A shortcut to the shortest even cycle

by

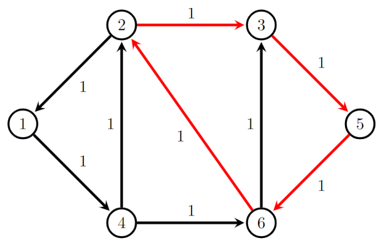
Jordi Velthuizen

To obtain the degree of Bachelor of Science
at the Delft Univeristy of Technology,
To be defended on June 21 2024 at 10:30.

Student number:	5485940	
Project duration:	April 15, 2024-June 21, 2024	
Thesis committee:	Dr. C.E. (Carla) Groenland	TU Delft, supervisor
	Dr. N.D. (Nikolaas) Verhulst	TU Delft, supervisor
	Dr. ir. W.G.M. (Wolter) Groenevelt	TU Delft

Lay summary

In graph theory, mathematicians have been exploring cycles, essentially loops within a network, for well over 50 years. These cycles are used in various real-world applications such as research on ecosystems, network theory for computer science, and traffic flow within logistics. A specific subgroup of these cycles are the cycles where the length of the cycle is even. A visual representation of such a cycle can be found in Figure 1 (a). Finding an algorithm that discovers the shortest one of these even cycles in a given network, especially when one-way streets are allowed, has been a longstanding problem in graph theory.



(a) Visual

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(b) Matrix

Figure 1: Two representations of the same graph in which an even cycle, indicated by the color red, is depicted.

In the meantime, numerous other advancements were made such that the problem of finding an algorithm for these shortest even cycles could be reframed as another problem, finding two special numbers that could be extracted from its matrix representation. In Figure 1, both a visual representation of a graph (a) and its matrix representation (b) can be found.

These numbers, called the permanent and determinant, while solvable in some scenarios, appeared to be much harder to solve in the scenario where the shortest even cycle had to be found. Therefore, the challenge remained.

Until recently, when Björklund, Husfeldt, and Kaski [1] discovered an algorithm that could find both the permanent and the determinant for this specific case. Their approach involves constructing two mathematical concepts that help in finding the permanent and determinant.

The focus of this thesis is to explain these mathematical concepts in a way that is accessible to bachelor students of mathematics. These concepts are then applied to explain the algorithm that finds the determinant. Additionally, the algorithm for computing the determinant, along with one that does computations with one of the mathematical objects, is implemented in Python. By explaining this object and the algorithm for the determinant, readers will gain a small insight into the concepts Björklund, Husfeldt, and Kaski [1] used to find an algorithm for the shortest even cycle in graphs.

Abstract

The problem of finding even cycles efficiently in a directed graph can be rewritten to a problem of finding the permanent and determinant of the graph's adjacency matrix. Although this has been known for a long time, it did not solve the shortest even cycle problem, as mathematicians lacked an efficient method to compute the permanent.

In 1979 Valiant [2] found out that in some specific cases, when one works modulo a power of 2, the permanent could be computed efficiently. Although this did not directly lead to a breakthrough in finding even cycles, it laid the groundwork for Björklund, Husfeldt and Kaski [1] to solve the problem of finding the shortest even cycle. Instead of working with the integers modulo 2, Björklund, Husfeldt and Kaski [1] used a polynomial quotient ring $\mathbb{E}_{4^d}$, which is an extension of some polynomial quotient ring $\mathbb{F}_{2^d}$. The polynomial quotient ring $\mathbb{E}_{4^d}$, it turns out, benefits from having a characteristic of four and relies on $\mathbb{F}_{2^d}$ being a field. Surprisingly, these two properties are exactly what is needed to efficiently compute both the permanent and determinant.

The aim of this thesis is to give the reader a better understanding of the polynomial quotient rings $\mathbb{F}_{2^d}$ and $\mathbb{E}_{4^d}$ that are used by Björklund, Husfeldt and Kaski in [1]. These structures are then used to explain the part of their algorithm that computes the determinant of matrices with entries in $\mathbb{E}_{4^d}$. The reader is given a deeper insight into the algorithm by comparing it to the standard algorithm for computing the determinant learned in most linear algebra classes. By using the polynomial rings and algorithm for the determinant described in [1], a Python script is written. This script can perform calculations within $\mathbb{E}_{4^d}$ and compute the determinant for a matrix with entries in $\mathbb{E}_{4^d}$. Finally, the difference between computing the permanent and the determinant will be briefly explained.

Contents

1 Introduction

Graph theory, originally a pure math subject, has become useful in many real-life scenarios involving networks. Within this field, some mathematicians have focused on finding cycles, which, just as graphs, have many practical applications. Environmental science uses them to understand ecosystems and predict climate changes, in transportation and logistics, cycles help optimize traffic flow and supply chain management. Other fields that benefit from understanding cycles include biology, electrical engineering, computer science, economics and robotics. While not all of these applications always require the shortest even cycle, some specific problems may benefit from tailored solutions involving shortest even cycles.

Whether for its practical applications or the challenge of an unsolved problem, researchers have been actively studying even cycles in both directed and undirected graphs for over 50 years. Consequently, numerous advancements have been made leaving most even cycle problems solved. By the end of the twentieth century, Yuster and Zwick [3] found an algorithm that could find the shortest even cycle in undirected graphs. A mere two years later, Robertson, Seymour and Thomas [4] came up with a way to find out if these even cycles exist in some given directed graph.

An algorithm to find the shortest even cycle in a directed graph proved to be more difficult to find. Only recently, in 2022, such an algorithm was discovered by Björklund, Husfeldt and Kaski [1].

The discovered algorithm heavily relies on computing both the permanent and the determinant. Although efficient algorithms exist for the determinant, calculating the permanent is notoriously harder. It turns out that the permanent of a matrix with entries of some specific polynomial quotient ring, called $\mathbb{E}_{4^d}$, as explained in Section 4, can be found in polynomial time. Part of this algorithm also finds the determinant, which is covered in Section 6. Explaining the algorithm that computes the determinant is the main goal of this thesis and hopefully, gives the reader a slight understanding in how the permanent is computed in [1]. Both the algorithm for finding the determinant of a matrix with entries in $\mathbb{E}_{4^d}$ and the implementation of the ring $\mathbb{E}_{4^d}$ with its operations are also programmed using Python, as detailed in Sections 5 and 7.

2 The problem

Before diving into the main part of this thesis, which focuses on algebraic structures, we will briefly review some definitions in graph theory. This will help the reader understand what Björklund, Husfeldt and Kaski [1] were investigating in graphs and how they addressed the problem.

Definition 2.1. *(Graph).* A graph G is a pair (V, E), where V is a non-empty set of vertices and E is a set of pairs $\{u, v\} : u, v \in V$ also called edges. Whenever the pairs $\{u, v\} \in E$ are unordered we call G an undirected graph. When the pairs $(u, v) \in E$ are ordered it is called a directed graph.

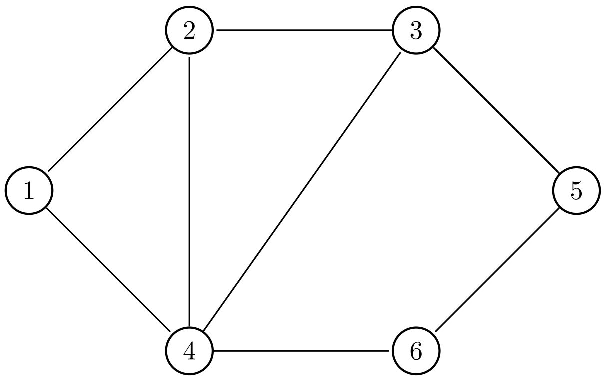


Figure 2: A representation of an undirected graph is depicted.

The edges $e \in E$ in a graph G connect its vertices $v \in V$. In undirected graphs, these connections are two-way streets: if vertex $u \in V$ has an edge to vertex $v \in V$, then $v \in V$ also has an edge to $u \in V$. Directed graphs, on the contrary, permit one-way connections. Hence, the existence of an edge from vertex u to v does not imply the existence of an edge from v to u.

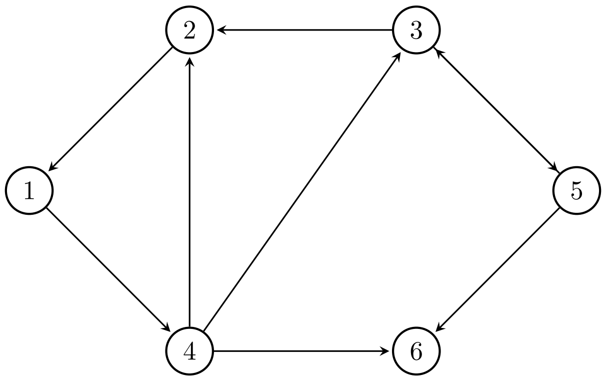


Figure 3: A representation of a directed graph is depicted

Figure 3 illustrates a directed graph, where the edges are depicted with arrows indicating the direction of traversal. For instance, in this graph, one can directly traverse from vertex 2 to vertex 1, but not vice versa.

Definition 2.2. *(Walk).* Let $G = (V, E)$ be a graph. A walk W on G is a sequence of vertices $v_1, v_2, v_3, \ldots$ such that $\{v_i, v_{i+1}\} \in E$. Whenever the walk starts and ends on the same vertex v_1, it is called a closed walk.

In a directed graph, the existence of a walk from node n_i to n_j does not imply that a walk from n_j to n_i exists. Whenever a walk can be traversed in each direction a closed walk could be constructed.

Definition 2.3. *(Cycle).* Let $G = (V, E)$ be a graph. A cycle C on graph G is a closed walk where the first and therefore last vertex is visited twice while the other vertices are visited at most once.

Figure 4 shows a graph where a closed walk is drawn. This closed walk is not a cycle since it visits vertex 2 multiple times.

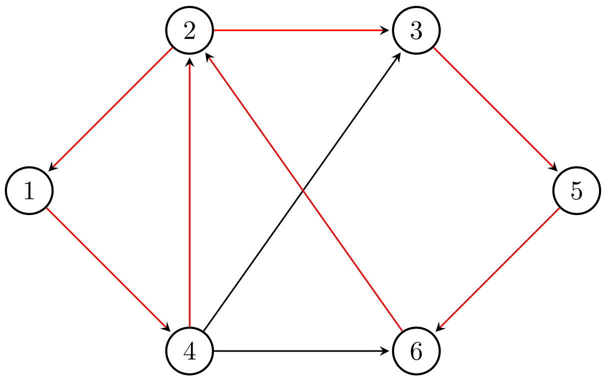


Figure 4: A representation of a closed walk which is not a cycle.

Definition 2.4. *(Even cycle).* Let $G = (V, E)$ be a graph. An even cycle C in graph G is a cycle that visits an even number of distinct vertices.

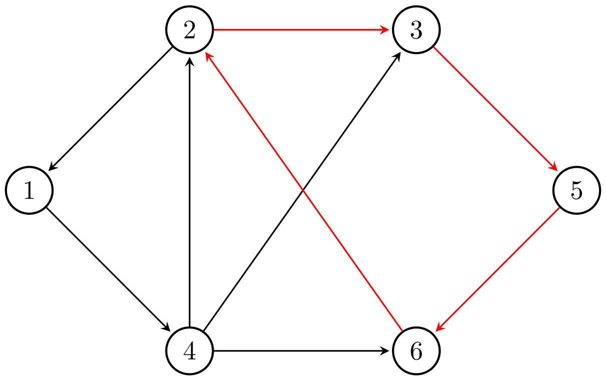


Figure 5: An even cycle of length 4.

Figure 5 shows a cycle which is contained in the closed walk in Figure 4. Furthermore, Figure 5 depicts an even cycle since it contains four vertices and four edges. Note that if a cycle contains four edges it also contains four vertices and vice versa.

Traditional methods efficiently find even cycles in undirected graphs. This thesis, however, explores an algorithm detailed in [1], which identifies even cycles within directed graphs.

In directed graph theory, there is an established algorithm for finding closed walks of specific lengths, which can be used to identify the shortest odd cycle within a graph. Sometimes, closed walks can be divided into cycles, as shown in Figures 4 and 5. A closed walk of odd length falls into one of two categories: it is a cycle or it can split into odd and even cycles. Therefore, searching for the shortest closed walk with an odd length naturally leads to discovering the shortest odd cycle.

However, this method fails when searching for an even cycle, as a closed walk of even length might split into two odd cycles. Hence, finding even cycles requires a different approach, which involves transforming the problem into finding the permanent and determinant of a yet to be defined adjacency matrix.

Definition 2.5. *(Weighted directed graph).* Let $G = (V, E)$ be a directed graph. Whenever G is equipped with a function $\omega : E \to \mathbb{R}$ to create a tuple $G' = (V, E, \omega)$ we call G' a weighted directed graph.

The function ω in weighted directed graphs is a function that assigns a weight to each edge in the graph. This weight can be seen as the length of an edge, where a high weight represents a longer edge. Whenever graphs represent real-life scenarios, it makes sense to assign weights to edges. For example, a road network can be thought of as a graph where the weight of each edge represents the traversal time between intersections, which are represented by nodes.

Definition 2.6. *(Self loop).* A self loop is an edge from a vertex to itself.

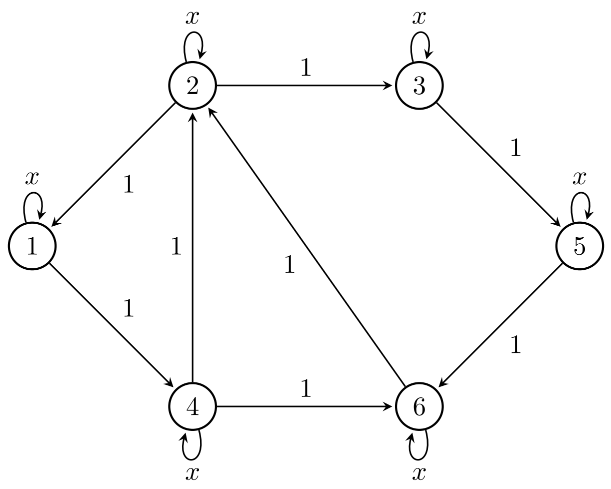


Figure 6: A weighted directed graph is depicted.

In Figure 6 it can be seen that each vertex is equipped with a self loop of weight x. Although weight x seems to be a peculiar choice as weight for an edge, it is a necessity for Theorem 2.1 to work.

Definition 2.7. *(Adjacency matrix).* Let $G = (V, E)$ be a graph with n vertices. The adjacency matrix A is created as follows:

$$A_{i,j} = \begin{cases} w_{i,j}, & \text{if } \{i,j\} \in E \\ 0, & \text{otherwise} \end{cases}$$

Example 2.1. Let G be the weighted directed graph depicted in Figure 6. Then its adjacency matrix A is:

$$\begin{bmatrix} x & 0 & 0 & 1 & 0 & 0 \\ 1 & x & 1 & 0 & 0 & 0 \\ 0 & 0 & x & 0 & 1 & 0 \\ 0 & 1 & 0 & x & 0 & 1 \\ 0 & 0 & 0 & 0 & x & 1 \\ 0 & 1 & 0 & 0 & 0 & x \end{bmatrix}.$$

The following theorem transforms the problem of finding even cycles into a problem that involves computing both the determinant and the permanent. The permanent and determinant will be defined in Section 6. In this thesis, we will discuss methods for computing the determinant and briefly address how the permanent is computed in [1].

Theorem 2.1. *Let G be a weighted directed graph with n vertices and a self loop at each vertex of unspecified weight x. Then for the adjacency matrix $A(x)$ of G,*

$$\text{per}(A(x)) - \det(A(x)) = f(x)$$

for some polynomial $f(x) = a_0 + a_1x + ... + a_nx^n$. Moreover, let us write $[x^\ell]_{f(x)}$ for the coefficient for x^ℓ in $f(x)$ and n for the number of vertices in graph G. Then the number of edges used in the shortest even cycle is equal to the smallest positive even k such that $[x^{n-k}]_{f(x)}$ is not identically zero.

The proof is given in [1, p.8].

Theorem 2.1 finds the cycle with the least number of edges such that an even number of edges are traversed. However, this algorithm can also be applied to a similar problem in directed weighted graphs. In these cases, a cycle must be found with the smallest summed weight over the edges used while also having an even total weight. By constructing a second graph that resembles the original but with a few modifications, the same theorem can be used to find the length of the shortest cycle with an even weight. While the construction of this second graph varies depending on the situation, a small example will be shown to provide insight into some of the possible concepts that can be used.

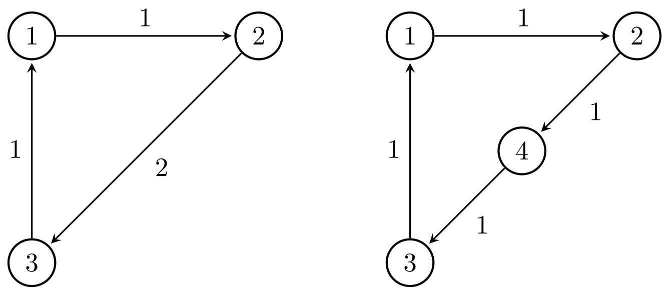


Figure 7: A graph and its adaptation such that Theorem 2.1 can still be used is shown.

In Figure 7 it is shown that Theorem 2.1 can still be used on graphs where not every edge has a weight of one. This is done by replacing the edge between vertices two and three with two edges and an intermediate node, so that the new edges each have a length of one and their combined length is equal to the original edge. Although similar techniques can be applied when edges have non-integer weights, additional questions arise. For instance, how would one categorize a cycle with a weight of $\frac{5}{2}$? Would that be considered an even or an odd cycle? For this thesis, we will primarily focus on computing the determinant and leave these questions for the reader to ponder.

3 Algebraic structures

3.1 Rings and fields

The key advancement in [1] lies in the creation of two algebraic structures which are called polynomial quotient rings. These polynomial quotient rings will be defined at the end of this section. Before inspecting polynomial quotient rings, we will lay the groundwork by exploring definitions and examples of rings. These definitions will then be used to define and explain the more complicated polynomial rings and polynomial quotient rings.

Definition 3.1. *(Ring).* A ring is a set R equipped with two operations, $+$ (addition) and $\cdot$ (multiplication) satisfying the first three of the following axioms, called the ring axioms. The ring R is called commutative whenever axiom four is also satisfied.

1. R is an abelian group under addition.
   - $a + (b + c) = (a + b) + c$ for all $a, b, c \in R$ (associativity).
   - $a + b = b + a$ for all $a, b \in R$ (commutativity).
   - There exists an element $0 \in R$ such that $a + 0 = a$ for all $a \in R$ (additive identity).
   - For each $a \in R$ there exists an inverse $(-a) \in R$ such that $a + (-a) = 0$ (additive inverse).
2. R is a monoid under multiplication.
   - $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ for all $a, b, c \in R$.
   - There exists an element $1 \in R$ such that $a \cdot 1 = a$ and $1 \cdot a = a$ for all $a \in R$ (multiplicative identity).
3. Multiplication is distributive with respect to addition.
   - $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ for all $a, b, c \in R$.
   - $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$ for all $a, b, c \in R$.
4. R is commutative.
   - $a \cdot b = b \cdot a$ for all $a, b \in R$.

For the keen reader, it becomes apparent that a ring functions as a group under addition while also featuring a secondary operation, namely multiplication.

Example 3.1. $\mathbb{Z}$ is a commutative ring.

It is worth noting that although a ring operates as a group under addition, it does not contain multiplicative inverses for all of its elements. Hence, a ring does not necessarily qualify as a group under multiplication.

Definition 3.2. *(Ring homomorphism).*
Let R_1, R_2 be rings. A map $M : R_1 \to R_2$ is called a ring homomorphism if the following conditions hold:

- $M(a+b) = M(a) + M(b)$ for all $a, b \in R_1$.
- $M(a \cdot b) = M(a) \cdot M(b)$ for all $a, b \in R_1$.
- $M(1) = 1$, where 1 denotes the multiplicative identity element of the groups R_1 and R_2.

Definition 3.3. *(Field).* A field is a ring F satisfying:

- For each $a \in F : a \neq 0$ there exists an inverse $a^{-1} \in R$ such that $a \cdot a^{-1} = 1$ (multiplicative inverse).

In contrast to a ring, a field is a group under multiplication, provided that zero is excluded from the multiplicative group due to its lack of an inverse.

Definition 3.4. *(Isomorphism).* Let R_1, R_2 be rings and $M : R_1 \to R_2$ be a bijective ring homomorphism. Then M is called an isomorphism.

Lemma 3.1. *Let F be a field with q elements. Then F is the only finite field with q elements up to isomorphism. That is, if another field F' with q elements exists, then there exists an isomorphism M from F to F'.*

Proof. Cf. e.g. [5, p.14]. $\square$

Elements and subsets of two fields that are isomorphic share the same properties. Whenever two fields are isomorphic they can be roughly seen as the same mathematical object with a different representation.

Example 3.2. The commutative ring $\mathbb{Z}$ is not a field since not all non-zero elements have an inverse. All non-zero elements $a \in \mathbb{Q}$ do have a multiplicative inverse which can be written as $\frac{1}{a}$, thus $\mathbb{Q}$ is a field.

Definition 3.5. *($\mathbb{Z}_i$).* The ring $\mathbb{Z}_i$ consists of all the elements $0, 1, 2, ..., i-1$ which are representatives of the equivalence classes when divided with remainder by i. The two numbers n, m are said to be equivalent whenever they share the same remainder when divided by i. When two numbers k and l are equivalent we write $k \equiv l$.

Example 3.3. In $\mathbb{Z}_2$ $1 \equiv 3$ since dividing 3 by 2 gives us the remainder 1.

Example 3.4. In $\mathbb{Z}_4$ $3 \equiv 15$ since dividing 15 by 4 gives us the remainder 3.

Both $\mathbb{Z}_2$ and $\mathbb{Z}_4$ play significant roles in [1] and consequently in this thesis. While they share a similar construction, they are fundamentally different as can be seen in Example 3.5.

Example 3.5. Let $2 \in \mathbb{Z}_4$. Then $2^2 \equiv 4 \equiv 0$.

Example 3.5 illustrates a scenario in which the element 2 possesses another element (2 itself), such that their multiplication results in zero. These elements, known as zero divisors, signify that $\mathbb{Z}_4$ lacks the property of being a field, unlike $\mathbb{Z}_2$.

3.2 Polynomial rings

We are now ready to explore polynomial rings and polynomial quotient rings.

Definition 3.6. *(Polynomial ring).* Let R be a ring. Then the polynomial ring $R[x]$ over R consists of the polynomials $f(x) = \sum_{i=0}^{n} a_i x^i$ where $a_i \in R$ and n the highest non-zero coefficient. The operators addition and multiplication are defined as follows:

- $\sum_{i=0}^{n} a_i x^i + \sum_{i=0}^{m} b_i x^i = \sum_{i=0}^{\max(n,m)} (a_i + b_i) x^i$ (addition),
- $(\sum_{i=0}^{n} a_i x^i) \cdot (\sum_{i=0}^{m} b_i x^i) = \sum_{i=0}^{nm} x^i \sum_{j=0}^{i} (a_j \cdot b_{i-j})$ (multiplication).

The multiplication and addition of the coefficients happen according to the rules of the ring R.

Example 3.6. $\mathbb{Z}_2[x]$ a polynomial ring with coefficients in $\mathbb{Z}_2$. One of these polynomials is $1 + x$.

In $\mathbb{Z}_2[x]$, coefficients are restricted to only 0 or 1. Any other coefficient can be represented as a polynomial with coefficients of 0 and 1.

Example 3.7. In $\mathbb{Z}_4[x]$, $1 + 4x + 6x^2$ equals $1 + 2x^2$, since $4 \equiv 0$ and $6 \equiv 2$ in $\mathbb{Z}_4$.

Definition 3.7. *(Reducible polynomial).* A polynomial $f(x) \in F[x]$ with coefficients in a field F is called reducible over F if it can be factored into two non-constant polynomials taking coefficients in F, that is $f(x) = g(x)h(x)$ for some polynomials $g(x), h(x) \in F[x]$. Whenever a polynomial is not reducible, it is called irreducible.

Lemma 3.2. *The polynomial $f(x) \in F[x]$ is irreducible over a finite field F if and only if there are no elements $y \in F$ such that $f(y) \equiv 0$.*

Proof. Cf. e.g. [6, p.127]. □

Example 3.8. $f(x) = x + x^2$ is a reducible polynomial in $\mathbb{Z}_2[x]$ since $f(1) \equiv 0$ and $f(x) = x \cdot (1 + x) = g(x)h(x)$, where $g(x) = x$ and $h(x) = (1 + x)$. Also note that $f(1) = f(0) = 0$.

Irreducible polynomials share many similarities with prime numbers. For that reason they are often referred to as prime polynomials. As with prime numbers each polynomial has a unique factorization in irreducible polynomials up to invertible elements. Although not of importance for this thesis an intrigued reader can find the proof in [6].

Example 3.9. Let $x^2 + x + 1 = f(x) \in \mathbb{Z}_2[x]$ be a polynomial then $f(x)$ is an irreducible polynomial since it is not possible to write $f(x) = g(x) \cdot h(x)$ with $g(x), h(x) \in \mathbb{Z}_2[x]$.

Bear in mind that an irreducible polynomial over a certain polynomial ring R might be reducible in another polynomial ring.

Definition 3.8. *(Monic polynomial).* A monic polynomial is a polynomial with the highest non zero coefficient equal to 1.

Example 3.10. $1 + x + 2x^2$ is not a monic polynomial but $1 + x + x^2$ is.

Proposition 3.1. *Let $R[x]$ be a polynomial ring. Then given a monic polynomial $g(x) \in R[x]$, every polynomial $f(x) \in R[x]$ may be expressed as $f(x) = g(x)q(x) + r(x)$ for some $q(x), r(x) \in R[x]$, where $deg(r(x)) < deg(g(x))$. The polynomial $r(x)$ is called the remainder and $q(x)$ is called the quotient. Moreover these $q(x)$ and $r(x)$ are unique.*

Proof. Cf. e.g. [6, p.121]. □

The above proposition can be loosely interpreted as division with remainder using numbers in $\mathbb{Z}$. The uniqueness of $q(x)$ and $r(x)$, as the reader will find out is rather important to create some of the mathematical structures that are used to find the shortest even cycle.

Definition 3.9. *(Polynomial quotient rings).* Let $Q \subset R[x]$ be the set of all the remainders when divided by some polynomial $g(x) \in R[x]$. Then the set Q combined with the operations $+$ (addition) and $\cdot$ (multiplication) form the polynomial quotient ring $R[x]/\langle g(x) \rangle$.

In these polynomial quotient rings, addition and multiplication operate similarly to regular polynomials. Both operations are associative and commutative. Additionally, when $g(x)$ is selected as an irreducible polynomial, no elements $f(x)$ and $h(x)$ exist such that $g(x) = f(x) \cdot h(x)$. Moreover, if $g(x)$ is of degree d, then the polynomial quotient ring with $g(x)$ as the divisor contains $|R|^d$ elements, where $|R|$ represents the number of elements in the underlying ring of coefficients.

Example 3.11. Let $g(x) = 1 + x^3 + x^4, f(x) = x + x^4 + x^5 \in \mathbb{Z}_2[x]$ and let $R = \mathbb{Z}_2[x]/\langle g(x) \rangle$ be a polynomial quotient ring. Since $f(x) = x \cdot g(x) + r(x)$, where $r(x) = 0$, we have that $f(x) = x + x^4 + x^5 \equiv 0$ in R.

Example 3.12. Let $g(x) = 1 + x^3, f(x) = 1 + x + x^4 \in \mathbb{Z}_2[x]$ and $\mathbb{Z}_2[x]/\langle g(x)\rangle$ be a polynomial quotient ring. Since $f(x) = x \cdot g(x) + r(x)$, where $r(x) = 1$. We have that $f(x) = 1 + x + x^4 \equiv 1$.

Theorem 3.1. *Let F be a field and $f(x) \in F[x]$ be an irreducible polynomial. Then the quotient ring $F[x]/\langle f(x)\rangle$ forms a field.*

Proof. Cf. e.g. [6, p.128]. $\square$

Theorem 3.1 tells us that, for every irreducible polynomial $g(x) \in \mathbb{Z}_2[x]$ the quotient ring $\mathbb{Z}_2[x]/\langle g(x)\rangle$ forms a field. This might not be the case in $\mathbb{Z}_4[x]$.

4 The construction of $\mathbb{F}_{2^d}$ and $\mathbb{E}_{4^d}$

We are now prepared to introduce two fundamental algebraic structures: the polynomial ring $\mathbb{F}_{2^d}$ and its extension $\mathbb{E}_{4^d}$. These structures are the foundation of the algorithm described in [1] where they are used to calculate the determinant and permanent.

4.1 Defining $\mathbb{F}_{2^d}$

We define $\mathbb{F}_{2^d}$ as the finite field constructed over the binary field $\mathbb{Z}_2$, represented as $\mathbb{F}_{2^d} = \mathbb{Z}_2[x]/\langle g(x)\rangle$, where $g(x)$ denotes an irreducible polynomial of degree d. This construction creates a polynomial quotient ring where elements are equivalence classes of remainders of polynomials divided by $g(x)$. Note that $\mathbb{F}_{2^d}$ is indeed a finite field according to Theorem 3.1.

The choice of the irreducible polynomial $g(x)$ is arbitrary, as any irreducible polynomial of degree d generates a field with 2^d elements. Using Lemma 3.1 we see that all fields with 2^d elements are isomorphic.

4.2 The lift

To extend $\mathbb{F}_{2^d}$ to $\mathbb{E}_{4^d}$, a certain operation is needed which will be called the lift. This operation will first be discussed as an operation from $\mathbb{Z}_2[x]$ to $\mathbb{Z}_4[x]$ and from there will be defined as a function from $\mathbb{F}_{2^d}$ to $\mathbb{E}_{4^d}$.

Definition 4.1. *(The lift).* Given a polynomial $f(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n \in \mathbb{Z}_2[x]$ the lift $\mathcal{L}[f(x)]$: $\mathbb{Z}_2[x] \to \mathbb{Z}_4[x]$ maps all coefficients $a_i \in \mathbb{Z}_2$ to $\mathbb{Z}_4$ such that $0 \in \mathbb{Z}_2 \mapsto 0 \in \mathbb{Z}_4$ and $1 \in \mathbb{Z}_2 \mapsto 1 \in \mathbb{Z}_4$. This creates the polynomial $\overline{f(x)} = \overline{a_0} + \overline{a_1}x + \overline{a_2}x^2 + ... + \overline{a_n}x^n \in \mathbb{Z}_4[x]$

From now on we will say that $\overline{f(x)}$ is the lift of $f(x)$.

Example 4.1. Let $f(x) = 1 + x^2 + x^3 \in \mathbb{Z}_2[x]$. Then $\mathcal{L}[f(x)] = 1 + x^2 + x^3 = \overline{f(x)} \in \mathbb{Z}_4[x]$.

Note that the polynomial might not visibly change but the created polynomial is an element of a different ring and therefore a different polynomial.

Now, using the definition of the lift and the earlier defined polynomial quotient ring $\mathbb{F}_{2^d}$, an extension can be formed which we will call $\mathbb{E}_{4^d}$.

Definition 4.2. *($\mathbb{E}_{4^d}$).* Let $\mathbb{F}_{2^d} = \mathbb{Z}_2[x]/\langle g_2(x)\rangle$ for some irreducible $g_2(x) \in \mathbb{Z}_2[x]$ of degree d and $g_4(x) = \overline{g_2(x)}$ be the lift of $g_2(x)$. Then $\mathbb{E}_{4^d}$ is defined as $\mathbb{Z}_4[x]/\langle g_4(x)\rangle$.

4.2.1 The lift from $\mathbb{F}_{2^d}$ to $\mathbb{E}_{4^d}$

Now that we have defined both $\mathbb{F}_{2^d}$ and $\mathbb{E}_{4^d}$ it is time for a function that maps elements from $\mathbb{F}_{2^d}$ to $\mathbb{E}_{4^d}$. In Example 4.2 it becomes clear that using the lift on the polynomial quotient rings $\mathbb{F}_{2^d}$ and $\mathbb{E}_{4^d}$ can lead to some complications. Therefore, some adaptions to the lift must be made.

Example 4.2. Let $g(x) = 1+x+x^2+x^3+x^4 \in \mathbb{Z}_2[x]$ be an irreducible polynomial of degree 4 such that we can construct $\mathbb{F}_{2^4}$. Now take $f_1(x) = 1+x+x^2+x^3 \in \mathbb{Z}_2[x]$ and $f_2(x) = x^4 \in \mathbb{Z}_2[x]$ such that $f_1(x) \equiv f_2(x) \in \mathbb{F}_{2^4}$. By taking the lift of both polynomials $f_1(x)$ and $f_2(x)$ to obtain $\overline{f_1(x)} = 1+x+x^2+x^3 \in \mathbb{Z}_4[x]$ and $\overline{f_2(x)} = x^4 \in \mathbb{Z}_4[x]$ it can be shown that $\overline{f_1(x)} = 1+x+x^2+x^3 \neq 3+3x+3x^2+3x^3 \equiv x^4 = \overline{f_1(x)} \in \mathbb{E}_{4^d}$.

It would be desirable that all elements of the same equivalence class have the same lift. With a naive definition for the lift, this is clearly not the case, as can be seen in Example 4.2. Luckily there seems to be an easy solution by defining the lift only on the remainders of each polynomial in the quotient ring. A polynomial that is not equal to its remainder will first be reduced to its remainder and only then the lift operator can take place.

Definition 4.3. *(The lift from $\mathbb{F}_{2^d}$ to $\mathbb{E}_{4^d}$).* Let $g(x) \in \mathbb{Z}_2[x]$ be an irreducible polynomial of degree d that constructs $\mathbb{F}_{2^d}$ such that $\overline{g(x)}$ constructs $\mathbb{E}_{4^d}$.

Let $\mathcal{H}_2[f(x)] : \mathbb{F}_{2^d} \to \mathbb{Z}_2[x]$ map $f(x)$ to its remainder $r(x)$ in $\mathbb{Z}_2[x]$ when divided by $g(x)$. Secondly, we define $\mathcal{H}_4[f(x)] : \mathbb{Z}_4[x] \to \mathbb{E}_{4^d}$ to map $f(x)$ to its remainder $r(x)$ when divided by $\overline{g(x)}$. These remainders are unique according to Proposition 3.1.

Now define the lift on polynomial quotient rings $\mathcal{L}^*[f(x)] : \mathbb{F}_{2^d} \to \mathbb{E}_{4^d}$ for all $f(x) \in \mathbb{F}_{2^d}$ by

$$\mathcal{L}^*[f(x)] = \mathcal{H}_4 \circ \mathcal{L} \circ \mathcal{H}_2[f(x)].$$

4.3 The projection

To properly link $\mathbb{F}_{2^d}$ and $\mathbb{E}_{4^d}$ a function from $\mathbb{E}_{4^d}$ to $\mathbb{F}_{2^d}$ is needed. This function will be called the projection and will first be discussed as an operator from $\mathbb{Z}_4[x]$ to $\mathbb{Z}_2[x]$ before extending it to the quotient rings $\mathbb{F}_{2^d}$ and $\mathbb{E}_{4^d}$.

Definition 4.4. *(The projection).* Given a polynomial $f(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n \in \mathbb{Z}_4$ the projection $\mathcal{P}[f(x)] : \mathbb{Z}_4 \to \mathbb{Z}_2$ maps all coefficients $a_i \in \mathbb{Z}_4$ to the remainder of division by 2 in $\mathbb{Z}_2$. This creates the polynomial $\underline{f(x)} = a_0^* + a_1^*x + a_2^*x^2 + ... + a_n^*x^n \in \mathbb{Z}_2$, where $a_i^* \equiv a_i \pmod 2$.

From now on we will say that $\underline{f(x)}$ is the projection of $f(x)$.

Example 4.3. Let $f(x) = 1+3x^2+2x^3 \in \mathbb{E}_{4^d}$ for some $d > 3$. Then $\mathcal{P}[f(x)] = 1 + x^2 = \underline{f(x)} \in \mathbb{F}_{2^d}$. The coefficient for x^3 gets mapped to 0 since $0 \equiv 2 \pmod 2$

Lemma 4.1. *The projection inverts the lift.*

Proof. Let $f(x) = a_0+a_1x+a_2x^2+...+a_nx^n \in \mathbb{Z}_2$ then $\overline{f(x)} = a_0+a_1x+a_2x^2+...+a_nx^n \in \mathbb{Z}_4$. Since all coefficients are either zero or one we can conclude that $\underline{\overline{f(x)}} = a_0 + a_1x + a_2x^2 + ... + a_nx^n = f(x) \in \mathbb{Z}_2$. □

In the following example it is shown that the lift does not invert the projections and some information might be lost during the projection.

Example 4.4. Let $f(x) = 1 + 3x^2 \in \mathbb{Z}_4[x]$. Taking first the projection and then the lift gives us $g(x) = 1 + x^2 \in \mathbb{Z}_4[x]$. Thus giving a different polynomial than our starting polynomial.

Lemma 4.2. *The projection $\mathbb{Z}_4[x] \to \mathbb{Z}_2[x]$ is a ring homomorphism*

Proof. Let $f(x), h(x) \in \mathbb{Z}_4[x]$ and lets denote

$$f(x) = \sum_{i=0}^{n} a_i x^i, \quad h(x) = \sum_{j=0}^{m} b_j x^j,$$

$$L = \max(n, m),$$

where $a_i = 0, i > n,\ b_j = 0, j > m$ and $c^* \equiv c$ (*modulo* 2). Then

$$\underline{f(x) + h(x)} = \sum_{i=0}^{L} (a_i + b_i)^* x^i = \sum_{i=0}^{n} a_i^* x^i + \sum_{j=0}^{m} b_j^* x^j = \underline{f(x)} + \underline{h(x)}$$

$$\underline{f(x) \cdot h(x)} = \sum_{i=0}^{L} \sum_{j=0}^{i} (a_{i-j} b_j)^* x^i = \sum_{i=0}^{L} \sum_{j=0}^{i} a_{i-j}^* b_j^* x^i = \underline{f(x)} \cdot \underline{h(x)}$$

$$\underline{1} \equiv 1 \ (\textit{modulo } 2) \equiv 1,$$

proving that the projection defined on $\mathbb{Z}_4[x]$ to $\mathbb{Z}_2[x]$ is a ring homomorphism. □

The proof of Lemma 4.3 is strongly based on a proof found in [1].

Lemma 4.3. *The projection $\mathcal{P}[f(x)] \ : \ \mathbb{E}_{4^d} \to \mathbb{F}_{2^d}$ is well defined.*

Proof. Let $s, s' \in \mathbb{Z}_4[x]$ be two polynomials such that $s - s' = g_4(x)h(x)$, that is, s and s' share the same remainder when divided by $g_4(x)$. Since the projection is a ring homomorphism it follows that

$$\underline{s - s'} = \underline{g_4(x)h(x)} = \underline{g_4(x)} * \underline{h(x)} = g_2(x) * \underline{h(x)}$$

In other words if s and s' share the same remainder before the projection they also share the same remainder after the projection thus making them members of the same equivalence class and the projection well defined. □

5 Implementing $\mathbb{E}_{4^d}$ in Python

Performing calculations within the constructed quotient ring $\mathbb{E}_{4^d}$ for small d can be manageable by hand, but it quickly becomes impractical as d grows larger. This scalability issue becomes particularly apparent when tasked with identifying even cycles within larger graphs.

In order to gain insight into the behavior of the constructed quotient ring $\mathbb{E}_{4^d}$ and how these rings could be implemented within a computer, a simple script in Python was developed. This script enables basic arithmetic operations such as addition, multiplication and polynomial reduction.

5.1 Overview of the code

The script comprises three classes: Polynomial, QuotientRing and E4d. The class Polynomial represents polynomials in $\mathbb{Z}_4[x]$, storing their coefficients and defining operations such as addition, multiplication and subtraction. The class QuotientRing extends this functionality to operate within a quotient ring $\mathbb{Z}_4[x]/\langle g(x)\rangle$, where $g(x) \in \mathbb{Z}_4[x]$ is given as input whenever the object is created. Additional to Polynomial, it includes a method for reducing polynomials to their remainders within the quotient ring. Finally the class E4d extends the class QuotientRing to a polynomial in $\mathbb{E}_{4^d}$ using as input the lift of an irreducible polynomial in $\mathbb{F}_{2^d}$.

Polynomials in different quotient rings and different constructions of E4d, defined by distinct divisors, cannot be combined or operated on together because they have different structures.

5.2 Finding an irreducible polynomial of degree d

The script supports the creation of a polynomial quotient ring $\mathbb{Z}_4[x]/\langle g_4(x)\rangle$, where the divisor $g_4(x)$ is specified as an input to the class. However, to construct the quotient ring $\mathbb{E}_{4^d}$ for a given d, an irreducible polynomial $g_2(x) \in \mathbb{Z}_2[x]$ of degree d must be identified. The polynomial $g_4(x)$ can then be obtained by taking the lift of $g_2(x)$.

The process of finding such an irreducible polynomial of a specified degree can be achieved in $\mathcal{O}(d^2)$ time using an algorithm outlined in [7]. This becomes negligible compared to the primary algorithm's complexity in [1] of $\mathcal{O}(n^{3+\omega})$. Here, ω denotes the exponent of square matrix multiplication.

In this case, finding an irreducible polynomial is even easier since it must only be irreducible over $\mathbb{Z}_2$. According to Lemma 3.2, a polynomial $f(x)$ including 1 and an even number of non-zero terms is irreducible because in that case, both $f(0)$ and $f(1)$ are equal to 1. The chosen irreducible polynomial is $g(x) = 1 + x + x^2 + x^3 + x^4$.

6 An efficient algorithm for the determinant

6.1 The determinant visually

Definition 6.1. *(The determinant).*
The determinant of an $n \times n$ matrix is equal to

$$\det(M) = \sum_{\sigma \in S_n} \text{sign}(\sigma) \cdot M_{1,\sigma(1)} \cdot M_{2,\sigma(2)} \cdot ... \cdot M_{n,\sigma(n)}$$

The equation presented above may appear complex at first glance and could benefit from a more intuitive visual explanation.

$$M = \begin{bmatrix} 1 & 3 & 1 & 7 & 8 \\ 2 & 0 & 5 & 1 & 0 \\ 9 & 4 & 1 & 6 & 2 \\ 9 & 3 & 2 & 1 & 2 \\ 1 & 2 & 2 & 6 & 7 \end{bmatrix} \tag{1}$$

Considering the matrix M defined in equation (1), its determinant represents the sum of products derived from selecting one element from each row and each column, with no repetition. An illustrative example of such a combination is depicted in equation (2), where the contribution of this specific combination to the determinant is $\text{sign}(\sigma) \cdot 1 \cdot 4 \cdot 5 \cdot 6 \cdot 2 = \text{sign}(\sigma) \cdot 240$.

$$M' = \begin{bmatrix} \textcircled{1} & 3 & 1 & 7 & 8 \\ 2 & 0 & \textcircled{5} & 1 & 0 \\ 9 & \textcircled{4} & 1 & 6 & 2 \\ 9 & 3 & 2 & 1 & \textcircled{2} \\ 1 & 2 & 2 & \textcircled{6} & 7 \end{bmatrix} \tag{2}$$

The sign of σ can either be minus one or one and is determined by rearranging rows such that the chosen elements form a diagonal line, as demonstrated in equation (3). Notably, in transitioning from equation (2) to equation (3), rows 2 and 3, as well as rows 4 and 5, are swapped, with each swap altering the sign. Given that the determinant of the unity matrix is one, indicative of a positive sign, and considering that two row swaps occurred to transform (2) into (3), the sign effectively switches by $(-1)^2 = 1$. Thus, the sign of matrix M' remains the same as matrix M.

$$M = \begin{bmatrix} \textcircled{1} & 3 & 1 & 7 & 8 \\ 9 & \textcircled{4} & 1 & 6 & 2 \\ 2 & 0 & \textcircled{5} & 1 & 0 \\ 1 & 2 & 2 & \textcircled{6} & 7 \\ 9 & 3 & 2 & 1 & \textcircled{2} \end{bmatrix} \tag{3}$$

Whenever $\det(M) < 0$ for some matrix M we say that the sign is -1. A matrix N with $\det(N) > 0$ has a sign of 1. The sign of a matrix is also referred to as negative or positive respectively.

Lemma 6.1. *Switching two rows in a square matrix changes the sign of its determinant.*

Proof. Cf. e.g. [8, p.256]. □

Lemma 6.1 becomes quit intuitive when you look at how the determinant is defined visually.

Lemma 6.2. *If the product of a scalar and one row of a square matrix is added to a different row then the determinant stays the same.*

Proof. Cf. e.g. [8, p.258]. □

Lemma 6.3. *If the product of a scalar and one column of a square matrix is added to a different column then the determinant stays the same.*

Proof. Cf. e.g. [8, p.256-258]. □

6.2 Even and odd elements in $\mathbb{E}_{4^d}$

Before diving into the algorithm that computes the determinant for a matrix with elements in $\mathbb{E}_{4^d}$, let us explore a few lemmas for elements in $\mathbb{E}_{4^d}$. These lemmas provide some of the motivation for why $\mathbb{F}_{2^d}$ and $\mathbb{E}_{4^d}$ are used. Without these lemmas, it would be impossible to compute the permanent and determinant with the method described in [1].

Definition 6.2. *(Even elements in $\mathbb{E}_{4^d}$).*
An element $\sigma = \sum a_i x^i \in \mathbb{E}_{4^d}$ is called even if every coefficient a_i is even.

It is important to note that every coefficient of a polynomial in $\mathbb{E}_{4^d}$ must be even for the polynomial to be considered even. For instance, the polynomial $2 + 2x + 3x^2 \in \mathbb{E}_{4^d}$ for some $d > 2$ is not even, as the coefficient for x^2 is not even.

Corollary 6.1. *Let $\tau, \sigma \in \mathbb{E}_{4^d}$ and τ be even. Then $\tau\sigma$ is even.*

Proof. Write $\tau = 2 \cdot (a_0 + a_1x + ... + a_nx^n)$ and $\sigma = b_0 + b_1x + ... + b_nx^n$. Then $\tau\sigma = 2 \cdot (a_0 + a_1x + ... + a_nx^n)(b_0 + b_1x + ... + b_nx^n)$, which is even. $\square$

The algorithm for computing the determinant with entries in $\mathbb{E}_{4^d}$, explained in the next subsection, performs row operations to change certain matrix entries from odd to even elements. It is important that no even elements change back into odd elements during these row operations. This is ensured by using Lemma 6.1.

Corollary 6.2. *Let $\tau, \sigma \in \mathbb{E}_{4^d}$ and τ, σ be even. Then $\tau\sigma \equiv 0 \in \mathbb{E}_{4^d}$.*

Proof. Write $\tau = 2 \cdot (a_0 + a_1x + ... + a_nx^n)$ and $\sigma = 2 \cdot (b_0 + b_1x + ... + b_nx^n)$ then $\tau\sigma = 4 \cdot (a_0 + a_1x + ... + a_nx^n)(b_0 + b_1x + ... + b_nx^n)$. Since $4 \equiv 0$ (*modulo* 4) we can conclude that $\tau\sigma \equiv 0$. $\square$

The following proof is closely based on the proof in [1] and is provided to give the reader insight into the algorithm for the determinant. Some extra steps have been added to make the proof easier to understand.

Lemma 6.4. *For all $\sigma, \upsilon \in \mathbb{E}_{4^d}$ with σ odd, there exists a $\tau \in \mathbb{E}_{4^d}$ such that $\upsilon - \sigma\tau$ is even.*

Proof. Since σ is odd the projection $\underline{\sigma}$ is nonzero and thus has a multiplicative inverse $\underline{\sigma}^{-1} \in \mathbb{F}_{2^d}$. Take $\tau = \overline{\underline{\sigma}^{-1}}\upsilon$ and observe $\underline{\upsilon - \sigma\tau} = \underline{\upsilon} - \underline{\sigma} \cdot \underline{\tau} = \underline{\upsilon} - \underline{\sigma} \cdot \underline{\sigma}^{-1} \cdot \underline{\upsilon} = \underline{\upsilon} - \underline{\upsilon} = 0$.
Since $\underline{\upsilon - \sigma\tau} = 0$, all of its coefficients are either 0 or 2 from which we can conclude that $\upsilon - \sigma\tau$ is even. $\square$

Lemma 6.4 is a prime example of why Björklund, Husfeldt, and Kaski [1] needed not only $\mathbb{E}_{4^d}$ but also the polynomial quotient ring $\mathbb{F}_{2^d}$. Although $\mathbb{E}_{4^d}$ is used to compute the permanent and determinant, it lacks the property of being a field. By defining $\mathbb{E}_{4^d}$ as an extension of $\mathbb{F}_{2^d}$, which is a field, all necessary properties for efficient permanent and determinant computing are retained in $\mathbb{E}_{4^d}$. If $\mathbb{F}_{2^d}$ were not a field, Lemma 6.4 could not rely on the fact that every element in $\mathbb{F}_{2^d}$ has an inverse, and thus it could not find a τ. This τ, as will become clear to the reader in the following subsection, is essential for performing row operations on matrices with entries in $\mathbb{E}_{4^d}$.

One might wonder why $\mathbb{F}_{2^d}$ itself is not used to compute the permanent and determinant. As will become clear in Subsection 6.7, $\mathbb{F}_{2^d}$, which has characteristic two, does not meet the requirements to find even cycles. Since there is no field of characteristic four, another construction had to be made, leading to the field $\mathbb{F}_{2^d}$ and its extension with characteristic four, $\mathbb{E}_{4^d}$.

6.3 The determinant in $\mathbb{E}_{4^d}$

We now are ready to perform the algorithm that computes the determinant of a matrix whose elements are in $\mathbb{E}_{4^d}$. For the following example we take, $n = 3$ and $g(x) = 1+x+x^2+x^3+x^4$. Since $g(x)$ is of degree 4, we will be working in $\mathbb{E}_{4^4}$.

We will start with introducing the matrix I, which will help us decide which row operations should be executed. Take matrix M as an $n \times n$ matrix with entries in $\mathbb{E}_{4^4}$ and let the indicator matrix I be constructed such that $I_{i,j} = 0$ if $M_{i,j}$ is even and $I_{i,j} = 1$ if $M_{i,j}$ is odd. Let

$$M^{(0)} = \begin{bmatrix} 2+2x & 1+2x & 2 \\ 1+x & 1 & 3x \\ 2+x & 1+3x & 3 \end{bmatrix} \text{ then } I^{(0)} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

6.3.1 The first step

The first step is to perform row operations on $M^{(0)}$ such that we end up with a matrix that has only even lower diagonal elements. For these row operations it is needed that each diagonal entry contains an odd element. We therefore start by switching the first and second row from $M^{(0)}$ changing the sign of the determinant. This leaves us with the matrices

$$M^{(1)} = \begin{bmatrix} 1+x & 1 & 3x \\ 2+2x & 1+2x & 2 \\ 2+x & 1+3x & 3 \end{bmatrix} \text{ and } I^{(1)} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

We are now ready to transform all entries below the main diagonal to even elements, starting with the first column. In this case the only odd element in the first column beneath the diagonal is the element in the third row.

$$M^{(1)} = \begin{bmatrix} 1+x & 1 & 3x \\ 2+2x & 1+2x & 2 \\ 2+x & 1+3x & 3 \end{bmatrix}, \; I^{(1)} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}, \; \tau^{(1)} = 3 + x + x^3.$$

By taking $\upsilon = M^{(1)}_{1,3}$ and $\sigma = M^{(1)}_{1,1}$ and using Lemma 6.4 we get our scalar $\tau^{(1)}$. This scalar τ is used to perform our first row operation, subtracting τ times the first row of the third row. This changes $M^{(1)}$ and its indicator matrix $I^{(1)}$ to

$$M^{(2)} = \begin{bmatrix} 1+x & 1 & 3x \\ 2+2x & 1+2x & 2 \\ 2x & 2+2x+3x^3 & 2+2x+3x^3 \end{bmatrix} \text{ and } I^{(2)} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

Note that these row operations do not change the determinant per Lemma 6.2 so it follows that $\det(M^{(1)}) = \det(M^{(2)})$. Once all the odd elements below the first diagonal are transformed to even elements we start looking at the elements under the second diagonal entry.

Looking at $I^{(2)}$ it becomes clear that we need to transform the third element of the second column. We now take $\upsilon = M^{(2)}_{3,2}$ and $\sigma = M^{(2)}_{2,2}$ to get $\tau_{(2)} = 2 + 2x + 3x^3$.

$$M^{(2)} = \begin{bmatrix} 1+x & 1 & 3x \\ 2+2x & 1+2x & 2 \\ 2x & 2+2x+3x^3 & 2+2x+3x^3 \end{bmatrix}, \; I^{(2)} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

Subtracting τ times the second row from the third one leaves us with the matrix $M^{(3)}$, which has an upper triangular indicator matrix $I^{(1)}$.

$$M^{(3)} = \begin{bmatrix} 1+x & 1 & 3x \\ 2+2x & 1+2x & 2 \\ 2+2x^2 & 2+2x+2x^2+2x^3 & 2+2x+x^3 \end{bmatrix}, \; I^{(3)} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

6.3.2 The second step

The triangular form of $M^{(3)}$ enables us to remove all leftover non diagonal odd elements by column operations using Lemma 6.3.

$$M^{(3)} = \begin{bmatrix} 1+x & 1 & 3x \\ 2+2x & 1+2x & 2 \\ 2+2x^2 & 2+2x+2x^2+2x^3 & 2+2x+x^3 \end{bmatrix}, \; I^{(3)} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Let $\upsilon = M^{(3)}_{1,2}$ and $\sigma = M^{(0)}_{1,1}$ then $\tau = x + x^3$. When subtracting τ times the first column from the second column in $M^{(3)}$ we create $M^{(4)}$ and its indicator matrix $I^{(4)}$.

$$M^{(4)} = \begin{bmatrix} 1+x & 2 & 3x \\ 2+2x & 3+2x & 2 \\ 2+2x^2 & 2x^2+2x^3 & 2+2x+x^3 \end{bmatrix} \text{ and } I^{(4)} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

We are now left with just one odd element that does not sit on the main diagonal, namely the first element of the third column.

$$M^{(4)} = \begin{bmatrix} 1+x & 2 & 3x \\ 2+2x & 3+2x & 2 \\ 2+2x^2 & 2x^2+2x^3 & 2+2x+x^3 \end{bmatrix}, \quad I^{(4)} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } \tau = 1+x+x^3.$$

To transform this element we use Lemma 6.4 one last time and set $\upsilon = M^{(4)}_{1,3}$ and $\sigma = M^{(4)}_{1,1}$ to get our scalar τ. We finish the second step by subtracting the first column τ times from the third column leaving us with

$$M^{(5)} = \begin{bmatrix} 1+x & 2 & 2x \\ 2+2x & 3+2x & 2+2x \\ 2+2x^2 & 2x^2+2x^3 & 2+2x^2+x^3 \end{bmatrix} \text{ and } I^{(5)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

6.3.3 The third step

We are now ready for the third and final step. Using Lemma 6.2, we see that only one specific combination contributes to the determinant of $M^{(5)}$, the diagonal elements. Any other combination includes two or more even elements, leading to their product being zero. Therefore we can conclude that,

$$\det(M^{(5)}) = (1+x)\cdot(3+2x)\cdot(2+2x^2+x^3) \equiv 3+x+x^2.$$

Remember that, while the subtraction of rows and columns using a scalar that were performed did not change the determinant, the switching of rows between $M^{(0)}$ and $M^{(1)}$ did change the sign. We can therefore conclude that

$$\det(M^{(0)}) \equiv (-1)\cdot\det(M^{(5)}) \equiv (-1)\cdot(3+x+x^2) \equiv 1+3x+3x^2.$$

6.4 Some particular cases

With the implementation of this technique, the calculation of the determinant unfolds into three different scenarios. The first scenario is when each row and column contains exactly one odd element as can be seen in equation (4). In that case the determinant is the product of the odd elements and the sign of the matrix which was shown in Subsection 6.3.

$$P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Two different scenarios occur whenever one or more diagonal entries are even. By prioritizing the presence of odd elements in lower rows it is still possible to perform the operations in step two of the algorithm. This gives us the following cases: either one diagonal entry is even and the rest odd or two or more diagonal entries are even. The two cases will form the indicator matrices

$$I_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \text{ and } I_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{5}$$

In the case of I_2 it is also possible that more diagonal entries are even. By Lemma 6.1 the determinant of a matrix with indicator matrix I_1 is even. Using Lemma 6.2 we see that a matrix that creates the indicator matrix I_2 has a determinant of zero.

6.5 Time complexity

Efficiently computing the determinant is crucial, and this algorithm achieves just that. Let us take a look at its runtime complexity to see why it is efficient. Rearranging all rows so that each contains only one element in every column takes a maximum of n^2 computations. Each computation involves multiplying a row and then adding or subtracting it from another row, which takes n computations. In $\mathbb{E}_{4^d}$, computations can be completed in d steps, as shown in [7]. This gives us a time complexity of $\mathcal{O}(n^3 \cdot d)$. To find the scalar for each row operation the inverse must be found. To find the inverse we need to solve a linear system of length d, which takes $\mathcal{O}(d^2)$ time. In [1] d is set equal to $5 \cdot \log_2(n)$. Since this operation occurs only once for each element in the matrix, and considering that for larger n it follows that $d < n$, we can deduce that $d \cdot n < n^2$ and that finding the inverse in each row operation does not add to the runtime complexity.

6.6 A different algorithm for the determinant

When dealing with matrices whose entries belong to $\mathbb{R}$, a similar algorithm exists for finding the determinant, commonly taught in most linear algebra classes. This method employs row eliminations to create an upper triangular matrix similar to step 1 in Subsection 6.3. The determinant of this triangular matrix is trivial to compute since any combination that includes off-diagonal entries will involve a zero from the lower triangle, resulting in a product of zero. Thus, the determinant is simply the product of the diagonal entries.

For a matrix with entries in $\mathbb{E}_{4^d}$, it turns out, it is not always possible to transform row operations such that a upper triangular matrix is obtained.

Therefore more column operations are needed which are shown in the second step. Using Lemma 6.2 the determinant then becomes trivial to compute.

6.7 The permanent

The method to find even cycles described in [1] uses both the determinant and the permanent, which will be defined shortly. Finding the permanent, although similar to the determinant, is often a much harder problem then the determinant. The difficulties in computing the permanent will become clear once we have defined and explained the permanent.

Definition 6.3. *(The permanent).*
The permanent of an $n \times n$ matrix M is equal to

$$\operatorname{per}(M) = \sum_{\sigma \in S_n} M_{1,\sigma(1)} \cdot M_{2,\sigma(2)} \cdot \ldots \cdot M_{n,\sigma(n)}.$$

The key difference between the permanent and the determinant is that there is no sign associated with each combination. As a consequence, Lemmas 6.2 and 6.3 do not hold, and the permanent of the matrix obtained by applying row or column operations may be different. Therefore, the algorithm described in Subsection 6.3 does not work for calculating the permanent.

In most cases, there is no simple solution to work around this problem. However, there are a few special cases. One such case is when the permanent needs to be calculated modulo 2. Since $1 \equiv -1 \pmod 2$, the permanent equals the determinant modulo 2, for which polynomial-time algorithms exist.

In [1], the algorithm heavily relies on Theorem 2.1, which deduces the length of the shortest even cycle by subtracting the determinant from the permanent and examining the highest non-zero even coefficient. However, when working modulo 2, this approach is ineffective since $0 \equiv 2$. The highest non-zero even coefficient, therefore, does not exist, making the use of modulo 2 ineffective in finding the shortest even cycle.

By 1979, Valiant had proven that the permanent can be computed modulo 2^k for some k in $\mathcal{O}(n^{4k-3})$ steps [2, p.190]. Moreover, Valiant showed that computing the permanent modulo l, where l is not a power of 2, can only be achieved in polynomial time if every single-valued function that can be verified in polynomial time can also be computed in polynomial time. However, since this remains unproven and may never be proven, mathematicians had to find a different approach.

Instead of working modulo 2, Björklund [1] created the ring $\mathbb{E}_{4^d}$, which shares some of those characteristics which makes computing the permanent easier. Although both Lemma 6.2 and Lemma 6.3 still do not apply, there is a way to compute the permanent in polynomial time. In [1], each row operation is accompanied by a subroutine that calculates the change in the permanent. By tracking these changes and adding them to the final result, the permanent of the original matrix can be calculated. If these changes are disregarded, the algorithm is essentially the same as that for the determinant.

7 Coding the algorithm for the determinant

The code builds upon the code described in Section 5 and uses the created class E4D which represents a polynomial in $\mathbb{E}_{4^d}$ with irreducible polynomial $g(x) = 1 + x + x^2 + x^3 + x^4$. Additionally, two functions were added to the class E4D. One of these functions, GETTAU(σ), finds τ by using the method described in Lemma 6.4. This τ is later used in row operations as a scalar. The other function is called ISEVEN() and returns true whenever the polynomial is even and false otherwise. With this function, the indicator matrix I can be constructed.

7.1 Overview of the code

Within the code, a main function DETERMINANT(M) is defined that uses several helper functions. The function follows the steps described in Subsection 6.3. First, the determinant function reduces the input matrix M to a matrix M' which has only even elements below the diagonal. During this process the sign of the determinant might flip. The function DETERMINANT keeps track of these switches to ensure that the correct sign is used in the end. The matrix M' is reduced even further to the matrix M'' where only diagonal entries can be odd. At last, the product of the diagonal entries is returned using the correct sign that the function kept track of in the first step.

7.2 The inverse in $\mathbb{F}_{2^d}$

Within the class E4D the function GETTAU(σ) is given. This function needs to find the inverse of the projection of the input polynomial σ. Finding the inverse after projecting σ to $\mathbb{F}_{2^d}$ is done using the help of some open source code that can be found at Github [9]. The code in B.4 consists of both the code that was found on Github and some extra coding to make all the functions compatible.

7.3 Discussion

The code can generate random polynomials in $\mathbb{E}_{4^d}$, which can be used as entries in a square matrix to test the code. Calculations have been checked manually for 3×3 and 4×4 matrices. Although not tested for correctness, the Python script can compute the determinant of matrices larger than 50×50. Between the steps of the algorithm, the indicator matrix is printed to ensure that the row operations have the desired effect.

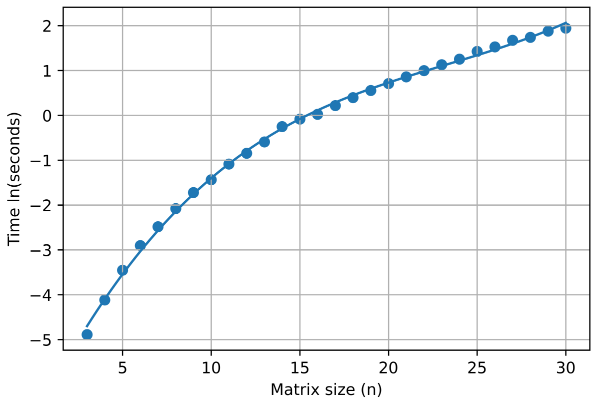


Figure 8: The average of the logarithm of the time, in seconds, to compute the determinant of a matrix of size n.

The time complexity is tested by computing the determinant of 20 random matrices for each size from $n = 4$ to $n = 30$. The logarithm of the elapsed time for each matrix size shows a logarithmic trend. Although it is not a formal proof of a polynomial-time algorithm, it is a strong indicator that the programmed algorithm operates faster than exponential-time algorithms.

7.4 Conclusion

The advancement in computing the permanent using $\mathbb{E}_{4^d}$, described in [1], would be of no use if the determinant of a matrix with entries in $\mathbb{E}_{4^d}$ could not be computed in polynomial time. By first explaining and understanding the polynomial quotient rings $\mathbb{F}_{2^d}$ and $\mathbb{E}_{4^d}$, which were then used to explain an algorithm that computes the determinant in Section 6, a Python script was written. This Python script, which can perform operations in $\mathbb{E}_{4^d}$ and compute the determinant for matrices with entries in $\mathbb{E}_{4^d}$, was manually checked for correctness on smaller matrices and then timed to provide an indicator of the time complexity. The resulting data conforms to Subsection 6.5, where the algorithm is shown to be polynomial. Therefore, we can conclude that the determinant with entries in $\mathbb{E}_{4^d}$ can be computed in polynomial time.

A Bibliography

References

[1] A. Björklund, T. Husfeldt, and P. Kaski. The shortest even cycle problem is tractable. *SIAM Journal on Computing*, pages STOC22–22–STOC22–45, 2022.

[2] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[3] R. Yuster and U. Zwick. Finding even cycles even faster. *SIAM Journal on Discrete Mathematics*, 10(2):209–222, 1997.

[4] N Robertson, P. D. Seymour, and Robin Thomas. Permanents, Pfaffian Orientations, and Even Directed Circuits. *Annals of Mathematics*, 150(3):929–975, 1999.

[5] G.L. Mullen and D. Panario. *Handbook of Finite Fields*. Chapman and Hall/CRC, 1 edition, 2013.

[6] D.M. Burton. *A First Course in Rings and Ideals*. Addison-Wesley series in mathematics. Addison-Wesley Publishing Company, 1970.

[7] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 3 edition, 2013.

[8] J.B. Fraleigh and R.A. Beauregard. *Linear Algebra*. Student's solutions manual. Addison-Wesley, 1995.

[9] Mildsunrise. Integer (and polynomial) modular arithmetic for Python! https://gist.github.com/mildsunrise/e21ae2b1649532813f2594932f9e9371.

B Code

B.1 Operations in $\mathbb{E}_{4^d}$

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 15 10:02:05 2024

@author: jordi
"""
import numpy as np
from testBEP_E4d import getProjectionInverse


class polynomial:
    """
    Creates a polynomial in the polynomial ring Z_i[x]
    where i is given as input
    """
    def __init__(self, coefficients, modulo):
        self.len = len(coefficients)
        self.coefficients = np.array(coefficients) % (modulo)
        self.modulo = modulo

    def lenn(self):
        return len(self.coefficients)


    def __str__(self):
        """
        Returns a string which is printed in the console whenever
        the polynomial is printed
        """
        string = ""
        for i, coeff in enumerate(self.coefficients):
            string = string + " "+ str(coeff) + "x**" + str(i)
        return string

    def normalize(self, poly):
        """
        Removes coefficients that are higher
        than the degree of the polynomial
        """
        while poly[-1] == 0 and len(poly) > 2:
            poly = poly[:-1]
        return poly

    def checkOther(self, other):
        """
        Make sure that only polynomials can interact with eachother
        """
        if isinstance(other, polynomial):
            if self.lenn() > other.lenn():
                other.coefficients = np.pad(other.coefficients,
                        (0,self.lenn()-other.lenn()), \
                            'constant', constant_values=(0,0))
            if other.lenn() > self.lenn():
                self.coefficients = np.pad(self.coefficients, \
                        (0,other.lenn()-self.lenn()), 'constant', \
                            constant_values=(0,0))
            return

        else:
            raise TypeError("Other is not of class polynomial")
        return self

    def add(self, other):
        """
        Returns itself plus a polynomial that is given as input
        """
        self.checkOther(other)
        return polynomial((self.coefficients + other.coefficients) \
                          % (self.modulo), self.modulo)

    def minus(self, other):
        """
        Returns itself minus a polynomial that is given as input
        """
        self.checkOther(other)
        return polynomial((self.coefficients - other.coefficients) \
                          % (self.modulo), self.modulo)

    def multiply(self, other):
        """
        Returns the product of the polynomial and another polynomial
        which is given as input
        """
        self.checkOther(other)
        newSelf = np.zeros(2*len(self.coefficients) - 1)
        for i, coeffSelf in enumerate(self.coefficients):
            for j, coeffOther in enumerate(other.coefficients):
                newSelf[i+j] += coeffSelf*coeffOther

        newSelf = newSelf % (4)
        return polynomial(self.normalize(newSelf), self.modulo)

class QuotientRing(polynomial):
    """
    Creates a polynomial within a polynomial quotient ring. The function
    g(x) that defines the quotient ring is given as input.
    """
    def __init__(self, coefficients, modulo, irreduciblePol):
        super(QuotientRing, self).__init__(coefficients, modulo)
        self.irr = np.array(irreduciblePol)%self.modulo


    def reduce(self):
        """
        Reduces the polyonomial to its remainder within
        the polynomial quotient ring
        """
        #print(self.coefficients)
        #print("\n\n\n\n")
        while self.lenn() < len(self.irr) - 1:
            self.coefficients = np.append(self.coefficients, [0])
            self.len += 1

        if self.lenn() >= len(self.irr):
            for i in range(self.lenn()-1, len(self.irr)-2, -1):

                adjusterCoeff = np.zeros(len(self.coefficients))

                adjusterCoeff[i-(len(self.irr)-1)] = self.coefficients[i]

                adjuster = polynomial(adjusterCoeff, self.modulo)

                irrReversed = polynomial(np.zeros(len(self.irr)-1), \
                                                     self.modulo)
                irrReversed = irrReversed.minus(polynomial(self.irr[:-1], \
                                                     self.modulo))

                replacement = irrReversed.multiply(adjuster)
                #print(i)
                #print(self)
                self.coefficients[i] = 0
                self.coefficients = self.add(replacement).coefficients
        self.coefficients = self.normalize(self.coefficients)
        self.coefficients = self.coefficients % self.modulo
        return self

    def ADD(self, other):
        """
        Adds itself with another polynomial and returns the reduced result
        """
        if np.array_equal(self.irr, other.irr):
            poly = self.add(other)
            return QuotientRing(poly.coefficients, self.irr).reduce()
        else:
            raise TypeError("Polynomials do not belong to the same ring")

    def MINUS(self, other):
        """
        Subtracts itself with another polynomial and returns
        the reduced result
        """
        if np.array_equal(self.irr, other.irr):
            poly = self.minus(other)
            return QuotientRing(poly.coefficients, self.modulo, \
                                self.irr).reduce()
        else:
            raise TypeError("Polynomials do not belong to the same ring")

    def MUL(self, other):
        """
        Multiplies itself with another polynomial and returns
        the reduced result
        """
        if np.array_equal(self.irr, other.irr):
            poly = self.multiply(other)
            output = QuotientRing(poly.coefficients, self.modulo,\
                                  self.irr).reduce()

            return output
        else:
            raise TypeError("Polynomials do not belong to the same ring")


class E4d(QuotientRing):
    """
    Creates a polynomial in the E4d polynomial quotient ring
    """
    def __init__(self, coefficients, irreduciblePol):
        super(E4d, self).__init__(coefficients, 4, irreduciblePol)

    def __add__(self, other):
        """
        Returns the sum of itself plus another
        polynomial that is given as input
        """
        if isinstance(other, E4d):
            newPoly = self.ADD(other)
            return E4d(newPoly.coefficients, newPoly.irr)
        else:
            print("Not of the same class")

    def __sub__(self, other):
        """
        Returns itself minus another
        polynomial that is given as input
        """
        if isinstance(other, E4d):
            newPoly = self.MINUS(other)
            return E4d(newPoly.coefficients, newPoly.irr)
        else:
            print("Not of the same class")

    def __mul__(self, other):
        """
        Returns the product of itself plus another
        polynomial that is given as input
        """
        if isinstance(other, E4d):
            newPoly = self.multiply(other)
            newPoly = QuotientRing(newPoly.coefficients, newPoly.modulo, \
                                   self.irr)
            newPoly.reduce()
            return E4d(newPoly.coefficients, newPoly.irr)
        else:
            print(type(self), type(other),"Not of the same class")

    def getTau(self, sigma):
        """
        Returns a polynopmial tau
        such that self - tau * sigma is an even polynomial
        """
        if sigma.isEven():
            print("Sigma cannot be even")
            print(sigma)
        else:
            sigmaInvCoeff = getProjectionInverse(sigma.coefficients, \
                                                 sigma.irr % 2)
            sigmaInv = E4d(sigmaInvCoeff, sigma.irr)
            tau = sigmaInv * self
            return tau

    def makeEven(self, sigma):
        """
        Returns itself - tau * sigma
        such that the returned value is even
        """
        # Sigma should be a polynomial in E4d
        tau = self.getTau(sigma)
        return self - (tau*sigma)

    def isEven(self):
        """
        Checks whether it is an even polynomial and
        returns true if it is, false otherwise
        """
        even = True
        for coeff in self.coefficients:
            if coeff % 2 != 0:
                even = False
        return even

if __name__ == "__main__":
    gx = [1,1,1,1,1]
    a = E4d([1,1], gx)
    b = E4d([3,2], gx)
    c = E4d([2,0,2,1], gx)

    d = a.multiply(b).multiply(c)
    print(d)
```

B.2 The determinant in $\mathbb{E}_{4^d}$

```
"""
Created on Fri May  3 12:25:27 2024

@author: jordi
"""
from BEP_Computing_E4d import QuotientRing
from BEP_Computing_E4d import E4d
import numpy as np
from testBEP_E4d import getProjectionInverse
import random as r
import multiprocessing
import time

"""
Seeds:
    11
    10
"""
#r.seed(11)
"""
gx is the irreducible polynomial over F2d
"""
gx = [1,1,1,1,1]








def getRandomPoly():
    """
    Create a random polynomial in E4d
    """
    coeffs = []
    for i in range(len(gx)-1):

        coeffs.append(r.randint(0,3))
    return E4d(coeffs, gx)

def getRandomPolyMatrix(n):
    """
    Get an NxN matrix with random polynomials
    in E4d as entries
    """
    rows = []
    for i in range(n):
        column = []
        for j in range(n):
            column.append(getRandomPoly())
        rows.append(column)
    return rows

def rowOp(matrix, row1, col1, row2):
    """
    Performs the row operation that is needed
    given the rows and columns that should be made even
    """

    """entry row1, col1 gets changed to an even element"""

    v = matrix[row1][col1]
    s = matrix[row2][col1]

    if s.isEven():
        print(row1, col1, row2)

    t = v.getTau(s)

    #print('tau = ' , t)
    for i, elem in enumerate(matrix[row1]):
        elem2 = elem
        matrix[row1][i] = elem - t*matrix[row2][i]


    return matrix


def Indicator(matrix):
    """
    Given a matrix with polynomials as input it returns a matrix of
    the same size with 1 if the entry is odd and 0 if the entry is odd
    """
    n = len(matrix)
    indicators = np.ones((n,n))
    for i in range(n):
        for j in range(n):
            if matrix[i][j].isEven():
                indicators[i][j] = 0
            else:
                indicators[i][j] = 1
    return indicators



def Swap(matrix, row1, row2):
    """Swaps two rows of a matrix"""
    matrix[row1][:], matrix[row2][:] = matrix[row2][:], matrix[row1][:].copy()
    return matrix

def TriangleMatrix(matrix):
    """Returns a matrix that has only even elements
    in the lower left triangle"""
    signChange = 1
    for k in range(len(matrix) - 1):
        #print("New it",k)
        I = Indicator(matrix)
        #print(I)
        ones = np.where(I[:,k])
        if I[k][k] != 1:
            a = np.where(ones[0] > k)[0]
            #print("a = ",a, ones)
            if len(a) > 0:
                """If everything breaks switch ones[0][0] to ones[0][-1]"""
                matrix = Swap(matrix, k, ones[0][-1])
                signChange =  signChange*(-1)
                #print("change")

        I = Indicator(matrix)
        #print(I, "Swapped?")
        for i, row in enumerate(I):
            if i > k and I[i][k] == 1:
                try:

                    #print("d")
                    I = Indicator(matrix)
                    #print(I)
                    matrix = rowOp(matrix, i, k, k)
                    #Mprint(matrix)
                except:
                    #print("\n\n\n\n Hier")
                    I = Indicator(matrix)
                    #print(I)
                    #print(i, k)
    return matrix, signChange
```

```
def colOp(matrix, col1, row1, col2):
    """entry row1, col1 gets changed to an even element"""
    v = matrix[row1][col1]
    s = matrix[row1][col2]
    t = v.getTau(s)
    #print("tau = ", t)

    for i in range(len(matrix)):
        matrix[i][col1] = matrix[i][col1] - t*matrix[i][col2]


    return matrix

def Mprint(matrix):
    """
    Print the matrix with polynomials in the console
    """
    for row in matrix:
        string = ""
        for col in row:
            string = string + " \n " + str(col)
        print(string + " \n\n")

def noSimilarRows(matrix):
    """
    Make sure that there are no rows which have
    odd and even polynomials on the exact same entries
    """
    I = Indicator(matrix)
    for i in range(len(matrix)):
        if I[i][i] == 0:
            for j in range(len(matrix)):
                if j != i and np.array_equal(I[i,:], I[j,:]):
                    matrix = rowOp(matrix, i, j, j)
    return matrix

def Determinant(matrix):
    """
    Computes the determinant
    """
    sign = 1
    I = Indicator(matrix)

    print(I)
    """Getting an upper triangle matrix """
    #print("Triangular follows \n")
    matrix, signchange = TriangleMatrix(matrix)
    sign = sign * signchange


    I = Indicator(matrix)
    print("\n\n\n Should be triangular\n")
    print(I)
    matrix = noSimilarRows(matrix)
    I = Indicator(matrix)
    print("\n\n\n Should be triangular 2\n")
    print(I)

    """Gaussian elimination with the columns such that only the diagonal
    can have odd elements as entries"""
    for k in range(len(matrix) - 1):
        #print("\n\n", k)
        for i in range(k+1, len(matrix)):
            if I[k][i] == 1:
                matrix = colOp(matrix, i, k, k)

    I = Indicator(matrix)

    """Calculating the now trivial determinant"""
    product = E4d([1], gx)
    for i in range(len(matrix)):
        #print(matrix[i][i])
        product *= matrix[i][i]


    print(I)
    print('sign = ', sign)
    return product * E4d([sign], gx)

def DetNoPrint(matrix):
    """
    Compute the determinant without printing in the console
    """
    sign = 1
    I = Indicator(matrix)

    #print(I)
    """Getting an upper triangle matrix """
    #print("Triangular follows \n")
    matrix, signchange = TriangleMatrix(matrix)
    sign = sign * signchange


    I = Indicator(matrix)
    matrix = noSimilarRows(matrix)
    I = Indicator(matrix)
    #print("\n\n\n Should be triangular\n")
    #print(I)

    """Gaussian elimination with the columns such that only the diagonal
    can have odd elements as entries"""
    for k in range(len(matrix) - 1):
        #print("\n\n", k)
        for i in range(k+1, len(matrix)):
            if I[k][i] == 1:
                matrix = colOp(matrix, i, k, k)

    I = Indicator(matrix)

    """Calculating the now trivial determinant"""
    product = E4d([1], gx)
    for i in range(len(matrix)):
        #print(matrix[i][i])
        product *= matrix[i][i]


    #print(I)
    #print('sign = ', sign)
    return product * E4d([sign], gx)

"""
M = [[E4d([2,2],gx),E4d([1,2],gx), E4d([2],gx), E4d([1,2,1], gx) ],
     [E4d([1,1],gx),E4d([1],gx), E4d([0,3],gx), E4d([1,1,1], gx) ],
     [E4d([2,1],gx),E4d([1,3],gx), E4d([3],gx), E4d([1,1,3,1], gx) ],
     [E4d([1,3], gx), E4d([1,3,2], gx), E4d([1,1,2], gx), E4d([3,1], gx)]
     ]


M = [[E4d([2,2],gx),E4d([1,2],gx), E4d([2],gx) ],
     [E4d([1,1],gx),E4d([1],gx), E4d([0,3],gx) ],
     [E4d([2,1],gx),E4d([1,3],gx), E4d([3],gx) ]
     ]


"""
```

```python
if __name__ == '__main__':
    for i in range(1):
        n = 10
        M = getRandomPolyMatrix(n)

        det = Determinant(M)
        print(det)
```

B.3 Time complexity of the determinant in $\mathbb{E}_{4^d}$

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Jun  3 16:26:48 2024

@author: jordi
"""
from BEP_Computing_E4d import QuotientRing
from BEP_Computing_E4d import E4d
import numpy as np
from testBEP_E4d import getProjectionInverse
import random as r
import multiprocessing
import time
from determinant_E4d_second import DetNoPrint
from determinant_E4d_second import getRandomPolyMatrix
import matplotlib.pyplot as plt
plt.rcParams['figure.dpi'] = 1000



data = []
checkTill = 30 #Maximum matrix size checked
checkMany = 10 #Amount of matrics checked for each size

"""
The following loops keep track of the amount of time
it takes to compute the determinant
"""
beginT = time.time()
for i in range(3, checkTill + 1):
    t = time.time()
    print(i, time.time() - beginT)
    for j in range(0,checkMany):

        n = i
        M = getRandomPolyMatrix(n)

        det = DetNoPrint(M)
        #print(det)
    data.append([i, time.time() - t])


"""
```

```
Reform data to make it easier to plot and perform linear regression
"""
data = np.array(data)
nn = data[:,0]
times = np.log(data[:,1]/checkMany)

model = np.poly1d(np.polyfit(nn, times, 3))


fig, ax = plt.subplots(
    #figsize=(6, 5)
)  # This sets the figure size to 6 inches wide by 5 inches high

# Plot the baseline

line = np.linspace(3, checkTill, 100)

"""
Plotting the data
"""
ax.grid()
ax.plot(line, model(line))
ax.scatter(nn,times)
ax.set_xlabel("Matrix size (n)")
ax.set_ylabel("Time ln(seconds)")

plt.show()
```

B.4 The inverse in $\mathbb{F}_{2^d}$

```
"""
BINARY POLYNOMIAL ARITHMETIC
These functions operate on binary polynomials (Z/2Z[x]),
expressed as coefficient bitmasks, etc:
  0b100111 -> x^5 + x^2 + x + 1
As an implied precondition, parameters are assumed to be *nonnegative*
integers unless otherwise noted.
This code is time-sensitive and thus NOT safe to use for online cryptography.
"""

from typing import Tuple
import numpy as np

# descriptive aliases (assumed not to be negative)
Natural = int
BPolynomial = int

def p_mul(a: BPolynomial, b: BPolynomial) -> BPolynomial:
    """ Binary polynomial multiplication (peasant). """
    result = 0
    while a and b:
        if a & 1: result ^= b
        a >>= 1; b <<= 1
    return result

def p_mod(a: BPolynomial, b: BPolynomial) -> BPolynomial:
    """ Binary polynomial remainder / modulus.
        Divides a by b and returns resulting remainder polynomial.
        Precondition: b != 0 """
    bl = b.bit_length()
    while True:
        shift = a.bit_length() - bl
        if shift < 0: return a
        a ^= b << shift

def p_divmod(a: BPolynomial, b: BPolynomial) -> Tuple[BPolynomial, \
                                                      BPolynomial]:
    """ Binary polynomial division.
        Divides a by b and returns resulting (quotient, remainder) polynomials.
        Precondition: b != 0 """
    q = 0; bl = b.bit_length()
    while True:
        shift = a.bit_length() - bl
        if shift < 0: return (q, a)
        q ^= 1 << shift; a ^= b << shift

def p_mod_mul(a: BPolynomial, b: BPolynomial, modulus: BPolynomial) \
    -> BPolynomial:
    """ Binary polynomial modular multiplication (peasant).
        Returns p_mod(p_mul(a, b), modulus)
        Precondition: modulus != 0 and b < modulus """
    result = 0; deg = p_degree(modulus)
    assert p_degree(b) < deg
    while a and b:
        if a & 1: result ^= b
        a >>= 1; b <<= 1
        if (b >> deg) & 1: b ^= modulus
    return result

def p_exp(a: BPolynomial, exponent: Natural) -> BPolynomial:
    """ Binary polynomial exponentiation by squaring (iterative).
        Returns polynomial `a` multiplied by itself `exponent` times.
        Precondition: not (x == 0 and exponent == 0) """
    factor = a; result = 1
    while exponent:
        if exponent & 1: result = p_mul(result, factor)
        factor = p_mul(factor, factor)
        exponent >>= 1
    return result

def p_gcd(a: BPolynomial, b: BPolynomial) -> BPolynomial:
    """ Binary polynomial euclidean algorithm (iterative).
        Returns the Greatest Common Divisor of polynomials a and b. """
    while b: a, b = b, p_mod(a, b)
    return a

def p_egcd(a: BPolynomial, b: BPolynomial) -> tuple[BPolynomial, BPolynomial, \
                                                    BPolynomial]:
    """ Binary polynomial Extended Euclidean algorithm (iterative).
        Returns (d, x, y) where d is the Greatest Common Divisor of
        polynomials a and b.
        x, y are polynomials that satisfy: p_mul(a,x) ^ p_mul(b,y) = d
        Precondition: b != 0
        Postcondition: x <= p_div(b,d) and y <= p_div(a,d) """
    a = (a, 1, 0)
    b = (b, 0, 1)
    while True:
        q, r = p_divmod(a[0], b[0])
        if not r: return b
        a, b = b, (r, a[1] ^ p_mul(q, b[1]), a[2] ^ p_mul(q, b[2]))

def p_mod_inv(a: BPolynomial, modulus: BPolynomial) -> BPolynomial:
    """ Binary polynomial modular multiplicative inverse.
        Returns b so that: p_mod(p_mul(a, b), modulus) == 1
        Precondition: modulus != 0 and p_coprime(a, modulus)
        Postcondition: b < modulus """
    d, x, y = p_egcd(a, modulus)
    assert d == 1, print(a, modulus, "inverse does not exist") #inverse exists
    return x

def p_mod_pow(x: BPolynomial, exponent: Natural, modulus: BPolynomial) \
    -> BPolynomial:
    """ Binary polynomial modular exponentiation by squaring (iterative).
        Returns: p_mod(p_exp(x, exponent), modulus)
        Precondition: modulus > 0
        Precondition: not (x == 0 and exponent == 0) """
    factor = x = p_mod(x, modulus); result = 1
    while exponent:
        if exponent & 1:
            result = p_mod_mul(result, factor, modulus)
        factor = p_mod_mul(factor, factor, modulus)
        exponent >>= 1
    return result

def p_degree(a: BPolynomial) -> int:
    """ Returns degree of a. """
    return a.bit_length() - 1

def p_congruent(a: BPolynomial, b: BPolynomial, modulus: BPolynomial) \
    -> bool:
    """ Checks if a is congruent with b under modulus.
        Precondition: modulus > 0 """
    return p_mod(a^b, modulus) == 0

def p_coprime(a: BPolynomial, b: BPolynomial) -> bool:
    """ Checks if a and b are coprime. """
    return p_gcd(a, b) == 1




"""
##########################
Jordi's code below vvvv
```

```
#########################
"""

def bpoly_to_list(bpoly):
    """
    Get a list of all the coefficients of the bit polynomial
    """
    bits  = bin(bpoly)
    coeffs = list(bits[2:])
    for i, coeff in enumerate(coeffs):
        coeffs[i] = int(coeff)
    coeffs.reverse()
    return coeffs

def list_to_bpoly(coeffs):
    """
    Gets as input a list of the coefficients
    of the polynomial and creates a bit polynomial
    with the same coefficients
    """
    accumulator = 0
    for i, coeff in enumerate(coeffs):
        accumulator += (2**i)*coeff
    return int(accumulator)

def getProjectionInverse(coefficients, modulus):
    """
    # Input = coefficients for the polynomial and
    #the modulus polynomial g(x)
    # returns the coefficients of the inverse of the projection
    # of the polynomial.
    """


    newCoeff = coefficients % 2
    newMod = modulus % 2
    bpolyCoeff = list_to_bpoly(newCoeff)
    bpolyMod = list_to_bpoly(newMod)
    inv = p_mod_inv(bpolyCoeff, bpolyMod)
    inv_coeff = bpoly_to_list(inv)
    return np.array(inv_coeff)
```