Un acercamiento a Magma a través de Sage por medio de hashing

César Agustín García Vázquez

2 de abril de 2009

ÍNDICE GENERAL

INTRODUCCIÓN

La Teoría de Códigos comenzó a finales de los años 40's del siglo XX con el trabajo de Shannon, Hamming, Golay y otros ([34]).

Un *código* $\mathcal{C}$ de longitud n y tamaño M consiste en un conjunto de M cadenas donde cada una de estas cadenas tiene n componentes en cierto alfabeto finito S. A los elementos de un código se les llama *palabras-código*[1]. *La distancia de Hamming* entre dos palabras-código es el número de componentes donde éllas difieren y d es el mínimo de las distancias de Hamming entre dos cualesquiera palabras-código distintas. Así, un *código* $\mathcal{C}$ *sobre* S es un subconjunto de S^n cuyos parámetros son (n, M, d).

En la teoría clásica el alfabeto S es un campo finito $\mathbb{F}_q = GF(q)$ (cf. [31, 48]), donde $q = p^n$, con p un primo y un entero positivo $n \geq 1$.

El campo finito $\mathbb{F}_2$ es muy especial en la teoría de códigos. Los códigos sobre $\mathbb{F}_2$ son llamados códigos binarios, éstos representan el caso más simple e importante, y con éllos se inició la teoría de códigos. Posteriormente se consideraron códigos sobre los campos $\mathbb{F}_3$ y $\mathbb{F}_4$, y más tarde, se estudiaron los códigos sobre cualquier campo finito $\mathbb{F}_q$.

Más recientemente, el interés de los matemáticos en los códigos sobre anillos finitos se despertó por el artículo escrito por Hammons et al. ([23]). En este artículo, se demostró que algunos códigos binarios no lineales interesantes, como los códigos de Kerdock, Preparata y Goethals, pueden ser vistos como imágenes de códigos lineales sobre $\mathbb{Z}_4$ vía el mapeo de Gray de $\mathbb{Z}_4^n$ a $\mathbb{F}_2^{2n}$ y que su aparente dualidad podría describirse en términos de su dualidad en $\mathbb{Z}_4^n$. En los años recientes, han aparecido muchos artículos de códigos sobre anillos finitos $\mathcal{R}$, por ejemplo, los anillos $\mathbb{Z}_m$, los

[1] palabras código

anillos de cadena finita y anillos de Galois (ver [9, 12, 24, 32, 33, 36, 39, 49]).

Para introducirse en el lenguaje de la teoría de códigos se requieren algunos conceptos básicos fundamentales como los que a continuación se enuncian.

Sea $\mathbb{F}_q^n$ el espacio vectorial de todos los vectores de longitud n sobre el campo finito $\mathbb{F}_q$. Si $\mathcal{C}$ es un subespacio k-dimensional de $\mathbb{F}_q^n$ se dice que $\mathcal{C}$ es un $[n, k, d]$ *código lineal sobre* $\mathbb{F}_q$.

Sea $\mathcal{R}$ un anillo (conmutativo). Un $\mathcal{R}$-módulo es un grupo abeliano M en el cual $\mathcal{R}$ actúua linealmente: más precisamente, es una pareja (M, μ) donde M es un grupo abeliano y μ es un mapeo de $\mathcal{R} \times M$ en M tal que, si se escribe rx para $\mu(r, x)$ $(r \in \mathcal{R}, x \in M)$, los siguientes axiomas se satisfacen:

$$\begin{aligned} r(x+y) &= rx + ry \\ (r+b)x &= rx + bx \\ (rb)x &= r(bx) \\ 1x &= x \qquad (r, b \in \mathcal{R}; x, y \in M) \end{aligned}$$

Sea $\mathcal{R}^n$ el $\mathcal{R}$-módulo de todas las n-tuplas sobre un anillo finito con identidad $\mathcal{R}$, por un *código lineal* $\mathcal{C}$ *sobre* $\mathcal{R}$ se entiende cualquier $\mathcal{R}$-submódulo de $\mathcal{R}^n$.

Un código lineal $\mathcal{C}$ sobre $\mathbb{F}_q$ (ó $\mathcal{R}$) es llamado *cíclico* si siempre que $(a_0, a_1, \ldots\ldots, a_{n-1}) \in \mathcal{C}$ entonces $(a_{n-1}, a_0, a_1, \ldots, a_{n-2}) \in \mathcal{C}$. Se dice que dos códigos son *equivalentes salvo permutación* si uno puede ser obtenido a partir del otro por medio de la permutación de sus coordenadas.

En la teoría de códigos algebraicos, las matrices generadoras juegan un rol muy importante.

Para códigos lineales de dimensión k sobre un campo, una matriz generadora G está definida como una matriz cuyas filas forman una base para el código. En otras palabras, las filas generan el código, y éllas son linealmente independientes. Se dice que G está en *forma estándar* si $G = [I_k|A]$ donde I_k es la matriz identidad de tamaño $k \times k$.

Para módulos definidos sobre anillos con divisores de cero, $\mathcal{R}$, por supuesto no es posible hablar del concepto de dimensión (los módulos no son libres). No obstante, los códigos sobre los anillos $\mathbb{Z}_{p^k}$ han sido extensivamente estudiados ([8, 24, 36]) y es en parte debido a que para un código lineal $\mathcal{C}$ no nulo sobre $\mathbb{Z}_{p^k}$ una matriz G es una matriz generadora de $\mathcal{C}$ si sus vectores[2] fila generan

[2] Abusando de la terminología nos referimos a un n-tupla en $\mathcal{R}^n$ como un "vector" [36].

a $\mathcal{C}$ y ninguno de ellos puede escribirse como combinación lineal de los demás [36] la cual después de una permutación apropiada de sus coordenadas puede ser escrita en la *forma estándar*

$$\begin{bmatrix} I_{k_0} & A_{01} & A_{02} & A_{03} & \dots & A_{0,k-1} & A_{0k} \\ 0 & pI_{k_1} & pA_{12} & pA_{13} & \dots & pA_{1,k-1} & pA_{1k} \\ 0 & 0 & p^2I_{k_2} & p^2A_{23} & \dots & p^2A_{2,k-1} & p^2A_{2k} \\ . & . & . & . & \dots & . & . \\ 0 & 0 & 0 & 0 & \dots & p^{k-1}I_{k_{k-1}} & p^{k-1}A_{k-1,k} \end{bmatrix}$$

donde las columnas son agrupadas en bloques de tamaños $k_0, k_1, \dots, k_{k-1}, k_k$, y los k_i son enteros no negativos cuya suma es n, i.e., la matriz generadora es equivalente salvo permutación a la forma estándar.

Recientemente se ha introducido una nueva noción de independencia modular para definir a las bases y las matrices generadoras para los códigos sobre el anillo de enteros módulo m, $\mathbb{Z}_m$, para cualquier entero $m > 1$. También se definen formas estándar para tales matrices generadoras (ver versión en línea [37]).

Por medio de estas matrices generadoras se pueden construir diversos códigos lineales.

El CAS (acrónimo de Computer Algebra System) MAGMA [4] es un importante y bien soportado paquete de software diseñado para resolver problemas computacionalmente duros en álgebra, teoría de números, geometría y combinatoria. Magma es producido y distribuido por el Grupo de Álgebra Computacional que se encuentra en la escuela de Matemáticas y Estadística de la Universidad de Sydney, Australia.

Muchos investigadores en Teoría de Códigos utilizan Magma para construir sus ejemplos y realizar sus cálculos, en particular, usando Magma se pueden construir códigos lineales sobre anillos finitos ($\mathbb{Z}_m$, anillos de Galois $GR(p^n, m)$) pero Magma es software propietario, aún pagando no se distribuye el código fuente[3]. Por tal motivo, nace la necesidad de diseñar software que construya los mismos objetos que contruye Magma pero que puedan funcionar en una plataforma de software libre. El CAS Sage (acrónimo de Software for Algebra and Geometry Experimentation) [44] es una excelente opción para tal fin.

El objetivo de este trabajo es diseñar software (libre) para construir códigos lineales sobre el anillo de enteros módulo m, $\mathbb{Z}_m$, para cualquier entero $m > 1$, basado en *estructuras de datos avanzadas*. Se generarán códigos lineales sobre el anillo finito $\mathbb{Z}_m$ a partir de una "matriz generadora",

[3] Maple distribuye parte de su código fuente en algunas versiones

i.e., se contruirán submódulos de $\mathbb{Z}_m$ generados por el conjunto de filas de una matriz dada, dicha construcción se sustenta en el concepto de submódulo generado por un subconjunto del módulo.

Si X es un subconjunto de un $\mathcal{R}$-módulo M entonces

$$\langle X \rangle = \{\sum_{\text{finito}} r_i x_i : r_i \in \mathcal{R}, x_i \in X\}$$

es el conjunto de todas las $\mathcal{R}$-combinaciones lineales de elementos en X y se dice que $\langle X \rangle$ es el *submódulo generado por* X [40].

Cabe remarcar, que para el proceso de construir códigos lineales se requiere de un mínimo de m^n operaciones, donde m se refiere al módulo del anillo $\mathbb{Z}_m$ y n es el tamaño de las tuplas. Dado que tanto m como n son variables se tiene un problema que requiere de un tratamiento no trivial para resolverlo.

CAPÍTULO 1

NOMENCLATURA

1.1. Notación

La notación empleada, para analizar la complejidad de un algoritmo, en este trabajo es la presentada en [6], que es complementada con la notación $\mathcal{O}$, introducida en [35]. Esta notación se encuentra resumida en la tabla 1.1.

$$
\begin{aligned}
O(f(n)) &= \Big\{g(n)|\exists c \in \mathbb{R}^+, n \geq n_0 \in \mathbb{N}, g(n) \leq cf(n)\Big\} \\
\Omega(f(n)) &= \Big\{g(n)|\exists c \in \mathbb{R}^+, n \geq n_0 \in \mathbb{N}, g(n) \geq cf(n)\Big\} \\
\Theta(f(n)) &= \Big\{g(n)|\exists c, d \in \mathbb{R}^+, n \geq n_0 \in \mathbb{N}, df(n) \leq g(n) \leq cf(n)\Big\}
\end{aligned}
$$

Tabla 1.1: Notación asintótica

Se denota por $\mathcal{O}(f(n))$ una función que se encuentra en $O(f(n))$, pero con constantes que son muy grandes.

1.2. Frameworks

En esta sección se presentan los sistemas utilizados durante la elaboración de este trabajo, así como también algunas razones por las cuales se optó por usarlos.

1.2.1. Magma

Magma, cuyo predecesor es Cayley (1982-1993), fue oficialmente liberado en agosto de 1993 con la versión 1.0. La versión 2.0 de Magma fue liberada en junio de 1996 y subsecuentes versiones de la forma 2.X han sido liberadas aproximadamente cada año. Las áreas que cubre Magma son: Teoría de Grupos con permutaciones, matrices y grupos abelianos (finitos o infinitos), policíclicos, finitamente representados, solubles; Teoría de Números con algoritmos asintóticamente rápidos para todas las operaciones fundamentales sobre enteros y polinomios, como el algoritmo de Schonhage-Strassen para la rápida multiplicación de enteros y polinomios, algoritmos de factorización de enteros que incluyen el método de curvas elípticas, Teoría Algebraica de Números donde incluye el sistem KANT para amplios cálculos en campos de números algebraicos, incluyendo la posibilidad de realizar cálculos en la cerradura algebraica de un campo; Teoría Modular y álgebra Lineal con algoritmos asintóticamente rápidos para todas las operaciones sobre matrices densas, como el algoritmo de multiplicación de Strassen; así como también álgebra Conmutativa, Bases de Grobner, Teoría de Representaciones, Teoría de Invariantes, Teoría de Lie, Geometría Algebraica, Geometría Aritmética, Estructuras de Incidencia Finita, Criptografía, Teoría de Códigos y Optimización. Muchas funciones de Magma están programadas en C, por lo cual, para igualar la velocidad de Magma se tiene que utilizar un lenguaje compilado, ya que funciones programadas en algún lenguaje interpretado difícilmente podrán llegar a ser tan rápidas como las de Magma. Magma contiene muchos de los más avanzados y eficientes algoritmos conocidos para las áreas que cubre.

Magma es software propietario, por lo cual, sólo se puede tener acceso a él pagando una cantidad que depende del tipo de licencia que se desee. La versión de estudiante está limitada a solamente 150 MB y ciertas características avanzadas son omitidas, esta versión tiene un costo de $100.00 dólares; también se puede obtener esta versión por un precio de $50.00 dólares para estudiantes que demuestren que se encuentran en países en vías de desarrollo. Magma no es un sistema comercial, el dinero recaudado de las licencias es para recuperar los costos de su distribución y soporte. Estos costos incluyen la portabilidad de numerosos procesadores diferentes, la preparación de la documentación para el usuario, costos de distribución y soporte general para el usuario. La licencia completa está sujeta a ciertas condiciones, como es el uso en una sola computadora, para lo cual, se solicita la dirección MAC de la computadora donde va a ser ejecutado Magma, y esta licencia tiene una duración de tres años. Durante este periodo, se cuentan con todas las actualizaciones disponibles y soporte general, al final del periodo de suscripción, se puede renovar la suscripción o continuar ejecutando la última versión recibida mientras la suscripción esté activa. El costo de la licencia depende del tipo de procesador. Como un indicativo, una institución de educación que tenga la suscripción durante tres años para una o hasta 3 computadoras tendrá que pagar $1200.00

dólares, $1800.00 dólares por cuatro computadoras, $2400.00 dólares por ocho computadoras y así sucesivamente. Los costos para organizaciones comerciales y gubernamentales son mucho más altos. Así como la versión para estudiantes, la versión completa cuenta con un precio especial para instituciones de educación que se encuentren en países en vías de desarrollo.

Otra desventaja que presenta Magma, es el hecho de no distribuir parte del código para analizar el funcionamiento de ciertos comandos; esto provoca que la comparación entre otros sistemas sea a través de entradas y salidas y el tiempo que tardan cada uno. Si se tiene programado un algoritmo en Sage que se encuentra en $\Theta(n)$y el algoritmo de Magma se encuentra en $\Theta(n\lg n)$, las funciones sólo se podrán comparar con respecto al tiempo que tarden en ser ejecutados. Esto no siempre es lo adecuado, puede haber problemas que en el tiempo de salida en un CAS sea de 15 meses y en otro sea de 10 meses, estos resultados no son muy objetivos ya que la diferencia de tiempo no siempre se debe a la eficiencia del algoritmo, también puede ser que la administración de memoria, del sistema operativo sobre el cual esté corriendo el sistema, no sea la adecuada para ciertos tipos de datos de uno o de otro o el preprocesamiento necesario por alguno de los dos sistemas.

1.2.2. Sage

Sage es un sistema algebraico de computación escrito en Python y en una versión modificada de Pyrex llamada Cython, el nombre es el acrónimo de Software for Algebra and Geometry Experimentation (ver [44]). Este sistema trata de rellenar los huecos de funcionalidad dejados por unos y otros. La primera versión de Sage se liberó el 24 de febrero del 2005, bajo los términos de la Licencia General Pública GNU, con el objetivo inicial de recrear un pequeño subconjunto del sistema algebraico computacional Magma y reducir así la dependencia del software matemático propietario y cerrado. Actualmente el objetivo del proyecto es crear una alternativa de código abierto a Magma, Maple, Mathematica y MatLab. El líder del proyecto, William Stein (ver [45]), es un matemático de la Universidad de Washington y emplea estudiantes becados para el desarrollo del mismo. Sage proporciona una interfaz de Python al siguiente software especializado en matemáticas: GAP, Pari, Maxima y Singular, todos distribuidos con Sage. De esta forma, se hace evidente la filosofía de Sage: "No reinventar la rueda, sino construir el carro". En el 2007, Sage ganó el primer lugar en la división de software científico de *Les Trophées du libre*, competencia internacional para el software libre. Sage consite de un servidor web local que provee una interfaz gráfica para interactuar con una interfaz de Python para programar, esta interfaz es parecida a la interfaz que provee Mathematica. La mayoría de los cálculos son realizados por una distribución incluida de diferente software matemático y librerías de código abierto. Algunas operaciones usan estas librerías automáticamente, otras requieren que el usuario haga llamadas explícitas a ellas. El estado de las variables dentro de

cada una de las librerías es independiente y la transferencia de objetos entre las librerías se lleva a cabo típicamente al convertir a cadenas o viceversa. Sage combina varios modos de uso para distintas aplicaciones, provee una interfaz llamada notebook, una interfaz basada en línea de comandos, incluye MoinMoin como sistema Wiki para la administración de información, es posible insertar Sage en documentos de LaTeX, provee soporte para cómputo distribuido y provee interfaces para software de terceros como Mathematica, Magma y Maple, que permite a los usuarios combinar sistemas y comparar salidas y rendimiento.

Sage también cuenta con un lenguaje compilado para programar funciones que requieran de una velocidad similar a la de funciones programadas en C. Este lenguaje es una versión modificada de Pyrex, que inicialmente se llamó SAgeX y posteriormente Cython. Cython es tan rápido como C y pude interactuar con objetos de Sage. Cython, al ser una versión modificada de Pyrex, no cuenta con muchos manuales para aprender la sintaxis, se recomienda revisar documentación sobre Pyrex y revisar el código de Sage que tiene extensión pyx para apreciar las diferencias entre Pyrex y Cython. Desafortunadamente, Sage no tiene aun implementada la aritmética sobre anillos, ésta es una de las ventajas que tiene Magma sobre Sage. En [45] se presenta una rápida introducción a Sage. Un video introductorio por parte de Jose Unpingco se encuentra en [46]. En [43] se puede conseguir el dvd que incluye Sage así como toda su documentación. A diferencia de Ubuntu, este dvd tiene un costo así como el envío. Sage cuenta con dos formas de trabajo para facilitar su uso, que se describen brevemente a continuación.

Línea de Comandos: La línea de comandos es muy parecida a la de Magma. No se necesita terminar cada instrucción por medio de un punto y coma (;), si solamente hay una instrucción por línea. En el caso de que haya más de una instrucción en una sola línea se deberá usar el punto y coma para separar cada una, mas no se utiliza como parte de la sintaxis para determinar el fin de una instrucción.

Notebook: Esta interfaz es invocada desde la línea de comandos por medio de la instrucción `notebook()`. Inmediatamente, el explorador se abrirá mostrando las hojas de trabajo que hayan sido creadas.

CAPÍTULO 2

DESARROLLO

Los conceptos fundamentales de álgebra moderna pueden ser consultados en distintos libros como [17, 18, 20, 38, 50], los conceptos de teoría de números pueden ser revisados en varios libros [1, 2]. Los conceptos sobre campos finitos y teoría de códigos pueden ser profundizados en [13, 31, 34, 47]. En [16] se presenta un resumen de muchos conceptos que se utilizan en este trabajo.

2.1. Rappel

En esta sección se presenta un resumen de los conceptos utilizados durante el trabajo. Toda la teoría converge en la estructura heap y principalmente en la estructura D-heap ($\beta-$heap). Los temas referentes a árboles y grafos pueden ser profundizados en [7, 22, 42]. El tema de Hashing puede ser consultado en [10, 27, 30].

2.1.1. Tipos de Datos Abstractos Básicos

Durante todo este trabajo, se refiere a los Tipos de Datos Abstractos como ADT's y su definición es tal como se encuentra en [41] y se presenta a continuación:

Definición 2.1. *Un* **tipo de dato abstracto** *es un tipo de dato (un conjunto de valores y una colección de operaciones sobre esos valores) que se puede acceder solamente a través de una* **interfaz**. *Un programa que usa un ADT es referido como un* **cliente** *y un programa que especifica el tipo de dato como una* **implementación**.

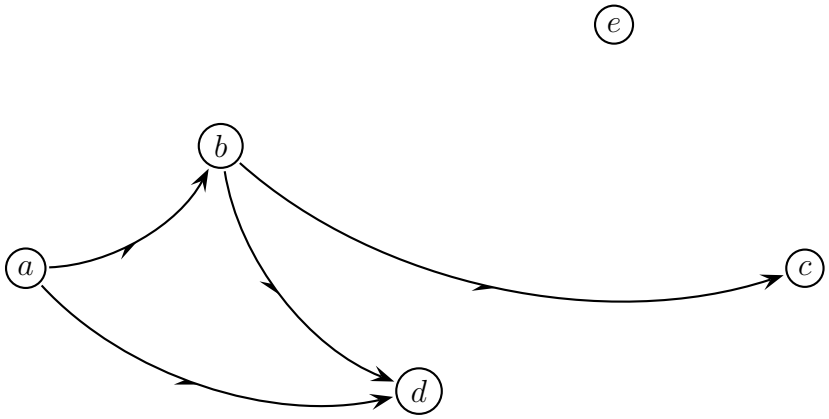

Figura 2.1: Grafo dirigido

Grafos

Definición 2.2. *Sea V un conjunto finito no vacío, y sea $E \subseteq V \times V$. El par $(V, E$ es un* **grafo dirigido** *(sobre V), o digrafo, donde V es el conjunto de vértices o nodos y E es su conjunto de aristas. Se escribe $G = (V, E)$ para denotar tal digrafo.*

Ejemplo 2.1. *En la figura 2.1 se proporciona un ejemplo de un grafo dirigido sobre $V = \{a, b, c, d, e\}$ con $E = \{(a, b), (a, d), (b, c), (b, d)\}$. La dirección de la arista se indica al colocar una flecha dirigida sobre ella. Para cualquier arista, como (b, c), se dice que la arista es incidente con los vértices b y c; b es adyacente hacia c, mientras que c es adyacente desde b. Además el vértice b es el origen, o fuente, de la arista (b, c) y el vértice c es el término o vértice terminal. El vértice e que no tiene aristas incidentes es un vértice aislado.*

Definición 2.3. *Cuando no importa la dirección de las aristas, la estructura $G = (V, E)$, donde E es ahora un conjunto de pares no ordenados sobre V, es un grafo no dirigido.*

Ejemplo 2.2. *La figura 2.2 muestra un grafo no dirigido. Ésta es una forma más compacta de describir el grafo dirigido dado en la figura 2.3. En un grafo no dirigido, hay aristas no dirigidas, como las aristas $\{a, b\}, \{b, c\}, \{a, c\}, \{c, d\}$ de la figura 2.2.*

Se puede escribir (a, a) para designar un *lazo* en un grafo no dirigido. Cuando un grafo no contiene lazos, se dice que es un grafo *sin lazos*. Se pueden asociar aristas distintas con el mismo par de vértices, tales aristas se llaman *paralelas*. Un grafo sin lazos ni aristas paralelas es un **grafo simple**.

Definición 2.4. *Sean x, y vértices (no necesariamente distintos) de un grafo no dirigido $G = (V, E)$. Un camino $x - y$ en G es una sucesión alternada finita (sin lazos)*

$$x = x_0, e_1, x_1, e_2, x_2, e_3, \ldots, e_{n-1}, x_{n-1}, e_n, x_n = y$$

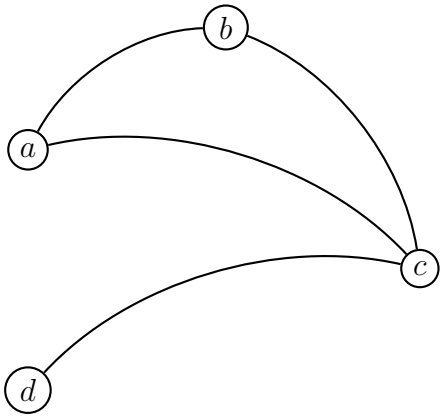

Figura 2.2: Grafo no dirigido

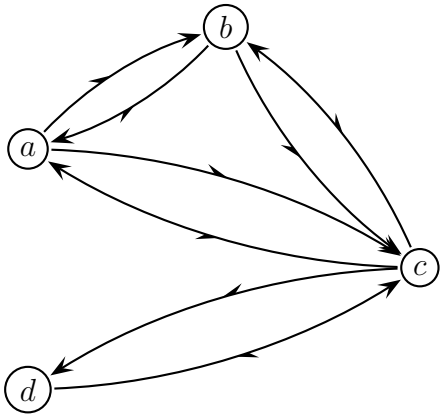

Figura 2.3: Un grafo dirigido de la figura 2.2

de vértices y aristas de G, que comienzan en el vértice x y termina en el vértice y y que contiene las n aristas $e_i = \{x_{i-1}, x_i\}$ donde $1 \leq i \leq n$. La longitud de un camino es n, el número de aristas usadas en el camino. (Si $n = 0$, no existen aristas, $x = y$, y el camino se denomina trivial. Cualquier camino $x-y$ donde $x = y$ (y $n > 1$) es un camino cerrado. En caso contrario, el camino es abierto.

Ejemplo 2.3. *Para el grafo de la figura 2.4, se tienen 3 caminos abiertos. Se pueden enumerar solamente las aristas o solamente los vértices (si el otro queda determinado claramente). Los caminos son los siguientes:*

1. $\{a, b\}, \{b, d\}, \{d, c\}, \{c, e\}, \{e, d\}, \{d, b\}$*: éste es un camino $a - b$ de longitud 6 en el que se repiten los vértices d y b, así como la arista $\{b, d\} = \{d, b\}$*

2. $b \to c \to d \to e \to c \to f$*: aquí se tiene un camino $b - f$ de longitud 5, donde se repite el vértice c, sin que aparezcan las aristas más de una vez.*

3. $\{f, c\}, \{c, e\}, \{e, d\}, \{d, a\}$*: en este caso, el camino $f - a$ tiene longitud 4, sin repetición de vértices o aristas.*

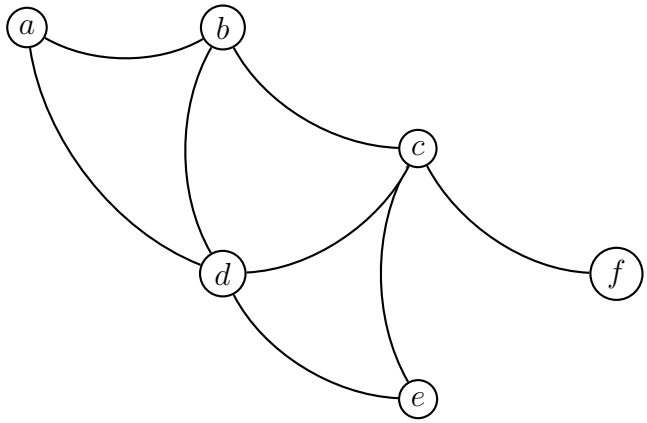

Figura 2.4: Grafo no dirigido con caminos abiertos

Definición 2.5. *Se considera un camino $x - y$ en un grafo no dirigido $G = (V, E)$*

- *Si no se repite ninguna arista en el camino $x - y$, entonces el camino es un recorrido $x - y$. Un recorrido $x - x$ cerrado es un circuito.*
- *Cuando ningún vértice del camino $x - y$ se presenta más de una vez, el camino es un camino simple $x - y$. El término ciclo se usa para describir un camino simple cerrado $x - x$.*

Árboles

Definición 2.6. *Un árbol T es un grafo simple que satisface: Si v y w son vértices en T, entonces existe un único camino simple de v a w.*

Un árbol con raíz tiene las siguientes propiedades:

1. Se distingue un nodo como raíz.
2. A cada nodo c, exceptuando la raíz, le llega una arista desde exactamente otro nodo p, al cual se le llama padre de c. Se dice que c es uno de los hijos de p.

Definición 2.7. *La profundidad de un nodo en un árbol es la longitud del camino que va desde la raíz hasta ese nodo.*

Definición 2.8. *La altura de un nodo es la longitud del camino que va desde el nodo hasta la hoja más profunda bajo él.*

Definición 2.9. *Los nodos que tienen el mismo padre son hermanos.*

Definición 2.10. *Los nodos que no tienen hijos son llamados hojas.*

Definición 2.11. *Un árbol completo es un árbol que debe llenarse de izquierda a derecha que presenta las siguientes propiedades:*

1. *En un árbol completo todas las hojas tienen profundidad N o $N-1$.*
2. *Su altura es a lo sumo $\lfloor log_\beta N \rfloor$, esto es porque un árbol completo de altura N tiene entre β^N y $\beta^{N+1}-1$ nodos.*
3. *Un árbol completo no necesita referencias a ninguno de sus hijos, dado que se puede representar sin ambigüedad almacenando su recorrido por niveles en un arreglo.*

2.1.2. Hashing[1]

Definición 2.12. *Un* **mapa** *es una estructura de datos que permite guardar elementos de tal forma que puedan ser localizados rápidamente usando llaves. Específicamente, un mapa guarda parejas de llave-valor (k, v), que son llamadas entradas, donde k es la llave y v es el valor correspondiente. De igual forma, se requiere que cada llave sea única, así la asociación de llaves con los valores define un mapeo.*

Una de las formas más eficientes de implementar un mapa es el uso de una **tabla hash**.

Hashing está basado en la idea de distribuir llaves en un arreglo $H[0 \ldots m-1]$ llamado una **tabla hash**. La distribución es hecha al calcular para cada una de las llaves, el valor de alguna función predefinida h llamada la **función hash**. Esta función asigna un entero entre 0 y $m-1$, llamado la **dirección hash**, a una llave. Por ejemplo, si las llaves son enteros no negativos, una función hash puede ser de la forma $h(K) = K \bmod m$ (obviamente, el resto de la división por m siempre se encuentra entre 0 y $m-1$). Si las llaves son letras de algún alfabeto, se puede asignar primero su posición en el alfabeto (denotado como *ord(K)*) y luego aplicar el mismo tipo de función usada para enteros. Finalmente, si K es una cadena de caracteres $c_0 c_1 \cdots c_{s-1}$, se puede usar una opción no muy sofisticada, $\left(\sum_{i=0}^{s-1} ord(ci)\right) \bmod m$; una mejor opción es calcular $h(K)$ como sigue:[2]

$h \leftarrow 0$;
para $i \leftarrow 0$ to $s-1$ **hacer**
  $h \leftarrow (h \times C + ord(c_i)) \bmod m$
fin para

[1] Tablas de localización, tablas de asignación, tablas asociativas.

[2] Esto puede ser logrado al tratar *ord(c_i)* como dígitos de un número en C, calcular su valor decimal por la regla de Horner, y encontrar el residuo del número después de dividirlo entre m.

donde C es una constante más grande que cualquier $ord(c_i)$.

En general, una función hash necesita satisfacer dos requisitos que son de alguna forma conflictivos:

- Una función hash necesita distribuir las llaves entre las celdas de una tabla hash tan uniforme como sea posible. A causa de este requisito, el valor de m es usualmente un primo. Este requisito también hace que para la mayoría de las aplicaciones, se desee tener una tabla hash dependiente de todos los bits de una llave, no solamente de algunos de ellos.
- Una función hash tiene que ser fácil de calcular.

Obviamente, si se escoge el tamaño de la tabla m más pequeño que el número de llaves n, se obtendrán **colisiones**, un fenómeno de que dos (o más) llaves, al ser evaluadas por la función hash su valor es el mismo, por lo cual, tienen que ser almacenadas en la misma celda. Pero las colisiones pueden ocurrir, aún cuando m es considerablemente más grande que n. De hecho, en el peor caso, todas las llaves, al ser evaluadas por la función hash, podrían arrojar el mismo valor e intentar ser almacenadas en la misma celda; afortunadamente, con un tamaño de la tabla hash escogido propiamente y una buena función hash, esta situación ocurrirá muy pocas veces. Aun así, todo esquema de hashing debe tener un mecanismo de resolución de colisiones. Este mecanismo es diferente en las dos versiones principales de hashing: **hashing abierto** (también llamado **con encadenamiento**) y **hashing cerrado** (también llamado **direccionamiento abierto**).

Hashing abierto (con encadenamiento)

En esta versión, las llaves son almacenadas en listas ligadas adjuntas a celdas de una tabla hash; cada lista contiene todas las llaves asignadas a esta celda. Para buscar, se aplica el mismo procedimiento que fue usado para crear la tabla. En general, la eficiencia de la búsqueda depende de calidad de la función hash así como de la longitud de las listas ligadas, que resulta igual al tamaño de la tabla. Si la función hash distribuye n llaves entre m celdas de la tabla hash de manera uniforme, cada lista tendrá n/m llaves de longitud. La razón $\alpha = n/m$, llamada **factor de carga** de la tabla hash, juega un papel crucial en la eficiencia de la asignación. En particular, el número promedio de punteros (a la siguiente lista) inspeccionados en una búsqueda exitosa, S, y en una búsqueda sin éxito, U, resulta ser $S \approx 1 + \frac{\alpha}{2}$ y $U = \alpha$, respectivamente. Estos resultados son muy naturales; de hecho, son casi idénticos a buscar secuencialmente en una lista ligada; lo que se ahorra por la asignación es la reducción en el tamaño promedio de las lista por un factor de m, el tamaño de la tabla hash.

Normalmente, se desea que el factor de carga no esté muy lejos de 1. Siendo muy pequeño implicaría muchas listas vacías y así un uso ineficiente de espacio; siendo muy grande significaría que hay listas ligadas más largas, por lo cual es mayor el tiempo de búsqueda. Pero si el factor de carga se encuentra alrededor de 1, se tiene un sorprendente esquema eficiente que hace posible que la busqueda de una llave dada, en promedio, sea del precio de una o dos comparasiones. Además de las comparasiones, es necesario gastar tiempo calculando el valor de la función hash de la llave a buscar, pero es una operación que se encuentra en tiempo constante, independientemente de n y m. Cabe hacer énfasis en que se obtiene esta remarcable eficiencia no solamente como un resultado de un método ingenuo, sino también por el uso de espacio extra.

Hashing cerrado (direccionamiento abierto)

En esta versión, todas las llaves son almacenadas en la misma tabla hash sin el uso de listas ligadas. Claramente, ésto implica que el tamaño m de la tabla debe ser al menos tan grande como el número de llaves n. Distintas estrategias se pueden aplicar para la resolución de colisiones. La más sencilla, llamada **prueba lineal**, verifica la celda siguiente de donde ocurrió la colisión, si esa celda está vacía, la nueva llave se almacena ahí; si la siguiente celda ya se encuentra ocupada, se verifica la disponibilidad del sucesor inmediato a esa celda y así sucesivamente. Cabe notar que si se alcanza el final de la tabla hash, la búsqueda continua en el inicio de la tabla; esto es, se trata como un arreglo circular.

Para buscar una llave dada k, se comienza calculando $h(K)$ donde h es la función hash usada en la construcción de la tabla. Si la celda $h(K)$ está vacía, la búsqueda no tiene éxito, si la celda no está vacía, se debe comparar K con el ocupante de la celda, si son iguales, la búsqueda es exitosa; si no, se compara K con la llave en la siguiente celda y se continua de esta manera hasta encontrar una llave que coincida (una búsqueda exitosa) o una celda vacía (búsqueda sin éxito).

Mientras que las operaciones de inserción y búsqueda son directas para esta versión de hashing, la eliminación no lo es. Puede darse el caso, que el borrado de un elemento evite que otro sea encontrado. Una solución a este probleme es usando un **borrado flojo**, esto es, marcar las celdas previamente ocupadas por un símbolo especial para distinguirlas de celdas que no han sido ocupadas. El número promedio de veces que el algoritmo debe acceder a una tabla hash con un factor de carga α en una búsqueda exitosa es $S \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ y sin éxito es $U \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$. La exactitud de estas aproximaciones aumenta con tablas hash más grandes. Estos valores son sorprendemente pequeños inclusive para tablas densamente pobladas.

Aun así, mientras la tabla hash se va llenando, el desempeño de la prueba lineal se deteriora a causa

de un fenómeno llamado clustering[3]. Un **cluster** en la prueba lineal es una secuencia de celdas ocupadas contiguamente. Los clusters son malas noticias en hashing porque hacen que las operaciones sean menos eficientes. También hay que notar, que conforme los clusters se vuelven más grandes, la probabilidad de que un nuevo elemento se agregue a un cluster incremente. Además, clusters grandes aumentan la probabilidad de que dos clusters se unan después de una nueva inserción, causando aun más clustering.

Hashing universal

Esta sección y las secciones 2.1.2 y 2.1.2 son requeridas para entender el artículo presentado en el apéndice, mas no necesarias para entender el trabajo en si.

Si un adversario malicioso escoge un conjunto de llaves para alguna función hash fija, entonces puede escoger n llaves cuya evaluación de la función hash calcule la misma celda, produciendo un tiempo de recuperación de $\Theta(n)$. Cualquier función hash fija es vulnerable a este comportamiento del peor caso; la única forma efectiva de mejorar la situación es escoger la función hash *aleatoriamente* de una forma que sea *independiente* de las llaves que van a ser almacenadas en ese momento. Este enfoque, llamado **hashing universal**, puede producir probablemente un buen desempeño en el caso promedio, sin importar cuales sean las llaves escogidas por el adversario.

La idea principal detrás del hashing universal es seleccionar la función hash de forma aleatoria a partir de una clase de funciones diseñada cuidadosamente, al inicio de la ejecución. La aleatoriedad garantiza que ninguna entrada siempre evocará el comportamiento del peor caso. A causa de la aleatoriedad, el algoritmo puede comportarse de forma diferente en cada ejecución, inclusive para la misma entrada, garantizando un desempeño en el caso promedio para cualquier entrada. Un desempeño pobre ocurre solamente cuando se escoge una función hash aleatoria que causa que el conjunto de llaves sean asignadas deficientemente, pero la probabilidad de que esta situación ocurra es pequeña y es la misma para cualquier conjunto de llaves del mismo tamaño.

Sea $\mathbb{H}$ una colección finita de funciones hash que mapean un universo dado U de llaves en el rango $\{0, 1, \ldots, m-1\}$. Tal colección se dice que es **universal** si para cada par de llaves distintas $k, l \in U$, el número de funciones hash $h \in \mathbb{H}$ para las cuales $h(k) = h(l)$ es a lo más $|\mathbb{H}|/m$. En otras palabras, con una función hash escogida aleatoriamente de $\mathbb{H}$, la probabilidad de una colisión entre distintas llaves k y l no es mayor que la probabilidad $1/m$ de una colisión si $h(k) = h(l)$ fueran escogidas de manera aleatoria e independiente del conjunto $\{0, 1, \ldots, m-1\}$.

[3] agrupación

Diseño de una clase universal de funciones hash

Es fácil diseñar una clase universal de funcioens hash. Se comienza eligiendo un número primo p lo suficientemente grande que para cada posible llave k se encuentre en el rango de 0 a $p-1$. Sea $\mathbb{Z}_p$ el conjunto $\{0, 1, \ldots, p-1\}$, y sea $\mathbb{Z}_p^*$ el conjunto $\{1, 2, \ldots, p-1\}$. Dado que p es primo, se pueden resolver ecuaciones módulo p. Como se asume que el tamaño del universo de llaves es más grande que el número de celdas en la tabla hash, se tiene que $p > m$.

A continuación se define una función hash $h_{a,b}$ para cualquier $a \in \mathbb{Z}_p^*$ y cualquier $b \in \mathbb{Z}_p$ usando una función lineal seguida de reducciones módulo p y luego módulo m:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m \tag{2.1}$$

Por ejemplo, con $p = 17$ y $m = 6$, se tiene $h_{3,4}(8) = 5$. La familia de todas las funciones de este tipo de funciones hash es

$$\mathbb{H}_{p,m} = \left\{h_{a,b} : a \in \mathbb{Z}_p^* \text{ y } b \in \mathbb{Z}_p\right\} \tag{2.2}$$

Cada función hash $h_{a,b}$ mapea $\mathbb{Z}_p$ a $\mathbb{Z}_m$. Esta clase de funciones tiene la propiedad de que el tamaño $m$del rango de salida es arbitrario (no necesariamente primo). Dado que hay $p-1$ valores para a y hay p posibles valores para b, entonces existen $p(p-1)$ funciones hash en $\mathbb{H}_{p,m}$.

Teorema 2.1. *La clase $\mathbb{H}_{p,m}$ de funciones hash definida por las ecuaciones 2.1 y 2.2 es universal.*

Demostración. Se consideran dos llaves distintas k y l de $\mathbb{Z}_p$, de tal forma que $k \neq l$. Para una función hash dada $h_{a,b}$:

$$\begin{aligned} r &= (ak + b) \bmod p, \\ s &= (al + b) \bmod p \end{aligned}$$

Se nota que $r \neq s$ dado que

$$r - s \equiv a(k - l) \bmod p$$

Si $r = s$, entonces $a(k-l) = 0$, lo que solamente ocurre si $a = 0$ o $k - l = 0$, pero $a \in \mathbb{Z}_p^*$ por lo cual, $a \neq 0$; como k y l fueron tomadas de tal forma que $k \neq l$, entonces $k - l \neq 0$. Por lo tanto, durante el cálculo de cualquier $h_{a,b}$ en $\mathbb{H}_{p,m}$, distintas entradas k y l mapean a distintos valores r y s módulo p; no hay colisiones aun cuando se realiza el módulo p. Más aun, cada valor posible $p(p-1)$ para la pareja (a, b) con $a \neq 0$ produce una pareja *(r,s)* con $r \neq s$, de la cual, se puede

resolver para a y b teniendo r y s.
A partir de

$$\begin{aligned} r - s &= a(k-l) \bmod p \\ (r-s)(k-l)^{-1} \bmod p &= a \bmod p \end{aligned} \qquad \begin{aligned} r &= (ak+b) \bmod p \\ r &= ak \bmod p + b \bmod p \\ (r-ak) \bmod p &= b \bmod p \end{aligned}$$

se obtiene

$$\begin{aligned} a &= ((r-s)((k-l)^{-1} \bmod p)) \bmod p \\ b &= (r-ak) \bmod p \end{aligned}$$

Como solamente existen $p(p-1)$ posibles parejas (r,s) con $r \neq s$, existe una correspondencia uno a uno entre las parejas (a,b) y (r,s). Así, para cualquier pareja de entrada k y l, si se toma (a,b) uniformemente de manera aleatoria de $\mathbb{Z}_p^* \times \mathbb{Z}_p$, la pareja resultante (r,s), es igualmente probable, que sea cualquier pareja de valores distintos módulo p.

En seguida, se tiene que la probabilidad de que llaves distintas k y l colisionen es igual a la probabilidad de que $r \equiv s \bmod m$ cuando r y s son tomados aleatoriamente como valores distintos módulo p. Para un valor dado de r, de los posibles $p-1$ valores restantes para s, el número de valores s tales que $s \neq r$ y $s \equiv r \bmod m$ es a lo más:

$$\begin{aligned} \lceil p/m \rceil - 1 &\leq ((p+m-1)/m) - 1 \\ &= (p-1)/m \end{aligned}$$

La probabilidad de que s colisione con r cuando se hace la reducción módulo m es a lo más

$$\frac{(p-1)/m}{p-1} = \frac{1}{m}$$

Por lo tanto, para cualquier pareja de valores distintos $k, l \in \mathbb{Z}_p$,

$$P\left(h_{a,b}(k) = h_{a,b}(l)\right) \leq \frac{1}{m}$$

así que $\mathbb{H}_{p,m}$ es universal. □

Hashing perfecto

Aunque la mayoría de las veces, el hashing tiene un excelente desempeño, se puede obtener un excelente desempeño en el peor de los casos cuando el conjunto de llaves es **estático**: una vez que las llaves han sido almacenadas en la tabla, el conjunto de llaves nunca cambia. Algunas aplicaciones naturalmente tienen conjuntos estáticos de llaves; por ejemplo, al considerar el conjunto de palabras reservadas en un lenguaje de programación o el conjunto de nombres de archivos en un CD-ROM. Se le llama técnica de **hashing perfecto** si el número de accesos a memoria en el peor caso requiere desempeñar una búsqueda en $O(1)$.

La idea básica de crear un esquema de hashing perfecto es simple. Se usa un esquema de dos niveles de hashing con hashing universal para cada nivel.

El primer nivel es esencialmente el mismo que el hashing con encadenamiento: las n llaves son asignadas a m celdas usando una función hash h cuidadosamente seleccionada a partir de una familia de funciones de hash universal.

En lugar de hacer una lista de llaves que son asignadas a la celda j, se usa una pequeña **tabla hash secundaria** S_j con una función hash asociada h_j. Al escoger las funciones hash h_j cuidadosamente, se puede garantizar que no existan colisiones en el segundo nivel.

Para garantizar que no existan colisiones en el segundo nivel, se necesita que el tamaño m_j de la tabla S_j sea el cuadrado del número n_j de llaves que son asignadas a j. Dado que se tiene una dependencia cuadrática de m_j sobre n_j, puede parecer que los requerimientos de almacenamiento total sean excesivos, pero al escoger las funciones hash del primer nivel adecuadamente, la cantidad total esperada de espacio utilizado aun se encuentra en $O(n)$.

Utilizando las funciones hash escogidas de las clases universales de funciones hash de la sección 2.1.2, la función hash del primer nivel se toma de la clase $\mathbb{H}_{p,m}$, donde p es un número primo más grande que el valor de cualquier llave. Las llaves que son asignadas a la celda j son reasignadas a una segunda tabla hash S_j de tamaño m_j usando una función hash h_j tomada de la clase $\mathbb{H}_{p,m_j}$[4].

[4]Cuando $n_j = m_j = 1$, no se necesita realmente una función hash para la celda j; cuando se escoge una función hash $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m_j$ para tal celda, se usa $a = b = 0$

2.1.3. Heaps[5]

La estructura de datos llamada "heap", definitivamente no es una pila desordenada de objetos, como la definición de la palabra en un diccionario podría sugerir. Más bien se trata de una estructura de datos inteligente y parcialmente ordenada que es adecuada para la implementación de colas de prioridad. Una **cola de prioridad** es un multiconjunto de objetos con una característica ordenable llamada **prioridad** del objeto, con las siguientes operaciones:

- Encontrar un objeto con la más alta (i.e., más grande) prioridad.
- Eliminar un objeto con la más alta prioridad.
- Agregar un nuevo elemento al multiconjunto.

Es principalmente una aplicación eficaz de estas operaciones que hace que la estructura heap sea de gran interes y utilidad.

Noción de Heap

Definición 2.13. *El* **heap** *pued ser definido como un árbol binario con llaves asignadas a sus nodos (una llave por nodo) asegurando que las dos condiciones siguientes se cumplan:*

1. *El requisito de la forma del árbol: el árbol binario es* **esencialmente completo** *(o simplemente* **completo***), esto es, todos su niveles están llenos excepto posiblemente el último nivel, donde solamente las hojas de la derecha no se encuentran.*
2. *El requisito de dominancia parental: la llave en cada nodo es mayor o igual que las llaves de sus hijos. (Esta condición se considera automáticamente satisfecha para todas las hojas.)*[6]

Ejemplo 2.4. *Al considerar los árboles de la figura 2.5, el primer árbol es un heap. El segundo no lo es porque el requisto de la forma no es cumplida. El tercero no lo es porque el requisito de dominancia parental falla por el nodo con llave 5.*

Los valores de las llaves en un heap están ordenados de arriba hacia abajo; esto es, una secuencia de valores en cualquier trayectoria de la raíz a una hoja es decreciente (no decreciente, si llaves iguales son permitidas). De cualquier forma, no hay ningún orden de izquierda a derecha en los

[5] Montículos

[6] Algunos autores requieren que la llave en cada nodo sea menor o igual a las llaves de sus hijos. Esta variación se conoce como **min-heap**.

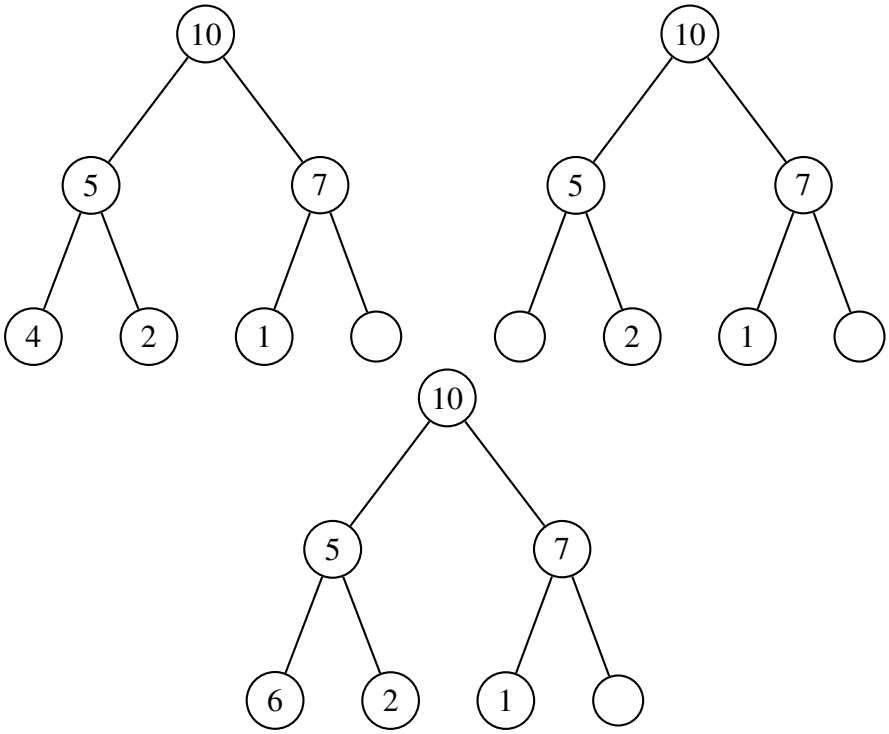

Figura 2.5: Ilustración de la definición de *heap*: solamente el árbol que se encuentra a la izquiera es un heap

valores de las llaves; esto es, no existe ninguna relación entre los valores de las llaves para los nodos en el mismo nivel del árbol o, más generalmente, en los subárboles izquierdos y derechos del mismo nodo.
A continuación se presenta una lista de propiedades importantes de los heaps, que no son difíciles de verificar, como se puede comprobar en el primer árbol de la figura 2.5, cuya representación se presenta en la tabla 2.1.

1. Existe exactamente un árbol binario esencialmente completo con n nodos. Su altura es igual a $\lfloor \lg n \rfloor$.

2. La raíz siempre contiene su elemento más grande.

3. Un nodo considerado con todos sus descendientes también es un heap.

4. Un heap puede ser implementado como un arreglo al almacenar sus elementos de arriba hacia abajo y de izquierda a derecha. Es conveniente almacenar los elementos en las posiciones 1 a n de tal arreglo, dejando $H[0]$ sin usar o con un sentinela cuyo valor es mayor que cualquier elemento en el heap. En tal representación:

   - el nodo padre de las llaves se encuentra en las primeras $\lfloor n/2 \rfloor$ posiciones del arreglo, mientras que las llaves hojas ocupan las últimas $\lceil n/2 \rceil$ posiciones.

- los hijos de una llave en la posición padre del arreglo i ($1 \leq i \leq \lfloor n/2 \rfloor$) se encuentran en las posiciones $2i$ y $2i+1$ y correspondientemente, el padre de una llave en la posición i ($2 \leq i \leq n$) en la posición $\lfloor i/2 \rfloor$.

0	1	2	3	4	5	6
	10	5	7	-4	2	1

Tabla 2.1: Representación del primer árbol de la figura 2.5

Así, también se puede definir una estructura heap como un arreglo $H[1, \ldots n]$ en el cual todo elemento en la posición i en la primera mitad del arreglo es mayor o igual que los elementos en las posiciones $2i$ y $2i + 1$, i.e.,

$$H[i] \geq \max \{H[2i], H[2i + 1]\} \qquad \text{para } i = 1, \ldots, \lfloor n/2 \rfloor$$

Por supuesto, si $2i + 1 > n$, sólo $H[i] \geq H[2i]$ necesita satisfacerse. Mientras que las ideas dentrás de la mayoría de algoritmos que tratan con heaps son más fáciles de entender si se ven los heaps como árboles binarios, sus implementaciones son usualmente mucho más simples y más eficientes con arreglos.

Existen dos alternativas principales para la construcción de heaps a partir de una lista dada de llaves. La primera, es el algoritmo **de construcción de abajo hacia arriba**, inicializa el árbol binario esencialmente completo con n nodos al colocar las llaves en el orden dado y luego *amontona* el árbol como sigue. Comienza con el último nodo padre, el algoritmo verifica si la dominancia parental se cumple para la llave en este nodo; si no, el algoritmo intercambia la llave K del nodo con una llave más grande de sus hijos y verifica si la dominancia parental se cumple para k en su nueva posición. Este proceso continua hasta que el requisito de dominancia parental para K se satisfaga. Eventualmente, este proceso tiene que terminar porque se cumple automáticamente para cualquier llave en una hoja. Después de completar el proceso del subárbol cuya raíz es el padre nodo actual, el algoritmo procede a hacer lo mismo con el predecesor inmediato del nodo. El algoritmo se detiene después de que se haya hecho este proceso para la raíz del árbol.

Dado que el valor de la llave de un nodo no cambia durante el proceso de selección que establece el árbol, no hay necesidad de involucrar intercambios durante el mismo.
Se puede ver este mejoramiento como intercambiar el nodo vacío con llaves más grandes en sus hijos hasta que una posición final es alcanzada donde acepta el valor *borrado* otra vez.

Algoritmo: HeapBottomUp($H[1,\ldots,n]$)
  para $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **hacer**
    $k \leftarrow i$
    $v \leftarrow H[k]$
    $heap \leftarrow$ **falso**
    mientras no $heap$ **y** $2 \times k \leq n$ **hacer**
      $j \leftarrow 2 \times k$
      si $j < n$ **entonces**
        si $H[j] < H[j+1]$ **entonces**
          $j \leftarrow j + 1$
        fin si
      fin si
      si $v \geq H[j]$ **entonces**
        $heap \leftarrow$ **verdadero**
      si no
        $H[k] \leftarrow H[j]$
        $k \leftarrow j$
      fin si
    fin mientras
    $H[k] \leftarrow v$
  fin para

2.1.4. Colas de prioridad con actualización y búsqueda de árboles mínimos[7]

Introducción

Una cola de prioridad aparece en cualquier secuencia δ de ciertas operaciones que pueden ser ejecutadas en tiempo lineal en $|\delta|$ (donde $|\delta|$ es la longitud de δ) siempre que δ contenga un número suficiente de actualizaciones (cambios prioritarios). El mejor resultado posible sin actualizaciones es $O(|\delta|\log|\delta|)$ y esta cota se logra cuando la actualización se realiza sobre colas de prioridad convencionales. El resultado es aplicado al problema del árbol de recubrimiento mínimo, dando un algoritmo $O(e)$ para grafos densos (grafos en donde $e = \Omega(n^{1+\epsilon})$) para n el número de vértices, e el número de aristas y alguna constante positiva ϵ).

Colas de Prioridad con Actualización

Una *cola de prioridad* es una estructura de datos para un conjunto finito de elementos junto con algoritmos para desempeñar ciertas operaciones sobre un conjunto. Cada elemento tiene una etiqueta dibujada, posiblemente con repetición, de un conjunto universo con un orden lineal. Los elementos en la cola de prioridad se encuentran esperando un servicio que provee un solo servidor al cual, ellos deben suministrar en un orden determinado por las etiquetas. Un elemento atendido es un elemento con la menor etiqueta y es borrado de la cola de prioridad cuando termina de ser atendido. Puede ser que las etiquetas sean asignadas a elementos en el orden de su llegada a la cola de prioridad. En este caso, la cola de prioridad se reduce a una cola ordinaria con la metodología

[7]Esta sección presenta la traducción de [21]

FIFO[8]; o puede ser que ningún elemento sea atendido hasta que todos los elementos que vayan a ser atendido hayan llegado. En este caso, el problema de la cola de prioridad se vuelve un problema de ordenamiento. El caso más general, es cuando las etiquetas son independientes del orden de llegada y se permiten secuencias en que las llegadas y los servicios son intercalados, ésto ocurre en muchos contextos. En un sistema computarizado, por ejemplo, el orden en que los trabajos son procesados puede ser determinado por prioridades suministradas externamente independientes del orden de llegada o por un algoritmo de calendarización que puede determinar prioridades por el tiempo de proceso esperado o algún otro criterio.
Un objetivo primordial en el diseño de colas de prioridad es el optimizar el tiempo de ejecución sobre algún conjunto de secuencias anticipadas de operaciones, llegadas y servicios. En este caso, se considera un caso más general en el cual, las etiquetas pueden cambiar después de que los elementos hayan llegado. Los elementos de los que una cola de prioridad se encarga son conocidos por *nombre*. Para cada elemento en la cola de prioridad existe una *ubicación*, esto es, un puntero en la estructura de datos de la cola. Si un elemento no se encuentra en la cola de prioridad su ubicación es indefinida.
Se trabajan con colas de prioridad con cuatro operaciones: INSERTAR, BORRAR, MIN, y ACTUALIZAR. La operación INSERTAR(x, b) asocia la etiqueta b con el elemento llamado x y agrega x a la cola de prioridad. BORRAR(x) remueve el elemento x de la cola de prioridad. MIN regresa el nombre de un elemento en la cola de prioridad con la menor etiqueta y lo remueve de la misma. ACTUALIZAR(x, b) cambia la etiqueta de x a b dado que la antigua etiqueta de x no era menor que b. Si BORRAR o ACTUALIZAR se ejecuta para un elemento que no se encuentra en la cola de prioridad, ésta no tiene ningún efecto. Si MIN se ejecuta sobre una cola de prioridad vacía, se regresa el nombre del elemento nulo. Una cola de prioridad (ordinaria) acepta la secuencias de las operaciones INSERTAR, BORRAR y MIN. Si la operación ACTUALIZAR es también permitida, se tiene una *cola de prioridad con actualización*. Tales colas tienen aplicaciones, por ejemplo, en la programación de trabajos de acuerdo a la prioridad donde las prioridades pueden mejorar después de la llegada del trabajo. Las colas de prioridad con actualización también aparecen en ciertos algoritmos tales como para encontrar los caminos más cortos y para árboles de recubrimiento mínimo.

Complejidad de las secuencias de operaciones sobre colas

Sea δ una secuencia finita, de operaciones definidas como arriba, en la cuales hay n operaciones INSERTAR, p operaciones ACTUALIZAR y m operaciones MIN y BORRAR juntas. Por lo cual, la longitud de δ, denotada $|\delta|$, es $n + p + m$. Se evalua la complejidad de δ de acuerdo al modelo de

[8]First-In–First-Out

máquina RASP, en la cual las operaciones aritméticas, de comparación y asignación de enteros son todas de costo unitario. Para las estructuras de datos, se considera que puede ser fácilmente vista que las siguientes operaciones pueden ser implementadas para ser ejecutadas en tiempo constante: para x, un elemento `name` y l una `dirección` en la estructura de datos de la cola, `loc(x)` da la dirección de x en la cola o la dirección indefinida si x no se encuentra presente, `name(l)` da el nombre del ememento asociado con la dirección l o el nombre del elemento nulo si la dirección está vacía, y `label(x)` da la etiqueta de x si x se encuentra en la cola o una etiqueta indefinida en caso contrario. La forma en que m y p se anticipan a que dependan de n, determina la selección de la cola de prioridad a ser usada para procesar δ. Los casos $m = O(1)$ y $m = O(n)$ con p insignificante han sido muy bien tratados en distintos trabajos. Si $m = O(1)$ entonces una lista no ordenada es suficiente. INSERTAR y ACTUALIZAR son de tiempo constante y MIN es de costo $O(n)$, produciendo un costo para δ de $O(|\delta|)$. Si $m = O(n)$ y p es insignificante, es claro que un árbol binario con ordenamiento heap (en el cual las etiquetas son no decrecientes sobre cualquier camino a partir de la raíz) es óptimo en el caso peor. Con algoritmos habituales, todas las operaciones son de costo $O(\log n)$ dando un costo para δ de $O(n\log n)$, el cual es óptimo dado que un tiempo $O(n\log n)$ es insuficiente para ordenar en este modelo. (Todos los logaritmos están en base 2, al menos que se indique lo contrario). Por supuesto, $m \leq n$ debe cumplirse para cualquier δ. En lo que sigue, $m = O(n)$ y se considera el efecto de p sobre la elección de una cola de prioridad. Primero que nada, es fácil extender el resultado usual a $p = O(n)$. Dado que ACTUALIZAR puede ser implementada con un costo de $O(\log n)$ sobre el árbol binario heap ordenado, el costo de $O(n)$ ACTUALIZACIONES puede ser absorbido por el costo de $O(n\log n)$ para δ. También es fácil ver que, si $p = \Omega(n^2)$, una lista lineal es suficiente para dar un costo óptimo para δ en el caso peor. (Se denota por $\Omega(f(n))$ cualquier función que es mayor que $c \cdot f(n)$ para algún c positivo y para todo n muy grande pero finito.) Se considera que p satisface $p = \Omega(n)$ y $p = O(n^2)$ que por brevedad se escribe como $\Omega(n) = p = O(n^2)$. Si se permite que el *nivel* de un vértice en un árbol sea su distancia a partir de la raíz, entonces un $\beta - heap$ es un árbol con vértices heap ordenados en sus etiquetas en las cuales todos los vértices excepto posiblemente uno, tienen 0 o β hijos y aquellos con un número distinto de β se encuentran en los dos niveles más grandes del árbol. Un 2-heap es un árbol binario completo. En la aplicación a colas de prioridad los vértices son las direcciones de los elementos de la cola. Por conveniencia, se omitirán consideraciones de nombres de elementos, asociando la etiqueta de un elemento de la cola x directamente con una dirección, teniendo que $label(l)$ equivale a $label(x)$ donde $l = loc(x)$. Bajo esta simplificación MIN regresará una etiqueta. Será obvio cómo encontrar las direcciones actuales de etiquetas cuando etiquetas sean desplazadas de una dirección a otra. También, el heap puede ser construido de tal forma que una dirección libre del menor nivel y una dirección ocupada del mayor nivel siempre son conocidas. Se omiten estos detalles comunes. Si bien es posible aplazar el restablecimiento del orden del heap hasta que

es necesario, esto es, cuando MIN o BORRAR son llevadas a cabo, los algoritmos usualmente empleados que restauran el orden siguiente a cada operación sobre la cola son suficientes para los propósitos aquí presentados. En la cola de prioridad, el tamaño del heap está acotado por n. Así, en el peor caso, las operaciones heap cuestan: El costo de una secuencia δ (para $m = O(n)$) es

INSERTAR	$O(\log_\beta n)$
BORRAR	$O(\beta \log_\beta n)$
MIN	$O(\beta \log_\beta n)$
ACTUALIZAR	$O(\log_\beta n)$

por lo tanto $O((n\beta + p)\log_\beta n)$. En el análisis convencional de heaps, donde β es una pequeña constante que satisface $\beta \geq 2$, la cota es $O(|\delta| \log n)$ y así es lineal en $|\delta|$, y $\Omega(n) = p = O(n^2)$. En vez de fijar β, se fija la profundidad del heap permitiendo que la profundidad de la dirección de la mayor profundidad sea menor o igual que k para algún entero fijo $k \geq 1$. A continuación se toma $\beta = \lceil n^{1/k} \rceil$, dando un costo para δ de $O(n^{1+1/k} + p)$. En cada término el factor constante k desaparece de la notación O. Este análisis da resultados que son óptimos con un factor constante. Para cualquier positivo ϵ, cualquier δ produce una cola de prioridad que no crece más de $O(p^{1-\epsilon})$ puede ser procesado en tiempo $O(|\delta|)$. Cuando n es conocida desde el principio, β también es conocida y un heap de profundidad fija puede ser almacenado en un arreglo lineal con la raíz en el subíndice 0. El subíndice del padre del elemento en i está dado por $\lfloor (i-1)/\beta \rfloor$. En algunos casos donde n no es conocida desde el principio, la definición del $\beta-$heap puede ser variada y el heap es creado al principio con un valor inicial de $\beta = 2$. Cuando el heap ha alcanado su profundidad permitida, los vértices son agregados en el heap en algún transversal sistemático del heap hasta que el heap haya alcanzado su profundidad con $\beta = 3$, y así sucesivamente. Puede ser mucho más sencillo en este caso, guardar el heap con apuntadores explícitos.

Árboles de recubrimiento mínimo

Existen dos algoritmos comunmente utilizados para encontrar árboles de recubrimiento mínimo, uno por Kruskal de complejidad $O(e \log n)$ y uno descubierto independientemente por Prim y Dijkstra de complejidad $O(n^2)$ donde el grafo dado tiene n vértices y e aristas. Usando una cola de prioridad heap ordenada con reordenamiento diferido, Kerschenbaum y Van Slyke reducen el tiempo de ejecución del algoritmo de Prim a $O(e\text{n})$. También existe un algoritmo $O(e \log \log n)$ por Yao. Ninguno de estos resultados es conocido como óptimo excepto sobre grafos completos. En la versión del algoritmo de Prim (ver figura 2.6), se asume que los vértices V son representados por enteros de 1 a n y que las aristas del grafo son especificadas por n listas de adyacencia: para $i = 1, 2, \ldots n$, la lista A_i contiene el par ordenado (i, j) si y sólo si (i, j) es una arista no dirigida del grafo. Asociado con acada arista (i, j) en la lista de adyacencia A_i está el peso de la arista,

$d(i, j)$. El árbol de recubrimiento es construido en la lista T. El conjunto S es un arreglo boleano almacenando los vértices del árbol T, U es una cola de prioridad con aristas (respresentadas por parejas ordenadas) a partir del cual la siguiente arista del árbol será dibujada y B es un arreglo representando U independientemente de la cola.

Figura 2.6: Algoritmo de Prim

$T := \emptyset$
Sea i algún vértice en V
$S := \{i\}$
$U := A_i$
Inicializar B
para $i \leftarrow 1\text{to}N$ **hacer**
  si $(i, j) \in U$ **entonces**
    $B[j] = i$
  si no
    0
  fin si
fin para
mientras $U \neq \emptyset$ **hacer**
  Sea (i, j) que satisface $d(i, j) = \min_{(k,l)\in U} d(k, l)$
  $U := U - \{(i, j)\}$
  para $(j, k) \in A_j$ tal que $k \in V - S$ **hacer**
    si $B[k] = 0$ **entonces**
      $U := U \bigcup \{(j, k)\}$
      $B[k] := j$
    si no
      si $d(j, k) < d(B[k], k)$ **entonces**
        Remplazar $(B[k], k)$ con (j, k) en U

        $B[k := j]$
      fin si
    fin si
    $T := T \bigcup \{(i, j)\}$
    $S := S \bigcup \{j\}$
  fin para
fin mientras

2.2. De palabras código a D-heap

En esta sección se presenta la idea fundamental de todo el trabajo. Se recomienda leer la sección 2.1.4 para entender la generalización de la estructua heap.

Dado el código de la figura 2.7, se pueden obtener árboles como los de la figura 2.8, que permiten representar el código utilizando menos espacio en memoria.

$$\left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\}$$

Figura 2.7: Código binario

De los árboles de la figura 2.8 se pueden generar los árboles de las figuras 2.9 y 2.10, agregando los *sentinelas* -1 para tener árboles completos. Estos sentinelas son agregados en los nodos y hojas del árbol según sea el caso.

Los árboles de las figuras 2.9 y 2.10 se pueden representar con los heaps de la figura 2.11.

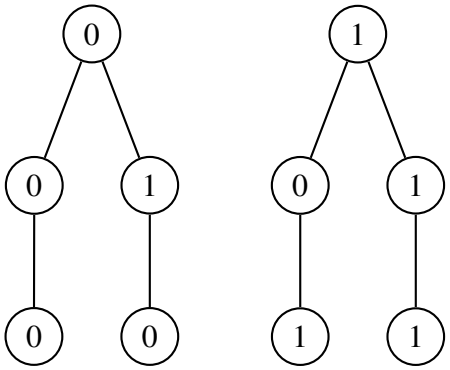

Figura 2.8: Árboles a partir del código de la figura 2.7

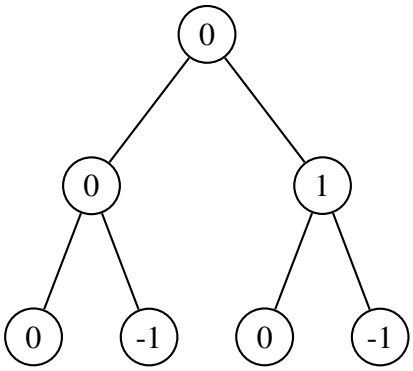

Figura 2.9: Árbol binario cuya raíz es 0

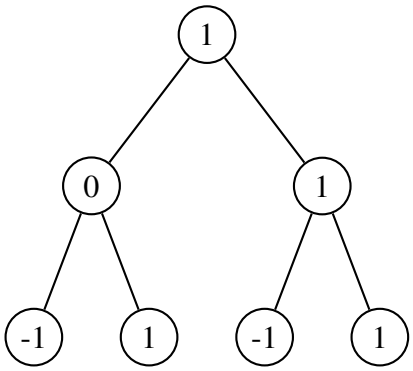

Figura 2.10: Árbol binario cuya raíz es 1

Si se tiene el código de la figura 2.12.

Cuyos árboles se representan en las figuras 2.13, 2.14 y 2.15, puede ser representado en el heap mostrado en la figura 2.16

0	0	1	0	-1	0	-1
1	0	1	-1	1	-1	1

Figura 2.11: Heaps de los árboles de las figuras 2.9 y 2.10

$$\left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 0 \\ 2 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix} \right\}$$

Figura 2.12: Palabras código módulo 3

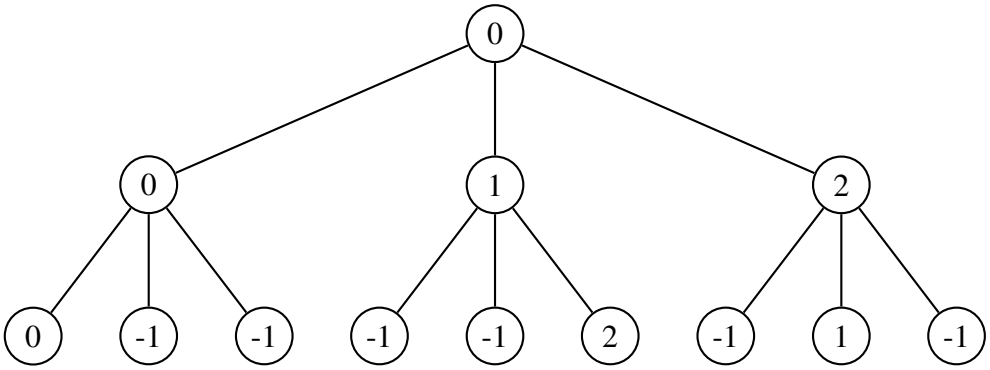

Figura 2.13: Árbol ternario cuya raíz es 0

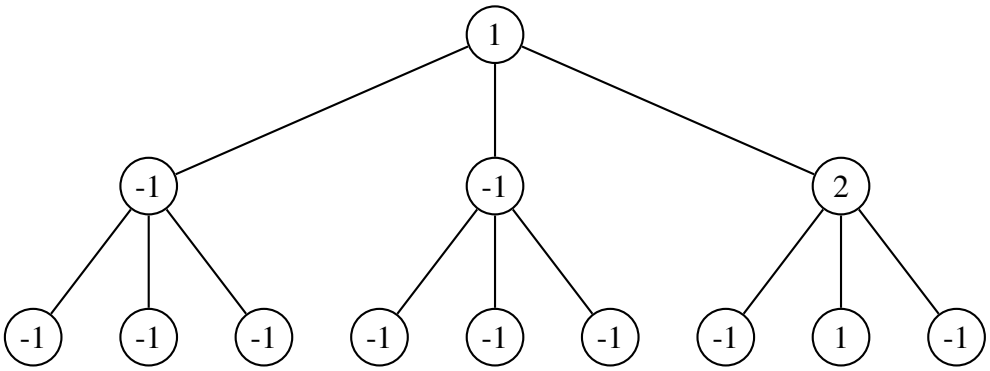

Figura 2.14: Árbol ternario cuya raíz es 1

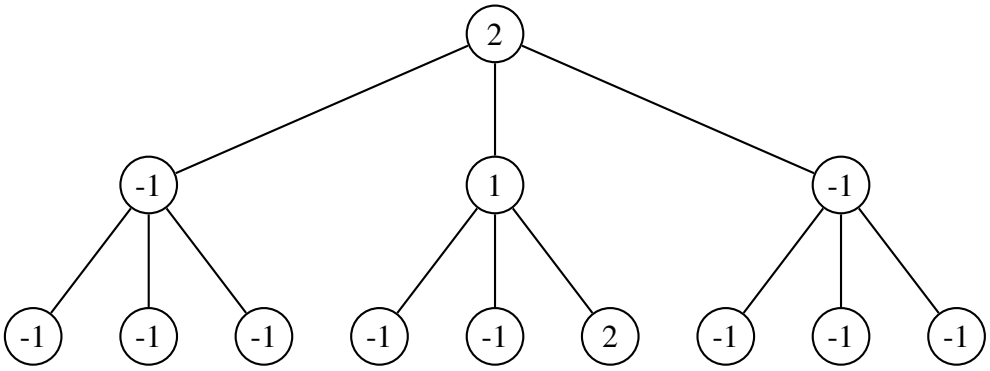

Figura 2.15: Árbol ternario cuya raíz es 2

0	0	1	2	0	−1	−1	−1	−1	2	−1	1	−1
1	−1	−1	2	−1	−1	−1	−1	−1	−1	−1	1	−1
2	−1	1	−1	−1	−1	−1	−1	−1	2	−1	−1	−1

Figura 2.16: Heap que almacena los tres árboles de las figuras 2.13, 2.14 y 2.15

2.3. Matrices Generadoras

En la tabla 2.2 se presentan algunas matrices generadores de códigos lineales sobre $\mathbb{Z}_{p^2}$, junto con el módulo con el que trabajan y la cantidad de palabras código que generan.

Matriz	Módulo	Cardinalidad	Matriz	Módulo	Cardinalidad
$\begin{pmatrix} 1 & 1 & 6 \\ 1 & 6 & 1 \\ 6 & 1 & 1 \end{pmatrix}$	25	625	$\begin{pmatrix} 6 & 6 & 6 \\ 6 & 6 & 6 \\ 6 & 6 & 6 \end{pmatrix}$	25	25
$\begin{pmatrix} 5 & 4 & 8 & 1 \\ 1 & 5 & 4 & 8 \\ 8 & 1 & 5 & 4 \\ 4 & 8 & 1 & 5 \end{pmatrix}$	9	81	$\begin{pmatrix} 1 & 0 & 4 \\ 0 & 4 & 1 \\ 4 & 1 & 0 \end{pmatrix}$	25	3125
$\begin{pmatrix} 5 & 1 & 2 & 1 \\ 1 & 5 & 1 & 2 \\ 2 & 1 & 5 & 1 \\ 1 & 2 & 1 & 5 \end{pmatrix}$	9	81	$\begin{pmatrix} 5 & 0 & 0 & 0 & 4 \\ 4 & 5 & 0 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 4 & 5 & 0 \\ 0 & 0 & 0 & 4 & 5 \end{pmatrix}$	9	6561
$\begin{pmatrix} 3 & 0 & 0 & 0 & 1 & 2 & 1 \\ 1 & 3 & 0 & 0 & 0 & 1 & 2 \\ 2 & 1 & 3 & 0 & 0 & 0 & 1 \\ 1 & 2 & 1 & 3 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 3 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 3 & 0 \\ 0 & 0 & 0 & 1 & 2 & 1 & 3 \end{pmatrix}$	4	256	$\begin{pmatrix} 7 & 1 & 4 \\ 4 & 7 & 1 \\ 1 & 7 & 4 \end{pmatrix}$	9	27

$\begin{pmatrix} 6 & 0 & 0 & 0 & 3 \\ 3 & 6 & 0 & 0 & 0 \\ 0 & 3 & 6 & 0 & 0 \\ 0 & 0 & 3 & 6 & 0 \\ 0 & 0 & 0 & 3 & 6 \end{pmatrix}$	9	81	$\begin{pmatrix} 2 & 0 & 0 & 2 \\ 1 & 3 & 0 & 1 \end{pmatrix}$	4	8

Tabla 2.2: Matrices generadoras

2.4. Código en Sage

En esta sección, se presenta la implementación en Sage de la generación de las palabras código, así como su almacenamiento en un heap.

Código 2.1: codewordsheap.sage

```
def codeword_generatorHeap( matrix, modulus ):
    """
    This functions generates a codewords set taking a matrix and a modulus
    """
    size = matrix.nrows()
    vectorGen = zero_matrix( size, 1)
    sizeArray = (modulus**size - 1)/(modulus - 1) - 1
    heap_matrix = zero_matrix(modulus, sizeArray)
    for index1 in range(0, modulus):
        for index2 in range(0,sizeArray):
            heap_matrix[index1, index2] = -1
    limit = modulus**size
    vectorGen[0, 0] = -1
    for index in range(0, limit):
        localLimit = ceil( index / modulus)
        #print "Local Limite: " + str(localLimit)
        value = vectorGen[0,0] + 1
        vectorGen[0,0] = value % modulus
        localIndex = 1
        for ind in range(0, localLimit):
            if value == modulus:
                value = vectorGen[ localIndex, 0] + 1
                vectorGen[localIndex, 0] = value % modulus
                localIndex = localIndex + 1
```

```
            else:
               break
         #print "Vector generado: " + str(vectorGen.transpose())
         newVector = (matrix * vectorGen)  % modulus
         #print "Vector multiplicado: " + str(newVector.transpose())
         head = newVector[0,0]
         element = newVector[1,0]
         index1 = 1;
         while index1 < size:
            heap_matrix[ head, element] = 1
            index1 = index1 + 1
            #print index1
            if index1 != size:
               element = (element + 1)* modulus + newVector[index1,0]
   codewords = 0
   beginArray = sizeArray - modulus**(size - 1)
   for index in range(0, modulus):
      for index1 in range(beginArray, sizeArray):
         if heap_matrix[index, index1] != -1:
            codewords = codewords + 1
   #print heap_matrix
   print codewords
```

A continuación se muestran los resultados obtenidos utilizando esta nueva versión.

Código 2.2: codewordsheap.sage

```
sage: mat3 = matrix([[5, 4, 8, 1],[1, 5, 4, 8],[8, 1, 5, 4],[4, 8, 1, 5]])
sage: time codeword_generatorHeap( mat3, 9)
81
CPU times: user 1.59 s, sys: 0.04 s, total: 1.64 s
Wall time: 1.63 s

sage: mat5 = matrix([[5, 1, 2, 1],[1, 5, 1, 2],[2, 1, 5, 1],[1, 2, 1, 5]])
sage: time codeword_generatorHeap( mat5, 9)
81
CPU times: user 1.31 s, sys: 0.00 s, total: 1.31 s
Wall time: 1.31 s

sage: mat2 = matrix([[6, 6, 6],[6, 6, 6],[6, 6, 6]])
sage: time codeword_generatorHeap( mat2, 25)
25
CPU times: user 2.46 s, sys: 0.02 s, total: 2.48 s
Wall time: 2.49 s
```

```
sage: mat1 = matrix([[1, 1, 6],[1, 6, 1],[6, 1, 1]])
sage: time codeword_generatorHeap( mat1, 25)
625
CPU times: user 2.61 s, sys: 0.05 s, total: 2.66 s
Wall time: 2.67 s

sage: mat4 = matrix([[1, 0, 4],[0, 4, 1],[4, 1, 0]])
sage: time codeword_generatorHeap( mat4, 25)
3125
CPU times: user 2.64 s, sys: 0.03 s, total: 2.67 s
Wall time: 2.69 s

sage: mat6 = matrix([[5, 0, 0, 0, 4],[4, 5, 0, 0, 0],[0, 4, 5, 0, 0],[0, 0, 4,>
sage: time codeword_generatorHeap( mat6, 9)
6561
CPU times: user 14.05 s, sys: 0.20 s, total: 14.25 s
Wall time: 14.25 s
```

Estos resultados se encuentran resumidos en la tabla 2.3

Matriz	Módulo	Cardinalidad	Tiempo de CPU
$\begin{pmatrix} 1 & 1 & 6 \\ 1 & 6 & 1 \\ 6 & 1 & 1 \end{pmatrix}$	25	625	2.61 s
$\begin{pmatrix} 6 & 6 & 6 \\ 6 & 6 & 6 \\ 6 & 6 & 6 \end{pmatrix}$	25	25	2.46 s
$\begin{pmatrix} 5 & 4 & 8 & 1 \\ 1 & 5 & 4 & 8 \\ 8 & 1 & 5 & 4 \\ 4 & 8 & 1 & 5 \end{pmatrix}$	9	81	1.59 s

$\begin{pmatrix} 1 & 0 & 4 \\ 0 & 4 & 1 \\ 4 & 1 & 0 \end{pmatrix}$	25	3125	2.64 s
$\begin{pmatrix} 5 & 1 & 2 & 1 \\ 1 & 5 & 1 & 2 \\ 2 & 1 & 5 & 1 \\ 1 & 2 & 1 & 5 \end{pmatrix}$	9	81	1.31 s
$\begin{pmatrix} 5 & 0 & 0 & 0 & 4 \\ 4 & 5 & 0 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 4 & 5 & 0 \\ 0 & 0 & 0 & 4 & 5 \end{pmatrix}$	9	6561	14.05 s

Tabla 2.3: Resultados con codewordgeneratorHeap

2.5. Código en C

¿Qué tan rápido puede ser?

La notación $O(f(n))$ es útil para tener una cota superior sobre la complejidad tiempo-espacio del programa. Dado que Cython puede verse como la versión compilada de Python, programar en C da la mejor cota inferior sobre el programa. Si el tiempo de ejecución del algoritmo programado en Python es de n segundos, se espera que en Cython tarde m segundos con $n \geq m$. De aquí que $m \geq t$, donde t segundos es el tiempo que tarda el algoritmo programado en C.

D-Heap $\in \Omega(f(n))$

El listado de código que se presenta a continuación, da el código utilizado para realizar esta comparación. Con conjuntos cuya cardinalidad es mayor de 6000 palabras código, se presentan los mismos resultados que con Magma. Como se desean realizar operaciones sobre las palabras código

así como una interacción con otros objetos algebráicos, el programa en C, sólo sirve como una referencia. Desarrollar una librería en C de Teoría de Códigos sería como reinventar la rueda, dado que ya existen librerías (no completas) que son muy eficientes.

Código 2.3: heapInC.c

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   /*
   This functions generates a set of codewords by taking a matrix and a
       modulus
   """*/
   int modulus,   //modulus to be used
      codewords,  //number of codewords in the set
      size,       //size of the codewords
      index1,     //index used in some fors
      index2,     //another index used in some fors
      limit,      //limit used to determinate how many codewords will
         be generated
      localLimit, //local limit to modify the values of the vectorGen
      localIndex, //local index to determinate which index is going to
         change
      value,      //takes the first element of the vector to check if
         the second has to change
      element,    //element that must be in the matrix
      head,       //second element of the vector generated by
         multiplying the matrix and the vector
      beginArray, //where to start counting to know how many codewords
         there are
      sizeArray;  //number of columns of the matrix
   long index;
   printf("Give the size of the codewords: ");
   scanf("%d", &size);
   printf("Give the modulus: ");
   scanf("%d",&modulus);
   sizeArray = ( (int)pow(modulus,size) - 1)/(modulus - 1) - 1;
```

```
28      int matrix[size][size];
29      int vectorGen[size];
30      int newVector[size];
31      int heap_matrix[modulus][sizeArray];
32
33      printf("Give the elements of the matrix\n");
34      for ( index1 = 0; index1 < size; index1++)
35         for ( index2 = 0; index2 < size; index2++){
36            printf("Give the element %d-%d: ", index1, index2);
37            scanf("%d",&matrix[index1][index2]);
38         }
39      for ( index1 = 0; index1 < size; index1++)
40          vectorGen[ index1 ] = 0;
41      for (index1 = 0; index1 < modulus; index1++){
42         for (index2 = 0; index2 < sizeArray; index2++)
43            heap_matrix[index1][index2] = -1;
44      }
45      limit = (int)pow(modulus,size);
46      vectorGen[0] = -1;
47      for (index= 0; index < limit; index++)
48      {
49         localLimit = (int)ceil( (double)index / (double)modulus);
50         //print "Local Limite: " + str(localLimit)
51         value = vectorGen[0] + 1;
52         vectorGen[0] = value % modulus;
53         localIndex = 1;
54         for (index2 = 0; index2 < localLimit; index2++){
55            if (value == modulus){
56               value = vectorGen[localIndex] + 1;
57               vectorGen[localIndex] = value % modulus;
58               localIndex++;
59            }
60            else{
61               break;
62            }
63         }
64
65         for (index1 = 0; index1 < size; index1++)
```

```
66            newVector[index1] = 0;
67
68         //In the following nested for, the generated vector
69         //is multiplied by the given matrix.
70
71         for (index1 = 0; index1 < size; index1++){
72            for ( index2 = 0; index2 < size; index2++)
73               newVector[index1] += matrix[index1][index2] * vectorGen[
                    index2];
74            newVector[index1]  %= modulus;
75         }
76
77         //print "Multiplied vector: " + str(newVector.transpose())
78         head = newVector[0];
79         element = newVector[1];
80         index1 = 1;
81         while (index1 < size){
82            heap_matrix[head][element] = 1;
83            index1 = index1 + 1;
84            //print index1
85            if (index1 != size)
86               element = (element + 1)* modulus + newVector[index1];
87         }
88      }
89
90      codewords = 0;
91      beginArray = sizeArray - pow(modulus,(size - 1));
92      for (index1 = 0; index1 < modulus; index1++){
93         for (index2 = beginArray; index2 < sizeArray; index2++){
94            if ( heap_matrix[index1][index2] != -1)
95               codewords++;
96         }
97      }
98      printf("There are %d codewords\n", codewords);
99      return 0;
100   }
```

2.6. Código en Cython

Cython es un lenguaje que hace la escritura de extensiones en C para el lenguaje Python tan fácil como el mismo Python. Cython está basado en el muy conocido Pyrex, pero soporta más funcionalidad y optimizaciones de vanguardia. Cython es un lenguaje muy parecido al lenguaje Python, pero Cython adicionalmente soporta llamadas a funciones en C y la declaracion de tipos de C en variables y atributos de clases. Esto le permite al compilador generar código muy eficiente en C a partir del código en Cython. Esto hace que Cython sea el lenguaje ideal para incluir librerías externas de C y para rápidos módulos en C que aceleran la ejecución del código en Python.[9]
Desde la liga [3] se puede obtener la versión más reciente que permite mejorar el código que se escribe para evitar realizar llamadas de Python. Si se desea generar el código en C a partir del código en Cython se procede de la siguiente manera:

```
$ cython programa.pyx
```

Si no hay ningún error, se generará el archivo *programa.pyx* con el código en C correspondiente; en caso contrario, se genera un archivo con el mismo nombre, pero su contenido indica que el archivo no debe ser utilizado dado que hubo un error durante la conversión, esto, posiblemente a causa de llamadas inadecuadas a funciones de Python. Para evitar cometer esta clase de errores, se puede invocar *cython* con el argumento *-a* de la siguiente forma:

```
$ cython -a programa.pyx
```

Ésto, genera un archivo *programa.html* donde se muestra el código en *programa.pyx* pero en líneas amarillas las llamadas a Python que deberían ser removidas para acelerar u optimizar las funciones programadas.

Código de *elements_in_linear_span.spyx*

El listado de código 2.4 presenta la versión en Cython del programa en Sage. El ciclo `while` es el encargado de almacenar los elementos en el heap. Ésto se realiza en un tiempo $O(n)$, donde n es la longitud de la palabra código. Dada las operaciones implicadas dentro de este ciclo, puede darse el caso, de que el ciclo se encuentre en $OO(n)$, aun cuando n no es extremadamente grande.

Código 2.4: codewordsPyrex.pyx

[9] http://www.cython.org/

```
cdef extern from *:
    void* malloc(int)
    int memset(void*, int, int)
    void free(int*)
    double ceil( double )
    double pow( double, double)

def codewordsPyrex( matrixGen, unsigned int modulus, unsigned int size
    ):
    """
    This functions generates a codewords set taking a matrixGen and a
       modulus
    """
    #cdef int size = (int)matrixGen.nrows()
    cdef int* vectorGen = <int*>malloc(sizeof(int) * size)
    cdef int* newVector = <int*>malloc(sizeof(int) * size)
    #Number of columns of the matrix
    cdef long sizeArray = (<int>pow(modulus,size) - 1)/(modulus - 1) - 1
    cdef int limit = <int>pow(modulus,size) #Limit used to determinate
       how many codewords are going to be generated.
    cdef int localLimit        #Local limit to modify the values of the
       vector that multiplies the matrixGen.
    cdef int localIndex        #Local index to determinate which index
       is going to change.
    cdef int value             #Takes the first element of the vector to
        check if the second has to change.
    cdef int element           #Element that must be in the matrixGen.
    cdef int head              #Second element of the vector generated
       by multiplying the matrixGen and the vector.
    cdef int index1            #Index used in some fors.
    cdef int index2            #Another index used in some fors
    cdef int codewords         #Number of codewords in the set
    cdef int mat_elem
    cdef int vec_elem
    cdef int mul
    #Where to start counting to know how many codewords there are.
    cdef long beginArray = sizeArray - <int>pow(modulus,(size - 1))
    cdef long index
```

```
32      cdef int** heap_matrix = <int**>malloc(sizeof(int*)*modulus)
33      for index1 from 0 <= index1 < modulus:
34         heap_matrix[index1]= <int*>malloc(sizeof(int)*sizeArray)
35         memset(heap_matrix[index1],0,sizeof(int)*sizeArray)
36
37      cdef int** matrixGenerator = <int**>malloc(sizeof(int*)*size)
38      for index1 from 0 <= index1 < size:
39         matrixGenerator[index1]= <int*>malloc(sizeof(int)*size)
40         memset(matrixGenerator[index1],0,sizeof(int)*size)
41      for index1 from 0 <= index1 < size:
42         for index2 from 0 <= index2 < size:
43            matrixGenerator[index1][index2] = matrixGen[index1][index2]
44
45      for index1 from 0 <= index1 < modulus:
46         for index2 from 0 <= index2 < sizeArray:
47            heap_matrix[index1][index2] = -1
48      for index1 from 0 <= index1 < size:
49         vectorGen[index1] = 0
50      vectorGen[0] = -1
51      for index from 0 <= index < limit:     #main for to obtain the whole
            set
52         localLimit = <int>ceil( <double>index / <double>modulus )
53         value = vectorGen[0] + 1
54         vectorGen[0] = value % modulus
55         localIndex = 1
56         for index2 from 0 <= index2 < localLimit:
57            if value == modulus:
58               value = vectorGen[localIndex] + 1
59               vectorGen[localIndex] = value % modulus
60               localIndex = localIndex + 1
61            else:
62               break
63         for index1 from 0 <= index1 < size:
64            newVector[index1] = 0
65         for index1 from 0 <= index1 < size:
66            for index2 from 0 <= index2 < size:
67               mat_elem = matrixGenerator[index1][index2]
68               vec_elem = vectorGen[index2]
```

```
69              mul = mat_elem * vec_elem
70              newVector[index1] = newVector[index1] + mul
71           newVector[index1] = newVector[index1]  % modulus
72        head = newVector[0]
73        element = newVector[1]
74        index1 = 1;
75        while index1 < size:
76           heap_matrix[head][element] = 1
77           index1 = index1 + 1
78           if index1 != size:
79              element = (element + 1)* modulus + newVector[index1]
80     free(vectorGen)
81     free(newVector)
82     codewords = 0
83     for index1 from 0 <= index1 < modulus:
84        for index2 from beginArray <= index2 < sizeArray:
85           if heap_matrix[index1][index2] != -1:
86              codewords = codewords + 1
87     return codewords
```

CAPÍTULO 3

RESULTADOS

Este capítulo presenta algunos resultados obtenidos durante el desarrollo de este trabajo. Más resultados del mismo pueden ser revisados en el artículo que se encuentra en el apéndice B.

3.1. Comparaciones

En la tabla 3.1 se pueden apreciar las comparaciones de Sage, tanto compilado como intepretado, con respecto al tiempo de ejecución. La cardinalidad se refiere a la del código $\mathcal{C}$, el módulo se toma como el p^2, tal que el anillo en el cual se está trabajando es $\mathbb{Z}_{p^2}$

Matriz	Módulo	Cardinalidad	Sage (Python)	Sage (Cython)
$\begin{pmatrix} 1 & 1 & 6 \\ 1 & 6 & 1 \\ 6 & 1 & 1 \end{pmatrix}$	25	625	2.61 s	0.10 s
$\begin{pmatrix} 6 & 6 & 6 \\ 6 & 6 & 6 \\ 6 & 6 & 6 \end{pmatrix}$	25	25	2.46 s	0.02 s

$\begin{pmatrix} 5 & 4 & 8 & 1 \\ 1 & 5 & 4 & 8 \\ 8 & 1 & 5 & 4 \\ 4 & 8 & 1 & 5 \end{pmatrix}$	9	81	1.59 s	0.02 s
$\begin{pmatrix} 1 & 0 & 4 \\ 0 & 4 & 1 \\ 4 & 1 & 0 \end{pmatrix}$	25	3125	2.64 s	0.44 s
$\begin{pmatrix} 5 & 1 & 2 & 1 \\ 1 & 5 & 1 & 2 \\ 2 & 1 & 5 & 1 \\ 1 & 2 & 1 & 5 \end{pmatrix}$	9	81	1.31 s	0.02 s
$\begin{pmatrix} 5 & 0 & 0 & 0 & 4 \\ 4 & 5 & 0 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 4 & 5 & 0 \\ 0 & 0 & 0 & 4 & 5 \end{pmatrix}$	9	6561	14.05 s	0.69 s

Tabla 3.1: Tabla Comparativa

3.2. Distancia mínima

En el listado de código 3.1, se puede apreciar que Magma no provee la distancia mínima, de un código cuya cardinalidad no es muy grande. De hecho, Magma da la distancia mínima de códigos cuya cardinalidad es mayor, como se puede apreciar en el listado de código 3.2.

Código 3.1: Distancia mínima con Magma

```
1 > R := Integers(9);
2 > Code := LinearCode(sub<RSpace(R,5) | [5,0,0,0,4], [4,5,0,0,0],
    [0,4,5,0,0],[0,0,4,5,0], [0,0,0,4,5] >);
```

```
> Code;
(5, 6561) Cyclic Linear Code over IntegerRing(9)
Generator matrix:
[1 0 0 0 8]
[0 1 0 0 8]
[0 0 1 0 8]
[0 0 0 1 8]
```

Código 3.2: Distancia mínima de un código más grande

```
> R := Integers(9);
> Code := LinearCode(sub<RSpace(R,5) | [2,1,0,0,0], [0,2,1,0,0], [0,0,2,1,0],
    [0,0,0,2,1], [1,0,0,0,2]>);
> Code;
(5,19683,1)Cyclic Linear Code over IntegerRing(9)
Generator matrix:
[1 0 0 0 2]
[0 1 0 0 2]
[0 0 1 0 2]
[0 0 0 1 2]
[0 0 0 0 3]
```

Pero los métodos de la clase LinearCodeRing, siempre calculan la distancia mínima de cualquier código como se puede apreciar en el listado de código 3.3.

Código 3.3: Generación de un código, así como su distancia mínima

```
sage: M = Matrix(IntegerModRing(9), [[5, 0, 0, 0, 4],[4, 5, 0, 0, 0],[0, 4, 5,
    0, 0],[0, 0, 4, 5, 0], [0, 0, 0, 4, 5]])
sage: CS = RingCodes(M)
sage: CS
Linear Code over the Ring of integers modulo 9
(5, 6561, 2)
```

3.3. Conclusiones

Se puede apreciar que al escoger la estructura de datos adecuada, un problema puede ser resuelto por medio de software libre casi tan eficiente como lo resuelve software propietario como Magma. En este trabajo se cumple claramente el esquema de que *estructuras de datos + algoritmos =*

programas remarcando la importancia de la estructura de datos d-heap que en la tabla 3.1 se muestra la mejora en el desempeño del programa a pesar de que el algoritmo es similar.

3.4. Publicaciones

- Algunos de los resultados de este trabajo se encuentran en el artículo titulado *D-Heaps as Hash Tables for Vectors Over a Finite Ring*, con id 1589, que fue aceptado en el *2009 World Congress on Computer Science and Information Engineering (CSIE 2009)*, que se llevó a cabo del 31 de marzo al 2 de abril del 2009, en Los Angeles/Anaheim, Estados Unidos. Las memorias de las conferencias son publicadas por la IEEE Computer Society. CSIE recibió más de 2200 artículos de todo el mundo.
- Las comparaciones entre Magma y Sage pueden ser apreciadas en los screencasts que se encuentran en [15].

Agradecimientos

A Alexander Hulpke, encargardo del proyecto GAP, por canalizarme con la persona indicada para poder empezar este proyecto. A Robert Bradshaw, lider desarrollador del proyecto Cython, por su valiosa ayuda durante la iniciación en Cython. A David Joyner por sus valiosos comentarios en la realización del artículo, así como por la información proporcionada para poder mejorar varias líneas de código. A mis abuelitos por su apoyo durante tantos años. A mi mamá, a mi familia en general y un agradecimiento especial a mi asesor M.C. Carlos Alberto López Andrade por todo el tiempo dedicado a este trabajo.

APÉNDICE A

LENGUAJES

En este sección se presentan los lenguajes utilizados o que tienen una relación muy estrecha con el trabajo presentado.

C

Los programas presentados en C, sirven para tener una perspectiva de qué tan rápido puede ser un algoritmo programado en otro lenguaje. Dado que C es un lenguaje compilado, es muy difícil que un lenguaje interpretado (como Python) sea más rápido que C, de esta forma, si se tiene un algoritmo programado en C que no es muy rápido, no se puede esperar más rapidez por parte del mismo algoritmo programado en un lenguaje interpretado. De igual forma, Cython (Pyrex modificado), maneja muchas sentencias parecidas a C, así como también, ciertos tipos de C, son los mismos en Cython. Para poder entender el código en C, si no se tiene mucha experiencia con algún lenguaje de programación, se recomienda revisar [11] para entender el código, así como también las diferencias que tiene con otros lenguajes. El capítulo 5 de [26] es útil para comprender conceptos básicos utilizados para trabajar con matrices en Cython. [25] presenta las diferencias entre varios estándares de C, se utiliza el presentado como *C tradicional* en este libro.

LaTeX

El principal uso de LaTeX fue para la correcta realización de este documento. Sin embargo, durante la programación de ciertas clases en Python y en Cython, se utilizó latex para la utilización de la función

```
latex(objeto)
```

Este es uno de varios métodos especiales como estándares de Sage y es indispensable programarlo en toda clase que se desee distribuir con Sage.

Python

Python es el lenguaje utilizado por Sage. Python es un lenguaje interpretado que se basa en la indentación para la definición de bloques, es decir, en vez de utilizar llaves o palabras como `begin` y `end`, utiliza espacios. Python utiliza cuatro espacios para definir los bloques, se puede trabajar proceduralmente, así como orientado a objetos. Se recomienda [19] para aprender Python.

Pyrex

Pyrex es un lenguaje que permite escribir código que maneje tanto los tipos de Python como los tipos de C de la forma que uno desee y compilarlo en una extensión de C para Python. De esta forma, se pueden escribir módulos que extiendan a Python. Pyrex puede ser visto como Python con tipos de datos de C. Más información sore Pyrex puede ser encontrada en [14].

Cython

Cython, visto de una forma, es la versión compilada de Python. Cython es una versión modificada de Pyrex para poder ser utilizado en Sage. Cython es un lenguaje que hace que el escribir extensiones de C para Python sea tan fácil como Python. Cython está basado en Pyrex, pero soporta más funcionalidad y optimizaciones de vanguardia. Cython es muy parecido a Python, pero Cython además soporta llamadas a funciones de C, así como también la declaración de tipos de C en variables y atributos de clases. Esto permite al compilador generar código muy eficiente en C a partir del código en Cython. Esto hace que Cython sea el lenguaje ideal para escribir rápidos módulos en C que aceleran la ejecución de código en Python. Cython aun no está terminado, por lo que cuenta con algunas limitantes, como la anidación de declaraciones, así como también la imposibilidad de usar las funciones *globals()* y *locals()*. Por el momento, las declaraciones de clases y funciones no pueden ser colocadas dentro de estructuras de control. Más información sobre Cython se puede encontrar en [3].

APÉNDICE B

D-HEAPS AS HASH TABLES

D-Heaps as Hash Tables for Vectors Over a Finite Ring[1]

César A. García-Vázquez & Carlos A. López-Andrade

We present a method to store a set of vectors whose coefficients are elements over a finite ring or a prime field. A D-heap is used with the space-time tradeoff technique, to avoid collisions, to store a vector in a set in $\Theta(n)$ time, where n is the length of the vector. We compare this method to hash tables and show why their worst case is the best case of this method. Also the implementation is given in Sage to work with coding theory libraries.

B.1. Introduction

Following [9] and [39], we have the definitions:

Definition B.1. *We denote by* $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$ *the ring of integers module m.*

Definition B.2. *The set* $\mathbb{Z}_m^n$ *of* n*-tuples from* $\mathbb{Z}_m$ *is a* $\mathbb{Z}_m$*-module and by a linear code* $\mathcal{C}$ *over* $\mathbb{Z}_m$ *we mean any* $\mathbb{Z}_m$ *submodule of* $\mathbb{Z}_m^n$

Definition B.3. *A* $\mathbb{Z}_m$*-linear code* $\mathcal{C}$ *is called cyclic if whenever* $(a_0, a_1, \ldots, a_{n-1}) \in \mathcal{C}$ *then* $(a_{n-1}, a_0, a_1, \ldots, a_{n-2}) \in \mathcal{C}$

We will generate linear codes over a finite ring $\mathbb{Z}_m$ from a generator matrix, i.e., we build submodules of $\mathbb{Z}_m^n$ spanned by the set of rows of the given matrix. We denote m, the modulus and n the length of the vector generated. We need to generate m^n vectors and multiply them by the matrix generator. The resulting vectors will form the set *codeSet*. As can be seen, n and m are variables. Anyway, this problem can be solved rapidly if some factors are considered. The notation for complexity is the same as the one presented in [6].

[1] Artículo publicado por la IEEE

B.1.1. Generating codes over finite rings with Magma

Using Magma [4], we can generate codes over finite rings $\mathbb{Z}_m$ as in the following examples

Example B.1. *We are going to generate a code over $\mathbb{Z}_9$ with generator matrix*

$$\begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 \\ 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 2 & 1 \\ 1 & 0 & 0 & 0 & 2 \end{bmatrix}$$

```
> R := Integers(9);
> Code := LinearCode(sub<RSpace(R,5) | [2,1,0,0,0],
      [0,2,1,0,0], [0,0,2,1,0], [0,0,0,2,1],
      [1,0,0,0,2]>) ;
> Code;
(5,19683,1) Cyclic Linear Code over IntegerRing(9)
Generator matrix:
[1 0 0 0 2]
[0 1 0 0 2]
[0 0 1 0 2]
[0 0 0 1 2]
[0 0 0 0 3]
```

According to [5], in an (n, M, d) code the parameter d corresponds to the minimum Hamming weight.

Example B.2. *We are going to generate a code over $\mathbb{Z}_9$ with generator matrix*

$$\begin{bmatrix} 5 & 0 & 0 & 0 & 4 \\ 4 & 5 & 0 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 4 & 5 & 0 \\ 0 & 0 & 0 & 4 & 5 \end{bmatrix}$$

```
> R := Integers(9);
> Code := LinearCode(sub<RSpace(R,5) | [5,0,0,0,4],
      [4,5,0,0,0], [0,4,5,0,0],[0,0,4,5,0],
      [0,0,0,4,5] >);
> Code;
(5, 6561) Cyclic Linear Code over IntegerRing(9)
Generator matrix:
[1 0 0 0 8]
[0 1 0 0 8]
[0 0 1 0 8]
[0 0 0 1 8]
```

In this example, we focus on the fact that Magma does not give the minimum hamming distance.

B.1.2. Generating codes over finite rings with Sage

Using Sage [44], we can compute the same set of codewords with code B.1

Código B.1: Generating codewords

```
def CodeRing( gen_mat ):
    rows = gen_mat.nrows()
    mod = gen_mat.base_ring().order()
    L = range(0, mod)
    M = Tuples(L,rows)
    codeSet = set([])
    for code in M:
        code = vector(IntegerModRing(mod), code)
```

```
            code = code * gen_mat
            code.set_immutable()
            codeSet.add( code)
        return codeSet
```

and calling function `CodeRing` as follows:

```
sage: MS = MatrixSpace(IntegerModRing(9), 5,5)
sage: Mat = MS  ([[5,0,0,0,4],[4,5,0,0,0],
        [0,4,5,0,0],[0,0,4,5,0],[0,0,0,4,5]])
sage: CS = CodeRing(Mat)
```

Even though this implementation is quite easy, it is not as fast as Magma. Hence, we present and compare two methods coded in Cython [3], to compute codes over finite rings of the form $\mathbb{Z}_m$ (or in particularly the prime field $\mathbb{Z}_p$), in much less time. Both methods use space and time tradeoffs as presented in [30]

B.2. Development

During the development, several variables are used, some are global or are used most of the time, then we present a little description about these variables as well as declarations of some of them.

`size` is the number of rows of the generator matrix and `sCols` is the number of columns. `sizeArray` is the number of columns used by each matrix (hash table or d-heap). The variable `mod` is the modulus, `vGen` is the vector which will multiply the generator matrix and `nVec` is the result of this multiplication.

```
cdef int* vGen = <int*>malloc(sizeof(int)*size)
cdef int* nVec = <int*>malloc(sizeof(int)*sCols)
```

B.2.1. Hashing codewords

The first method uses hashing [27], code B.2 presents the declaration of the hash table.

Código B.2: Declaring the hash table

```
cdef long sizeArray = <int>pow(modulus, sCols)
cdef int** hash_matrix = <int**>malloc(sizeof(int*)
    *sCols)

for index from 0 <= index1 < mod:
    hash_matrix[index]= <int*>malloc(sizeof(int)*
        sizeArray)
    memset(hash_matrix[index ],0, sizeof(int)*
        sizeArray)
```

Despite [29], since the objects are codewords, it is possible to generate an injective hash function to avoid collisions. We take each codeword as a m-representation of a number (Basis Representation Theorem as shown in [1]), which will be our slot. Given that no collisions occur, this is a perfect hashing scheme, different from what is presented as perfect hashing in [10]

Example B.3. *If we have the codeword* $(\ 1\ \ 0\ \ 3\ )$ *over* $\mathbb{Z}_5$*, the corresponding slot is* $1+0\times5^1+3\times5^2 = 1+0+3\times25 = 1+75 = 76$*. This means that, if we stored the codewords in a matrix, this codeword would be in the column number 76.*

Using Horner's rule as presented in [35], we can hash each codeword as in code B.3.

Código B.3: Horner's rule in Cython

```
P = nVec[ sCols - 1]
for i from 0 <= i < sCols - 1:
    P = modulus * P
    P = P + nVec[sCols - 2 - i]
```

Code B.4 shows how each codeword is stored.

Código B.4: Storing hashed codewords

```
for i from 0 <= i < sCols:
        hash_matrix[i][P] = nVec[i]
```

Hence, we only need a matrix with n rows and s columns, where $s \leq m^n$, s is the cardinality of the code, the space complexity of this algorithm is $O(n \cdot s)$.

$$\left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\}$$

Figura B.1: Linear code

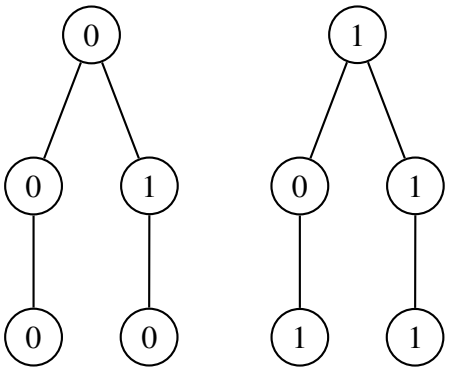

Figura B.2: Forest from figure B.1

B.2.2. Completing the trees

To be able to use d-heaps, we need to work with complete trees. We can't balance the trees because the codewords will be lost, we need to add **sentinels**, in this case *-1*.

Example B.4. *Suppossing that we have the code over $\mathbb{Z}_2$ in the figure B.1.*

The figure B.2 shows the corresponding forest to that code. Since we have a binary code, there are only two trees. The figure B.3 shows the complete forest after adding the sentinels.

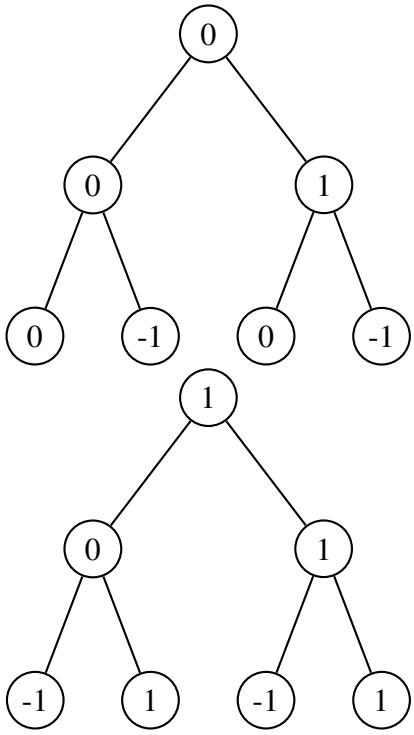

Figura B.3: Complete forest from figure B.2

B.2.3. Generating the tuples

Code B.5 shows a slight modification of the algorithm presented in [28] to generate all the tuples. This code is used by both methods.

Código B.5: Generating the tuples

```
localLimit = <int>ceil(<double>index/<double>mod)
value = vGen[0] + 1
vGen[0] = value % mod
localIndex = 1

for index2 from 0 <= index2 < localLimit:
    if value == mod:
        value = vGen[localIndex] + 1
        vGen[localIndex] = value % mod
        localIndex = localIndex + 1
    else:
        break
```

B.2.4. D-Heaps

As presented in [21], we can construct a d-heap from figure B.3 to have the table B.1.

0	0	1	0	-1	0	-1
1	0	1	-1	1	-1	1

Tabla B.1: D-heap from figure B.3

Code B.7 shows the code that stores each codeword into the d-heap structure. The formula varies a little from the one presented in [21] because there is no need to store the first element, as it is the same as the row number. The variable `head` is the first element of the codeword, so, it is the number of the row where the codeword is going to be stored and `elem` is the column where next element is stored. Since the first element needn't be stored as is the same as the row number, the number of units needed to store each tree is:

$$m + m^2 + \cdots + m^{n-1} = \sum_{i=1}^{n-1} m^i = \frac{m^n - 1}{m - 1} - 1 \tag{B.1}$$

Código B.6: Declaring D-Heap

```
cdef long sizeArray = (<int>pow(mod,sCols) - 1)/(
    mod - 1) - 1
cdef int** heap_matrix = <int**>malloc(sizeof(int*)
    *mod)

for index from 0 <= index < modulus:
    heap_matrix[index]= <int*>malloc(sizeof(int)*
        sizeArray)
    memset(heap_matrix[index],0, sizeof(int)*
        sizeArray)
```

Código B.7: Storing vectors

```
head = nVec[0]
elem = -1    #To start filling
index = 1;

#Filling While
while index < sCols:
    elem = (elem + 1) * mod + nVec[index]
    heap_matrix[head][elem] = nVec[index]
    index = index + 1
```

Theorem B.1. *The space complexity of this algorithm is* $O(m^n)$

Proof: From (B.1), we need $\frac{m^n - 1}{m - 1} - 1$ units for each tree and we have m trees. So we

have that

$$\begin{aligned} m \cdot \left(\frac{m^n - 1}{m - 1} - 1 \right) &\leq m \cdot \left(\frac{m^n - 1}{m - 1} \right) \\ &\leq \frac{m}{m - 1} \cdot (m^n - 1) \\ &\leq 2 \cdot (m^n - 1) \\ &\leq 2 \cdot m^n \end{aligned}$$

□

B.2.5. Removing 0

To obtain the minumum distance, the codeword 0 must be removed to avoid unnecessary comparisons. Code B.8 shows how this codeword is removed from the d-heap and code B.9 removes it from the hash table.

Código B.8: Removing 0 from D-Heap

```
elem = -1
head = 0
index = 1
while index < sCols:
    element = (elem + 1) * mod
    heap_matrix[head][elem] = -1
    index = index + 1
```

Código B.9: Removing 0 from hash table

```
for i from 0 <= i < sCols:
    hash_matrix[i][0] = -1
```

B.2.6. D-Heaps in Cython

In code B.10, we show how each codeword in cython is changed to a codeword over the corresponding finite ring, as well as the minimum hamming distance is obtained.

Código B.10: Generating codewords for Sage

```
for index1 from 0 <= index1 < modulus:
 for index2 from 0 <= index2 < endArray:
  for index from 0 <= index < modulus:
   if self.heap_matrix[index1][beginArray + index] != -1:
    addVector = ([])
    codewords = codewords + 1
    vGen[limit] = index
    localMin = 0
    if index != 0:
     localMin = localMin + 1
     head = beginArray
      for index3 from limit > index3 >= 1:
       head = <int> floor( (head - 1) / modulus)
       vGen[index3] = self.heap_matrix[index1][head]
       if vGen[index3] != 0:
        localMin = localMin + 1
        head = head - vGen[index3]
       vGen[0] = index1
       if index1 != 0:
        localMin = localMin + 1
       if localMin < minimum:
        minimum = localMin
       for index3 from 0 <= index3 < sCols:
        addVector.append(vGen[index3])
       addVector = vector(IntegerModRing(modulus), addVector)
       codeSet.append(addVector)
  beginArray = beginArray + modulus
 beginArray = index2Start
```

B.2.7. Returning to Sage

To be able to interact with the coding theory functions in Sage, we need to return the set in an appropriate way. Code B.11 return the codewords as a list from the hash table.

Código B.11: Returning from Hash Table

```
for i from 0 <= i < sizeArray:
  if heap_matrix[0][i] != -1:
    addVector = ([])
    localMin = 0
    codewords = codewords + 1
    for index from 0 <= index < sCols:
        addVector.append(hash_matrix[index][i])
        if hash_matrix[index][i] != 0:
            localMin = localMin + 1
        if localMin < minimum:
            minimum = localMin
        addVector = vector(IntegerModRing(mod),
            addVector)
        codeSet.append(addVector)
```

B.3. Results

With these two methods presented, we can generate codes over finite rings, $\mathbb{Z}_m$, almost as efficient as Magma.

B.3.1. The minimum distance

Not only have the vectors been generated, but also the minimum (Hamming) distance has been obtained. This is done in a trivial way, verifying every element in the set. As code B.10 shows, for d-heap, in line 21, a comparison is made between the actual minimum distance with a probable new minimum distance, if so, a new minimum distance has been found. The same procedure is done for the hash table, as code B.11 shows. This way, the minimum distance is **always** obtained and we have the following result

```
1 sage: MS = MatrixSpace(IntegerModRing(9), 5,5)
2 sage: Mat = MS  ([[5,0,0,0,4],[4,5,0,0,0],
          [0,4,5,0,0],[0,0,4,5,0],[0,0,0,4,5]])
3 sage: CS = LinearCodeRing( Mat )
4 sage: CS
5 Linear Code over the Ring of integers modulo 9
6 (5, 6561, 2)
```

From line 6, we now know that the minimum hamming distance of the code generated is 2.

B.3.2. Comparisons

Even though most of the code for hashing vectors is shorter and clearer than the one for d-heap, there are some factors that must be considered as the space complexity:

- D-heap technique is $O(m^n)$
- Hashing is $O(n \cdot s)$

If we have that $s \approx m^n$, hashing uses $O(nm^n)$ units of space. Hence, the worst case for hashing is the best case for d-heap because it uses all the space declared from the beginning.

B.3.3. License

Codes B.1, B.2, B.3, B.4, B.5, B.6, B.7, B.8, B.9, B.10 and B.11 are distributed under the terms of the GNU General Public License version 2 or above, at your preference, (GPLv2+) and in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY

B.3.4. Acknowledgements

Special thanks to Carlos Guillén Galván for giving the idea about the injective function to hash vectors and to David Joyner for his valuable suggestions.

BIBLIOGRAFÍA

[1] G. E. Andrews. *Number Theory*, pages 8–10. Dover Publications, first edition, 1994.

[2] T. M. Apostol. *Introducción a la Teoría Analítica de Números*. Reverté, España, 2002.

[3] S. Behnel, R. Bradshaw, and G. Ewing. *Cython: C-Extensions for Python*, 2008. `http://www.cython.org`.

[4] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *Symbolic Computation*, 24(3-4):235–265, 1997.

[5] W. Bosma, J. Cannon, and C. Playoust. *Handbook of Magma Functions*, volume 13: Coding Theory and Cryptography, page 4267. Magma, 2.14 edition, 2008.

[6] G. Brassard. Crusade for a better notation. *SIGACT News*, 17(1):60–64, 1985.

[7] G. Brassard and P. Bratley. *Fundamentos de Algoritmia*. Prentice Hall, Madrid, 1ra edition, 1997.

[8] A. R. Calderbank and N. J. A. Sloane. Modular and p-adic cyclic codes. *Des., Codes and Cryptogr.*, 6:21–35, 1995.

[9] I. Constantinescu and T. Heise. A metric for codes over residue class rings of integers. *Problemy Peredachi Informatsii*, 33(3):22–28, 1997.

[10] T. H. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd, third printing edition, 2002.

[11] H. M. Deitel and P. J. Deitel. *Como Programar en C/C++*. Prentice Hall, 1995.

[12] H. Q. Dinh and S. R. López-Permouth. Cyclic and negacyclic codes over finite chain rings. *IEEE Trans. Inform. Theory*, 50:1728–1744, August 2004.

[13] J. Dumas, J. Roch, E. Tannier, and S. Varrette. *Thèorie des codes*. Dunod, 1ère edition, 2007.

[14] G. Ewing. Pyrex, un lenguaje para escribir módulos de extensión de python. `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`.

[15] C. A. García-Vázquez. Magma vs sage. `http://sage.math.washington.edu/home/wdj/expository/vazquez/`.

[16] J. v. z. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge, 2nd edition, 2003.

[17] R. P. Grimaldi. *Matemáticas Discreta y Combinatoria: Una introducción con aplicaciones*. Addison Wesley Longman, México, 1ra edition, 1998.

[18] I. N. Herstein. *Álgebra Moderna*. Trillas, México, 2da edition, 2006.

[19] M. L. Hetland. *Beginning Python: From Novice to Professional*. Apress, EUA, 2005.

[20] T. W. Hungerford. *Abstract Algebra: An introduction*. Saunders College Publishing, Philadelphia, 1990.

[21] D. B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4(3):53–57, December 1975.

[22] R. Johnsonbaugh. *Matemáticas Discretas*. Prentice Hall, México, 4ta edition, 1999.

[23] A. R. H. Jr., P. V. Kumar, A. R. Calderbank, N. J. A. Sloane, and P. Solé. The $\mathbb{Z}_4$-linearity of kerdock, preparata, goethals, and related codes. *IEEE Trans. Inform. Theory*, 40(2):301–319, March 1994.

[24] P. Kanwar and S. R. López-Permouth. Cyclic codes over the integers modulo p^m. *Finite Fields Their Applic.*, 3(4):334–352, 1997.

[25] A. Kelley and I. Pohl. *C by Dissection*. Addison-Wesley, 1996.

[26] B. W. Kernighan and D. M. Ritchie. *El Lenguaje de Programción C*. Prentice Hall, 1991.

[27] D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching, pages 513–558. Addison-Wesley, Reading, MA, second edition, 1997.

[28] D. E. Knuth. *The Art of Computer Programming*, volume 4: (Fascicle 4) Generating All Trees – History of Combinatorial Generation. Addison-Wesley, Reading, MA, first edition, 2006.

[29] J. Körner and K. Marton. New bounds for perfect hashing via information theory. *Eur. J. Comb.*, 9(6):523–530, 1988.

[30] A. Levitin. *Introduction to The Design & Analysis of Algorithms*, pages 249–277. Addison Wesley, 2007.

[31] R. Lidl and H. Niederreiter. *Finite Fields*. Cambrige University Press, MA, 1997.

[32] S. Ling and J. T. Blackford. $\mathbb{Z}_{p^{k+1}}$-linear codes. *IEEE Trans. Inform. Theory*, 48(9):2592–2605, September 2002.

[33] C. A. López-Andrade and H. Tapia-Recillas. On the quasi-cyclicity of the gray map image of a class of codes over galois rings. *ICMCTA, LNCS*, 5228:107–116, 2008.

[34] F. J. Macwilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North Holland, 1977.

[35] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[36] G. H. Norton and A. Sălăgean. On the structure of linear and cyclic codes over a finite chain ring. *Appl. Algebra Eng. Commun. Comput.*, 10:489–506, 2000.

[37] Y. H. Park. Modular independence and generator matrices for codes over $\mathbb{Z}_m$. *Des., Codes and Cryptogr.*, 50:147–162, 2009.

[38] J. Quinet. *Curso de Matemáticas Superiores: Algebra*. Paraninfo, 1983.

[39] B. S. Rajan and M. U. Siddiqi. Transform domain characterization of cyclic codes over $\mathbb{Z}_m$. *Appl. Algebra Eng. Commun. Comput.*, 5:261–275, 1994.

[40] J. J. Rotman. *Advanced Modern Algebra*. Prentice Hall, 2003.

[41] R. Sedgewick. *Algorithms in Java: Parts 1-4*. Addison Wesley, 3ra edition, 2003.

[42] R. Sedgewick. *Algorithms in Java: Parts 5*. Addison Wesley, 3ra edition, 2003.

[43] W. Stein. Sage dvd: http://www.lulu.com/content/2461899.

[44] W. Stein. *Sage: Open Source Mathematical Software (Version 3.2.2)*. The Sage Group, 2008. http://www.sagemath.org.

[45] W. Stein. *Sage Tutorial*. CreateSpace, 2008.

[46] J. Unpincgo. Video introductorio a sage. http://sage.math.washington.edu/home/wdj/expository/unpingco/.

[47] J. F. Wakerly. *Error Detecting Codes, Self-Checking Circuits and Applications*. North-Holland, Amsterdam, The Netherlands, 1978.

[48] Z. X. Wan. *Lectures on Finite Fields and Galois Rings*. Singapure: World Scientific Publish. Co., 2003.

[49] J. Wolfmann. Binary images of cyclic codes over $\mathbb{Z}_4$. *IEEE Trans. Inform. Theory*, 47:1773–1779, July 2001.

[50] F. Zaldívar. *Teoría de Galois*. Anthropos, España, 1ra edition, 1996.