

BestHeuristic $\leftarrow \infty$
BestRow $\leftarrow$ None
BestColumn $\leftarrow$ None
for *PivotRow* **in** $[1, ..., n] \setminus$ LockedRows **do**
  for *PivotColumn* **in** $[1, ..., n] \setminus$ LockedColumns **do**
    if $M[PivotRow][PivotColumn] \neq 0$ **then**
      continue
    fi
    Cost $\leftarrow 0$
    HammingWeight $\leftarrow \text{sum}(M[PivotRow])$
    MoveCount $\leftarrow 0$
    for *Row* **in** $[1, ..., n]$ **do**
      if *Row* $\neq$ *PivotRow* **and** $M[Row][PivotColumn] \neq 0$ **then**
        MoveCount $\leftarrow$ *MoveCount* + 1
        Cost $\leftarrow$ *Cost* + *HammingWeight* - 2 **M[*Row*]** M[*PivotRow*]
      fi
    end
    Heuristic $\leftarrow$ *Cost* / *MoveCount*.
    if *Heuristic* $<$ *BestHeuristic* **then**
      BestHeuristic $\leftarrow$ *Heuristic*
      BestRow $\leftarrow$ *PivotRow*
      BestColumn $\leftarrow$ *PivotColumn*
    fi
  end
end
return *BestRow*, *BestColumn*

Example 7. We perform Algorithm 5 with the Lookahead heuristic Algorithm 7 matrix:

$$M = \left(\begin{array}{cccc|c} 1 & 1 & 0 & 0 & \\ 0 & 0 & 1 & 1 & \\ 1 & 1 & 1 & 0 & \\ 0 & 1 & 1 & 1 & \end{array}\right). \tag{34}$$

The first option is pivot row 1 and pivot column 1 we obtain a cost of -2, since we remove two non-zero entries in row 3. If we pick pivot row 1 and pivot column 2 we would add row 1 to row

3 and 4, which only reduces the number of non-zero entries by 2. There are 2 row additions and XOR-count is reduced by 2, so the heuristic is -1, which is worse compared to the first option.

$$\left(\begin{array}{cccc|c} * & & & & \\ \hline 1 & 1 & 0 & 0 & * \\ 0 & 0 & 1 & 1 & \\ 0 & 0 & 1 & 0 & \\ 0 & 1 & 1 & 1 & \end{array}\right). \tag{35}$$

With the Markowitz heuristic we would pick row 3, which results in an s-XOR-count of 5. Because we divide by row additions this is however not optimal for this heuristic. With the same logic as before we pick pivot row 2 and pivot column 4:

$$\left(\begin{array}{cccc|c} * & & & * & \\ \hline 1 & 1 & 0 & 0 & * \\ 0 & 0 & 1 & 1 & * \\ 0 & 0 & 1 & 0 & \\ 0 & 1 & 0 & 0 & \end{array}\right). \tag{36}$$

Now there are 2 equally scored options. The first occurence is selected, so the pivot row is 3 and the pivot column is 3:

$$\left(\begin{array}{cccc|c} * & & * & * & \\ \hline 1 & 1 & 0 & 0 & * \\ 0 & 0 & 0 & 1 & * \\ 0 & 0 & 1 & 0 & * \\ 0 & 1 & 0 & 0 & \end{array}\right). \tag{37}$$

and now there is only one option

$$\left(\begin{array}{cccc|c} * & * & * & * & \\ \hline 1 & 0 & 0 & 0 & * \\ 0 & 0 & 0 & 1 & * \\ 0 & 0 & 1 & 0 & * \\ 0 & 1 & 0 & 0 & * \end{array}\right). \tag{38}$$

The s-XOR-count is now only 4. Finally we algorithmically get what we manually also found for the working example.

3.1.7 Boyar-Peralta's algorithm

The sequential XOR-count does not always give the lowest possible XOR-count even if we find the best possible s-XOR-count. This metric has another disadvantage as well: it only works for matrices that are square and invertible. Boyar-Peralta's algorithm is another algorithm to find the Shortest Linear Program (SLP) for any $m \times n$ binary matrix M [13].

Let the distance of an expression be the minimum number of additions of elements from a set S. We keep a set *Base* of already computed subexpressions, which initially are the input variables $x_1, ..., x_n$. Then we maintain the vector *Dist* of distances from the rows in M to *Base*, which initially are

$\mathrm{HW}(M[1]) - 1, ..., \mathrm{HW}(M[m]) - 1$. The goal is to make every element of the *Dist* vector 0. The algorithm proceeds as follows:

1. Generate all possible combinations of 2 elements in *Base*.
2. For every combination compute the new distance vector which will be called *DistCopy*.
3. Use a heuristic on *DistCopy* to find the best combination and set *Dist* equal to *DistCopy*.
4. Repeat until $\mathrm{sum}(Dist) = 0$.

This is shown in Figure 2. In step 3 we use a heuristic to score *DistCopy*. We will use the heuristic from the original paper [13] where you first minimize the sum of *DistCopy* and if those sums are equal we maximize the Euclidean norm. In [14] the researchers found that minimizing the norm worked better. There is no definitive better option, but we will stick to the original heuristic.

It is also possible to add randomness to the heuristic [13]. If two possible combinations have the same sum of distances, then there is a 50% chance to apply the norm heuristic and a 50% chance to keep the current best. This makes it so the algorithm is no longer deterministic, but can give better results. There is also Randomised Normal heuristic [15] where both the sum and norm are applied. If after that there are ties, pick a random combination. The main difference is that with Randomised Normal there is a uniform probability that any combination is picked. With the original random algorithm this is not the case. We want to keep the algorithm deterministic, so we will not use this. Our focus is on XOR-count, but it is also possible to do Boyar-Peralta with a Depth limit [23], which reduces the XOR gates within a limited depth.

Example 8. We will perform Boyar-Peralta's algorithm on the same example as before

$$M = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix}. \tag{39}$$

Initialize all variables:
$Base = [x_1, x_2, x_3, x_4]$
$Dist = [1, 1, 2, 2]$.

We now evaluate every possible pairwise XOR of elements in *Base* and select the candidate that yields the best distance vector. This results in $\binom{4}{2} = 6$ possible combinations. The sum heuristic yields $x_1 \oplus x_2, x_3 \oplus x_4$ or $x_2 \oplus x_3$ with the exact same sum: 4. The norms are $\sqrt{6}, 2$ and $\sqrt{6}$ respectively. Again, two sums have the same heuristic, in which case we simply pick the first one, which is $x_1 \oplus x_2$, so the first iteration yields:
$Base = [x_1, x_2, x_3, x_4, x_1 \oplus x_2]$
$Dist = [0, 1, 1, 2]$.

Now we have $\binom{5}{2} = 10$ possible combinations. The one with the lowest sum is now $x_3 \oplus x_4$, so after the second iteration we find:
$Base = [x_1, x_2, x_3, x_4, x_1 \oplus x_2, x_3 \oplus x_4]$
$Dist = [0, 0, 1, 1]$.

Now we have $\binom{6}{2} = 15$ possible combinations. The best ones are now $x_1 \oplus x_2 \oplus x_3$ or $x_2 \oplus x_3 \oplus x_4$, with the same sum and norm, so we will pick the first one again:
$Base = [x_1, x_2, x_3, x_4, x_1 \oplus x_2, x_3 \oplus x_4, x_1 \oplus x_2 \oplus x_3]$
$Dist = [0, 0, 0, 1]$.

The only possible combination is now $x_2 \oplus x_3 \oplus x_4$, so the order did not matter:
$Base = [x_1, x_2, x_3, x_4, x_1 \oplus x_2, x_3 \oplus x_4, x_1 \oplus x_2 \oplus x_3, x_2 \oplus x_3 \oplus x_4]$
$Dist = [0, 0, 0, 0]$.

This gives us a distance sum of 0, which is what we want. All rows are now part of the *Base* array, which contains all the intermediate results we have to compute.

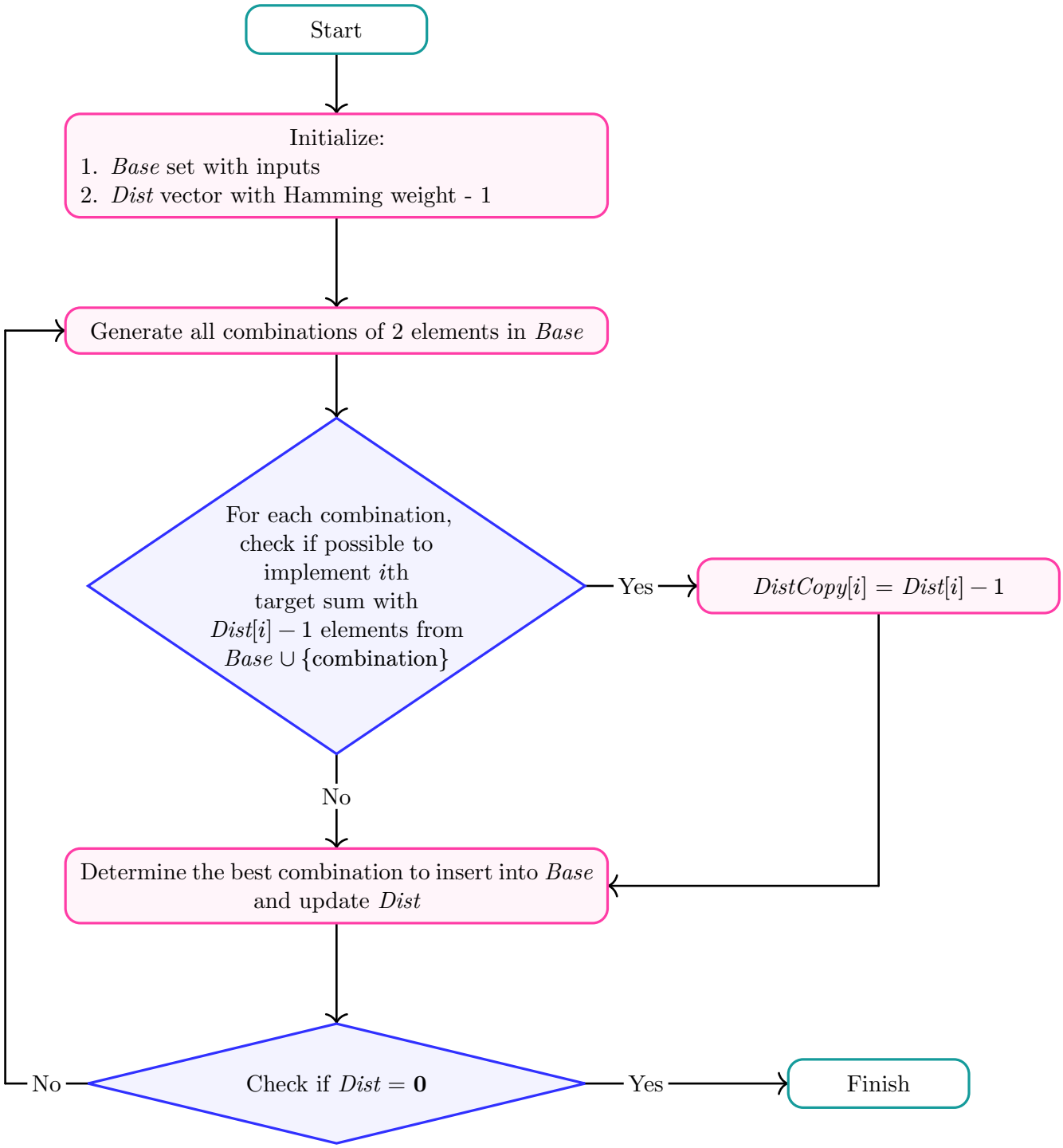

Figure 2: Flowchart for Boyar-Peralta's algorithm

As you can see the number of sums we have to check grows quite quickly and computing the new distance also gets increasingly slower as *Base* grows, so we will discuss some optimizations. In Algorithm 8 we can see the main logic of Boyar-Peralta.

Algorithm 8: `mainBoyarPeralta` entrypoint of Boyar-Peralta's algorithm

Input : An $m \times n$ matrix M
Output: An array *Base* with all intermediate results required to compute every row of M

1: $Base \leftarrow [x_1, ..., x_n]$
2: $Dist \leftarrow [\text{HW}(M[1]) - 1, ..., \text{HW}(M[n]) - 1]$
3: **while** $\text{sum}(Dist) \neq 0$ **do**
4: $Combination$, $DistCopy \leftarrow$ findBestCombination(M, $Base$, $Dist$)
5: $Base \leftarrow Base \cup [Combination]$
6: $Dist \leftarrow DistCopy$
7: **end**
8: **return** *Base*

Inside `findBestCombination` Algorithm 9 we find an optimization used in the the original paper [13]. If any two elements inside *Base* form a combination that is also a target sum, we pick this combination. This is called the pre-emptive choice. Not only does it make the code faster, it can also slightly improve the result of Boyar-Peralta. The observation one has to make is that if a combination is one of the target sums, then this combination always has to made at some point and there is no more efficient way to form this target. This result can later be used as intermediate result for the other target sums.

Notice that if adding a pair of elements in *Base* results in a target sum, which is not already in *Base*, then the distance to the corresponding entry in *Dist* would be 1. This means before checking every possible pair in *Base*, we first check if there are target sums with distance 1 and if so, that is the "best" combination. This saves us from having to call `updateDistance` as often. If the distance is 0 that target is already inside *Base*.

Algorithm 9: `findBestCombination()` in Boyar-Peralta's algorithm

Input : The target vectors M, the current *Base* and the previous distance *Dist*
Output: The optimal combination according to the heuristic and the updated distance vector

1: **for** i **in** [1 .. Length(*Dist*)] **do**
2: **if** *previousDist*[i] = 1 **then**
3: *Combination* ← copy(M[i])
4: *DistCopy* ← UpdateDistance(M, *Base*, *Combination*, *Dist*)
5: **return** *Combination*, *DistCopy*
6: **fi**
7: **end**
8: *BestCombination* ← None
9: *BestDist* ← None
10: *BestDistSum* ← None
11: *BestDistNorm* ← None
12: **for** $[b_1, b_2]$ **in** Base **do**
13: *Combination* ← $b_1 \oplus b_2$
14: *DistCopy* ← updateDist(M, *Base*, *Combination*, *Dist*)
15: *DistSum* ← sum(*DistCopy*)
16: *DistNorm* ← *DistCopy* · *DistCopy*
17: **if** *DistSum* < *BestDistSum* **or** *DistSum* = *BestDistSum* **and** *DistNorm* > *BestDistNorm* **then**
18: *BestCombination* ← *Combination*
19: *BestDist* ← *DistCopy*
20: *BestDistSum* ← *DistSum*
21: *BestDistNorm* ← *DistNorm*
22: **fi**
23: **end**
24: **return** *bestCombination*, *BestDist*

Inside Algorithm 9 we find the new distance *DistCopy* using `updateDist` Algorithm 10. The easiest way to find the new distance is to try every possible sum with the elements in *Base* to see if it adds up to your target sums, but this is very slow. There is a lot of information we already have which can be used to make it faster. Adding a combination to *Base*, can only reduce the distance by 1 and if it does, that the new combination is a part of that sum. This gives us Algorithm 10. On line 5 we call `reachable` Algorithm 11, to check if $A[i]$ + *Combination* is a sum of *Dist[i]* - 1 elements in *Base*.

Algorithm 10: `updateDist` in Boyar-Peralta's algorithm

Input : The target vector A, the current *Base*, the candidate combination *Combination* and the current distance *Dist*
Output: The new distance *NewDist*

```
1: | NewDist ← copy(Dist)
2: | for i in [1 .. length(A)] do
3: |  | if Dist[i] = 0 then
4: |  |  | continue
5: |  | elif reachable(A[i] + Combination, Dist[i] - 1, 1, Base)
6: |  |  | NewDist[i] ← Dist[i] - 1
7: |  | fi
8: | end
9: | return NewDist
```

Algorithm 11: `reachable()` in Boyar-Peralta's algorithm

Input : A target vector *Target*, the maximum number of elements that need to be added *Elements*, the starting index *Start* and the current *Base*
Output: Return true if *Target* is a sum of *Elements* or fewer elements of *Base*[*Start* .. length(*Base*)]

```
1: | if termCount(Target) ≤ Elements) then return true fi
2: | if Length(Base) - Start + 1 < Elements then return false fi
3: | if Elements = 0 then return false fi
4: | if Elements = 1 then
5: |  | for i in [ Start .. length(Base)] do
6: |  |  | if target = base[i] then return true fi
7: |  | end
8: | fi reachable(Target + Base[Start], Elements - 1, Start + 1, Base) then
9: |  | return true
10: | fi
11: | fi reachable(Target, Elements, Start + 1, Base) then
12: |  | return true
13: | fi
14: | return false
```

3.1.8 Benchmark

To see when each algorithm is optimal, we will look at a benchmark where we consider the XOR-count and the runtime. The benchmark uses all or at most 200 $n \times n$ matrices generated from random irreducible polynomials and random elements in $\mathbb{F}_{2^n}$ excluding 1 and 0, where $n = 4, 6, 8, 12, 16$.

System specifications:

- GAP Version: 4.16.0-dirty
- GAP Architecture: x86_64-pc-linux-gnu-default64-kv11
- CPU: AMD Ryzen 7 7800X3D
- Memory: 64GB
- OS: Arch Linux
- Kernel: Linux 7.1.4-arch1-1

Reference implementation is the d-XOR-count Algorithm 1. The time it takes to run all the algorithms is shown in Figure 3. Exhaustive search did not finish in a reasonable amount of time for $n = 12, 16$. Although exhaustive search is very slow, it is useful for seeing how close other Gaussian-based algorithms are to the ideal. After exhaustive search, the slowest algorithm is Boyar-Peralta, but on average it yields the lowest XOR-count for all n. See column Count under Average XOR-count in Table 1, Table 2, Table 3, Table 4 and Table 5. The best case XOR-count is also the lowest for Boyar-Peralta, which is shown in column Count under Best XOR-count. Given sufficient time, Boyar-Peralta is the best with respect to XOR-count. The other Gaussian-based algorithm run significantly faster than the aforementioned algorithms, which might make them better choices when searching through very large fields or when there is not enough time to run Boyar-Peralta.

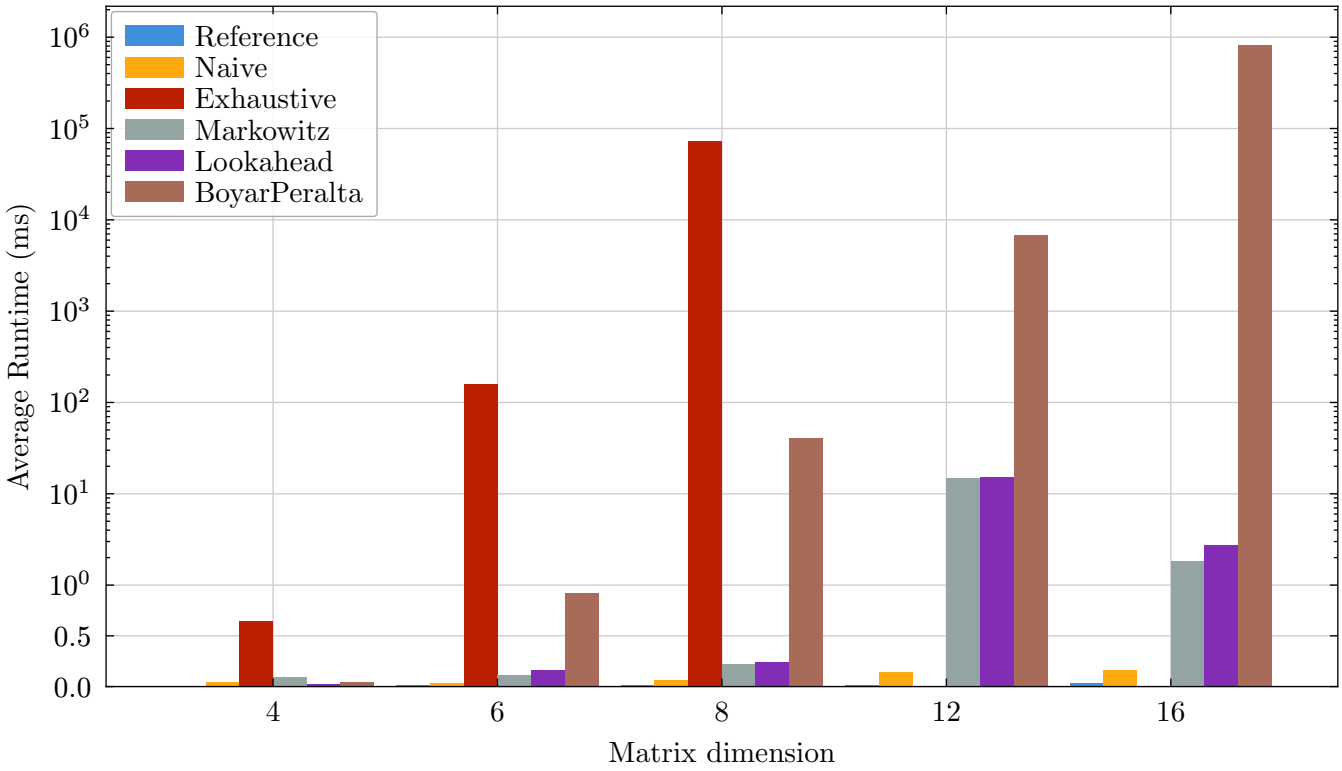

Figure 3: Average runtime for computing XOR-count of a matrix. Lower is better

Algorithm	Average XOR-count	Best XOR-count		
	Count	Count	Polynomial	Element
Reference	4.86	1	$x^4 + x + 1$	α
Naive	5.52	1	$x^4 + x + 1$	α
Exhaustive	3.43	1	$x^4 + x + 1$	α
Markowitz	4.21	1	$x^4 + x + 1$	α
Lookahead	3.71	1	$x^4 + x + 1$	α
BoyarPeralta	3.43	1	$x^4 + x + 1$	α

Table 1: XOR-counts for 4×4 matrices. α is a root of $x^4 + x + 1$.

Algorithm	Average XOR-count	Best XOR-count		
	Count	Count	Polynomial	Element
Reference	12.67	1	$x^6 + x^3 + 1$	α^{56}
Naive	13.75	2	$x^6 + x^3 + 1$	α^{56}
Exhaustive	8.54	1	$x^6 + x^3 + 1$	α^{56}
Markowitz	10.91	1	$x^6 + x^3 + 1$	α^{56}
Lookahead	9.29	1	$x^6 + x^3 + 1$	α^{56}
BoyarPeralta	7.7	1	$x^6 + x^3 + 1$	α^{56}

Table 2: XOR-counts for 6×6 matrices. α is a root of $x^6 + x^4 + x^3 + x + 1$.

Algorithm	Average XOR-count	Best XOR-count		
	Count	Count	Polynomial	Element
Reference	24.43	3	$x^8 + x^6 + x^3 + x^2 + 1$	α^{196}
Naive	25.2	9	$x^8 + x^7 + x^3 + x^2 + 1$	α^{74}
Exhaustive	16.3	3	$x^8 + x^6 + x^3 + x^2 + 1$	α^{196}
Markowitz	20.67	3	$x^8 + x^6 + x^3 + x^2 + 1$	α^{196}
Lookahead	17.85	3	$x^8 + x^6 + x^3 + x^2 + 1$	α^{196}
BoyarPeralta	13.62	3	$x^8 + x^6 + x^3 + x^2 + 1$	α^{196}

Table 3: XOR-counts for 8×8 matrices. α is a root of $x^8 + x^4 + x^3 + x^2 + 1$.

Algorithm	Average XOR-count	Best XOR-count		
	Count	Count	Polynomial	Element
Reference	60.67	31	$x^{12} + x^6 + x^4 + x + 1$	α^{420}
Naive	61.73	34	$x^{12} + x^{10} + x^6 + x^3 + x^2 + x + 1$	α^{3906}
Markowitz	50.59	23	$x^{12} + x^{10} + x^8 + x^7 + x^6 + x^4 + x^2 + x + 1$	α^{1914}
Lookahead	43.32	24	$x^{12} + x^{10} + x^8 + x^7 + x^6 + x^4 + x^2 + x + 1$	α^{1914}
BoyarPeralta	29.32	19	$x^{12} + x^{10} + x^8 + x^7 + x^6 + x^4 + x^2 + x + 1$	α^{1914}

Table 4: XOR-counts for 12×12 matrices. α is a root of $x^{12} + x^7 + x^6 + x^5 + x^3 + x + 1$.

Algorithm	Average XOR-count	Best XOR-count		
	Count	Count	Polynomial	Element
Reference	111.28	61	$x^{16} + x^{15} + x^{11} + x^7 + x^2 + x + 1$	α^{65145}
Naive	113.29	67	$x^{16} + x^{15} + x^{13} + x^{10} + x^9 + x^6 + x^5 + x^4 + 1$	α^{65531}
Markowitz	92.78	56	$x^{16} + x^{15} + x^{14} + x^{12} + x^9 + x^6 + x^5 + x + 1$	α^{9396}
Lookahead	79.8	49	$x^{16} + x^{15} + x^{14} + x^{12} + x^9 + x^6 + x^5 + x + 1$	α^{9396}
BoyarPeralta	50.11	36	$x^{16} + x^{15} + x^{14} + x^{12} + x^9 + x^6 + x^5 + x + 1$	α^{9396}

Table 5: XOR-counts for 16×16 matrices. α is a root of $x^{16} + x^5 + x^3 + x^2 + 1$.

The Naive Gaussian has a higher XOR-count on average and in the best case compared to Reference, so it is never worth using. Lookahead and Markowitz are quite similar. Lookahead has a slightly lower average XOR-count, but it is slightly slower. It is noticeable that Markowitz and Lookahead always have the same best XOR-count polynomial and element. In Table 1 we always get the same best Polynomial and best element, but we can already start seeing a different in average XOR-count. The same happens in Table 2 and Table 3, except for Naive. In Table 4 we see that Markowitz finds a better best XOR-count compared to Lookahead, so Lookahead does not always give better result. It does give better average XOR-count.

For $n = 4$, we can look at every element individually. See Table 6, Table 7 and Table 8.

Reference Repr.	Element	XOR-count					
		Reference	Naive	Exhaustive	Markowitz	Lookahead	BoyarPeralta
α	α	1	1	1	1	1	1
α^2	α^2	2	2	2	2	2	2
α^3	α^3	3	3	3	3	3	3
α^4	$\alpha + 1$	5	4	4	4	4	4
α^5	$\alpha^2 + \alpha$	5	6	5	5	6	5
α^6	$\alpha^3 + \alpha^2$	5	8	4	5	4	4
α^7	$\alpha^3 + \alpha + 1$	6	9	4	5	4	4
α^8	$\alpha^2 + 1$	6	5	4	5	4	4
α^9	$\alpha^3 + \alpha$	8	6	4	6	6	4
α^{10}	$\alpha^2 + \alpha + 1$	9	7	5	7	7	5
α^{11}	$\alpha^3 + \alpha^2 + \alpha$	8	8	4	6	6	4
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$	6	6	3	6	3	3
α^{13}	$\alpha^3 + \alpha^2 + 1$	3	4	2	3	2	2
α^{14}	$\alpha^3 + 1$	1	2	1	1	1	1

Table 6: XOR-counts for elements with defining polynomial $x^4 + x + 1$ with root α

Reference Repr.	Element	XOR-count					
		Reference	Naive	Exhaustive	Markowitz	Lookahead	BoyarPeralta
α	$\beta^3 + \beta$	6	5	4	5	4	4
α^2	$\beta^2 + \beta$	6	4	4	5	4	4
α^3	β	3	3	3	3	3	3
α^4	$\beta^3 + \beta + 1$	6	5	4	5	4	4
α^5	$\beta^3 + \beta^2$	6	7	4	5	4	4
α^6	β^2	3	6	3	3	3	3
α^7	$\beta^3 + 1$	5	5	4	4	4	4
α^8	$\beta^2 + \beta + 1$	6	6	4	5	4	4
α^9	β^3	3	9	3	3	3	3
α^{10}	$\beta^3 + \beta^2 + 1$	6	7	4	5	4	4
α^{11}	$\beta^3 + \beta^2 + \beta$	5	8	4	4	4	4
α^{12}	$\beta^3 + \beta^2 + \beta + 1$	3	12	3	3	3	3
α^{13}	$\beta^2 + 1$	5	6	4	4	4	4
α^{14}	$\beta + 1$	5	5	4	4	4	4

Table 7: XOR-counts for elements with defining polynomial $x^4 + x^3 + x^2 + x + 1$ with root β

Reference Repr.	Element	XOR-count					
		Reference	Naive	Exhaustive	Markowitz	Lookahead	BoyarPeralta
α	$\gamma^2 + \gamma$	2	4	2	2	2	2
α^2	$\gamma^3 + \gamma^2 + 1$	5	10	4	5	4	4
α^3	$\gamma^2 + 1$	5	5	4	5	4	4
α^4	$\gamma^2 + \gamma + 1$	6	9	4	5	4	4
α^5	$\gamma^3 + \gamma + 1$	9	6	5	7	6	5
α^6	γ^3	6	3	3	6	3	3
α^7	γ	1	1	1	1	1	1
α^8	$\gamma^3 + \gamma^2$	1	2	1	1	1	1
α^9	$\gamma + 1$	3	6	3	3	3	3
α^{10}	$\gamma^3 + \gamma$	5	7	5	5	6	5
α^{11}	$\gamma^3 + \gamma^2 + \gamma$	6	5	4	5	4	4
α^{12}	$\gamma^3 + \gamma^2 + \gamma + 1$	8	9	4	6	6	4
α^{13}	$\gamma^3 + 1$	8	4	4	6	5	4
α^{14}	γ^2	3	2	2	3	2	2

Table 8: XOR-counts for elements with defining polynomial $x^4 + x^3 + 1$ with root γ

Reference Repr. is the element written as a power of the generator α, which is the root of $x^4 + x + 1$. This means that row k in Table 6 is the same element as row k in Table 7 and Table 8. What can be seen is that one polynomial is not always the best for every element. The defining polynomial $x^4 + x^3 + x^2 + x + 1$ in Table 7 seems to have bad XOR-counts. Every element in Table 7 has an XOR-count of at least 3, while Table 6 and Table 8 have elements with an XOR-count of 1. The worst case XOR-count in Table 7 does perform better than in the other two tables.

The most notable inputs are α, β and γ: the root of the defining polynomial. They have the lowest XOR-count for every defining polynomial. The root of a defining polynomial seems to have a lower XOR-count than other elements. We know that the root is a basis element and we observe that all basis elements seem to have lower XOR-count on average compared to other elements. This is especially noticeable in Table 7, where three of the four elements with an XOR-count of 3 are basis elements. In the other two tables, we do see that higher powers of the root perform worse than lower powers. Below are a few interesting examples. Especially Example 13 is interesting, because it is an exception to this trend. Also note how XOR-count for α is way lower than the average XOR-count in Table 3 and Table 4 for Boyar-Peralta.

Example 9. We will take a look at how well the multiplication by basis elements of the defining polynomial $x^8 + x^7 + x^6 + x + 1$ with a root α perform with respect to XOR-count. Only Boyar-Peralta is considered, since it consistently has the lowest XOR-count. The reused subexpressions will be shown in brackets.

We now consider multiplication by α, where we see 3 XOR gates required by the original matrix, and no subexpressions found, hence no reduction in XOR-count.

$$\begin{pmatrix} 0&0&0&0&0&0&0&1\\ 1&0&0&0&0&0&0&1\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&1\\ 0&0&0&0&0&0&1&1 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_8\\ x_1+x_8\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6+x_8\\ x_7+x_8 \end{pmatrix}$$

Now consider multiplication by α^2. Here we find a reusable subexpression $x_7 + x_8$ shown in brackets in row 7 and first seen in row 1. The original matrix required 5 XOR gates, but by reusing this subexpression we get 4 XOR gates.

$$\begin{pmatrix} 0&0&0&0&0&0&1&1\\ 0&0&0&0&0&0&1&0\\ 1&0&0&0&0&0&0&1\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&1&1\\ 0&0&0&0&0&1&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_7+x_8\\ x_7\\ x_1+x_8\\ x_2\\ x_3\\ x_4\\ x_5+(x_7+x_8)\\ x_6+x_7 \end{pmatrix}$$

Now consider multiplication by α^3. The required XOR gates goes from 7 to 5.

$$\begin{pmatrix} 0&0&0&0&0&1&1&0\\ 0&0&0&0&0&1&0&1\\ 0&0&0&0&0&0&1&0\\ 1&0&0&0&0&0&0&1\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&1&1&0\\ 0&0&0&0&1&1&0&1 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_6+x_7\\ x_6+x_8\\ x_7\\ x_1+x_8\\ x_2\\ x_3\\ x_4+(x_6+x_7)\\ x_5+(x_6+x_8) \end{pmatrix}$$

Now consider multiplication by α^4. The matrix is getting a lot more complex, but that also allows for more subexpressions. The required XOR gates goes from 12 to 7.

$$\begin{pmatrix} 0&0&0&0&1&1&0&1\\ 0&0&0&0&1&0&1&1\\ 0&0&0&0&0&1&0&1\\ 0&0&0&0&0&0&1&0\\ 1&0&0&0&0&0&0&1\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&1&1&0&1\\ 0&0&0&1&1&0&1&1 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_5+(x_6+x_8)\\ x_8+(x_5+x_7)\\ x_6+x_8\\ x_7\\ x_1+x_8\\ x_2\\ x_3+(x_5+(x_6+x_8))\\ x_4+(x_8+(x_5+x_7)) \end{pmatrix}$$

Now consider multiplication by α^5. The required XOR gates goes from 16 to 9.

$$\begin{pmatrix} 0&0&0&1&1&0&1&1\\ 0&0&0&1&0&1&1&0\\ 0&0&0&0&1&0&1&1\\ 0&0&0&0&0&1&0&1\\ 0&0&0&0&0&0&1&0\\ 1&0&0&0&0&0&0&1\\ 0&1&0&1&1&0&1&1\\ 0&0&1&1&0&1&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8 \end{pmatrix} = \begin{pmatrix} (x_4+x_7)+(x_5+x_8)\\ x_6+(x_4+x_7)\\ x_7+(x_5+x_8)\\ x_6+x_8\\ x_7\\ x_1+x_8\\ x_2+((x_4+x_7)+(x_5+x_8))\\ x_3+(x_6+(x_4+x_7)) \end{pmatrix}$$

Now consider multiplication by α^6. This matrix is again a step up in complexity, so we will analyze the result. The original matrix required 20 XOR gates. We go row by row:

1. This is the first row, so every subexpression is new giving us 3 XOR gates.
2. Again every subexpression in brackets is new, so we need 3 XOR gates again.
3. This row is more interesting, because $x_6 + (x_4 + x_7)$ is already computed for row 1, so now we need 0 XOR gates,
4. This is the most complicated row, but we can eliminate $x_3 + (x_6 + (x_4 + x_7))$, since that is exactly row 1. Additionally we see that $x_5 + (x_3 + (x_6 + x_8))$ is exactly row 2, which leaves us with 2 leftover additions. This means this row only requires 2 XOR gates.
5. $x_6 + x_8$ is already computed in row 2, so this required 0 XOR gates.
6. This is simply the input x_7, so it of course required 0 XOR gates.
7. Again we can eliminate $x_3 + (x_6 + (x_4 + x_7))$, because it is row 1. This leaves us with 2 additions, so 2 XOR gates are required.
8. Here we find row 2: $x_5 + (x_3 + (x_6 + x_8))$, which leaves one addition, so 1 XOR gate is required.

Adding this all up gives $3 + 3 + 0 + 2 + 0 + 0 + 2 + 1 = 11$, so 11 XOR gates are required.

$$\begin{pmatrix} 0&0&1&1&0&1&1&0\\ 0&0&1&0&1&1&0&1\\ 0&0&0&1&0&1&1&0\\ 0&0&0&0&1&0&1&1\\ 0&0&0&0&0&1&0&1\\ 0&0&0&0&0&0&1&0\\ 1&0&1&1&0&1&1&1\\ 0&1&1&0&1&1&0&1 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8 \end{pmatrix} = \begin{pmatrix} x_3+(x_6+(x_4+x_7))\\ x_5+(x_3+(x_6+x_8))\\ x_6+(x_4+x_7)\\ (x_3+(x_6+(x_4+x_7)))+(x_4+(x_5+(x_3+(x_6+x_8))))\\ x_6+x_8\\ x_7\\ (x_3+(x_6+(x_4+x_7)))+(x_1+x_8)\\ x_2+(x_5+(x_3+(x_6+x_8))) \end{pmatrix}$$

Now consider multiplication by α^7. The required XOR gates goes from 25 to 13. This can be checked in the same way as before, but it takes a little longer.

$$\begin{pmatrix} 0&1&1&0&1&1&0&1\\ 0&1&0&1&1&0&1&1\\ 0&0&1&0&1&1&0&1\\ 0&0&0&1&0&1&1&0\\ 0&0&0&0&1&0&1&1\\ 0&0&0&0&0&1&0&1\\ 0&1&1&0&1&1&1&1\\ 1&1&0&1&1&0&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8 \end{pmatrix} = \begin{pmatrix} x_2+(x_5+(x_3+(x_6+x_8)))\\ (x_8+(x_5+x_7))+(x_2+x_4)\\ x_5+(x_3+(x_6+x_8))\\ ((x_8+(x_5+x_7))+(x_2+x_4))+(x_3+(x_2+(x_5+(x_3+(x_6+x_8)))))\\ x_8+(x_5+x_7)\\ x_6+x_8\\ x_7+(x_2+(x_5+(x_3+(x_6+x_8))))\\ ((x_8+(x_5+x_7))+(x_2+x_4))+(x_1+x_8) \end{pmatrix}$$

The matrix becomes increasingly chaotic, which can also be seen in the XOR-count. Increasing the power increases the XOR-count. See Table 9 for an overview of all the XOR-counts before and after applying Boyar-Peralta.

	α	α^2	α^3	α^4	α^5	α^6	α^7
XOR-count before Boyar-Peralta	3	5	7	12	16	20	25
XOR-count after Boyar-Peralta	3	4	5	7	9	11	13

Table 9: Overview of XOR-counts for all basis elements before and after Boyar-Peralta

Example 10. We will take a look at how well the multiplication by basis elements of the defining polynomial $x^8 + x^4 + x^3 + x + 1$ with a root α perform with respect to XOR-count. Only Boyar-Peralta is considered, since it consistently has the lowest XOR-count. The reused subexpressions will be shown in brackets.

Only interesting elements are shown. The other matrices will be in the appendix.

We now consider multiplication by α, where we see 3 XOR gates required by the original matrix, and no subexpressions found, hence no reduction in XOR-count. The same as in Example 9.

$$\begin{pmatrix} 0&0&0&0&0&0&0&1\\ 1&0&0&0&0&0&0&1\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&1\\ 0&0&0&1&0&0&0&1\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_8\\ x_1 + x_8\\ x_2\\ x_3 + x_8\\ x_4 + x_8\\ x_5\\ x_6\\ x_7 \end{pmatrix}$$

Now consider multiplication by α^7. The required XOR gates goes from 22 to 14. This defining polynomial follows the trend where higher powers of α have higher XOR-count.

$$\begin{pmatrix} 0&1&0&0&0&1&1&0\\ 0&1&1&0&0&1&0&1\\ 0&0&1&1&0&0&1&0\\ 0&1&0&1&1&1&1&1\\ 0&1&1&0&1&0&0&1\\ 0&0&1&1&0&1&0&0\\ 0&0&0&1&1&0&1&0\\ 1&0&0&0&1&1&0&1 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_7 + (x_2 + x_6)\\ x_3 + (x_8 + (x_2 + x_6))\\ x_3 + (x_4 + x_7)\\ (x_5 + (x_4 + x_7)) + (x_8 + (x_2 + x_6))\\ ((x_5 + (x_4 + x_7)) + (x_8 + (x_2 + x_6))) + (x_6 + (x_3 + (x_4 + x_7)))\\ x_7 + (x_6 + (x_3 + (x_4 + x_7)))\\ x_5 + (x_4 + x_7)\\ (x_1 + x_2) + (x_5 + (x_8 + (x_2 + x_6))) \end{pmatrix}$$

The matrix for multiplication by α is very similar to Example 9 and the XOR-count behaves very similar as well. See Table 10 for an overview of all the XOR-counts before and after applying Boyar-Peralta.

	α	α^2	α^3	α^4	α^5	α^6	α^7
XOR-count before Boyar-Peralta	3	6	9	12	16	19	22
XOR-count after Boyar-Peralta	3	5	7	9	11	13	14

Table 10: Overview of XOR-counts for all basis elements before and after Boyar-Peralta

Example 11. We will take a look at how well the multiplication by basis elements of the defining polynomial $x^8 + x^7 + x^6 + x^5 + x^4 + x + 1$ with a root α perform with respect to XOR-count. Only Boyar-Peralta is considered, since it consistently has the lowest XOR-count. The reused subexpressions will be shown in brackets.

Only interesting elements are shown. The other matrices will be in the appendix.

We now consider multiplication by α, where we see 5 XOR gates required by the original matrix, and no subexpressions found, hence no reduction in XOR-count.

$$\begin{pmatrix} 0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&1 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&1 \\ 0&0&0&0&1&0&0&1 \\ 0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&1&1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} = \begin{pmatrix} x_8 \\ x_1 + x_8 \\ x_2 \\ x_3 \\ x_4 + x_8 \\ x_5 + x_8 \\ x_6 + x_8 \\ x_7 + x_8 \end{pmatrix}$$

Now consider multiplication by α^7. The required XOR gates goes from 20 to 12.

$$\begin{pmatrix} 0&1&1&0&0&0&1&1 \\ 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 0&0&0&1&0&1&0&0 \\ 0&1&1&0&1&0&0&1 \\ 0&1&0&1&0&1&1&1 \\ 0&1&0&0&1&0&0&0 \\ 1&1&0&0&0&1&1&1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} = \begin{pmatrix} (x_2 + x_7) + (x_3 + x_8) \\ x_4 + (x_2 + x_7) \\ x_5 + (x_3 + x_8) \\ x_4 + x_6 \\ x_2 + (x_5 + (x_3 + x_8)) \\ (x_4 + x_6) + (x_8 + (x_2 + x_7)) \\ x_2 + x_5 \\ ((x_4 + x_6) + (x_8 + (x_2 + x_7))) + (x_1 + x_4) \end{pmatrix}$$

As before, increasing the power of α increases the XOR-count, but it is notable that multiplication by α^5 and α^6 results in the same XOR-count. See Table 11 for an overview of all the XOR-counts before and after applying Boyar-Peralta.

	α	α^2	α^3	α^4	α^5	α^6	α^7
XOR-count before Boyar-Peralta	5	7	9	11	13	16	20
XOR-count after Boyar-Peralta	5	6	7	9	11	11	12

Table 11: Overview of XOR-counts for all basis elements before and after Boyar-Peralta

Example 12. We will take a look at how well the multiplication by basis elements of the defining polynomial $x^{12} + x^3 + 1$ with a root α perform with respect to XOR-count. Only Boyar-Peralta

is considered, since it consistently has the lowest XOR-count. The reused subexpressions will be shown in brackets.

Only interesting elements are shown. The other matrices will be in the appendix.

We now consider multiplication by α, where we see 1 XOR gates required by the original matrix, which cannot have subexpressions.

$$\begin{pmatrix} 0&0&0&0&0&0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_{12} \\ x_1 \\ x_2 \\ x_3 + x_{12} \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \end{pmatrix}$$

Now consider multiplication by α^{10}. The required XOR gates goes from 11 to 10. This is a very small improvement compared to other matrices.

$$\begin{pmatrix} 0&0&1&0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&1&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&1 \\ 1&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_3 + x_{12} \\ x_4 \\ x_5 \\ x_6 + (x_3 + x_{12}) \\ x_4 + x_7 \\ x_5 + x_8 \\ x_6 + x_9 \\ x_7 + x_{10} \\ x_8 + x_{11} \\ x_9 + x_{12} \\ x_1 + x_{10} \\ x_2 + x_{11} \end{pmatrix}$$

Now consider multiplication by α^{11}. The required XOR gates goes from 13 to 11. Again not a very big improvement.

$$\begin{pmatrix} 0&1&0&0&0&0&0&0&0&0&1&0 \\ 0&0&1&0&0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&1&0&0&1&0&0&0&0&0&1&0 \\ 0&0&1&0&0&1&0&0&0&0&0&1 \\ 0&0&0&1&0&0&1&0&0&0&0&0 \\ 0&0&0&0&1&0&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0&0&1&0 \\ 0&0&0&0&0&0&0&0&1&0&0&1 \\ 1&0&0&0&0&0&0&0&0&1&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_2 + x_{11} \\ x_3 + x_{12} \\ x_4 \\ x_5 + (x_2 + x_{11}) \\ x_6 + (x_3 + x_{12}) \\ x_4 + x_7 \\ x_5 + x_8 \\ x_6 + x_9 \\ x_7 + x_{10} \\ x_8 + x_{11} \\ x_9 + x_{12} \\ x_1 + x_{10} \end{pmatrix}$$

This example is interesting because of the consistent pattern we find in the matrix. We see this consistency reflected in the XOR-count which is equal to the exponent above the α after Boyar-Peralta. See Table 12 for an overview of all the XOR-counts before and after applying Boyar-Peralta.

	α	α^2	α^3	α^4	α^5	α^6	α^7	α^8	α^9	α^{10}	α^{11}
XOR-count before Boyar-Peralta	1	2	3	4	5	6	7	8	9	11	13
XOR-count after Boyar-Peralta	1	2	3	4	5	6	7	8	9	10	11

Table 12: Overview of XOR-counts for all basis elements before and after Boyar-Peralta

Example 13. We will take a look at how well the multiplication by basis elements of the defining polynomial $x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$ with a root α perform with respect to XOR-count. Only Boyar-Peralta is considered, since it consistently has the lowest XOR-count. The reused subexpressions will be shown in brackets.

Only interesting elements are shown. The other matrices will be in the appendix.

We now consider multiplication by α, where we see 11 XOR gates required by the original matrix, and no subexpressions found, hence no reduction in XOR-count. This XOR-count is high compared to Example 12.

$$\begin{pmatrix} 0&0&0&0&0&0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0&0&0&0&0&1 \\ 0&1&0&0&0&0&0&0&0&0&0&0&1 \\ 0&0&1&0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&1&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&1&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&1&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0&1&0&0&0&1 \\ 0&0&0&0&0&0&0&0&0&1&0&0&1 \\ 0&0&0&0&0&0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&0&0&0&0&0&1&1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_{12} \\ x_1 + x_{12} \\ x_2 + x_{12} \\ x_3 + x_{12} \\ x_4 + x_{12} \\ x_5 + x_{12} \\ x_6 + x_{12} \\ x_7 + x_{12} \\ x_8 + x_{12} \\ x_9 + x_{12} \\ x_{10} + x_{12} \\ x_{11} + x_{12} \end{pmatrix}$$

Now consider multiplication by α^{11}. The required XOR gates remains 11 and again no subexpressions found.

$$\begin{pmatrix} 0&1&1&0&0&0&0&0&0&0&0&0 \\ 0&1&0&1&0&0&0&0&0&0&0&0 \\ 0&1&0&0&1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&1&0&0&0&0&0 \\ 0&1&0&0&0&0&0&1&0&0&0&0 \\ 0&1&0&0&0&0&0&0&1&0&0&0 \\ 0&1&0&0&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&1&0 \\ 0&1&0&0&0&0&0&0&0&0&0&1 \\ 0&1&0&0&0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_2 + x_3 \\ x_2 + x_4 \\ x_2 + x_5 \\ x_2 + x_6 \\ x_2 + x_7 \\ x_2 + x_8 \\ x_2 + x_9 \\ x_2 + x_{10} \\ x_2 + x_{11} \\ x_2 + x_{12} \\ x_2 \\ x_1 + x_2 \end{pmatrix}$$

Like in Example 12, we find a very noticeable pattern in the matrix. The result is an inefficient matrix where the XOR-count found with Boyar-Peralta is equal to the d-XOR-count. It has this behavior for every power of α, so the XOR-count is the same for all powers of α. This makes it an exception to the trend that higher powers of the root of a defining polynomial have higher XOR-count. See Table 13 for an overview of all the XOR-counts before and after applying Boyar-Peralta.

	α	α^2	α^3	α^4	α^5	α^6	α^7	α^8	α^9	α^{10}	α^{11}
XOR-count before Boyar-Peralta	11	11	11	11	11	11	11	11	11	11	11
XOR-count after Boyar-Peralta	11	11	11	11	11	11	11	11	11	11	11

Table 13: Overview of XOR-counts for all basis elements before and after Boyar-Peralta

3.2 Multiplication with two variable inputs

When both inputs are unknown, we first turn one input into a matrix, which we then use for matrix multiplication with the second variable.

Example 14. Consider the extension $\mathbb{F}_{2^4}$ constructed using the irreducible polynomial f given by $f(x) = x^4 + x^3 + x^2 + x + 1$ with a root α. Since neither input is fixed, we construct the multiplication matrix for a general field elemnt

$$A = a_1 + a_2\alpha + a_3\alpha^2 + a_4\alpha^3, \text{ where } a_i \in \mathbb{F}_2 \tag{40}$$

Now we compute the matrix in the same way as before

$$
\begin{aligned}
A &= a_1 + a_2\alpha + a_3\alpha^2 + a_4\alpha^3 \\
A\alpha &= a_1\alpha + a_2\alpha^2 + a_3\alpha^3 + a_4\alpha^4 \\
&= a_1\alpha + a_2\alpha^2 + a_3\alpha^3 + a_4(1 + \alpha + \alpha^2 + \alpha^3) \\
&= a_4 + (a_1 + a_4)\alpha + (a_2 + a_4)\alpha^2 + (a_3 + a_4)\alpha^3 \\
A\alpha^2 &= a_4\alpha + (a_1 + a_4)\alpha^2 + (a_2 + a_4)\alpha^3 + (a_3 + a_4)\alpha^4 \\
&= a_4\alpha + (a_1 + a_4)\alpha^2 + (a_2 + a_4)\alpha^3 + (a_3 + a_4)(1 + \alpha + \alpha^2 + \alpha^3) \\
&= (a_3 + a_4) + a_3\alpha + (a_1 + a_3)\alpha^2 + (a_2 + a_3)\alpha^3 \\
A\alpha^3 &= (a_3 + a_4)\alpha + a_3\alpha^2 + (a_1 + a_3)\alpha^3 + (a_2 + a_3)\alpha^4 \\
&= (a_3 + a_4)\alpha + a_3\alpha^2 + (a_1 + a_3)\alpha^3 + (a_2 + a_3)(1 + \alpha + \alpha^2 + \alpha^3) \\
&= (a_2 + a_3) + (a_2 + a_4)\alpha + a_2\alpha^2 + (a_1 + a_2)\alpha^3,
\end{aligned}
\tag{41}
$$

so we get the matrix

$$
U_1 = \begin{pmatrix}
a_1 & a_4 & a_3 + a_4 & a_2 + a_3 \\
a_2 & a_1 + a_4 & a_3 & a_2 + a_4 \\
a_3 & a_2 + a_4 & a_1 + a_3 & a_2 \\
a_4 & a_3 + a_4 & a_2 + a_3 & a_1 + a_2
\end{pmatrix}, \tag{42}
$$

which we will call an expression matrix. Of course the matrix depends on the basis, which depends on the polynomial that defined the field. This is the expression matrix defined by $f(x) = x^4 + x + 1$:

$$
U_2 = \begin{pmatrix}
a_1 & a_4 & a_3 & a_2 \\
a_2 & a_1 + a_4 & a_3 + a_4 & a_2 + a_3 \\
a_3 & a_2 & a_1 + a_4 & a_3 + a_4 \\
a_4 & a_3 & a_2 & a_1 + a_4
\end{pmatrix}. \tag{43}
$$

3.2.1 Optimizing matrix creation

Notice that every element inside the matrix U is a sum using the input variables a_1, a_2, a_3 and a_4, so what we want to find are the shared subexpressions of these elements. What we are really looking at is vectorizing an expression matrix of dimension $n \times n$ by stacking its columns into a vector of length n^2 and then optimizing the coefficient matrix which will be of size $n^2 \times n$.

Example 15. Using the expression matrix U_2 from Example 14 vectorization would result in

$$y = \begin{pmatrix} \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_4 \\ a_1 + a_4 \\ a_2 \\ a_3 \\ a_3 \\ a_3 + a_4 \\ a_1 + a_4 \\ a_2 \\ a_2 \\ a_2 + a_3 \\ a_3 + a_4 \\ a_1 + a_4 \end{pmatrix}. \tag{44}$$

This is the result of the following matrix-vector multiplication

$$y = Ma = \begin{pmatrix} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \tag{45}$$

Our goal is to optimize the coefficient matrix M as we did in the previous section.

The aforementioned procedure does not produce square matrix, so we cannot use the Gaussian-based optimization algorithms. This is not a big loss, however, since we saw that Boyar-Peralta always produces better XOR-counts. This complete procedure for optimizing the creation of the expression matrix is shown in Algorithm 12.

Expressions consisting of a single term are ignored, since they are already solved. In Algorithm 12 we ensure *Search* does not contain duplicates to avoid biasing Boyar-Peralta towards duplicate target expressions. This is because Boyar-Peralta selects intermediate subexpressions based on the reduction of the *Dist* vector. If the same target expression appears multiple times, each duplicate contributes to the

reduct of *Dist*, making intermediate subexpressions that benefit the duplicated target appears better. However, the duplicated rows are the same, so they will use the exact same final expression, which means the heuristic overestimates how useful the intermediate subexpression for duplicated targets is.

Algorithm 12: Matrix computation

Input : A size n and an $n \times n$ expression matrix A
Output: An array *base* with all intermediate results required to compute every element.

```
1: Search ← []
2: for Row in A do
3:     for El in Row do
4:         if termCount(El) > 1 and El not in Search then
5:             Search.push(El)
6:         fi
7:     end
8: end
9: return mainBoyarPeralta(search) # see Algorithm 8
```

If we run Algorithm 12 on U_1 from Example 14 we get 18 additions, but on U_2 we get 15 additions. As expected, choosing the right polynomial is again very important.

3.2.2 Benchmark

Similar to the previous benchmark we will generate all or at most 200 $n \times n$ expression matrices for $n = 4, 6, 8, 12, 16$. Reference will be the d-XOR-count of the matrix in Equation 45, where we write out the matrix multiplication and count the number of additions.

As expected, Boyar-Peralta improves the XOR-count by a lot at the cost of runtime. Comparing Figure 4 and Figure 3 we can see that Boyar-Peralta runs faster for expression matrices than for fixed matrices with equal dimensions. Furthermore, the best XOR-count polynomial in Table 2 and in Table 14 for 6 dimensions are the same .

In Table 14 we also notice that the relative improvement of Boyar-Peralta over Reference increases with dimension. For 4 dimension the reduction is around 20%, while for 16 dimensions Boyar-Peralta reduces the average XOR-count by around 68%. This makes sense, since for n dimensions there are n different inputs and the expression matrix has n^2 entries. It follows that for large n the number of common subexpressions is expected to increase.

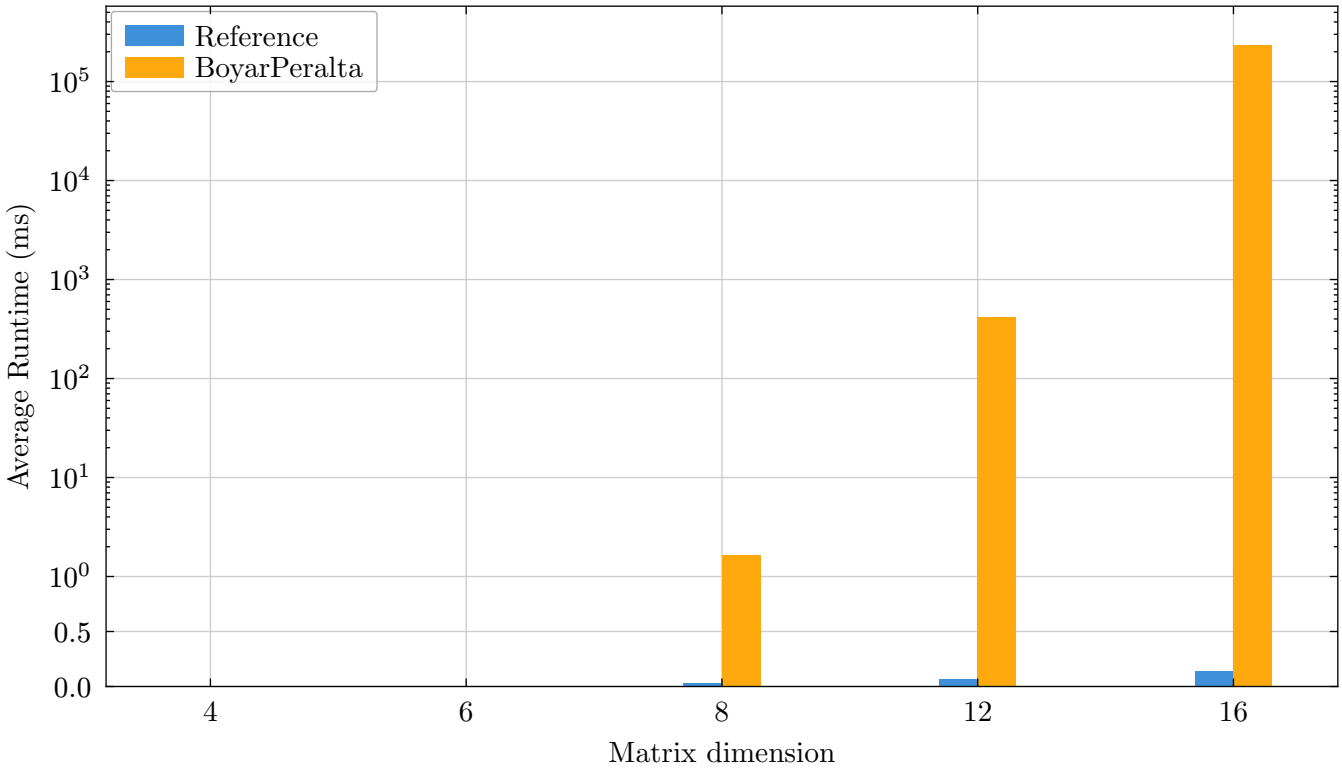

Figure 4: Average runtime for computing XOR-count of an expression matrix. Lower is better

Dimension	Matrix count	Average XOR-count		Best XOR-count	
		Reference	BoyarPeralta	Count	Polynomial
4	3	20.33	16	15	$x^4 + x^3 + 1$
6	9	64.22	40.11	33	$x^6 + x^3 + 1$
8	30	152.4	79.83	71	$x^8 + x^7 + x^6 + x + 1$
12	200	474.41	187.76	143	$x^{12} + x^5 + 1$
16	200	1092.03	346.41	285	$x^{16} + x^{13} + x^{12} + x^2 + 1$

Table 14: XOR-count matrices. Lower is better

Example 16. We will take a loot at how well the expression matrix of the defining polynomial $x^4 + x^3 + x^2 + x + 1$ does after applying Algorithm 12. The original matrix required 21 XOR-gates, but after the algorithm it only takes 18 XOR gates. This is because $a_3 + a_4, a_2 + a_4$ and $a_2 + a_3$ are used twice.

$$\begin{pmatrix} a_1 & a_4 & a_3 + a_4 & a_2 + a_3 \\ a_2 & a_1 + a_4 & a_3 & a_2 + a_4 \\ a_3 & a_2 + a_4 & a_1 + a_3 & a_2 \\ a_4 & a_3 + a_4 & a_2 + a_3 & a_1 + a_2 \end{pmatrix}$$

Using $x^4 + x^3 + 1$ as a defining polynomial yields a different matrix requiring 22 XOR-gates, which is worse than the previous one. After applying Algorithm 12, however we only require 15 XOR

gates. We see that $a_3 + a_4$ gets used 6 times and $a_2 + (a_3 + a_4)$ gets used 3 times. This reduces the number of required XOR gates by 7.

$$\begin{pmatrix} a_1 & a_4 & a_3 + a_4 & a_2 + a_3 + a_4 \\ a_2 & a_1 & a_4 & a_3 + a_4 \\ a_3 & a_2 & a_1 & a_4 \\ a_4 & a_3 + a_4 & a_2 + a_3 + a_4 & a_1 + a_2 + a_3 + a_4 \end{pmatrix} = \begin{pmatrix} a_1 & a_4 & a_3 + a_4 & a_2 + (a_3 + a_4) \\ a_2 & a_1 & a_4 & a_3 + a_4 \\ a_3 & a_2 & a_1 & a_4 \\ a_4 & a_3 + a_4 & a_2 + (a_3 + a_4) & a_1 + (a_2 + (a_3 + a_4)) \end{pmatrix}$$

See Table 15 for an overview of the XOR-count.

	$x^4 + x^3 + x^2 + x + 1$	$x^4 + x^3 + 1$
XOR-count before Boyar-Peralta	21	22
XOR-count after Boyar-Peralta	18	15

Table 15: Overview of XOR-counts for the example defining polynomial of degree 4

Example 17. We will take a loot at how well the expression matrix of the defining polynomial $x^6 + x^5 + 1$ does after applying Algorithm 12. The original matrix required 65 XOR gates, but after the algorithm it only takes 35 XOR gates. This is a big reduction, which we can see in the matrix. It has a pattern with a lot of common subexpressions. Especially $a_5 + a_6$ is extremely common. It has 15 occurrences.

$$\begin{pmatrix} a_1 & a_6 & a_5 + a_6 & a_4 + a_5 + a_6 & a_3 + a_4 + a_5 + a_6 & a_2 + a_3 + a_4 + a_5 + a_6 \\ a_2 & a_1 & a_6 & a_5 + a_6 & a_4 + a_5 + a_6 & a_3 + a_4 + a_5 + a_6 \\ a_3 & a_2 & a_1 & a_6 & a_5 + a_6 & a_4 + a_5 + a_6 \\ a_4 & a_3 & a_2 & a_1 & a_6 & a_5 + a_6 \\ a_5 & a_4 & a_3 & a_2 & a_1 & a_6 \\ a_6 & a_5 + a_6 & a_4 + a_5 + a_6 & a_3 + a_4 + a_5 + a_6 & a_2 + a_3 + a_4 + a_5 + a_6 & a_1 + a_2 + a_3 + a_4 + a_5 + a_6 \end{pmatrix} =$$

$$\begin{pmatrix} a_1 & a_6 & a_5 + a_6 & a_4 + (a_5 + a_6) & a_3 + (a_4 + (a_5 + a_6)) & a_2 + (a_3 + (a_4 + (a_5 + a_6))) \\ a_2 & a_1 & a_6 & a_5 + a_6 & a_4 + (a_5 + a_6) & a_3 + (a_4 + (a_5 + a_6)) \\ a_3 & a_2 & a_1 & a_6 & a_5 + a_6 & a_4 + (a_5 + a_6) \\ a_4 & a_3 & a_2 & a_1 & a_6 & a_5 + a_6 \\ a_5 & a_4 & a_3 & a_2 & a_1 & a_6 \\ a_6 & a_5 + a_6 & a_4 + (a_5 + a_6) & a_3 + (a_4 + (a_5 + a_6)) & a_2 + (a_3 + (a_4 + (a_5 + a_6))) & a_1 + (a_2 + (a_3 + (a_4 + (a_5 + a_6)))) \end{pmatrix}$$

Using $x^6 + x^4 + x^2 + x + 1$ as a defining polynomial yields a different matrix requiring 69 XOR-gates. This is slightly worse than the previous one. There are a few subexpressions we can reuse like $a_5 + a_6$, $a_3 + a_5$ and $a_4 + a_6$. Although is has a lot of subexpressions to reuse, it does not reuse them very often. It requires 45 XOR gates after Boyar-Peralta. See Table 16 for an overview of the XOR-count.

$$\begin{pmatrix} a_1 & a_6 & a_5 & a_4+a_6 & a_3+a_5 & a_2+a_4 \\ a_2 & a_1+a_6 & a_5+a_6 & a_4+a_5+a_6 & a_3+a_4+a_5+a_6 & a_2+a_3+a_4+a_5 \\ a_3 & a_2+a_6 & a_1+a_5+a_6 & a_4+a_5 & a_3+a_4+a_6 & a_2+a_3+a_5+a_6 \\ a_4 & a_3 & a_2+a_6 & a_1+a_5+a_6 & a_4+a_5 & a_3+a_4+a_6 \\ a_5 & a_4+a_6 & a_3+a_5 & a_2+a_4 & a_1+a_3+a_6 & a_2+a_5 \\ a_6 & a_5 & a_4+a_6 & a_3+a_5 & a_2+a_4 & a_1+a_3+a_6 \end{pmatrix} =$$

$$\begin{pmatrix} a_1 & a_6 & a_5 & a_4+a_6 & a_3+a_5 & a_2+a_4 \\ a_2 & a_1+a_6 & a_5+a_6 & a_4+(a_5+a_6) & a_3+(a_4+(a_5+a_6)) & (a_3+a_5)+(a_2+a_4) \\ a_3 & a_2+a_6 & a_1+(a_5+a_6) & a_4+a_5 & a_3+(a_4+a_6) & (a_3+a_5)+(a_2+a_6) \\ a_4 & a_3 & a_2+a_6 & a_1+(a_5+a_6) & a_4+a_5 & a_3+(a_4+a_6) \\ a_5 & a_4+a_6 & a_3+a_5 & a_2+a_4 & a_3+(a_1+a_6) & a_2+a_5 \\ a_6 & a_5 & a_4+a_6 & a_3+a_5 & a_2+a_4 & a_3+(a_1+a_6) \end{pmatrix}$$

	x^6+x^5+1	$x^6+x^4+x^2+x+1$
XOR-count before Boyar-Peralta	65	69
XOR-count after Boyar-Peralta	35	45

Table 16: Overview of XOR-counts for the example defining polynomial of degree 6

Example 18. These examples dont fit on a page, so we just give an overview in Table 17. We can clearly see a big reduction in the required XOR gates even for very different polynomials.

	$x^8+x^7+x^3+x+1$	$x^8+x^5+x^4+x^3+x^2+x+1$
XOR-count before Boyar-Peralta	153	183
XOR-count after Boyar-Peralta	71	91

Table 17: Overview of XOR-counts for the example defining polynomial of degree 8

Example 19. These examples dont fit on a page, so we just give an overview in Table 18. We can clearly see a big reduction in the required XOR gates even for very different polynomials.

	$x^{12}+x^{11}+x^2+x+1$	$x^{12}+x^{11}+x^{10}+x^8+x^6+x^4+x^3+x+1$
XOR-count before Boyar-Peralta	433	581
XOR-count after Boyar-Peralta	155	199

Table 18: Overview of XOR-counts for the example defining polynomial of degree 12

The data is generated using a polynomial basis, but there are other options like a normal basis. Figure 5 shows that running Algorithm 12 on a normal basis takes longer compared to a polynomial basis. Furthermore, in Table 19 we see that it has a worse average XOR-count and best XOR-count. It follows that the normal basis is not a good candidate for optimizing expression matrices. A likely explanation is that the polynomial basis includes 1 as a basis element, so the first column can always be computed without XOR gates.

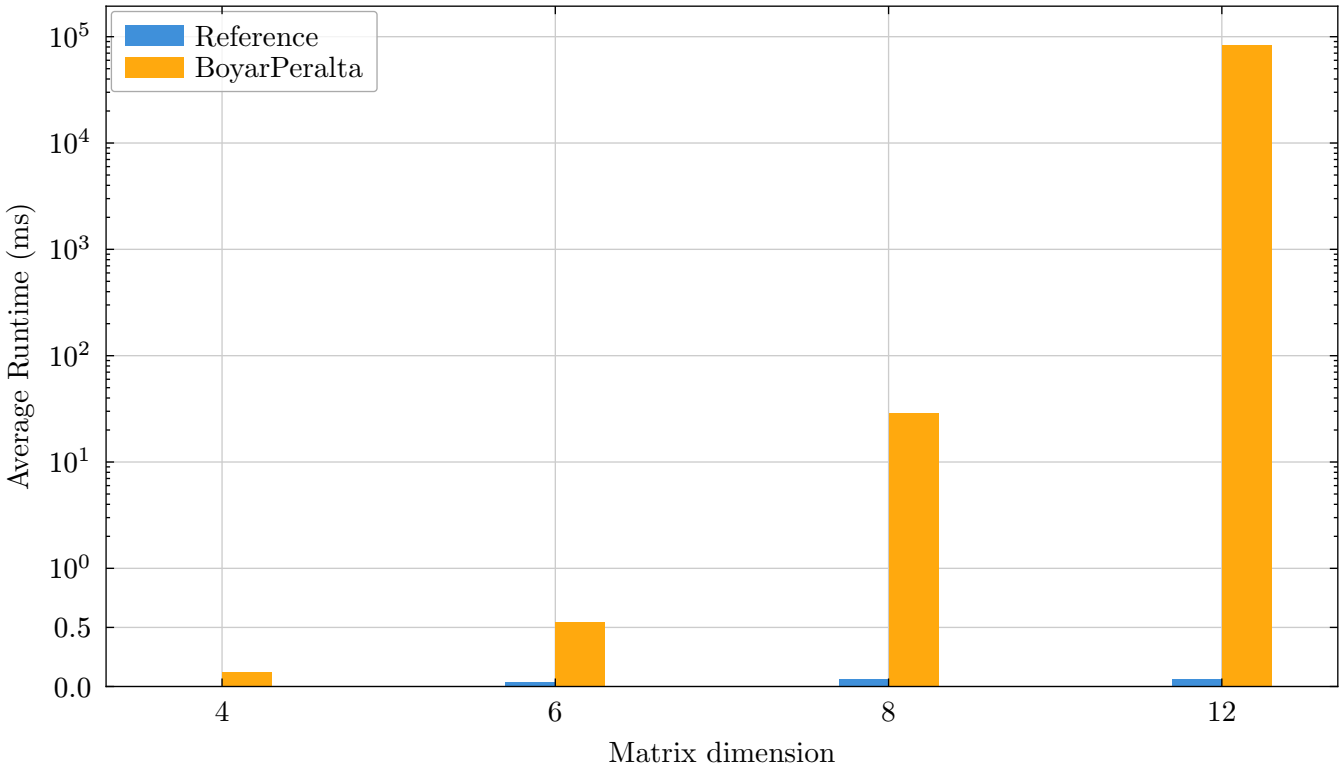

Figure 5: Average runtime for computing XOR-count of an expression matrix with normal basis. Lower is better

Dimension	Matrix count	Reference XOR-count	BoyarPeralta XOR-count	Best XOR-count
4	8	28	20	18
6	24	84	50.25	42
8	128	224	112.44	104
12	200	759.84	275.05	180

Table 19: XOR-count matrices with normal basis. Lower is better

4 Discussion, Conclusion and Further Research

We looked at ways to reuse algorithms found for multiplication with a fixed matrix to optimize multiplication with variable inputs. The runtime results were as expected, although the exact numbers themselves will depend on the hardware you use.

Some of the results for $n = 12, 16$ are also not as useful as we would have wanted, especially for fixed inputs. 200 elements are a very small fraction of the corresponding finite fields. For example, there are already 335 irreducible polynomials of degree 12 over $\mathbb{F}_2$.

We investigated the implementation efficiency of multiplication in binary finite fields through matrix representations over $\mathbb{F}_2$ and we reviewed the direct XOR-count and sequential XOR-count. The experimental results confirm that you can reach a better XOR-count using Boyar-Peralta given that there is enough time. Boyar-Peralta's algorithm consistently achieves the lowest XOR-count among the tested methods, but at the cost of a higher runtime. Gaussian-based heuristics provide faster alternatives, with

the Lookahead heuristic generally producing lower XOR-counts than the Markowitz heuristic while the increase in runtime is relatively small.

The main contribution of this work is the extension of these optimization techniques to expression matrices, which represent multiplication by an arbitrary field element, together with an experimental study of the influence of the irreducible polynomial used to define the binary finite field. The results show that the choice of irreducible polynomial has a big impact on the XOR-count, both for multiplication by a fixed field element and for multiplication of two variable field elements. For fixed multiplication the root of the defining polynomial seems to be consistently good choice for efficient multiplication with a fixed element

For future research there are a lot of options. Trying to optimize Boyar-Peralta can help by making it easier to run or by giving room for more computationally expensive heuristics. Finding mathematical explanation for why in Section 3.1.8 higher degrees of α often have higher XOR-count could help reduce the search space for possible inputs when trying to minimize XOR-count. We primarily looked at how to optimize the matrix creation, but you could also look at how to optimize the matrix-vector multiplication that happens after that, by using for example the distributive property. Alternatively, other variations of Boyar-Peralta or different algorithms entirely could be applied to the matrix creation. It would also be interesting to investigate matrices over finite fields other than $\mathbb{F}_2$ or to study multiplication in towers of finite fields.

Finally, the experimental evaluation was restricted to (expression) matrices for elements according to a polynomial basis. Investigating matrices that are used in the real world, such as those used in cryptography, may prove that some algorithms are more effective in real-world applications.

References

[1] E. G. AbdAllah, C. Huang, and Y. R. Kuang, "Advanced Encryption Standard New Instructions (AES-NI) Analysis: Security, Performance, and Power Consumption," in *Proceedings of the 2020 12th International Conference on Computer and Automation Engineering (ICCAE)*, ACM, 2020, pp. 167–172. doi: 10.1145/3384613.3384648.

[2] I. Kuzminykh, M. Yevdokymenko, and V. Sokolov, "Encryption Algorithms in IoT: Security vs Lifetime," *SSRN Electronic Journal*, 2021, doi: 10.2139/ssrn.4636161.

[3] J. Boyar, P. Matthews, and R. Peralta, "On the Shortest Linear Straight-Line Program for Computing Linear Forms," in *Mathematical Foundations of Computer Science 2008*, E. Ochmański and J. Tyszkiewicz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 168–179.

[4] G. L. Mullen and D. Panario, Eds., *Handbook of Finite Fields*. Boca Raton, FL: CRC Press, 2013.

[5] L. Kölsch, "XOR-Counts and Lightweight Multiplication with Fixed Elements in Binary Finite Fields," in *Advances in Cryptology – EUROCRYPT 2019*, in Lecture Notes in Computer Science, vol. 11476. Springer, Cham, 2019, pp. 285–312. doi: 10.1007/978-3-030-17653-2_10.

[6] Z. Xiang, X. Zeng, D. Lin, Z. Bao, and S. Zhang, "Optimizing Implementations of Linear Layers," *IACR Transactions on Symmetric Cryptology*, vol. 2020, no. 2, pp. 120–145, Jul. 2020, doi: 10.13154/tosc.v2020.i2.120-145.

[7] R. Zhao, B. Wu, R. Zhang, and Q. Zhang, "Designing Optimal Implementations of Linear Layers (Full Version)." [Online]. Available: https://eprint.iacr.org/2016/1118

[8] J. Jean, T. Peyrin, S. M. Sim, and J. Tourteaux, "Optimizing Implementations of Lightweight Building Blocks," *IACR Transactions on Symmetric Cryptology*, vol. 2017, no. 4, pp. 130–168, Dec. 2017, doi: 10.13154/tosc.v2017.i4.130-168.

[9] H. M. Markowitz, "The Elimination Form of the Inverse and Its Application to Linear Programming," *Management Science*, vol. 3, no. 3, pp. 255–269, 1957, Accessed: Aug. 05, 2026. [Online]. Available: http://www.jstor.org/stable/2627454

[10] M. Kauers and J. Moosbauer, "Good Pivots for Small Sparse Matrices," in *Computer Algebra in Scientific Computing*, Springer International Publishing, 2020, pp. 358–367. doi: 10.1007/978-3-030-60026-6_20.

[11] A. Rupp, J. Pelzl, C. Paar, M. Mertens, and A. Bogdanov, "A Parallel Hardware Architecture for fast Gaussian Elimination over GF(2)," in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 237–248. doi: 10.1109/FCCM.2006.12.

[12] S. Linton, G. Nebe, A. C. Niemeyer, R. Parker, and J. Thackray, "A parallel algorithm for Gaussian elimination over finite fields," *Journal of Algebra*, vol. 703, pp. 239–265, 2026, doi: https://doi.org/10.1016/j.jalgebra.2025.11.009.

[13] J. Boyar and R. Peralta, "A New Combinational Logic Minimization Technique with Applications to Cryptology," in *Experimental Algorithms*, P. Festa, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 178–189.

[14] A. Maximov, "AES MixColumn with 92 XOR gates." [Online]. Available: https://eprint.iacr.org/2019/833

[15] Q. Q. Tan and T. Peyrin, "Improved Heuristics for Short Linear Programs," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 1, pp. 203–230, Nov. 2019, doi: 10.13154/tches.v2020.i1.203-230.

[16] S. Li, S. Sun, C. Li, Z. Wei, and L. Hu, "Constructing Low-latency Involutory MDS Matrices with Lightweight Circuits," *IACR Transactions on Symmetric Cryptology*, vol. 2019, no. 1, pp. 84–117, Mar. 2019, doi: 10.13154/tosc.v2019.i1.84-117.

[17] M. Kurt Pehlivanoğlu and M. A. Demir, "Optimizing implementations of linear layers using two and higher input XOR gates," *PeerJ Computer Science*, vol. 10, p. e1820, Jan. 2024, doi: 10.7717/peerj-cs.1820.

[18] A. Baksi, V. A. Dasu, B. Karmakar, A. Chattopadhyay, and T. Isobe, "Three Input Exclusive-OR Gate Support for Boyar-Peralta's Algorithm," in *Progress in Cryptology – INDOCRYPT 2021: 22nd International Conference on Cryptology in India, Jaipur, India, December 12–15, 2021, Proceedings*, Jaipur, India: Springer-Verlag, 2021, pp. 141–158. doi: 10.1007/978-3-030-92518-5_7.

[19] S. Banik, Y. Funabiki, and T. Isobe, "More Results on Shortest Linear Programs," in *Advances in Information and Computer Security: 14th International Workshop on Security, IWSEC 2019, Tokyo, Japan, August 28–30, 2019, Proceedings*, Tokyo, Japan: Springer-Verlag, 2019, pp. 109–128. doi: 10.1007/978-3-030-26834-3_7.

[20] N. Zidarič, G. Gong, M. Aagaard, A. Jurišić, and O. Konovalov, "FFCSA - Finite Field Constructions, Search, and Algorithms," *ACM Commun. Comput. Algebra*, vol. 57, no. 2, pp. 57–64, Aug. 2023, doi: 10.1145/3614408.3614416.

[21] N. Zidarič and M. Aagaard, "Tower field support for synthesis of datapaths," in *Proceedings of the 20th ACM International Conference on Computing Frontiers*, in CF '23. Bologna, Italy: Association for Computing Machinery, 2023, pp. 217–218. doi: 10.1145/3587135.3592184.

[22] H. Huang and K. Tikhomirov, "Average-case analysis of the Gaussian elimination with partial pivoting," *Probability Theory and Related Fields*, vol. 189, pp. 1–67, Apr. 2024, doi: 10.1007/s00440-024-01276-2.

[23] Y. Jeon, S. Baek, G. Kim, and J. Kim, "A Framework for Generating S-Box Circuits with Boyar–Peralta Algorithm-Based Heuristics, and Its Applications to AES, SNOW3G, and Saturnin," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, no. 1, pp. 586–631, Dec. 2024, doi: 10.46586/tches.v2025.i1.586-631.

Appendix

Examples for fixed multiplication

For $x^8 + x^7 + x^6 + x + 1$
Multiplication by α:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} = \begin{pmatrix} x_8 \\ x_1 + x_8 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 + x_8 \\ x_7 + x_8 \end{pmatrix}$$

Multiplication by α^2:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} = \begin{pmatrix} x_7 + x_8 \\ x_7 \\ x_1 + x_8 \\ x_2 \\ x_3 \\ x_4 \\ x_5 + (x_7 + x_8) \\ x_6 + x_7 \end{pmatrix}$$

Multiplication by α^3:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} = \begin{pmatrix} x_6 + x_7 \\ x_6 + x_8 \\ x_7 \\ x_1 + x_8 \\ x_2 \\ x_3 \\ x_4 + (x_6 + x_7) \\ x_5 + (x_6 + x_8) \end{pmatrix}$$

Multiplication by α^4:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} = \begin{pmatrix} x_5 + (x_6 + x_8) \\ x_8 + (x_5 + x_7) \\ x_6 + x_8 \\ x_7 \\ x_1 + x_8 \\ x_2 \\ x_3 + (x_5 + (x_6 + x_8)) \\ x_4 + (x_8 + (x_5 + x_7)) \end{pmatrix}$$

Multiplication by α^5:

$$\begin{pmatrix} 0&0&0&1&1&0&1&1\\ 0&0&0&1&0&1&1&0\\ 0&0&0&0&1&0&1&1\\ 0&0&0&0&0&1&0&1\\ 0&0&0&0&0&0&1&0\\ 1&0&0&0&0&0&0&1\\ 0&1&0&1&1&0&1&1\\ 0&0&1&1&0&1&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8 \end{pmatrix} = \begin{pmatrix} (x_4+x_7)+(x_5+x_8)\\ x_6+(x_4+x_7)\\ x_7+(x_5+x_8)\\ x_6+x_8\\ x_7\\ x_1+x_8\\ x_2+((x_4+x_7)+(x_5+x_8))\\ x_3+(x_6+(x_4+x_7)) \end{pmatrix}$$

Multiplication by α^6:

$$\begin{pmatrix} 0&0&1&1&0&1&1&0\\ 0&0&1&0&1&1&0&1\\ 0&0&0&1&0&1&1&0\\ 0&0&0&0&1&0&1&1\\ 0&0&0&0&0&1&0&1\\ 0&0&0&0&0&0&1&0\\ 1&0&1&1&0&1&1&1\\ 0&1&1&0&1&1&0&1 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8 \end{pmatrix} = \begin{pmatrix} x_3+(x_6+(x_4+x_7))\\ x_5+(x_3+(x_6+x_8))\\ x_6+(x_4+x_7)\\ (x_3+(x_6+(x_4+x_7)))+(x_4+(x_5+(x_3+(x_6+x_8))))\\ x_6+x_8\\ x_7\\ (x_3+(x_6+(x_4+x_7)))+(x_1+x_8)\\ x_2+(x_5+(x_3+(x_6+x_8))) \end{pmatrix}$$

Multiplication by α^7:

$$\begin{pmatrix} 0&1&1&0&1&1&0&1\\ 0&1&0&1&1&0&1&1\\ 0&0&1&0&1&1&0&1\\ 0&0&0&1&0&1&1&0\\ 0&0&0&0&1&0&1&1\\ 0&0&0&0&0&1&0&1\\ 0&1&1&0&1&1&1&1\\ 1&1&0&1&1&0&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8 \end{pmatrix} = \begin{pmatrix} x_2+(x_5+(x_3+(x_6+x_8)))\\ (x_8+(x_5+x_7))+(x_2+x_4)\\ x_5+(x_3+(x_6+x_8))\\ ((x_8+(x_5+x_7))+(x_2+x_4))+(x_3+(x_2+(x_5+(x_3+(x_6+x_8)))))\\ x_8+(x_5+x_7)\\ x_6+x_8\\ x_7+(x_2+(x_5+(x_3+(x_6+x_8))))\\ ((x_8+(x_5+x_7))+(x_2+x_4))+(x_1+x_8) \end{pmatrix}$$

For $x^8+x^4+x^3+x+1$
Multiplication by α:

$$\begin{pmatrix} 0&0&0&0&0&0&0&1\\ 1&0&0&0&0&0&0&1\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&1\\ 0&0&0&1&0&0&0&1\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8 \end{pmatrix} = \begin{pmatrix} x_8\\ x_1+x_8\\ x_2\\ x_3+x_8\\ x_4+x_8\\ x_5\\ x_6\\ x_7 \end{pmatrix}$$

Multiplication by α^2:

$$\begin{pmatrix} 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&1&1\\ 1&0&0&0&0&0&0&1\\ 0&1&0&0&0&0&1&0\\ 0&0&1&0&0&0&1&1\\ 0&0&0&1&0&0&0&1\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_7\\ x_7+x_8\\ x_1+x_8\\ x_2+x_7\\ x_3+(x_7+x_8)\\ x_4+x_8\\ x_5\\ x_6 \end{pmatrix}$$

Multiplication by α^3:

$$\begin{pmatrix} 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&1&1&0\\ 0&0&0&0&0&0&1&1\\ 1&0&0&0&0&1&0&1\\ 0&1&0&0&0&1&1&0\\ 0&0&1&0&0&0&1&1\\ 0&0&0&1&0&0&0&1\\ 0&0&0&0&1&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_6\\ x_6+x_7\\ x_7+x_8\\ x_8+(x_1+x_6)\\ x_2+(x_6+x_7)\\ x_3+(x_7+x_8)\\ x_4+x_8\\ x_5 \end{pmatrix}$$

Multiplication by α^4:

$$\begin{pmatrix} 0&0&0&0&1&0&0&0\\ 0&0&0&0&1&1&0&0\\ 0&0&0&0&0&1&1&0\\ 0&0&0&0&1&0&1&1\\ 1&0&0&0&1&1&0&1\\ 0&1&0&0&0&1&1&0\\ 0&0&1&0&0&0&1&1\\ 0&0&0&1&0&0&0&1 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_5\\ x_5+x_6\\ x_6+x_7\\ x_5+(x_7+x_8)\\ (x_5+x_6)+(x_1+x_8)\\ x_2+(x_6+x_7)\\ x_3+(x_7+x_8)\\ x_4+x_8 \end{pmatrix}$$

Multiplication by α^5:

$$\begin{pmatrix} 0&0&0&1&0&0&0&1\\ 0&0&0&1&1&0&0&1\\ 0&0&0&0&1&1&0&0\\ 0&0&0&1&0&1&1&1\\ 0&0&0&1&1&0&1&0\\ 1&0&0&0&1&1&0&1\\ 0&1&0&0&0&1&1&0\\ 0&0&1&0&0&0&1&1 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_4+x_8\\ x_5+(x_4+x_8)\\ x_5+x_6\\ (x_4+x_8)+(x_6+x_7)\\ (x_5+(x_4+x_8))+(x_7+x_8)\\ (x_5+x_6)+(x_1+x_8)\\ x_2+(x_6+x_7)\\ x_3+(x_7+x_8) \end{pmatrix}$$

Multiplication by α^6:

$$\begin{pmatrix} 0&0&1&0&0&0&1&1\\ 0&0&1&1&0&0&1&0\\ 0&0&0&1&1&0&0&1\\ 0&0&1&0&1&1&1&1\\ 0&0&1&1&0&1&0&0\\ 0&0&0&1&1&0&1&0\\ 1&0&0&0&1&1&0&1\\ 0&1&0&0&0&1&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8 \end{pmatrix} = \begin{pmatrix} x_8+(x_3+x_7)\\ x_4+(x_3+x_7)\\ x_8+(x_4+x_5)\\ (x_8+(x_3+x_7))+(x_5+x_6)\\ (x_4+(x_3+x_7))+(x_6+x_7)\\ x_7+(x_4+x_5)\\ (x_5+x_6)+(x_1+x_8)\\ x_2+(x_6+x_7) \end{pmatrix}$$

Multiplication by α^7:

$$\begin{pmatrix} 0&1&0&0&0&1&1&0\\ 0&1&1&0&0&1&0&1\\ 0&0&1&1&0&0&1&0\\ 0&1&0&1&1&1&1&1\\ 0&1&1&0&1&0&0&1\\ 0&0&1&1&0&1&0&0\\ 0&0&0&1&1&0&1&0\\ 1&0&0&0&1&1&0&1 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8 \end{pmatrix} = \begin{pmatrix} x_7+(x_2+x_6)\\ x_3+(x_8+(x_2+x_6))\\ x_3+(x_4+x_7)\\ (x_5+(x_4+x_7))+(x_8+(x_2+x_6))\\ ((x_5+(x_4+x_7))+(x_8+(x_2+x_6)))+(x_6+(x_3+(x_4+x_7)))\\ x_7+(x_6+(x_3+(x_4+x_7)))\\ x_5+(x_4+x_7)\\ (x_1+x_2)+(x_5+(x_8+(x_2+x_6))) \end{pmatrix}$$

For $x^8+x^7+x^6+x^5+x^4+x+1$
Multiplication by α:

$$\begin{pmatrix} 0&0&0&0&0&0&0&1\\ 1&0&0&0&0&0&0&1\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&1\\ 0&0&0&0&1&0&0&1\\ 0&0&0&0&0&1&0&1\\ 0&0&0&0&0&0&1&1 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8 \end{pmatrix} = \begin{pmatrix} x_8\\ x_1+x_8\\ x_2\\ x_3\\ x_4+x_8\\ x_5+x_8\\ x_6+x_8\\ x_7+x_8 \end{pmatrix}$$

Multiplication by α^2:

$$\begin{pmatrix} 0&0&0&0&0&0&1&1\\ 0&0&0&0&0&0&1&0\\ 1&0&0&0&0&0&0&1\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&1&1\\ 0&0&0&1&0&0&1&0\\ 0&0&0&0&1&0&1&0\\ 0&0&0&0&0&1&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8 \end{pmatrix} = \begin{pmatrix} x_7+x_8\\ x_7\\ x_1+x_8\\ x_2\\ x_3+(x_7+x_8)\\ x_4+x_7\\ x_5+x_7\\ x_6+x_7 \end{pmatrix}$$

Multiplication by α^3:

$$\begin{pmatrix} 0&0&0&0&0&1&1&0\\ 0&0&0&0&0&1&0&1\\ 0&0&0&0&0&0&1&0\\ 1&0&0&0&0&0&0&1\\ 0&1&0&0&0&1&1&0\\ 0&0&1&0&0&1&0&1\\ 0&0&0&1&0&1&0&0\\ 0&0&0&0&1&1&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_6 + x_7\\ x_6 + x_8\\ x_7\\ x_1 + x_8\\ x_2 + (x_6 + x_7)\\ x_3 + (x_6 + x_8)\\ x_4 + x_6\\ x_5 + x_6 \end{pmatrix}$$

Multiplication by α^4:

$$\begin{pmatrix} 0&0&0&0&1&1&0&0\\ 0&0&0&0&1&0&1&0\\ 0&0&0&0&0&1&0&1\\ 0&0&0&0&0&0&1&0\\ 1&0&0&0&1&1&0&1\\ 0&1&0&0&1&0&1&0\\ 0&0&1&0&1&0&0&1\\ 0&0&0&1&1&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_5 + x_6\\ x_5 + x_7\\ x_6 + x_8\\ x_7\\ (x_6 + x_8) + (x_1 + x_5)\\ x_2 + (x_5 + x_7)\\ x_8 + (x_3 + x_5)\\ x_4 + x_5 \end{pmatrix}$$

Multiplication by α^5:

$$\begin{pmatrix} 0&0&0&1&1&0&0&0\\ 0&0&0&1&0&1&0&0\\ 0&0&0&0&1&0&1&0\\ 0&0&0&0&0&1&0&1\\ 0&0&0&1&1&0&1&0\\ 1&0&0&1&0&1&0&1\\ 0&1&0&1&0&0&1&0\\ 0&0&1&1&0&0&0&1 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_4 + x_5\\ x_4 + x_6\\ x_5 + x_7\\ x_6 + x_8\\ x_4 + (x_5 + x_7)\\ (x_6 + x_8) + (x_1 + x_4)\\ x_7 + (x_2 + x_4)\\ x_8 + (x_3 + x_4) \end{pmatrix}$$

Multiplication by α^6:

$$\begin{pmatrix} 0&0&1&1&0&0&0&1\\ 0&0&1&0&1&0&0&1\\ 0&0&0&1&0&1&0&0\\ 0&0&0&0&1&0&1&0\\ 0&0&1&1&0&1&0&0\\ 0&0&1&0&1&0&1&1\\ 1&0&1&0&0&1&0&0\\ 0&1&1&0&0&0&1&1 \end{pmatrix} \cdot \begin{pmatrix} x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8 \end{pmatrix} = \begin{pmatrix} x_4 + (x_3 + x_8)\\ x_5 + (x_3 + x_8)\\ x_4 + x_6\\ x_5 + x_7\\ x_3 + (x_4 + x_6)\\ x_7 + (x_5 + (x_3 + x_8))\\ x_6 + (x_1 + x_3)\\ (x_7 + (x_5 + (x_3 + x_8))) + (x_2 + x_5) \end{pmatrix}$$

Multiplication by α^7:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} = \begin{pmatrix} (x_2 + x_7) + (x_3 + x_8) \\ x_4 + (x_2 + x_7) \\ x_5 + (x_3 + x_8) \\ x_4 + x_6 \\ x_2 + (x_5 + (x_3 + x_8)) \\ (x_4 + x_6) + (x_8 + (x_2 + x_7)) \\ x_2 + x_5 \\ ((x_4 + x_6) + (x_8 + (x_2 + x_7))) + (x_1 + x_4) \end{pmatrix}$$

For $x^{12} + x^3 + 1$
Multiplication by α:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_{12} \\ x_1 \\ x_2 \\ x_3 + x_{12} \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \end{pmatrix}$$

Multiplication by α^2:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_{11} \\ x_{12} \\ x_1 \\ x_2 + x_{11} \\ x_3 + x_{12} \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{pmatrix}$$

Multiplication by α^3:

$$\begin{pmatrix} 0&0&0&0&0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&1\\ 1&0&0&0&0&0&0&0&0&1&0&0&0\\ 0&1&0&0&0&0&0&0&0&0&1&0&0\\ 0&0&1&0&0&0&0&0&0&0&0&1&0\\ 0&0&0&1&0&0&0&0&0&0&0&0&1\\ 0&0&0&0&1&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&1&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&1&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&1&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&1&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8\\x_9\\x_{10}\\x_{11}\\x_{12} \end{pmatrix} = \begin{pmatrix} x_{10}\\x_{11}\\x_{12}\\x_1+x_{10}\\x_2+x_{11}\\x_3+x_{12}\\x_4\\x_5\\x_6\\x_7\\x_8\\x_9 \end{pmatrix}$$

Multiplication by α^4:

$$\begin{pmatrix} 0&0&0&0&0&0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&0&1&0&0&1\\ 1&0&0&0&0&0&0&0&0&0&1&0&0\\ 0&1&0&0&0&0&0&0&0&0&0&1&0\\ 0&0&1&0&0&0&0&0&0&0&0&0&1\\ 0&0&0&1&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&1&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&1&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&1&0&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8\\x_9\\x_{10}\\x_{11}\\x_{12} \end{pmatrix} = \begin{pmatrix} x_9\\x_{10}\\x_{11}\\x_9+x_{12}\\x_1+x_{10}\\x_2+x_{11}\\x_3+x_{12}\\x_4\\x_5\\x_6\\x_7\\x_8 \end{pmatrix}$$

Multiplication by α^5:

$$\begin{pmatrix} 0&0&0&0&0&0&0&0&1&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&0&1&0&0&1&0\\ 0&0&0&0&0&0&0&0&0&1&0&0&1\\ 1&0&0&0&0&0&0&0&0&0&1&0&0\\ 0&1&0&0&0&0&0&0&0&0&0&1&0\\ 0&0&1&0&0&0&0&0&0&0&0&0&1\\ 0&0&0&1&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&1&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&1&0&0&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8\\x_9\\x_{10}\\x_{11}\\x_{12} \end{pmatrix} = \begin{pmatrix} x_8\\x_9\\x_{10}\\x_8+x_{11}\\x_9+x_{12}\\x_1+x_{10}\\x_2+x_{11}\\x_3+x_{12}\\x_4\\x_5\\x_6\\x_7 \end{pmatrix}$$

Multiplication by α^6:

$$\begin{pmatrix} 0&0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0&0&1&0 \\ 0&0&0&0&0&0&0&0&1&0&0&1 \\ 1&0&0&0&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&1&0 \\ 0&0&1&0&0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_7 \\ x_8 \\ x_9 \\ x_7 + x_{10} \\ x_8 + x_{11} \\ x_9 + x_{12} \\ x_1 + x_{10} \\ x_2 + x_{11} \\ x_3 + x_{12} \\ x_4 \\ x_5 \\ x_6 \end{pmatrix}$$

Multiplication by α^7:

$$\begin{pmatrix} 0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0&0&1&0 \\ 0&0&0&0&0&0&0&0&1&0&0&1 \\ 1&0&0&0&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&1&0 \\ 0&0&1&0&0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_6 \\ x_7 \\ x_8 \\ x_6 + x_9 \\ x_7 + x_{10} \\ x_8 + x_{11} \\ x_9 + x_{12} \\ x_1 + x_{10} \\ x_2 + x_{11} \\ x_3 + x_{12} \\ x_4 \\ x_5 \end{pmatrix}$$

Multiplication by α^8:

$$\begin{pmatrix} 0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&1&0&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0&0&1&0 \\ 0&0&0&0&0&0&0&0&1&0&0&1 \\ 1&0&0&0&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&1&0 \\ 0&0&1&0&0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_5 \\ x_6 \\ x_7 \\ x_5 + x_8 \\ x_6 + x_9 \\ x_7 + x_{10} \\ x_8 + x_{11} \\ x_9 + x_{12} \\ x_1 + x_{10} \\ x_2 + x_{11} \\ x_3 + x_{12} \\ x_4 \end{pmatrix}$$

Multiplication by α^9:

$$\begin{pmatrix} 0&0&0&1&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&1 \\ 1&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&1&0 \\ 0&0&1&0&0&0&0&0&0&0&0&0&1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_4 \\ x_5 \\ x_6 \\ x_4 + x_7 \\ x_5 + x_8 \\ x_6 + x_9 \\ x_7 + x_{10} \\ x_8 + x_{11} \\ x_9 + x_{12} \\ x_1 + x_{10} \\ x_2 + x_{11} \\ x_3 + x_{12} \end{pmatrix}$$

Multiplication by α^{10}:

$$\begin{pmatrix} 0&0&1&0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&1&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&1 \\ 1&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_3 + x_{12} \\ x_4 \\ x_5 \\ x_6 + (x_3 + x_{12}) \\ x_4 + x_7 \\ x_5 + x_8 \\ x_6 + x_9 \\ x_7 + x_{10} \\ x_8 + x_{11} \\ x_9 + x_{12} \\ x_1 + x_{10} \\ x_2 + x_{11} \end{pmatrix}$$

Multiplication by α^{11}:

$$\begin{pmatrix} 0&1&0&0&0&0&0&0&0&0&1&0 \\ 0&0&1&0&0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&1&0&0&1&0&0&0&0&0&1&0 \\ 0&0&1&0&0&1&0&0&0&0&0&1 \\ 0&0&0&1&0&0&1&0&0&0&0&0 \\ 0&0&0&0&1&0&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0&0&1&0 \\ 0&0&0&0&0&0&0&0&1&0&0&1 \\ 1&0&0&0&0&0&0&0&0&1&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_2 + x_{11} \\ x_3 + x_{12} \\ x_4 \\ x_5 + (x_2 + x_{11}) \\ x_6 + (x_3 + x_{12}) \\ x_4 + x_7 \\ x_5 + x_8 \\ x_6 + x_9 \\ x_7 + x_{10} \\ x_8 + x_{11} \\ x_9 + x_{12} \\ x_1 + x_{10} \end{pmatrix}$$

For $x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$
Multiplication by α:

$$\begin{pmatrix} 0&0&0&0&0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0&0&0&0&1 \\ 0&1&0&0&0&0&0&0&0&0&0&1 \\ 0&0&1&0&0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0&0&0&0&1 \\ 0&0&0&0&1&0&0&0&0&0&0&1 \\ 0&0&0&0&0&1&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0&0&0&0&1 \\ 0&0&0&0&0&0&0&1&0&0&0&1 \\ 0&0&0&0&0&0&0&0&1&0&0&1 \\ 0&0&0&0&0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&0&0&0&0&1&1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_{12} \\ x_1 + x_{12} \\ x_2 + x_{12} \\ x_3 + x_{12} \\ x_4 + x_{12} \\ x_5 + x_{12} \\ x_6 + x_{12} \\ x_7 + x_{12} \\ x_8 + x_{12} \\ x_9 + x_{12} \\ x_{10} + x_{12} \\ x_{11} + x_{12} \end{pmatrix}$$

Multiplication by α^2:

$$\begin{pmatrix} 0&0&0&0&0&0&0&0&0&0&1&1 \\ 0&0&0&0&0&0&0&0&0&0&1&0 \\ 1&0&0&0&0&0&0&0&0&0&1&0 \\ 0&1&0&0&0&0&0&0&0&0&1&0 \\ 0&0&1&0&0&0&0&0&0&0&1&0 \\ 0&0&0&1&0&0&0&0&0&0&1&0 \\ 0&0&0&0&1&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&0&0&0&1&0 \\ 0&0&0&0&0&0&1&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1&0&0&1&0 \\ 0&0&0&0&0&0&0&0&1&0&1&0 \\ 0&0&0&0&0&0&0&0&0&1&1&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_{11} + x_{12} \\ x_{11} \\ x_1 + x_{11} \\ x_2 + x_{11} \\ x_3 + x_{11} \\ x_4 + x_{11} \\ x_5 + x_{11} \\ x_6 + x_{11} \\ x_7 + x_{11} \\ x_8 + x_{11} \\ x_9 + x_{11} \\ x_{10} + x_{11} \end{pmatrix}$$

Multiplication by α^3:

$$\begin{pmatrix} 0&0&0&0&0&0&0&0&0&1&1&0 \\ 0&0&0&0&0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&0&0&0&1&0&0 \\ 1&0&0&0&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&1&0&0 \\ 0&0&1&0&0&0&0&0&0&1&0&0 \\ 0&0&0&1&0&0&0&0&0&1&0&0 \\ 0&0&0&0&1&0&0&0&0&1&0&0 \\ 0&0&0&0&0&1&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0&1&0&0 \\ 0&0&0&0&0&0&0&0&1&1&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_{10} + x_{11} \\ x_{10} + x_{12} \\ x_{10} \\ x_1 + x_{10} \\ x_2 + x_{10} \\ x_3 + x_{10} \\ x_4 + x_{10} \\ x_5 + x_{10} \\ x_6 + x_{10} \\ x_7 + x_{10} \\ x_8 + x_{10} \\ x_9 + x_{10} \end{pmatrix}$$

Multiplication by α^4:

$$\begin{pmatrix} 0&0&0&0&0&0&0&0&1&1&0&0\\ 0&0&0&0&0&0&0&0&1&0&1&0\\ 0&0&0&0&0&0&0&0&1&0&0&1\\ 0&0&0&0&0&0&0&0&1&0&0&0\\ 1&0&0&0&0&0&0&0&1&0&0&0\\ 0&1&0&0&0&0&0&0&1&0&0&0\\ 0&0&1&0&0&0&0&0&1&0&0&0\\ 0&0&0&1&0&0&0&0&1&0&0&0\\ 0&0&0&0&1&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0&1&0&0&0\\ 0&0&0&0&0&0&1&0&1&0&0&0\\ 0&0&0&0&0&0&0&1&1&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8\\x_9\\x_{10}\\x_{11}\\x_{12} \end{pmatrix} = \begin{pmatrix} x_9+x_{10}\\x_9+x_{11}\\x_9+x_{12}\\x_9\\x_1+x_9\\x_2+x_9\\x_3+x_9\\x_4+x_9\\x_5+x_9\\x_6+x_9\\x_7+x_9\\x_8+x_9 \end{pmatrix}$$

Multiplication by α^5:

$$\begin{pmatrix} 0&0&0&0&0&0&0&1&1&0&0&0\\ 0&0&0&0&0&0&0&1&0&1&0&0\\ 0&0&0&0&0&0&0&1&0&0&1&0\\ 0&0&0&0&0&0&0&1&0&0&0&1\\ 0&0&0&0&0&0&0&1&0&0&0&0\\ 1&0&0&0&0&0&0&1&0&0&0&0\\ 0&1&0&0&0&0&0&1&0&0&0&0\\ 0&0&1&0&0&0&0&1&0&0&0&0\\ 0&0&0&1&0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&1&0&0&0&0\\ 0&0&0&0&0&1&0&1&0&0&0&0\\ 0&0&0&0&0&0&1&1&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8\\x_9\\x_{10}\\x_{11}\\x_{12} \end{pmatrix} = \begin{pmatrix} x_8+x_9\\x_8+x_{10}\\x_8+x_{11}\\x_8+x_{12}\\x_8\\x_1+x_8\\x_2+x_8\\x_3+x_8\\x_4+x_8\\x_5+x_8\\x_6+x_8\\x_7+x_8 \end{pmatrix}$$

Multiplication by α^6:

$$\begin{pmatrix} 0&0&0&0&0&0&1&1&0&0&0&0\\ 0&0&0&0&0&0&1&0&1&0&0&0\\ 0&0&0&0&0&0&1&0&0&1&0&0\\ 0&0&0&0&0&0&1&0&0&0&1&0\\ 0&0&0&0&0&0&1&0&0&0&0&1\\ 0&0&0&0&0&0&1&0&0&0&0&0\\ 1&0&0&0&0&0&1&0&0&0&0&0\\ 0&1&0&0&0&0&1&0&0&0&0&0\\ 0&0&1&0&0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&1&0&0&0&0&0\\ 0&0&0&0&1&0&1&0&0&0&0&0\\ 0&0&0&0&0&1&1&0&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8\\x_9\\x_{10}\\x_{11}\\x_{12} \end{pmatrix} = \begin{pmatrix} x_7+x_8\\x_7+x_9\\x_7+x_{10}\\x_7+x_{11}\\x_7+x_{12}\\x_7\\x_1+x_7\\x_2+x_7\\x_3+x_7\\x_4+x_7\\x_5+x_7\\x_6+x_7 \end{pmatrix}$$

Multiplication by α^7:

$$\begin{pmatrix} 0&0&0&0&0&1&1&0&0&0&0&0 \\ 0&0&0&0&0&1&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0&0&1&0&0 \\ 0&0&0&0&0&1&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&0&0&0&0&1 \\ 0&0&0&0&0&1&0&0&0&0&0&0 \\ 1&0&0&0&0&1&0&0&0&0&0&0 \\ 0&1&0&0&0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&1&0&0&0&0&0&0 \\ 0&0&0&1&0&1&0&0&0&0&0&0 \\ 0&0&0&0&1&1&0&0&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_6 + x_7 \\ x_6 + x_8 \\ x_6 + x_9 \\ x_6 + x_{10} \\ x_6 + x_{11} \\ x_6 + x_{12} \\ x_6 \\ x_1 + x_6 \\ x_2 + x_6 \\ x_3 + x_6 \\ x_4 + x_6 \\ x_5 + x_6 \end{pmatrix}$$

Multiplication by α^8:

$$\begin{pmatrix} 0&0&0&0&1&1&0&0&0&0&0&0 \\ 0&0&0&0&1&0&1&0&0&0&0&0 \\ 0&0&0&0&1&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0&1&0&0&0 \\ 0&0&0&0&1&0&0&0&0&1&0&0 \\ 0&0&0&0&1&0&0&0&0&0&1&0 \\ 0&0&0&0&1&0&0&0&0&0&0&1 \\ 0&0&0&0&1&0&0&0&0&0&0&0 \\ 1&0&0&0&1&0&0&0&0&0&0&0 \\ 0&1&0&0&1&0&0&0&0&0&0&0 \\ 0&0&1&0&1&0&0&0&0&0&0&0 \\ 0&0&0&1&1&0&0&0&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_5 + x_6 \\ x_5 + x_7 \\ x_5 + x_8 \\ x_5 + x_9 \\ x_5 + x_{10} \\ x_5 + x_{11} \\ x_5 + x_{12} \\ x_5 \\ x_1 + x_5 \\ x_2 + x_5 \\ x_3 + x_5 \\ x_4 + x_5 \end{pmatrix}$$

Multiplication by α^9:

$$\begin{pmatrix} 0&0&0&1&1&0&0&0&0&0&0&0 \\ 0&0&0&1&0&1&0&0&0&0&0&0 \\ 0&0&0&1&0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&1&0&0&0&0 \\ 0&0&0&1&0&0&0&0&1&0&0&0 \\ 0&0&0&1&0&0&0&0&0&1&0&0 \\ 0&0&0&1&0&0&0&0&0&0&1&0 \\ 0&0&0&1&0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0&0&0&0&0 \\ 1&0&0&1&0&0&0&0&0&0&0&0 \\ 0&1&0&1&0&0&0&0&0&0&0&0 \\ 0&0&1&1&0&0&0&0&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_4 + x_5 \\ x_4 + x_6 \\ x_4 + x_7 \\ x_4 + x_8 \\ x_4 + x_9 \\ x_4 + x_{10} \\ x_4 + x_{11} \\ x_4 + x_{12} \\ x_4 \\ x_1 + x_4 \\ x_2 + x_4 \\ x_3 + x_4 \end{pmatrix}$$

Multiplication by α^{10}:

$$\begin{pmatrix} 0&0&1&1&0&0&0&0&0&0&0&0 \\ 0&0&1&0&1&0&0&0&0&0&0&0 \\ 0&0&1&0&0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&1&0&0&0&0&0 \\ 0&0&1&0&0&0&0&1&0&0&0&0 \\ 0&0&1&0&0&0&0&0&1&0&0&0 \\ 0&0&1&0&0&0&0&0&0&1&0&0 \\ 0&0&1&0&0&0&0&0&0&0&1&0 \\ 0&0&1&0&0&0&0&0&0&0&0&1 \\ 0&0&1&0&0&0&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0&0&0&0&0 \\ 0&1&1&0&0&0&0&0&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_3 + x_4 \\ x_3 + x_5 \\ x_3 + x_6 \\ x_3 + x_7 \\ x_3 + x_8 \\ x_3 + x_9 \\ x_3 + x_{10} \\ x_3 + x_{11} \\ x_3 + x_{12} \\ x_3 \\ x_1 + x_3 \\ x_2 + x_3 \end{pmatrix}$$

Multiplication by α^{11}:

$$\begin{pmatrix} 0&1&1&0&0&0&0&0&0&0&0&0 \\ 0&1&0&1&0&0&0&0&0&0&0&0 \\ 0&1&0&0&1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&1&0&0&0&0&0 \\ 0&1&0&0&0&0&0&1&0&0&0&0 \\ 0&1&0&0&0&0&0&0&1&0&0&0 \\ 0&1&0&0&0&0&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&1&0 \\ 0&1&0&0&0&0&0&0&0&0&0&1 \\ 0&1&0&0&0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0&0&0&0&0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{pmatrix} = \begin{pmatrix} x_2 + x_3 \\ x_2 + x_4 \\ x_2 + x_5 \\ x_2 + x_6 \\ x_2 + x_7 \\ x_2 + x_8 \\ x_2 + x_9 \\ x_2 + x_{10} \\ x_2 + x_{11} \\ x_2 + x_{12} \\ x_2 \\ x_1 + x_2 \end{pmatrix}$$

Files

gap

boyarPeralta.gap

```
OneHot := function(n, i)
  local result;
  result := List([1..n], x -> 0);
  result[i] := 1;
  return result;
end;

Reachable := function(target, maxElements, start, base)
  local i;

  if Sum(target) <= maxElements then return true; fi;

  # If Reachable is called every iteration as intended
  # the minimum number of elements required is also maxElements,
  # Length(base) - start + 1 is the number of elements left
  if Length(base) - start + 1 < maxElements then return false; fi;
```

```
17
18   if maxElements = 0 then return false; fi;
19
20   if maxElements = 1 then
21     for i in [ start .. Length(base) ] do
22       if target = base[i] then return true; fi;
23     od;
24
25     return false;
26   fi;
27
28   if Reachable(target, maxElements, start + 1, base) then return true; fi;
29
30   if Reachable(List(target + base[start], x -> x mod 2), maxElements - 1, start + 1,
   base) then return true; fi;
31
32
33   return false;
34 end;
35
36 UpdateDistance := function(A, base, b, previousDist)
37   local newDist, i;
38
39   newDist := ShallowCopy(previousDist);
40   for i in [1 .. Length(A)] do
41     if previousDist[i] = 0 then continue; fi;
42
43     if Reachable(List(A[i] + b, x -> x mod 2), previousDist[i] - 1, 1, base) then
44       newDist[i] := previousDist[i] - 1;
45     fi;
46   od;
47
48   return newDist;
49 end;
50
51 FindBestCombination := function(A, base, previousDist)
52   local norm, bestDist, bestDistScore, bestCombination, combination, distC, score, i, j,
   bestNorm;
53
54   bestDistScore := fail;
55   bestDist := fail;
56   bestNorm := -1;
57
58   for i in [1 .. Length(previousDist)] do
59     if previousDist[i] = 1 then
60       combination := ShallowCopy(A[i]);
61       distC := UpdateDistance(A, base, combination, previousDist);
62
63       return rec(
64         combination := combination,
65         dist := distC
```

```
      );
    fi;
  od;

  for i in [1..Length(base)] do
    for j in [i+1..Length(base)] do
      combination := List(base[i] + base[j], x -> x mod 2);
      if combination in base then continue; fi;

      distC := UpdateDistance(A, base, combination, previousDist);

      score := Sum(distC);
      norm := distC * distC;

      if score < bestDistScore or score = bestDistScore and norm > bestNorm then
        bestDistScore := score;
        bestNorm := norm;
        bestCombination := combination;
        bestDist := distC;
      fi;
    od;
  od;

  return rec(
    combination := bestCombination,
    dist := bestDist
  );
end;

BoyarPeralta := function(A)
  local base, dist, colCount, result;
  colCount := Length(A[1]);

  base := List([1..colCount], x -> OneHot(colCount, x));
  dist := List(A, row -> Sum(row) - 1);

  while Sum(dist) <> 0 do
    result := FindBestCombination(A, base, dist);

    # Print("Combination: ", result.combination, "\n");
    # Print("New dist:    ", result.dist, "\n");
    # Print("Sum:         ", Sum(result.dist), "\n\n");

    dist := result.dist;
    Add(base, result.combination);
  od;

  # Print("Steps required = ", Length(base) - Length(A[1]), "\n");

  return base;
end;
```

```
117
118 ReconstructBase := function(base)
119   local expr, v, lhs, rhs, i, j, k, options, optionAdditions, additions, index;
120   expr := [];
121   additions := [];
122
123   for i in [1 .. Length(base)] do
124     if Sum(base[i]) = 1 then
125       for k in [1 .. Length(base[i])] do
126         if base[i][k] = 1 then
127           expr[i] := Concatenation("x_", String(k));
128           additions[i] := 0;
129           break;
130         fi;
131       od;
132
133       continue;
134     fi;
135
136     options := [];
137     optionAdditions := [];
138
139     for j in [1 .. i-1] do
140       for k in [j+1 .. i-1] do
141         v := List(base[j] + base[k], x -> x mod 2);
142
143         if v = base[i] then
144           Add(options, [j, k]);
145           Add(optionAdditions, additions[j] + additions[k] + 1);
146         fi;
147       od;
148     od;
149
150     if not IsEmpty(options) then
151       index := PositionMinimum(optionAdditions);
152       j := options[index][1];
153       k := options[index][2];
154
155       if Position(expr[j], '+') <> fail then
156         lhs := Concatenation("(", expr[j], ")");
157       else
158         lhs := expr[j];
159       fi;
160
161       if Position(expr[k], '+') <> fail then
162         rhs := Concatenation("(", expr[k], ")");
163       else
164         rhs := expr[k];
165       fi;
166
167       expr[i] := Concatenation(
```

```
168         lhs, " + ", rhs
169       );
170       additions[i] := optionAdditions[index];
171     else
172       Display("ERROR BP RECONSTRUCT");
173     fi;
174
175   od;
176
177   return expr;
178 end;
179
180 FormatBPBase := function(A, base)
181   local result, row, pos, exp, i;
182   result := [];
183   exp := ReconstructBase(base);
184
185   for i in [1 .. Length(A)] do
186     row := A[i];
187     pos := Position(base, row);
188
189     result[i] := exp[pos];
190   od;
191
192   return result;
193 end;
194
```

gap

exhaustive.gap

```
 1 ExhaustiveGaussian := function(A)
 2   local M, n, bestOps, P, Search;
 3
 4   M := List(A, ShallowCopy);
 5   n := Length(A);
 6   bestOps := fail;
 7   P := fail;
 8
 9   Search := function(M, ops, lockedRows, lockedCols)
10     local row, col, otherRow, newM, newOps, pivotRow, pivotCol;
11
12     # pruning
13     if bestOps <> fail and Length(ops) >= Length(bestOps) then
14       return;
15     fi;
16
17     # termination
18     if Length(lockedRows) = n then
19       if bestOps = fail or Length(ops) < Length(bestOps) then
20         bestOps := ShallowCopy(ops);
21         P := List(M, ShallowCopy);
```

```
      fi;
      return;
    fi;

    for pivotRow in Filtered([1..n], x -> not x in lockedRows) do
      for pivotCol in Filtered([1..n], x -> not x in lockedCols) do
        if M[pivotRow][pivotCol] = 0 then continue; fi;

        newM := List(M, ShallowCopy);
        newOps := ShallowCopy(ops);

        for otherRow in [1..n] do
          if otherRow <> pivotRow and newM[otherRow][pivotCol] <> 0 then
            newM[otherRow] := List(newM[otherRow] + newM[pivotRow], x -> x mod 2);
            Add(newOps, [otherRow, pivotRow]);
          fi;
        od;

        Search(
          newM,
          newOps,
          Concatenation(lockedRows, [pivotRow]),
          Concatenation(lockedCols, [pivotCol])
        );
      od;
    od;
  end;

  Search(M, [], [], []);

  # Print(bestOps, "\n");

  return [P, bestOps];
end;

```

gap

markowitz.gap

```
MarkowitzGaussian := function(A)
  local M, n, ops, lockedRows, lockedCols, row, col, bestCost, cost, pivotRow, pivotCol,
  rowWeight, colWeight, otherRow, _;

  M := List(A, ShallowCopy);
  n := Length(M);
  ops := [];

  lockedRows := [];
  lockedCols := [];

  for _ in [1..n] do

```

```
    bestCost := fail;
    for row in Filtered([1..n], x -> not x in lockedRows) do

      rowWeight := Sum(M[row]);

      for col in Filtered([1..n], x -> not x in lockedCols) do

        if M[row][col] = 0 then continue; fi;

        colWeight := Sum([1..n], r -> M[r][col]);

        cost := (rowWeight - 1) * (colWeight - 1);

        if cost < bestCost then
          bestCost := cost;
          pivotRow := row;
          pivotCol := col;
        fi;
      od;
    od;

    if bestCost = fail then break; fi;

    Add(lockedRows, pivotRow);
    Add(lockedCols, pivotCol);

    for otherRow in [1..n] do
      if otherRow <> pivotRow and M[otherRow][pivotCol] <> 0 then
        M[otherRow] := List(M[otherRow] + M[pivotRow], x -> x mod 2);
        Add(ops, [otherRow, pivotRow]);
      fi;
    od;
  od;

  return [M, ops];
end;

```

gap

lookahead.gap

```
LookaheadGaussian := function(A)
  local M, n, ops, lockedRows, lockedCols, row, col, heuristic, bestHeuristic, pivotRow,
  pivotCol, otherRow, cost, _, sum, moveCount;

  M := List(A, ShallowCopy);
  n := Length(M);
  ops := [];

  lockedRows := [];
  lockedCols := [];

```

```
  for _ in [1..n] do

    bestHeuristic := fail;
    for row in Filtered([1..n], x -> not x in lockedRows) do
      for col in Filtered([1..n], x -> not x in lockedCols) do
        if M[row][col] = 0 then continue; fi;

        cost := 0;
        sum := Sum(M[row]);
        moveCount := 0;

        for otherRow in [1..n] do
          if otherRow <> row and M[otherRow][col] <> 0 then
            moveCount := moveCount + 1;
            cost := cost + sum - 2 * M[otherRow] * M[row];
          fi;
        od;

        if moveCount = 0 then
          moveCount := 1;
        fi;

        heuristic := cost / moveCount;
        if heuristic < bestHeuristic then
          bestHeuristic := heuristic;
          pivotRow := row;
          pivotCol := col;
        fi;
      od;
    od;

    if bestHeuristic = fail then break; fi;

    # Print("bestHeuristic (", pivotRow, ", ", pivotCol, ")\n");

    Add(lockedRows, pivotRow);
    Add(lockedCols, pivotCol);

    for otherRow in [1..n] do
      if otherRow <> pivotRow and M[otherRow][pivotCol] <> 0 then
        M[otherRow] := List(M[otherRow] + M[pivotRow], x -> x mod 2);
        Add(ops, [otherRow, pivotRow]);
      fi;
    od;
  od;
  return [M, ops];
end;

```
gap

naive.gap

```
NaiveGaussian := function(A)
  local M, n, ops, lockedRows, pivotRow, pivotCol, otherRow;

  M := List(A, ShallowCopy);
  n := Length(M);
  ops := [];
  lockedRows := [];

  for pivotCol in [1..n] do
    for pivotRow in Filtered([1..n], x -> not x in lockedRows) do
      if M[pivotRow][pivotCol] <> 0 then break; fi;
    od;
    Add(lockedRows, pivotRow);

    for otherRow in [1..n] do
            if otherRow <> pivotRow and M[otherRow][pivotCol] <> 0 then
        M[otherRow] := List(M[otherRow] + M[pivotRow], x -> x mod 2);
        Add(ops, [otherRow, pivotRow]);
      fi;
    od;
  od;
  return [M, ops];
end;

```