UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

CRIPTOANÁLISIS DEL CRIPTOSISTEMA DE MCELIECE CLÁSICO MEDIANTE ALGORITMOS GENÉTICOS

Autor

PAULA VILLANUEVA NÚÑEZ

Director

GABRIEL NAVARRO GARULO

Facultad de Ciencias

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

FACULTAD DE CIENCIAS

E.T.S. DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

Granada, a 19 de junio de 2022

Declaración de originalidad

Dña. Paula Villanueva Núñez

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2021-2022, es original, entendida esta, en el sentido de que no ha utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 19 de junio de 2022

Fdo: Paula Villanueva Núñez

AGRADECIMIENTOS

Me gustaría agradecer a mi tutor, Gabriel Navarro Garulo, por haberme guiado y aconsejado a lo largo del desarrollo de este trabajo, así como su tiempo dedicado para ello.

También a toda mi familia por su gran apoyo durante estos cinco años, aunque no hayamos podido pasar mucho tiempo juntos.

A mis compañeros de clase, por habernos ayudado mutuamente, por haber pasado tantas horas de estudio juntos y por haber compartido muchos buenos momentos fuera de la vida académica.

A mis amigos, por haberme escuchado y apoyado durante todo este tiempo, y sobre todo por haber estado cuando más los necesitaba.

RESUMEN

La posible existencia de ordenadores cuánticos amenaza la seguridad de los sistemas criptográficos más conocidos y usados en la actualidad, como RSA. En este trabajo se propone, se estudia y se implementa un criptosistema resistente a estos ataques, el criptosistema de McEliece clásico, junto a una variante, el criptosistema de Niederreiter. Ambos sistemas se construyen a partir de los códigos de Goppa, una clase de códigos de corrección de errores lineales que también desarrollaremos e implementaremos. Estos códigos permiten codificar mensajes y decodificarlos mediante un algoritmo eficiente, el algoritmo de Sugiyama. Previamente, introduciremos y presentaremos los conceptos necesarios para poder abordar el objetivo de este trabajo. Por último, se proponen ataques usando algoritmos genéticos para vulnerar la seguridad del criptosistema de McEliece.

PALABRAS CLAVE: teoría de códigos, Goppa, criptografía, criptografía post-cuántica, McEliece, Niederreiter, algoritmos genéticos, SageMath

SUMMARY

The main goal of this project is to present, study and implement a cryptosystem resistant to attacks by quantum computers, the McEliece cryptosystem.

CHAPTER 1

In this chapter we will present the previous mathematical concepts necessary to understand the subsequent development of this work. We will need to introduce concepts such as finite fields, since they are the algebraic structure on which we will work. We will also deal with the complexity of algorithms and problems, which will be useful to know if we can obtain a solution in polynomial time. Eventually, we will introduce the basic concepts of population-based metaheuristics to further develop genetic algorithms, which will be useful in attempting to break the security of McEliece's cryptosystem.

CHAPTER 2

In the second chapter we will study linear codes, defined as a subspace of a finite field. We will introduce two concepts that will be of great importance throughout the development of this work: the generating matrix and the parity matrix. Each of these matrices presents a unique code. We will also define some significant measures along with their properties, such as the Hamming distance or the Hamming weight. We will deal with the complexity of the problem of finding a word with minimum weight and related issues. We will finish by presenting the Brouwer-Zimmermann algorithm for calculating the distance of a code.

CHAPTER 3

In this chapter we will develop and implement a new class of linear error correcting codes, the Goppa codes. We will study the parity matrix associated with these codes and we will obtain from it the generating matrix, which will be useful for encoding the messages. We will also analyze the corrective capacity, which will depend on the degree of the polynomial on which the code is defined. In addition, we will develop the encoding and decoding process, including examples of the implementation carried out. For the decoding we will use an efficient algorithm, the Sugiyama algorithm, which is based on the Euclid algorithm and will allow us to recover the messages.

CHAPTER 4

The fourth chapter contains the study of the goal of this project, the McEliece cryptosystem. Previously, we will introduce the concept and objectives of cryptography, as well as two main types of cryptosystems. Next, we will present the possibility of the existence of quantum computers and the threat that they present to the security of some existing cryptosystems, proposing some alternatives that are capable of resisting their attacks, such as the McEliece cryptosystem. This system is constructed from the Goppa codes studied in the previous chapter. We will study the generation of its keys (public and private), as well as the encryption and decryption processes. We will also analyze a variant, the Niederreiter system, which is equivalent in security. We will present examples of the McEliece system using the implementation carried out to show its operation.

CHAPTER 5

In this last chapter, we will develop a couple of attacks using genetic algorithms to break the security of McEliece's cryptosystem. These attacks are based on calculating the distance of a code from an echelon matrix of a matrix that generates the equivalent code, which will be obtained by swapping the code columns. We will study and implement adaptations of the GGA and CHC algorithms. Finally, we will analyze the results obtained after applying these algorithms to the McEliece system.

To carry out this project, the following has been implemented in SageMath.

- A class to define the Goppa codes, another class for your encoder, and another class for your decoder using Sugiyama's algorithm. The documentation of this class can be found in the annex A, along with the helpful functions used.
- A class to define the McEliece cryptosystem, whose documentation can be found in the annex B, along with the helpful functions used.
- One function for the GGA genetic algorithm and another function for the CHC genetic algorithm. The documentation of these functions, along with other helpful functions used, can be found in the annex C.

KEYWORDS: coding theory, Goppa, criptography, post-quantum criptography, McEliece, Niederreiter, genetic algorithms, SageMath

ÍNDICE GENERAL

Introducción y objetivos 13

1. PRELIMINARES 17
   - 1.1. Anillos . . . 17
   - 1.2. Cuerpos finitos . . . 19
     - 1.2.1. Anillo de polinomios sobre cuerpos finitos . . . 20
   - 1.3. Teoría de la complejidad . . . 24
     - 1.3.1. Complejidad de los algoritmos . . . 24
     - 1.3.2. Complejidad de los problemas . . . 25
   - 1.4. Metaheurísticas basadas en poblaciones . . . 26
     - 1.4.1. Algoritmos evolutivos . . . 27
2. INTRODUCCIÓN A LA TEORÍA DE CÓDIGOS LINEALES 31
   - 2.1. Códigos lineales . . . 31
   - 2.2. Código dual . . . 33
   - 2.3. Pesos y distancias . . . 34
   - 2.4. Clasificación por isometría . . . 37
   - 2.5. Algoritmo para el cálculo de la distancia . . . 38
3. CÓDIGOS DE GOPPA 45
   - 3.1. Códigos clásicos de Goppa . . . 45
     - 3.1.1. Códigos binarios de Goppa . . . 51
     - 3.1.2. Codificación de los códigos de Goppa . . . 51
     - 3.1.3. Decodificación de los códigos de Goppa . . . 52
4. CRIPTOGRAFÍA POST-CUÁNTICA BASADA EN CÓDIGOS 61
   - 4.1. Introducción . . . 61
   - 4.2. Objetivos de la criptografía . . . 62
   - 4.3. Criptografía asimétrica . . . 63
     - 4.3.1. RSA . . . 64
   - 4.4. Criptografía post-cuántica . . . 66
     - 4.4.1. Criptosistema de McEliece . . . 67
     - 4.4.2. Criptosistema de Niederreiter . . . 72
     - 4.4.3. Seguridad del criptosistema de McEliece . . . 74
5. ATAQUE USANDO ALGORITMOS GENÉTICOS 77
   - 5.1. Esquema basado en permutaciones . . . 77
   - 5.2. Algoritmos genéticos . . . 80

5.2.1. Algoritmo GGA 81
5.2.2. Algoritmo CHC 83
5.2.3. Ejemplos 84
Conclusión y vías futuras 89
A. IMPLEMENTACIÓN EN SAGEMATH DE LOS CÓDIGOS DE GOPPA 91
A.1. Clase para códigos de Goppa 91
A.2. Codificador para códigos de Goppa 96
A.3. Decodificador para códigos de Goppa 98
A.4. Funciones auxiliares 101
B. IMPLEMENTACIÓN EN SAGEMATH DEL CRIPTOSISTEMA DE MCELIECE 105
B.1. Clase para el criptosistema McEliece 105
B.2. Funciones auxiliares 109
C. IMPLEMENTACIÓN EN SAGEMATH DE LOS ALGORITMOS GENÉTICOS 111
C.1. Algoritmo genético GGA 111
C.2. Algoritmo genético CHC 112
C.3. Funciones auxiliares 113
Bibliografía 118

INTRODUCCIÓN Y OBJETIVOS

CONTEXTO

El inicio de la teoría de información surgió a partir de la publicación de Claude Shannon sobre "Una teoría matemática sobre la comunicación" en 1948 [17]. En este artículo, Shannon explica que es posible transmitir mensajes fiables en un canal de comunicación que puede corromper la información enviada a través de él siempre y cuando no se supere la capacidad de dicho canal. En dicho artículo pone como ejemplo la corrección de un código de Hamming, una construcción de Richard Hamming. Dos años más tarde, Hamming publica "Error Detecting and Error Correcting Codes" [11] y con dicha publicación aparece el inicio de la teoría de códigos. La motivación de este artículo surgió a partir de la preocupación sobre los errores producidos en los ordenadores al operar con grandes números. Es por esto que Hamming expone la importancia de detectar y corregir los errores que se puedan producir y, además, propone algunas soluciones.

Con la teoría de códigos, podemos codificar datos antes de transmitirlos de tal forma que los datos alterados puedan ser decodificados al grado de precisión especificado. Así, el principal problema es determinar el mensaje que fue enviado a partir del recibido. El Teorema de Shannon nos garantiza que el mensaje recibido coincidirá con el que fue enviado un cierto porcentaje de las veces. Esto hace que el objetivo de la teoría de códigos sea crear códigos que cumplan las condiciones de este teorema.

Supongamos que queremos enviar un mensaje, por lo que habrá un emisor y un receptor que se comunican, en general, en una dirección. Este mensaje es una secuencia finita de elementos de un alfabeto dado y es enviado por un *canal de comunicación*, a través del cual es posible que la información se altere por las interferencias y el ruido, lo que se conoce como *ruido del canal*. Es por esto que hay que hacer una *traducción* entre el mensaje original (o *palabra fuente*) x y el tipo de mensaje c que el canal está capacitado para enviar (*palabras código*). Esta manipulación consiste en proteger el mensaje original, ya sea por ejemplo añadiendo redundancia o repitiéndolo, para que posteriormente se pueda corregir el ruido hasta cierto punto. Este proceso se llama *codificación*. Una vez codificado el mensaje, lo enviamos a través del canal, y nuestro intermediario (el receptor) recibe un mensaje codificado (*palabra recibida*) posiblemente erróneo. Una vez recibido, empieza el proceso llamado *corrección de errores*, que consiste en recuperar el mensaje original corrigiendo los errores que se hubieran producido. El mensaje recibido c' es traducido nuevamente a términos originales x', es decir, es *decodificado*. La siguiente figura representa un esquema de este proceso.

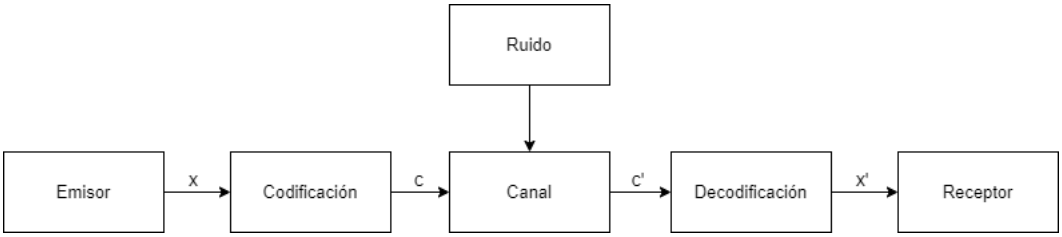


Figura 1: Esquema del modelo de comunicación, [16].

Las flechas indican que la comunicación es en un solo sentido.

En general, $x' \neq x$ y es deseable que este error sea detectado (lo cual permite pedir una retransmisión del mensaje) y en lo posible corregido.

La *Teoría de Códigos Autocorrectores* se ocupa del segundo y cuarto pasos del esquema anterior, es decir, de la codificación y decodificación de mensajes, junto con el problema de detectar y corregir errores. A veces no es posible pedir retransmisión de mensajes y es por eso que los códigos autocorrectores son tan útiles y necesarios.

La calidad de un código con mensajes de longitud k y palabras código de longitud n vendrá dada por las siguientes características.

- El cociente $\frac{k}{n}$, el *ratio de información* del código, que mide el esfuerzo necesario para transmitir un mensaje codificado.
- La *distancia mínima relativa d* que es aproximadamente el doble de la proporción de errores que se pueden corregir en cada mensaje codificado.
- La *complejidad* de los procedimientos de codificar y decodificar.

De esta forma, uno de los objetivos centrales de la teoría de códigos autocorrectores es construir códigos que sean de calidad. Esto es, códigos que permitan codificar muchos mensajes, que se puedan trasmitir rápida y eficientemente, que detecten y corrijan simultáneamente la mayor cantidad de errores posibles y que haya algoritmos de decodificación eficientes y efectivos. Por lo que habrá que encontrar un balance entre estas distintas metas, pues suelen ser contradictorias entre sí.

DESCRIPCIÓN DEL TRABAJO

El objetivo principal de este trabajo se basa en emplear algoritmos genéticos para realizar un análisis del criptosistema de McEliece clásico. En consecuencia, haremos uso de las técnicas y áreas de las matemáticas, tales como los cuerpos finitos, los anillos de polinomios sobre cuerpos finitos y la complejidad de los problemas y de los algoritmos. En cuanto a las herramientas informáticas, hablaremos de las metaheurísticas basadas en poblaciones, en concreto los algoritmos genéticos. A lo largo del desarrollo de este trabajo, ilustraremos en los ejemplos las implementaciones realizadas en el sistema algebraico computacional SageMath.

CONTENIDO DEL TRABAJO

El contenido de este trabajo comienza con un capítulo dedicado a describir y desarrollar las herramientas matemáticas e informáticas que necesitaremos para facilitar el seguimiento del posterior desarrollo. Estudiaremos los conceptos básicos relacionados con la teoría de códigos lineales, que nos permitirá introducir las bases para que en el siguiente capítulo podamos desarrollar los códigos de Goppa. A partir de estos códigos, podremos construir el criptosistema de McEliece clásico y emplear los algoritmos genéticos para concluir con su respectivo análisis.

OBJETIVOS DEL TRABAJO

Los objetivos de este trabajo son los siguientes.

- Estudiar la teoría básica de códigos lineales.
- Estudiar los códigos de Goppa y su decodificación.
- Estudiar la criptografía basada en códigos como modelo de criptografía post-cuántica.
- Estudiar e implementar el criptosistema de McEliece.
- Estudiar e implementar algoritmos evolutivos para el cálculo de la distancia de un código lineal.
- Estudiar el criptoanálisis del criptosistema de McEliece mediante algoritmos evolutivos.

1

PRELIMINARES

En este capítulo se desarrollarán las herramientas necesarias para poder afrontar el criptosistema de McEliece que precisa este trabajo. Empezaremos abordando la idea de anillo y algunas de sus propiedades más importantes. A partir de este noción, podremos definir los cuerpos finitos y los conceptos relacionados con ellos, tales como los anillos de polinomios sobre cuerpos finitos y su utilidad para construir cuerpos finitos. También será de especial relevancia tratar los conceptos relacionados con la teoría de la complejidad para clasificar los problemas de acuerdo a su dificultad. Finalmente, estudiaremos las metaheurísticas basadas en poblaciones para introducir los algoritmos genéticos, que podrán ser de utilidad para realizar un ataque al criptosistema de McEliece.

1.1 ANILLOS

En esta sección introduciremos el concepto de anillo para poder definir el concepto de cuerpo, así como las principales propiedades de esta estructura algebraica.

Definición 1. Un *anillo* $(A, +, \cdot)$ es un conjunto A junto con dos operaciones binarias $A \times A \to A$ denotadas por suma $(+)$ y producto $(\cdot)$ que verifican los siguientes axiomas:

- Propiedad asociativa de la suma:

$$a + (b + c) = (a + b) + c \qquad \forall a, b, c \in A$$

- Existencia del elemento neutro para la suma:

$$0 + a = a = a + 0 \qquad \forall a \in A$$

- Existencia del elemento inverso para la suma:

$$\forall a \in A \;\; \exists - a \in A \quad a + (-a) = 0 = (-a) + a$$

- Propiedad conmutativa de la suma:

$$a + b = b + a \qquad \forall a, b \in A$$

- Propiedad asociativa del producto:

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \qquad \forall a, b, c \in A$$

- Propiedad distributiva del producto:

$$a \cdot (b + c) = a \cdot b + a \cdot c, \quad (b + c) \cdot a = b \cdot a + c \cdot a \qquad \forall a, b, c \in A$$

Un anillo se llama *conmutativo o abeliano* si se verifica la propiedad conmutativa del producto

$$ab = ba \qquad \forall a, b \in A$$

Es fácil comprobar que en todo anillo se verifica $0 \cdot x = x \cdot 0 = 0$ (obsérvese que $0 \cdot x = (0 + 0) \cdot x = 0 \cdot x + 0 \cdot x$ y simplifíquese por el simétrico de $0 \cdot x$).

Además del anillo conmutativo, existen otros casos particulares de anillos. Diremos que un anillo es *unitario* si es un anillo cuyo producto tiene elemento neutro, esto es, $\exists 1 \in A \ : \ x \cdot 1 = 1 \cdot x = x$, para todo $x \in A$.

Diremos que un elemento a del anillo A es *invertible* si existe un elemento a' en el anillo A tal que $a \cdot a' = a' \cdot a = 1$. Este elemento a' es único, lo llamaremos *elemento inverso* y lo denotaremos por a^{-1}.

Cuando se da la igualdad $1 = 0$, diremos que el anillo es *trivial* y tendrá un solo elemento.

Definición 2. Sea A un anillo, $1 \in A$ el elemento neutro del producto y $n \geq 1$ un número natural, definimos la característica de A como:

$$Car(A) = \begin{cases} 0 & \text{si } n \cdot 1 \neq 0 \text{ para cualquier } n \geq 1 \\ \\ n & \text{si } n \text{ es el menor número natural no nulo para el que } n \cdot 1 = 0 \end{cases}.$$

Ejemplo 1. Veamos algunos ejemplos de anillos.

- Los conjuntos $\mathbb{Z}, \mathbb{R}, \mathbb{Q}$ y $\mathbb{C}$ con las operaciones de suma y producto usuales son anillos conmutativos.
- El conjunto M de las matrices reales de orden 2 con las operaciones de adición y multiplicación de matrices es un anillo no conmutativo.

- Sea $A = \{a\}$ un conjunto con un único elemento. La suma y el producto coinciden, pues solo hay una operación binaria posible: $a + a = a = aa$ y $0 = a = 1$. Este conjunto con estas operaciones es un anillo trivial y es el más pequeño posible.

1.2 CUERPOS FINITOS

Para presentar el concepto de cuerpo finito, necesitaremos definir previamente el concepto de cuerpo junto con algunas de sus propiedades más relevantes.

Definición 3. Un *cuerpo* $(K, +, \cdot)$ es un anillo conmutativo no trivial en el que todo elemento no nulo tiene un inverso multiplicativo. Se dice que un cuerpo es *finito* si tiene un número finito de elementos.

Sea $(K, +, \cdot)$ un cuerpo y $E \subset K$, diremos que E es un *subcuerpo* de K o que K es una *extensión* de E si se cumple que $(E, +, \cdot)$ es un cuerpo cuando las operaciones $+$ y $\cdot$ se restringen a E.

Diremos que la *característica* de un cuerpo finito $\mathbb{F}_q$ con $q = p^t$ es p, con p primo.

Todos los cuerpos finitos tienen un número de elementos $q = p^n$, para algún número primo p y algún entero positivo n. Denotaremos por $\mathbb{F}_q$ al cuerpo finito de q elementos, aunque otra notación común es $GF(q)$.

Ejemplo 2. El cuerpo finito $\mathbb{F}_{3^4}$ tiene 81 elementos y su característica es 3.

Observemos que si p un número primo y q es un número entero tal que $q = p^n$, entonces $\mathbb{F}_q$ es un espacio vectorial sobre $\mathbb{F}_p$ de dimensión n. Además, hay q vectores en el espacio vectorial de dimensión n sobre $\mathbb{F}_p$.

Notemos que todos los cuerpos finitos de orden q son isomorfos, aunque cada cuerpo puede tener diferentes representaciones.

Proposición 1. *Sea $\mathbb{F}_q$ un cuerpo finito con $q = p^n$ elementos, entonces*

$$p \cdot \alpha = 0, \qquad \forall \alpha \in \mathbb{F}_q.$$

Demostración. La característica del cuerpo finito $\mathbb{F}_q$ es p, luego $p1 = 0$, donde 1 es el elemento neutro para la multiplicación. Por la propiedad distributiva tenemos que

$$p \cdot \alpha = \underbrace{\alpha + \cdots + \alpha}_{p \text{ veces}} = a \underbrace{(1 + \cdots + 1)}_{p \text{ veces}} = a0 = 0.$$

□

Proposición 2. *Sea $\mathbb{F}_q$ un cuerpo finito con característica p, se cumple que*

$$(\alpha+\beta)^p = \alpha^p + \beta^p, \qquad \forall \alpha, \beta \in \mathbb{F}_q.$$

Demostración. Por el Teorema del binomio, podemos desarrollar la n-ésima potencia de un binomio. En este caso tenemos que

$$(\alpha+\beta)^p = \sum_{k=0}^{p} \binom{p}{k} \alpha^{p-k}\beta^k = \binom{p}{0}\alpha^p + \binom{p}{1}\alpha^{p-1}\beta + \cdots + \binom{p}{p-1}\alpha\beta^{p-1} + \binom{p}{p}\beta^p. \quad (1)$$

Por otra parte, sabemos que

$$\binom{p}{i} = \frac{n!}{i!(p-i)!}.$$

En consecuencia, $\binom{p}{i}$ es divisible por p para $1 \leq i \leq p-1$.

Por lo tanto, los coeficientes de la parte de la derecha de (1), salvo el primero y el último, son múltiplos de p. Por 1, estos términos se anulan y concluimos que

$$(\alpha+\beta)^p = \alpha^p + \beta^p.$$

□

1.2.1 *Anillo de polinomios sobre cuerpos finitos*

En esta sección vamos a introducir el concepto de polinomio junto con sus operaciones.

Definición 4. Sea A un anillo conmutativo. El *conjunto de polinomios* en la variable x con coeficientes en A está compuesto por el siguiente conjunto

$$A[x] := \{a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \ : \ a_0, ..., a_n \in A\}.$$

En el conjunto de polinomios definimos una suma y un producto.

Sean $f = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ y $g = b_m x^m + b_{m-1}x^{m-1} + \cdots + b_1 x + b_0$ dos polinomios. Supongamos que $m \leq n$, tomando $b_i = 0$ para todo $n \geq i > m$, definimos las operaciones de suma y producto de polinomios:

$$f + g = (a_n + b_n)x^n + (a_{n-1} + b_{n-1})x^{n-1} + \cdots + (a_1 + b_1)x + (a_0 + b_0).$$
$$f \cdot g = a_n b_m x^{n+m} + (a_n b_{m+1} + a_{n-1}b_m)x^{n+m-1} + \cdots + (a_1 b_0 + a_0 b_1)x + a_0 b_0.$$

De esta forma, diremos que el conjunto $A[x]$ con las operaciones anteriores es un *anillo de polinomios en X con coeficientes en A*.

Definición 5. Para un polinomio $f = a_n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0 \neq 0$ el mayor índice n tal que $a_n \neq 0$ se llama *grado de f* y se representa por $\mathrm{gr}(f)$. Si $f = 0$ definimos $\mathrm{gr}(f) = -\infty$.

Llamaremos *término (de grado i)* a cada uno de los sumandos a_iX^i del polinomio f. El *término líder* es el término no nulo de mayor grado. El coeficiente $a_n \neq 0$ del término líder se llama *coeficiente líder* y el término de grado cero a_0 se llama *término constante*. Si el coeficiente líder es 1, diremos que el polinomio es *mónico*.

A continuación tenemos algunas propiedades de los polinomios.

Proposición 3. *Sea A un anillo conmutativo y sean $f, g \in A[x]$ dos polinomios, tenemos que*

$$\mathrm{gr}(f+g) \leq \mathrm{máx}\,(\mathrm{gr}(f),\ \mathrm{gr}(g)),$$
$$\mathrm{gr}(f \cdot g) \leq \mathrm{gr}(f) + \mathrm{gr}(g)$$

Si $\mathrm{gr}(f) \neq \mathrm{gr}(g)$*, se verifica*

$$\mathrm{gr}(f+g) = \mathrm{máx}\,(\mathrm{gr}(f),\ \mathrm{gr}(g))$$

Sean $f(x)$ y $g(x)$ polinomios en $\mathbb{F}_q[x]$, diremos que $f(x)$ divide a $g(x)$ si existe un polinomio $h(x) \in \mathbb{F}_q[x]$ tal que $g(x) = f(x)h(x)$ y lo denotaremos por $f(x)|g(x)$. El polinomio $f(x)$ se llama *divisor* o *factor* de $g(x)$.

El *máximo común divisor* de $f(x)$ y $g(x)$ es el polinomio mónico en $\mathbb{F}_q[x]$ con mayor grado que divide a $f(x)$ y a $g(x)$. Este polinomio es único y se denota por $\mathrm{mcd}(f(x),\ g(x))$. Diremos que los polinomios $f(x)$ y $g(x)$ son *primos relativos* si se satisface que $\mathrm{mcd}(f(x),\ g(x)) = 1$.

El siguiente resultado es de gran utilidad, pues sirve para calcular los divisores de un polinomio e incluso para calcular el máximo común divisor. Nos dará las bases para definir posteriormente el Algoritmo de Euclides.

Teorema 1. *Sean $f(x)$ y $g(x)$ polinomios en $\mathbb{F}_q[x]$ con $g(x)$ no nulo.*

- *Existen dos polinomios únicos $c(x), r(x) \in \mathbb{F}_q[x]$ tales que*

$$f(x) = g(x)c(x) + r(x), \qquad \textit{donde } \mathrm{gr}(r(x)) < \mathrm{gr}(g(x)) \textit{ o } r(x) = 0.$$

- *Si $f(x) = g(x)c(x) + r(x)$, entonces* $\mathrm{mcd}(f(x),\ g(x)) = \mathrm{mcd}(g(x),\ r(x))$.

Los polinomios $c(x)$ y $r(x)$ se llaman *cociente* y *resto*, respectivamente.

Usando este resultado de forma recursiva, obtendremos el máximo común divisor de los polinomios $f(x)$ y $g(x)$. Este procedimiento se conoce como *Algoritmo de Euclides*. El siguiente resultado describe este algoritmo.

Teorema 2 (Algoritmo de Euclides)**.** *Sean $f(x)$ y $g(x)$ polinomios definidos en $\mathbb{F}_q[x]$ con $g(x)$ no nulo.*

1. *Realizar los siguientes pasos hasta que $r_n(x) = 0$ para algún n:*

$$\begin{aligned}
f(x) &= g(x)c_1(x) + r_1(x), && \text{donde } \operatorname{gr}(r_1(x)) < \operatorname{gr}(x),\\
g(x) &= r_1(x)c_2(x) + r_2(x), && \text{donde } \operatorname{gr}(r_2(x)) < \operatorname{gr}(r_1),\\
r_1(x) &= r_2(x)c_3(x) + r_3(x), && \text{donde } \operatorname{gr}(r_3(x)) < \operatorname{gr}(r_2),\\
&\vdots\\
r_{n-3}(x) &= r_{n-2}(x)c_{n-1}(x) + r_{n-1}(x), && \text{donde } \operatorname{gr}(r_{n-1}(x)) < \operatorname{gr}(r_{n-2}),\\
r_{n-2}(x) &= r_{n-1}(x)c_n(x) + r_n(x), && \text{donde } r_n(x) = 0.
\end{aligned}$$

   Entonces $\operatorname{mcd}(f(x),\ g(x)) = cr_{n-1}(x)$*, donde $c \in \mathbb{F}_q$ es una constante para que $cr_{n-1}(x)$ sea mónico.*
2. *Existen polinomios $a(x), b(x) \in \mathbb{F}_q[x]$ tales que*

$$a(x)f(x) + b(x)g(x) = \operatorname{mcd}(f(x),\ g(x)).$$

En cada paso el grado del resto se decrementa al menos en 1, por lo que podemos asegurar que la secuencia de pasos anterior terminará en algún momento.

A continuación se muestran algunos resultados relevantes.

Proposición 4. *Sean $f(x)$ y $g(x)$ polinomios en $\mathbb{F}_q[x]$.*

- *Si $k(x)$ es un divisor de $f(x)$ y $g(x)$, entonces $k(x)$ es un divisor de $a(x)f(x) + b(x)g(x)$ para algunos $a(x), b(x) \in \mathbb{F}_q[x]$.*
- *Si $k(x)$ es un divisor de $f(x)$ y $g(x)$, entonces $k(x)$ es un divisor de* $\operatorname{mcd}(f(x),\ g(x))$.

Proposición 5. *Sea $f(x)$ un polinomio en $\mathbb{F}_q[x]$ de grado n.*

- *Si $\alpha \in \mathbb{F}_q$ es una raíz de $f(x)$, entonces $x - \alpha$ es un factor de $f(x)$.*
- *El polinomio $f(x)$ tiene como mucho n raíces en cualquier cuerpo que contenga a $\mathbb{F}_q$.*

Teorema 3. *Los elementos de $\mathbb{F}_q$ son las raíces de $x^q - x$.*

1.2.1.1 *Construcción de cuerpos finitos*

Para realizar la construcción de cuerpos finitos, previamente necesitaremos conocer el siguiente concepto y algunos resultados relacionados.

Definición 6. Sea $f(x) \in \mathbb{F}_q[x]$ un polinomio no constante, decimos que es *irreducible* sobre $\mathbb{F}_q$ si no se factoriza como producto de dos polinomios en $\mathbb{F}_q[x]$ de menor grado.

Teorema 4. *Sea $f(x)$ un polinomio no constante. Entonces*

$$f(x) = p_1(x)^{a_1} \cdots p_k(x)^{a_k},$$

donde cada $p_i(x)$ es irreducible y único salvo orden, y los elementos a_i son únicos.

Como consecuencia de este resultado, tenemos que $\mathbb{F}_q[x]$ es un *dominio de factorización única*.

El siguiente resultado nos muestra cómo construir un cuerpo finito de característica p a partir del cociente de anillos de polinomios por polinomios irreducibles.

Proposición 6. *Sea p un número primo y sea el polinomio $f(x) \in \mathbb{F}_p[x]$ irreducible en $\mathbb{F}_p$ y de grado m. Tenemos que el anillo cociente $\mathbb{F}_q[x]/\,(f(x))$ es un cuerpo finito con $q = p^m$ elementos, es decir, con característica p.*

Escribiremos los elementos del anillo cociente, que son las clases laterales $g(x) + (f(x))$ como vectores en $\mathbb{F}_p^m$ con la siguiente correspondencia:

$$g_{m-1}x^{m-1} + g_{m-2}x^{m-2} + \cdots + g_1x + g_0 + (f(x)) \leftrightarrow (g_{m-1}, g_{m-2}, \ldots, g_1, g_0). \tag{2}$$

Esta notación facilita la operación de sumar dos elementos. Sin embargo, la multiplicación es algo más complicada. Supongamos que queremos multiplicar $g_1(x) + (f(x))$ por $g_2(x) + (f(x))$. Para ello, usaremos el resultado 1 y obtenemos

$$g_1(x)g_2(x) = f(x)h(x) + r(x), \tag{3}$$

donde $\operatorname{gr}(r(x)) \leq m - 1$ o $r(x) = 0$. Entonces

$$(g_1(x) + (f(x)))(g_2(x) + (f(x))) = r(x) + (f(x)).$$

Podemos simplificar la notación si reemplazamos x por α donde $f(\alpha) = 0$. Por (3), se cumple que $g_1(\alpha)g_2(\alpha) = r(\alpha)$ y la correspondencia (2) queda como sigue

$$g_{m-1}g_{m-2}\cdots g_1g_0 \leftrightarrow g_{m-1}\alpha^{m-1}g_{m-2}\alpha^{m-2}\cdots g_1\alpha g_0.$$

De esta forma, multiplicamos los polinomios en α de forma usual y aplicamos al resultado que $f(\alpha) = 0$ para reducir las potencias de α mayores que $m-1$ a polinomios en α de grado menor que m.

1.3 TEORÍA DE LA COMPLEJIDAD

El objetivo de esta sección se basa en clasificar los problemas de acuerdo a su dificultad. Nos centraremos en estudiar la complejidad de los problemas decidibles, es decir, los problemas que tienen algoritmos para resolverlos.

1.3.1 *Complejidad de los algoritmos*

A la hora de resolver un problema con un algoritmo, tendrán gran importancia los recursos que consume dicho algoritmo, sobre todo el espacio y el tiempo. La complejidad del tiempo de un algoritmo es el número de pasos que realiza la máquina de Turing que define dicho algoritmo. Alternativamente, la complejidad del tiempo de un algoritmo algebraico se puede definir como el número de operaciones básicas sobre el cuerpo base. La complejidad de un algoritmo siempre se define en el peor de los casos, esto es, se obtiene una cota para el recuento de los pasos en el peor de los casos.

La siguiente definición nos proporciona la notación más importante en el análisis de los algoritmos.

Definición 7 (Notación big-$\mathcal{O}$)**.** Sean $f, g : \mathbb{N}^r \longrightarrow \mathbb{R}$. Decimos que $f(n) = O(g(n))$ si existen dos constantes $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}^r$ tales que $\forall n \geq n_0$, $f(n) \leq c \cdot g(n)$.

Con esta notación podemos analizar el tiempo esperado que va a tardar un algoritmo en realizar una tarea.

Proposición 7. *Para cualquier $\lambda \in \mathbb{R}^+$ y $g : \mathbb{N}^r \longrightarrow \mathbb{R}$, tenemos que*

$$\mathcal{O}(g(n)) = \mathcal{O}(\lambda g(n)), \qquad \forall n \in \mathbb{N}^r.$$

A continuación presentaremos algunos conceptos importantes para especificar la complejidad de los algoritmos que nos serán de utilidad más adelante.

Definición 8 (Algoritmo en tiempo polinomial)**.** Un algoritmo es un *algoritmo en tiempo polinomial* si su complejidad es $\mathcal{O}(p(n))$, donde $p(n)$ es una función polinómica de n.

Definición 9 (Algoritmo en tiempo exponencial)**.** Un algoritmo es un *algoritmo en tiempo exponencial* si su complejidad es $\mathcal{O}(c^n)$, donde c es una constante real estrictamente mayor que 1.

1.3.2 *Complejidad de los problemas*

La complejidad de un problema es equivalente a la complejidad del mejor algoritmo que resuelve ese problema. Diremos que un problema es *fácil* si existe un algoritmo en tiempo polinomial que lo resuelve, mientras que un problema será *difícil* si no existe ningún algoritmo en tiempo polinomial que lo resuelva.

La teoría de la complejidad de los problemas se ocupa de los *problemas de decisión*. Un problema de decisión siempre tiene una respuesta de sí o no.

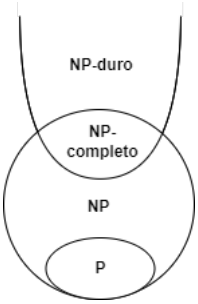


Figura 2: Clases de complejidad de los problemas de decisión.

Existen dos clases importantes de problemas: P y NP (Figura 2).

La clase de complejidad P representa a la familia de problemas de decisión que se pueden resolver mediante un algoritmo en tiempo polinomial. Los problemas que pertenecen a esta clase son *fáciles* de resolver.

La clase de compejidad NP representa a la familia de problemas de decisión que se pueden resolver por un algoritmo no determinista en tiempo polinomial. Estos problemas se dicen que son más *difíciles* de resolver, ya que no se conoce un algoritmo polinomial que los resuelva.

Para definir la clase de los problemas NP-completos, necesitaremos previamente definir el concepto de reducción.

Definición 10 (Reducción polinómica)**.** Un problema de decisión A se *reduce polinómicamente* a otro problema de decisión B si, para todas las instancias de entrada I_A de A, siempre se puede construir una instancia de entrada I_B para B en una función de tiempo polinomial, de tal forma que I_A es una instancia positiva de A si y solo si I_B es una instancia positiva de B.

Un problema de decisión es *NP-completo* si es NP y cualquier otro problema de NP se reduce a él. Los problemas que pertenecen a esta clase son los *más difíciles* dentro de la clase NP.

La figura 2 muestra la relación entre los problemas P, NP y NP-completos.

Los problemas *NP-duros* son problemas para los que existe una reducción desde todo problema NP. Por tanto, los problemas NP-completos son NP-duros, pero lo contrario no tiene

por qué ser cierto. Un problema de optimización cuyo problema de decisión asociado es NP-completo es NP-duro.

1.4 METAHEURÍSTICAS BASADAS EN POBLACIONES

Las *metaheurísticas* representan una familia de técnicas de optimización aproximada. Proporcionan soluciones aceptables en un tiempo razonable para resolver problemas difíciles y complejos, esto es, problemas que no pertenecen a la clase de complejidad P o que no se conoce que pertenezcan a dicha clase. Sin embargo, no garantizan la optimalidad de las soluciones obtenidas.

Las *metaheurísticas basadas poblaciones* se pueden ver como una mejora iterativa en una población de soluciones. Estas metaheurísticas consisten en inicializar la población, posteriormente generan una nueva población de soluciones y finalmente esta nueva población reemplaza a la actual usando algunos procedimientos de selección. Una vez se alcanza la condición dada, el proceso de búsqueda finaliza y obtenemos una solución.

El esquema de este procedimiento se muestra a continuación.

Output: Mejor solución encontrada

1 $t \longleftarrow 0$
2 $P_t \longleftarrow P_0$ // Generación de la población inicial
3 **while** *No se cumpla la condición de parada* **do**
4 Generar(P'_t) // Generación de la nueva población
5 $P_{t+1} \longleftarrow$ SeleccionarPoblación($P_t \cup P'_t$) // Selección de la nueva población
6 $t \longleftarrow t + 1$
7 **end**

Algoritmo 1: Esquema de una metaheurística basada en poblaciones.

Vamos a estudiar en detalle este procedimiento concretando los conceptos en los que se basa.

1. **Generación de la población inicial**. Las metaheurísticas basadas en poblaciones comienzan con una población inicial de soluciones. Este paso juega un papel fundamental en la efectividad del algoritmo y su eficiencia. El principal criterio para generar la población es que haya diversidad.
2. **Generación de una nueva población**. En este paso, se genera una nueva población de soluciones. Se puede realizar de varias formas, sin embargo en este trabajo estudiaremos la que se basa en la evolución. En esta categoría, las soluciones que componen la población se seleccionan y se reproducen usando operadores de variación (por ejemplo, mutación o cruce) y actúan directamente sobre sus representaciones. Una nueva

solución surge a partir de los diferentes atributos de las soluciones de la población actual.

3. **Selección de la nueva población**. Este último paso consiste en seleccionar las nuevas soluciones y reemplazar a la población actual. Este reemplazamiento se puede realizar de varias maneras, por ejemplo una estrategia se basa en seleccionar la nueva población generada como la población actual, otra estrategia se fundamenta en escoger las dos mejores soluciones de la nueva población y reemplazarlas por las dos peores de la población actual, etc.

El criterio de parada suele ser un número máximo de iteraciones (generaciones), un número máximo de evaluaciones de la función objetivo, etc.

1.4.1 *Algoritmos evolutivos*

Los *algoritmos evolutivos* son algoritmos basados en poblaciones. Estos se fundamentan en el concepto de *competición*. Se representan como una clase de algoritmos de optimización iterativos que simulan la evolución de las especies, en este caso, la evolución de la población de soluciones.

Inicialmente, tenemos una población de soluciones que se suele generar aleatoriamente. También tenemos una función objetivo, que es la que queremos optimizar, que evalúa cada individuo perteneciente a la población indicando la idoneidad de pertenencia al problema (*fitness*). En cada paso, los individuos se seleccionan para formar los padres. En esta selección los individuos con mejor fitness tendrán altas probabilidades de ser escogidos. Luego, los individuos seleccionados se reproducen usando operadores de variación (por ejemplo, cruce, mutación) para generar nuevos descendientes, de forma que nos permite explorar el espacio de búsqueda y obtendremos soluciones diversas. Finalmente, el esquema de reemplazamiento se basa en determinar los individuos que sobrevivirán entre los descendientes y los padres. Esta iteración representa una generación. Este proceso es iterativo hasta que se cumpla el criterio de parada.

Cada solución que forma parte de la población se suele llamar *cromosoma*. A su vez, cada cromosoma está formado por *genes*.

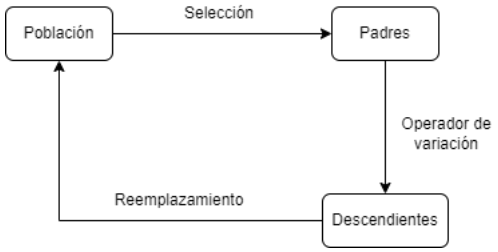


Figura 3: Generación en algoritmos evolutivos.

El algoritmo 2 ilustra este procedimiento.

Output: Mejor solución encontrada

1 $t \longleftarrow 0$
2 $P_t \longleftarrow$ Generar(P_0) // Generación de la población inicial
3 **while** *No se cumpla la condición de parada* **do**
4 Evaluar(P_t)
5 $P'_t \longleftarrow$ Seleccionar(P_t)
6 $P'_t \longleftarrow$ Reproducción(P'_t)
7 Evaluar(P'_t)
8 $P_{t+1} \longleftarrow$ Reemplazar(P_t, P'_t)
9 $t \longleftarrow t + 1$
10 **end**

Algoritmo 2: Esquema de un algoritmo evolutivo.

1.4.1.1 *Algoritmos genéticos*

Los *algoritmos genéticos* son algoritmos de optimización, búsqueda y aprendizaje inspirados en los procesos de evolución natural y evolución genética.

Estos algoritmos suelen aplicar como operador de variación un operador de cruce a las dos soluciones (cromosomas) que juegan un papel importante, además de un operador de mutación que modifica aleatoriamente el contenido del individuo (gen) para fomentar la diversidad. Los algoritmos genéticos usan una selección probabilística. El reemplazamiento puede ser generacional, esto es, los padres serán reemplazados por sus descendientes.

El operador de cruce es un operador binario y, a veces, n-ario. Este operador se basa en adquirir las características de los dos padres y generar un descendiente. La principal característica de este operador es la *heredabilidad*, pues crea un cromosoma a partir de los genes heredados de los padres. Esto es, el descendiente tendrá una recombinación de las características de sus

padres. Los padres cruzarán con una cierta probabilidad p_c. Es importante asegurar que el operador de cruce produce soluciones válidas.

El operador de mutación es un operador unario que actúa sobre un solo individuo. Las mutaciones representan pequeños cambios de los individuos seleccionados. La probabilidad p_m define la probabilidad de mutar cada elemento (gen) de la representación. Este operador debe garantizar que cualquier solución estará en el espacio de búsqueda y que sea válida. Con la mutación conseguimos introducir diversidad en los individuos de una población. Al igual que con el operador de cruce, el de mutación también debe producir soluciones válidas.

El algoritmo 3 ilustra este procedimiento.

Output: Mejor solución encontrada

```
1  t ⟵ 0
2  P_t ⟵ Generar(P_0) // Generación de la población inicial
3  while No se cumpla la condición de parada do
4      Evaluar(P_t)
5      P'_t ⟵ Seleccionar(P_t)
6      P'_t ⟵ OperadorCruce(P'_t)
7      P'_t ⟵ OperadorMutación(P'_t)
8      P_{t+1} ⟵ Reemplazar(P_t, P'_t)
9      t ⟵ t + 1
10 end
```

Algoritmo 3: Esquema de un algoritmo genético.

El modelo puede ser elitista (conserva a los mejores individuos), lo que produce una convergencia rápida cuando se reemplazan los peores cromosomas de la población.

2

INTRODUCCIÓN A LA TEORÍA DE CÓDIGOS LINEALES

En este capítulo introduciremos los conceptos y resultados fundamentales sobre la teoría de códigos lineales. Empezaremos con la definición básica de código para estudiar posteriormente la idea de código lineal y algunas de sus propiedades más relevantes, junto con otras nociones relacionadas. Finalmente, introduciremos el algoritmo de Brouwer-Zimmermann, que permite calcular la distancia de un código lineal, muy importante en la teoría de códigos.

El desarrollo de este capítulo se ha basado, principalmente, en [12, Capítulo 1].

2.1 CÓDIGOS LINEALES

Sea $\mathbb{F}_q$ el cuerpo finito con q elementos, denotamos por $\mathbb{F}_q^n$ el espacio vectorial de las n-tuplas sobre el cuerpo finito $\mathbb{F}_q$. Generalmente los vectores $(a_1, ..., a_n)$ de $\mathbb{F}_q^n$ se denotarán por $a_1 \cdots a_n$.

Definición 11. Un (n, M) *código* $\mathcal{C}$ sobre $\mathbb{F}_q$ es un subconjunto de $\mathbb{F}_q^n$ de tamaño M. A los elementos de $\mathcal{C}$ los llamaremos *palabras código*.

Ejemplo 3.

- Un código sobre $\mathbb{F}_2$ se llama *código binario* y un ejemplo es $\mathcal{C} = \{00, 01, 10, 11\}$.
- Un código sobre $\mathbb{F}_3$ se llama *código ternario* y un ejemplo es $\mathcal{C} = \{21, 02, 10, 20\}$.

Si $\mathcal{C}$ es un subespacio k-dimensional de $\mathbb{F}_q^n$, entonces decimos que $\mathcal{C}$ es un $[n, k]$ *código lineal* sobre $\mathbb{F}_q$. De esta forma, los códigos lineales tendrán q^k palabras código. Al imponer linealidad sobre los códigos, nos permite conseguir algoritmos de codificación y decodificación más eficientes que otros códigos. Estos se pueden presentar con una matriz generadora o con una matriz de paridad.

Definición 12. Una *matriz generadora* para un $[n, k]$ código $\mathcal{C}$ es una matriz $k \times n$ donde sus filas forman una base de $\mathcal{C}$.

Definición 13. Para cada conjunto de k columnas independientes de una matriz generadora G, se dice que el conjunto de coordenadas correspondiente conforman un *conjunto de información* de $\mathcal{C}$. Las $r = n - k$ restantes coordenadas se denominan *conjunto de redundancia* y el número r es la *redundancia* de $\mathcal{C}$.

En general, la matriz generadora no es única pues si realizamos un cambio de base del código podemos obtener otra matriz generadora distinta. Sin embargo, si las k primeras coordenadas conforman un conjunto de información, entonces el código tiene una única matriz generadora de la forma $(I_k|A)$, donde I_k denota a la matriz identidad $k \times k$. Esta matriz se dice que está en *forma estándar*.

Como un código lineal es un subespacio de un espacio vectorial, es el núcleo de alguna transformación lineal.

Definición 14. Una *matriz de paridad* H de dimensión $(n-k) \times n$ de un $[n,k]$ código $\mathcal{C}$ es una matriz que verifica que

$$C = \left\{ \mathbf{x} \in \mathbb{F}_q^n : H\mathbf{x}^T = 0 \right\}.$$

Al igual que con la matriz generadora, la matriz de paridad no es única. Con el siguiente resultado podremos obtener una matriz de paridad cuando $\mathcal{C}$ tiene una matriz generadora en forma estándar.

Teorema 5. *Si $G = (I_k|A)$ es una matriz generadora para un $[n,k]$ código $\mathcal{C}$ en forma estándar, entonces $H = \left(-A^T|I_{n-k}\right)$ es una matriz de paridad de $\mathcal{C}$.*

Demostración. Como $HG^T = -A^T + A^T = 0$, se tiene que $\mathcal{C}$ está contenido en el núcleo de la transformación lineal $x \mapsto Hx^T$. Esta transformación lineal tiene un núcleo de dimensión k, pues H tiene rango $n-k$, que coincide con la dimensión de $\mathcal{C}$. □

Ejemplo 4. Sea la matriz $G = (I_4|A)$, donde

$$G = \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right)$$

es una matriz generadora en forma estándar para un $[7,4]$ código binario que denotaremos por $\mathcal{H}_3$. Por el Teorema 5, una matriz de paridad de $\mathcal{H}_3$ es

$$H = \left(\begin{array}{c|c} -A^T & I_{7-4} \end{array}\right) = \left(\begin{array}{c|c} -A^T & I_3 \end{array}\right) = \left(\begin{array}{cccc|ccc} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}\right)$$

Este código se denomina el $[7,4]$ *código de Hamming*.

2.2 CÓDIGO DUAL

Sabemos que $\mathcal{C}$ es un subespacio de un espacio vectorial, por lo que podemos calcular el subespacio ortogonal a dicho subespacio y así obtener lo que se denomina *espacio dual u ortogonal* de $\mathcal{C}$, denotado por $\mathcal{C}^\perp$. Se define este concepto con la operación del producto escalar como sigue.

Definición 15. El *espacio dual* de $\mathcal{C}$ viene dado por

$$\mathcal{C}^\perp = \left\{ \mathbf{x} \in \mathbb{F}_q^n \ : \ \mathbf{c} \cdot \mathbf{x} = 0 \quad \forall \mathbf{c} \in \mathcal{C} \right\}.$$

El siguiente resultado nos muestra cómo obtener las matrices generadora y de paridad de $\mathcal{C}^\perp$ a partir de las de $\mathcal{C}$.

Proposición 8. *Si tenemos una matriz generadora G y una matriz de paridad H de un código $\mathcal{C}$, entonces H y G son matrices generadoras y de paridad, respectivamente, de $\mathcal{C}^\perp$.*

Demostración. Sea G una matriz generadora y H una matriz de paridad de un código $\mathcal{C}$. Sabemos que $G \cdot H^T = 0$. Por otra parte, tenemos que

$$\mathcal{C}^\perp = \left\{ \mathbf{x} \in \mathbb{F}_q^n \ : \ \mathbf{c} \cdot \mathbf{x} = 0 \quad \forall \mathbf{c} \in \mathcal{C} \right\} = \left\{ \mathbf{x} \in \mathbb{F}_q^n \ : \ G \cdot \mathbf{x}^T = 0 \quad \forall \mathbf{c} \in \mathcal{C} \right\}.$$

Luego la matriz H es una matriz generadora de $\mathcal{C}^\perp$.

Además, como $H \cdot G^T = 0$, entonces G es una matriz de paridad de $\mathcal{C}^\perp$. □

De la proposición anterior se deduce lo siguiente.

Proposición 9. *$\mathcal{C}^\perp$ es un $[n, n-k]$ código.*

Demostración. Sabemos que, por ser G una matriz de paridad de $\mathcal{C}^\perp$,

$$\mathcal{C}^\perp = \left\{ x \in \mathbb{F}_q^k : Gx^T = 0 \right\},$$

o sea $\mathcal{C}^\perp$ es el espacio solución de k ecuaciones con n incógnitas. Luego, como G tiene rango k, hay $n-k$ variables libres, por lo tanto $\dim \mathcal{C}^\perp = n-k$. □

Diremos que un código $\mathcal{C}$ es *auto-ortogonal* si $\mathcal{C} \subseteq \mathcal{C}^\perp$ y *auto-dual* cuando $\mathcal{C} = \mathcal{C}^\perp$.

Ejemplo 5. Una matriz generadora para el $[7,4]$ código de Hamming $\mathcal{H}_3$ se presenta en el Ejemplo 4. Sea $\mathcal{H}'_3$ el código de longitud 8 y dimensión 4 obtenido de $\mathcal{H}_3$ añadiendo una

coordenada de verificación de paridad general a cada vector de G y por lo tanto a cada palabra código de $\mathcal{H}_3$. Entonces

$$G' = \left(\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}\right)$$

es una matriz generadora para $\mathcal{H}'_3$. Además, veamos que $\mathcal{H}'_3$ es un código auto-dual.

Tenemos que $G' = (I_4|A')$, donde

$$A' = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}.$$

Como $A'(A')^T = I_4$, entonces $\mathcal{H}'_3$ es auto-dual.

2.3 PESOS Y DISTANCIAS

A la hora de corregir errores es importante establecer una medida que nos establezca cuánto de diferentes son las palabras enviadas y recibidas. En este apartado estudiaremos esta idea y cómo puede influir a la teoría de códigos.

Definición 16. La *distancia de Hamming* $\mathrm{d}(\mathbf{x}, \mathbf{y})$ entre dos vectores $\mathbf{x}, \mathbf{y} \in \mathbb{F}_q^n$ se define como el número de coordenadas en las que $\mathbf{x}$ e $\mathbf{y}$ difieren.

Ejemplo 6. Sean $\mathbf{x} = 012$, $\mathbf{y} = 210$ $\mathbf{x}, \mathbf{y} \in \mathbb{F}_3^4$. Entonces la distancia de Hamming entre los dos vectores es $d(\mathbf{x}, \mathbf{y}) = 2$.

Teorema 6. *La función distancia* $\mathrm{d}(\mathbf{x}, \mathbf{y})$ *satisface las siguientes propiedades.*

1. *No negatividad:* $\mathrm{d}(\mathbf{x}, \mathbf{y}) \geq 0 \quad \forall \mathbf{x}, \mathbf{y} \in \mathbb{F}_q^n$.
2. $\mathrm{d}(\mathbf{x}, \mathbf{y}) = 0 \Leftrightarrow \mathbf{x} = \mathbf{y}$.
3. *Simetría:* $\mathrm{d}(\mathbf{x}, \mathbf{y}) = \mathrm{d}(\mathbf{y}, \mathbf{x}) \quad \forall \mathbf{x}, \mathbf{y} \in \mathbb{F}_q^n$.
4. *Desigualdad triangular:* $\mathrm{d}(\mathbf{x}, \mathbf{z}) \leq \mathrm{d}(\mathbf{x}, \mathbf{y}) + \mathrm{d}(\mathbf{y}, \mathbf{z}) \quad \forall \mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{F}_q^n$

Demostración. Las tres primeras afirmaciones se obtienen directamente a partir de la definición. La cuarta propiedad se obtiene a partir de la no negatividad. Esto es, sean $x, y, z \in \mathbb{F}_q^n$ distingamos dos casos. Si $x \neq z$ no puede ocurrir que $x = y = z$, luego $y \neq x$ o $y \neq z$. Es decir,

se cumple que $d(x,y) \neq 0$ o $d(y,z) \neq 0$. Entonces por la no negatividad se da la desigualdad. En el caso en el que $x = z$, tendríamos que $d(x,z) = 0$ y también se da la afirmación. □

Diremos que la *distancia mínima* de un código $\mathcal{C}$ es la distancia más pequeña de todas las distancias entre dos palabras distintas del código. Esta medida es fundamental a la hora de determinar la capacidad de corregir errores de $\mathcal{C}$.

Ejemplo 7. Sea $\mathcal{C} = \{010101, 212121, 111000\}$ un código ternario. Entonces

$$d(010101, 212121) = 3, \qquad d(010101, 111000) = 4, \qquad d(212121, 111000) = 5.$$

Por lo que la distancia mínima del código $\mathcal{C}$ es $d(\mathcal{C}) = 3$.

Teorema 7 (Decodificación de máxima verosimilitud)**.** *Es posible corregir hasta*

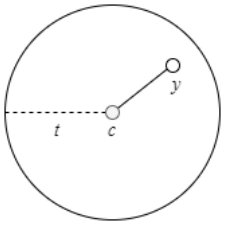
$$t := \left\lfloor \frac{d(\mathcal{C}) - 1}{2} \right\rfloor$$

errores, donde $d(\mathcal{C})$ denota la distancia mínima del código $\mathcal{C}$.

Demostración. Usando la decodificación de máxima verosimilitud, un vector $y \in \mathbb{F}^n$ es decodificado en una palabra código $c \in \mathcal{C}$, que es cercana a y con respecto a la distancia de Hamming. Formalmente, y es decodificado en una palabra código $c \in \mathcal{C}$ tal que $d(c,y) \leq d(c',y)$, $\forall c' \in \mathcal{C}$. Si hay varios $c \in \mathcal{C}$ con esta propiedad, se elige uno arbitrariamente.

Si la palabra código $c \in \mathcal{C}$ fue enviada y no han ocurrido más de t errores durante la transmisión, el vector recibido es

$$y = c + e \in \mathbb{F}^n,$$



donde e denota al vector error.

Esto satisface

$$d(c,y) = d(e,0) \leq t,$$

y por lo tanto c es el único elemento de $\mathcal{C}$ que se encuentra en una bola de radio t alrededor de y. Un decodificador de máxima verosimilitud produce este elemento c, y así se obtiene el código correcto. □

Definición 17. El *peso de Hamming* $\mathrm{wt}(\mathbf{x})$ de un vector $\mathbf{x} \in \mathbb{F}_q^n$ se define como el número de coordenadas no nulas en $\mathbf{x}$.

Ejemplo 8. Sea $\mathbf{x} = 2001021 \in \mathbb{F}_3^7$ un vector, entonces su peso de Hamming es $wt(\mathbf{x}) = 4$.

El siguiente resultado nos muestra la relación entre la distancia y el peso.

Teorema 8. *Si* $\mathbf{x}, \mathbf{y} \in \mathbb{F}_q^n$, *entonces* $\mathrm{d}(\mathbf{x}, \mathbf{y}) = \mathrm{wt}(\mathbf{x} - \mathbf{y})$. *Si* $\mathcal{C}$ *es un código lineal, entonces la distancia mínima d coincide con el peso mínimo de las palabras código no nulas de* $\mathcal{C}$.

Demostración. Sean $x, y \in \mathbb{F}_q^n$, por la definición de distancia de Hamming tenemos que $d(x, y) = wt(x - y)$. Se supone ahora que C es un código lineal, luego para todo $x, y \in \mathcal{C}$, $x - y \in \mathcal{C}$, luego para cualquier par de elementos $x, y \in \mathcal{C}$, existe $z \in \mathcal{C}$ tal que $d(x, y) = wt(z) \geq wt(\mathcal{C})$, donde $wt(\mathcal{C})$ es el peso mínimo de $\mathcal{C}$. Por tanto, $d \geq wt(\mathcal{C})$. Por otro lado, para todo $x \in \mathcal{C}$, se tiene que $wt(x) = d(x, 0)$. Como $\mathcal{C}$ es lineal, $0 \in \mathcal{C}$, luego $d(x, 0) \geq d$. Entonces, $wt(\mathcal{C}) \geq d$. Se concluye que $wt(\mathcal{C}) = d$, como se quería. □

Como consecuencia de este teorema, para códigos lineales, la distancia mínima también se denomina *peso mínimo* de un código. Si se conoce el peso mínimo d de un $[n, k]$ código, se dice entonces que es un $[n, k, d]_q$ código.

En el artículo [22] se ha demostrado que el problema de calcular la distancia mínima de un código lineal binario es NP-duro, y el problema de decisión correspondiente es NP-completo.

Problema 1 (Distancia mínima)**.** *Dada una matriz binaria* H *de dimensión* $m \times n$ *y un número entero* $w > 0$. *¿Existe un vector no nulo* $x \in \mathbb{F}_2^n$ *de peso menor que w tal que* $Hw^T = 0$*?*

El problema 1 es un problema NP-completo. Veamos un esquema de la demostración de este resultado. Para ello, haremos uso de una transformación polinomial del problema de Decodificación de Máxima Verosimilitud al problema de Distancia Mínima.

Teorema 9 (Decodificación de Máxima Verosimilitud)**.** *Dados una matriz binaria* H *de dimensión* $m \times n$, *un vector* $s \in \mathbb{F}_2^m$ *y un número entero* $w > 0$. *Saber si existe un vector* $x \in \mathbb{F}_2^n$ *de peso menor o igual que w tal que* $Hx^t = s$ *es un problema NP-completo.*

Demostración. Veamos que este problema es NP-completo reduciendo el problema del emparejamiento tridimensional [1].

Problema 2 (Emparejamiento Tridimensional)**.** *Dado un subconjunto* $U \subset T \times T \times T$, *donde* T *es un conjunto finito. ¿Existe un conjunto* $W \subset U$ *tal que* $|W| = |T|$, *y que dos elementos de* W *no coincidan en alguna coordenada?*

Antes de reducir este problema, vamos a codificar el conjunto U de 3-tuplas en una matriz de incidencia binaria $|U| \times 3|T|$, en la que cada fila corresponde a una de las 3-tuplas y tiene peso 3, y cada 1 corresponde a una componente de la 3-tupla. En términos de esta matriz, una solución al problema de Emparejamiento Tridimensional es la existencia de $|T|$ filas cuya suma módulo 2 es $111 \cdots 111$.

Supongamos entonces que tenemos un algoritmo en tiempo polinómial para el problema de Decodificación de Máxima Verosimilitud. Ahora, dado un conjunto $U \subset T \times T \times T$ para el problema de Emparejamiento Tridimensional, sea A la matriz de incidencia $|U| \times 3|T|$ descrita anteriormente. Entonces, ejecutando el algoritmo para la Decodificación de Máxima Verosimi-

litud con las entradas A, $y = (111\cdots111)$, $w = |T|$, obtendríamos, en tiempo polinomial, si existen las coincidencias. Esto es, un algoritmo en tiempo polinomial para la Decodificación de Máxima Verosimilitud implica un algoritmo en tiempo polinomial para el Emparejamiento Tridimensional, que a su vez implica un algoritmo en tiempo polinomial para cualquier NP-problema. Esto demuestra que el problema de Decodificación de Máxima Verosimilitud es NP-completo. □

El problema 9 es NP-completo y sigue siendo NP-completo bajo ciertas restricciones, luego reformularemos este problema como la versión para cuerpos finitos de Suma de Subconjuntos, un problema NP-completo conocido. Además, calcular la distancia mínima para la clase de códigos lineales sobre un cuerpo de característica 2 es NP-duro, y el problema de decisión correspondiente Distancia Mínima sobre $GF(2^m)$, abreviado MD_{2^m}, es NP-completo. Luego esta prueba se basa en una transformación polinomial de Decodificación de Máxima Verosimilitud a MD_{2^m}. Sin embargo, esto no prueba que Distancia Mínima sea NP-completo, ya que el posible conjunto de entradas a Distancia Mínima es un pequeño subconjunto del conjunto de posibles entradas a MD_{2^m}. Para ello, se construye una aplicación del código $\mathcal{C}$ definido por la matriz de paridad H sobre $GF(2^m)$ a un código binario $\mathbb{C}$, de tal forma que la distancia mínima de $\mathcal{C}$ puede determinarse a partir de la distancia mínima de $\mathbb{C}$. Dado que la longitud de $\mathbb{C}$ está acotada por la longitud de un polinomio de $\mathcal{C}$, y la aplicación en sí se puede lograr en tiempo polinomial, esto completa la prueba de la NP-completitud de Distancia Mínima.

Definición 18. Sea A_i, también denotada por $A_i(\mathcal{C})$, el número de palabras código con peso i en $\mathcal{C}$. Se dice que la lista A_i para $0 \leq i \leq n$ es la *distribución del peso* o *espectro del peso* de $\mathcal{C}$.

2.4 CLASIFICACIÓN POR ISOMETRÍA

Como hemos visto, las propiedades de un código dependen principalmente de las distancias de Hamming entre sus palabras y entre palabras codificadas y no codificadas. Además, puede ser que un código pueda relacionarse con otro por medio de una aplicación que conserve las distancias de Hamming. De esta forma, podemos definir una relación de equivalencia entre dos códigos que preservan la distancia de Hamming.

Sean C y C' dos $[n,k]_q$ códigos, se dice que son de la *misma cualidad* si existe una aplicación

$$\iota : F_q^n \to F_q^n,$$

con $\iota(C) = C'$ que preserva la distancia de Hamming, es decir,

$$d(w,w') = d(\iota(w),\iota(w')), \qquad \forall w,w' \in H(n,q).$$

Las aplicaciones con la propiedad anterior se llaman *isometrías*.

Definición 19. Dos códigos lineales $C, C' \subseteq H(n,q)$ se llaman *isométricos* si existe una isometría de $H(n,q)$ que aplica C sobre C'.

Las permutaciones de las coordenadas son isometrías, que se denominan *isometrías permutacionales*.

Definición 20. Sea S_n el grupo isométrico en el conjunto $X = n = \{0, ..., n-1\}$. Dos códigos lineales $C, C' \subseteq H(n,q)$ son isométricos permutacionalmente si existe una isometría permutacional de $H(n,q)$ que aplica C sobre C'. Esto es, hay una permutación π en el grupo simétrico S_n tal que

$$C' = \pi(C) = \{\pi(c) : c \in C\}, \quad \text{and} \quad d(c,\tilde{c}) = d(\pi(c), \pi(\tilde{c})), \qquad \forall c, \tilde{c} \in C,$$

donde

$$\pi(c) = \pi(c_0, ..., c_{n-1}) := \left(c_{\pi^{-1}(0)}, ..., c_{\pi^{-1}(n-1)}\right).$$

2.5 ALGORITMO PARA EL CÁLCULO DE LA DISTANCIA

Como hemos visto, la distancia mínima es importante en un código lineal. Sin embargo, calcular este parámetro para un código dado puede resultar laborioso. A continuación presentaremos el algoritmo de Brouwer-Zimmermann (BZ) para el cálculo de la distancia, que tiene eficiencia exponencial [5, Sección 1.8]. Este algoritmo destaca por ser el algoritmo más conocido para calcular la distancia mínima de un código lineal. En particular, si el código es binario se considera eficaz.

Input: $G_1 = (I_k|A_1)$: matriz generadora de $\mathcal{C}$
Output: $\bar{d}_i$: distancia mínima

$m \longleftarrow 2$
$k_1 \longleftarrow k$
while $rango(A_m) \neq 0$ **do**
  Aplicar la eliminación de Gauss y posibles permutaciones de las columnas de la matriz A_{m-1} desde $G_{m-1} = \left(\begin{array}{c|c|c} & I_{k_{m-1}} & A_{m-1} \\ A'_{m-1} & \hline 0 & 0 \end{array} \right)$ para obtener la matriz generadora $G_m = \left(\begin{array}{c|c|c} & I_{k_m} & A_m \\ A'_m & \hline 0 & 0 \end{array} \right)$
end
$C_0 \longleftarrow \{0\}$
$i \longleftarrow 0$
while $\bar{d}_i > \underline{d}_i$ **do**
  $i \longleftarrow i + 1$
  $C_i \longleftarrow C_{i-1} \cup \bigcup_{j=1}^{m} \left\{ v \cdot G_j : v \in \mathbb{F}_q^k,\ wt(v) = i \right\}$
  $\bar{d}_i \longleftarrow \text{mín}\,\{wt(c) : c \in C_i,\ c \neq 0\}$
  $\underline{d}_i \longleftarrow \sum_{\substack{j=1 \\ k-k_j \leq i}}^{m} \left((i+1) - (k-k_j) \right)$
end

Algoritmo 4: Algoritmo de Brouwer-Zimmermann: cálculo de la distancia mínima de un $[n,k]$ código lineal $\mathcal{C}$.

Veamos, en efecto, que este algoritmo determina la distancia mínima. Consideramos $\mathcal{C}$ un $[n,k]_q$ código lineal y G una matriz generadora. Como el código es lineal, existen matrices $M \in M_n(q)$ y $B \in GL_k(q)$ tales que $B \cdot G \cdot M^T$ es una matriz generadora en forma estándar. De hecho, usando el método de Gauss, podemos obtener una matriz generadora en forma estándar mediante operaciones elementales en las filas y permutaciones de las columnas. En este caso para realizar las operaciones elementales en las filas multiplicaremos desde la izquierda por una matriz $B_1 \in GL_k(q)$, y para permutar las columnas multiplicaremos desde la derecha por la traspuesta de una matriz de permutaciones M_{π_1}, de tal forma que

$$G_1 := B_1 \cdot G \cdot M_{\pi_1}^T = (I_{k_1}|A_1)\,,$$

donde $k_1 = k$.

Ahora que tenemos la matriz generadora en forma estándar, necesitaremos tranformarla de nuevo. Si A_1 no es una matriz nula ni vacía, su rango será k_2 con $0 < k_2 \leq k_1$. Aplicando la eliminación de Gauss, podemos obtener k_2 vectores unitarios diferentes en las $n - k_1$ co-

lumnas. Esto es, podemos multiplicar a G_1 desde la izquierda por una matriz $B_2 \in GL_k(q)$ y desde la derecha por la traspuesta de una matriz de permutaciones M_{π_2} con $\pi_2(j) = j$ para $0 \leq j < k_1$, obteniendo

$$G_2 := B_2 \cdot G_1 \cdot M_{\pi_2}^T = \left(\begin{array}{c|c|c} A_2' & I_{k_2} & A_2 \\ \hline & 0 & 0 \end{array} \right).$$

De esta forma, obtenemos que la matriz A_2' es una matriz $k \times k_1$ y la matriz A_2 es una matriz $k_2 \times (n - k_1 - k_2)$. Los ceros indican las matrices nulas.

Una vez que tenemos la matriz generadora de esta forma, realizaremos un procedimiento iterativo aplicando la eliminación de Gauss y posibles permutaciones de las columnas de la matriz A_2 hasta que su rango sea o. Esto es, sea $i \geq 2$, la matriz A_i tiene rango k_{i+1} con $0 < k_{i+1} \leq k_i$. En cada iteración, obtendremos matrices regulares $B_{i+1} \in GL_k(q)$, matrices de permutación $M_{\pi+1} \in M_n(q)$ con $\pi_{i+1}(j) = j$ para $0 \leq j < k_1 + \cdots + k_i$ y matrices generadoras

$$G_{i+1} := B_{i+1} \cdot G_i \cdot M_{\pi_{i+1}}^T = \left(\begin{array}{c|c|c} A_{i+1}' & I_{k_{i+1}} & A_{i+1} \\ \hline & 0 & 0 \end{array} \right),$$

donde A_{i+1}' es una matriz $k \times (k_1 + \cdots + k_i)$ y A_{i+1} es una matriz $k_{i+1} \times (n - k_1 - \cdots - k_{i+1})$. Repitiendo este procedimiento, finalmente obtendremos una matriz generadora G_m tal que

$$G_m = B_m \cdot G_{m-1} \cdot M_{\pi_m}^T = \left(\begin{array}{c|c|c} A_m' & I_{k_m} & A_m \\ \hline & 0 & 0 \end{array} \right),$$

donde A_m es una matriz vacía (que significa que no tiene columnas) o es una matriz nula. Entonces, $k_1 + \cdots + k_m \leq n$ y A_m tiene $n - k_1 - \cdots - k_m$ columnas. En consecuencia, la matriz generadora G_m tiene $n - k_1 - \cdots - k_m$ columnas nulas y por lo cual todos los elementos del código generado por G_m tiene peso menor que $k_1 + \cdots + k_m$.

Definimos para $1 \leq i \leq k$ los siguientes conjuntos:

$$C_i := \bigcup_{j=1}^{m} \left\{ v \cdot G_j : v \in \mathbb{F}_q^k,\ wt(v) \leq i \right\}.$$

Claramente, estos conjuntos forman una cadena ascendente

$$C_1 \subseteq \cdots \subseteq C_k,$$

y por lo tanto los pesos mínimos

$$\bar{d_i} := \text{mín}\,\{wt(c) : c \in C_i,\ c \neq 0\}$$

forman una secuencia decreciente

$$\bar{d}_1 \geq \cdots \geq \bar{d}_k = dist(C).$$

En la mayoría de los casos, no será necesario calcular todos estos valores. En efecto, primero calcularemos $\bar{d}_1$. Después, si $\bar{d}_i$ ya ha sido calculada para algún $i \geq 1$, lo compararemos con $\underline{d}_i$, que es un límite inferior para el peso de los elementos en $C \setminus C_i$. Si $\bar{d}_i \leq \underline{d}_i$, hemos terminado. Si no, repetiremos el mismo proceso para $\bar{d}_{i+1}$.

Para calcular los límites inferiores para los pesos en los complementos $C \setminus C_i$, elegimos un elemento $c \in C \setminus C_i$. Como $c \notin C_i$, entonces existe, para cada j, un vector $v^{(j)} \in \mathbb{F}_q^k$ tal que

$$c = v^{(j)} \cdot G_j, \quad 1 \leq j \leq m, \text{ y } wt(v^{(j)}) \geq i+1.$$

Para estimar el peso de c, nos basaremos en los distintos lugares de información en G_i, es decir, las columnas que contienen la matriz identidad I_{k_j}. Estas son las columnas de índice r para $k_i + \cdots + k_{j-1} \leq r < k_1 + \cdots + k_j$. Nos interesan las k_j coordenadas c_r de $c = v^{(j)} \cdot G_j$ correspondientes a estas k_j columnas. Como $v^{(j)}$ tiene longitud k, estas entradas de c contribuyen al menos el valor de $i + 1 - (k - k_j)$ al peso de c. Además, como los conjuntos son disjuntos, para diferentes j, tenemos que

$$wt(c) \geq \sum_{j=1}^{m} \left(i + 1 - (k - k_j)\right).$$

Restringiendo esta suma a sumandos positivos, obtenemos el límite inferior

$$wt(c) \geq \sum_{j\,:\,k-k_j \leq i} \left(i + 1 - (k - k_j)\right) =: \underline{d}_i.$$

La secuencia de estos límites es incremental, pues el primer sumando es $i + 1$:

$$2 \leq \underline{d}_1 < \cdots < \underline{d}_k.$$

Además,

$$\underline{d}_k = m + \sum_{j=1}^{m} k_j.$$

Finalmente, como $wt(c) \leq k_1 + \cdots + k_m$ para todo $c \in C$, existe un índice i_0 tal que

$$\bar{d}_{i_0} \leq \underline{d}_{i_0}.$$

Para este i_0 se cumple que

$$\bar{d}_{i_0} := \text{mín}\left\{wt(c) : c \in C_{i_0},\ c \neq 0\right\},$$

y se sigue cumpliendo que

$$\underline{d}_{i_0} \leq \text{mín}\left\{wt(c) : c \in C \setminus C_{i_0}\right\}.$$

Por lo que

$$\bar{d}_{i_0} = dist(C),$$

y la palabra código de peso $dist(C)$ se encuentra en C_{i_0}.

Ejemplo 9. Consideramos el $[7,3]$ código binario $\mathcal{C}$ con matriz generadora

$$G_1 = \left(\begin{array}{ccc|cccc} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{array}\right).$$

Aplicaremos el algoritmo de BZ para calcular la distancia mínima de este código.

Observamos que la matriz generadora G_1 ya está en forma estándar y su conjunto de información es $\{0,1,2\}$. Luego lo primero que haremos será aplicar el método de Gauss y permutaciones de las columnas para transformar esta matriz generadora en una matriz generadora de la forma

$$\left(\begin{array}{c|c|c} A_2' & I_{k_2} & A_2 \\ & 0 & 0 \end{array}\right),$$

para alguna matriz A_2' de dimensión $3 \times k_1$ y alguna matriz A_2 de dimensión $k_2 \times (7 - k_1 - k_2)$. En nuestro caso, tenemos que

$$G_2 := \left(\begin{array}{ccc|ccc|c} 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{array}\right).$$

Esta matriz generadora tiene conjunto de información $\{3,4,5\}$. El siguiente paso será realizar de nuevo la eliminación de Gauss y posibles permutaciones de las columnas de la matriz A_2 hasta que su rango sea 0. El algoritmo nos dará la siguiente matriz generadora

$$G_3 := \left(\begin{array}{cccccc|c} 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{array}\right).$$

Esta matriz generadora tiene conjunto de información $\{6\}$. El conjunto C_1 está formado por las filas de las tres matrices generadoras G_1, G_2 y G_3. Ahora, calculamos el peso de cada uno de ellos, esto es, el número de coordenadas no nulas de cada elemento y obtenemos que todos

tienen peso 4, por lo que $\bar{d}_1 = 4$. El límite inferior del peso mínimo de los vectores que no pertenecen a C_1 es $\underline{d}_1 = 1 + 1 - (3 - 3) + 1 + 1 - (3 - 3) = 4$. Concluimos que $d = \bar{d}_1 = 4$ es la distancia mínima de C.

3

CÓDIGOS DE GOPPA

En [2], Goppa describió una nueva clase de códigos de corrección de errores lineales. Lo más importante es que algunos de estos códigos excedían el límite asintótico de Gilbert-Varshamov [21], una hazaña que muchos teóricos códigos pensaban que nunca podría lograrse. En este capítulo vamos a estudiar los códigos de Goppa [12, Sección 13.2.2] y sus propiedades más importantes, así como su codificación y decodificación [2].

3.1 CÓDIGOS CLÁSICOS DE GOPPA

Sea $q = p^r$ con p primo y $r > 0$ y $t > 0$. Consideramos la extensión de cuerpos $\mathbb{F}_{q^t}$ de $\mathbb{F}_q$, sea $L = \{\gamma_0, ..., \gamma_{n-1}\}$ una tupla de n elementos distintos de $\mathbb{F}_{q^t}$ y sea $g(x) \in \mathbb{F}_{q^t}[x]$ con $g(\gamma_i) \neq 0$ para $0 \leq i \leq n-1$. Entonces el *código de Goppa* $\Gamma(L, g)$ es el conjunto de vectores $c_0 \cdots c_{n-1} \in \mathbb{F}_q^n$ tal que

$$\sum_{i=0}^{n-1} \frac{c_i}{x - \gamma_i} \equiv 0 \pmod{g(x)}. \tag{4}$$

De esta forma, cuando la parte de la izquierda está escrita como una función racional, significa que el numerador es un múltiplo de $g(x)$. Además, trabajar módulo $g(x)$ es como trabajar en el anillo $\mathbb{F}_{q^t}[x]/(g(x))$, y la hipótesis $g(\gamma_i) \neq 0$ garantiza que $x - \gamma_i$ es invertible en este anillo. Se llama a $g(x)$ el *polinomio de Goppa* de $\Gamma(L, g)$.

A continuación, buscaremos una matriz de paridad para $\Gamma(L, g)$. Para ello, observamos que

$$\frac{1}{x - \gamma_i} \equiv -\frac{1}{g(\gamma_i)} \frac{g(x) - g(\gamma_i)}{x - \gamma_i} \pmod{g(x)}$$

ya que, comparando numeradores, $1 \equiv -g(\gamma_i)^{-1} (g(x) - g(\gamma_i)) \pmod{g(x)}$. Así que por (4) $\mathbf{c} = c_0 \cdots c_{n-1} \in \Gamma(L, g)$ si y solo si

$$\sum_{i=0}^{n-1} c_i \frac{g(x) - g(\gamma_i)}{x - \gamma_i} g(\gamma_i)^{-1} \equiv 0 \pmod{g(x)}. \tag{5}$$

Supongamos que $g(x) = \sum_{j=0}^{w} g_j x^j$ con $g_j \in \mathbb{F}_{q^t}$, donde $w = \mathrm{gr}(g(x))$. Entonces

$$\frac{g(x) - g(\gamma_i)}{x - \gamma_i} g(\gamma_i)^{-1} = g(\gamma_i)^{-1} \sum_{j=1}^{w} g_j \sum_{k=0}^{j-1} x^k \gamma_i^{j-1-k} = g(\gamma_i)^{-1} \sum_{k=0}^{w-1} x^k \left(\sum_{j=k+1}^{w} g_j \gamma_i^{j-1-k} \right).$$

Por lo tanto, por (5), estableciendo los coeficientes de x^k iguales a 0, en el orden $k = w-1, w-2, ..., 0$, tenemos que $\mathbf{c} \in \Gamma(L, g)$ si y solo si $Hc^T = 0$, donde

$$H = \begin{pmatrix} h_0 g_w & \cdots & h_{n-1} g_w \\ h_0 (g_{w-1} + g_w \gamma_0) & \cdots & h_{n-1} (g_{w-1} + g_w \gamma_{n-1}) \\ & \vdots & \\ h_0 \sum_{j=1}^{w} \left(g_j + \gamma_0^{j-1} \right) & \cdots & h_{n-1} \sum_{j=1}^{w} \left(g_j + \gamma_{n-1}^{j-1} \right) \end{pmatrix}, \tag{6}$$

con $h_i = g(\gamma_i)^{-1}$.

Proposición 10. *La matriz H se puede reducir a una matriz H' de dimensión $w \times n$, donde*

$$H' = \begin{pmatrix} g(\gamma_0)^{-1} & \cdots & g(\gamma_{n-1})^{-1} \\ g(\gamma_0)^{-1} \gamma_0 & \cdots & g(\gamma_{n-1})^{-1} \gamma_{n-1} \\ & \vdots & \\ g(\gamma_0)^{-1} \gamma_0^{w-1} & \cdots & g(\gamma_{n-1})^{-1} \gamma_{n-1}^{w-1} \end{pmatrix}. \tag{7}$$

Las entradas de H' están en $\mathbb{F}_{q^t}$. Eligiendo una base de $\mathbb{F}_{q^t}$ sobre $\mathbb{F}_q$, cada elemento de $\mathbb{F}_{q^t}$ se puede representar como un vector columna $t \times 1$ sobre $\mathbb{F}_q$. Reemplazando cada entrada de H' por su correspondiente vector columna, obtenemos una matriz H'' de dimensión $tw \times n$ sobre $\mathbb{F}_q$ que tiene la propiedad de que $\mathbf{c} \in \mathbb{F}_q^n$ está en $\Gamma(L, g)$ si y solo si $H''c^T = 0$.

No obstante, veamos que existe otra forma de construir la matriz de paridad de un código de Goppa $\Gamma(L, g)$, que es la que utilizaremos para la implementación. Sea $q = p^r$ con p primo y $r > 0$ y $t > 0$. Consideramos la extensión de cuerpos $\mathbb{F}_{q^t}$ de $\mathbb{F}_q$. Sean $\gamma_1, ..., \gamma_n \in \mathbb{F}_{q^t}$ y $g \in \mathbb{F}_{q^t}[x]$, tenemos que

$$g(\gamma_i) \neq 0 \quad \Leftrightarrow \quad x - \gamma_i \nmid g(x) \quad \Leftrightarrow \quad (x - \gamma_i, g(x)) = 1.$$

Por el Teorema de Bezout, existen dos polinomios $h_i, v_i \in \mathbb{F}_{q^t}[x]$ tales que

$$1 = h_i(x) \cdot (x - \gamma_i) + v_i(x) g(x).$$

Tomando módulo g, se cumple que

$$1 \equiv h_i(x) \cdot (x - \gamma_i) \pmod{g(x)}.$$

Luego el polinomio $h_i(x)$ es el inverso de $x - \gamma_i$ en $\mathbb{F}_{q^t}[x]$, para $i = 0, ..., n-1$. A los polinomios $h_i(x)$ con $i = 0, ..., n-1$ los llamaremos *polinomios de paridad*.

Como $\sum_{i=0}^{n-1} \frac{c_i}{x-\gamma_i} \equiv 0 \pmod{g(x)}$, entonces $\sum_{i=0}^{n-1} c_i h_i \equiv 0 \pmod{g(x)}$ y $\mathrm{gr}(h_i) \leq \mathrm{gr}(g) = s$.

Ahora, supongamos que $h_i(x) = \sum_{j=0}^{s} (h_i)_j x^j$ para $i = 0, ..., n-1$. Podemos definir la matriz H de dimensión $s \times n$ a partir de los coeficientes de los polinomios de paridad por columnas. Esto es,

$$H = \begin{pmatrix} (h_0)_1 & \cdots & (h_{n-1})_1 \\ \vdots & & \vdots \\ (h_0)_s & \cdots & (h_{n-1})_s \end{pmatrix}.$$

Sea $\mathcal{B}$ una base del cuerpo $\mathbb{F}_q$, las coordenadas del polinomio $(h_i)_j$, para $i = 0, ..., n-1$ y $j = 0, ..., s$ son

$$h_{ij} = \begin{pmatrix} h_{ij}^m \\ \vdots \\ h_{ij}^m \end{pmatrix},$$

donde m es el grado de la extensión $[\mathbb{F}_{q^t} : \mathbb{F}_q] = m$.

Ahora, colocando los polinomios h_{ij} por columnas, obtenemos la matriz H' de dimensión $sm \times n$, que es la matriz de paridad del código de Goppa $\Gamma(L, g)$.

El siguiente resultado nos muestra los límites en la dimensión y la distancia mínima de un código de Goppa.

Teorema 10. *Con la notación de esta sección, sea $\Gamma(L, g)$ un código de Goppa tal que $\mathrm{gr}(g(x)) = w$ entonces es un $[n, k, d]$ código con $k \geq n - wt$ y $d \geq w + 1$.*

Demostración. Las filas de H'' pueden ser dependientes, luego esta matriz tiene rango como máximo wt. Por lo que $\Gamma(L, g)$ tiene dimensión al menos $n - wt$. Si una palabra código $\mathbf{c} \in \Gamma(L, g)$ tiene peso w o menos, entonces el lado izquierdo de 4 es una función racional, donde el numerador tiene grado $w - 1$ o menos; pero este numerador tiene que ser múltiplo de $g(x)$, lo cual es una contradicción pues el grado de g es w. □

Corolario 1. *Si $\Gamma(L, g)$ es un código de Goppa tal que $\mathrm{gr}(g(x)) = w$, entonces puede corregir hasta*

$$\left\lfloor \frac{w}{2} \right\rfloor$$

errores.

Demostración. El Teorema 7 afirma que es posible corregir hasta

$$\left\lfloor \frac{d(\Gamma(L,g))-1}{2} \right\rfloor$$

errores. Además, por el Teorema 10 sabemos que la distancia mínima de un código de Goppa $\Gamma(L,g)$ tal que $\mathrm{gr}(g(x)) = w$ es $d(\Gamma(L,g)) = w+1$. Por lo que

$$\left\lfloor \frac{d(\Gamma(L,g))-1}{2} \right\rfloor = \left\lfloor \frac{(w+1)-1}{2} \right\rfloor = \left\lfloor \frac{w}{2} \right\rfloor.$$

□

Para ilustrar la teoría que acabamos de ver sobre los códigos de Goppa, usaremos el código descrito en el apéndice A en los siguientes ejemplos.

Ejemplo 10. Sea $\mathbb{F}_{2^2} \subset \mathbb{F}_{2^4}$ extensión de cuerpos finitos y sean a y b elementos de $\mathbb{F}_{2^4}$ y $\mathbb{F}_{2^2}$, respectivamente. Definimos el polinomio $g(x) = x^3 + ax^2 + 1 \in \mathbb{F}_{2^4}[x]$ y tomamos $n = 10$ como longitud del conjunto de definición. Podemos obtener el código de Goppa definido por dicho conjunto de definición y polinomio g escribiendo lo siguiente:

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: defining_set
> [z4^2 + 1, z4^3 + 1, z4^2, z4, z4^3 + z4^2, z4^2 + z4, z4^3 + z4 + 1,
    z4^3 + z4, z4^2 + z4 + 1, 1]
sage: C = Goppa(defining_set, g, F)
sage: C
> [10, 4] Goppa code
```

Observamos que el código es un $[10,4]$-código de Goppa, esto es, la longitud de este código es 10 y la dimensión es 4. Calculemos ahora la matriz de paridad y la matriz generadora asociada a este código.

```
sage: H = C.parity_check_matrix()
sage: show(H)
```

La matriz de paridad H que nos devuelve el código anterior es la siguiente:

$$H = \begin{pmatrix} 0 & b & 0 & b+1 & 1 & b+1 & 1 & 0 & 0 & 1 \\ 1 & 1 & b+1 & 1 & b+1 & b+1 & 0 & 0 & b+1 & 0 \\ 1 & 1 & 1 & b+1 & b & b & b+1 & 0 & 0 & b \\ 1 & b & b+1 & 0 & 1 & 0 & 1 & 0 & b & b \\ 0 & 1 & b & 1 & b & b & b & 1 & 1 & 1 \\ b & b & 0 & b & b & 1 & b & 0 & 1 & b \end{pmatrix}.$$

Análogamente, podemos calcular la matriz generadora G como se indica a continuación.

```
sage: G = C.get_generator_matrix()
sage: show(G)
```

Este código devuelve la siguiente matriz generadora:

$$G = \begin{pmatrix} 1 & 0 & 0 & b+1 & 0 & b+1 & b & b+1 & 0 & b \\ 0 & 1 & 0 & 0 & 0 & b+1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & b & 0 & b+1 & 0 & b+1 & 1 & b+1 \\ 0 & 0 & 0 & 0 & 1 & 1 & b+1 & 0 & 0 & 1 \end{pmatrix}.$$

Observamos que efectivamente el rango de la matriz generadora coincide con la dimensión del código, que era 4.

Además, podemos comprobar efectivamente que se da la igualdad $H \cdot G^T = 0$:

```
sage: H*G.T == 0
> True
```

Ejemplo 11. Sea $\mathbb{F}_{3^3} \subset \mathbb{F}_{3^6}$ extensión de cuerpos finitos y sean a y b elementos de $\mathbb{F}_{3^6}$ y $\mathbb{F}_{3^3}$, respectivamente. Definimos el polinomio $g(x) = x^2 + ax + 1 + a^2 \in \mathbb{F}_{3^6}[x]$ y tomamos $n = 8$ como longitud del conjunto de definición. Obtengamos el código de Goppa definido por dicho conjunto de definición y polinomio g.

```
sage: F = GF(3^3)
sage: L = GF(3^6)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^2 + a*x + 1 + a^2
```

```
sage: n = 8
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: C
> [8, 4] Goppa code
```

Observamos que el código es un $[8,4]$-código de Goppa, esto es, la longitud de este código es 8 y la dimensión es 4. Calculemos ahora la matriz de paridad y la matriz generadora asociada a este código.

```
sage: H = C.parity_check_matrix()
sage: show(H)
```

La matriz de paridad H que nos devuelve el código anterior es la siguiente:

$$H = \begin{pmatrix} 2b^2+2b+2 & b^2+2b & 2b & 2b^2+1 & 2b^2+b+2 & b^2 & 2b+2 & 2b+1 \\ 2b+1 & b^2+1 & b^2+b+1 & 2 & 2b^2+1 & 2b^2+b & b^2+2b & 2 \\ b^2+2b+2 & 2b^2+b+1 & b^2+2b+2 & b+2 & b^2+2b & b^2+b & 2 & b^2+2b+1 \\ 2b^2+2b+2 & 2b^2+b+2 & 2b^2+b+1 & 2b^2+2b & b & 2 & 2b^2+b+1 & b+1 \end{pmatrix}.$$

Análogamente, podemos calcular la matriz generadora G como se indica a continuación.

```
sage: G = C.get_generator_matrix()
sage: show(G)
```

Este código devuelve la siguiente matriz generadora:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 2b^2+b+1 & 2 & b^2+2 & 2b+1 \\ 0 & 1 & 0 & 0 & b^2+b & 2 & b^2+2b & b^2+1 \\ 0 & 0 & 1 & 0 & 2b^2+1 & b^2 & b^2+1 & b^2+2b+2 \\ 0 & 0 & 0 & 1 & 2b^2 & 1 & 1 & 2b+2 \end{pmatrix}.$$

Observamos que efectivamente el rango de la matriz generadora coincide con la dimensión del código, que era 4.

Además, podemos comprobar efectivamente que se da la igualdad $H \cdot G^T = 0$:

```
sage: H*G.T == 0
> True
```

3.1.1 *Códigos binarios de Goppa*

Los códigos binarios de Goppa son códigos de corrección de errores que pertenecen a la clase de los códigos de Goppa que acabamos de estudiar. La estructura binaria le da más ventajas matemáticas sobre variantes no binarias y, además, tienen propiedades interesantes para la construcción del criptosistema de McEliece.

Definición 21. Sea m un entero positivo. Consideramos la extensión de cuerpos $\mathbb{F}_{2^m}$ de $\mathbb{F}_2$, sea $L = \{\gamma_0, ..., \gamma_{n-1}\} \in \mathbb{F}_{2^m}^n$ una tupla de n elementos distintos de $\mathbb{F}_{2^m}$ y sea $g(x) \in \mathbb{F}_{2^m}[x]$ con $g(\gamma_i) \neq 0$ para $0 \leq i \leq n-1$. Entonces el *código binario de Goppa* $\Gamma(L, g)$ es el conjunto de vectores $c_0 \cdots c_{n-1} \in \{0,1\}^n$ tal que

$$\sum_{i=0}^{n-1} \frac{c_i}{x - \gamma_i} \equiv 0 \pmod{g(x)} \tag{8}$$

Observemos que si $g(x)$ es un polinomio irreducible, todos los elementos $\gamma \in \mathbb{F}_{2^m}$ satisfacen $g(\gamma) \neq 0$. A los códigos que cumplan esta propiedad los llamaremos *códigos binarios de Goppa irreducibles*.

La importancia de estos códigos se debe a que pueden doblar la capacidad correctora de los códigos generales de Goppa.

3.1.2 *Codificación de los códigos de Goppa*

La codificación de un mensaje consiste en escribirlo como una palabra código de un código. Fijado un código de Goppa $\Gamma(L, g)$ sobre $\mathbb{F}_q$ con matriz generadora G. Dado el mensaje $\mathbf{m} \in \mathbb{F}_q$, lo podemos codificar realizando la operación

$$\mathbf{c} = \mathbf{m}G.$$

De esta forma obtenemos la palabra código $\mathbf{c}$ que está en $\Gamma(L, g)$.

En los siguientes ejemplos vamos a usar el código descrito en el apéndice A para mostrar la codificación de algunos mensajes.

Ejemplo 12. Sea $\mathbb{F}_{3^3} \subset \mathbb{F}_{3^6}$ extensión de cuerpos finitos y sean a y b elementos de $\mathbb{F}_{3^6}$ y $\mathbb{F}_{3^3}$, respectivamente. Consideramos el polinomio $g(x) = x^2 + ax + 1 + a^2 \in \mathbb{F}_{3^6}[x]$ y tomamos $n = 10$ como longitud del conjunto de definición. Sabemos que, por el ejemplo 11, los mensajes deben tener longitud 6. Dada la palabra

$$word := (1, 0, b, b+1, b, 0, b+1) \in \mathbb{F}_{3^3},$$

la podemos transformar en una palabra código del código Goppa definido por el conjunto de definición y el polinomio g:

```
sage: F = GF(3^3)
sage: L = GF(3^6)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^2 + a*x + 1 + a^2
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: E = GoppaEncoder(C)
sage: word = vector(F, (1, 0, b + 1, b, 0, b + 1))
sage: x = E.encode(word)
sage: x
> (1, 0, z3 + 1, z3, 0, z3 + 1, 2*z3^2 + z3 + 1, z3^2 + z3, 0, 2*z3^2 +
    1)
```

Por lo que la palabra *word* codificada es la siguiente:

$$(1, 0, b+1, b, 0, b+1, 2b^2+b+1, b^2+b, 0, 2b^2+1).$$

3.1.3 *Decodificación de los códigos de Goppa*

Como hemos visto, al transmitir una palabra código a un receptor, éste podría recibir la palabra alterada. Para que el receptor pueda determinar el mensaje original, necesitaremos decodificar el mensaje recibido, [2]. Sea $\Gamma(L, g)$ un código de Goppa, donde $L = \{\gamma_0, ..., \gamma_{n-1}\}$ es una tupla de n elementos distintos de $\mathbb{F}_{q^t}$. Supongamos que e es el vector de errores que se añade a la palabra código c transmitida, entonces la palabra recibida y está dada por

$$y = c + e,$$

de donde

$$\sum_{\gamma \in L} \frac{y_\gamma}{x - \gamma} = \sum_{\gamma \in L} \frac{c_\gamma}{x - \gamma} + \sum_{\gamma \in L} \frac{e_\gamma}{x - \gamma}.$$

Como c es una palabra código, la primera sumatoria de la parte derecha desaparece al aplicar el módulo $g(x)$, y tenemos que

$$\sum_{\gamma \in L} \frac{y_\gamma}{x-\gamma} = \sum_{\gamma \in L} \frac{e_\gamma}{x-\gamma} \pmod{g(x)}.$$

Definiremos su síndrome como el polinomio $S(x)$ de grado menos que $\mathrm{gr}(g(x))$ tal que

$$S(x) = \sum_{\gamma \in L} \frac{y_\gamma}{x-\gamma} \pmod{g(x)}.$$

Acabamos de ver que

$$S(x) = \sum_{\gamma \in L} \frac{e_\gamma}{x-\gamma} \pmod{g(x)}.$$

Sea M un subconjunto de L tal que $e_\gamma \neq 0$ si y solo si $\gamma \in M$. Entonces

$$S(x) = \sum_{\gamma \in M} \frac{e_\gamma}{x-\gamma} \pmod{g(x)}. \tag{9}$$

De esta forma, ahora podemos introducir el polinomio cuyas raíces son las ubicaciones de los errores,

$$\sigma(x) = \prod_{\gamma \in M} (x-\gamma). \tag{10}$$

Sin embargo, para los códigos de Goppa es más conveniente definir una variante de este polinomio de la siguiente forma.

$$\eta(x) = \sum_{\gamma \in M} e_\gamma \prod_{\partial \in M \setminus \{\gamma\}} (x-\partial) \tag{11}$$

Observemos que de esta forma $\sigma(x)$ y $\eta(x)$ deben ser primos relativos.

Derivando la expresión de $\sigma(x)$, tenemos que

$$\sigma'(x) = \sum_{\gamma \in M} \prod_{\partial \in M \setminus \{\gamma\}} (x-\partial), \tag{12}$$

de donde, para cada $\gamma \in M$,

$$\eta(\gamma) = e_\gamma \prod_{\partial \in M \setminus \{\gamma\}} (\gamma-\partial) = e_\gamma \sigma'(\gamma),$$

por lo que $e_\gamma = \frac{\eta(\gamma)}{\sigma'(\gamma)}$. De esta forma, una vez que hemos calculado los polinomios σ y η, las coordenadas del vector error vienen dadas por

$$e_\gamma = \begin{cases} 0 & \text{si} \quad \sigma(\gamma) \neq 0 \\ \\ \frac{\eta(\gamma)}{\sigma'(\gamma)} & \text{si} \quad \sigma(\gamma) = 0 \end{cases},$$

donde $\sigma'(x)$ es la derivada de $\sigma(x)$.

Lo esencial para decodificar los códigos de Goppa es determinar los coeficientes de los polinomios σ y η. Para ello, tenemos que relacionar σ y η al síndrome de la ecuación (9). Esto se consigue multiplicando las ecuaciones (9) y (10), obteniendo

$$S(x) \cdot \sigma(x) \equiv \eta(x) \pmod{g(x)}. \tag{13}$$

La ecuación (13) es la *ecuación clave* para decodificar los códigos de Goppa. Dado $g(x)$ y $S(x)$, el problema de decodificar consiste en encontrar polinomios de grado bajo $\sigma(x)$ y $\eta(x)$ que satisfacen (13).

Reduciendo cada potencia de $x \pmod{g(x)}$ e igualando coeficientes de $1, x, ..., x^{\mathrm{gr}(g)-1}$, tenemos que (13) es un sistema de $\mathrm{gr}(G)$ ecuaciones lineales donde las incógnitas son los coeficientes de σ y η. Por lo tanto, para probar que el decodificador es capaz de corregir todos los patrones hasta t errores, basta con probar que (13) tiene una única solución con grados de σ y de η suficientemente pequeños. Esto equivale a que el conjunto de ecuaciones lineales correspondientes sean linealmente independientes.

Supongamos que existen dos pares diferentes de soluciones a (13):

$$S(x)\sigma^{(1)}(x) \equiv \eta^{(1)}(x) \pmod{g(x)}, \tag{14}$$

$$S(x)\sigma^{(2)}(x) \equiv \eta^{(2)}(x) \pmod{g(x)}, \tag{15}$$

donde $\sigma^{(1)}(x)$ y $\eta^{(1)}(x)$ son primos relativos, al igual que $\sigma^{(2)}(x)$ y $\eta^{(2)}(x)$. Además, $\sigma^{(1)}(x)$ y $g(x)$ no pueden tener ningún factor en común, pues en ese caso ese factor podría dividir a $\eta^{(1)}(x)$, contradiciendo que $\sigma^{(1)}(x)$ y $\eta^{(1)}(x)$ son primos relativos. Así, podemos dividir (14) por $\sigma^{(1)}(x)$ y obtenemos

$$S(x) \equiv \frac{\eta^{(1)}(x)}{\sigma^{(1)}(x)} \pmod{g(x)}.$$

De la misma forma para (15),

$$S(x) \equiv \frac{\eta^{(2)}(x)}{\sigma^{(2)}(x)} \pmod{g(x)}$$

de donde,

$$\sigma^{(1)}(x)\eta^{(2)}(x) \equiv \sigma^{(2)}(x)\eta^{(1)}(x) \pmod{g(x)}. \tag{16}$$

Si $\mathrm{gr}(G) = 2t$ y $\mathrm{gr}(\sigma^{(1)}) \leq t$, $\mathrm{gr}(\sigma^{(2)}) \leq t$, $\mathrm{gr}(\eta^{(2)}) < t$ y $\mathrm{gr}(\eta^{(1)}) < t$, entonces se da la siguiente igualdad

$$\sigma^{(1)}(x)\eta^{(2)}(x) = \sigma^{(2)}(x)\eta^{(1)}(x). \tag{17}$$

Así, $\sigma^{(1)}$ divide a $\sigma^{(2)}\eta^{(1)}$, y como $\sigma^{(1)}$ y $\eta^{(1)}$ son primos relativos, $\sigma^{(1)}$ tiene que dividir a $\sigma^{(2)}$. Análogamente, $\sigma^{(2)}$ tiene que dividir a $\sigma^{(1)}$. Como ambos son mónicos, se tiene que $\sigma^{(1)} = \sigma^{(2)}$ y así, $\eta^{(1)} = \eta^{(2)}$. Con esto hemos probado que si el grado de g es $2t$, entonces (13) tiene una única solución cuando $\mathrm{gr}(\eta) < \mathrm{gr}(\sigma) \leq t$, luego el correspondiente sistema de ecuaciones lineales donde las incógnitas son los coeficientes de σ y η tiene que ser no singular. En el siguiente teorema se concluye este resultado.

Teorema 11. *Si* $\mathrm{gr}(g(x)) = 2t$*, entonces hay un algoritmo de decodificación algebraica de corrección de t errores para el código q-ario de Goppa con el polinomio de Goppa* $g(x)$*.*

Estudiemos ahora este resultado en el caso binario, primero observamos que ya que todos los e_γ distintos de cero son iguales a 1, entonces (11) y (12) coinciden. De esta forma, la ecuación (16) ahora pasa a ser

$$\sigma^{(1)}\left(\sigma^{(2)}\right)' \equiv \sigma^{(2)}\left(\sigma^{(1)}\right)' \pmod{g(x)}$$

Ahora, cuando σ sea par escribiremos en su lugar $\hat{\sigma}$, mientras que cuando σ sea impar escribiremos en su lugar $x\sigma'$. Así, tenemos que

$$\begin{aligned}\left(\hat{\sigma}^{(1)} + x\sigma^{(1)'}\right)\sigma^{(2)'} &\equiv \left(\hat{\sigma}^{(2)} + x\sigma^{(2)'}\right)\sigma^{(1)'},\\ \hat{\sigma}^{(1)}\sigma^{(2)'} + \hat{\sigma}^{(2)}\sigma^{(1)'} &\equiv 0 \pmod{g(x)}.\end{aligned}$$

El lado izquierdo es un cuadrado perfecto, pues todos los polinomios de ese lado son pares. Esto implica que

$$\hat{\sigma}^{(1)}\sigma^{(2)'} + \hat{\sigma}^{(2)}\sigma^{(1)'} \equiv 0 \pmod{\bar{g}(x)}$$

donde $\bar{g}(x)$ es múltiplo de $g(x)$ de menor grado ya que $\bar{g}$ es un cuadrado perfecto. Por lo que, si $\mathrm{gr}(\bar{g}) = 2t$, $\mathrm{gr}(\sigma^{(1)}) \leq t$ y $\mathrm{gr}(\sigma^{(2)}) \leq t$, entonces

$$\hat{\sigma}^{(1)}\left(\sigma^{(2)}\right)' = \hat{\sigma}^{(2)}\sigma^{(1)'}.$$

Por la primalidad relativa, $\sigma^{(1)} = \sigma^{(2)}$. En el siguiente teorema se concluye este resultado.

Teorema 12. *Si* $\mathrm{gr}(g(x)) = t$ *y si* $g(x)$ *no tiene factores irreducibles repetidos, entonces hay un algoritmo de decodificación algebraica de corrección de t errores para el código binario de Goppa con el polinomio de Goppa* $g(x)$*.*

3.1.3.1 *Algoritmo de decodificación de Sugiyama*

El algoritmo de Sugiyama, descrito en [20], es una aplicación simple del algoritmo de Euclides para determinar el polinomio localizador de errores de una manera más eficiente.

A continuación vamos a describir alguna de las propiedades el algoritmo de Euclides que están relacionadas con el método para resolver la ecuación clave para decodificar los códigos de Goppa. El algoritmo de Euclides que presentamos en 2, en este caso aplicado a los polinomios $r_{-1}(x)$ y $r_0(x)$ donde $\mathrm{gr}(r_{-1}) > \mathrm{gr}(r_0)$, se puede reescribir en forma de matriz como sigue:

$$\begin{pmatrix} r_{i-2}(x) \\ r_{i-1}(x) \end{pmatrix} = \begin{pmatrix} q_i(x) & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} r_{i-1}(x) \\ r_i(x) \end{pmatrix}. \tag{18}$$

Definimos los polinomios $U_i(x)$ y $V_i(x)$ como

$$U_i(x) = q_i(x)U_{i-1}(x) + U_{i-2}(x) \tag{19}$$

y

$$V_i(x) = q_i(x)V_{i-1}(x) + V_{i-2}(x),$$

donde $U_0(x) = 1$, $U_{-1}(x) = 0$, $V_0(x) = 0$ y $V_{-1}(x) = 1$, tenemos que

$$\begin{pmatrix} U_i(x) & U_{i-1} \\ V_i(x) & V_{i-1} \end{pmatrix} = \begin{pmatrix} q_1(x) & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} q_i(x) & 1 \\ 1 & 0 \end{pmatrix}. \tag{20}$$

De (18) y (20), obtenemos

$$\begin{pmatrix} r_{-1}(x) \\ r_0(x) \end{pmatrix} = \begin{pmatrix} U_i(x) & U_{i-1}(x) \\ V_i(x) & V_{i-1}(x) \end{pmatrix} \begin{pmatrix} r_{i-1}(x) \\ r_i(x) \end{pmatrix}.$$

Como el determinante de la matriz de la izquierda de 20 es $(-1)^i$, tenemos que

$$\begin{pmatrix} r_{-1}(x) \\ r_0(x) \end{pmatrix} = (-1)^i \begin{pmatrix} V_{i-1}(x) & -U_{i-1}(x) \\ -V_i(x) & U_i(x) \end{pmatrix} \begin{pmatrix} r_{-1}(x) \\ r_0(x) \end{pmatrix}.$$

Esta ecuación nos permite relacionar el resto $r_i(x)$ de la iteración i-ésima con los polinomios $r_{-1}(x)$ y $r_0(x)$ de la siguiente manera

$$r_i(x) = (-1)^i \left(-V_i(x)r_{-1}(x) + U_i(x)r_0(x)\right). \tag{21}$$

Como consecuencia, observamos que $r_{-1}(x)$ divide al polinomio $\big(-r_{-1}(x) + (-1)^i U_i(x) r_0(x)\big)$. Esto conlleva a la siguiente relación:

$$r_i(x) \equiv (-1)^i U_i(x) r_0(x) \pmod{r_{-1}(x)}. \tag{22}$$

Observamos que esta relación es similar a la ecuación clave para decodificar los códigos de Goppa. Veamos otras propiedades del algoritmo de Euclides que también están relacionadas con dicha ecuación clave.

Aplicando 2 a los polinomios $r_{-1}(x)$ y $r_0(x)$, sabemos que en cada iteración i-ésima se cumple la siguiente relación:

$$\operatorname{gr}(r_i) < \operatorname{gr}(r_{i-1}). \tag{23}$$

Por (19), se tiene que $\operatorname{gr}(U_i) = \sum_{j=1}^{i} \operatorname{gr}(q_j)$. Como $\operatorname{gr}(r_{i-1}) = \operatorname{gr}(r_{-1}) - \sum_{j=1}^{i} \operatorname{gr}(q_j)$, aplicando 2, es claro que

$$\operatorname{gr}(U_i) = \operatorname{gr}(r_{-1}) - \operatorname{gr}(r_{i-1}). \tag{24}$$

A partir de (20), obtenemos

$$U_i(x) V_{i-1}(x) = U_{i-1}(x) V_i(x) = (-1)^i. \tag{25}$$

De esta forma, hemos probado que los polinomios $U_i(x)$ y $V_i(x)$ son primos relativos.

A continuación se describe el teorema fundamental para resolver la ecuación clave para decodificar los códigos de Goppa, usando el algoritmo de Euclides.

Teorema 13 (Algoritmo de Sugiyama). *Fijado un código de Goppa $\Gamma(L, g)$ sobre el cuerpo $\mathbb{F}_q$ y sean $S(x)$ su síndrome, $\sigma(x)$ el polinomio localizador de errores y $\eta(x)$ el polinomio evaluador de errores. Definimos $r_{-1}(x) = g(x)$ y $r_0(x) = S(x)$. Empezamos las divisiones 2 del algoritmo de Euclides para calcular el máximo común divisor de $r_{-1}(x)$ y $r_0(x)$ desde $i = 0$ hasta $i = k$, de tal forma que se cumpla que $\operatorname{gr}(r_{k-1}) \geq t$ y $\operatorname{gr}(r_k) \leq t - 1$. Entonces la solución viene dada por*

$$\begin{aligned} \sigma_s(x) &= \delta U_k(x), \\ \eta_s(x) &= (-1)^k \delta r_k(x), \end{aligned} \tag{26}$$

donde δ es una constante no nula que hace que $\sigma_s(x)$ sea mónico.

Demostración. Comenzamos viendo que el número de iteraciones k cuando se satisfacen $\operatorname{gr}(r_{k-1}) \geq t$ y $\operatorname{gr}(r_k) \leq t - 1$ es único. Tenemos que $\operatorname{gr}(r_0) = \operatorname{gr}(S) \leq t$ [20, Lema 1]. Sea $p(x)$ el máximo común divisor de $g(x)$ y $S(x)$, existe la iteración i en la que se cumple que $\operatorname{gr}(p) \leq \operatorname{gr}(r_i) \leq t - 1$. Por (23), es claro que la secuencia de $\operatorname{gr}(r_i)$ para $i = 0, 1, 2$ decrece

monótonamente conforme i se incrementa. Concluimos que el número de iteraciones k es único.

Por otra parte, como $U_k(x)S(x) \equiv (-1)^k r_k(x) \pmod{g(x)}$ se satisface por (22), tenemos que $\sigma_s(x)$ y $\eta_s(x)$ dados por (26) satisfacen la relación

$$\sigma_s(x)S(x) \equiv \eta_s(x) \pmod{g(x)}.$$

Deduzcamos ahora el grado de los polinomios σ_s y η_s. Sabemos que $\operatorname{gr}(r_k) \leq t-1$, luego η cumple que $\operatorname{gr}(\eta_s) \leq t-1$. En cuanto a σ, por (24) tenemos que $\operatorname{gr}(U_k) = \operatorname{gr}(g) - \operatorname{gr}(r_{k-1})$ y como $\operatorname{gr}(r_{k-1}) \geq t$, entonces $\operatorname{gr}(\sigma_s) \leq t$.

Veamos ahora que $\sigma_s(x)$ y $\eta_s(x)$ son primos relativos. Para ello, supongamos que no lo son. Entonces estos polinomios satisfacen

$$\begin{aligned} \sigma_s(x) &= \mu(x)\sigma(x) \\ \eta_s(x) &= \mu(x)\eta(x), \end{aligned} \tag{27}$$

donde $\mu(x)$ es un polinomio conveniente [20, Página 92].

Por (21) y (27), tenemos que

$$\begin{aligned} \mu(x)\eta(x) &= \mu(x)\sigma(x)S(x) - \delta V_k(x)g(x) \\ \eta(x) &= \sigma(x)S(x) + \phi(x)g(x), \end{aligned}$$

donde $\phi(x)$ es un polinomio conveniente. A partir de estas relaciones, es evidente que $\mu(x)$ divide a $V_k(x)$. El polinomio $\mu(x)$ también divide a $U_k(x)$. Pero esto contradice a (25) que implica que $U_k(x)$ y $V_k(x)$ son primos relativos. Por lo que $\sigma_s(x)$ y $\eta_s(x)$ son primos relativos.

Finalmente, el factor δ hace que $\sigma_s(x)$ sea mónico. Por lo tanto, $\sigma_s(x)$ y $\eta_s(x)$ dados por (26) son la solución del problema. □

Corolario 2. *Sea e el número de errores que realmente se produjeron y sea k el número de iteraciones del algoritmo descrito en el teorema, entonces $k \leq e$.*

Demostración. Como el número de errores que se produjeron es e, entonces $\operatorname{gr}(\sigma) = e$. Por (19) y (26),

$$\operatorname{gr}(\sigma_s) = \sum_{i=1}^{k} \operatorname{gr}(q_i).$$

Como $\operatorname{gr}(q_i) \geq 1$ para todo i, concluimos que $k \leq e$. □

En resumen, el algoritmo de Sugiyama consiste en:

I. Calcular el síndrome $S(x)$.

II. Sean $r_{-1}(x) = g(x)$, $r_0(x) = S(x)$, $U_{-1}(x) = 0$, $U_0(x) = 1$, $V_0(x)$ y $V_{-1}(x) = 1$.
III. Buscar $U_i(x)$, $V_i(x)$, $q_i(x)$ y $r_i(x)$ aplicando el algoritmo de Euclides para encontrar el máximo común divisor de $r_{-1}(x)$ y $r_0(x)$ para $i = 1, ..., k$, hasta que k cumpla que $\text{gr}(r_{k-1}(x)) \geq t$ y $\text{gr}(r_k(x)) < t$:

$$\begin{aligned} r_{i-2}(x) &= r_{i-1}(x)q_i(x) + r_i(x), \qquad \text{gr}(r_i(x)) < \text{gr}(r_{i-1})(x) \\ U_i(x) &= q_i(x)U_{i-1}(x) + U_{i-2}(x) \\ V_i(x) &= q_i(x)V_{i-1}(x) + V_{i-2}(x) \end{aligned}$$

IV. La solución viene dada por:

$$\begin{aligned} \sigma_s(x) &= \delta U_k(x), \\ \eta_s(x) &= (-1)^k \delta r_k(x). \end{aligned}$$

V. Calcular las raíces de $\sigma_s(x)$.
VI. Para cada raíz P, buscar la posición en la que se encuentra en el conjunto de definición y el error en esa posición tendrá el valor:

$$\frac{\eta(P)}{\sigma'(P)}.$$

VII. Restar a la palabra codificada el error para obtener el mensaje original codificado.

En el siguiente ejemplo vamos a usar el código descrito en el apéndice A para mostrar la decodificación de una palabra codificada. Esta decodificación se basa en el algoritmo de Sugiyama explicado en esta sección.

Ejemplo 13. Sea $\mathbb{F}_{3^3} \subset \mathbb{F}_{3^6}$ extensión de cuerpos finitos y sean a y b elementos de $\mathbb{F}_{3^6}$ y $\mathbb{F}_{3^3}$, respectivamente. Consideramos el polinomio $g(x) = x^2 + ax + 1 + a^2 \in \mathbb{F}_{3^6}[x]$ y tomamos $n = 10$ como longitud del conjunto de definición. En el ejemplo 12 obtuvimos la siguiente palabra codificada:

$$x := (1, 0, b+1, b, 0, b+1, 2b^2+b+1, b^2+b, 0, 2b^2+1) \in \mathbb{F}_{3^3}.$$

Supongamos que el error que se le añade a la palabra anterior es:

$$e := (0, 0, 2b^2+2b+1, 0, 0, 0, 0, 0, 0, 0).$$

Por lo que la palabra que recibiría el destinatario sería:

$$y = x + e = (1, 0, 2b^2+2, b, 0, b+1, 2b^2+b+1, b^2+b, 0, 2b^2+1).$$

Para recuperar el mensaje original, vamos a aplicar el algoritmo de decodificación de Sugiyama.

```
sage: F = GF(3^3)
sage: L = GF(3^6)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^2 + a*x + 1 + a^2
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: D = GoppaDecoder(C)
sage: y = (1, 0, 2*b^2 + 2, b, 0, b + 1, 2*b^2 + b + 1, b^2 + b, 0, 2*b^2
    + 1)
sage: D.decode_to_code(y)
> (1, 0, z3 + 1, z3, 0, z3 + 1, 2*z3^2 + z3 + 1, z3^2 + z3, 0, 2*z3^2 +
   1)
```

Por lo que el mensaje original codificado es el siguiente:

$$(1, 0, b+1, b, 0, b+1, 2b^2+b+1, b^2+b, 0, 2b^2+1),$$

que efectivamente coincide con la palabra codificada x. Además, si queremos recuperar el mensaje original, debemos llamar al método `decode_to_message`:

```
sage: D.decode_to_message(y)
> (1, 0, z3 + 1, z3, 0, z3 + 1)
```

Observamos que obtenemos el mensaje original que fue transmitido, $word = (1, 0, b+1, b, 0, b+1)$.

4

CRIPTOGRAFÍA POST-CUÁNTICA BASADA EN CÓDIGOS

El principal objetivo de la criptografía es proporcionar confidencialidad mediante métodos de cifrado. Cuando queremos enviar un mensaje a un destinatario, el canal de comunicación puede ser inseguro y otras personas podrían leerlo o incluso modificarlo de tal forma que el destinatario no se diera cuenta. Para prevenir estos ataques nos será de utilidad la criptografía.

En este capítulo introduciremos las nociones básicas de la criptografía, junto a sus objetivos, y estudiaremos los dos principales tipos de criptosistemas (simétricos y asimétricos) [4]. Definiremos el primer y más utilizado algoritmo de los sistemas criptográficos asimétricos, el RSA. Finalmente, discutiremos la seguridad de los criptosistemas ante la existencia de ordenadores cuánticos y mostraremos algunas alternativas que sean capaces de resistir sus ataques [3]. En concreto, estudiaremos el criptosistema de McEliece y el de Niederreiter [19].

4.1 INTRODUCCIÓN

En general, los métodos de cifrado consisten en encriptar un mensaje, llamado *texto plano*, antes de ser transmitido, de esta forma obtenemos un *texto cifrado* o *criptograma*. Este texto cifrado se transmite al destinatario, quien lo *descifra* mediante una *clave de descifrado*, la cual solo conocen el receptor y el emisor, que previamente ha sido intercambiada.

Formalmente, dado un conjunto de los mensajes $\mathcal{M}$, un conjunto de los criptogramas $\mathcal{C}$ y el espacio de claves $\mathcal{K}_p \times \mathcal{K}_s$, un *criptosistema* viene definido por dos aplicaciones

$$E : \mathcal{K}_p \times \mathcal{M} \to \mathcal{C},$$

$$D : \mathcal{K}_s \times \mathcal{C} \to \mathcal{M},$$

tales que para cualquier clave $k_p \in \mathcal{K}_p$, existe una clave $k_s \in \mathcal{K}_s$ de manera que dado cualquier mensaje $m \in \mathcal{M}$,

$$D(k_s, E(k_p, m)) = m. \tag{28}$$

Para simplificar la notación de las funciones de cifrado y descifrado usaremos, fijadas las claves $k_p \in \mathcal{K}_p$ y su correspondiente $k_s \in \mathcal{K}_s$:

$$E_{k_p} : \mathcal{M} \to \mathcal{C}, \quad [E_{k_p}(m) = E(k_p, m)]$$
$$D_{k_s} : \mathcal{C} \to \mathcal{M}, \quad [D_{k_s}(c) = D(k_s, c)]$$

La propiedad 28 se transforma en

$$D_{k_s}\left(E_{k_p}(m)\right) = m.$$

Para encriptar y descifrar existen *algoritmos de cifrado* y *de descifrado*, respectivamente, y cada uno usará una clave secreta. Si esta clave es la misma en ambos algoritmos, diremos que los métodos de encriptación son *simétricos*. Algunos ejemplos importantes de estos métodos son DES (*Data Encryption Standard*) y AES (*Advanced Encryption Standard*) [4, Sección 2.1].

En 1976, Diffie y Hellman [9] introdujeron un concepto revolucionario, la *criptografía de Clave Pública*, también llamada *criptografía asimétrica*, que permitió dar una solución al antiguo problema del intercambio de claves e indicar el camino a la firma digital. Los métodos de cifrado de *clave pública* son *asimétricos*. Cada receptor tiene una clave personal $k = (k_p, k_s)$, que consiste en dos partes: k_p es la clave de cifrado y es pública, y k_s es la clave de descifrado, que es privada. De esta forma, si queremos enviar un mensaje, lo encriptaremos mediante la clave pública k_p del receptor. Así, el receptor podrá descifrar el texto cifrado usando su clave privada k_s, que solo conoce él. Al ser la clave pública, cualquiera puede encriptar un mensaje usándola, sin embargo descifrarlo sin saber la clave privada será casi imposible.

4.2 OBJETIVOS DE LA CRIPTOGRAFÍA

Además de proporcionar confidencialidad, la criptografía proporciona soluciones para otros problemas:

1. *Confidencialidad*. La información solo puede ser accesible por las entidades autorizadas.
2. *Integridad de datos*. El receptor de un mensaje debería ser capaz de determinar que el mensaje no ha sido modificado durante la transmisión.
3. *Autenticidad*. El receptor de un mensaje debería ser capaz de verificar su origen.
4. *No repudio*. El emisor de un mensaje debería ser incapaz de negar posteriormente que envió el mensaje.

Para garantizar la integridad de los datos, hay métodos simétricos y de clave pública. El mensaje m es aumentado por un *código de autenticación de mensaje* (MAC). Este código es generado por un algoritmo que depende de la clave secreta. Así, el mensaje aumentado $(m, MAC(k, m))$

está protegido contra modificaciones. El receptor ahora puede comprobar la integridad del mensaje $(m, \bar{m})$ verificando que $MAC(k, m) = \bar{m}$.

Las firmas digitales requieren métodos de clave pública y proporcionan autenticación y no repudiabilidad. Cualquier persona puede verificar si una firma digital es válida con la clave pública del firmante. Esto es, si firmamos con nuestra clave privada k, obtenemos la firma $s = Sign(k_s, m)$. El receptor recibe la firma s del mensaje m y comprueba con el algoritmo de verificación *Verify* que se cumple que $Verify(k_p, s, m) = ok$, siendo k_p la clave pública del emisor.

4.3 CRIPTOGRAFÍA ASIMÉTRICA

A diferencia de la criptografía simétrica, en la criptografía asimétrica los participantes en la comunicación no comparten una clave secreta. Cada uno tiene un par de claves: la *clave secreta* k_s conocida solo por él y una *clave pública* conocida por todos.

Supongamos que Bob tiene un par de claves (k_p, k_s) y Alice quiere encriptar un mensaje m para Bob. Alice, como cualquier otra persona, conoce la clave pública k_p de Bob. Alice usa una función de encriptación E con la clave pública k_p de Bob para obtener el texto cifrado $c = E_{k_p}(m)$. Esto solo puede ser seguro si es prácticamente inviable calcular m de $c = E_{k_p}(m)$. Sin embargo, Bob sí es capaz de calcular el mensaje m, ya que puede usar su clave secreta. La función de encriptación E_{k_p} debe tener la propiedad de que su pre-imagen m del texto cifrado $c = E_{k_p}(m)$ sea fácil de calcular usando la clave secreta k_s de Bob, quien es el único que puede descifrar el mensaje encriptado.

En la criptografía de clave pública, necesitamos unas funciones $\left(E_{k_p}\right)_{k_p \in K_P}$ tales que cada función E_{k_p} se pueda cacular con un algoritmo eficiente. Sin embargo, su pre-imagen debería ser prácticamente inviable de calcular. Estas familias $\left(E_{k_p}\right)_{k_p \in K_P}$ se denominan *funciones de una sola dirección*. En cada función E_{k_p} de la familia, tiene que haber una información secreta k_s para que el algoritmo sea eficiente y calcule la inversa de E_{k_p}. Las funciones con esa propiedad se denominan *funciones con trampilla*.

En 1976, Diffie y Hellman [9] presentaron la idea de la criptografía de clave pública, es decir, introdujeron métodos de clave pública para el acuerdo de clave y, además, describieron cómo las firmas digitales funcionarían. El primer criptosistema de clave pública que podía servir como un mecanismo de acuerdo de clave y como una firma digital fue el criptosistema RSA, que actualmente es el más conocido y usado. Este criptosistema lleva el nombre de sus inventores: Rivest, Shamir y Adleman. El criptosistema RSA se basa en la dificultad de factorizar grandes números, lo que le permite construir funciones de una sola dirección con una trampilla. Otras funciones de una sola dirección se basan en la dificultad de extraer

logaritmos discretos. Estos dos problemas de la teoría de números son los cimientos de los criptosistemas de clave pública más usados actualmente.

Cada participante en un criptosistema de clave pública tiene una clave $k = (k_p, k_s)$, que consiste en una clave pública (k_p) y una clave privada o secreta (k_s). Para garantizar la seguridad del criptosistema, debería ser inviable obtener la clave privada k_s a partir de la clave pública k_p. Un algoritmo eficiente debería ser el encargado de elegir aleatoriamente ambas claves en un gran espacio de parámetros. Así, cualquiera puede usar k_p para encriptar mensajes, pero solo quien posea k_s podrá descifrar.

En cuanto a las firmas digitales, supongamos que tenemos una familia de funciones con trampilla $\left(E_{k_p}\right)_{k_p \in K_P}$ donde cada función E_{k_p} es biyectiva. Sea k_p la clave pública de Alice, quien es la única capaz de calcular la inversa $E_{k_p}^{-1}$ de E_{k_p} pues para ello se necesita la clave privada k_s de Alice. De esta forma, si Alice quiere firmar un mensaje m, tiene que calcular $E_{k_p}^{-1}(m)$, que será el valor de la firma s de m. Todo el mundo puede verificar la firma de Alice s pues cualquiera puede usar su clave pública k_p y calcular $E_{k_p}(s)$. Si $E_{k_p}(s) = m$, entonces podemos asegurarnos de que Alice realmente firmó m porque es la única que es capaz de calcular $E_{k_p}^{-1}(m)$.

Una importante aplicación de los criptosistemas de clave pública es que permiten intercambiar claves en sistemas de clave secreta. Si Alice conoce la clave pública de Bob, ella puede generar una clave de sesión, cifrarla con la clave pública de Bob y enviársela.

Algunos sistemas conocidos de clave pública son:

- *RSA*: está basado en el problema de factorización de enteros.
- *ElGamal*: está basado en el problema del logaritmo discreto.
- *McEliece*: está basado en la teoría de los códigos Goppa.
- *Curvas Elípticas*: son una generalización del sistema ElGamal y se basan en el problema de calcular logaritmos discretos en curvas elípticas.

4.3.1 *RSA*

El criptosistema RSA consiste en multiplicar dos números primos muy grandes y publicar su producto n. Una parte de la clave pública la conformará n, mientras que los factores de n se mantienen en secreto y se usarán como clave privada. La idea básica es que los factores de n no puedan recuperarse de n en un tiempo razonable. Por lo que la seguridad de RSA radica en la dificultad del problema de factorización de enteros.

4.3.1.1 *Generación de claves*

Cada usuario del criptosistema RSA posee una clave pública y otra privada. Para generar este par de claves, se siguen los siguientes tres pasos:

1. Se eligen aleatoriamente dos grandes números primos distintos p y q y se calcula su producto $n = p \cdot q$. También calculamos $\phi(n) = n + 1 - (p + q)$.
2. Se elige un entero e tal que $1 < e < \phi(n)$ y sea primo con $\phi(n)$.
3. Se calcula d que verifique $ed \equiv 1 \pmod{\phi(n)}$, es decir, $d \equiv e^{-1} \pmod{\phi(n)}$.
4. La clave privada es (p, q, d).
5. La clave pública es (n, e).

Los números n, e y d se denominan *módulo*, *exponente de cifrado* y *exponente de descifrado*, respectivamente. El exponente de descifrado d se puede obtener con el algoritmo extendido de Euclides. Con este exponente es posible descifrar un texto cifrado y generar una firma digital.

4.3.1.2 *Cifrado y descifrado*

Supongamos que queremos cifrar un mensaje. Para ello, usaremos la clave pública (n, e). Para cifrar un texto plano m podemos usar la función de cifrado:

$$RSA_{n,e}(m) = m^e \pmod{n},$$

es decir, el texto cifrado c es m^e módulo n.

Para descifrar el criptograma c, se usa la función de descifrado:

$$RSA_{n,d}(c) = c^d \pmod{n}.$$

De esta forma, se puede recuperar el texto plano, es decir, $m = c^d$ módulo n, pues las funciones de cifrado y descifrado $RSA_{n,e}$ y $RSA_{n,d}$ son inversas entre sí.

Con este procedimiento de cifrado, podemos cifrar secuencias de bit hasta $k := \lfloor \log_2 n \rfloor$ bits. Si los mensajes son más largos, podemos dividirlos en bloques de longitud k y cifrar cada uno por separado.

Ejemplo 14. Consideremos dos números primos $p = 7$ y $q = 11$, tenemos que $n = p \cdot q = 77$ y $\phi(n) = 60$. Busquemos ahora un número e mayor que 1 y menor que 60 y que sea primo con $\phi(n) = 60$, elegimos $e = 13$. Por lo que el exponente de descifrado es

$$d \equiv e^{-1} \equiv 37 \pmod{\phi(n)}.$$

La clave pública es $(e, n) = (13, 77)$ y la clave privada es $(d, n) = (37, 77)$. Vamos a cifrar el mensaje $m = 42$ con la función de cifrado:

$$RSA_{n,e}(m) = m^e \equiv 42^{13} \equiv 14 \pmod{n}.$$

Luego el mensaje cifrado m es 14. Ahora, si queremos descifrarlo, calcularemos lo siguiente:

$$RSA_{n,d}(c) = c^d \equiv 14^{37} \equiv 42 \pmod{n}.$$

Observamos que efectivamente coincide con el mensaje original m.

4.3.1.3 *Firma digital*

El criptosistema RSA también se puede usar para realizar firmas digitales. Sea (n, e) la clave pública y d el exponente de descifrado, si queremos firmar un mensaje m, le aplicamos el algoritmo de descifrado y obtenemos la *firma* de m, $\sigma = m^d$. Decimos que (m, σ) es un *mensaje firmado*. Para verificar ese mensaje, basta con calcular σ^e, donde e es la clave pública del firmante, y comprobar que coincide con m.

4.4 CRIPTOGRAFÍA POST-CUÁNTICA

En [3] se comenta la creencia de que algunos sistemas criptográficos, tales como RSA, resisten los ataques de ordenadores clásicos, pero no de los ordenadores cuánticos. Es por esto que surgen diversas preocupaciones ante la amenaza de los ordenadores cuánticos: se duda sobre si seguir usando RSA o simplemente cambiar a otros sistemas criptográficos que sean resistentes a dichos ordenadores. Sin embargo, esto último no es tan sencillo pues necesitamos tiempo para mejorar la eficiencia, fomentar la confanza y mejorar de la usabilidad de los criptosistemas post-cuánticos. En resumen, todavía no estamos preparados para que el mundo cambie a la criptografía post-cuántica.

En esta sección estudiaremos los sistemas criptográficos basados en códigos, es decir, criptosistemas que usan una familia de códigos correctores de errores $\mathcal{C}$. Para ello, agregan un error a una palabra de alguno de los códigos $\mathcal{C}$ o al calcular un síndrome relativo a la matriz de paridad de $\mathcal{C}$.

El primer criptosistema desarrollado es un sistema de cifrado de clave pública y fue propuesto por Robert J. McEliece en 1978 [14]. Este sistema esencialmente usa como clave privada un código de Goppa binario aleatorio y, como clave pública, una matriz generadora de una versión permutada aleatoriamente de ese código. El texto cifrado es una palabra código a la que se le han añadido algunos errores, y solo el que posee la clave privada puede eliminar esos

errores en un tiempo razonable. Actualmente, no se conoce ningún ataque que presente una amenaza grave a este sistema, ni siquiera utilizando un ordenador cuántico. Cabe destacar la seguridad y la rapidez del criptosistema de McEliece, ya que tanto los procedimientos de cifrado y descifrado son de baja complejidad.

4.4.1 *Criptosistema de McEliece*

En 1978, R. J. McEliece propuso un criptosistema de clave pública [14]. El criptosistema de McEliece se basa en códigos lineales de corrección de errores y hasta ahora es el criptosistema más exitoso basado en nociones de teoría de codificación.

La construcción original en [14] usa códigos de Goppa binarios para cifrar y descifrar mensajes. Sin embargo, han surgido otras variantes de este criptosistema que usan otros códigos lineales, pero la mayoría han resultado ser inseguros. A día de hoy, la construcción original de 1978 se considera segura con la correcta elección de parámetros.

Por esta razón, el criptosistemade McEliece se pone a la altura del RSA. Existen algunas diferencias entre ambos criptosistemas, el de McEliece es capaz de cifrar y descifrar mensajes más rápido. Sin embargo, los tamaños de las claves son mayores que los del RSA, que es la razón por la que el criptosistema de McEliece apenas se usa. La principal razón por la que está creciendo el interés en el criptosistema de McEliece es porque es uno de los mejores candidatos para criptosistemas de clave pública seguros post-cuánticos. El esquema de Niederreiter [15], una variante del criptosistema de McEliece, también permite construir un esquema de firma digital segura [7, Sección 4] [6].

En esta sección presentaremos el criptosistema de McEliece incluyendo algunas de sus variantes y estudiaremos su seguridad y los ataques más conocidos.

4.4.1.1 *Construcción original*

El criptosistema de McEliece usa códigos lineales de corrección de errores para cifrar mensajes. Este criptosistema posee una clave privada, que se elige al generar la clave y contiene la descripción del código lineal estructurado, y una clave pública que se basa en una versión suficientemente aleatorizada de ese mismo código. De esta forma, será difícil descifrar un mensaje sin conocer la estructura del código lineal (clave privada), pues es lo que proporciona un descifrado rápido.

Los códigos que usa la construcción original son los códigos de Goppa binarios irreducibles. Estos códigos son muy adecuados pues poseen altas capacidades de corrección de errores y matrices generadoras densas, que son difíciles de distinguir de matrices binarias aleatorias.

4.4.1.2 *Generación de claves*

La generación de la clave de este criptosistema se obtiene a partir de los siguientes pasos:

1. Se elige un $[n, k, 2t+1]$-código lineal aleatorio $\mathcal{C}$ sobre $\mathbb{F}_2$ que tenga un algoritmo de decodificación eficiente $\mathcal{D}$ que sea capaz de corregir hasta t errores.
2. Se calcula la matriz generadora G de dimensión $k \times n$ para $\mathcal{C}$.
3. Se genera una matriz no singular binaria aleatoria S de dimensión $k \times k$.
4. Se genera una matriz de permutaciones aleatoria P de dimensión $n \times n$.
5. Se calcula la matriz $G' = SGP$ de dimensión $k \times n$. La clave pública es (G', t) y la clave privada es $(S, G, P, \mathcal{D})$.

Es decir, la matriz G' se obtiene al permutar las columnas de G a partir de la matriz P y luego realizar un cambio de base con la matriz S. De esta forma, la matriz G' corresponde a un $[n, k, 2t+1]$-código lineal que es equivalente permutacionalmente a la clave privada elegida. Llamaremos a G' la *matriz generadora pública*.

4.4.1.3 *Cifrado y descifrado*

Una vez tenemos la clave, podemos **cifrar** un texto plano $\mathbf{m} \in \{0,1\}^k$ eligiendo un vector aleatorio $\mathbf{e} \in \{0,1\}^n$ de peso t y calcular el texto cifrado como

$$\mathbf{c} = \mathbf{m}G' + \mathbf{e}.$$

Para recuperar $\mathbf{m}$ eficientemente, podemos usar el siguiente algoritmo de **descifrado**. Sea un criptograma $\mathbf{c} \in \{0,1\}^n$, primero calculamos

$$\mathbf{c}P^{-1} = (\mathbf{m}S)G + \mathbf{e}P^{-1}.$$

Ahora ya podemos aplicar el algoritmo de descifrado $\mathcal{D}$ a $\mathbf{c}P^{-1}$ para obtener $\mathbf{c'} = \mathbf{m}S$, pues $(\mathbf{m}S)G$ es una palabra código válida para el código lineal elegido y $\mathbf{e}P^{-1}$ tiene peso t. Finalmente, podemos calcular el mensaje $\mathbf{m}$ con

$$\mathbf{m} = \mathbf{c'}S^{-1}.$$

Por otra parte, a la hora de aplicar el algoritmo de Sugiyama, sabemos que este algoritmo de descifrado necesita saber el polinomio generador. De este modo consideraremos la clave privada como $(S, G, P, g(x))$, donde $g(x)$ es el polinomio de Goppa para el código elegido.

En el siguiente ejemplo vamos a usar el código descrito en el apéndice B para mostrar el cifrado de un mensaje y su correspondiente descifrado usando el criptosistema de McEliece que acabamos de estudiar.

Ejemplo 15. Sea $\mathbb{F}_{2^5}$ el cuerpo finito de 32 elementos y sea a un elemento de dicho cuerpo. Definimos el polinomio $g(x) = x^3 + (a^4 + a^3 + 1)x^2 + (a^4 + 1)x + a^4 + a^2 + a + 1 \in F_{2^5}[x]$ y tomamos $n = 20$. Podemos obtener el criptosistema de McEliece definido por el polinomio $g(x)$ sobre el cuerpo finito $\mathbb{F}_{2^5}$ y de tamaño $n = 20$ escribiendo lo siguiente:

```
sage: L = GF(2^5)
sage: a = L.gen()
sage: R.<x> = L[]
sage: g = x^3 + (a^4 + a^3 + 1)*x^2 + (a^4 + 1)*x + a^4 + a^2 + a + 1
sage: n = 20
sage: ME = McEliece(n, q, g)
sage: ME
> McEliece cryptosystem over [20, 5] Goppa code
```

Observamos que con los datos introducidos, el código de Goppa sobre el que se construye este criptosistema tiene longitud 20 y dimensión 5. Este criptosistema está formado por una clave privada $(S, G, P, g(x))$ y una clave pública (G', t).

La matriz S de la clave privada es una matriz no singular binaria aleatoria de dimensión 5×5, que podemos obtenerla con el método `get_S`:

```
sage: S = ME.get_S()
sage: show(S)
```

Este código devuelve la siguiente matriz:

$$S = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

De la misma forma, la matriz P, que es una matriz de permutaciones aleatoria de dimensión 20×20 es la siguiente:

```
sage: P = ME.get_P()
```

```
sage: show(P)
```

Este código devuelve la siguiente matriz:

$$P = \begin{pmatrix}
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Análogamente, podemos obtener la matriz G, que es la matriz generadora asociada al código de Goppa de dimensión 5×20:

```
sage: G = ME.get_G()
sage: show(G)
```

Este código devuelve la siguiente matriz:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}.$$

Podemos calcular la clave pública a partir de las matrices anteriores: $G' = SGP$. El método `get_public_key` nos permite obtener esta matriz:

```
sage: PK = ME.get_public_key()
sage: show(PK)
```

La matriz que nos devuelve el código anterior es la siguiente:

$$G' = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}.$$

Esta matriz nos permite cifrar mensajes. Durante este proceso, el criptosistema le añadirá errores aleatorios cuyo peso coincidirá con la mitad del grado del polinomio $g(x)$, en este caso se añadirá un error. Supongamos que queremos enviar el siguiente mensaje:

$$\mathbf{m} := (1,1,0,1,1) \in \{0,1\}^5.$$

Para ello, debemos escribir lo siguiente:

```
sage: m = vector(GF(2), (1, 1, 0, 1, 1))
sage: encrypted_message = ME.encrypt(m)
> (0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1)
```

Por lo que el mensaje $\mathbf{m}$ cifrado es $(0,1,1,1,0,0,0,0,1,1,0,1,1,1,1,1,1,1,0,1)$.

Ahora, usando la clave privada podemos decodificar el criptograma anterior:

```
sage: ME.decrypt(encrypted_message)
> (1, 1, 0, 1, 1)
```

Observamos que el código anterior devuelve $(1,1,0,1,1)$, que efectivamente coincide con el mensaje original $\mathbf{m}$.

4.4.2 *Criptosistema de Niederreiter*

En 1986, H. Niederreiter [15] propuso una variante relevante del criptosistema de McEliece. A diferencia del criptosistema de McEliece, este criptosistema usa una matriz de paridad en vez de una matriz generadora.

Antes de introducir este criptosistema, necesitamos definir el concepto de síndrome de una palabra en $\mathbb{F}^n$.

Definición 22. Sea $\mathcal{C}$ un $[n,k,d]$-código lineal sobre $\mathbb{F}$ y sea H su matriz de paridad. El *síndrome* de una palabra $\mathbf{y} \in \mathbb{F}^n$ se define como

$$\mathbf{s} = \mathbf{y}H^T.$$

Sabemos que una palabra código de $\mathcal{C}$ tiene síndrome igual a $\mathbf{0}$, según la definición de la matriz de paridad. Sean $\mathbf{y}_1$, $\mathbf{y}_2 \in \mathbb{F}^n$ dos vectores, entonces

$$\mathbf{y}_1 - \mathbf{y}_2 \in \mathcal{C} \Leftrightarrow (\mathbf{y}_1 - \mathbf{y}_2)H^T = \mathbf{0} \Leftrightarrow \mathbf{y}_1 H^T = \mathbf{y}_2 H^T.$$

El hecho de que $\mathbf{y}_1 - \mathbf{y}_2$ sea una palabra código de $\mathcal{C}$ si y solo si los síndromes de $\mathbf{y}_1$ e $\mathbf{y}_2$ son iguales es la base para un método eficiente para implementar la decodificación de la palabra código más cercana que se llama *decodificación del síndrome*.

Sea una palabra $\mathbf{y} \in \mathbb{F}^n$, un algoritmo de decodificación de síndrome $\mathcal{D}$ encuentra una palabra con el mínimo peso $\mathbf{e} \in \mathbb{F}^n$ tal que

$$\mathbf{y}H^T = \mathbf{e}H^T.$$

Si $\mathbf{y}$ tiene la forma $\mathbf{y} = \mathbf{c} + \mathbf{e'}$, donde $\mathbf{c} \in \mathcal{C}$ y $\mathbf{w}(\mathbf{e'}) \leq t$, entonces $\mathbf{e} = \mathbf{e'}$, esto es, el algoritmo de decodificación de síndrome encuentra exactamente el vector error que se le añadió a la palabra código.

Ahora, ya podemos definir el criptosistema de Niederreiter, que se basa en la idea de la decodificación del síndrome.

4.4.2.1 *Generación de claves*

La generación de la clave de este criptosistema se obtiene a partir de los siguientes pasos:

1. Se elige un $[n, k, 2t+1]$-código lineal aleatorio $\mathcal{C}$ sobre $\mathbb{F}_2$ que tenga un algoritmo de decodificación del síndrome eficiente $\mathcal{D}$ que sea capaz de corregir hasta t errores.
2. Se calcula la matriz de paridad H de dimensión $(n-k) \times n$ para $\mathcal{C}$.
3. Se genera una matriz no singular binaria aleatoria S de dimensión $(n-k) \times (n-k)$.
4. Se genera una matriz de permutaciones aleatoria P de dimensión $n \times n$.
5. Se calcula la matriz $H' = SHP$ de dimensión $(n-k) \times n$. La clave pública es (H', t) y la clave privada es $(S, H, P, \mathcal{D})$.

4.4.2.2 *Cifrado y descifrado*

Una vez tenemos la clave, podemos **cifrar** un texto plano $\mathbf{m} \in \{0,1\}^n$ con peso t, calculando el texto cifrado como el síndrome de $\mathbf{m}$

$$\mathbf{c} = \mathbf{m}H'^T.$$

Para recuperar $\mathbf{m}$ eficientemente, podemos usar el siguiente algoritmo de **descifrado**. Sea un criptograma $\mathbf{c}$, primero calculamos

$$S^{-1}\mathbf{c}^T = HP\mathbf{m}^T.$$

Luego, buscamos un vector $\mathbf{z} \in \mathbb{F}^n$ tal que $H\mathbf{z}^T = S^{-1}c^T$. Como además $H\mathbf{z}^T = HP\mathbf{m}^T$, entonces $\mathbf{z} - (P\mathbf{m}^T)^T = \mathbf{z} - \mathbf{m}P^T$ es una palabra código válida en $\mathcal{C}$ debido a que los síndromes de las palabras son iguales. Como $\mathbf{m}P^T$ tiene peso t, podemos aplicar $\mathcal{D}$ a $\mathbf{z}$ para encontrar el vector error $\mathbf{m}P^T$ y por lo tanto $\mathbf{m}$.

Observemos que el mensaje en texto plano se representa como el error de la palabra código en lugar de la palabra con la información original. De este modo, cualquier mensaje que queramos enviar se tendrá que codificar adicionalmente en un vector binario de peso t antes de cifrarlo.

A diferencia con el criptosistema de McEliece, el algoritmo de decodificación del síndrome es más eficiente que el algoritmo de decodificación del criptosistema de McEliece. Además, tambien se puede usar para construir un esquema de firma digital.

Sin embargo, si un ataque es capaz de romper el criptosistema de McEliece, también puede romper el esquema Niederreiter y viceversa. Esto se debe a que los criptosistemas de McEliece y Niederreiter se basan en el uso de códigos lineales que son duales entre sí y la matriz generadora se puede obtener eficientemente a partir de la matriz de paridad y viceversa.

4.4.2.3 *Firma digital*

Para obtener una firma digital eficiente necesitamos dos cosas: un algoritmo capaz de calcular la firma para cualquier documento de tal forma que identifiquen a un único autor, y un algoritmo de verificación rápida disponible para todo el mundo.

Sea h una función hash que devuelve una palabra binaria de longitud $n-k$ (la longitud del síndrome). Sea D nuestro documento, aplicamos la función h a dicho documento y definimos $s = h(D)$. Denotamos por $[\cdots s \cdots | \cdot i \cdot]$ la concatenación de s e i y $s_i = h([\cdots s \cdots | \cdot i \cdot])$.

El algoritmo de firma calculará s_i para i empezando en 0 e incrementándolo en una unidad hasta que uno de los síndromes s_i sea decodificable. Denotemos por i_0 al primer índice para el cual s_i es decodificable, y usaremos este síndrome para la firma. A continuación, la firma será el mensaje descifrado, esto es, la palabra z de longitud n tal que $Hz^T = s_{i_0}$. Sin embargo, la firma también tendrá que incluir el valor de i_0 para la verificación. La firma entonces será $[\cdots z \cdots | \cdot i_0 \cdot]$.

El algoritmo de verificación es mucho más simple (y rápido). Este algoritmo consiste en aplicar la función de cifrado público a la firma y verificar que el resultado es el valor hash del documento. Esto es, primero calculamos $s_1 = Hz^T$ con la clave pública H. Luego, calculamos $s_2 = h([\cdots h(D) \cdots | \cdot i_0 \cdot])$ con la función hash pública. Finalmente, si s_1 y s_2 son iguales entonces la firma es válida.

4.4.3 *Seguridad del criptosistema de McEliece*

En esta sección estudiaremos la seguridad del criptosistema de McEliece original y analizaremos los ataques más conocidos ante este criptosistema. Como la seguridad de los esquemas de McEliece y Niederreiter es equivalente [13], todos los ataques que analicemos también estarán relacionados implícitamente con el esquema Niederreiter.

Fijemos un código de Goppa $\Gamma(L, g) \subset \mathbb{F}_2^n$, con $g(x) \in \mathbb{F}_{2^m}[x]$ y L una tupla de n elementos distintos de $\mathbb{F}_{2^m}$, capaz de corregir hasta t errores. Tenemos que la dimensión del código es $k = n - tm$. Sea G la matriz generadora binaria de dimensión $k \times n$ de Γ y definimos $G' = SGP$ como la clave pública de McEliece, donde S es la matriz binaria no singular de dimensión $k \times k$ y P es la matriz de permutaciones de dimensión $n \times n$.

Para estudiar la seguridad de este criptosistema, tenemos que determinar cómo de difícil es determinar el mensaje $\mathbf{m}$ a partir de conocer G' y haber interceptado $\mathbf{c}$. Como se indica en [14], pueden darse dos ataques básicos:

1. Intentar recuperar la clave secreta G a partir de G' y así descifrar el mensaje.

2. Intentar recuperar el mensaje original $\mathbf{m}$ a partir del criptograma $\mathbf{c}$ sin conocer la clave privada G.

El primer ataque parece inviable si n y t son suficientemente grandes, pues existen demasiadas posibilidades tanto para G como para S y P.

El segundo ataque puede ser más prometedor para el adversario pues puede aproximarse mediante la decodificación de conjuntos de información.

La seguridad del criptosistema de McEliece se puede insinuar por la intratabilidad de los siguientes problemas fundamentales en la teoría de la codificación.

Problema 3 (Problema general de decodificación para códigos lineales)**.** *Sea $\mathcal{C}$ un $[n,k]$-código lineal sobre $\mathbb{F}$ e $\boldsymbol{y} \in \mathbb{F}^n$. Encontrar una palabra código $\boldsymbol{c} \in \mathcal{C}$ tal que la distancia $d(\boldsymbol{y}, \boldsymbol{c})$ sea mínima.*

Problema 4 (Problema de encontrar una palabra código dado un peso)**.** *Sea $\mathcal{C}$ un $[n,k]$-código lineal sobre $\mathbb{F}$ y $w \in \mathbb{N}$. Encontrar una palabra código $\boldsymbol{c} \in \mathcal{C}$ tal que $\boldsymbol{w}(\boldsymbol{c}) = w$.*

Se ha demostrado que ambos problemas son NP-duro [1]. No obstante, esto no implica que romper el criptosistema de McEliece sea NP-duro, pues los códigos binarios de Goppa solo cubren una fracción de todos los códigos lineales posibles. Es por esto que la seguridad del criptosistema de McEliece se basa en la suposición de que la clave pública es indistinguible de cualquier matriz aleatoria.

4.4.3.1 *Seguridad post-cuántica*

El criptosistema de McEliece es inmune ante el algoritmo de Shor [18], que si se implementara en un ordenador cuántico podría romper otros criptosistemas de clave pública tales como RSA.

5

ATAQUE USANDO ALGORITMOS GENÉTICOS

En el capítulo 2 vimos que la capacidad de corrección de un código depende de la distancia de un código. Sin embargo, encontrar una palabra con distancia mínima de un código lineal no es una tarea fácil. Recordemos que el problema de decisión asociado al cálculo de la distancia mínima de un código lineal binario es NP-completo.

En esta sección estudiaremos varios ataques usando algoritmos genéticos basados en el cálculo de la distancia de un código propuestos en [8]. Probaremos que, dada una matriz generadora, existe una permutación de las columnas cuya matriz escalonada contiene una fila cuyo peso es la distancia del código. Los algoritmos trabajarán con un espacio de soluciones igual al conjunto de posibles permutaciones de las columnas de la matriz generadora. En concreto, adaptaremos e implementaremos algoritmos genéticos generacionales (GGA) y CHC para revolver este problema.

5.1 ESQUEMA BASADO EN PERMUTACIONES

Sea $\mathbb{F}_q$ un cuerpo finito con q elementos y sean k, n dos números enteros tales que $0 < k \leq n$. Denotamos por $\mathcal{S}_n$ al conjunto de permutaciones de n símbolos. Decimos que dos $[n,k]_q$-códigos lineales $\mathcal{C}_1$ y $\mathcal{C}_2$ son *equivalentes permutacionalmente* si son iguales con una permutación fija de las coordenadas de una palabra código, esto es, si existe una permutación $x \in \mathcal{S}_n$ tal que $(c_0, ..., c_{n-1}) \in \mathcal{C}_1$ si y solo si $(c_{x(0)}, ..., c_{x(n-1)}) \in \mathcal{C}_2$. Observemos que G es la matriz generadora de $\mathcal{C}_1$ si y solo si GP es la matriz generadora de $\mathcal{C}_2$, donde P es la matriz asociada a la permutación x. Los códigos equivalentes permutacionalmente tienen en común la distancia mínima, ya que la permutación de sus componentes no modifica el peso del vector.

El siguiente resultado nos proporciona un método para calcular la distancia mínima a partir de la matriz escalonada (*reduced row echelon form*) de la matriz generadora del código equivalente.

Teorema 14. *Sea G una matriz generadora de dimensión $k \times n$ de un $[n,k]_q$-código lineal $\mathcal{C}$ sobre el cuerpo finito $\mathbb{F}_q$. Existe una permutación $x \in \mathcal{S}_n$ tal que la matriz escalonada, R, de GP_x, donde P_x es la matriz asociada a la permutación x, cumple que el peso de alguna de sus filas alcanza la distancia mínima de $\mathcal{C}$. Por lo tanto, si b es una fila de R verificando esa propiedad, entonces bP_x^{-1} es una palabra código no nula de $\mathcal{C}$ con peso mínimo.*

Demostración. Sea d la distancia mínima de $\mathcal{C}$ y sea $a \in \mathcal{C}$ una palabra código tal que $w(a) = d$. Como a es distinto de cero, existe una matriz regular A de dimensión $k \times k$ sobre $\mathbb{F}$ tal que

$$AG = \left(\frac{G_1}{a}\right)$$

para alguna matriz G_1 de dimensión $(k-1) \times n$ sobre $\mathbb{F}$. Por otra parte, como $w(a) = d$, existe una permutación $x \in \mathcal{S}_n$ que mueve las entradas no nulas de a a las últimas posiciones, esto es,

$$aP_x = (0, ..., 0, a_1, ..., a_d),$$

donde $a_1, ..., a_d \in \mathbb{F}_q \backslash \{0\}$. Luego

$$AGP_x = \left(\frac{G_1P_x}{aP_x}\right) = \left(\frac{G_1P_x}{0\cdots 0 | a_1 \cdots a_d}\right).$$

Ahora, definimos la matriz A' de dimensión $(k-1) \times (k-1)$ sobre $\mathbb{F}$ de tal forma que sea invertible y que $A'G_1P_x = H'$ sea una matriz escalonada. Tenemos que

$$\left(\begin{array}{c|c} A' & 0 \\ \hline 0 & 1 \end{array}\right) AGP_x = \left(\begin{array}{c|c} A' & 0 \\ \hline 0 & 1 \end{array}\right) \left(\begin{array}{c|c} G_1P_x & \\ \hline 0\cdots 0 & a_1 \cdots a_d \end{array}\right) = \left(\begin{array}{c|c} H' & \\ \hline 0\cdots 0 & a_1 \cdots a_d \end{array}\right). \quad (29)$$

La última fila de H' es distinta de cero, pues la matriz G_1 tiene rango $k-1$. Supongamos que el pivote de esta fila está en la columna i_0-ésima. Supongamos que $i_0 \geq n - d + 1$, entonces las últimas dos filas de (29) son linealmente independientes y las coordenadas no nulas se encuentran en las últimas d coordenadas. Por tanto, existe una combinación lineal de ambas cuyo peso es menor que d, lo que lleva a una contradicción. Luego se tiene que $i_0 < n - d + 1$ y en consecuencia, la última fila de (29) coincide con la última fila de la matriz escalonada de GP_x, salvo multiplicación escalar no nula, como queríamos. □

Este teorema expone que encontrar la distancia mínima de un $[n,k]_q$-código lineal se reduce a encontrar el mínimo de la aplicación $\mathfrak{d} : \mathcal{S}_n \to \mathbb{N}$ definida por

$$\mathfrak{d}(x) = \text{mín}\{\text{w}(b) \ : \ b \text{ es una fila de la matriz escalonada de } FP_x\}$$

para cualquier $x \in \mathcal{S}_n$, donde G es la matriz generadora del código y P_x representa la matriz asociada a la permutación x.

Sin embargo, existen pocas permutaciones que nos proporcionarán el mínimo de $\mathfrak{d}$. En concreto, supongamos que solo hay una palabra código que alcanza el mínimo peso d de $\mathcal{C}$ salvo multiplicación escalar.

En [8] se obtiene que la probabilidad de encontrar el mínimo de $\mathfrak{d}$ por una búsqueda aleatoria es

$$\frac{d!(n-d)!}{n!} = \binom{n}{d}^{-1}.$$

Ejemplo 16. Vamos a ilustrar el esquema descrito para obtener el peso mínimo de un código a partir de su matriz escalonada. Sea $\mathcal{C}$ un $[8,4]_4$-código lineal sobre el cuerpo finito $\mathbb{F}_4 = \{0, 1, a, a+1\}$ con matriz generadora

$$G = \begin{pmatrix} 1 & 0 & a+1 & a+1 & a+1 & 0 & 0 & a \\ a+1 & 0 & 0 & a+1 & a & a+1 & 1 & a+1 \\ 1 & a+1 & 1 & a+1 & a & 0 & a & 0 \\ 0 & 0 & 0 & a & a & a & a+1 & a \end{pmatrix}.$$

Sea $x \in \mathcal{S}_8$ la permutación dada por $x(0) = 1$, $x(1) = 0$, $x(2) = 3$, $x(3) = 2$, $x(4) = 5$, $x(5) = 4$, $x(6) = 7$ y $x(7) = 6$. La matriz asociada a la permutación x es la siguiente:

$$P_x = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Aplicamos esta permutación a la matriz generadora G para obtener la matriz generadora G^x del código equivalente permutacionalmente $\mathcal{C}_x$:

$$G^x = \begin{pmatrix} 0 & 1 & a+1 & a+1 & 0 & a+1 & a & 0 \\ 0 & a+1 & a+1 & 0 & a+1 & a & a+1 & 1 \\ a+1 & 1 & a+1 & 1 & 0 & a & 0 & a \\ 0 & 0 & a & 0 & a & a & a & a+1 \end{pmatrix}.$$

La matriz escalonada de G^x es

$$G^x = \begin{pmatrix} 1 & 0 & 0 & 0 & a+1 & 0 & a & a \\ 0 & 1 & 0 & 0 & 0 & a & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & a \\ 0 & 0 & 0 & 1 & 1 & a+1 & a & a \end{pmatrix},$$

donde el peso de la primera fila es 4, el de la segunda es 2 y el de la tercera y cuarta es 5. Como la segunda fila tiene peso mínimo entre todas las filas, tenemos que $\mathfrak{d}(x) = 2$. Realmente, la distancia de $\mathcal{C}$ es 2, luego x alcanza el mínimo de $\mathfrak{d}$.

5.2 ALGORITMOS GENÉTICOS

En esta sección estudiaremos las adaptaciones de los algoritmos GGA y CHC propuestos en [8] para resolver el problema de encontrar la distancia mínima de un código lineal. Además, implementaremos ambos algoritmos y analizaremos los resultados.

La representación clásica de estos algoritmos es la discreta. Esto es, fijado un código $\mathcal{C}$ de longitud n y dimensión k sobre un cuerpo finito con q elementos $\mathbb{F}_q$, consideramos su matriz generadora G. Buscaremos una k-tupla $m = (m_1, ..., m_k) \in \mathbb{F}_q^k$ con $m \neq 0$ de tal forma que el peso correspondiente a la palabra código mG sea mínimo. De este modo, cada solución (cromosoma) x estará asignada a un mensaje m como un fenotipo, mientras que los genotipos se componen de k variables de decisión o genes cuyos valores están en el conjunto $\mathbb{F}_q$. De esta forma, la función fitness definida en $\mathbb{F}_q^k \longrightarrow \mathbb{N}$ consistirá en minimizar el peso que producen las soluciones que componen la población.

No obstante, según el resultado de la sección 5.1 podemos cambiar el espacio de las soluciones del planteamiento tradicional del problema al espacio de las permutaciones de n elementos, $\mathcal{S}_n$, donde n es la longitud del código lineal que estudiaremos. Esto tiene una consecuencia inmediata: nos permite cambiar el problema de encontrar la distancia mínima $d(\mathcal{C})$ de un código $\mathcal{C}$, para que la búsqueda de la palabra código con peso mínimo equivalga a encontrar una permutación $x \in \mathcal{S}_n$ adecuada de las columnas de una matriz generadora de tal forma que la fila r de su matriz escalonada tenga peso $w(r) = d(\mathcal{C})$. Así, las soluciones (cromosomas) se representarán como permutaciones x (fenotipos) de las n columnas de la matriz generadora G, y que contienen n variables de decisión (genotipos) cuyos valores pueden ser $\{1, ..., n\}$, donde cada número representa una posición de cada columna de la matriz. Esto es, dada una permutación $x = (x_1, ..., x_n)$, un valor x_i significa que la columna de la matriz que se encuentra en la posición i cambiará su ubicación a la columna en la posicición x_i en la permutación.

Por tanto, el fitness de una permutación $x \in \mathcal{S}_n$ se puede calcular como

$$f(x) = \text{mín}\{w(r_i) : r_i \text{ es una fila de la matriz escalonada de } GP_x\},$$

donde P_x denota a la matriz de permutaciones de x.

5.2.1 *Algoritmo GGA*

El algoritmo GGA comienza con la inicialización de la población $P(t)$ con N soluciones aleatorias y las evalúa en la iteración $t = 0$. Luego, el principal bucle del algoritmo se ejecuta hasta que se cumpla la condición de parada, que en este caso será un número máximo de generaciones.

El bucle principal empieza seleccionando N padres según el operador de selección de torneo binario. Este operador consiste en seleccionar dos individuos de la población aleatorios y seleccionar como padre al individuo que tenga mejor fitness.

Después, comienza la etapa de generación de una nueva población. De esta forma, se le aplica el operador de cruce a dos padres elegidos aleatoriamente para generar un nuevo par de soluciones con probabilidad p_c. Este operador consiste en componer ambos padres, es decir, sean p_1 y p_2 dos padres, el operador de cruce generará dos descencientes d_1 y d_2 a partir de la composición de cada uno en distintos órdenes:

$$d_1 = p_1 \circ p_2,$$

$$d_2 = p_2 \circ p_1.$$

Si no se combinan, se le aplica el operador de mutación a cada padre para generar una solución mutada. Este operador elige una columna de las primeras k columnas y otra de las $n - k$ columnas restantes de la matriz generadora y las permuta para crear un nuevo descendiente.

En ambos casos, habrá que comprobar que las nuevas soluciones son válidas. Esto es, una solución no válida en el algoritmo GGA es la solución nula $(0, ..., 0)$. De esta forma, todas las nuevas soluciones N se habrán generado por cruce o mutación y formarán la población de la siguiente iteración $P(t+1)$. Finalmente, se evalúan las soluciones de $P(t+1)$.

Se ha incluído una componente elitista antes de que comience la siguiente iteración: si la solución $P(t+1)$ no tiene un fitness igual o superior que la mejor en $P(t)$, entonces la peor solución de $P(t+1)$ es reemplazada por la mejor solución de $P(t)$. Además, si se alcanza un número fijo de evaluaciones de las soluciones que no producen mejora en el fitness de la mejor solución encontrada, se reiniciará $P(t+1)$ con N nuevas soluciones aleatorias.

En resumen, el Algoritmo GGA consiste en:

Input: N: número par con el tamaño de la población
Input: p_c: probabilidad de cruce
Input: *max_reinit*: número de evaluaciones de las soluciones sin mejorar el fitness antes de la reinicialización
Output: Mejor solución de $P(t)$

```
1  t ⟵ 0
2  Inicializar la población P(t) con N soluciones aleatorias válidas
3  Evaluar las soluciones de P(t)
4  while no se cumpla la condición de parada do
5      P(t+1) ⟵ ∅
6      padres(1,...,N) ⟵ Seleccionar N soluciones de P(t) con selección de torneo binario
7      for i in 0..N/2−1 do
8          if número aleatorio de la distribución uniforme [0,1] es menor que p_c then
9              c1, c2 ⟵ soluciones generadas a partir del cruce de los padres padres(2i) y
                padres(2i+1)
10         else
11             c1 ⟵ mutación del padre padres(2i)
12             c2 ⟵ mutación del padre padres(2i+1)
13         end
14         if c1 (resp. c2) no es válido then
15             reemplazar c1 (resp. c2) con una solución aleatoria válida
16         end
17         P(t+1) ⟵ P(t+1) ⋃ {c1, c2}
18     end
19     Evaluar las soluciones de P(t+1)
20     if ningún fitness de P(t+1) es igual o superior que el mejor fitness de P(t) then
21         Reemplazar la peor solución de P(t+1) con la mejor solución de P(t)
22     end
23     if max_reinit soluciones han sido evaluadas sin mejorar la mejor solución de P(t+1) then
24         Reemplazar las soluciones de P(t+1) con N−1 soluciones aleatorias y la mejor
            solución de P(t)
25     end
26     t ⟵ t+1
27 end
```

Algoritmo 5: Algoritmo GGA.

5.2.2 *Algoritmo CHC*

El algoritmo CHC es un algoritmo evolutivo cuya versión inicial fue propuesta para codificación binaria [10]. Este algoritmo mantiene un equilibrio entre la diversidad de los genotipos en las soluciones de la población y la convergencia a óptimos locales. Se basa en cuatro componentes similares: la selección elitista, el operador de recombinador de soluciones HUX, una verificación de prevención de incesto para evitar la recombinación de soluciones similares y un método de reiniciación de la población cuando se encuentra un óptimo local. En [8] se propone una adaptación de este algoritmo, que es la que vamos a estudiar.

Para calcular la distancia entre dos soluciones de la población usaremos la distancia de Hamming (líneas 4-5 y 22-23 del algoritmo 6). En cuanto al decremento *dec* del cruce (líneas 5 y 21), se actualiza como un porcentaje τ de la máxima distancia de Hamming entre los individuos de la población, donde $\tau \in [0, 1]$ es la tasa de actualización, un parámetro de entrada al algoritmo. Si se produce una solución no válida en el cruce, se le asignará un fitness igual a infinito para asegurarnos de que no se incluirá en la población de la siguiente generación.

El algoritmo CHC comienza con la inicialización de la población $P(t)$ con N soluciones aleatorias y las evalúa en la iteración $t = 0$. Luego, se calcula la distancia media y la máxima entre todas las soluciones. Así, la variable d se inicializará a la distancia media y dec se inicializará a la máxima distancia multiplicada por τ. Luego, el principal bucle del algoritmo se ejecuta hasta que se cumpla la condición de parada, que en este caso será un número máximo de generaciones.

El bucle principal empieza seleccionando N padres aleatorios entre las soluciones de $P(t)$.

Después, comienza la etapa de generación de una nueva población. De esta forma, se le aplica el operador de cruce a los padres que estén a una distancia menor que el umbral de distancia d. Este operador es análogo al operador de cruce del algoritmo GGA. Una vez generados todos los descendientes, se evaluarán. A continuación, se formará la población de la siguiente iteración $P(t+1)$ que contendrá a N individuos, escogidos entre las mejores soluciones de $P(t)$ y las nuevas soluciones generadas por el cruce. Sin embargo, puede que $P(t+1)$ y $P(t)$ sean iguales, en este caso el umbral de distancia d se decrementa en dec. Si este umbral d alcanza un valor negativo o nulo, la población se reinicializará. Para ello, conservaremos la mejor solución de $P(t)$ en la siguiente población y el resto la reemplazaremos por $N-1$ soluciones aleatorias. Los valores d y dec se recalcularán para esta nueva población.

En resumen, el Algoritmo CHC consiste en:

Input: N: número par con el tamaño de la población
Input: τ: frecuencia de actualización del umbral de cruce
Output: Mejor solución de $P(t)$

```
1  t ⟵ 0
2  Inicializar la población P(t) con N soluciones aleatorias válidas
3  Evaluar las soluciones de P(t)
4  d ⟵ Media de la distancia de las soluciones de P(t)
5  dec ⟵ τ· Distancia máxima de las soluciones de P(t)
6  while no se cumpla la condición de parada do
7      C(t) ⟵ ∅
8      padres(1, ..., N) ⟵ Seleccionar N soluciones aleatorias de P(t)
9      for i in 0..N/2 − 1 do
10         if distancia(padres(2i), padres(2i + 1)) < d then
11             c1, c2 ⟵ soluciones generadas a partir del cruce de los padres padres(2i) y
                padres(2i + 1)
12             C(t) ⟵ C(t) ⋃ {c1, c2}
13         end
14     end
15     Evaluar las soluciones de C(t)
16     P(t + 1) ⟵ Mejores N soluciones de C(t) ∪ P(t)
17     if P(t) = P(t + 1) then
18         d ⟵ d − dec
19         if d ≤ 0 then
20             Inicializar P(t + 1) con la mejor solución de P(t) y N − 1 soluciones aleatorias
21             Evaluar las nuevas soluciones de P(t + 1)
22             d ⟵ Media de la distancia de las soluciones de P(t + 1)
23             dec ⟵ τ· Distancia máxima de las soluciones de P(t + 1)
24         end
25     end
26     t ⟵ t + 1
27 end
```

Algoritmo 6: Algoritmo CHC.

5.2.3 *Ejemplos*

Cabe destacar que los algoritmos genéticos GGA y CHC han logrado romper el criptosistema de McEliece hasta una longitud de $n = 128$. Para ilustrar el funcionamiento de ambos algorit-

mos, a continuación vamos a mostrar un ejemplo de sus ejecuciones, en el que intentaremos vulnerar la seguridad del criptosistema de McEliece para dicha longitud.

Ejemplo 17. Sea $\mathbb{F}_{2^7}$ el cuerpo finito de 128 elementos y sea a un elemento de dicho cuerpo. Tomando $n = 128$, vamos a calcular un polinomio irreducible y mónico aleatorio de grado 8:

```
sage: n = 128
sage: q = 7
sage: F = GF(2)
sage: L = GF(2^q)
sage: a = L.gen()
sage: R.<x> = L[]
sage: g = R.random_element(8)
sage: while g.is_irreducible() == False:
+         g = R.random_element(8)
sage: g = g*g.leading_coefficient()^-1
sage: g
> x^8 + (z7^6 + z7^5 + z7^2 + z7)*x^7 + x^6 + (z7^5 + z7^4 + z7^2 +
  1)*x^5 + (z7^2 + z7)*x^4 + x^3 + (z7^6 + z7^5 + z7^2 + z7 + 1)*x^2 +
  (z7^5 + z7^4 + z7^3)*x + z7^3
```

Podemos definir el criptosistema de McEliece a partir de estos parámetros con las siguientes órdenes:

```
sage: ME = McEliece(n, q, g); ME
> McEliece cryptosystem over [128, 72] Goppa code
```

Ahora, encriptamos el siguiente mensaje:

$$\begin{aligned} \mathit{message} := (&1,1,0,1,1,1,0,0,0,0,1,1,1,0,1,1,0,1,0,1,1,1,0,1,1,1,0,0,0,1,1,1,1,0,1,1,1,1,0,\\ &0,1,1,1,0,1,0,0,1,0,0,0,1,1,0,1,1,0,1,0,0,0,0,1,0,1,1,0,0,0,0,1,1), \end{aligned} \tag{30}$$

que posteriormente averiguaremos.

```
sage: message = vector(F, [1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1,
+     0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0,
+     0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0,
+     1, 0, 1, 1, 0, 0, 0, 0, 1, 1])
sage: encrypted_message = ME.encrypt(message)
```

El método `encrypt()` cifra el mensaje *message* multiplicándolo por la clave pública PK y añadiéndole un error aleatorio e al resultado.

A continuación, añadimos una fila a la clave pública PK que contendrá el mensaje encriptado, llamaremos a esta matriz G.

```
sage: PK = ME.get_public_key()
sage: G = PK.stack(encrypted_message)
```

La matriz G define un código lineal que tiene una distancia mínima $t = \mathrm{gr}(g) = 8$, y cuyo peso mínimo se obtiene en una única palabra: el error e. Para averiguar dicho error, debemos ejecutar los algoritmos genéticos GGA y CHC, los cuales nos darán una permutación p con fitness t.

Primero, vamos a ejecutar el algoritmo genético GGA para un tamaño de población $N = 60$, con probabilidad de cruce $p_c = 0{,}8$, con un máximo de evaluaciones de $max\_reinit = 30$ y con un fitness mínimo igual a $min\_fitness = 8$.

```
sage: p, f = GGA(G, 60, 0.8, 30, 8)
```

Obtenemos una permutación p que, tras aplicarla, existirá una fila con fitness mínimo $f = 8$ a partir de la cual podremos obtener el error. Por tanto, primero debemos calcular la matriz P asociada a la permutación p, aplicársela a la matriz generadora G y obtener la matriz escalonada reducida del resultado M:

```
sage: P = permutation_matrix(p)
sage: M = (G*P).rref()
```

Obtenemos ahora la fila de la matriz M que tenga peso f:

```
sage: for row in M:
+         if fitness_vector(row) == f:
+             v = vector(GF(2), row)
+             break
```

Ahora ya podemos obtener el vector error si deshacemos la permutación al vector v que acabamos de calcular:

```
sage: e = v * P^(-1)
```

Una vez obtenido el error, se lo restamos al criptograma y resolvemos el sistema $y - e = mPK$,

```
sage: z = encrypted_message - e
sage: m = PK.solve_left(z)
sage: m
>  (1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1,
    1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1,
    0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1)
sage: m == message
> True
```

Observamos que el mensaje m que obtenemos coincide con el mensaje original (30). Por lo que hemos conseguido romper el criptosistema de McEliece para $n = 128$ con el algoritmo genético GGA.

Ahora, ejecutaremos el algoritmo genético CHC. Este algoritmo lo ejecutaremos para un tamaño de población $N = 60$, con $tau = 0{,}7$ y con un fitness mínimo igual a $min\_fitness = 8$. Una vez ejecutado dicho algoritmo, los pasos son análogos.

```
sage: p, f = CHC(G, 350, 0.8, 8)
sage: P = permutation_matrix(p)
sage: M = (G*P).rref()
sage: for row in M:
+         if fitness_vector(row) == f:
+             v = vector(GF(2), row)
+             break
sage: e = v * P^(-1)
```

Una vez obtenido el error, se lo restamos al criptograma y resolvemos el sistema $y - e = mPK$,

```
sage: z = encrypted_message - e
sage: m = PK.solve_left(z)
sage: m
>  (1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1,
    1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1,
    0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1)
sage: m == message
> True
```

Observamos que, igualmente, el mensaje m que obtenemos coincide con el mensaje original (30). Por lo que hemos conseguido romper el criptosistema de McEliece para $n = 128$ con el algoritmo genético CHC.

CONCLUSIÓN Y VÍAS FUTURAS

El objetivo principal de este trabajo se ha alcanzado satisfactoriamente, esto es, se ha conseguido realizar un criptoanálisis al criptosistema de McEliece con algoritmos genéticos. Para ello, se han adquirido los conocimientos necesarios para lograr estudiar e implementar este sistema a partir de los códigos de Goppa, para el cual ha sido necesario realizar un desarrollo previo de los conceptos matemáticos e informáticos tales como los cuerpos finitos, los anillos de polinomios sobre cuerpos finitos y los códigos lineales. También se han adquirido los conocimientos necesarios para llevar a cabo el criptoanálisis, como el estudio que se ha realizado para la criptografía basada en códigos como modelo de criptografía post-cuántica y el estudio de algoritmos evolutivos para el cálculo de la distancia de un código lineal. Además, se ha logrado que las implementaciones en SageMath de los códigos de Goppa, del criptosistema de McEliece clásico y de los algoritmos genéticos funcionen correctamente.

Como desarrollo posterior, sería interesante contribuir con las implementaciones de los códigos de Goppa y del criptosistema de McEliece a SageMath, pues solo dispone de los códigos de Goppa binarios. En cuanto a los algoritmos genéticos que hemos empleado, se ha conseguido romper el criptosistema de McEliece hasta una longitud igual a $n = 128$. Sin embargo, pienso que se podrían mejorar para conseguir romper una longitud mayor, ya sea modificando los operadores de cruce y/o mutación, o añadir más diversidad para que no se estanquen en mínimos locales, o incluso mejorar su eficiencia calculando menos números aleatorios y/o permutaciones de las matrices.

A

IMPLEMENTACIÓN EN SAGEMATH DE LOS CÓDIGOS DE GOPPA

En este anexo describimos la documentación de las clases desarrolladas para implementar en SageMath los códigos de Goppa. Para ello, primero se ha implementado la clase `Goppa` que hereda de la clase `AbstractLinearCode` de SageMath, que permite representar los aspectos básicos de los códigos de Goppa, tales como la matriz de paridad y la matriz generadora. Para representar la codificación de los códigos de Goppa ha sido necesario crear la clase `GoppaEncoder`, que hereda de la clase `Encoder` y es capaz de proporcionar una palabra código a partir de un mensaje. Finalmente, se ha implementado la clase `GoppaDecoder`, que hereda de la clase `Decoder`, para simular la decodificación de los códigos de Goppa, esto es, se han desarrollado métodos para calcular el síndrome de una palabra codificada y el algoritmo de Sugiyama para transformar una palabra codificada con posibles errores en la palabra código original y en el mensaje original. También se describen las funciones auxiliares desarrolladas.

Para usar las clases que acabamos de describir, es necesario cargar el fichero con la orden `load()` tal y como se indica a continuación:

```
sage: load(Goppa.sage)
```

El código desarrollado se encuentra en

https://github.com/paula1999/TFG/tree/main/src.

A.1 CLASE PARA CÓDIGOS DE GOPPA

Esta clase simula el comportamiento básico de los códigos de Goppa, es decir, proporciona métodos para calcular el código de Goppa definido por un conjunto de definición y un polinomio sobre un cuerpo finito. Además, proporciona métodos para obtener las matrices de paridad y generadora.

`class Goppa(self, defining_set, generating_pol, field)`

Hereda de: `AbstractLinearCode`

Representación de un código de Goppa como un código lineal.

ARGUMENTOS

`field` Cuerpo finito sobre el que se define el código de Goppa.

`generating_pol` Polinomio mónico con coeficientes en un cuerpo finito $GF(p^m)$ que extiende de `field`.

`defining_set` Tupla de n elementos distintos de $GF(p^m)$ que no son las raíces de `generating_pol`.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: C
> [10, 4] Goppa code
```

`get_generating_pol(self)`

Calcula el polinomio generador del código.

SALIDA

El polinomio generador del código.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: C.get_generating_pol()
```

```
> x^{3} + z_{4} x^{2} + 1
```

get_defining_set(self)

Calcula el conjunto de definición del código.

SALIDA

El conjunto de definición del código.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: C.get_defining_set()
> [z4^3 + z4^2, 0, z4^2 + 1, z4^2 + z4 + 1, z4 + 1, z4^3 + z4 + 1, z4^2
    + z4, z4^3 + z4^2 + 1, z4^3 + z4^2 + z4 + 1, z4^3 + z4]
```

get_parity_pol(self)

Calcula el polinomio de paridad del código.

SALIDA

El polinomio de paridad del código.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: C.get_parity_pol()
```

```
> [(z4^3 + z4^2 + 1)*x^2 + (z4^3 + z4)*x + 1, x^2 + z4*x, z4^3*x^2 + (
   z4^3 + z4^2 + 1)*x + z4^3 + z4^2, (z4^3 + z4)*x^2 + z4^2*x + z4^3 +
    z4^2 + z4 + 1, (z4^3 + z4^2 + 1)*x^2 + (z4^3 + z4^2 + 1)*x + z4^2,
    (z4^3 + z4)*x^2 + (z4^2 + 1)*x + 1, (z4 + 1)*x^2 + (z4^3 + z4^2)*x
    + z4^3 + z4^2 + z4, (z4^3 + z4^2 + 1)*x^2 + (z4^2 + z4 + 1)*x + z4
   ^2 + 1, (z4^3 + z4^2)*x^2 + (z4 + 1)*x + z4, z4^2*x^2 + (z4^2 + z4)
   *x + z4^3 + 1]
```

get_parity_check_matrix(self)

Calcula la matriz de paridad del código.

SALIDA

La matriz de paridad del código.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: C.get_parity_check_matrix()
> [1    0      0      1    z2      1      0 z2 + 1     0 z2 + 1]
  [0    0     z2 z2 + 1     1      0 z2 + 1      1     1 z2 + 1]
  [z2   0      1     z2     1 z2 + 1      0 z2 + 1     1     z2]
  [z2   1     z2      1    z2      1     z2      0     1      0]
  [1    1     z2     z2     1     z2      1      1     0     z2]
  [z2   0 z2 + 1     z2    z2     z2      1     z2    z2      1]
```

get_generator_matrix(self)

Calcula la matriz generadora del código.

SALIDA

La matriz generadora del código.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: C.get_generator_matrix()
> [1   0   0   0   z2   1      z2    z2   z2 + 1   z2 + 1]
  [0   1   0   0   z2   0       0    z2       z2   z2 + 1]
  [0   0   1   0    0   0  z2 + 1    z2   z2 + 1       z2]
  [0   0   0   1   z2  z2      z2    z2   z2 + 1       z2]
```

get_dimension(self)

Calcula la dimensión del código.

SALIDA

La dimensión del código.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: C.get_dimension()
> 4
```

A.2 CODIFICADOR PARA CÓDIGOS DE GOPPA

En esta sección presentaremos la implementación del codificador para los códigos de Goppa. Proporciona un método para codificar un mensaje.

`class GoppaEncoder(self, code)`

Hereda de: `Encoder`

Representación de un codificador para un código de Goppa usando su matriz generadora.

ARGUMENTOS

`code` Código asociado a este codificador.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: GoppaEncoder(C)
> Encoder for [10, 4] Goppa code
```

`get_generator_matrix(self)`

Devuelve la matriz generadora del código asociado a `self`.

SALIDA

La matriz generadora del código asociado.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
```

```
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: E = GoppaEncoder(C)
sage: E.get_generator_matrix()
> [1   0   0   0   z2   1      z2    z2   z2 + 1   z2 + 1]
  [0   1   0   0   z2   0       0    z2       z2   z2 + 1]
  [0   0   1   0    0   0  z2 + 1    z2   z2 + 1       z2]
  [0   0   0   1   z2  z2      z2    z2   z2 + 1       z2]
```

encode(self, m)

Transforma `m` en una palabra código del código asociado a `self`.

ARGUMENTOS

`m` Vector asociado a un mensaje de `self`.

SALIDA

Una palabra código del código asociado a `self`.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^6)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^5 + a*x^2 + 1
sage: n = 30
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: E = GoppaEncoder(C)
sage: word = vector(F, (0, 1, 0, 0, b, b + 1, b, 0, 0, b, 0, b, b +
    1, 1, b + 1))
sage: x = E.encode(word)
sage: x
> (0, 1, 0, 0, z2, z2 + 1, z2, 0, 0, z2, 0, z2, z2 + 1, 1, z2 + 1, z2
     + 1, 0, z2 + 1, 0, z2, z2, 1, 0, z2, z2, z2 + 1, z2, 0, z2 + 1,
    1)
```

A.3 DECODIFICADOR PARA CÓDIGOS DE GOPPA

En esta sección presentaremos la implementación del decodificador para los códigos de Goppa. Este decodificador usa el algoritmo de Sugiyama descrito en 13.

`class GoppaDecoder(self, code)`

Hereda de: `Decoder`

Representación de un decodificador para un código de Goppa usando el algoritmo de Sugiyama.

ARGUMENTOS

`code` Código asociado a este decodificador.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: GoppaDecoder(C)
> Decoder for [10, 4] Goppa code
```

`get_syndrome(self, c)`

Calcula el polinomio síndrome asociado a la palabra código `c`.

ARGUMENTOS

`c` Vector del espacio de entrada del código asociado a `self`.

SALIDA

Polinomio síndrome asociado al elemento `c`.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
```

```
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: G = C.get_generator_matrix()
sage: E = GoppaEncoder(C)
sage: D = GoppaDecoder(C)
sage: word = random_word(G.nrows(), F)
sage: x = E.encode(word); x
> (1, 0, z2, z2, z2, 0, 0, z2, 1, 0)
sage: num_errors = floor(g.degree()/2)
sage: e = random_error(len(x), num_errors, F)
sage: y = x + e
sage: D.get_syndrome(y)
> (z4^3 + 1)*x^2 + z4^3*x + z4^3
```

get_generating_pol(self)

Calcula el polinomio generador del código asociado a `self`.

SALIDA

Polinomio generador del código asociado a `self`.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: D = GoppaDecoder(C)
sage: D.get_generating_pol()
> x^3 + z4*x^2 + 1
```

decode_to_code(self, word)

Corrige los errores de `word` y devuelve una palabra código del código asociado a `self`.

ARGUMENTOS

`word` Vector del espacio de entrada del código asociado a `self`.

SALIDA

Vector asociado a una palabra código del código asociado a `self`.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: G = C.get_generator_matrix()
sage: E = GoppaEncoder(C)
sage: D = GoppaDecoder(C)
sage: word = random_word(G.nrows(), F)
sage: x = E.encode(word); x
> (1, 0, z2 + 1, 0, 1, 0, 0, z2 + 1, 1, z2)
sage: num_errors = floor(g.degree()/2)
sage: e = random_error(len(x), num_errors, F)
sage: y = x + e
sage: D.decode_to_code(y)
> (1, 0, z2 + 1, 0, 1, 0, 0, z2 + 1, 1, z2)
```

decode_to_message(self, word)

Decodifica `word` al espacio de mensajes del código asociado a `self`.

ARGUMENTOS

`word` Vector del espacio de entrada del código asociado a `self`.

SALIDA

Vector asociado a un mensaje del espacio del código asociado a `self`.

EJEMPLOS

```
sage: F = GF(2^2)
sage: L = GF(2^4)
sage: a = L.gen()
sage: b = F.gen()
sage: R.<x> = L[]
sage: g = x^3 + a*x^2 + 1
sage: n = 10
sage: defining_set = get_defining_set(n, g, L)
sage: C = Goppa(defining_set, g, F)
sage: G = C.get_generator_matrix()
sage: E = GoppaEncoder(C)
sage: D = GoppaDecoder(C)
sage: word = random_word(G.nrows(), F); word
> (z2 + 1, 0, 0, 0)
sage: x = E.encode(word); x
> (z2 + 1, 0, 0, 0, z2, 1, z2, z2 + 1, z2, z2 + 1)
sage: num_errors = floor(g.degree()/2)
sage: e = random_error(len(x), num_errors, F)
sage: y = x + e
sage: D.decode_to_message(y)
> (z2 + 1, 0, 0, 0)
```

A.4 FUNCIONES AUXILIARES

En esta sección presentaremos las funciones auxiliares que han sido de ayuda para poder desarrollar las clases anteriores.

`get_defining_set(n, pol, field)`

Obtiene un conjunto de definición a partir del polinomio `pol` con longitud `n` y pertenece al cuerpo finito `field`.

ARGUMENTOS

`n` Tamaño del conjunto de definición.

`pol` Polinomio generador del conjunto de definición. Sus raíces no pertenecen a dicho conjunto.

`field` Cuerpo finito sobre el que se define el conjunto de definición.

SALIDA

El conjunto de definición asociado al polinomio `pol` con longitud `n` y pertenece al cuerpo finito `field`.

EJEMPLOS

```
sage: L = GF(2^6)
sage: a = L.gen()
sage: R.<x> = L[]
sage: g = x^5 + a*x^2 + 1
sage: n = 5
sage: get_defining_set(n, g, L)
> [z6^5 + z6^3 + z6^2 + z6, z6^4 + z6^3 + 1, z6^5 + z6^4 + z6^3 + z6 + 1,
    z6^5 + z6^2, z6 + 1]
```

`random_word(n, field)`

Obtiene un vector aleatorio de tamaño `n` que pertenece al cuerpo finito `field`.

ARGUMENTOS

`n` Tamaño de la palabra.

`field` Cuerpo finito sobre el que se define el conjunto de definición.

SALIDA

Un vector aleatorio de tamaño `n` que pertenece al cuerpo finito `field`.

EJEMPLOS

```
sage: n = 6
sage: field = GF(3^3)
sage: random_word(n, field)
> (z3^2 + 1, 2*z3^2 + z3 + 2, 2, 2*z3^2 + 2*z3 + 1, 2, 2*z3^2 + 1)
```

`random_error(n, num_errors, field)`

Obtiene un vector aleatorio de tamaño `n` con peso máximo `num_errors` que pertenece al cuerpo finito `field`.

ARGUMENTOS

`n` Tamaño del vector.

`num_errors` Número máximo de errores a añadir.

`field` Cuerpo finito al que pertenece la palabra `x`.

SALIDA

Un vector aleatorio de tamaño `n` con peso máximo `num_errors` que pertenece al cuerpo finito `field`.

EJEMPLOS

```
sage: field = GF(2^6)
sage: n = 10
sage: num_errors = 3
sage: random_error(n, num_errors, field)
> (z6^4 + z6^3 + z6^2 + z6 + 1, 0, 0, z6^3 + z6 + 1, 0, 0, 0, 0, 0, 0)
```

B

IMPLEMENTACIÓN EN SAGEMATH DEL CRIPTOSISTEMA DE MCELIECE

En este anexo describimos la documentación de la clase desarrollada para implementar en SageMath el criptosistema de McEliece. Se ha implementado la clase `McEliece` para representar dicho sistema, esto es, permite generar la clave pública y la clave privada, las cuales se emplearán para cifrar un texto plano y descifrar un criptograma, respectivamente. También se describen las funciones auxiliares desarrolladas.

Para usar esta clase, es necesario cargar el fichero con la orden `load()` tal y como se indica a continuación:

```
sage: load(McEliece.sage)
```

El código desarrollado se encuentra en

https://github.com/paula1999/TFG/tree/main/src.

B.1 CLASE PARA EL CRIPTOSISTEMA MCELIECE

Esta clase simula el comportamiento básico del criptosistema de McEliece, es decir, proporciona métodos para calcular dicho criptosistema definido por un tamaño, la dimensión del cuerpo finito y un polinomio sobre dicho cuerpo finito. Además, proporciona métodos para encriptar mensajes y descifrar criptogramas.

`class McEliece(self, n, q, g)`

Representación del criptosistema de McEliece.

ARGUMENTOS

`n` Tamaño del conjunto de definición.

`q` Dimensión del cuerpo finito $GF(2^q)$.

`g` Polinomio mónico con coeficientes en un cuerpo finito $GF(2^q)$.

EJEMPLOS

```
sage: L = GF(2^5)
sage: a = L.gen()
sage: R.<x> = L[]
sage: g = x^3 + (a^4 + a^3 + 1)*x^2 + (a^4 + 1)*x + a^4 + a^2 + a + 1
sage: n = 20
sage: McEliece(n, q, g)
> McEliece cryptosystem over [20, 5] Goppa code
```

get_S(self)

Obtiene la matriz no singular binaria aleatoria que es parte de la clave privada.

SALIDA

Matriz no singular binaria aleatoria que es parte de la clave privada.

EJEMPLOS

```
sage: L = GF(2^5)
sage: a = L.gen()
sage: R.<x> = L[]
sage: g = x^3 + (a^4 + a^3 + 1)*x^2 + (a^4 + 1)*x + a^4 + a^2 + a + 1
sage: n = 20
sage: ME = McEliece(n, q, g)
sage: ME.get_S()
> [1 1 1 1 1]
  [0 1 1 1 0]
  [1 0 1 0 1]
  [0 1 0 0 0]
  [0 1 1 1 1]
```

get_G(self)

Obtiene la matriz generadora asociada al código de Goppa que es parte de la clave privada.

SALIDA

Matriz generadora asociada al código de Goppa que es parte de la clave privada.

EJEMPLOS

```
sage: L = GF(2^5)
sage: a = L.gen()
sage: R.<x> = L[]
sage: g = x^3 + (a^4 + a^3 + 1)*x^2 + (a^4 + 1)*x + a^4 + a^2 + a + 1
sage: n = 20
sage: ME = McEliece(n, q, g)
sage: ME.get_G()
> [1 0 0 0 1 0 0 1 1 1 0 0 1 0 0 1 0 0 0 0]
  [0 1 0 0 0 0 0 1 1 1 1 0 0 1 0 1 1 0 1 1]
  [0 0 1 0 0 0 0 1 0 0 1 0 1 1 1 0 0 1 1 0]
  [0 0 0 1 1 1 0 0 0 0 0 0 1 1 0 0 0 1 0 1]
  [0 0 0 0 0 0 1 1 0 1 0 1 1 0 0 1 0 1 1 0]
```

get_P(self)

Obtiene la matriz de permutaciones aleatoria que es parte de la clave privada.

SALIDA

Matriz de permutaciones aleatoria que es parte de la clave privada.

EJEMPLOS

```
sage: L = GF(2^5)
sage: a = L.gen()
sage: R.<x> = L[]
sage: g = x^3 + (a^4 + a^3 + 1)*x^2 + (a^4 + 1)*x + a^4 + a^2 + a + 1
sage: n = 20
sage: ME = McEliece(n, q, g)
sage: ME.get_P()
> [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 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 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
  [0 1 0 0 0 0 0 0 0 0 0 0 0 0 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 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 0 0 0 0 0 0 0 0 0 0 0 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 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
  [0 0 0 0 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 1 0 0 0 0 0 0 0 0 0 0]
```

```
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 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 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 1 0 0 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 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
```

get_public_key(self)

Devuelve la matriz que es parte de la clave pública.

SALIDA

Clave pública.

EJEMPLOS

```
sage: L = GF(2^5)
sage: a = L.gen()
sage: R.<x> = L[]
sage: g = x^3 + (a^4 + a^3 + 1)*x^2 + (a^4 + 1)*x + a^4 + a^2 + a + 1
sage: n = 20
sage: McEliece(n, q, g)
sage: ME.get_public_key()
> [0 0 1 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 1]
  [1 1 0 1 1 1 1 1 1 1 1 0 1 1 0 0 1 1 1 1 1]
  [0 1 1 1 0 1 1 0 1 0 0 0 0 0 1 1 1 0 0 0]
  [0 0 0 0 1 0 1 1 1 0 0 1 1 0 0 0 1 1 1 0]
  [1 0 0 1 0 1 0 1 1 0 0 1 1 0 0 1 0 0 1 0]
```

encrypt(self, m)

Devuelve el criptograma asociado al texto plano `m`.

ARGUMENTOS

`m` Texto plano que se va a encriptar.

SALIDA

Mensaje `m` cifrado.

EJEMPLOS

```
sage: L = GF(2^5)
sage: a = L.gen()
sage: R.<x> = L[]
sage: g = x^3 + (a^4 + a^3 + 1)*x^2 + (a^4 + 1)*x + a^4 + a^2 + a + 1
sage: n = 20
sage: McEliece(n, q, g)
sage: message = vector(GF(2), (1, 1, 0, 0, 0))
sage: ME.encrypt(message)
> (1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0)
```

decrypt(self, c)

Devuelve el texto plano asociado al criptograma c.

ARGUMENTOS

c Criptograma que se va a descifrar.

SALIDA

Criptograma c descifrado.

EJEMPLOS

```
sage: L = GF(2^5)
sage: a = L.gen()
sage: R.<x> = L[]
sage: g = x^3 + (a^4 + a^3 + 1)*x^2 + (a^4 + 1)*x + a^4 + a^2 + a + 1
sage: n = 20
sage: McEliece(n, q, g)
sage: encrypted_message = vector(GF(2), (1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
    0, 1, 1, 1, 0, 1, 1, 0, 1, 0))
sage: ME.decrypt(encrypted_message)
> (1, 1, 0, 0, 0)
```

B.2 FUNCIONES AUXILIARES

En esta sección presentaremos las funciones auxiliares que han sido de ayuda para poder desarrollar la clase anterior.

`get_weight(c)`

Calcula el peso del vector c.

ARGUMENTOS

c Vector al que se le va a calcular su peso.

SALIDA

Peso del vector c.

EJEMPLOS

```
sage: v = vector(GF(2), (1, 0, 1, 1, 0))
sage: get_weight(v)
> 3
```

C

IMPLEMENTACIÓN EN SAGEMATH DE LOS ALGORITMOS GENÉTICOS

En este anexo describimos la documentación de las funciones desarrolladas para implementar en SageMath los algoritmos genéticos GGA y CHC. También se describen las funciones auxiliares desarrolladas.

Para usar las funciones que acabamos de describir, es necesario cargar el fichero con la orden `load()` tal y como se indica a continuación:

```
sage: load(Genetic_algorithms.sage)
```

El código desarrollado se encuentra en

https://github.com/paula1999/TFG/tree/main/src.

C.1 ALGORITMO GENÉTICO GGA

En esta sección presentaremos las función desarrollada para poder implementar el algoritmo genético GGA.

`GGA(G, N, pc, max_reinit, min_fitness)`

Obtiene una permutación de la matriz `G` cuya matriz escalonada tiene una fila con peso `min_fitness`.

ARGUMENTOS

`G` Matriz conformada por la clave pública de un criptosistema de McEliece y una fila con el mensaje encriptado.

`N` Tamaño de la población.

`pc` Probabilidad de cruce.

`max_reinit` Número máximo de evaluaciones de las soluciones que no producen mejora en el fitness de la mejor solución encontrada.

`min_fitness` Peso del vector error.

SALIDA

Tupla formada por una permutación el mínimo fitness que produce.

EJEMPLOS

```
sage: q = 4
sage: n = 14
sage: t = 2
sage: F = GF(2)
sage: L = GF(2^q)
sage: a = L.gen()
sage: R.<x> = L[]
sage: g = x^2 + (a^3 + a^2 + a)*x + a^2 + a + 1
sage: min_fitness = floor(t/2)
sage: ME = McEliece(n, q, g)
sage: G = ME.get_public_key()
sage: message = vector(F, (0, 0, 1, 1, 1, 0))
sage: encrypted_message = ME.encrypt(message)
sage: G = G.stack(encrypted_message)
sage: GGA(G, 400, 0.7, 100000, min_fitness)
> ((1, 11, 9, 14, 10, 3, 8, 6, 12, 7, 4, 5, 2, 13), 1)
```

C.2 ALGORITMO GENÉTICO CHC

En esta sección presentaremos las función desarrollada para poder implementar el algoritmo genético CHC.

`CHC(G, N, tau, min_fitness)`

Obtiene una permutación de la matriz `G` cuya matriz escalonada tiene una fila con peso `min_fitness`.

ARGUMENTOS

`G` Matriz conformada por la clave pública de un criptosistema de McEliece y una fila con el mensaje encriptado.

`N` Tamaño de la población.

`tau` Tasa de actualización. $\tau \in [0, 1]$.

`min_fitness` Peso del vector error.

SALIDA

Tupla formada por una permutación el mínimo fitness que produce.

EJEMPLOS

```
sage: q = 4
sage: n = 14
sage: t = 2
sage: F = GF(2)
sage: L = GF(2^q)
sage: a = L.gen()
sage: R.<x> = L[]
sage: g = x^2 + (a^3 + a^2 + a)*x + a^2 + a + 1
sage: min_fitness = floor(t/2)
sage: ME = McEliece(n, q, g)
sage: G = ME.get_public_key()
sage: message = vector(F, (0, 0, 1, 1, 1, 0))
sage: encrypted_message = ME.encrypt(message)
sage: G = G.stack(encrypted_message)
sage: CHC(G, 400, 0.8, min_fitness)
> ((9, 13, 7, 10, 11, 2, 8, 6, 14, 12, 5, 3, 4, 1), 1)
```

C.3 FUNCIONES AUXILIARES

En esta sección presentaremos las funciones auxiliares que han sido de ayuda para poder desarrollar las funciones anteriores.

`fitness_vector(v)`

Calcula el fitness de un vector `v`.

ARGUMENTOS

`v` Vector binario al que se le va a calcular el fitness.

SALIDA

Fitness del vector binario `v`.

EJEMPLOS

```
sage: fitness_vector([0, 1, 1, 0, 0, 0, 1])
> 3
```

`fitness_matrix(M)`

Calcula el fitness mínimo de las filas de la matriz `M`.

ARGUMENTOS

`M` Matriz binaria de la que queremos calcular su fitness mínimo.

SALIDA

Fitness mínimo de las filas de la matriz `M`.

EJEMPLOS

```
sage: M = matrix(GF(2), [[0, 1], [1, 1]])
sage: fitness_matrix(M)
> 1
```

`permutation_matrix(x)`

Calcula la matriz asociada a la permutación `x`.

ARGUMENTOS

`x` Permutación que se le aplica a la matriz `M`.

SALIDA

La matriz de permutaciones asociada a `x`.

EJEMPLOS

```
sage: x = Permutation([3, 2, 1])
sage: permutation_matrix(x)
> [0 0 1]
  [0 1 0]
  [1 0 0]
```

`fitness_permutation(M, x)`

Calcula el fitness mínimo de las filas de la forma escalonada reducida de la matriz `M` permutada por la permutación `x`.

ARGUMENTOS

`M` Matriz binaria de la que queremos calcular su fitness mínimo.

`x` Permutación que se le aplica a la matriz `M`.

SALIDA

Fitness mínimo de las filas de la forma escalonada reducida de la matriz `M` permutada por la permutación `x`.

EJEMPLOS

```
sage: M = matrix(GF(2), [[1, 0, 0], [0, 1, 1], [1, 0, 1]])
sage: x = Permutation([3,2,1])
sage: fitness_permutation(M, x)
> 1
```

`random_sol(n)`

Calcula una solución aleatoria de tamaño `n`.

ARGUMENTOS

`n` Tamaño de la solución.

SALIDA

Un vector aleatorio de tamaño `n`.

EJEMPLOS

```
sage: random_sol(5)
> [1, 4, 5, 2, 3]
```

`crossover(p1, p2)`

Operador cruce entre dos padres `p1` y `p2`.

ARGUMENTOS

`p1` Vector de dimensión `n`.

`p2` Vector de dimensión `n`.

SALIDA

Un vector aleatorio de tamaño `n`.

EJEMPLOS

```
sage: crossover([1, 3, 2], [3, 1, 2])
```

```
> ([3, 2, 1], [2, 1, 3])
```

`mutation(p)`

Operador mutación de `p`.

ARGUMENTOS

`p` Individuo que se va a mutar.

SALIDA

Un vector mutado.

EJEMPLOS

```
sage: crossover([1, 3, 2], [3, 1, 2])
> ([3, 2, 1], [2, 1, 3])
```

`distance(x, y)`

Calcula la distancia de Hamming entre `x` e `y`.

ARGUMENTOS

`x` Vector de tamaño `n`.

`y` Vector de tamaño `n`.

SALIDA

Distancia Hamming entre los vectores `x` e `y`.

EJEMPLOS

```
sage: distance([1, 3, 2], [3, 1, 2])
> 2
```

`actualizar(P, tau)`

Calcula el decremento del cruce como un porcentaje `tau` de la máxima distancia de Hamming entre los individuos de la población `P`.

ARGUMENTOS

`P` Población.

`tau` Tasa de actualización. $\tau \in [0, 1]$.

SALIDA

Tupla formada por la distancia y el decremento del cruce.

EJEMPLOS

```
sage: P = matrix([[1, 4, 2, 3], [2, 1, 3, 4]])
sage: tau = 0.8
sage: actualizar(P, tau)
> (1.33333333333333, 3.20000000000000)
```

BIBLIOGRAFÍA

[1] Berlekamp, E., McEliece, R., & Tilborg, H. V. (1978). On the inherent intractability of certain coding problems. *IEEE Transactions on Information Theory*, *24*(3), 384–386.

[2] Berlekamp, E. R. (1973). Goppa codes. *IEEE Transactions on Information Theory*, *19*(5).

[3] Bernstein, D. J., Buchmann, J., & Dahmen, E. (2009). *Post-quantum cryptography*. Springer. ISBN: 978-3-540-88701-0.

[4] Bernstein, D. J., Buchmann, J., & Dahmen, E. (2015). *Introduction to Cryptography*. Springer Berlin Heidelberg. ISBN: 978-3-662-47973-5.

[5] Betten, A., Braun, M., Fripertinger, H., Kerber, A., Kohnert, A., & Wassermann, A. (2006). *Error-Correcting Linear Codes*. Springer. ISBN: 978-3-540-28371-5. Accedido el 2022-03-07. URL http://www.cambridge.org/9780521782807

[6] Boyd, C. (Ed.) (2001). *Advances in Cryptology — ASIACRYPT 2001*. Springer. ISBN: 978-3-540-42987-6.

[7] Courtois, N., Finiasz, M., & Sendrier, N. (2001). *How to Achieve a McEliece-Based Digital Signature Scheme*, (pp. 157–174). Vol. 2248 of [6]. ISBN: 978-3-540-42987-6.

[8] Cuéllar, M. P., Gómez-Torrecillas, J., Lobillo, F. J., & Navarro, G. (2021). Genetic algorithms with permutation-based representation for computing the distance of linear codes. *Swarm and Evolutionary Computation*, *60*.

[9] Diffie, W., & Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, *IT-22*(6), 644–654.

[10] Eshelman, L. J. (1991). *The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination*. Elsevier. ISBN: 978-0-08-050684-5.

[11] Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System Technical Journal*, *29*(2), 147–160. Accedido el 2022-05-14.
URL https://ieeexplore.ieee.org/document/6772729

[12] Huffman, W. C., & Pless, V. (2010). *Fundamentals of Error-Correcting Codes*. Cambridge University Press. ISBN: 978-0-521-13170-4. Accedido el 2022-04-25.
URL http://www.cambridge.org/9780521782807

[13] Li, Y., Deng, R. H., & Wang, X. (1994). On the equivalence of mceliece's and niederreiter's public-key cryptosystems. *IEEE Transactions on Information Theory*, *40*(1).

[14] McEliece, R. J. (1978). A public-key cryptosystem based on algebraic coding theory. *Deep Space Network Progress Report*, (pp. 114–116). Accedido el 2022-04-15.
URL https://tmo.jpl.nasa.gov/progress_report2/42-44/44N.PDF

[15] Niederreiter, H. (1986). Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, *15*(2), 159–166.

[16] Podestá, & Ricardo (2006). Introducción a la teoría de códigos autocorrectores. *Universidad Nacional de Córdoba*. Accedido el 2022-05-15.
URL https://www.famaf.unc.edu.ar/documents/940/CMat35-3.pdf

[17] Shannon, C. (1948). *A mathematical theory of communication*. University of Illinois Press. ISBN: 978-0-252-72546-3.

[18] Shor, P. W. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, *26*(5), 1484–1509.

[19] Siim, S., & Skachek, V. (2015). Study of mceliece cryptosystem. *Deep Space Network Progress Report*.

[20] Sugiyama, Y., Kasahara, M., Hirashawa, Y., & Namekawa, T. (1975). A method for solving key equation for decoding goppa codes. *Information and Control*, *27*(1), 87–99.

[21] Tsfasman, M. A., Vladut, S. G., & Zink, T. (1982). Modular curves, shimura curves, and goppa codes, better than varshamov-gilbert bound. *Mathematische Nachrichten*, *109*(1), 21–22.

[22] Vardy, A. (1997). The intractability of computing the minimun distance of a code. *IEEE Transactions on Information Theory*, *43*(6).