Universidad Zaragoza
1542

Trabajo Fin de Grado

ZPERF: una familia de hash perfecto eficiente y de tamaño casi mínimo

ZPERF: a perfect hash family that is efficient and has nearly minimal size

Autor

Isaac Velasco Calvo

Directores

Elvira Mayordomo Cámara

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2022

Índice

1. Introducción

En este proyecto, vamos a dedicarnos a la implementación [7] de una nueva y moderna forma de generar **Familias de Hashes Perfectos** en forma de **Códigos Lineales**, como se describe en un artículo [1] escrito por los matemáticos Chaoping Xing y Chen Yuan, que poseen un *rate* menor con respecto a otros métodos parecidos. También se ha procesado el tiempo de cómputo de estos códigos al igual que su *rate* de forma experimental, elementos que el artículo original no atacaba.

Una **Función Hash** es una función que permite transformar un conjunto de **claves** a un valor, generalmente más pequeño, llamado su **hash**. Se dice que hay una colisión cuando dos claves distintas tienen el mismo hash. Esta función es **perfecta** si, para un conjunto determinado de claves, no existen colisiones. Las **Familias de Hashes Perfectos** son colecciones de n funciones hashes perfectas $f_1, ..., f_n$ para conjuntos S de $|S| \leq q$ claves. Sus usos incluyen la creación de tablas hash las cuales permiten acceso rápido a datos en tablas, listas de variables o palabras reservadas en lenguajes de programación, organización de paquetes en routers [6], etc.

La implementación ha sido complicada por una falta de familiaridad con la notación matemática usada en el artículo. También se ha encontrado con dificultades ya que el artículo mezcla propiedades y algoritmos usados para la construcción de códigos lineares, lo que se usa en este proyecto, con demostraciones matemáticas de características de estos códigos lineares.

2. Conceptos básicos

2.1. Funciones Hash

Una **Función Hash** es una función o algoritmo que permite transformar datos de entrada, de tipos específicos, a un valor numérico con un tamaño fijo y dentro de un rango conocido. A estos datos de entrada se les llama generalmente **keys** o **claves** en español, mientras que al resultado de la función normalmente se le llama **hash value** o simplemente **hash**.

Un ejemplo popular de función hash básica es aquella que obtiene un **hash** a partir de una palabra sumando la posición en el alfabeto de cada una de las letras que forman la palabra, mientras que otras funciones hash, como SHA-1, utilizan métodos más complejos.

Como algunas estas funciones puede tener un rango limitado de soluciones, existe la posibilidad de dos o más **claves** distintas tengan el mismo **hash** al aplicar la función. A esto se le llama una **colisión**. Continuando con el ejemplo anterior, las palabras `Sol` y `Cesar` ambos tienen el **hash** `46`.

2.2. Funciones Hash Perfecta

Una **Función Hash Perfecta** f es un tipo de función hash que, para un conjunto S determinado de claves, no existe ninguna colisión, es decir, si $s_1 \neq s_2$ para $s_1, s_2 \in S$, entonces $f(s_1) \neq f(s_2)$.

Este tipo de función hash es muy útil para **Tablas Hash**, una estructura de datos donde cada elemento se guarda con una clave y su acceso se realiza con una función hash, dando una velocidad de acceso a estos datos muy alta. Utilizando una función hash perfecta permite no tener que tener en cuenta colisiones, dando un acceso a datos aún más rápido.

2.3. Familia de Hashes Perfectos

Una **Familia de Hashes Perfectos** es una colección de $f_1, ..., f_n$ funciones hash perfectas para un conjunto de $\leq M(n, q)$ claves, con cada una de estas funciones siendo perfecta para un conjunto de claves S tal que $|S| \leq q$, siendo **mínima** si $|S| = q$. Comparado con otras formas de generar funciones hash perfectas, utilizando estas colecciones se pueden obtener un sistema con el que se puede cambiar entre distintas claves dinámicamente con facilidad.

2.3.1. Usos

Las familias de hashes perfectos son muy útiles para situaciones en las que se necesitan funciones hash perfectas cuyas claves pueden ir cambiando en determinadas situaciones. Re-calcular una función hash perfecta para adaptarse a cambios de las claves que se utilizan puede resultar muy costoso, pero una familia de hashes perfectos soluciona este problema dando una colección de funciones para un número de claves específico.

Un uso que le han dado los investigadores Yi Lu, Balaji Prabhakar y Flavio Bonomi a este concepto es la construcción de tablas hash más pequeñas para el enrutamiento de paquetes en routers [6]. Utilizando funciones perfectas mínimas, consiguen crear una tabla hash que reduce el espacio que necesitan usar en memoria comparado con otras opciones, manteniendo la capacidad de utilizar dichas tabla a las velocidades que se ven en comunicaciones, medidas en nanosegundos, al mismo tiempo que pueden modificar dinámicamente su tabla hash.

Un hipotético uso que se le podría dar a una familia de hashes perfectos es en un aeropuerto, donde para una colección de aviones conocidos, q de ellos están en tierra, esperando a que un avión nuevo aterrize para poder despegar. Esto permitiría una organización rápida en tablas hash de los aviones en tierra y la capacidad de modificar esta tabla a medida que el conjunto de aviones en tierra cambie.

2.4. Códigos Lineales

Los **Códigos Lineales** son una parte muy importante en el campo de la teoría de códigos, el estudio de la codificación de la información. Un código lineal esta compuesto de k n-tuplas que se llaman ***Codewords*** [1]. Cada elemento de una de estas tuplas sólo puede tener un valor natural en el rango de $[0, q-1]$, también escrito como $[q]$. Con esta notación, se puede decir que un *codeword* es un elemento del conjunto $[q]^n$.

En el artículo [1], a estos códigos lineales de k *codewords*, cada uno de estos siendo un elemento del conjunto $[q]^k$, también se les llama **q-ary de longitud k y tamaño k** o simplemente q-ary si se asume el resto de características. Con esta información, podemos representar todos los *codewords* de un q-ary con una matriz tamaño $k \times n$, con cada fila representando un *codeword* distinto.

$$C = \begin{pmatrix} 0 & 1 & 2 & 3 & 2 & 3 & 3 \\ 2 & 1 & 3 & 0 & 0 & 1 & 0 \\ 2 & 1 & 3 & 0 & 0 & 2 & 0 \\ 3 & 2 & 2 & 0 & 1 & 1 & 1 \\ 2 & 1 & 3 & 0 & 0 & 1 & 3 \end{pmatrix}$$

Figura 1: Ejemplo de un 4-ary longitud 7 tamaño 5, llamada C, representada como una matriz

2.4.1. Notación

Para este proyecto, la notación se inspira en aquella que se usa en el artículo original [1].

- Generalmente, se utiliza k para describir cuántas *codewords* hay en un código C (Longitud), n para el número de elementos en cada *codeword* (Tamaño) y q para describir el rango de valores $[0, 1, ..., q-1]$ que pueden tener estos elementos. (En la figura 1, el código mostrado tendría $q = 4, n = 7, k = 5$).
- Se denota $c \in C$ un *codeword* c pertenece al código lineal C.
- El elemento número $i \geq 0$ de un *codeword* c se denota como $c[i]$. (Por ejemplo, para el *codeword* $c = (1, 1, 2, 3, 0, 3, 3)$, $c[4] = 0$)

- El íncide de un *codeword* es la fila en la que aparece en la representación de su código lineal C en forma de matriz. (Por ejemplo, en el código C de la figura 1, el *codeword* $(3, 2, 2, 0, 1, 1, 1)$ tiene el íncide 3, ya que la primera fila se considera el índice 0).

- Un subconjunto de un código linear puede denotarse como un vector de valores booleanos. Si el elemento i de un subconjunto I es verdadero/1, el *codeword* con el mismo índice i esta en el subconjunto. (Por ejemplo, para $I = [\text{Verdadero}, \text{Falso}, \text{Verdadero}, \text{Verdadero}]$, I_0, al ser verdadero, indica que el *codeword* con índice 0 de su código lineal esta en ese subconjunto)

2.4.2. m-hash q-ary

Un **m-hash q-ary** es un tipo de q-ary donde todos los conjuntos de m *codewords* del código original están separados. Un conjunto de *codewords* están separados si existe un valor $i \in [0, 1, ..., n-1]$ tal que ningún *codeword* c del conjunto tienen el mismo valor $c[i]$.

$$\text{Subconjunto}_{(0,1,2)} \text{ de } C = \begin{pmatrix} 0 & 1 & 2 & 3 & 2 & 3 & 3 \\ 2 & 1 & 3 & 0 & 0 & 1 & 0 \\ 2 & 1 & 3 & 0 & 0 & 2 & 0 \end{pmatrix}$$

Figura 2: Ejemplo de tres *codewords* separadas, ya que la 6ª columna poseen valores distintos

$$\text{Subconjunto}_{(0,1,4)} \text{ de } C = \begin{pmatrix} 0 & 1 & 2 & 3 & 2 & 3 & 3 \\ 2 & 1 & 3 & 0 & 0 & 1 & 0 \\ 2 & 1 & 3 & 0 & 0 & 1 & 3 \end{pmatrix}$$

Figura 3: Ejemplo de tres *codewords* **no** separadas, ya que ninguna columna tiene valores sin repetirse

Ampliando en esta definición, un **q-hash** es un m-hash q-ary donde $m = q$, es decir, un q-hash q-ary.

Por ejemplo, el 4-ary en la figura 1 sería un 2-hash 4-ary, ya que ninguna *codeword* coincide en valor, pero no sería un 3-hash 4-ary, como se demuestra en la figura 3, y aún menos un 4-hash.

2.4.3. *Rate*

El ***Rate*** de un código lineal indica como de redundante es la información que contiene un Código Lineal. Específicamente, el valor es igual a la cantidad de información no redundante en un bit [5]. Para este proyecto, el *Rate* R_C de un Código Linear C se calcula de la siguiente manera, considerando $|C|$ el número de *codewords* en ese código:

$$R_C = \frac{\log_2 |C|}{n}$$

El estudio de los *Rates* de códigos lineales se remonta a la década de 1980, donde Michael L. Fredman y János Komolós [4] se sitúan como los pilares de este campo.

2.4.4. Características Adicionales

La propiedad de códigos lineales que no se usa en este proyecto que más usaron los autores del artículo [1] es la distancia entre dos *codewords* y la distancia dual de un código C. La **distancia** entre dos *codewords* es el número de elementos de uno de ellos que hay que cambiar para que acabe siendo igual al otro (Por ejemplo, la distancia entre los *codewords* $(1, 3, 4)$ y $(1, 2, 4)$ es 1, ya que solo se diferencian en el segundo valor).

El **código dual** de un código linear C, denotado como $C^{\perp}$, es un código lineal con el mismo número de *codewords* definido como $C^{\perp} = \{c \in [q]^n | c \cdot d = 0,\ d \in C\}$. Esto significa que para cada *codeword* d en C, $C^{\perp}$ tiene un *codeword* c tal que el producto escalar entre c y d es 0. Con esto se puede obtener la **distancia dual** del código C, que es la distancia mínima entre todos los pares de *codewords* de $C^{\perp}$.

El uso de estas propiedades son usadas en el artículo original [1] para dar a conocer el tamaño de la matriz $\mathcal{A}$ (Ver sección 3.2) sin tener que calcularla. Esto resulta redundante, ya que esta matriz se debe calcular obligatoriamente, quitándole un posible uso a estas propiedades adicionales.

2.4.5. Usos

El uso clásico de los códigos lineales es como un código de corrección de errores [5], un sistema que permite detectar y/o corregir cualquier error que se ha podido generar en la transmisión de un mensaje.

A nosotros, en cambio, nos interesa su capacidad de describir una familia de hashes perfectos [2]. Este es el objetivo del artículo original [1] que vamos a implementar.

3. Implementación

La implementación [7] se ha realizado en `C++` con las herramientas de este lenguaje en `MingW`. Se puede ver el código fuente en https://github.com/ivc31415/TFG.

La parte central de este algoritmo se encuentra el la concatenación de dos códigos lineares, llamados **Código Exterior** (C_1) y **Código Interior** (C_2), para formar un nuevo código linear C q-hash perfecto. Este nuevo código C sera la salida final del algoritmo, el cual posee las características que buscamos. Para poder calcular el código exterior C_1, se requiere la matriz $\mathcal{A}$, que se calcula a partir del código interior C_2.

Al contrario que algoritmos similares desarrollados por investigadores como Körner y Matron [3], la concatenación que proponen Chaoping y Chen permite que el código interior C_2 no sea un código q-hash perfecto necesariamente, lo que permite una construcción más simple, y un *rate* menor.

Es importante conocer las características que estos dos códigos lineares tienen:

- **Código Exterior** C_1
  - m-ary longitud n_1 tamaño k_1.
  - Cada *codeword* es una n_1-tupla, con valores en el rango $[0, ..., m-1]$.
  - Posee k_1 *codewords*
- **Código Interior** C_2
  - q-hash (q-ary) longitud n_2 tamaño k_2.
  - Cada *codeword* es una n_2-tupla, con valores en el rango $[0, ..., q-1]$.
  - Posee k_2 *codewords*

3.1. Códigos Interiores C_2

Al contrario que el código exterior C_1, existen diversos códigos interiores que podemos usar, cada uno de ellos especializado para un valor distinto de q. La construcción de estos códigos son muy similares entre ellas, y con procesos mucho más sencillos cuando lo comparamos con el código exterior C_1. Hemos utilizado los siguientes códigos:

3.1.1. C_2 para $q = 2^r$

El primer código interior implementado fue el descrito en el Teorema 3.14, Caso 1 del artículo original. Esta colección de códigos poseen $q = 2^r$, tal que $r \geq 2$. El código es un q-ary que posee $q^2 = m$ *codewords*, cada uno siendo 4-tuplas. Su construcción es sencilla:

$$C_1 = \{(x, y, x + y, x + \alpha y) : x, y \in [0, ..., q - 1]\}$$

Figura 4: Construcción del código interior C_2 con $q = 2^r, r \geq 2$

Esta formula indica que, tomando $\alpha \in [2, 3, ..., q-1]$ a elegir, para todas las combinaciones posibles de valores de x e y, existe un *codeword* $c_{x,y}$ igual $(x, y, x + y, x + \alpha y)$.

$$\alpha = 2 \mid (x, y) = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 2 \\ 0 & 3 \\ 1 & 0 \\ 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 2 & 0 \\ 2 & 1 \\ \vdots & \vdots \end{pmatrix} \rightarrow C_2 = \begin{pmatrix} 0 & 0 & 0+0 & 0+\alpha 0 \\ 0 & 1 & 0+1 & 0+\alpha 1 \\ 0 & 2 & 0+2 & 0+\alpha 2 \\ 0 & 3 & 0+3 & 0+\alpha 3 \\ 1 & 0 & 1+0 & 1+\alpha 0 \\ 1 & 1 & 1+1 & 1+\alpha 1 \\ 1 & 2 & 1+2 & 1+\alpha 2 \\ 1 & 3 & 1+3 & 1+\alpha 3 \\ 2 & 0 & 2+0 & 2+\alpha 0 \\ 2 & 1 & 2+1 & 2+\alpha 1 \\ & \vdots & \vdots & \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 0 & 2 & 2 & 0 \\ 0 & 3 & 3 & 2 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 \\ 1 & 3 & 0 & 3 \\ 2 & 0 & 2 & 2 \\ 2 & 1 & 3 & 0 \\ & \vdots & \vdots & \end{pmatrix}$$

Figura 5: Ejemplo de parte del código interior C_2 con $q = 2^r, r = 2$

La implementación es igualmente sencilla, con el uso de dos bucles, uno para x y otro para y, con cada iteración generando el *codeword* con índice $qx + y$:

Algorithm 1 q-ary GenerarC2_Caso1(entero r, entero $\alpha \geq 2$)

1: $q \leftarrow 2^r$
2: $C_2 \leftarrow q$-ary tamaño q^2 longitud 4
3: **for all** $x \in [0, ..., q - 1]$ **do**
4: **for all** $y \in [0, ..., q - 1]$ **do**
5: C_2, *codeword* con índice $(qx + y) \leftarrow (x, y, x + y, x + \alpha y)$
6: **end for**
7: **end for**
8: **return** C_2

3.1.2. C_2 para $(q \geq 3, \mathbf{mod}(q,2) = 1)$

Este código interior es la implementación del código descrito en el Lema 3.12 del artículo original [1]. Es el más sencillo de todos los que vamos a utilizar y da una gran gama de posibles valores de q. Para construirlo, al igual que en la sección 3.1.1, se usan todos los valores posibles de $x, y \in [0, ..., q-1]$, dando un código, también, de tamaño q^2 y longitud 4. Al contrario que el resto de formas de generar el código interior, este no requiere de una constante α. El código es el siguiente:

$$C_2 = \{(x, y, x+y, x-y) : x, y \in [0, ..., q-1]\}$$

Figura 6: Construcción del código interior C_2 con $(q \geq 3, \mathrm{mod}(q,2) = 1)$

$$(x,y) = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 2 \\ 1 & 0 \\ 1 & 1 \\ 1 & 2 \\ 2 & 0 \\ 2 & 1 \\ 2 & 2 \end{pmatrix} \rightarrow C_2 = \begin{pmatrix} 0 & 0 & 0+0 & 0-0 \\ 0 & 1 & 0+1 & 0-1 \\ 0 & 2 & 0+2 & 0-2 \\ 1 & 0 & 1+0 & 1-0 \\ 1 & 1 & 1+1 & 1-1 \\ 1 & 2 & 1+2 & 1-2 \\ 2 & 0 & 2+0 & 2-0 \\ 2 & 1 & 2+1 & 2-1 \\ 2 & 2 & 2+2 & 2-2 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 0 & 2 & 2 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 2 & 0 \\ 1 & 2 & 0 & 2 \\ 2 & 0 & 2 & 2 \\ 2 & 1 & 0 & 1 \\ 2 & 2 & 1 & 0 \end{pmatrix}$$

Figura 7: Ejemplo del código interior C_2 con $q = 3$

La implementación es igualmente sencilla, siguiendo los mismos pasos que en la sección 3.1.1:

Algorithm 2 q-ary GenerarC2_Caso2(entero q)

```
C_2 ← q-ary tamaño q^2 longitud 4
for all x ∈ [0, ..., q − 1] do
    for all y ∈ [0, ..., q − 1] do
        C_2, codeword con índice (qx + y) ← (x, y, x + y, x − y)
    end for
end for
return C_2
```

3.2. Matriz $\mathcal{A}$

$\mathcal{A}$ es una matriz que se calcula a partir del código interior C_2. Se usa para calcular el código exterior C_1, ya que este depende de esta matriz.

3.2.1. Colección $\mathcal{S}(C_2)$

El primer paso es calcular $\mathcal{S}(C_2)$. Esto denota la colección de todos los subconjuntos de q *codewords*, del código linear q-ary C_2, que estén separados. Recordemos que un conjunto de *codewords* está separado si existe un índice en el que todos los *codewords* del conjunto son distintos.

Para calcular $\mathcal{S}(C_2)$, empezamos con una lista vacía. Con una función recursiva, podemos iterar a través de todos los subconjuntos de q *codewords* de C_2 y comprobar si están separados. Si lo están, se añaden a la lista.

En vez de guardar el valor de todos los *codewords* en una lista, con lo que sería difícil de trabajar, se guarda una máscara de los índices de qué *codewords* corresponden a cada subconjunto. El índice de un *codeword* es la posición en la que se guarda en memoria dentro de su código lineal (En nuestro

caso, como cada *codeword* se guarda en las filas de una matriz, su índice es en qué fila está). Estas máscaras se guardan en forma de matriz, donde cada fila es una máscara de un subconjunto.

Por ejemplo, para el subconjunto $(1, 1, 1, 0, 0, 0, 0, ...)$ (Primer subconjunto en la figura 8), indica que ese subconjunto contiene los *codewords* en primera, segunda y tercera posición en C_2.

$$\mathcal{S}(C_2) = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Figura 8: Un ejemplo de un hipotético $\mathcal{S}(C_2)$ guardado en forma de matriz, donde cada fila es una máscara de un subconjunto

Para obtener todas las combinaciones de subconjuntos, se ha mencionado que se utiliza una función recursiva, la cual es la siguiente:

Algorithm 3 CalcularSC2(q-ary C_2, booleano $I[k]$, entero $d = 0$, entero $i = 0$, lista de *codewords* $L = \{\}$)

```
1: if i = k (Se han seleccionado q codewords) then
2:    if SubconjuntoSeparado(C_2, I) then
3:       Añadir I a L
4:    end if
5: else if k − d + i ≥ k then
6:    I_d = verdadero
7:    CalcularSC2(C, I, d + 1, i + 1, L)
8:    I_d = falso
9:    CalcularSC2(C, I, d + 1, i, L)
10: end if
```

Este tipo de algoritmo se llama un algoritmo de búsqueda en árbol binario. Esto se debe a que, al calcular $\mathcal{S}(C_2)$, hay que comprobar todas las combinaciones de subconjuntos de C_2, las cuales se pueden describir como un árbol binario, donde los hijos de un nodo son las dos llamadas que se realiza en la función, y las hojas (Donde la función recursiva acaba su búsqueda) son los subconjuntos que se buscan.

Este es un proceso lento, ya que hay que comprobar $\binom{k_2}{q}$ combinaciones, y para cada una de estas, se pueden necesitar hasta $q * n_2$ comparaciones para dictaminar si ese conjunto esta separado o no. En la sección 3.5.1 se puede ver como se ha intentado evitar este problema.

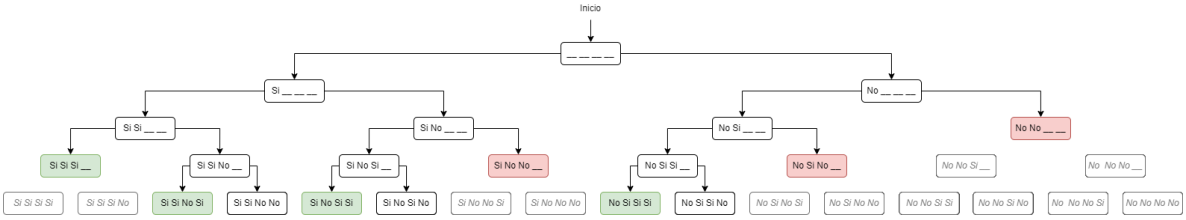


Figura 9: Árbol binario de búsqueda realizado por el algoritmo `CalcularSC2` para subconjuntos de 3 elementos de un grupo de 4 elementos. El interior de cada nodo describe el vector I. Nodos verdes describen un subconjunto del número deseado de elementos. Nodos rojos indican búsquedas no realizadas por optimización.

En este algoritmo, se tiene el vector I, en cual, con tamaño q, indica qué *codewords* se están teniendo en cuenta para el subconjunto (Si I_3 es verdadero, significa que la *codeword* con índice 3 esta en el conjunto). Si la condición de la linea `5` se cumple, se bifurca la búsqueda entre aquellos subconjuntos con el *codeword* de íncide d y aquellos sin él. Si se cumple la condición en la línea `2`, I describe un subconjunto de C_2 de tamaño q, y se comprueba si esta separado. Si lo está, ese subconjunto se añade a L.

Para comprobar que el subconjunto está separado, se utiliza el siguiente algoritmo basado en la descripción de un subconjunto separado del artículo original [1]:

Algorithm 4 SubconjuntoSeparado(q-ary C, booleano $I[q] = \{I_0, ..., I_{q-1}\}$

```
1:  for all i ∈ [0, ..., n_2 − 1] do
2:      v ← {(v_0, ..., v_{q−1}) | v_j = falso}
3:      c ← verdadero
4:      for all j ∈ [0, ..., q − 1] do
5:          w ← Codeword(C, j)
6:          if I_{w_i} = verdadero then
7:              c ← falso
8:              Terminar Bucle j
9:          else
10:             I_{w_i} ← verdadero
11:         end if
12:     end for
13:     if c then
14:         return Verdadero
15:     end if
16: end for
17: return Falso
```

Un subconjunto está separado si existe un índice donde los *codewords* tienen valores distintos. En este algoritmo, para cada índice i, se crea un vector v donde se guarda qué valores se han encontrado en los *codewords*. Si se encuentra un valor que ya se había visto, ese índice se indica que tiene valores repetidos con la variable c y se pasa al siguiente índice. Si se encuentra un índice donde todos los valores son distintos, se devuelve verdadero. Si se ha pasado por todos los índices y todos tenian valores repetidos, se devuelve falso.

Se utiliza el vector v ya que los valores de los *codewords* en cada índice no estas necesariamente ordenados.

3.2.2. Matriz $\mathcal{A}$

El segundo paso es computar la **Matriz** $\mathcal{A}$, tamaño $|\mathcal{A}| \times q$, la cual es definida como en el artículo original.

$$\bigcup_{\{c_1,...,c_q\} \in \mathcal{S}(C_2)} \{\pi(c_1), ..., \pi(c_q)\}$$

$\pi(c)$ refiere a una función biyectiva que transforma un *codeword* de C_2 a $i \in [0, k_2 - 1]$. En nuestro caso, $\pi(c)$ devuelve el índice del *codeword* c en su código lineal C, lo que hace el calculo de $\mathcal{A}$ muy fácil, ya que en la sección 3.2.1, habíamos guardado la lista $\mathcal{S}(C_2)$ como máscaras de índices de *codewords*.

$$\mathcal{S}(C_2) = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \rightarrow \mathcal{A} = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 1 & 2 & 4 \\ 4 & 5 & 6 \end{pmatrix}$$

Figura 10: La lista $\mathcal{S}(C_2)$ de la figura 8 transformada a la matriz $\mathcal{A}$

3.3. Código Exterior C_1

El código exterior, denotado como C_1, es el segundo de los dos códigos lineares que se utilizan en la concatenación, y el único de los dos que no es un m-hash perfecto. Para computar este código es necesario tener la matriz $\mathcal{A}$, la cual en sí depende del código exterior C_2. El calculo de esta matriz se describe en la sección 3.2.

3.3.1. Paso 1: Requerimientos

Para poder construir este código m-ary, como se ha dicho ya, se requiere los valores de la matriz $\mathcal{A}$ (de tamaño $|\mathcal{A}| \times q$) junto a los valores de $k_2 = m$ y q, los cuales dependen de C_2. También se requiere un valor n_1 dado por el usuario y M. M detalla aproximadamente cuantos k_1 *codewords* se tendrán en el código C_1 al final de su construcción, con $k_1 \geq \lceil \frac{M}{3} \rceil$.

No se pueden elegir estos parámetros libremente, si no que requieren que cumplan la siguiente desigualdad:

$$\binom{M}{q} \left(1 - \frac{q!|\mathcal{A}|}{m^q}\right)^{n_1} \leq \frac{M}{2q}$$

Figura 11: Desigualdad requerida para construir el código exterior C_1

Este requerimiento restringe cuántos *codewords* puede tener el código exterior C_1, al igual que cómo de grande tiene que ser n_1. La implementación, si el programador lo desea, puede comprobar si los parámetros de entrada se ajustan a la desigualdad e incluso modificar automáticamente el valor de M para que sea válido. Ver la sección 4.5 para comprobar los efectos de esta desigualdad.

3.3.2. Paso 2: Muestreo

La construcción de este código empieza con el muestreo de M *codewords* uniformemente aleatorios, cada uno de longitud n_1. Para esto, se ha usado la librería estándar `<random>` de `C++`. Con esto, podemos obtener números enteros uniformemente distribuidos en un rango deseado, en nuestro caso $[0, 1, ..., m - 1]$, ya que C_1 es un código m-ary.

Estos *codewords* muestreados se guardan como cualquier otro código linear, en una matriz, que en este caso será de tamaño M, n_1. El número de *codewords* en esta matriz ira decreciendo con los siguientes pasos. La librería también nos permite añadir una *semilla*, la cual podemos usar para muestrear los mismos *codewords* en caso de que queramos repetir una prueba. Se ha implementado de tal manera que si la semilla es igual a 0, la semilla tendrá el valor `rand_dev()`, otra funcionalidad de la librería.

3.3.3. Paso 3: Eliminación de *codewords* no amigables con $\mathcal{A}$

En este paso, tras obtener los *codewords* muestreados uniformemente y aleatoriamente, a los que llamaremos $C_{1,1}$, se eliminan, como máximo $\frac{M}{2}$, aquellas *codewords* que **no sean amigables con** $\mathcal{A}$. Un *codeword* $c \in C_{1,1}$ no es amigable con $\mathcal{A}$ si c pertenece a un subconjunto de q *codewords* de $C_{1,1}$ que es no amigable con $\mathcal{A}$.

Un subconjunto de q *codewords* $(c_1, c_2, ..., c_q)$ es amigable con $\mathcal{A}$ si existe $i \in [0, 1, ..., n_1]$ tal que $(c_1[i], c_2[i], ..., c_q[i]) \in \mathcal{A}$, o dicho de otra forma, si organizamos estos $(c_1, c_2, ..., c_q)$ *codewords* en filas en una tabla, una de las columnas, **tras ordenarla de menor a mayor**, será una fila de $\mathcal{A}$.

$$C_{1,1} = \begin{pmatrix} \mathbf{1} & \mathbf{4} & \mathbf{3} & \mathbf{0} \\ 1 & 0 & 1 & 1 \\ 2 & 3 & 3 & 2 \\ \mathbf{2} & \mathbf{3} & \mathbf{1} & \mathbf{1} \\ 3 & 1 & 1 & 0 \\ \mathbf{4} & \mathbf{0} & \mathbf{0} & \mathbf{2} \\ 2 & 1 & 2 & 2 \end{pmatrix} \rightarrow I_1 = \begin{pmatrix} 1 & 4 & \mathbf{3} & 0 \\ 2 & 3 & \mathbf{1} & 1 \\ 4 & 0 & \mathbf{0} & 2 \end{pmatrix}, \ \mathcal{A} = \begin{pmatrix} 0 & 1 & 2 \\ \mathbf{0} & \mathbf{1} & \mathbf{3} \\ 2 & 3 & 5 \\ 2 & 3 & 4 \\ 4 & 5 & 6 \end{pmatrix}$$

Figura 12: Ejemplo de un subconjunto I_1 de *codewords* de $C_{1,1}$ amigable con $\mathcal{A}$

$$I = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 2 & 3 & 1 & 1 \\ 4 & 0 & 0 & 2 \end{pmatrix}, \ \mathcal{A} = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 3 \\ 2 & 3 & 5 \\ 2 & 3 & 4 \\ 4 & 5 & 6 \end{pmatrix}$$

Figura 13: Ejemplo de un subconjunto I de *codewords* de $C_{1,1}$ no amigable con $\mathcal{A}$

Este es el paso de toda la generación del código lineal final que más tiempo tarda, ya que se deben realizar una gran cantidad de comparaciones, especialmente considerando que existen $\binom{M}{q}$ subconjuntos de q *codewords* de $C_{1,1}$ y $|\mathcal{A}|$ (El número de filas en $\mathcal{A}$ puede ser muy grande, ver sección 4.1), pero se va a introducir una manera para acelerar la búsqueda de estos conjuntos no amigables.

El algoritmo que busca estos conjuntos no amigables es muy parecido a aquel que calcula $\mathcal{S}(C_2)$ en la sección 3.2.1. Se realiza una búsqueda en árbol binario, pero donde el algoritmo que calculaba $\mathcal{S}(C_2)$ tenía que comprobar todas las posibles hojas de este árbol, en este caso, se pueden **podar** ciertas ramas, ya que si se detecta un subconjunto de *codewords* no amigable con $\mathcal{A}$, no hace falta tener en cuenta superconjuntos con los mismos *codewords* en el resto del algoritmo.

Algorithm 5 booleano BuscarConjuntosNoAmigables($C_{1,1}$, matriz $\mathcal{A}$, booleano $I[M]$, entero $d = 0$, entero $i = 0$, lista de *codewords* $L = \{\}$)

```
1: o ← verdadero
2: if i = q (Se han seleccionado q codewords) then
3:    o ← ConjuntoAmigable(C_{1,1}, A, I)
4:    if o = falso then
5:       Añadir I a L
6:    end if
7: else if M − d + i ≥ q then
8:    I_d = verdadero
9:    o ← BuscarConjuntosNoAmigables(C_{1,1}, A, I, d + 1, i + 1, L)
10:   I_d = falso
11:   if o = verdadero ∨ i = 0 then
12:      o ← BuscarConjuntosNoAmigables(C_{1,1}, A, I, d + 1, i, L)
13:   end if
14: end if
15: return o
```

Cuando el algoritmo computa la rama donde el conjunto I contiene el *codeword* de $C_{1,1}$ número d, si resulta que ese conjunto no es amigable de $\mathcal{A}$, podemos fácilmente podar la otra rama, ya que el conjunto I (sin contar el *codeword* con índice d), en ese momento, contiene *codewords* perteneciente a un subconjunto no amigable. Esta poda se ignora si $i = 0$, ya que I no contiene ningún *codeword* (Notemos que i indica cuántos *codewords* hay en I).

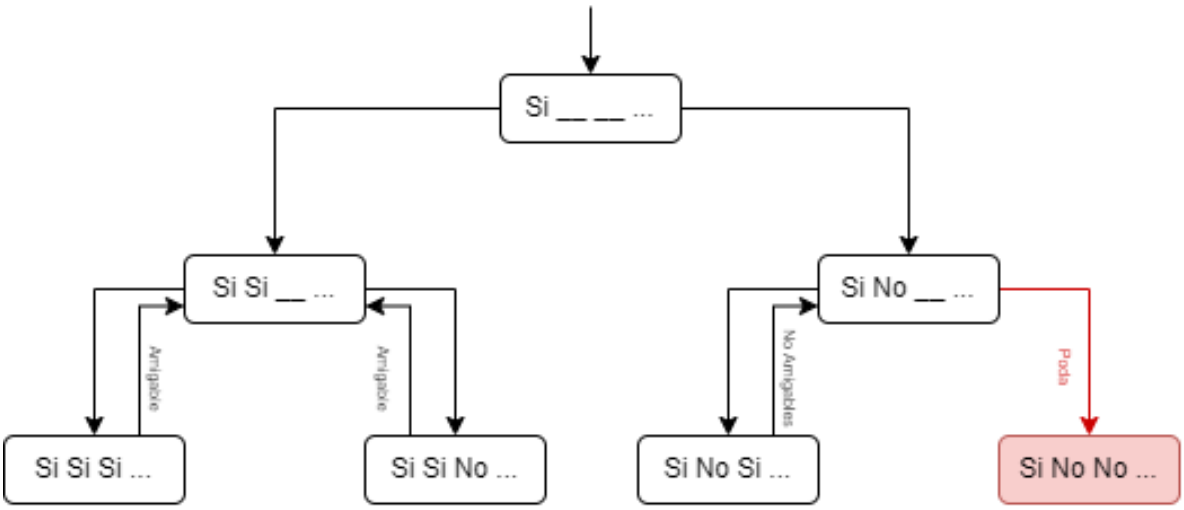


Figura 14: Extracto de un hipotético árbol, donde se puede ver una rama de este ha sido podada ya que contiene elementos en el subconjunto que se sabe que son no amigables a $\mathcal{A}$. (Este árbol no tiene relación con ningún ejemplo anterior)

Para comprobar si el subconjunto I es amigable con $\mathcal{A}$, se empezó con una estrategia que resultó ser lenta y requería de una matriz $\mathcal{A}$ diferente, pero se mejoró para reducir el tiempo que tarda en calcular esta amigabilidad y no necesitar una matriz $\mathcal{A}$ modificada.

Los M_2 *codewords* resultantes de este paso se llamaran $C_{1,2}$.

3.3.4. Amigabilidad de un subconjunto. Búsqueda Lineal

Como se ha mencionado al final de la sección anterior, se desarrollaron dos métodos para saber si un subconjunto es amigable con $\mathcal{A}$. El primero de estos métodos, el cual se describe en esta sección, tiene un coste temporal muy alto, ya que, en el peor caso, comprueba igualdad con cada uno de los elementos de $\mathcal{A}$.

También requiere de una versión modificada de $\mathcal{A}$ de como se calcula en la sección 3.2, donde no solo se incluyen los subconjuntos del código lineal C_2, si no también sus permutaciones.

Algorithm 6 booleano ConjuntoAmigable(*codewords* $C_{1,1}$, matriz $\mathcal{A}$, booleano $I[q] = \{\text{falso}, ..., \text{falso}\}$)

```
1: for all k ∈ [0, ..., |A| − 1] do
2:   v_a ← (A)_k (Fila k de A)
3:   for all i ∈ [0, ..., n_1 − 1] do
4:     o ← verdadero
5:     t ← verdadero
6:     l ← 0
7:     for all j ∈ [0, ..., M − 1] do
8:       v_b ← (C_{1,1})_j (Fila j de C_{1,1})
9:       if (I_j = verdadero) then
10:        if (v_a[l] ≠ v_b[i]) then
11:          o ← falso
12:          Terminar bucle j
13:        end if
14:        l ← l + 1
15:      end if
16:    end for
17:    if o = verdadero then
18:      return verdadero
19:    end if
20:  end for
21: end for
22: return falso
```

El algoritmo es muy simple. Para cada fila $\mathcal{A}_k$ de $\mathcal{A}$, comprobar si existe algún índice i de los *codewords* del subconjunto I cuyos valores sean igual a $\mathcal{A}_k$.

Un problema serio de este algoritmo es que la matriz $\mathcal{A}$ (Como se calcula en la sección 3.2) solo tiene una fila para cada subconjunto. Esto significa que, si los elementos del índice i del subconjunto I forman una fila que pertenece a $\mathcal{A}$, pero con los elementos están en otro orden, este algoritmo no considera ambas tuplas iguales. Para solucionar esto, se pueden incluir todas las permutaciones de todas las filas de $\mathcal{A}$ en la propia matriz, pero esto aumenta el número de filas a comprobar mucho más. Específicamente, el número de filas en $\mathcal{A}$ pasa a ser $q!$ veces más grande, lo que es un aumento muy drástico.

El número de comparaciones de igualdad entre tuplas, considerando lo anterior, es —$\mathcal{A}| \times n_1 \times q!$.

Teniendo todo esto en cuenta, se implementa un nuevo algoritmo que no necesite tener todas las permutaciones de las filas $\mathcal{A}$ y un tiempo de calculo mucho menor, el cual se ve en la siguiente sección.

3.3.5. Amigabilidad de un subconjunto. Divide y Vencerás

Para solucionar los problemas que tiene el algoritmo anterior, se ha implementado un algoritmo de búsqueda más rápido para saber si un elemento esta o no esta en la matriz $\mathcal{A}$. Teniendo en cuenta como calculamos esta matriz (Ver la sección 3.2), las filas de $\mathcal{A}$ están ordenadas, por lo que se utiliza, como en la sección 4.5, un algoritmo de divide y vencerás, el cual puede pasar el tiempo de búsqueda de un elemento en la matriz de depender completamente del número de elementos ($O(|\mathcal{A}|)$) a un tiempo que depende del logaritmo base 2 de este número ($\log_2(n)$).

La condición para dividir el espacio de búsqueda (Como se realiza en algoritmos de Divide y Vencerás) sera si la tupla a buscar es mayor o menor que aquella de $\mathcal{A}$ con la que se esta comparando. Una tupla v es menor que una tupla w si $v[i] < w[i]$ con i siendo el elemento número i donde v y w no coinciden en valor. Por ejemplo, $[1, 4, 6, 2]$ es menor que $[1, 4, 5, 4]$ ya que el primer elemento donde no coinciden (el tercer elemento) es menor en la primera tupla que en la segunda. De forma similar se

comprueba si una tupla es mayor.

El algoritmo modificado es el siguiente:

Algorithm 7 booleano ConjuntoAmigable(q-ary $C_{1,1}$, matriz $\mathcal{A}$, booleano $I[q]$)

1: $J \leftarrow$ Índices de I con valor verdadero
2: **for** $i \in [0, ..., n_1 - 1]$ **do**
3: $r \leftarrow$ Columna i del subconjunto, ordenada
4: **if** r tiene valores repetidos **then**
5: Pasar a la siguiente iteración del bucle
6: **end if**
7: $i_{start} \leftarrow 0$
8: $i_{end} \leftarrow 0$
9: $stage \leftarrow 0$
10: **while** $stage < q$ **do**
11: $i_{mid} = \lfloor \frac{i_{start}+i_{down}}{2} \rfloor$
12: $s \leftarrow$ Tupla número i_{mid} en $\mathcal{A}$
13: $stage \leftarrow 0$
14: **while** $stage < q \wedge s[stage] = r[stage]$ **do**
15: //Encontrar en qué elemento la tupla de $\mathcal{A}$ y la columna del subconjunto difieren
16: $stage \leftarrow stage + 1$
17: **end while**
18: **if** $stage = q$ **then**
19: //Una tupla de $\mathcal{A}$ y una columna del subconjunto coinciden
20: **return** Verdadero
21: **end if**
22: **if** $i_{start} = i_{end}$ **then**
23: //Final de la búsqueda por Divide y Vencerás. No se ha encontrado una tupla de $\mathcal{A}$ que tenga los elementos de la columna del subconjunto.
24: Pasar a la siguiente iteración del bucle
25: **end if**
26: **if** $s[stage] > r[stage]$ **then**
27: //Divide y Venceras. Reducir zona de búsqueda
28: $i_{end} = i_{mid}$
29: **else**
30: //Divide y Venceras. Reducir zona de búsqueda
31: $i_{start} = i_{mid} + 1$
32: **end if**
33: **end while**
34: **end for**
35: **return** Falso

Estas mejoras, comparado con el algoritmo visto en la sección 3.3.4, reduce drásticamente el número de comparaciones entre tuplas:

- Para $q = 4$, pasa de un máximo de $|\mathcal{A}| \times q! \times n_1 = 20544n_1$ comparaciones a $\lceil \log_2(|\mathcal{A}|) \rceil \times n_1 = 10n_1$.
- Para $q = 5$, pasa de un máximo de $|\mathcal{A}| \times q! \times n_1 = 1419600n_1$ comparaciones a $\lceil \log_2(|\mathcal{A}|) \rceil \times n_1 = 14n_1$.
- Para $q = 7$, pasa de un máximo de $|\mathcal{A}| \times q! \times n_1 = 1{,}64 \times 10^{10} n_1$ comparaciones a $\lceil \log_2(|\mathcal{A}|) \rceil \times n_1 = 22n_1$.

3.3.6. Paso 4: Eliminación de Colisiones

El último paso para construir el código exterior C_1 es eliminar las $o(M)$ colisiones en $C_{1,2}$, es decir, las *codewords* con el mismo valor. Esto significa pasar por todas las combinaciones de dos *codewords* en $C_{1,2}$, lo que implica como máximo $\frac{M_2(M_2-1)}{2}n_1$ comparaciones. El código resultante es C_1.

$$o(M) = \binom{M}{2} m^{-n_1}$$

Figura 15: Número de colisiones en $C_{1,1}$

3.4. Concatenación de Códigos. Códigos Exteriores e Interiores

Tras obtener ambos códigos C_1 y C_2, se puede realizar la concatenación de estos, lo cual da con el código lineal final C.

La formula para concatenar, tal como está descrita en el artículo original, es

$$C_F := \{\pi^{-1}(c) = (\pi^{-1}(c_1), \pi^{-1}(c_2), ..., \pi^{-1}(c_{n_1})) : c = (c_1, c_2, ..., c_{n_1}) \in C_1\}$$

la cual indica lo siguiente:

- El código lineal final C_F posee el mismo número de *codewords* que C_1
- La *codeword* número i de C_F es la concatenación de aplicar π^{-1} a cada elemento de la *codeword* número i de C_1.
- Recordemos que la función π^{-1} es igual que aquella descrita en la sección 3.2.2. $\pi^{-1}(j)$ tal que $i \in [0, ..., k_2 - 1]$ devuelve el *codeword* número j del código interior C_2.
- C_F tendrá *codewords* de longitud $n_1 * n_2$.
- Como los *codewords* de C_F son concatenaciones de aquellos de C_2, C_F será un q-ary.

En la siguiente página se muestra un ejemplo de concatenación.

$$C_2 = \begin{pmatrix} 0 & 2 & 1 \\ 1 & 0 & 0 \\ 1 & 2 & 0 \\ 2 & 2 & 1 \\ 2 & 2 & 0 \\ 2 & 0 & 2 \\ 1 & 1 & 1 \\ \vdots & \vdots & \vdots \end{pmatrix}, C_1 = \begin{pmatrix} 1 & 0 & 2 \\ 1 & 0 & 6 \\ 1 & 1 & 2 \\ 1 & 1 & 4 \\ 1 & 1 & 5 \\ 3 & 2 & 3 \\ 3 & 6 & 0 \\ \vdots & \vdots & \vdots \end{pmatrix}$$

(a) C_1 y C_2 de ejemplo

$$C_F = \begin{pmatrix} \pi^{-1}(1) & \pi^{-1}(0) & \pi^{-1}(2) \\ \pi^{-1}(1) & \pi^{-1}(0) & \pi^{-1}(6) \\ \pi^{-1}(1) & \pi^{-1}(1) & \pi^{-1}(2) \\ \pi^{-1}(1) & \pi^{-1}(1) & \pi^{-1}(4) \\ \pi^{-1}(1) & \pi^{-1}(1) & \pi^{-1}(5) \\ \pi^{-1}(3) & \pi^{-1}(2) & \pi^{-1}(3) \\ \pi^{-1}(3) & \pi^{-1}(6) & \pi^{-1}(0) \\ \vdots & \vdots & \vdots \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 2 & 1 & 1 & 2 & 0 \\ 1 & 0 & 0 & 0 & 2 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 2 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 2 & 2 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 2 & 0 & 2 \\ 2 & 2 & 1 & 1 & 2 & 0 & 2 & 2 & 1 \\ 2 & 2 & 1 & 1 & 1 & 1 & 0 & 2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

Figura 16: Ejemplo de concatenación de C_1 y C_2 para formar C_F

3.5. Optimizaciones

Mientras se ha implementado los algoritmos y funciones descritas en los apartados anteriores, se han ideado una serie de optimizaciones para acelerar el cálculo del código lineal final. Tras la implementación inicial, se han aplicado estas ideas, de las cuales la mayoría han conseguido reducir el tiempo de procesamiento.

3.5.1. Precálculo de la matriz $\mathcal{A}$

Ya que los códigos interiores C_2 son constantes para cada q (si se usa un mismo valor de α), estos códigos y la matriz $\mathcal{A}$ se pueden precalcular y guardar en disco para uso posterior.

En el hardware donde se ejecutó esta estrategia, los tiempos para calcular el código exterior C_1 bajaron, ya que el tiempo de carga de la matriz $\mathcal{A}$ desde disco resulto ser mucho menor que generar la matriz cada vez que se desea generar el código exterior C_1. Aun así, la mejora temporal decrece rápidamente cuanto más grandes son los *codewords*.

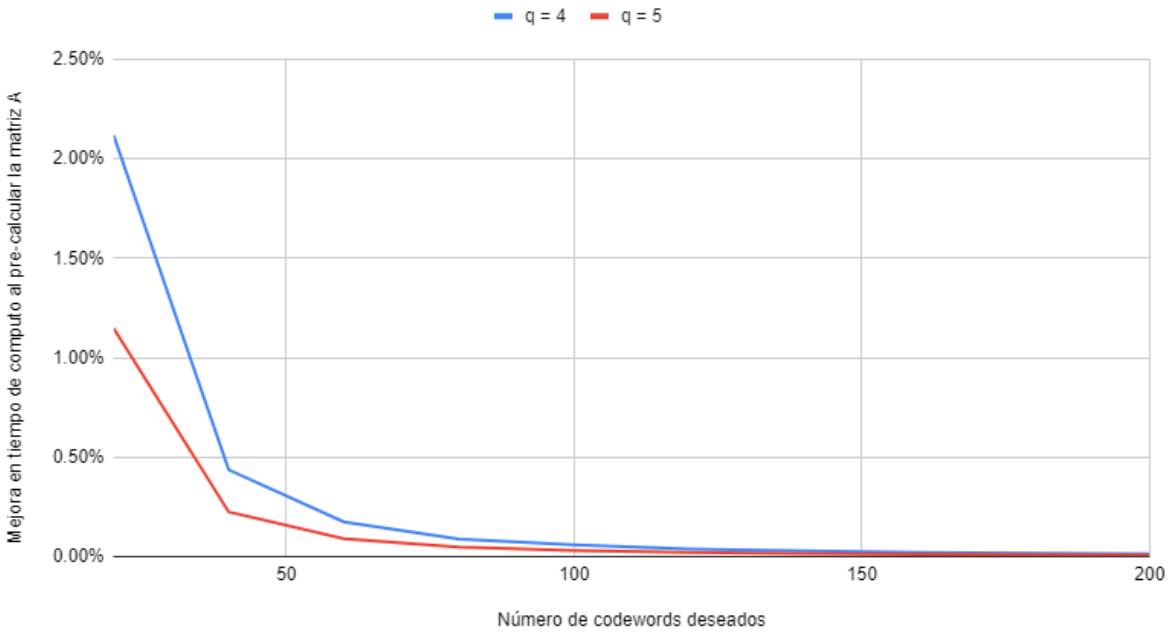


Figura 17: Mejora de tiempo con el pre-cálculo de la matriz $\mathcal{A}$

3.5.2. Cálculo de tamaño de *codewords* mínimo

Para poder construir el código exterior C_1, se necesitan insertar múltiples parámetros, pero uno de ellos, n_1, puede ser cualquier valor que desee el usuario o programador, mientras cumpla con la restricción $\binom{M}{q}\left(1-\frac{q!|\mathcal{A}|}{m^q}\right)^{n_1} \leq \frac{M}{2q}$ de la sección 3.3.1. Aquí buscamos una manera para obtener, de forma automática, un valor n_1 lo más pequeño posible para que se cumpla la desigualdad, así teniendo que computar menos datos.

Con un poco de álgebra, se puede despejar n_1 de la desigualdad original, dándonos una formula de los valores viables de n_1, del cual elegiremos el más pequeño:

$$\binom{M}{q}\left(1-\frac{q!|\mathcal{A}|}{m^q}\right)^{n_1} \leq \frac{M}{2q}$$

$$\left(1-\frac{q!|\mathcal{A}|}{m^q}\right)^{n_1} \leq \frac{M}{2q}\binom{M}{q}^{-1}$$

$$a = \left(1-\frac{q!|\mathcal{A}|}{m^q}\right) \mid b = \frac{M}{2q}\binom{M}{q}^{-1}$$

$$a^{n_1} \leq b \rightarrow \log_a a^{n_1} \geq \log_a b \rightarrow n_1 \geq \log_a b = \frac{\log(b)}{\log(a)}$$

$$n_1 = \left\lceil \log_{\left(1-\frac{q!|\mathcal{A}|}{m^q}\right)}\left(\frac{M}{\binom{M}{q}2q}\right)\right\rceil = \left\lceil \frac{\log\left(\frac{M}{\binom{M}{q}2q}\right)}{\log\left(1-\frac{q!|\mathcal{A}|}{m^q}\right)}\right\rceil$$

Figura 18: Obtención de una formula para calcular n_1 mínimo para C_1

Con esto, se añadió una opción al generar el código lineal final C donde no se necesita insertar el valor n_1, si no se que calcula automáticamente. Ver la sección 4 para más detalles en el comportamiento de esta variable.

3.5.3. Un intento de optimización fallido

Una idea que se tuvo para reducir el tiempo de cómputo de la sección 3.3.3 es, en vez de comprobar si cierta tupla c está o no en la matriz $\mathcal{A}$, la cual contiene los subconjuntos de C_2 que están separados y puede llegar a tener un gran tamaño, calcular en ese momento si el subconjunto de C_2 que describe c esta separado. Específicamente, en vez de comprobar que cada columna d de cada subconjunto este o no en $\mathcal{A}$, se comprueba que los *codewords* $d_1, d_2, ...$ de C_2 están separados.

Esta optimización se intentó implementar dos veces. Una a principios de la implementación, y otra tras la implementación final.

En teoría, esto puede ayudar a no tener que comprobar que c esté en una matriz $\mathcal{A}$ de gran tamaño (como aquella de $q = 7$, que llega a más de 3 millones de elementos), pero puede dañar a aquellos códigos con un tamaño de la matriz $\mathcal{A}$ más pequeño. Para probar esta estrategia, se calcularon múltiples códigos lineales con 50 *codewords* y tamaño de cada *codeword* calculado automáticamente. Los resultados vistos son los siguientes:

- Para $q = 4$, el cálculo del código C pasó a ser $\approx 2{,}62$ veces más lento.
- Para $q = 5$, el cálculo del código C pasó a ser $\approx 3{,}16$ veces más lento.

Al contrario de lo que se pensaba, esta estrategia era peor para valores de q más grandes. Esto se puede deber a que comprobar la separación de un subconjunto de C_2 con $q = 4$ requiere de menos tiempo que uno de $q = 5$ o más, ya que se pasa de comprobar la separación entre 4 *codewords* a 5.

La ventaja que se podría haber obtenido al no tener que realizar búsquedas en matrices $\mathcal{A}$ con millones de filas también se pierde, ya que el algoritmo implementado en la sección 3.3.5, que se implemento más tarde que esta estrategia, resuelve directamente ese problema.

Aun con estos resultados, esta estrategia se podría revisar en el futuro. Como la estructura de los códigos interiores C_2 son conocidas, se podría buscar alguna manera donde utilizando los valores de cada columna d, sin utilizar los *codewords* de un código C_2 ya calculado, comprobar directamente si describe un subconjunto amigable o no.

4. Pruebas y Resultados

4.1. Tamaño de la matriz $\mathcal{A}$

Tenemos una colección de códigos interiores C_2 según el valor q de estos códigos, construidos a partir de las instrucciones en la sección 3.1. Los códigos que hemos usado en nuestras pruebas son $q = 4$ (método en la sección 3.1.1) y $q = 5, 7$ (método en la sección 3.1.2).

A partir de estos códigos se han generado las matrices $\mathcal{A}$ de cada uno de estos. Como hemos visto en las secciones 3.5.1, esto se realiza para reducir el tiempo de computo del código linear final.

Los tamaños de cada $\mathcal{A}$ obtenidos son los siguientes (todos ellos se han calculado con $\alpha = 2$):

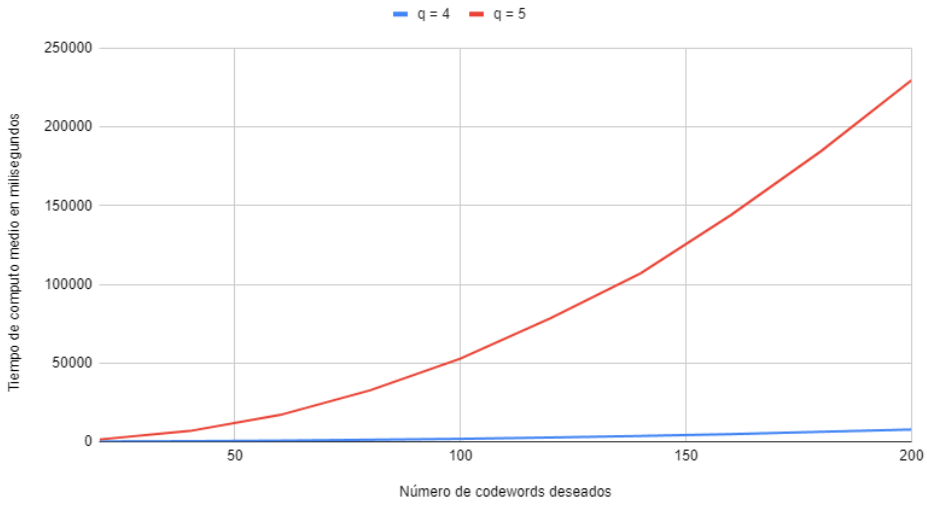
- Para $q = 4$: $|\mathcal{A}| = 856$
- Para $q = 5$: $|\mathcal{A}| = 11,830$
- Para $q = 7$: $|\mathcal{A}| = 3,264,436$

Como se puede ver, el tamaño de la matriz crece rápidamente con respecto a cuanto más grande es el valor q del código interior. Esto explica los tiempos tan grandes que veremos en el computo del código linear final (Se requiere hacer muchas búsquedas en la matriz $\mathcal{A}$, como se ha descrito en el apartado 3.3.3).

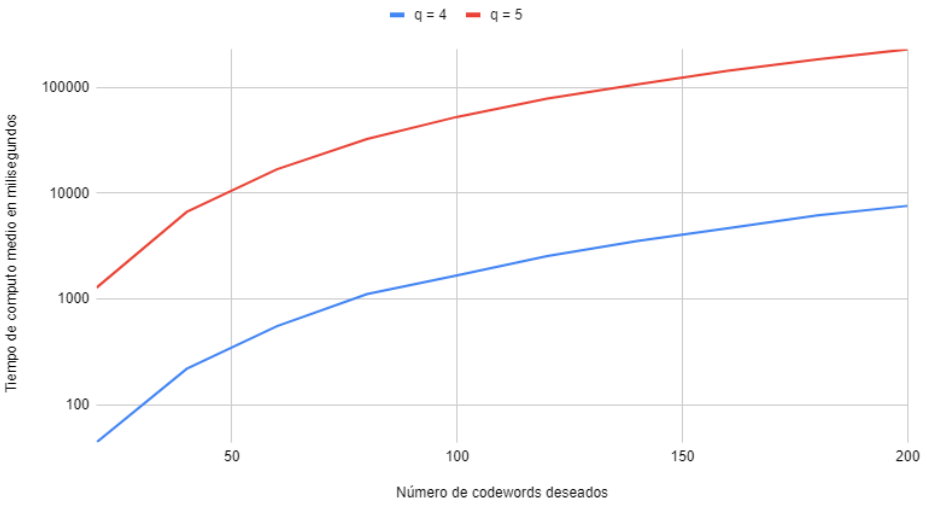
4.2. Tiempo de cómputo para el código linear final

Ahora vamos a ver el tiempo de cómputo que ha costado el cálculo de códigos finales según el código interior utilizado $(C_2, q, \mathcal{A})$ y el número de *codewords* y longitud deseados.

El siguiente gráfico muestra el tiempo de cómputo para una serie de códigos lineales que van desde 20 *codewords* deseados ($M = 60$), hasta 200 *codewords* deseados ($M = 600$). El tamaño de cada *codeword* se calcula automáticamente como se describe en el apartado 3.5.2.



(a) Eje Y lineal



(b) Eje Y en escala logarítmica

Figura 19: Tiempo de cómputo para códigos lineares con un número deseado de *codewords* (eje X) y el tiempo de computo en milisegundos (eje Y), para $q = 4, 5$

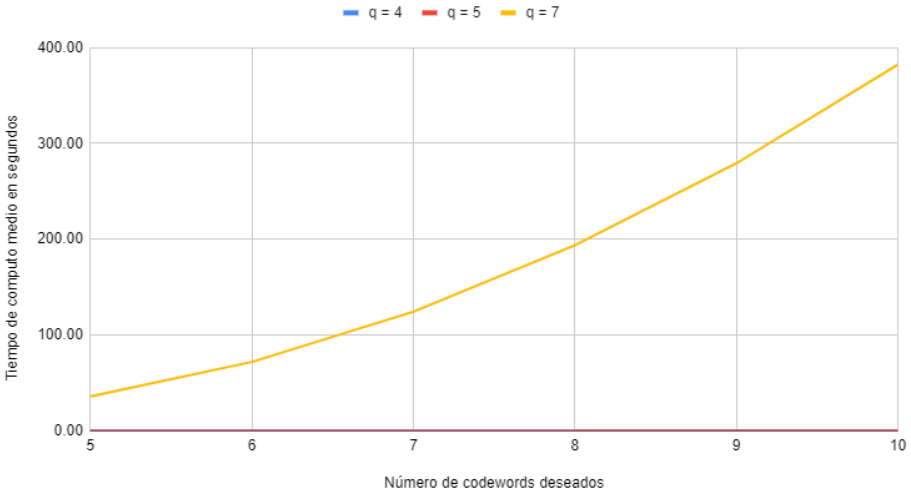
Cada código lineal se ha calculado 4 veces para amortiguar distintas cargas que el ordenador que realizo los cálculos pudiera tener en esos momentos. Como se puede ver, el tiempo de cómputo crece exponencialmente cuantas más *codewords* deseemos. Este gran gasto temporal se debe principalmente a la búsqueda de aquellos subconjuntos que no sean amigables con $\mathcal{A}$ (Ver sección 3.3.3), el cual es muy difícil de optimizar por su naturaleza, ya que se trata de una búsqueda en una lista muy grande.

Para esta prueba, se han considerado códigos lineales con $q = 4$ y $q = 5$. Pruebas con aquellos códigos lineales con $q = 7$ se muestran más tarde en este apartado por su gran diferencia de tiempo de calculo.

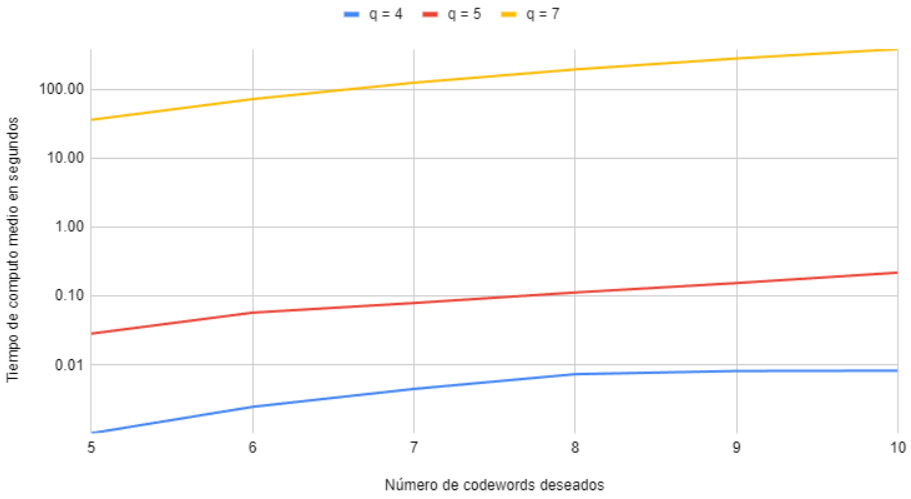
4.3. Precálculo de la matriz $\mathcal{A}$. Tiempo ganado

Como ya se explicó en la sección 3.5.1, la matriz $\mathcal{A}$, la cual existe una para cada q, es precalculada de ante mano y guardada en disco para que no se tenga que calcular cada vez que se desee generar un código lineal.

También se ha realizado una prueba con $q = 7$ para $[5, 6, ..., 10]$ *codewords* deseados por código. Se puede observar que el tiempo de computo para códigos lineales con $q = 7$ es mucho más alto que para el resto. Esto se debe a que la matriz $\mathcal{A}$ para $q = 7$ tiene un gran tamaño (como se ha visto en 4.1, tiene un tamaño de $3,264,436$), lo que significa muchas más comparaciones.



(a) Eje Y lineal ($q = 4, 5$ muy pequeñas para verse aquí, ver la siguiente figura)



(b) Eje Y en escala logarítmica

Figura 20: Tiempo de cómputo para códigos lineares con un número deseado de *codewords* (eje X) y el tiempo de computo en segundos (eje Y), para $q = 4, 5, 7$

Podemos ver el tiempo que se ha ahorrado al ver el tiempo que tardan las matrices en ser generadas (pero no en ser guardadas en disco), los cuales son los siguientes para $q = 4, 5, 7$:

- Tiempo para $\mathcal{A}$ con $q = 4$: 0,959ms
- Tiempo para $\mathcal{A}$ con $q = 5$: 14,908ms
- Tiempo para $\mathcal{A}$ con $q = 7$: 21430,1ms

Si comparamos con los tiempos obtenidos en la sección 4.2, podemos observar como este precalculo de la matriz ayuda en casos con pocos *codewords* y cuanto menor sea el valor q.

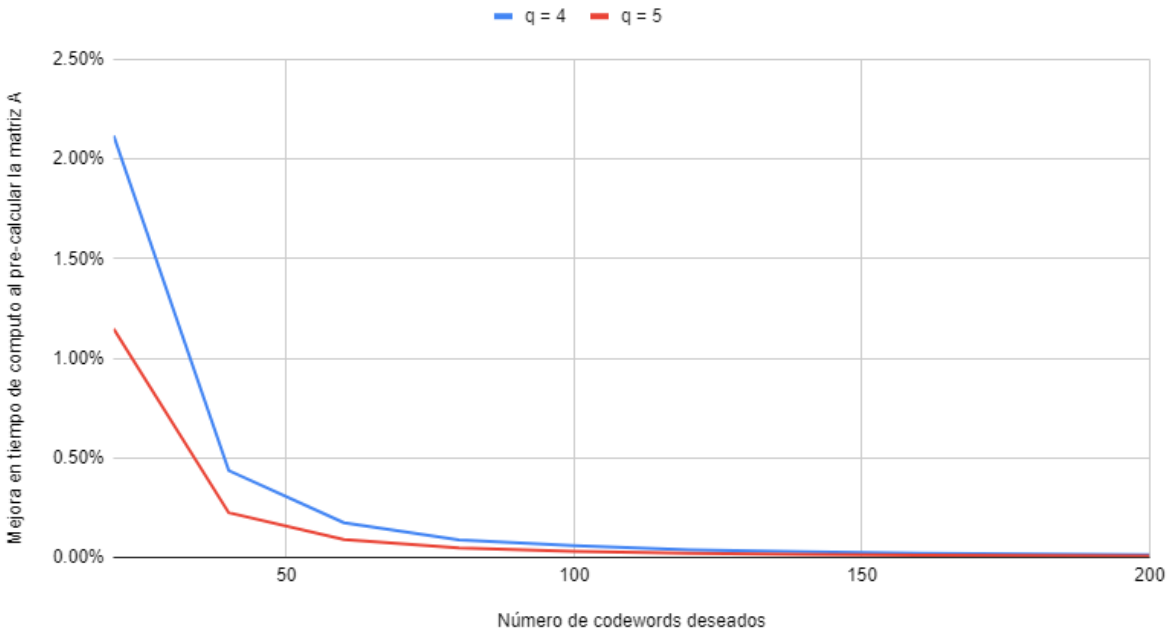


Figura 21: Mejora de tiempo con el pre-cálculo de la matriz $\mathcal{A}$

4.4. Longitud de *codewords* según cantidad

Ya se ha descrito en la sección 3.5.2 que se puede calcular la longitud de los *codewords* en el código lineal C_1 de forma automática para que la restricción del apartado 3.3.1 se cumpla, pero no hemos analizado que valores toma n_1 según el número de *codewords*, el valor de q o su correspondiente matriz $\mathcal{A}$.

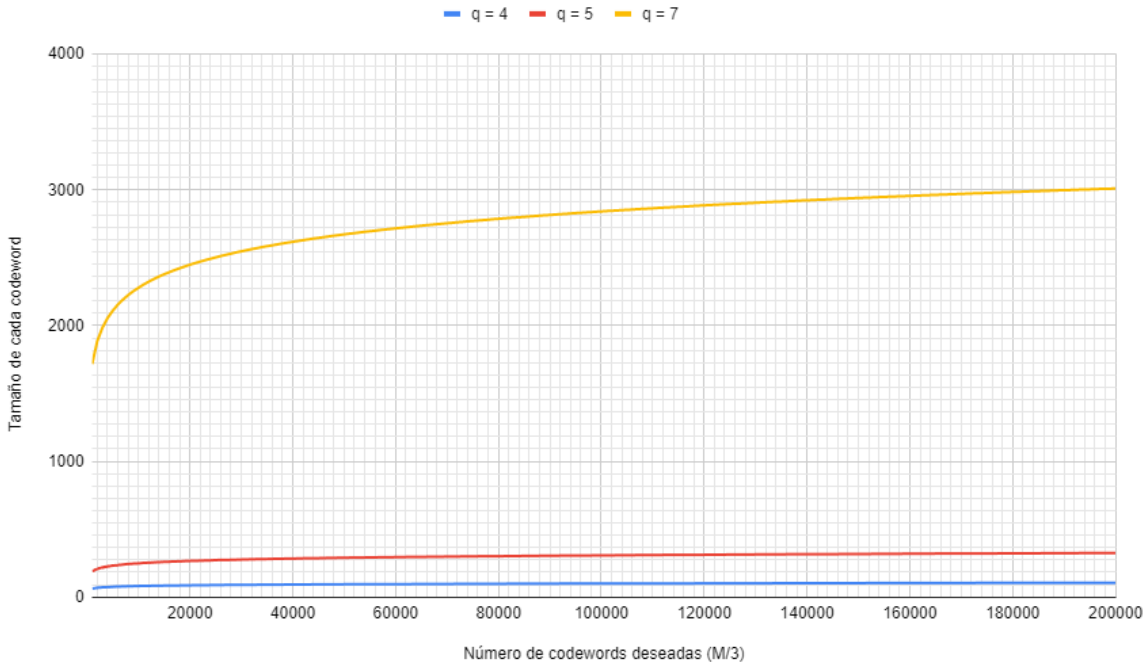


Figura 22: Crecimiento de n_1 en el código exterior C_1 según $\frac{M}{3}$ (número de *codewords* que se desean) y el código interior C_2 (q y $\mathcal{A}$)

Se puede observar en la figura 22 una gráfica que compara el número de *codewords* que se desean ($\frac{M}{3}$) en el eje X y la longitud mínima requerida de estos *codewords* en el eje Y. Se puede observar como se requiere que los *codewords* sean de mayor longitud cuanto más *codewords* queremos. Esto tiene sentido, ya que cuanto mayor longitud tengan, menor probabilidad de colisión existe. También se puede observar en la misma figura que cuanto más grande es el valor q, los *codewords* requieren una mayor longitud.

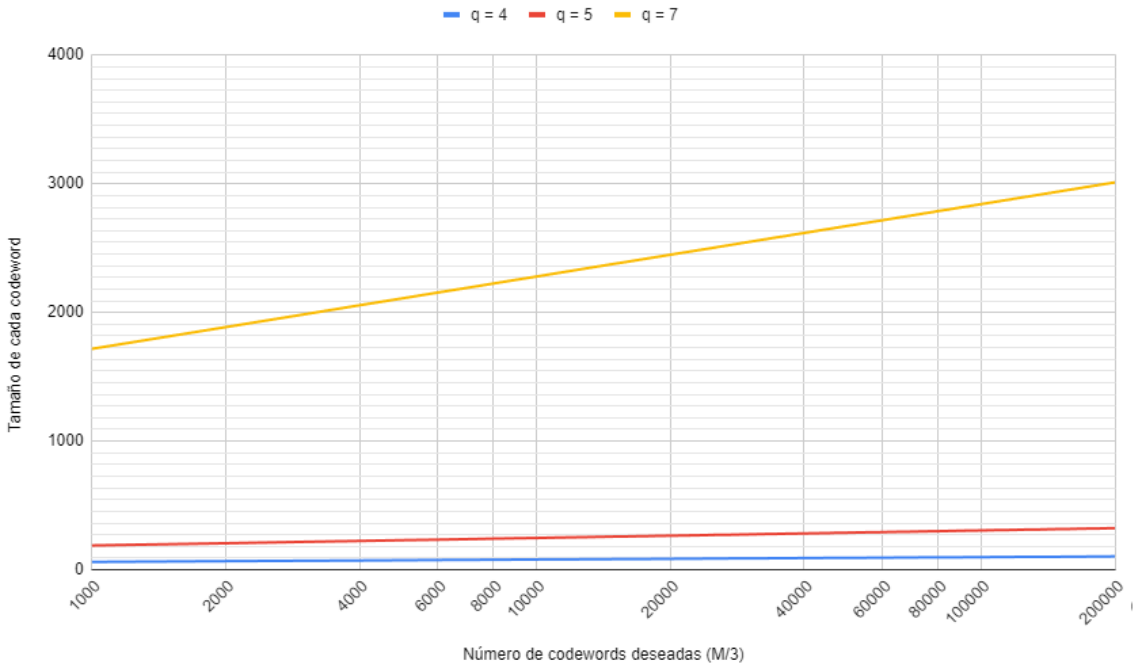


Figura 23: Figura 22 con eje X en escala logarítmica

En la Figura 23, se muestra la misma gráfica, pero con el eje X siendo mostrado en escala logarítmica. Se observa que el crecimiento de n_1 es muy logarítmico (Aunque no perfecto). Esto se explica al ver la función (Ver Figura 18), la cual tiene la forma $n_1 = \log_a(b)$, donde a es una variable que depende únicamente del código interior C_2, es decir, q y $\mathcal{A}$, por lo que los cambios a M modifican logarítmica-mente el valor final n_1.

4.5. Cantidad de *codewords* según longitud

En el artículo, se ha hablado mucho del **Ratio** R_c del código. Este se calcula con la fórmula $R_c \leq \frac{\log_2(k)}{n}$, con k y n siendo la longitud y tamaño del código lineal respectivamente. k, para el código final, es igual a $M(n, q)$, o el número de *codewords* que el código final tendrá para n y q determinados.

Por la restricción en la sección 3.3.1, existe un número de *codewords* máximo M para una longitud de *codeword* n_1 y código exterior C_2 determinados. Tenemos que recordar que M es un parametro de entrada para la construcción del código C_1 que no representa la cantidad de *codewords* en el código final.

Al contrario que al calcular la longitud óptima (ver sección 3.5.2), despejar M de la restricción (ver figura 18) es muy difícil, por lo que se ha usado un pequeño algoritmo de tipo *Divide y Vencerás*. Para nosotros, esto significa que nuestro algoritmo probará la desigualdad en un rango de valores de M pero sin probar todos los posibles valores.

Nuestro algoritmo empieza comprobando la desigualdad para el valor M en mitad de este rango, por ejemplo, $M = 150$ para el rango $[100, ..., 200]$. Si este M cumple con la condición pero $M + 1$ no, se ha encontrado el valor de M más grande que cumple con la restricción. Si no, dependiendo de si M cumple con la restricción, se reduce el rango por un lado u otro. Este proceso se repite hasta que se ha encontrado M óptimo o el rango es un único número (en su caso, se habrá encontrado M óptimo)

Aun así, primero se ha simplificado la desigualdad de la restricción para que tengamos que calcular menos datos, con la desigualdad ahora teniendo la forma de:

$$\frac{2q}{q!}\left(1-\frac{q!|\mathcal{A}|}{m^q}\right)^{n_1} \leq M\frac{(M-q)!}{M!} = \left(\prod_{i=1,\ldots,q-1}(M-i)\right)^{-1}$$

Figura 24: Restricción (Figura 11) con una forma más fácil para computar el valor de M óptimo

Específicamente, $l = \frac{2q}{q!}\left(1-\frac{q!|\mathcal{A}|}{m^q}\right)^{n_1}$ no depende de M, por lo que solo lo tendremos que calcular una sola vez. El lado derecho se puede calcular con el algoritmo *Calcular_M_Lado_Derecho*, el cual posee una propiedad que se usará a continuación:

$$\text{Calcular\_M\_Lado\_Derecho}(M+1,q) = \textit{Calcular\_M\_Lado\_Derecho}(M,q)\frac{M-q+1}{M}$$

El algoritmo para calcular el número de *codewords* M es el siguiente:

Algorithm 8 Calcular_M(entero m, entero q, entero n_1, entero $|\mathcal{A}|$, entero M_{MAX})

$l \leftarrow \frac{2q}{q!} * (1-\frac{q!|\mathcal{A}|}{m^q})^{n_1}$ *//Lado izquierdo de la desigualdad*
$r_{min} \leftarrow q$
$r_{max} \leftarrow M_{MAX}$
$M \leftarrow r_{max}$

Primero comprobamos que M_{MAX} esta dentro del rango para el código C_1
$r \leftarrow$ Calcular_M_Lado_Derecho(M,q) *//Lado derecho de la desigualdad*
if $l \leq r$ **then**
  return M_{MAX}
end if

Usar Divide y Vencer para encontrar M máximo que cumpla la restricción
while $r_{min} \neq r_{max}$ **do**
  $M \leftarrow r_{min} + \lfloor \frac{r_{max}-r_{min}}{2} \rfloor$
  $r \leftarrow$ Calcular_M_Lado_Derecho(M,q) *//Lado derecho de la desigualdad*
  if $l \leq r$ **then**
    $r \leftarrow \frac{M-q+1}{M}$ *// Lado derecho para $M+1$*
    if $l > r$ **then**
      return M
    else
      $r_{min} \leftarrow M$
    end if
  else
    $r_{max} \leftarrow M$
  end if
end while

Con los resultados que este algoritmo nos da, podemos ver que el valor óptimo de M crece muy rápidamente cuanto más grande es n_1.

Algorithm 9 Calcular_M_Lado_Derecho(entero M, entero q)

```
r ← 1
for all i ∈ [1, 2, ..., q] do
    r ← (M − i)r
end for
r ← r^{−1}
return r
```

A continuación se muestra un gráfico que indica el gran crecimiento del valor de M para distintas longitudes de *codewords*:

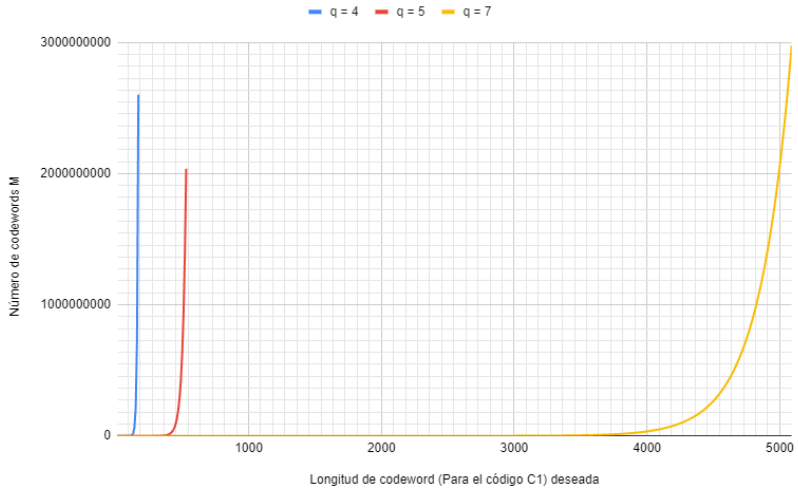


Figura 25: Crecimiento de M óptimo en el código exterior C_1 según n_1 (Longitud de *codeword*) y el código interior C_2 (q y $\mathcal{A}$). La búsqueda se realiza con el rango $[q, 3 * 10^9]$.

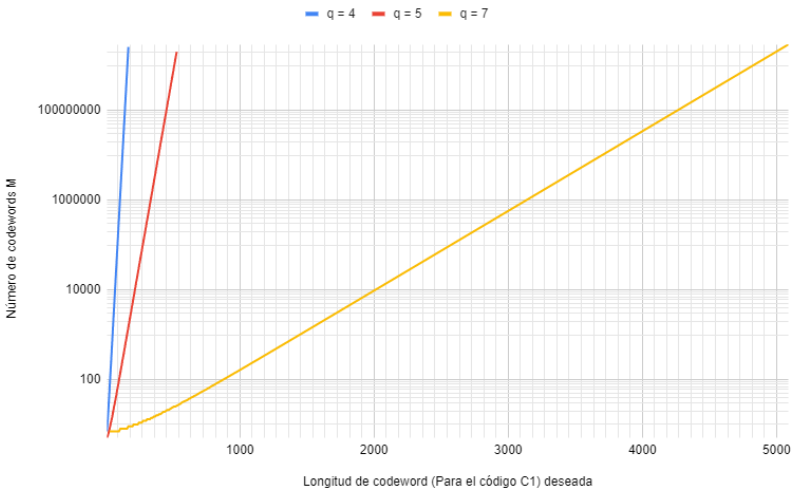


Figura 26: Figura 25 con eje Y en escala logarítmica

4.6. *Rate* mínimo teórico

Como ya se ha explicado en la sección 2.4.3, cada código lineal posee un valor llamado *Rate* que indica cómo de redundante es la información en este código. Hemos realizado dos pruebas con respecto a este valor, con una centrándose en el valor mínimo posible que se puede obtener con un número de *codewords* deseado, y otra prueba (ver sección 4.7) que analiza el *Rate* de códigos lineales generados con esta implementación.

Recordemos que en nuestro caso, la manera para calcular este valor es $R_C = \frac{\log_2|C|}{n}$, con $|C|$ siendo el número de *codewords* del código C, equivalente a su valor k.

Primero hemos calculado cuáles son los valores mínimos teóricos del *Rate* de diversos códigos lineales con distintos números deseados de *codewords* para valores distintos de q, los cuales son los siguientes:

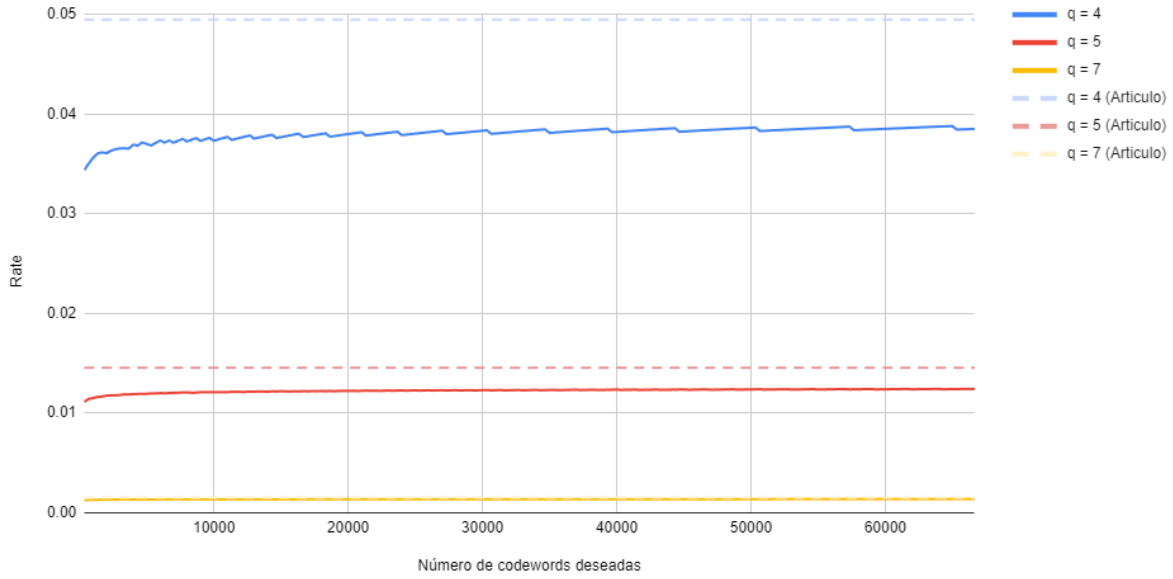


Figura 27: *Rate* mínimo teórico de diversos códigos lineares para $q = 4, 5, 7$ y número concreto de *codewords* deseados. Se muestran también los *Rate* dados por el artículo original.

Como se puede ver, el *Rate* de todos estos códigos lineales están por debajo del *Rate* descrito por el artículo para cada uno de los valores de q. Eso significa que si un código lineal acaba con $\lceil \frac{M}{3} \rceil$ *codewords* tras eliminar *codewords* no amigables en el código exterior (Ver sección 3.3.3), su *Rates* será mejor que el límite superior calculado por el artículo original [1].

4.7. *Rate* obtenido

Con los mismos códigos generados para la sección 4.2, se ha calculado el *Rate* de estos, con los siguientes resultados:

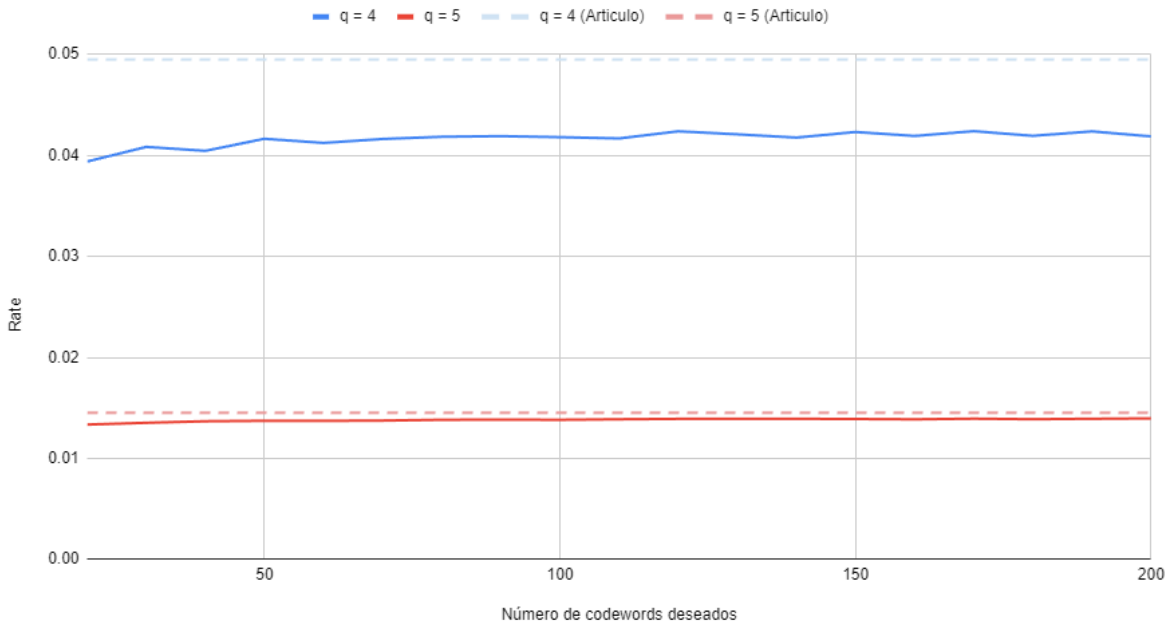


Figura 28: *Rate* de diversos códigos lineares generados para $q = 4, 5$ y número concreto de *codewords* deseados. Se muestran también los *Rates* dados por el artículo original.

Al igual que los *Rates* calculados en el apartado anterior, estos están por debajo del límite superior predicho por el artículo original [1].

5. Comparación con *GPERF*

GPERF [8] es un generador de funciones hash perfectas para C y C++. Parte de proyecto GNU, este es uno de los generadores de funciones hashes más populares de internet. Específicamente, este programa permite generar código en C/C++ que contiene funciones hashes para colecciones de claves, con el objetivo de poder crear **tablas hash** rápidas.

Una tabla hash es una tabla de pares de claves y datos, donde la posición en memoria del dato depende del valor hash de la clave. Este tipo de estructura de dato es muy usado en muchos campos de la programación para rápido acceso a información dinámica o leíble por un ser humano.

Para comparar ambos sistemas, aquel que hemos implementado en este proyecto y GPERF, podemos hacer que este último genere cada una de las funciones hash perfectas de la familia de hashes perfectos que da nuestra implementación una a una. El número de funciones de hash perfecto que nuestras familias de hashes perfectos tiene es igual a $\binom{M(n,q)}{q}$, con $M(n,q)$ siendo el número de *codewords* y el número de claves globales, mientras que q es el número de claves de cada función hash.

Para igualar la versatilidad de nuestra implementación, GPERF tiene que generar el siguiente número de funciones de hash perfecto:

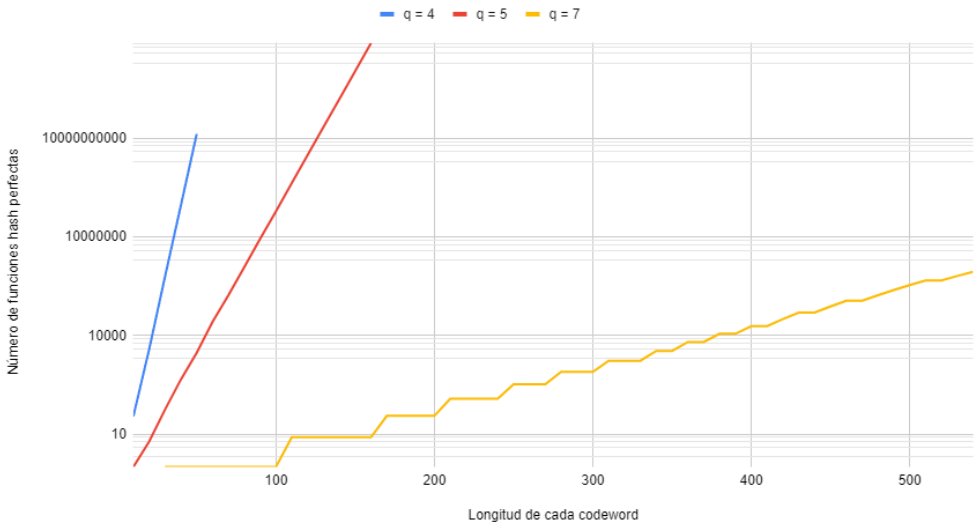


Figura 29: Número de funciones hash perfectas para un código lineal con una longitud específica de *codeword*, eje Y en escala logarítmica

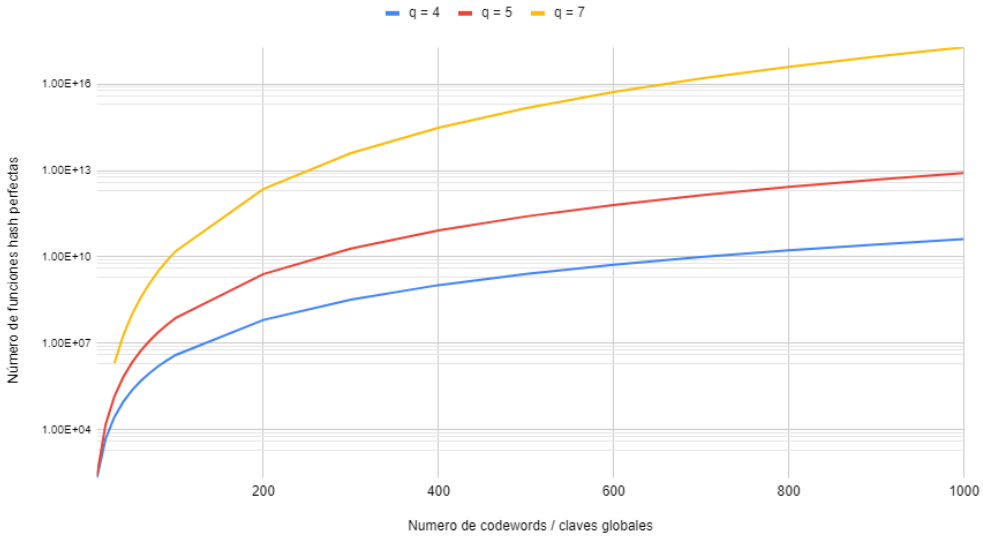


Figura 30: Número de funciones hash perfectas para un código lineal con un número especifico de *codewords* / claves globales, eje Y en escala logarítmica

El número de funciones que GPERF necesita generar para igualar la versatilidad de nuestras familias de hash perfectos crece exponencialmente con respecto a la longitud de cada *codeword* o el número de *codewords*. Por ejemplo, para imitar una familia dada por un código lineal de $q = 4$ y 200 *codewords*, GPERF necesita generar $6{,}47 \times 10^7$ funciones hash perfectas.

Comparar el tiempo que se tardaría en generar todas estas funciones con GPERF es dificil de calcular. Para $q = 4, 5, 7$ claves, se tardan $\approx 0{,}017$ segundos para generar una función hash, pero como GPERF genera código fuente para C++, eso no tiene en cuenta el tiempo que se tarda en compilar tantas funciones, o el tiempo que se tarda en generar todos los archivos (tantos como funciones se quieren generar) que contienen las claves para las funciones, algo que GPERF requiere.

Aunque GPERF genere funciones hash perfectas que procesan datos más rápidamente (Ya que la función es compilada), tener que guardar millones de funciones hash distintas resulta en un gran coste en memoria.

Referencias

[1] Chaoping Xing, Chen Yuan (2019) *Beating the probabilistic lower bound on perfect hashing (Congress y Revision 2)*, Electronic Colloquium on Computational Complexity.

[2] Rocert A. Walker II, Charles J. Colbourn (2007) *Perfect Hash Families: Constructions and Existence*, Journal of Mathematical Cryptology, pp 125-150.

[3] J. Körner and K. Marton. (1988) *New bounds for perfect hashing via information theory.* European Journal of Combinatorics, pp 523-530.

[4] Fredman, Michael L. and John Komlos. "On the Size of Separating Systems and Families of Perfect Hash Functions." Siam Journal on Algebraic and Discrete Methods 5 (1984): 61-68.

[5] Venkatesan Guruswami (2010) *Introduction to Coding Theory, Linear Codes*, Carnegie Mellon University, School of Computer Science.

[6] Y. Lu, B. Prabhakar and F. Bonomi, "Perfect Hashing for Network Applications,"2006 IEEE International Symposium on Information Theory, 2006, pp. 2774-2778, doi: 10.1109/ISIT.2006.261567.

[7] Código fuente de la implementación en *Github*

[8] *GPERF*