2032 - Codis no lineals en Magma: Noves Construccions

Memòria del projecte de final de carrera corresponent als estudis d'Enginyeria Superior en Informàtica presentat per Miriam Gutiérrez Alarcón i dirigit per Mercè Villanueva Gay.

Bellaterra, juny de 2010

La firmant, Mercè Villanueva Gay, professora del Departament d'Enginyeria de la Informació i de les Comunicacions de la Universitat Autònoma de Barcelona

CERTIFICA:

Que la present memòria ha sigut realitzada sota la seva direcció per Miriam Gutiérrez Alarcón

Bellaterra, juny de 2010

Firmat: Mercè Villanueva Gay

a la meva família

Agraïments

Voldria donar les gràcies al Raúl Jiménez, per llegir-se la memòria i comunicar-me tot el que no entenia. El seu punt de vista extern m'ha servit per a millorar. També per aguantar el meu nerviosisme durant aquests darrers mesos i fer-me costat en l'última etapa d'aquest llarg camí que ha sigut la carrera.

També agraïr a la Laura Vidal les seves paraules d'ànim i les vivències compartides des de l'experiència.

Al Jaume Pujol, per ajudar-me amb els seus amplis coneixements de MAGMA.

Especialment, voldria agraïr la inestimable ajuda de la meva directora de projecte, Mercè Villanueva, per la seva disponibilitat i atenció alhora de resoldre els dubtes. Per les seves paraules encoratjadores i mostres de reconeixement. S'ha establert una relació col·laborativa molt agradable.

A tots vosaltres, moltes gràcies.

Índex

Índex de figures

Capítol 1

Introducció

Ens trobem immersos en una societat on la informació és molt valuosa fins al punt que ens és indispensable diposar d'ella en qualsevol lloc i moment. Davant aquesta necessitat d'intercanvi d'informació, les xarxes de comunicació tenen un paper important, permetent la transferència de dades arreu del món.

Quan enviem dades a través d'un canal de comunicació digital es poden produïr errors que comporten la pèrdua d'informació. Per a garantir que el missatge rebut sigui el que s'ha enviat utilitzem els codis correctors d'errors. Aquests codis es basen en afegir redundància a la informació que volem transmetre, per tal de poder corregir els errors que s'hagin produït durant la transmissió. Segons les seves propietats, aquests codis es poden classificar com a lineals o no lineals. Normalment s'utilitzen els codis lineals, ja que són més fàcils de manipular. En aquest projecte ens centrarem principalment en els codis no lineals, sobre un alfabet binari, per tal de disposar d'eines que facilitin la seva manipulació.

L'objectiu principal d'aquest projecte de recerca del CCG[1] és el desenvolupament d'una llibreria, anomenada *BinaryCodes*, en el software Magma

[1]Grup de Combinatòria i Codificació

que permeti la construcció i manipulació de codis binaris no lineals de forma eficient. Això suposarà poder extrendre la llibreria actual de MAGMA que només permet treballar amb codis lineals, per al cas no lineal.

El projecte que presentaré tot seguit és una continuació del treball realitzat per altres projectistes, per tant, ha sigut necessari un estudi previ de la documentació d'altres PFCs. El paquet té llicència GNU i estarà disponible per a tota la comunitat científica interessada en la recerca en codis no lineals. A més, els desenvolupadors de MAGMA han mostrat el seu interès d'integra-lo en properes distribucions. Per aquest motiu, documentació i codificació són en anglès i segueixen l'estil de MAGMA definit al document *CCGStyleGuide* [dg10].

1.1 Objectius

Els objectius definits a la planificació inicial són els següents:

1. Estudiar en quina situació es troba el paquet *BinaryCodes*, necessari per a la seva continuació.
2. Estudiar els fonaments teòrics de les contruccions de codis no lineals que s'han d'implementar.
3. Ampliar la funcionalitat del paquet, desenvolupant funcions que ens permeten la manipulació de codis binaris lineals i no lineals.
4. Elaborar testos prou robustos, pensant en tots els casos possibles, que garanteixin que les funcions codificades fan exactament el que s'espera d'elles.
5. Elaborar exemples per a les funcions desenvolupades que il·lustrin les seves propietats principals.
6. Documentar en anglès funcions, testos i exemples, així com el manual de MAGMA que servirà d'ajuda per als futurs usuaris del paquet

BinaryCodes.

7. Unificar les funcions, testos, exemples i documentació desenvolupats seguint l'estil definit en el document *CCGStyleGuide* [dg10].

8. Codificar i documentar el paquet *BinaryCodes* de la manera més estructurada, organitzada i clara possible.

9. Redactar la present memòria que resumeix tota la feina feta per a assolir els objectius establerts.

1.2 Contingut de la memòria

Aquesta memòria es divideix en diversos capítols. A continuació es descriu el contingut de casdascun:

Capítol 2: Fonaments teòrics. Revisió dels coneixements matemàtics sobre codis binaris lineals i no lineals: propietats i representació utilitzada en la part del paquet *BinaryCodes* desenvolupat fins ara.

Capítol 3: Anàlisi i planificació del projecte. Estudi de la viabilitat del projecte en diversos àmbits, revisant objectius i estat de l'art. Inclou la planificació temporal inicial i real i el pressupost.

Capítol 4: Desenvolupament del projecte. Descripció del *software* utilitzat i de les tècniques de testeig emprades. Definició de les funcions desenvolupades.

Capítol 5: Anàlisi de resultats. Comparació, mitjançant taules, dels temps d'execució obtinguts per a les versions codificades. Justificació de la versió escollida en cada cas.

Capítol 6: Conclusions. Assoliment d'objectius respecte els plantejats a l'inici. Resum de les conclusions extretes i possibles millores per a futurs projectes.

Bibliografia.

Apèndix A. Manual de funcions desenvolupades en MAGMA.

Apèndix B. CD amb el codi font, tests, exemples i manual de la llibreria desenvolupada.

Capítol 2

Fonaments teòrics

En aquest capítol farem una descripció dels conceptes teòrics necessaris per a entendre les propietats i representació dels codis binaris lineals i no lineals. Aquesta base s'ha extret de l'article [GPV09] i dels projectes final de carrera [Cua10] i [Vid09].

Per a il·lustrar les definicions s'han inclòs exemples en MAGMA al llarg del capítol.

2.1 Codis correctors d'errors

Quan enviem informació a través d'un canal de comunicació digital pot succeïr que les pertorbacions modifiquin el missatge de tal manera que aquest sigui diferent a l'enviat (Figura 2.1).

Per tal d'aconseguir que el receptor rebi correctament el missatge original s'utilitzen els codis correctors d'errors. La seva funció és detectar els possibles errors del missatge rebut i corregir-los mitjançant la redundància d'informació. Aquests codis estan formats per vectors anomenats paraules codi. Donat que només tractarem amb codis binaris, cada paraula codi és un vector format per zeros i uns.

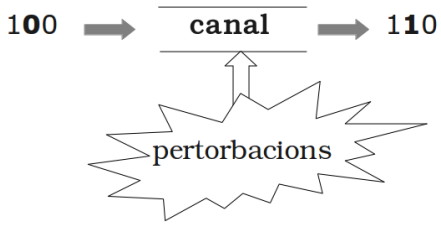


Figura 2.1: Les pertorbacions poden modificar el missatge enviat

2.2 Característiques dels codis binaris

Endinsant-nos una mica més en el llenguatge matemàtic, siguin $\mathbb{F}^k$ i $\mathbb{F}^n$ els espais vectorials de dimensió k i n respectivament definits sobre un cos finit $\mathbb{F}$. O el que és el mateix, siguin $\mathbb{F}^k$ i $\mathbb{F}^n$ el conjunt de tots els vectors de k i n coordenades sobre $\mathbb{F}$ respectivament. Diem que un codi C és la imatge d'una aplicació $f : \mathbb{F}^k \to \mathbb{F}^n$, per tant, que transforma (codifica) vectors de longitud k en vectors de longitud n: $C = f(\mathbb{F}^k) \subseteq \mathbb{F}^n$.

A partir d'ara parlarem només de codis binaris, és a dir, $\mathbb{Z}_2$ és el cos finit $GF(2) = \{0, 1\}$ i els elements de $\mathbb{Z}^k$ i $\mathbb{Z}^n$ són vectors binaris de longitud k i n respectivament.

EXEMPLE 2.1 ***Redundància d'informació***
Si C és un codi, la comanda $Set(C)$ de MAGMA ens retorna totes les paraules codi que formen el codi C. El codi de repetició de longitud 3 ens permet corregir un error.

```
> C := RepetitionCode(GF(2),3);
> Set(C);
{
    (1 1 1),
    (0 0 0)
}
```

El codi de repetició C de longitud $n = 3$ és la imatge de la funció

$f : \{0,1\} \to \{000,111\}$ que ens permet codificar vectors de longitud 1 en vectors de longitud 3. Com veiem a la Figura 2.2, si volem transmetre el missatge 100 el que enviem és el vector 111 000 000.

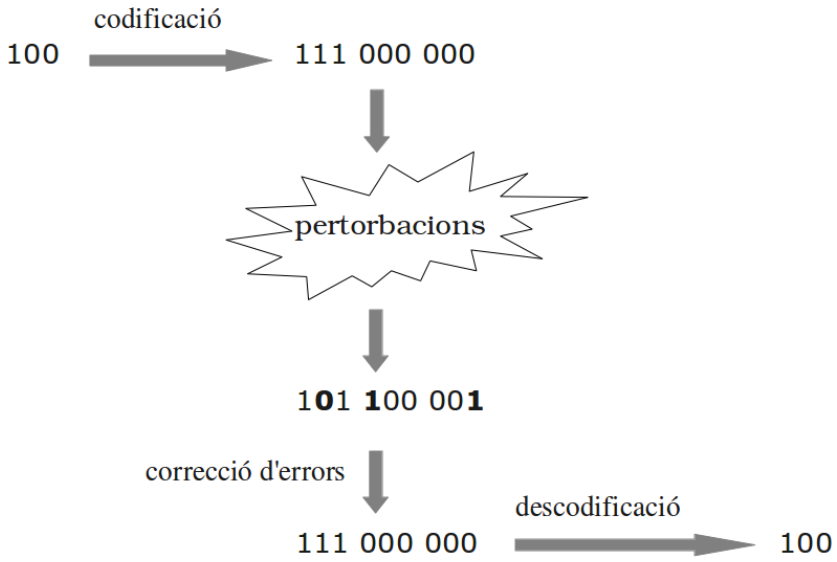


Figura 2.2: La redundància ens permet corregir errors en el missatge transmès

Per tant, per a cada bit d'informació s'afegeixen 2 bits de redundància. El codi C és 1-corrector, o sigui pot corregir com a màxim 1 bit a cada paraula codi (a cada bloc de 3 bits). Si es produís més d'un error en el bloc de 3 bits, Figura 2.3, l'error no es podria corregir.

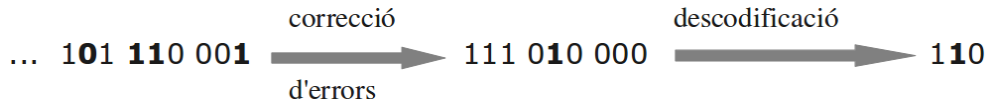


Figura 2.3: Un codi 1-corrector només corregeix 1 bit a cada paraula codi

Sigui $\mathbb{Z}_2^n$ l'espai vectorial de dimensió n sobre el cos finit $\mathbb{Z}_2$. Podem definir un codi binari com un subconjunt del conjunt de totes les paraules codi de $\mathbb{Z}_2^n$, format per vectors binaris de longitud n.

Definim la **distància de Hamming** $d_H(x, y)$ entre dos vectors binaris x, $y \in \mathbb{Z}_2^n$ tal que $x = (x_1, \cdots, x_n)$, $y = (y_1, \cdots, y_n)$ com el nombre de coordenades diferents entre ells:

$$d_H(x, y) = \#\{i \mid 1 \leq i \leq n, x_i \neq y_i\}.$$

La distància de Hamming compleix la propietat de simetria:

$$d_H(x, y) = d_H(y, x).$$

La **distància mínima de Hamming** d d'un codi binari C és la distància més petita entre cada parella de paraules codi:

$$d = \min\{d_H(u, v) \mid u \neq v,\ u, v \in C\}.$$

Arribats a aquest punt, podem definir un codi binari $C(n, M, d)$ com un subconjunt de vectors de $\mathbb{Z}_2^n$ de longitud n, cardinalitat M i distància mínima de Hamming d.

Exemple 2.2 ***Longitud, cardinalitat i distància mínima d'un codi***
Calculem un nou codi binari lineal aleatori.

```
> C := RandomLinearCode(GF(2),5,3);
 > C;
[5, 3, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 1 0]
[0 1 0 1 0]
[0 0 1 0 1]
> Length(C);
5
> #Set(C);
8
> 2^3 eq #Set(C);
true
```

```
> MinimumDistance(C);
2
```

Busquem la distància entre totes les parelles de paraules codi i comprovem que el mínim coincideix amb la distància mínima calculada.

```
> d := [Distance(u,v) : u,v in C | u ne v];
> d;
[ 2, 2, 2, 4, 4, 4, 2, 2, 2, 2, 4, 4, 2, 4, 2, 2, 2, 4,
2, 4, 4, 2, 2, 2, 2, 4, 4, 4, 4, 4, 4, 2, 2, 2, 2, 4, 4,
2, 4, 2, 2, 2, 4, 2, 4, 4, 2, 2, 2, 2, 4, 4, 4, 2, 2, 2 ]
> Minimum(d) eq MinimumDistance(C);
true
```

El **pes de Hamming** $w_H(x)$ d'un vector binari $x \in \mathbb{Z}_2^n$ és el nombre de components del vector que siguin diferents de 0:

$$w_H(x) = d_H(x, \mathbf{0}),$$

on $\mathbf{0}$ és la paraula zero.

El **pes mínim de Hamming** w d'un codi binari C és el pes de la paraula codi de menor pes:

$$w = \min\{w_H(v) \mid v \in C, v \neq \mathbf{0}\}.$$

La **capacitat correctora** e d'un codi binari $C(n, M, d)$ ve determinada per la seva distància mínima:

$$e = \lfloor \frac{d-1}{2} \rfloor.$$

Exemple 2.3 ***Pes mínim i capacitat correctora d'un codi***
La capacitat correctora per al codi C és zero.

```
> e := Floor((MinimumDistance(C) - 1) / 2);
> e;
0
```

```
> MinimumWeight(C);
2
```

Veiem la distribució de pesos del codi que ens mostra per a cada cas possible quantes paraules codi hi ha, excloent la paraula codi zero. Observem que, efectivament, el menor pes possible és 2.

```
> WeightDistribution(C);
[ <0, 1>, <2, 4>, <4, 3> ]
```

2.3 Codis binaris lineals i no lineals

Els codis binaris tenen una sèrie de propietats algebraiques que els classifiquen en codis lineals o no lineals. Com hem dit abans, els codis binaris són un subconjunt de $\mathbb{Z}_2^n$. Anomenem **lineals** a aquells que a més són un subespai vectorial. Això implica que la suma de qualsevol parella de paraules codi dóna com a resultat una paraula que també pertany al codi. Qualsevol codi lineal contindrà la paraula zero. En canvi, els **no lineals** no són un subespai vectorial i poden no contenir la paraula zero.

Com que els codis lineals tenen una dimensió k com a subespai de $\mathbb{Z}_2^n$, la cardinalitat és $M = 2^k$ i els denotem com $C(n, 2^k, d)$. Es poden representar de manera compacta amb una matriu on les seves k files formen una base del codi com a subespai vectorial. És a dir, a partir d'una base de k vectors es pot generar qualsevol paraula del codi. L'anomenem **matriu generadora** G i té n columnes i k files.

El **codi ortogonal**, també anomenat **dual**, d'un codi lineal C és un codi lineal $C^\perp$ $(n, 2^{n-k}, d')$ que compleix la propietat $(C^\perp)^\perp = C$. Definim $C^\perp = \{x \in \mathbb{Z}_2^n \mid x \cdot v = 0,\ \forall v \in C\}$ on "$\cdot$" és el producte escalar.

La **matriu de control** H d'un codi lineal C és la matriu generadora del dual $C^{\perp}$ i té n columnes i $n-k$ files. Ens permet saber si un vector pertany al codi perquè el seu producte escalar amb H s'anul·la.

Exemple 2.4 ***Matriu generadora i matriu de control d'un codi***
Calculem la matriu generadora i de control del codi C de longitud 5 i dimensió 3. Podem dir que el vector v_1 pertany al codi C perquè al multiplicar-lo per H dóna zero.

```
> GeneratorMatrix(C);
[1 0 0 1 0]
[0 1 0 1 0]
[0 0 1 0 1]
> H := ParityCheckMatrix(C);
> H;
[1 1 0 1 0]
[0 0 1 0 1]
> v1:=VectorSpace(GF(2),5)![1,1,0,0,0];
> v1*Transpose(ParityCheckMatrix(C));
(0 0)
```

En canvi el vector v_2 no pertany al codi C.

```
> v2:=VectorSpace(GF(2),5)![1,0,0,0,1];
> v2*Transpose(H);
(1 1)
```

Per tant, podem representar els codis binaris lineals de forma compacta amb la seva matriu generadora. Només necessitarem guardar $n \times k$ valors corresponents als k vectors de longitud n (n coordenades).

Per a codis no lineals se'ns planteja un problema, ja que no hi ha cap base del codi que generi totes les paraules codi. En aquest cas necessitem emmagatzemar 2^k vectors de n coordenades, és a dir, $n \times 2^k$ valors. Quan tractem amb codis molt grans resulta inviable tant guardar-los com treballar amb ells per problemes d'espai a memòria.

2.4 Rang i Kernel d'un codi binari

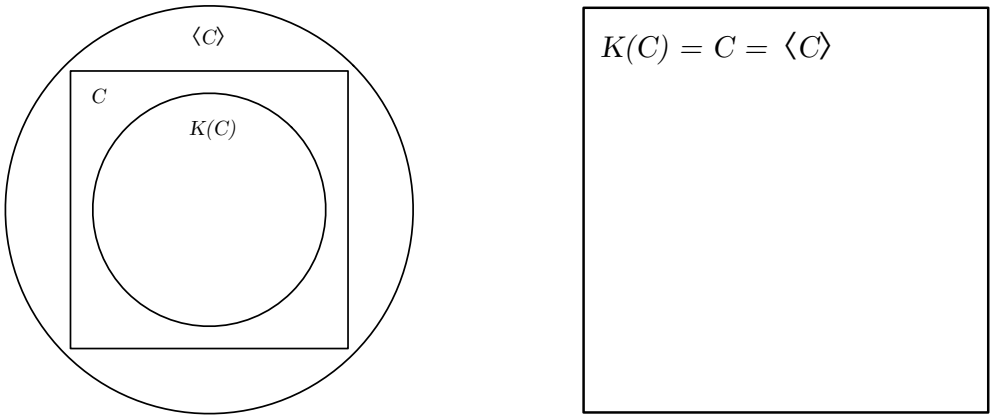
Definim l'**expansió lineal** $\langle C \rangle$ o *span* d'un codi C com un espai vectorial generat per totes les paraules del codi. És a dir, és el conjunt de totes les combinacions lineals que podem fer amb les paraules codi de C.

El **rang** r d'un codi C és la dimensió de l'expansió lineal de C.

El ***kernel*** $K(C)$ o nucli d'un codi C de longitud n està format per tots els vectors de $\mathbb{Z}_2^n$ tals que compleixen $x + C = C$. O sigui, són els vectors tals que si els hi sumem qualsevol paraula codi, obtenim de nou una paraula codi.

$$K(C) = \{x \in \mathbb{Z}_2^n \mid C = C + x\}$$

Si el codi conté la paraula zero, el *kernel* és un subcodi lineal i k és la dimensió de $K(C)$ com a subespai vectorial de $\mathbb{Z}_2^n$. Com veurem en el següent apartat, la linealitat del *kernel* ens serà molt útil per a la representació dels codis no lineals.



(a) Codis no lineals (b) Codis lineals

Figura 2.4: Relació entre codi, *kernel* i expansió lineal

Mitjançant el rang i el *kernel* podem saber si un codi és lineal o no.

Com veiem a la Figura 2.4, quan un codi és no lineal es compleix que $K(C) \subset C \subset \langle C \rangle$. En canvi quan el codi és lineal, el *kernel* coincideix amb el codi i amb l'expansió lineal: $K(C) = C = \langle C \rangle$.

2.5 Representació de codis binaris no lineals

MAGMA representa els codis binaris lineals de forma compacta amb la seva matriu generadora. En canvi, si volem emmagatzemar els no lineals hem de guardar totes les paraules codi i podem tenir problemes d'espai a memòria.

Hi ha un altre mètode per a representar codis no lineals que en molts casos serà més eficient que l'anterior. Aquest té en compte que un codi binari no lineal C sempre es pot escriure com la unió del *kernel* (part lineal del codi) i els traslladats del *kernel* (part no lineal) $K(C) + c_i$, o sigui:

$$C = \bigcup_{i=0}^{t} (K(C) + c_i),$$

on c_i és un representant qualsevol del traslladat del *kernel* corresponent, $c_0 = \mathbf{0}$ i t és el nombre de líders.

Per tant, subdividim el codi C en un seguit de traslladats (vectors) anomenats classes del *kernel* o cosets. D'aquests cosets en podem triar un representant (líder). De fet qualsevol element del coset es pot fer servir com a representant de la classe. Per tant, per a representar el codi C només hem de guardar els elements que formen el *kernel* i els líders de les seves classes (Figura 2.5).

En el pitjor dels casos, si cada coset només conté un element, perquè el *kernel* només conté un element (el vector zero), aleshores hauríem de guardar totes les paraules codi.

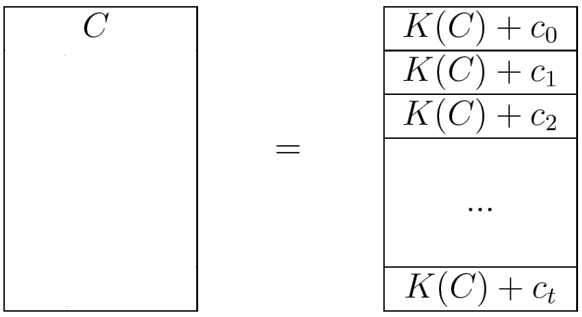


Figura 2.5: Representació d'un codi binari C utilitzant el *kernel* i els seus traslladats

El nombre de líders t es calcula en funció de la dimensió del *kernel* k i la cardinalitat M:

$$t + 1 = M/2^k$$

La dimensió del *kernel* k està directament lligada a la fragmentació del codi. Per exemple, si volem representar un codi no lineal C que conté 500 paraules codi podem fer-ho de dues maneres: emmagatzemar les 500 paraules codi o calcular el *kernel* i els líders.

k	#líders
0	$t = 499 \rightarrow 0 + 499 < 500$
1	$t = 249 \rightarrow 1 + 249 \ll 500$
2	$t = 124 \rightarrow 2 + 124 \ll 500$

Taula 2.1: Càlcul dels líders d'un codi C en funció de la dimensió del *kernel*

A la Taula 2.1 observem que quan la dimensió del *kernel* augmenta en una unitat, el nombre de líders es divideix per dos. Per tant, com més gran sigui k, el nostre codi no lineal serà més compacte perquè necessitarem menys paraules codi per a representar-lo.

Aquesta representació per a codis no lineals també és aplicable per a codis lineals. En aquest cas, $t = 0$ i llavors $C = K(C) + c_0$. Com c_0 és la paraula zero, tenim que $C = K(C)$. Així, només cal emmagatzemar els k vectors

de la base de l'espai vectorial que representa el codi lineal, o dit d'una altra manera, els k vectors de la matriu generadora del codi lineal.

Per a representar un codi no lineal C guardem els elements del *kernel* i els líders de les seves classes en una matriu. Definim el **sistema de paritat** com la matriu $(H|S)$ de mida $(n-k)$ x $(n+t)$ on H és la matriu generadora del dual del *kernel* de C $(K(C)^{\perp})$ i $S = (H \cdot c_1, H \cdot c_2, ..., H \cdot c_t)$ és el producte escalar d'H pels representants de les classes del *kernel*.

Anomenem **super dual** al codi lineal generat per la matriu $(H|S)$. La dimensió del *kernel* d'un codi no lineal $C(n, M, d)$ és $k = dim(H) - n$ i el rang és $r = k + dim(S)$. Per exemple, si tenim un codi no lineal C qualsevol de longitud $n = 6$ i cardinalitat $M = 20$ el podem expressar a través del codi super dual $C(10, 4, 3)$ generat per la matriu:

$$(H|S) = \left(\begin{array}{cccccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \end{array}\right)$$

on H és la matriu de control del *kernel* de C i S són els líders del *kernel* emmagatzemats en columnes.

Si el codi és lineal, com que no té líders no tindrà S, i quedarà representat pel dual del *kernel*, $K(C)^{\perp}$, amb matriu generadora H. Per exemple, si tenim un codi lineal D qualsevol de longitud $n = 5$ i cardinalitat $M = 8$ el podem expressar a través del codi super dual $D(5, 2, 2)$ generat per la matriu:

$$(H) = \left(\begin{array}{ccccc} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \end{array}\right)$$

La matriu generadora del codi super dual $(H|S)$ és una generalització del codi dual per al cas de codis no lineals. Tot i així, no es compleix la propietat $SuperDual(SuperDual(C)) \neq C$ mentres que per a codis lineals $(C^{\perp})^{\perp} = C$.

Definim la matriu $(H|S)$ com una generalització de la matriu de control per a codis no lineals. Podem saber si un vector x pertany al codi no lineal $x \in C$, si $H \cdot x^t = \mathbf{0}$ o bé $H \cdot x^t$ és una columna de S.

Un codi binari C super dual es representa amb una matriu de la forma $(H|S)$ però, a més, té una sèrie d'atributs on guarda informació sobre les seves propietats:

- **Kernel**: És el nucli del codi expressat com un codi lineal.
- **CosetLeaders**: Líders de les classes en forma de llista $[c_1, c_2, ..., c_t]$.
- **Length**: És la longitud de la matriu H. MAGMA ja té implementada una funció que ens dóna la longitud del codi però com que la representació de codis no lineals és diferent, ens retornaria la longitud de la matriu $(H|S)$.
- **IsLinear**: Aquest atribut té com a valor un booleà i ens permet saber si un codi és lineal o no.
- **MinimumDistance**: Com que la distància mínima de Hamming és complexa de calcular, guardem la distància de Hamming.

Existeix un altre atribut, anomenat **wordTranslation**, que no s'ha fet servir en aquest projecte, ja que només hem tingut en compte codis que contenen la paraula zero. Per al cas d'un codi no lineal C que no conté la paraula zero podem triar qualsevol paraula codi $c \in C$ i considerar el codi $C' = C + c$. El nou codi C' conté la paraula zero. Per a poder recuperar el codi C guardem el vector c a l'atribut *wordTranslation*.

Capítol 3

Anàlisi i planificació del projecte

En aquest capítol aprofundirem en els objectius del projecte que s'han comentat en la introducció. Parlarem de l'estat de l'art i de l'estudi de viabilitat previ necessari per a determinar que la seva realització és viable. També es comenten les diferències entre la planificació inicial esperada i la real.

3.1 Objectius del projecte

L'objectiu principal del projecte és dotar de major funcionalitat al paquet *BinaryCodes*, iniciat al 2007 pel projectista Víctor Ovalle [Ova08] i que van continuar al 2008 Laura Vidal [Vid09] i Joan Cuadros [Cua10]. Es tracta de construir una sèrie de funcions per a codis binaris lineals i no lineals. Quan el paquet sigui prou robust estarà disponible a través del web del departament. Per a fer-ho possible s'hauran de dur a terme les següents tasques:

- Estudiar la representació de codis binaris no lineals en MAGMA utilitzada en el paquet *BinaryCodes*.
- Estudiar els fonaments teòrics necessaris per al desenvolupament de les funcions.

- Desenvolupar en MAGMA les funcions *BinaryUnion*, *BinaryDual*, *BinaryIntersection* i *BinaryDirectSum* que ens permetran construir nous codis a partir de codis ja existents.
- Desenvolupar en MAGMA les funcions relacionades amb la distribució de pesos (*Weight Distribution*) com ara *BinaryMinimumWeight*, *BinaryMinimumWord*, etc...
- Analitzar la millora introduïda en les funcions anteriors respecte les desenvolupades per la projectista Laura Vidal [Vid09].
- Elaborar els testos que permetin assegurar que les funcions desenvolupades es comporten com esperem en tots els casos.
- Integrar els testos fets als projectes [Cua10] i [Vid09].
- Realitzar exemples en MAGMA de les funcions desenvolupades i afegir-los en un directori concret del paquet.
- Documentar en anglès funcions, testos i exemples seguint l'estil de MAGMA definit en el document *CCGStyleGuide* [dg10]. A més, a la guia d'usuari de la llibreria *BinaryCodes* ha de figurar la descripció de totes les funcions codificades i els exemples d'utilització.
- Codificar i documentar el paquet *BinaryCodes* de la manera més estructurada, organitzada i clara possible, per a futurs desenvolupadors que vulguin modificar, millorar o ampliar les funcions actuals.
- Redactar la memòria del projecte que reflecteixi el treball fet al llarg del curs.

3.2 Estat de l'art

El Departament d'Enginyeria de la Informació i de les Comunicacions (DEIC), i concretament el Grup de Combinatòria i Codificació (CCG),

permet que projectistes puguin col·laborar en el desenvolupament d'un dels seus projectes de recerca. Els membres del grup CCG treballen amb el *software* MAGMA, de llicència privada i desenvolupat pel Grup d'Àlgebra Computacional de l'Escola de Matemàtiques i Estadística de la Universitat de Sydney, que proporciona un entorn matemàtic rigorós per a resoldre problemes d'alt cost computacional en diversos àmbits com ara: àlgebra, teoria de números, geometria i combinatòria.

Per tal d'ampliar la funcionalitat de MAGMA, es pretén desenvolupar paquets que permetin superar les limitacions alhora de treballar amb codis binaris lineals i no lineals. Aquests paquets es desenvolupen en el propi llenguatge de MAGMA i estaran disponibles en la web del departament per a què els pugui fer servir tothom qui estigui interessat.

Al curs 2007-2008 el projectista Víctor Ovalle [Ova08] va iniciar la creació del paquet *BinaryCodes* on va definir l'estructura que permet guardar els codis binaris no lineals de manera que ocupin el mínim espai a memòria. Durant el curs 2008-2009 Joan Cuadros [Cua10] i Laura Vidal [Vid09] van desenvolupar una sèrie de funcions bàsiques per a codis binaris lineals i no lineals equivalents a les que estan implementades en MAGMA en el cas de codis binaris lineals. A més, en el projecte [Cua10] s'han optimitzat les funcions de creació de codis binaris no lineals definides en [Ova08].

Per tant, podríem definir l'objectiu final d'aquest projecte com l'ampliació de la funcionalitat de la llibreria *BinaryCodes* mitjançant el desenvolupament de noves funcions per al paquet.

3.3 Estudi de viabilitat

En aquest apartat es mostra la viabilitat del projecte en tots els àmbits.

3.3.1 Especificacions

L'ampliació del paquet consistirà en desenvolupar unes funcions concretes, per a codis binaris lineals i no lineals, i els testos que verifiquin que el seu comportament és l'esperat. Aquestes funcions ja existeixen a la llibreria de MAGMA que manipula codis binaris lineals. Per tant, sabem quines han de ser les entrades i sortides de cada funció.

És imprescindible que tant la codificació com la guia d'usuari del paquet segueixin l'estil de MAGMA definit a la *CCGStyleGuide* [dg10] que, com el seu nom indica, és una guia d'estil creada pel grup CCG amb l'objectiu d'unificar la codificació de testos i funcions. Com que la llibreria *BinaryCodes* s'està desenvolupant amb el treball de diversos projectistes, cal que tots segueixin el mateix estil.

3.3.2 Viabilitat tècnica

Per tal de dur a terme el projecte es requereix tenir accés a un ordinador amb el *software* MAGMA instal·lat i una llicència en vigor. El DEIC ofereix als projectistes la possibilitat de treballar amb ordinadors connectats a Internet dins l'anomenada sala de projectistes. Mitjançant aquests ordinadors es pot connectar remotament amb el servidor *macwilliams*, on està instal·lat el MAGMA amb una llicència en vigor. Tot i aquesta facilitat, també és possible treballar des de casa i accedir a l'ordinador on hi ha instal·lat MAGMA establint primer una connexió remota amb un ordinador de la sala de projectistes (Figura 3.1).

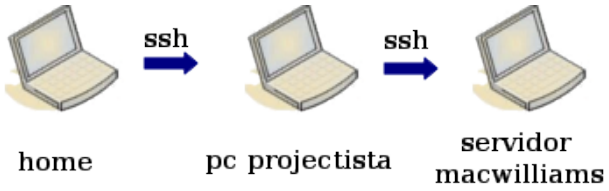


Figura 3.1: Podem establir connexió amb el servidor *macwilliams* des de casa

3.3.3 Viabilitat operativa

Per a desenvolupar les funcions de codis binaris no lineals és necessària una base de coneixements teòrics. La directora del projecte, Mercè Villanueva, proporciona els articles relacionats amb el tema així com les memòries dels projectistes que s'han mencionat a l'estat de l'art. Tota aquesta documentació és la base per a entendre els fonaments en què es basa aquest projecte. També proporciona les nocions matemàtiques necessàries sobre les funcions que s'han de construir, oferint-se a aclarir tants dubtes com calgui al seu despatx o per correu electrònic.

3.3.4 Viabilitat legal

A continuació es mostra una relació del *software* que s'utilitzarà i de les llicències necessàries:

- MAGMA és un *software* privatiu i el DEIC disposa de la llicència per a utilitzar-lo.
- El paquet *BinaryCodes* que es continuarà desenvolupant té llicència lliure *GPLv3*[1] , compatible amb la llicència de MAGMA.
- Es farà servir un sistema operatiu GNU/Linux i l'editor de text Gedit. Com que es tracta de *software* lliure amb llicència GPL, no suposen cap cost adicional.

[1]General Public License Version 3

- Per a la redacció de la memòria i documentació de la guia d'usuari s'utilitzarà LaTeX, un sistema de composició de textos amb llicència lliure *LPPL*[2] .

3.3.5 Viabilitat econòmica

Hem de tenir en compte tant la part *software* que es requereix per a fer el projecte com la part *hardware*.

Pel que fa al programari hem vist que no cal fer cap inversió ja que es disposa d'una llicència de Magma en vigor. Per a fer-nos una idea, en cas de necessitar-ne una suposaria un cost de 1.200€ vàlida dins d'un període de tres anys.

En quant al hardware, el departament ofereix un ordinador convencional per a treballar. Es tracta d'un Intel Pentium 4 a 2.00 GHz amb 2GB de RAM i sistema operatiu Fedora 5. És suficient per a connectar-nos al servidor i executar les funcions. Pel que fa al servidor, té un processador de doble nucli i 2GB de RAM.

Concepte	**Hores**	**Preu (€)**
Estudi dels fonaments teòrics	54	1.620
Elaboració de l'informe previ	12	360
Disseny de funcions	7	210
Disseny de tests	11	330
Implementació de funcions en Magma	29	870
Implementació de tests i testeig funcions	107	3.210
Implementació i documentació d'exemples	7	210
Elaboració de la memòria	177	5.310
TOTAL	404	**12.120**

Taula 3.1: Pressupost per al projecte

[2]LaTeX Project Public License

Pressupost

A la Taula 3.1 s'adjunta el pressupost estimat per al projecte tenint en compte un salari base d'uns 30 €/hora. Aquesta dada és orientativa, ja que si el contracte fos a la UAB el preu seria d'uns 25 €/hora segons conveni. En cas de ser un contracte en empresa el preu pot ser més alt.

3.3.6 Alternatives

El grup CCG està treballant actualment amb MAGMA, de llicència privada, i el paquet que s'ha d'ampliar està fet amb MAGMA. Però existeixen altres *softwares* lliures que també són útils per a resoldre aquest tipus de problemes com ara GAP[3] o SAGE[4] . Al projecte [Gas08] es va fer l'anàlisi inicial per a determinar quin *software* era més indicat per a desenvolupar el paquet *BinaryCodes*. En aquell moment es va escollir MAGMA perquè era el que millor cobria les necessitats. El motiu principal pel qual s'ha decidit continuar treballant amb MAGMA és per l'alt cost que suposaria la migració a un altre llenguatge, ara que el desenvolupament del paquet està tan avançat.

Després d'analitzar les característiques del problema a resoldre i els requisits necessaris per al seu desenvolupament en diversos àmbits, podem afirmar que el projecte és viable.

3.4 Planificació temporal del treball

Aquest projecte va començar el 15 d'octubre del 2009 amb la definició d'objectius i característiques. Segons la planificació prevista, resumida a la Taula 3.2, la data de finalització coincidint amb la data d'entrega de la memòria era el 17 de juny de 2010. Es van estimar unes 398 hores de feina,

[3]Groups, Algorithms and Programming
[4]Software for Algebra and Geometry Experimentation

suposant unes 3 hores de treball diari (exceptuant vacances i festius) entre la data d'inici i fi.

Tasca	**Termini**
Definició d'objectius i característiques	15/10/2009
Fonaments teòrics	
Estudi de la base teòrica	Octubre - novembre
Curs metodologia	Octubre - gener
Elaboració de l'informe previ	Octubre - gener
Entrega de l'informe previ	15/01/2010
Disseny i codificació	
Black-Box Tests	Novembre - maig
Disseny i codificació funcions	Desembre - maig
White-Box Tests	Desembre - maig
Elaboració de la memòria	Gener - maig
Elaboració de la presentació	Juny
Entrega de la memòria	17/06/2010

Taula 3.2: Tasques principals del projecte

La planificació inicial ha variat lleugerament degut a canvis en els objectius del projecte:

Implementació de funcions

Estava previst el desenvolupament de quatre funcions que construeixen nous codis a partir d'altres existents (*BinaryUnion*, *BinaryIntersection*, *BinaryDual* i *BinaryDirectSum*). Vam decidir afegir una altra versió de la suma directa binària, que té com a paràmetres d'entrada una llista de codis binaris, perquè en MAGMA està aquesta funció per a codis lineals. En canvi, no té versions per a la unió/intersecció de més de dos codis, tot i que ens podríem haver plantejat fer-les.

Desenvolupament de testos

S'han elaborat uns testos que comproven molts casos, i per primer cop, s'han documentat detalladament els codis utilitzats en les proves i els casos testejats. D'aquesta manera assolim un dels requisits del projecte, que siguin fàcils d'entendre per qualsevol desenvolupador que vulgui fer modificacions, millores o ampliacions de les funcions actuals. També, ens hem trobat amb una sèrie de problemes durant la construcció dels codis de prova i la fase de testeig, comentats al capítol de desenvolupament. Per aquests motius, el temps destinat a la construcció de testos ha sigut molt superior al que havíem estimat.

Elaboració de la memòria

Aquest projecte és una continuació del treball realitzat per altres projectistes, motiu pel qual vam decidir esperar a la finalització del projecte [Cua10] necessari per a conèixer l'estat inicial del paquet *BinaryCodes* que estem desenvolupant. A més, va ser un dels materials base utilitzats al capítol de fonaments teòrics. Tot i que es va posposar l'inici de la memòria gairebé dos mesos, ha donat temps d'entregar-la dins del període establert. Vam preveure tenir-la enllestida a inicis de juny de manera que donès temps de fer els últims retocs. Finalment, la data de lliurament ha sigut el 21 de juny.

A la Figura 3.2 es pot veure la planificació inicial estimada del projecte en contraposició amb la planificació real final, on l'eix temporal representa les setmanes de l'any. Els colors més clars indiquen les tasques que s'han desviat en el temps, que són bàsicament: disseny, codificació i elaboració de la memòria i de la presentació. Com es pot veure a la figura, els canvis en la planificació no han suposat un inconvenient per a complir amb les dates d'entrega.

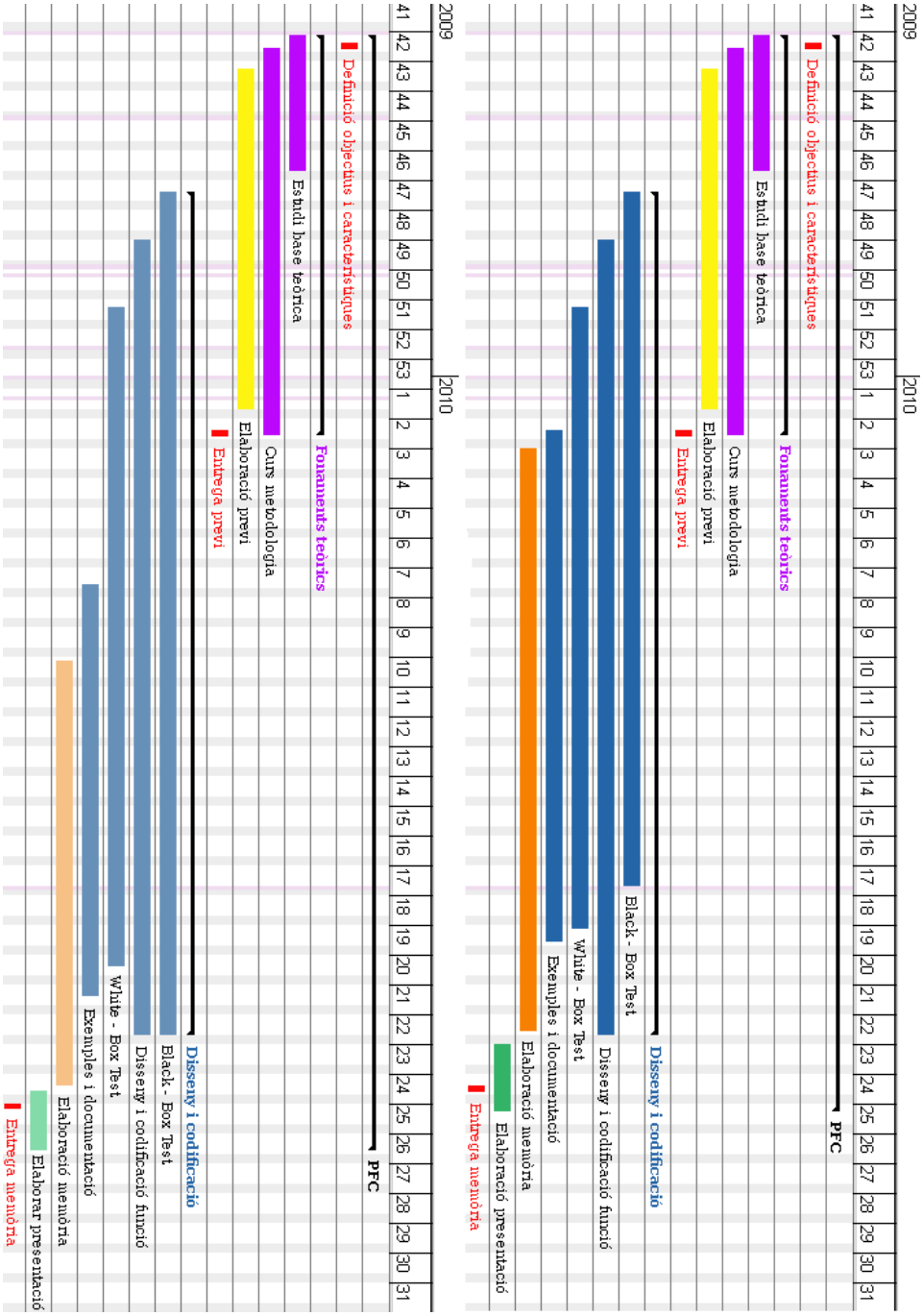


Figura 3.2: Planificació estimada i real del projecte

Capítol 4

Desenvolupament del projecte

A continuació descriurem el *software* MAGMA i les tècniques de testeig emprades per al desenvolupament del projecte. També veurem quines funcions s'han implementat i com hem aconseguit optimitzar-les per a reduir el temps d'execució. Parlarem dels problemes trobats i les solucions adoptades.

Per a il·lustrar el comportament de les funcions s'han inclòs exemples en MAGMA al llarg del capítol.

4.1 Entorn de desenvolupament

Al capítol anterior hem analitzat les eines que podem utilitzar per a la realització del projecte. Ara, definirem les característiques del *software* que farem servir.

MAGMA és un *software* de llicència privada, creat (1993) i desenvolupat pel Grup d'Àlgebra Computacional de la Universitat de Sydney. S'utilitza en els àmbits de recerca i ensenyament per a estudiar diferents àrees com ara: geometria aritmètica, combinatòria, àlgebra i teoria de codis.

Dins les seves característiques podem destacar:

- No té interfície gràfica i, de fet, tampoc fa falta. Es pot treballar sense problemes via línea de comandes.
- Disposa d'una amplia llibreria de funcions molt eficients, ja que utilitza els millors algorismes coneguts en resolució de problemes matemàtics. Aquest factor a favor seu fa que es consideri una bona eina en investigació.
- L'usuari final no pot definir els seus propis tipus de dades.
- Permet guardar les operacions fetes durant la sessió de treball.
- El seu llenguatge és molt pròxim al llenguatge formal matemàtic.
- No disposa de cap *debugger*, per tant, la programació es fa una mica feixuga. Però sí disposa d'un *profiler* que ens permet veure quines operacions són el coll d'ampolla del nostre codi.

Moltes de les funcions estàndar de MAGMA estan implementades en C, però cada vegada n'hi ha més en el llenguatge propi de MAGMA, ja que ofereix un llenguatge matemàtic formal i prou eficient. A més, l'usuari no té accés al codi font per a recompilar MAGMA per motius de privacitat. Per tant, l'única manera que té d'ampliar la seva funcionalitat és amb la creació de paquets que contenen principalment funcions intrínseques i també d'altres ordinàries.

4.2 El paquet *BinaryCodes* en MAGMA

MAGMA treballa eficientment amb codis binaris lineals representant-los de forma compacta. Ens permet tractar-los mitjançant una sèrie de funcions que té definides. Però no disposa de funcions per a treballar amb codis no lineals i els emmagatzema de forma no compacta guardant totes les paraules codi. En aquest cas, si el codi és gran podem tenir problemes d'espai a memòria.

Davant aquesta necessitat neix la idea de generar un paquet que permeti emmagatzemar i manipular codis binaris lineals i no lineals. El paquet *BinaryCodes* proporciona una representació i una sèrie de funcions que amplien la funcionalitat de MAGMA en aquest sentit. El codi font de la llibreria està format pels següents fitxers:

- *BinaryCodes_Core.m* conté les funcions per a la representació de codis binaris.
- *BinaryCodes_Extension.m* conté les funcions per a la manipulació de codis binaris.

Aquest projecte és una continuació del desenvolupament del paquet *BinaryCodes*. El fitxer *BinaryCodes_mgutierrez.m* conté les funcions implementades, que acabaran integrant-se dins *BinaryCodes_Extension.m* amb la resta de funcions.

4.3 Test de proves

Els testos són una fase molt important del procès de desenvolupament del paquet. Ens permeten garantir que les funcions creades es comporten com esperem. És per aquest motiu, que l'elaboració d'uns testos prou bons ens ha ocupat la major part del temps destinat a disseny i codificació.

En aquest projecte s'ha requerit la creació de **testos unitaris** (*Unit Test*). Aquesta tècnica té per objectiu analitzar la mínima unitat de codi testejable i determinar que el seu comportament sigui l'esperat. Existeixen dos tipus de test unitari:

- **Black-box Test**
  El test de caixa negra es crea abans de codificar la funció. Es basa en els requeriments i la funcionalitat. La idea és que desconeixem la seva implementació però coneixem quina ha de ser la sortida per a cada entrada, de manera que, per a cada possible entrada comprova que la sortida sigui l'esperada.

- **White-box Test**
  El test de caixa blanca es crea després de codificar la funció. Coneixem la seva implementació, per tant, ara podem filar més prim i analitzar totes les branques possibles. Per a cada condició comprova que donada una entrada la sortida és l'esperada.

Per a cada funció s'han generat els següents fitxers:

- *nomfuncio_BB_test.m* conté els diferents testos de caixa negra.
- *nomfuncio_WB_test.m* conté els diferents testos de caixa blanca.
- *nomfuncio_test_data.m* conté els codis de prova utilitzats en els testos de caixa negra i blanca.

Tant les capçaleres dels testos, com el plantejament dels diferents casos s'ha fet seguint l'estil definit en el document *CCGStyleGuide* [dg10].

4.4 Implementació

Per tal d'ampliar la funcionalitat del paquet *BinaryCodes* s'han creat una sèrie de funcions. La manera de seguir fidelment l'estàndar definit a la guia d'estil, i garantir que els testos són el més robusts possibles, ha consistit en dissenyar i codificar les funcions d'una en una. Amb més detall, els passos han sigut:

- Estudi dels conceptes matemàtics relatius a la funció entenent quina sortida tindrem per a diferents entrades.
- Construcció dels codis de prova, que utilitzarem en la fase de testeig, on per a cada entrada coneixem la sortida.
- Creació dels *Black-Box Tests*.
- Codificació de la funció.
- Testeig de la funció implementada amb els *Black-Box Tests*.
- Creació dels *White-Box Tests*.
- Testeig de la funció implementada amb els *White-Box Tests*.
- Creació dels exemples que il·lustren les propietats bàsiques de la funció. Aquests exemples s'han de documentar al *help* de MAGMA.

Per a algunes funcions s'han desenvolupat diverses versions, i s'ha hagut de comprovar que es continués validant els testos. A l'apartat de resultats veurem la comparació de temps d'execució obtinguts, fet decisiu per a triar la implementació més òptima de la nostra funció.

4.4.1 Funcions desenvolupades

A continuació descriurem les funcions que s'han codificat. Totes utilitzen la representació compacta per a codis binaris no lineals a través del *kernel* i la llista de líders definida al paquet *BinaryCodes*.

BinaryUnion(C,D)

Donats dos codis binaris C i D de longitud n retorna la unió. El codi resultant tindrà longitud n i estarà format per totes les paraules codi c de C i d de D, tals que no es repeteix cap.

Exemple 4.1 ***Unió de dos codis binaris no lineals C i D***
Construim dos codis binaris no lineals aleatoris, de longitud 5 i cardinalitat 4. Aquests codis els utilitzarem al llarg del capítol. Mostrem les seves paraules codi amb la comanda *BinarySet* i observem que, tant C com D poden generar les paraules codi (00000) i (10000).

```
> C := BinaryRandomCode(5,4);
> D := BinaryRandomCode(5,4);
> BinarySet(C);
{
    (0 0 0 0 0),
    (1 0 0 0 0),
    (0 1 0 0 0),
    (1 1 0 0 0)
}
> BinarySet(D);
{
    (0 0 0 0 0),
    (1 0 0 0 0),
    (0 1 0 1 1),
    (1 1 0 1 1)
}
> U := BinaryUnion(C,D);
> BinarySet(U);
{
    (0 0 0 0 0),
    (1 0 0 0 0),
    (0 1 0 0 0),
    (1 1 0 0 0),
    (0 1 0 1 1),
    (1 1 0 1 1)
}
```

El codi U, de longitud 5, és la unió dels codis C i D. Veiem que U està format per totes les paraules codi de C i D , de manera que no es repeteix cap.

Podem expressar els codis d'entrada de la següent manera:

$$C = \bigcup_{i=0}^{r} (K_c + a_i), \qquad D = \bigcup_{j=0}^{s} (K_d + b_j)$$

El mínim *kernel* K que podem garantir d'entrada, en la unió de C i D, és la intersecció dels *kernels* de C i D, $K = K_c \cap K_d$. Per tant, en funció d'aquest *kernel* parcial K, hem de considerar una nova llista de líders L'_c i L'_d, per a C i D respectivament, i comprovar quins d'ells formaran part de la unió, o sigui evitant elements repetits.

Per a calcular la nova llista de líders respecte de $K = K_c \cap K_d$ hem de dividir els *kernels* dels codis C i D en cosets respecte K. Siguin K_cLids i K_dLids la llista de líders dels cosets dels codis C i D respecte K:

$$K_cLids = \{\mathbf{0}, k_{c1}, ..., k_{cp}\}, \qquad K_dLids = \{\mathbf{0}, k_{d1}, ..., k_{dq}\}$$

on $p+1$ i $q+1$ són el nombre de cosets de K_c i K_d respecte de K. La nova llista de líders L'_c respecte K per al codi C és la formada pels cosets K_cLids de K en combinació amb els líders L_c del codi C:

$$L'_c = \{a + k_c,\ a \in L_c,\ k_c \in K_cLids\}$$

on $L_c = \{\mathbf{0}, a_1, ..., a_r\}$ són els líders de C respecte K_c. De manera similiar, la nova llista de líders L'_d respecte K per al codi D és la formada pels cosets K_dLids de K en combinació amb els líders L_d del codi D:

$$L'_d = \{b + k_d,\ b \in L_d,\ k_d \in K_dLids\}$$

on $L_d = \{\mathbf{0}, b_1, ..., b_s\}$ són els líders de D respecte K_d. Finalment, la nova llista de líders L respecte de K estarà formada per tots els elements de L'_c i els elements de L'_d que no estan a C.

Després d'aquest procés, s'ha de veure si podem trobar un codi el més compacte possible. Hem calculat el mínim *kernel* $K = K_c \cap K_d$ que podem garantir d'entrada i la nova llista de líders L respecte K, però potser es pot representar aquest codi unió $C \cup D$ amb un *kernel* més gran i, per tant, amb menys líders.

Hi ha una funció dins el paquet *BinaryCodes* que ens permet construir codis binaris a partir de diferents paràmetres d'entrada. Per exemple, a partir d'una llista de vectors sobre $\mathbb{Z}_2^n$ o d'un codi binari lineal de MAGMA entre d'altres. Aquesta funció s'anomena *BinaryCode* i la cridarem amb el nostre *kernel* i líders parcials perquè ens retorni el codi més compacte possible (si es pot optimitzar més). El codi obtingut serà el que retornarem com a unió dels codis binaris C i D.

Cal tenir en compte que:

- La unió de dos codis lineals pot ser un codi no lineal.
- La unió de dos codis no lineals pot ser un codi lineal.

Per tant, com que la linealitat del codi de sortida no depèn dels codis d'entrada, s'han de tractar tots els casos per igual. L'única diferència que es farà és:

- Si C i D són lineals, la nova llista de líders L respecte K estarà formada pels cosets de K dels codis C i D, o sigui per K_cLids i K_dLids.
- Si com a mínim un dels codis C o D és no lineal, la nova llista de líders L respecte K estarà formada per tots els elements de L'_c i els elements de L'_d que no estan a C.

Problemes trobats

Es van detectar errors en dues funcions del paquet *BinaryCodes* desenvolupades en projectes anteriors.

Vam tenir problemes durant la construcció dels codis de prova. Utilitzàvem la

funció *IsBinarySubset* per a comprovar si un codi és subconjunt d'un altre. Al detectar l'error, es van haver de refer els codis de prova utilitzant una funció auxiliar.

Durant l'execució dels testos vam detectar un error a la funció *IsBinaryEqual*, que ens serveix per a comparar la sortida esperada amb l'obtinguda en cridar *BinaryUnion* amb els codis de prova. Tot i que el nombre de líders en ambdues sortides era diferent, *IsBinaryEqual* ens deia que els codis de sortida eren iguals.

Resoldre aquests imprevistos va suposar una càrrega de treball adicional i, per tant, va afectar a la planificació.

Optimitzar la implementació

Amb l'objectiu de reduir el temps d'execució s'han desenvolupat tres versions. Primer vam comparar què era més eficient en el càlcul dels líders, si utilitzar bucles for o seqüències.

- **Versió1**:
  Genera els nous líders amb dos bucles for.

```
unionLeaders := [Kc+Lc : Kc in cKernelCosets, Lc in cLeaders];
for Kd in dKernelCosets do
    for Ld in dLeaders do
        KLdLeaders := Kd+Ld;
        if IsNotInBinaryCode(C,KLdLeaders) then
            Append(~unionLeaders, KLdLeaders);
        end if;
    end for;
end for;
```

- **Versió2**:
  Genera els nous líders utilitzant seqüències fent el codi més elegant i compacte.

```
unionLeaders := [Kc+Lc : Kc in cKernelCosets, Lc in cLeaders] cat
                [Kd+Ld : Kd in dKernelCosets, Ld in dLeaders |
                         IsNotInBinaryCode(C,Kd+Ld)];
```

Després de construir aquestes dues versions, vam pensar que seria interessant comprovar si l'ordre dels paràmetres d'entrada era un factor decisiu en l'eficiència de l'algorisme. Com que Versió2 és més òptima, la vam millorar a partir de les conclusions extretes de les taules on:

(a) Si C i D tenen la mateixa cardinalitat (nombre de paraules codi), els resultats milloren quan el codi C té **menys** líders que el codi D ($\#L_c < \#L_d$). Per exemple, si tenim dos codis C i D de longitud 1000 i cardinalitat 500. Si C té 499 líders i D en té 124, com que D té menys líders que C, la funció serà més eficient si intercanviem l'ordre dels codis *BinaryUnion(D,C)*.

(b) Si C i D tenen diferent cardinalitat, els resultats milloren quan el codi C és el de **major** cardinalitat ($\#C > \#D$).
Per exemple, si tenim dos codis C i D de longitud 1000 i cardinalitats 150 i 500 respectivament, com que D té una cardinalitat major que C, la funció serà més eficient si intercanviem l'ordre dels codis *BinaryUnion(D,C)*.

- **Versió3**:
  Millora de Versió2. Intercanvia l'ordre dels paràmetres d'entrada segons les condicions (a) i (b).

Al capítol de resultats es mostren les taules amb els temps d'execució obtinguts per a cada versió i la justificació de la versió escollida per la seva eficiència (Versió3).

BinaryIntersection(C,D)

Donats dos codis binaris C i D de longitud n retorna la intersecció. El codi resultant tindrà longitud n i estarà format per totes les paraules codi c de C i d de D tals que $c \in D$ i $d \in C$.

Exemple 4.2 ***Intersecció de dos codis binaris no lineals C i D***
Els codis C i D tenen dues paraules codi en comú: (00000) i (10000)

```
> BinarySet(C);
```

```
{
    (0 0 0 0 0),
    (1 0 0 0 0),
    (0 1 0 0 0),
    (1 1 0 0 0)
}
> BinarySet(D);
{
    (0 0 0 0 0),
    (1 0 0 0 0),
    (0 1 0 1 1),
    (1 1 0 1 1)
}
> I := BinaryIntersection(C,D);
> BinarySet(I);
{
    (0 0 0 0 0),
    (1 0 0 0 0)
}
```

El codi I, de longitud 5, és la intersecció dels codis C i D. Veiem que I està format per totes les paraules codi comunes a C i D.

La funció intersecció té similituds i diferències amb la funció unió. En aquest cas, el mínim *kernel* K que podem garantir d'entrada, en la intersecció de C i D, també és la intersecció dels *kernels* de C i D, $K = K_c \cap K_d$. En funció d'aquest *kernel* parcial K, hem de considerar una nova llista de líders L'_c de C respecte K i triar els cosets tals que el seu líder estigui també a D.

Per a calcular la nova llista de líders respecte de $K = K_c \cap K_d$ hem de dividir el *kernel* del codi C en cosets respecte K. Sigui K_cLids la llista de líders dels cosets del codi C respecte K:

$$K_cLids = \{\mathbf{0}, k_{c1}, ..., k_{cp}\}$$

on $p + 1$ és el nombre de cosets de K_c respecte de K. La nova llista de

líders L'_c respecte K per al codi C és la formada pels cosets K_cLids de K en combinació amb els líders L_c del codi C:

$$L'_c = \{a + k_c,\ a \in L_c,\ k_c \in K_cLids\}$$

on $L_c = \{\mathbf{0}, a_1, ..., a_r\}$ són els líders de C respecte K_c. Finalment, la nova llista de líders L respecte K estarà formada per tots els elements de L'_c tals que estiguin a D.

Després d'aquest procés, s'ha de veure si podem trobar un codi el més compacte possible. Hem calculat el mínim *kernel* $K = K_c \cap K_d$ que podem garantir d'entrada i la nova llista de líders L respecte K, però potser es pot representar aquest codi unió $C \cap D$ amb un *kernel* més gran i, per tant, amb menys líders. Per tant, el codi intersecció serà l'obtingut en cridar la funció *BinaryCode* amb el *kernel* i líders parcials. Aquesta funció calcula el major *kernel* possible i els seus líders associats i ens retorna el codi més compacte que pot trobar.

La principal diferència, respecte la funció unió, és que la linealitat dels codis d'entrada determina la linealitat del codi de sortida. Per tant, els casos es tracten per separat de manera que:

- Si C i D són lineals, el codi intersecció $C \cap D$ serà lineal.
- En cas contrari, el codi intersecció en general serà no lineal i estarà format per un *kernel* K i la llista de líders L associada a K. Tot i que ens podem trobar amb casos concrets on la intersecció de dos codis no lineals sigui lineal. L'exemple més clar és quan $C \cap D$ està format pel vector tot zeros. Un altre exemple seria el cas que la intersecció conté el vector tot zeros i el vector tot uns, o sigui $\{0, 1\}$, ja que aleshores també és lineal.

Problemes trobats

Durant la fase de testeig vam tenir problemes amb els codis de prova *Zero* i *Universe*. Són codis binaris lineals, per tant, no tenen assignat cap atribut com en el cas dels codis binaris no lineals.

EXEMPLE 4.3 ***Codis binaris Zero i Universe***

El codi Zero, de longitud 10, està format per la paraula zero (vector de 10 coordenades).

```
> Z := ZeroCode(GF(2),10);
> BinarySet(Z);
{
    (0 0 0 0 0 0 0 0 0 0)
}
```

El codi Universe, de longitud 10, està format per totes les paraules codi possibles (2^{10} vectors de 10 coordenades).

```
> Unv := UniverseCode(GF(2),10);
> #BinarySet(Unv);
1024
```

La primera execució dels testos, tant de caixa negra com de caixa blanca, finalitzava amb èxit. Però en execucions successives, es produïen errors quan es testejava un codi *Zero* o *Universe* perquè MAGMA els havia convertit en codis super duals. Ens vam adonar perquè de sobte, els codis tenien atributs. Aquest problema va suposar una càrrega de treball important, ja que va ser difícil trobar-ne l'origen. Vam executar cada cas del test per separat fins a detectar que després de l'execució d'un determinat cas, els codis *Zero* i *Universe* es guardaven com un codi super dual.

La solució consisteix en eliminar l'atribut que ens indica si un codi és lineal o no. Per tant, fem un *delete Zero'IsLinear* i *delete Universe'IsLinear* abans del test per al codi *Zero* i *Universe* respectivament.

Les funcions unió i dual també testejen casos amb aquests codis. Es va observar que en execucions sucessives es produïa el mateix error que a la

funció intersecció. Suprimir l'atribut *IsLinear* va solucionar el problema en tots dos casos.

Optimitzar la implementació

Amb l'objectiu de reduir el temps d'execució s'han desenvolupat tres versions. Com en el cas de la unió, primer vam implementar dues versions per a comparar qué és més eficient en el càlcul dels líders: utilitzar bucles for o seqüències.

- **Versió1**:
  Genera els nous líders amb dos bucles for.

```
for Kc in cKernelCosets do
    for Lc in cLeaders do
        KLcLeaders := Kc+Lc;
        if IsInBinaryCode(D,KLcLeaders) then
            Append(~interLeaders, KLcLeaders);
        end if;
    end for;
end for;
```

- **Versió2**:
  Genera els nous líders utilitzant seqüències fent el codi més elegant i compacte.

```
interLeaders := [Kc+Lc : Kc in cKernelCosets, Lc in cLeaders |
                         IsInBinaryCode(D,Kc+Lc)];
```

Després de construir aquestes dues versions, vam pensar que seria interessant comprovar si l'ordre dels paràmetres d'entrada era un factor decisiu en l'eficiència de l'algorisme. Com que Versió2 és més òptima, la vam millorar a partir de les conclusions extretes de les taules on:

(a) Si C i D tenen la mateixa cardinalitat (nombre de paraules codi), els resultats milloren quan el codi C té **més** líders que el codi D ($\#L_c > \#L_d$). Per exemple, si tenim dos codis C i D de longitud 1000 i cardinalitat 500. Si C té 124 líders i D en té 499, com que D té més líders que C, la funció serà més eficient si intercanviem l'ordre dels codis *BinaryUnion(D,C)*.

(b) Si C i D tenen diferent cardinalitat, els resultats milloren quan el codi C és el de **menor** cardinalitat ($\#C < \#D$).
Per exemple, si tenim dos codis C i D de longitud 1000 i cardinalitats 500 i 150 respectivament, com que D té una cardinalitat menor que C, la funció serà més eficient si intercanviem l'ordre dels codis *BinaryUnion(D,C)*.

- **Versió3**:
  Millora de Versió2. Intercanvia l'ordre dels paràmetres d'entrada segons les condicions (a) i (b).

Al capítol de resultats es mostren les taules amb els temps d'execució obtinguts per a cada versió i la justificació de la versió escollida per la seva eficiència (Versió3).

BinaryDual(C)

Donat un codi binari d'entrada C de longitud n ens retorna el seu dual $C^{\perp}$, format per tots els vectors de $\mathbb{Z}_2^n$ que són ortogonals a totes les paraules del codi C. Dit d'una altra manera, el producte escalar de les paraules codi de C per la matriu generadora del seu dual donarà zero.

Exemple 4.4 ***Dual d'un codi binari no lineal C***
Calculem el dual del codi C. Si multipliquem la matriu generadora del seu dual HDu per les paraules codi de C (Cpc) ens dóna zero.

```
> Du := BinaryDual(C);
> HDu := GeneratorMatrix(Du);
> Cpc := Setseq(BinarySet(C));
> Cpc;
[
    (0 0 0 0 0),
    (1 0 0 0 0),
    (0 1 0 0 0),
    (1 1 0 0 0)
]
```

```
> [Cpc[i]*Transpose(HDu): i in [1..4]];
[
    (0 0 0),
    (0 0 0),
    (0 0 0),
    (0 0 0)
]
```

En canvi, el producte de HDu per un vector v que no pertany al codi C és diferent de zero.

```
> v:=VectorSpace(GF(2),5)![1,1,1,1,1];
> v*Transpose(HDu);
(1 1 1)
```

En aquesta funció distingim principalment els següents casos:

(a) Si C és lineal retornem dual del codi C.
(b) Si C és no lineal retornem el dual de l'*span* del codi C, és a dir, el dual del codi binari lineal generat per les paraules codi de C.

En qualsevol cas, la funció sempre retornarà un codi lineal.

Construcció de nous codis a partir de codis existents

S'han desenvolupat dues funcions molt semblants que permeten construir nous codis a partir d'altres codis lineals o no lineals.

BinaryDirectSum(C,D)

Donats dos codis binaris C i D de longituds n_1 i n_2 respectivament, construeix la suma directa $C \oplus D$. El codi resultant, de longitud $(n_1 + n_2)$, estarà format per tots els vectors de la forma (u, v) on $u \in C$ i $v \in D$.

Exemple 4.5 ***Suma directa de dos codis binaris no lineals C i D***
El codi S, de longitud 10, és la suma directa dels codis C i D. Veiem que S està format per totes les combinacions de paraules codi de C i D.

```
> BinarySet(C);
{
    (0 0 0 0 0),
    (1 0 0 0 0),
    (0 1 0 0 0),
    (1 1 0 0 0)
}
> BinarySet(D);
{
    (0 0 0 0 0),
    (1 0 0 0 0),
    (0 1 0 1 1),
    (1 1 0 1 1)
}
> S := BinaryDirectSum(C,D);
> BinarySet(S);
{
    (1 1 0 0 0 1 0 0 0 0),
    (0 0 0 0 0 0 1 0 1 1),
    (1 0 0 0 0 0 1 0 1 1),
    (0 1 0 0 0 0 1 0 1 1),
    (0 0 0 0 0 1 1 0 1 1),
    (1 1 0 0 0 0 1 0 1 1),
    (1 0 0 0 0 1 1 0 1 1),
    (0 1 0 0 0 1 1 0 1 1),
    (1 1 0 0 0 1 1 0 1 1),
    (0 0 0 0 0 0 0 0 0 0),
    (1 0 0 0 0 0 0 0 0 0),
    (0 1 0 0 0 0 0 0 0 0),
    (0 0 0 0 0 1 0 0 0 0),
    (1 1 0 0 0 0 0 0 0 0),
    (1 0 0 0 0 1 0 0 0 0),
    (0 1 0 0 0 1 0 0 0 0)
}
```

Podem expressar els codis d'entrada de la següent manera:

$$C = \bigcup_{i=0}^{r} (K_c + a_i), \qquad D = \bigcup_{j=0}^{s} (K_d + b_j)$$

Per al càlcul del *kernel*, aprofitant la seva propietat de linealitat, podem utilitzar la funció pròpia de MAGMA per a codis lineals. Així, el nou kernel serà $K = DirectSum(K_c, K_d)$.

En quant als líders els calculem de la següent forma:

$L = \{(a_i, b_j),\ a_i \in L_c = \{\mathbf{0}, a_1, ..., a_r\},\ b_j \in L_d = \{\mathbf{0}, b_1, ..., b_s\}\}$

BinaryDirectSum(Q)

La idea és la mateixa que en la funció anterior però tenint com a paràmetre d'entrada una llista de com a mínim dos codis binaris. Donada una llista $Q = [C_1, ..., C_r]$ la suma directa dels codis C_i, $1 \leq i \leq r$ està formada pels vectors de longitud $(n_1 + n_2 + \cdots + n_r)$ de la forma $(u_1, ..., u_r)$ on $u_i \in C_i$, $1 \leq i \leq r$.

S'han desenvolupat diverses versions per a optimitzar el rendiment de la funció:

- **Versió1**:
  Fa la suma directa de forma recursiva. La funció principal comprova que els paràmetres siguin codis binaris i crida a un procediment auxiliar que construeix el nou codi fent crides recursives sobre la llista inicial.

- **Versió2**:
  Resol la suma directa d'una llista de codis binaris aprofitant el llenguatge de MAGMA. Utilitzant seqüències podem calcular *kernel* i líders en poques línies, fent el codi més net i entenedor.

A l'apartat de resultats veurem com s'ha triat la versió més òptima (Versió1)

comparant els temps d'execució obtinguts amb cadascuna.

Per a veure una definició més detallada de les funcions així com alguns exemples, es pot consultar el *help* de MAGMA a l'apèndix A.

Capítol 5

Anàlisi de resultats

Tot seguit analitzarem l'eficiència de les diferents versions que s'han fet per a les funcions codificades. L'objectiu és justificar l'opció triada en cada cas.

5.1 Com triar la millor versió

Per tal d'escollir la versió més eficient s'ha utilitzat la comanda *time* de MAGMA que ens retorna el temps que triguen en executar-se un conjunt d'instruccions o, en el nostre cas, la crida a una funció. Concretament, ens imprimeix a consola els segons que triga la CPU a resoldre les instruccions.

$>$ *time STATEMENT*

Per a donar una aproximació més fidel a les taules de resultats mostrem la mitjana d'executar cinc cops *time crida_funcio*.

5.2 Taules de resultats

Els resultats que es mostren a les següents taules s'han elaborat amb codis binaris aleatoris construits mitjançant la comanda *BinaryRandomCode(n,M)*,

on n és la longitud del codi i M la cardinalitat (paraules del codi). Incrementant els valors de n i M s'han generat codis prou grans que ens permeten veure una diferència en els temps de còmput entre versions.

Els codis aleatoris construits s'expressen a les taules mitjançant el nombre d'elements del *kernel* i el nombre de líders respecte d'aquest *kernel* (K i L respectivament). La columna Versió2 indica la diferència en segons respecte Versió1 i la columna Versió3 la diferència en segons respecte Versió2 (positiu si incrementa i negatiu si és menor). En el cas de les funcions unió i intersecció, la primera columna (Op) indica l'ordre dels paràmetres d'entrada. Per exemple, a la Taula 5.2 la primera fila mostra els temps d'execució obtinguts en fer la unió dels codis C i D ($C \cup D$). La segona fila mostra el temps d'execució dels mateixos codis però en fer la unió de D i C ($D \cup C$).

5.2.1 BinaryUnion(C, D)

Per a aquesta funció s'han implementat tres versions. Versió1 fa el càlcul dels líders utilitzant bucles for i Versió2 utilitza seqüències. Com que la segona versió és més eficient, es va millorar (Versió3) fent un intercanvi dels codis d'entrada C i D, tenint en compte el seu tamany i el nombre de líders.

Per a poder escollir la millor versió hem analitzat tres casos:

1. Unió de codis iguals (Taula 5.1).
2. Unió de codis del mateix tamany (Taula 5.2).
3. Unió de codis amb cardinalitat diferent (Taula 5.3).

En general, a les tres taules podem veure que les versions que utilitzen seqüències són més eficients. A mesura que els codis són més grans (quan tenim valors de n i M més grans), la diferència entre versions és més evident. Això ho podem comprovar si observem les taules 5.2 i 5.3. Per exemple, si ens fixem en la darrera línea de la Taula 5.2, per a codis de longitud 3000 i 2000 paraules codi la segona versió és 1,19 segons més ràpida que la primera.

A la Taula 5.3 pel cas de dos codis de longitud 3000, tals que el primer té 1000 paraules codi i el segon en té 2500, Versió2 és 1,82 segons més ràpida que Versió1.

Per a comparar la millora que aporta Versió3 respecte Versió2 analitzarem cada cas per separat.

Cas 1: Unió de codis iguals

Si C i D són el mateix codi ($C \cup C$) Versió3 mai farà intercanvi de codis però farà la comprovació igualment. Versió2 trigarà una mica menys en executar-se però aquesta diferència no és significativa. En aquest cas, podem dir que ambdues versions són eficients.

Op	n	M	K_c	L_c	K_d	L_d	Versió1	Versió2	Versió3
$C \cup C$	2000	1000	8	124	8	124	0,633	−0,001	**−0,002**
$C \cup C$	2000	1000	2	499	2	499	7,974	−0,104	**−0,002**
$C \cup C$	3000	2000	16	124	16	124	1,044	**−0,006**	+0,002
$C \cup C$	3000	2000	4	499	4	499	12,074	**−0,164**	+0,006

Taula 5.1: Unió de codis iguals

Cas 2: Unió de codis del mateix tamany

Siguin $\#L_c$ i $\#L_d$ el nombre de líders associats al *kernel* de C i D respectivament. La millora incorporada a Versió3 consisteix en intercanviar els codis d'entrada si ens trobem amb codis binaris de mateixa cardinalitat on $\#L_c > \#L_d$.

Quan $\#L_c < \#L_d$ (files on Op és $C \cup D$) Versió3 no fa intercanvi de codis. En aquest cas, no haurà de canviar l'ordre dels codis però ho comprovarà igualment. Versió2 és una mica més ràpida perquè no fa aquesta comprovació.

Quan $\#L_c > \#L_d$ (files on Op és $D \cup C$) Versió3 intercanvia els codis. En aquest cas sempre és molt millor que Versió2.

Op	n	M	K_c	L_c	K_d	L_d	Versió1	Versió2	Versió3
$C \cup D$	1000	500	4	124	1	499	**3,960**	+0,010	−0,008
$D \cup C$	1000	500	1	499	4	124	9,276	−0,136	**−5,180**
$C \cup D$	2000	1000	8	124	2	499	16,592	**−0,018**	+0,080
$D \cup C$	2000	1000	2	499	8	124	36,814	−0,530	**−19,656**
$C \cup D$	3000	2000	16	124	4	499	60,682	**−0,228**	+0,026
$D \cup C$	3000	2000	4	499	16	124	120,430	−1,190	**−58,834**

Taula 5.2: Unió de codis del mateix tamany

Cas 3: Unió de codis amb cardinalitat diferent

En la unió de codis amb cardinalitat diferent, el nombre de líders de cadascun no influeix en el temps d'execució.

Quan $\#C > \#D$ (files on Op és $C \cup D$) Versió3 no fa intercanvi de codis. Tot i que Versió2 no ha de fer la comprovació, Versió3 és més eficient.

Quan $\#C < \#D$ (files on Op és $D \cup C$) Versió3 intercanvia els codis. En aquest cas sempre és molt millor que Versió2.

Op	n	M_c	K_c	L_c	M_d	K_d	L_d	Versió1	Versió2	Versió3
$C \cup D$	1000	500	4	124	150	2	74	1,474	−0,004	**−0,001**
$D \cup C$	1000	150	2	74	500	4	124	2,120	−0,008	**−0,642**
$C \cup D$	1000	500	4	124	150	1	149	1,458	−0,004	**−0,002**
$D \cup C$	1000	150	1	149	500	4	124	3,106	−0,014	**−1,632**
$C \cup D$	2000	1000	4	249	500	4	124	12,100	−0,064	**−0,032**
$D \cup C$	2000	500	4	124	1000	4	249	12,708	−0,032	**−0,636**
$C \cup D$	2000	1000	4	249	500	1	499	12,078	**−0,088**	+0,018
$D \cup C$	2000	500	1	499	1000	4	249	32,768	−0,436	**−20,290**
$C \cup D$	3000	2500	4	624	1000	2	499	79,460	−0,710	**−0,110**
$D \cup C$	3000	1000	2	499	2500	4	624	133,160	−1,820	**−52,560**

Taula 5.3: Unió de codis amb cardinalitat diferent

Per tant, hem vist que la implementació que té en compte el tamany dels codis i el nombre de líders (Versió3) és la millor solució per a la unió de dos codis binaris.

5.2.2 BinaryIntersection(C, D)

Per a aquesta funció s'han implementat tres versions. Versió1 fa el càlcul dels líders utilitzant bucles for i Versió2 utilitza seqüències. Com que la segona versió és més eficient, es va millorar (Versió3) fent un intercanvi dels codis d'entrada C i D, tenint en compte el seu tamany i el nombre de líders.

Per a poder escollir la millor versió hem analitzat tres casos:

1. Intersecció de codis iguals (Taula 5.4).
2. Intersecció de codis del mateix tamany (Taula 5.5).
3. Intersecció de codis amb cardinalitat diferent (Taula 5.6).

En general, a les tres taules podem veure que les versions que utilitzen seqüències són més eficients. A mesura que els codis són més grans la diferència entre versions és més evident. Per exemple, a la darrera fila de la Taula 5.6 veiem que per a dos codis de longitud 3000, on el primer té 2500 paraules codi i el segon en té 1000, Versió2 és 1,764 segons més ràpida que Versió1.

Per a comparar la millora que aporta Versió3 respecte Versió2 analitzarem cada cas per separat.

Cas 1: Intersecció de codis iguals
Si C i D són el mateix codi ($C \cap C$) Versió3 mai farà intercanvi de codis però farà la comprovació igualment. En general obtenim els millors temps d'execució amb Versió3, encara que les diferències no són significatives.

Op	n	M	K_c	L_c	K_d	L_d	Versió1	Versió2	Versió3
$C \cap C$	2000	1000	8	124	8	124	0,632	**−0,004**	+0,002
$C \cap C$	2000	1000	2	499	2	499	7,988	−0,106	**−0,006**
$C \cap C$	3000	2000	16	124	16	124	1,036	−0,004	**−0,002**
$C \cap C$	3000	2000	4	499	4	499	12,128	−0,020	**−0,008**

Taula 5.4: Intersecció de codis iguals

Cas 2: Intersecció de codis del mateix tamany

Siguin $\#L_c$ i $\#L_d$ el nombre de líders associats al *kernel* de C i D respectivament. La millora incorporada a Versió3 consisteix en intercanviar els codis d'entrada si ens trobem amb codis binaris de mateixa cardinalitat on $\#L_c < \#L_d$.

Quan $\#L_c > \#L_d$ (files on Op és $C \cap D$) Versió3 no fa intercanvi de codis. Tot i fer la comprovació del nombre de líders de C i D sense haver de canviar l'ordre, obtenim uns millors temps d'execució.

Quan $\#L_c < \#L_d$ (files on Op és $D \cap C$) Versió3 intercanvia els codis. En aquest cas sempre és molt millor que Versió2.

Op	n	M	K_c	L_c	K_d	L_d	Versió1	Versió2	Versió3
$C \cap D$	1000	500	1	499	4	124	1,902	**−0,028**	+0,006
$D \cap C$	1000	500	4	124	1	499	7,118	−0,118	**−5,118**
$C \cap D$	2000	1000	2	499	8	124	7,784	−0,034	**−0,002**
$D \cap C$	2000	1000	8	124	2	499	27,916	−0,458	**−19,706**
$C \cap D$	3000	2000	4	499	16	124	24,812	−0,020	**−0,076**
$D \cap C$	3000	2000	16	124	4	499	84,500	−0,320	**−59,472**

Taula 5.5: Intersecció de codis del mateix tamany

Cas 3: Intersecció de codis amb cardinalitat diferent

En la intersecció de codis amb cardinalitat diferent, el nombre de líders de cadascun no influeix en el temps d'execució.

Quan $\#C < \#D$ (files on Op és $C \cap D$) Versió3 no fa intercanvi de codis. Encara que Versió2 no ha de fer la comprovació, en general Versió3 és més eficient.

Quan $\#C > \#D$ (files on Op és $D \cap C$) Versió3 intercanvia els codis. En aquest cas sempre és molt millor que Versió2.

Per tant, hem vist que la implementació que té en compte el tamany dels codis i el nombre de líders (Versió3) és la millor solució per a la intersecció de dos codis binaris.

Op	n	M_c	K_c	L_c	M_d	K_d	L_d	Versió1	Versió2	Versió3
$C \cap D$	1000	150	2	74	500	4	124	0,566	**−0,004**	+0,002
$D \cap C$	1000	500	4	124	150	2	74	1,214	−0,002	**−0,650**
$C \cap D$	1000	150	1	149	500	4	124	0,568	−0,002	**−0,002**
$D \cap C$	1000	500	4	124	150	1	149	2,218	−0,024	**−1,630**
$C \cap D$	2000	500	4	124	1000	4	249	7,158	−0,066	**−0,002**
$D \cap C$	2000	1000	4	249	500	4	124	7,732	−0,026	**−0,614**
$C \cap D$	2000	500	1	499	1000	4	249	7,158	−0,068	**−0,004**
$D \cap C$	2000	1000	4	249	500	1	499	27,856	−0,484	**−20,288**
$C \cap D$	3000	1000	2	499	2500	4	624	52,133	**−0,887**	+0,008
$D \cap C$	3000	2500	4	624	1000	2	499	105,367	−1,764	**−52,350**

Taula 5.6: Intersecció de codis amb cardinalitat diferent

5.2.3 BinaryDirectSum(Q)

S'han codificat dues versions per a la suma directa binària. Versió1 fa la suma directa de forma recursiva i Versió2 utilitza seqüències. A les següents taules veiem els resultats obtinguts per a una llista de dos (Taula 5.7) i tres codis (Taula 5.8).

n	M	K_1	L_1	K_2	L_2	Versió1	Versió2
1000	500	2	249	4	124	**13,226**	+0,640
1000	500	2	249	2	249	**27,232**	+2,152
1000	500	2	249	1	499	**52,704**	+6,910
2000	1000	4	249	8	124	**27,686**	+2,122
2000	1000	8	124	2	499	**57,626**	+3,680
8000	1000	8	124	8	124	**54,806**	+1,952

Taula 5.7: Temps de càlcul per a una llista de dos codis $[C_1, C_2]$

n	M	K_1	L_1	K_2	L_2	K_3	L_3	Versió1	Versió2
500	100	2	49	4	24	4	24	**10,038**	+0,288
500	100	2	49	1	99	4	24	**40,142**	+3,130
1000	50	2	24	1	49	2	24	**20,144**	+0,634
1000	100	1	99	4	24	2	49	**76,314**	+14,860
2000	50	2	24	2	24	1	49	**40,264**	+3,180
3000	50	2	24	2	24	2	24	**31,006**	+0,828
4000	50	2	24	2	24	2	24	**42,000**	+0,300

Taula 5.8: Temps de càlcul per a una llista de tres codis $[C_1, C_2, C_3]$

Com podem veure a les taules 5.7. i 5.8. tant per a l'execució d'una llista de dos codis com de tres, Versió1 és més eficient. A més, la diferència de temps és major quan el nombre de líders dels codis és més gran, encara que la longitud n dels codis sigui molt gran. Per exemple, si comparem la tercera i sisena fila de la Taula 5.7, per a codis de longitud 1000 i 8000 respectivament obtenim gairebé el mateix temps d'execució. De manera similar, a la Taula 5.8 a la segona i cinquena fila per a codis de longitud 500 i 2000 respectivament els temps de càlcul són molt semblants. Ara bé, en el cas que els codis tinguin el mateix nombre de líders, per a valors més grans de n també augmentarà el temps d'execució però la diferència no serà tan gran. Això ho podem veure si comparem les dues darreres files de la Taula 5.8.

n	M	K_1	L_1	K_2	L_2	K_3	L_3	Versió1	Versió2
1000	200	8	24	8	24	8	24	**10,200**	+0,100
1000	200	4	49	4	49	4	49	**79,300**	+0,330
1000	200	2	99	2	99	2	99	**643,800**	out mem

Taula 5.9: El temps de càlcul depèn del nombre de líders dels codis

Fins ara havíem vist funcions on la longitud del codi resultant era n, en canvi en fer la suma directa de la llista $Q = [C_1, ..., C_r]$ obtenim un codi de longitud $(n_1 + n_2 + \cdots + n_r)$ on n_i és la longitud de C_i per a tot C_i, $1 \leq i \leq r$. Per tant, cada cop treballem amb vectors de longitud més gran i necessitem més memòria per a emmagatzemar-ho. El nombre de líders de la suma directa dels codis de la llista Q es calcula com el producte del nombre de líders dels codis C_i. En cas de tenir diversos codis i que aquests tinguin molts líders, la memòria necessària i el temps de càlcul total pot crèixer ràpidament. A la Taula 5.9 podem veure com per a codis del mateix tamany els temps d'execució varien significativament depenent del nombre de líders dels codis. Per exemple, si tenim una llista de tres codis on $n = 1000$, $M = 200$ i el nombre de líders és 99, amb Versió1 obtenim un temps de càlcul de 10,7 minuts. Amb Versió2 MAGMA no té prou espai a memòria per a treballar.

Capítol 6

Conclusions

En aquest darrer capítol compararem els objectius definits inicialment amb els que s'han acabat assolint. Parlarem de les conclusions que podem extreure després de la realització del projecte i de les tasques que es poden continuar desenvolupant.

6.1 Assoliment d'objectius

Les fites marcades a l'inici s'han assolit en gran mesura, tot i que no ha donat temps de desenvolupar totes les funcions que ens havíem plantejat de bon començament.

1. S'ha estudiat la representació en MAGMA dels codis binaris no lineals, això implica l'anàlisi de la situació inicial del paquet *BinaryCodes*. També s'han estudiat els fonaments teòrics necessaris per al desenvolupament de les funcions.

2. S'han implementat les funcions que ens permeten construir nous codis a partir d'altres existents. D'aquestes, havíem planificat la codificació de quatre (*BinaryUnion*, *BinaryIntersection*, *BinaryDual* i *BinaryDirectSum*) però finalment han sigut cinc. Com que s'estava

treballant en la suma directa binària amb dos paràmetres, vam considerar oportú fer també la versió que rep com a paràmetre una llista de codis binaris. A més, les funcions han sigut sotmeses a diverses revisions, per tal de codificar-les de la manera més entenedora possible, i s'han generat diverses versions amb l'objectiu de reduir el temps de càlcul.

3. S'han el·laborat uns testos de caixa negra i blanca molt robustos, creant una base de dades de codis de prova. La quantitat de casos que es testejen demostren la fiabilitat de les funcions desenvolupades. En aquest sentit, s'ha destinat més temps de l'assignat a aquesta tasca perquè es va considerar que era més important fer menys funcions però de major qualitat.

4. S'han realitzat una sèrie d'exemples d'execució de les funcions desenvolupades, mostrant algunes de les seves propietats bàsiques. Aquests exemples s'han documentat en anglès seguint l'estil de MAGMA al document *Handbook of* MAGMA *functions* [1] . Es tracta d'un manual que servirà d'ajuda als usuaris que necessitin fer servir la llibreria *BinaryCodes*.

5. S'han documentat en anglès les funcions desenvolupades, tant al fitxer de codi font com a la guia d'usuari del paquet *BinaryCodes*. Igualment, s'han documentat en anglès els testos, i per primer cop, s'han comentat detalladament els codis utilitzats en les proves i els casos testejats. D'aquesta manera, aconseguim que el paquet sigui fàcil d'entendre per a qualsevol nou desenvolupador que vulgui fer modificacions, millores o ampliacions de les funcions actuals.

6. Un aspecte essencial era la unificació de funcions, testos i exemples seguint l'estil definit al document *CCGStyleGuide* [dg10]. Tot i que s'han fet modificacions d'estil en els darrers mesos, ens hem adaptat als nous canvis, fet que ha afectat a la planificació inicial establerta.

[1] Es pot consultar el *Handbook of* MAGMA *functions a l'apèndix A*

7. Finalment, s'ha redactat la present memòria que reflecteix el treball fet durant la vida del projecte.

Per tant, podem dir s'han assolit els objectius proposats a l'inici contribuint en el desenvolupament del paquet *BinaryCodes* amb unes funcions i testos de qualitat. A canvi, no ha sigut possible codificar les funcions que tracten la distribució de pesos de codis binaris. Es pretenia millorar la versió proposada per la projectista Laura Vidal [Vid09] i generar uns testos robustos seguint l'estil del projecte [Cua10]. Però els canvis d'estil, el temps destinat als testos i els problemes trobats durant el desenvolupament de les funcions han canviat les prioritats del projecte.

6.2 Conclusions

Aquest projecte ha suposat un repte personal. Des de que vaig cursar l'assignatura de Teoria de la codificació em vaig adonar que volia fer el meu projecte sobre codis. El curs passat una companya, Laura Vidal, va triar aquest camí i gràcies a això vaig conèixer millor la feina que estaven duent a terme al grup CCG. I aquí ens trobem, presentant el treball de tot un any que tancarà els meus estudis d'enginyeria en informàtica.

Aquesta memòria pretén ser un testimoni dels avenços fets, però també de la dificultat que suposa un projecte amb un rerefons matemàtic tan important. He hagut de refrescar els meus coneixement sobre Magma i aprendre a treballar amb LaTeX, ja que mai l'havia fet servir.

Col·laborar en el desenvolupament del paquet *BinaryCodes* ha sigut una tasca estimulant. M'ha permès aplicar els coneixements adquirits al llarg de la carrera de manera pràctica. A més, és molt satisfactori saber que el meu treball serà aprofitat per l'Escola d'Enginyeria i pels desenvolupadors de Magma.

6.3 Línies futures

Pensant en futurs projectes que continuïn amb la tasca de desenvolupament del paquet *BinaryCodes*, hi ha una sèrie d'aspectes que s'han de millorar:

- Desenvolupar en MAGMA les funcions relacionades amb la distribució de pesos i analitzar quina millora aporten respecte les mateixes funcions desenvolupades a [Vid09].

- Analitzar perquè el test 20 de caixa negra de la funció *BinaryUnion* (unió d'un codi binari no lineal amb el codi *Universe*) triga molt més temps en executar-se que la resta de testos i intentar optimitzar-lo per a reduir el temps de càlcul.

- Ampliar les funcions implementades en aquest projecte, i afegir els testos adients, per a què acceptin codis que no contenen la paraula zero. També, revisar altres funcions del paquet *BinaryCodes* desenvolupades als projectes [Cua10] i [Vid09] en el mateix sentit.

- Revisar les funcions *IsBinaryEqual i IsBinarySubset*, desenvolupades en projectes anteriors, ja que s'han detectat errors.

- Analitzar si es pot millorar el temps d'execució de la funció que calcula la suma directa (*BinaryDirectSum*) d'una llista de codis binaris. Comprovar si l'ordre dels codis d'entrada influeix en l'eficiència.

- Elaborar uns testos robustos. Moltes de les funcions que hi ha actualment al paquet no tenen testos que comprovin el seu correcte funcionament. D'altres en tenen però s'haurien de millorar ja que comprovem molts pocs casos.

- Unificar l'estil del paquet amb els canvis incorporats en el darrer any. Les funcions, testos i exemples desenvolupats en aquest projecte poden servir de guia.

- Comparar el temps de càlcul entre la llibreria *BinaryCodes* en MAGMA

i la llibreria *GUAVA* en GAP per a codis binaris no lineals. Analitzar el guany en temps d'execució i en memòria que suposa utilitzar la representació compacta de *kernel* i llista de líders de MAGMA respecte la representació de *GUAVA* que emmagatzema totes les paraules codi.

Bibliografia

[Cua10] Joan Cuadros. *Codis no lineals en* MAGMA *representació i construcció.* PFC Enginyeria Informàtica, Gener 2010.

[dg10] CCG development group. *CCG Style Guide.* Departament d'Enginyeria de la Informació i les Comunicacions, Abril 2010.

[gap08] The Gap Group. *GAP - Groups, Algorithms and Programming - a System for Computational Discrete Algebra*, Version 4.4.12, December 2008.
<http://www.gap-system.org/>

[Gas08] Bernat Gaston. *Codis $\mathbb{Z}_2\mathbb{Z}_4$-additius en* MAGMA. PFC Enginyeria Informàtica, Juny 2008.

[GPV08] B. Gastón, J. Pujol and M. Villanueva. *Development of algorithmic methods for binary non-linear codes in* MAGMA, pp. 345-351, July 2008.

[mag10] The Computational Algebra Group. MAGMA *Computational Algebra System*, Version 2.16-6, March 2010.
<http://magma.maths.usyd.edu.au/magma/>

[Ova08] Víctor Ovalle. *Códigos binarios no lineals en magma.* PFC Enginyeria Informàtica, Juny 2008.

[sag10] The SAGE Group. *Software for Algebra and Geometry Experimentation*, Version 4.4.3, June 2010. <http://www.sagemath.org/>

[Vid09] Laura Vidal. *Codis no lineals en* MAGMA *construcció de codis perfectes.* PFC Enginyeria Informàtica, Juny 2009.

Apèndix A

Handbook of MAGMA functions

A.1 Introduction

MAGMA currently supports the basic facilities for codes over finite fields and codes over integer residue rings and galois rings, all that are linear codes (see [3, Chapters 127-130]). Therefore, MAGMA provides functions for the special case of binary linear codes, that is when the finite field is $GF(2)$, or equivalently the finite ring is $\mathbb{Z}_2$. This chapter describes functions which are applicable to binary codes not necessarily linear. A (n, M, d) *binary code* C is a subset of $\mathbb{Z}_2^n$ with cardinality M and minimum Hamming distance d. If C is not linear, then the zero word does not need to belong to C. If the zero word is not in C, considering a new binary code $C' = C + c$ for any $c \in C$, we can assure that the zero word is in the binary code C', which is equivalent to C.

In this chapter, the term "code" will refer to a binary code not necessarily linear such that it contains the zero word, unless otherwise specified.

Two structural properties of binary codes are the rank and kernel. The *rank* of a binary code C, $r = rank(C)$, is simply the dimension of the linear span, $\langle C \rangle$, of C. The *kernel* of a binary code C is defined as

$K(C) = \{x \in \mathbb{Z}_2^n \mid x + C = C\}$. If the zero word is in C, then $K(C)$ is a linear subspace of C. We will denote the dimension of the kernel of C by $k = ker(C)$. In general, C can be written as the union of cosets of $K(C)$:

$$C = \bigcup_{i=0}^{t} \Big(K(C) + c_i \Big),$$

where $c_0 = \mathbf{0}$ and $t+1 = M/2^k$. These parameters can be used to distinguish between non-equivalent binary codes, since equivalent ones have the same parameters r and k.

Let C be a binary code of length n such that the zero word belongs to C, with kernel $K(C)$ of dimension k and coset leaders $[c_1, \ldots, c_t]$. The *parity check system* of the binary code C is a $(n-k)\times(n+t)$ binary matrix $(G|S)$, where G is a generator matrix of the dual code $K(C)^{\perp}$ and $S = (G\cdot c_1\ G\cdot c_2\ \ldots\ G\cdot c_t)$. The *super dual* of the binary code C is the binary linear code generated by the parity check system $(G|S)$. Note that if C is a binary linear code, the super dual is the dual code $C^{\perp}$. From these definitions, we can establish the following properties (see [8]):

- Let $col(S)$ denote the set of columns of the matrix S.
  Then, $c \in C$ if and only if $G \cdot c = \mathbf{0}$ or $G \cdot c \in col(S)$.

- Let $r = rank(C)$ and $k = ker(C)$.
  Then, $r = n - dim(G) + dim(S)$ and $k = n - dim(G)$.

A.2 Construction of Binary Codes

`BinaryCode(L)`

Creates a binary code C given by its super dual, which is a binary linear code of length $n+t$ generated by the parity check system $(G|S)$ for the binary code C. As it is explained above, the parity check system $(G|S)$ is constructed using the kernel $K(C)$ and the coset leaders $[c_1, \ldots, c_t]$ of the binary code obtained by L, where:

1. L is a sequence of elements of $V = \mathbb{Z}_2^n$,
2. or, L is a subspace of $V = \mathbb{Z}_2^n$,
3. or, L is a $m \times n$ matrix A over the ring $\mathbb{Z}_2$,
4. or, L is a binary linear code,
5. or, L is a quaternary linear code,
6. or, L is a -additive code.

Note that in general, depending on the cardinality M of the binary code C, this function could take some time to compute $K(C)$ and $[c_1, \ldots, c_t]$, in order to return the binary code given by its super dual.

If L is a quaternary linear code or a -additive code, then the binary code corresponds to the image of L under the Gray map. If the zero word is not in L, then L is substituted by $L + c$, where c is the first element in L.

This constructor appends five attributes to the code category:

- `Length`: The length n of the binary code.
- `Kernel`: The kernel of the binary code as a binary linear subcode.
- `CosetLeaders`: The sequence of coset leaders $[c_1, \ldots, c_t]$.
- `MinimumDistance`: The minimum (Hamming) distance.
- `IsLinear`: It is `true` if and only if C is a binary linear code.

If L is a subspace of $V = \mathbb{Z}_2^n$ or a binary linear code, this function returns the dual code. In this case, the `Kernel` is identical to the binary linear code L defined by the function `LinearCode()`, the `Cosets` attribute is the empty sequence, the `Length` attribute is n and the `IsLinear` attribute is `true`.

`BinaryRandomCode(`n`,`M`)`

Given two positive integers n and M, return a random binary code of length n and cardinality M.

`BinaryRandomCode(`n`,`M`,`k`)`

Given three positive integers n, M and k, return a random binary code of length n, cardinality M and with a kernel of minimum dimension k.

Example HAE1

We can define a binary code by giving a sequence of elements of a vector space $V = \mathbb{Z}_2^n$, or a matrix over $\mathbb{Z}_2$. Note that the `BinaryCode` function returns the super dual of a binary code C and the codewords of C can be generated as the union of the cosets of its kernel.

```
> V  := VectorSpace(GF(2),4);
> L  := [V!0,V![1,0,0,0],V![0,1,0,1],V![1,1,1,1]];
> C1 := BinaryCode(L);
> C1;
[7, 4, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1 0 1]
[0 1 0 0 0 1 1]
[0 0 1 0 0 0 1]
[0 0 0 1 0 1 1]
> IsBinaryLinearCode(C1);
false

> A  := Matrix(L);
> C2 := BinaryCode(A);
> C2;
[7, 4, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1 0 1]
[0 1 0 0 0 1 1]
[0 0 1 0 0 0 1]
[0 0 0 1 0 1 1]
```

```
> IsBinaryEqual(C1, C2);
true
```

A binary linear code C can be generated as a binary code with this constructor. Note that in this case the `BinaryCode` function returns the dual of C.

```
> C := LinearCode(sub<V|[[0,0,1,1],[1,0,1,1]]>);
> D := BinaryCode(sub<V|[[0,0,1,1],[1,0,1,1]]>);
> D;
[4, 2, 1] Linear Code over GF(2)
Generator matrix:
[0 1 0 0]
[0 0 1 1]
> IsBinaryLinearCode(D);
true
> Dual(C) eq D;
true
```

Example HAE2

We can also construct random binary codes with a given length, number of codewords and, optionally, with a given minimum kernel dimension.

```
> C1 := BinaryRandomCode(8,56);
> C1;
[14, 5, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 1 0 1 0 1 0]
[0 1 0 0 0 0 1 0 1 0 1 0 0 0]
[0 0 1 0 0 0 1 1 1 0 1 0 0 1]
[0 0 0 1 0 1 1 1 0 1 1 1 1 1]
[0 0 0 0 1 0 0 1 0 0 0 1 1 1]
> (BinaryLength(C1) eq 8) and (BinaryCardinal(C1) eq 56);
true

> C2 := BinaryRandomCode(8,48,2);
> C2;
[19, 6, 6] Linear Code over GF(2)
Generator matrix:
```

```
[1 0 0 1 0 0 0 0 1 1 0 1 0 0 0 1 0 1 0]
[0 1 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 0 1]
[0 0 1 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1 1]
[0 0 0 0 1 0 0 0 0 0 1 1 1 1 0 1 1 1 1]
[0 0 0 0 0 1 0 1 0 0 0 0 1 1 0 0 1 1 1]
[0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 1]
> BinaryKernelDimension(C2) ge 2;
true
```

A.3 Invariants of a Binary Code

`BinaryLength(C)`

Given a binary code C, return the length of the code.

`BinaryCardinal(C)`

Given a binary code C, return the number of words belonging to C.

`BinaryMinimumDistance(C)`

Given a binary code C, return the minimum (Hamming) distance of the words belonging to the code C.

`BinaryParameters(C)`

Given a binary code C, return a list with the parameters $[n, M, d]$, where n is the length of the code, M the number of codewords and d the minimum (Hamming) distance.

`BinarySpanZ2Code(C)`

Given a binary code C of length n, return the linear span of C, that is, the binary linear code generated by the codewords of C.

`BinaryDimensionOfSpanZ2(C)`
`BinaryRankZ2(C)`

Given a binary code C of length n, return its rank. The rank of a binary code C is the dimension of the linear span of C over $\mathbb{Z}_2$.

`BinaryKernelZ2Code(C)`

Given a binary code C of length n, return its kernel as a binary linear code, and the leaders of the cosets as a list of binary vectors of length n. The kernel of a binary code C is the set of codewords c such that $c + C = C$.

`BinaryDimensionOfKernelZ2(C)`

Given a binary code C of length n, return the dimension of its kernel.

Example HAE3

Given a binary code C, we compute its length, number of codewords, minimum distance, rank and kernel dimension.

```
> C := BinaryRandomCode(10,64);
> C;
[17, 7, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1]
[0 1 0 0 0 0 0 0 1 1 0 1 1 0 1 0 0]
[0 0 1 0 0 0 1 0 1 1 1 1 0 0 1 1 0]
[0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 1]
[0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 1 1]
[0 0 0 0 0 1 1 0 1 0 0 1 0 1 1 1 1]
[0 0 0 0 0 0 0 1 0 1 0 0 1 1 1 1 1]
> BinaryLength(C);
10
> BinaryCardinal(C);
64
> BinaryMinimumDistance(C);
1
> BinaryParameters(C);
[ 10, 64, 1 ]
> r := BinaryDimensionOfSpanZ2(C);
> r;
```

```
10
> k := BinaryDimensionOfKernelZ2(C);
> k;
3
> kernel, cosetLeaders := BinaryKernelZ2Code(C);
> ((2^k)*(#cosetLeaders+1)) eq BinaryCardinal(C);
true
```

A.4 Operations on Codewords

`BinarySet(C)`

Given a binary code C, return the set of all codewords of C.

`BinaryRandom(C)`

Return a random codeword of the binary code C.

`IsInBinaryCode(C, u)`

Return `true` if and only if the vector u of $V = \mathbb{Z}_2^n$ belongs to the binary code C of length n.

`IsNotInBinaryCode(C, u)`

Return `true` if and only if the vector u of $V = \mathbb{Z}_2^n$ does not belong to the binary code C of length n.

Example HAE4

```
> V  := RSpace(IntegerRing(4), 5);
> C1 := Z2Z4AdditiveCode([V![2,2,1,1,3],V![0,2,1,2,1],V![2,2,2,2,2],
                          V![2,0,1,1,1]] : Alpha:=2);
> C  := BinaryCode(C1);
> C;
[8, 2, 4] Quasicyclic of degree 4 Linear Code over GF(2)
```

```
Generator matrix:
[1 1 0 1 0 0 1 0]
[0 0 1 1 0 0 1 1]
> c := BinaryRandom(C);
> IsInBinaryCode(C,c);
true
```

A.5 Membership and Equality

`IsBinarySubset(C, D)`

Return `true` if and only if the binary code C is a subcode of the binary code D.

`IsBinaryNotSubset(C, D)`

Return `true` if and only if the binary code C is not a subcode of the binary code D.

`IsBinaryEqual(C, D)`

Return `true` if and only if the binary codes C and D are equal.

`IsBinaryNotEqual(C, D)`

Return `true` if and only if the binary codes C and D are not equal.

Example HAE5

```
> V  := VectorSpace(GF(2),5);
> C1 := BinaryCode([V![1,1,1,1,1],V![0,0,0,0,0],
                    V![1,1,0,0,0],V![1,0,1,1,1]]);
> C2 := BinaryCode(Matrix([V![1,1,1,1,1],V![0,0,0,0,0],
                           V![1,1,0,0,0],V![1,0,1,1,1]]));
> IsBinaryEqual(C1,C2);
true
```

```
> C3 := BinaryRandomCode(8,64,2);
> C4 := BinaryRandomCode(8,32);
> IsBinarySubset(C3,C4);
false

> C5 := BinaryCode(C4‘Kernel);
> IsBinarySubset(C5,C4);
true
```

A.6 Properties of Binary Codes

`IsBinaryCode(C)`

Return `true` if and only if C is a binary code.

`IsBinaryLinearCode(C)`

`IsZ2LinearCode(C)`

Return `true` if and only if C is a binary linear code.

`IsZ4LinearCode(C,p)`

Given a binary code C of length n and a permutation p, return `true` if and only if the binary code $p(C) = \{p(c) : c \in C\}$ is the anti-image under the Gray map of a code over $\mathbb{Z}_4$, or equivalently, a quaternary linear code.

`IsZ2Z4LinearCode(C,`α`,p)`

Given a binary code C of length n, a positive integer α such that $\alpha \leq n$, and a permutation p, return `true` if and only if the binary code $p(C) = \{p(c) : c \in C\}$ is the anti-image under the Gray map of a -additive code over $\mathbb{Z}_2^{\alpha} \times \mathbb{Z}_4^{\beta}$, where $\beta = (n - \alpha)/2$.

`IsBinaryPerfectCode(C)`

Return `true` if and only if C is a binary perfect code.

`IsBinaryExtendedPerfectCode(C)`

Return `true` if and only if C is a binary extended perfect code.

Example HAE6

```
> V  := VectorSpace(GF(2),3);
> C1 := BinaryCode([V!0,V![0,1,0],V![0,0,1]]);
> IsBinaryCode(C1);
true
> IsBinaryLinearCode(C1);
false
> C1`IsLinear;
false

> C2 := BinaryCode([V!0,V![1,1,1]]);
> IsBinaryPerfectCode(C2);
true
> C3 := BinaryRandomCode(4,2);
> IsBinaryExtendedPerfectCode(C3);
true

> V  := RSpace(IntegerRing(4),4);
> C  := Z2Z4AdditiveCode([V![2,2,1,1],V![0,2,1,2],V![2,2,2,2],V![2,0,1,1]] : Alpha:=2);
> Cb := BinaryCode(C);
> Cb;
[6, 0, 6] Cyclic Linear Code over GF(2)
> S1 := Sym(Cb`Length);
> p1 := S1!(1,2);
> Q  := RandomLinearCode(IntegerRing(4),5,2);
> Qb := BinaryCode(Q);
> Qb;
[13, 8, 2] Linear Code over GF(2)
Generator matrix:
```

```
[1 0 0 0 0 0 0 1 0 0 1 0 1]
[0 1 0 0 0 0 0 1 0 0 0 0 0]
[0 0 1 0 0 0 0 1 0 1 0 0 1]
[0 0 0 1 0 0 0 1 0 1 1 1 1]
[0 0 0 0 1 0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0 0 1 0 1 1]
[0 0 0 0 0 0 1 1 0 0 1 0 1]
[0 0 0 0 0 0 0 0 1 1 0 1 1]
> S2 := Sym(Qb`Length);
> p2 := S2!(1,2);
> IsZ2Z4LinearCode(Cb,2,p1) and IsZ4LinearCode(Qb,p2);
true
```

A.7 Union, Intersection and Dual

`BinaryUnion(C,D)`

Return the union of the binary codes C and D, both of the same length.

`BinaryIntersection(C,D)`

Return the intersection of the binary codes C and D, both of the same length.

`BinaryDual(C)`

Return the dual of a binary code C of length n. The dual consists of all vectors in the vector space $V = \mathbb{Z}_2^n$ which are orthogonal to all codewords of C.

Example HAE7

We verify some simple results from the union, intersection and dual of binary codes.

```
> C1:=BinaryRandomCode(5,4);
> C2:=BinaryRandomCode(5,4);
> C1UnionC2 := BinaryUnion(C1,C2);
```

```
> C1InterC2 := BinaryIntersection(C1,C2);
> BinaryCardinal(C1) + BinaryCardinal(C2) eq
> BinaryCardinal(C1UnionC2) + BinaryCardinal(C1InterC2);
true

> Universe5 := UniverseCode(GF(2), 5);
> // delete Universe5 field assigned after another execution
> delete Universe5`IsLinear;
> IsBinaryEqual(BinaryUnion(C1,Universe5), Universe5);
true
> IsBinaryEqual(BinaryIntersection(C1,Universe5), C1);
true
> Zero5 := ZeroCode(GF(2), 5);
> // delete Zero5 field assigned after another execution
> delete Zero5`IsLinear;
> IsBinaryEqual(BinaryUnion(C1,Zero5), C1);
true
> IsBinaryEqual(BinaryIntersection(C1,Zero5), Zero5);
true

> V := VectorSpace(GF(2),5);
> C_nonlinear := BinaryCode([V![1,0,0,1,0],V![1,1,1,1,1],
>                            V![0,0,0,0,0],V![0,0,0,0,1]]);
> C_linear1 := BinaryCode([V![0,0,0,0,0],V![0,1,1,0,1],
>                          V![0,0,1,1,0],V![0,1,0,1,1]]);
> C_linear2 := LinearCode(sub< V | [1,0,0,0,1], [0,1,0,0,1],
>                                  [0,0,1,0,1], [0,0,0,1,1]>);
> BinaryDual(C_nonlinear) eq Dual(BinarySpanZ2Code(C_nonlinear));
true
> BinaryDual(C_linear1) eq Dual(BinarySpanZ2Code(C_linear1));
true
> BinaryDual(C_linear2) eq Dual(BinarySpanZ2Code(C_linear2));
true
> BinaryDual(C_linear1) eq Dual(C_linear1`Kernel);
true
> BinaryDual(C_linear2) eq Dual(BinaryCode(C_linear2)`Kernel);
true
```

```
> GeneratorMatrix(BinaryDual(C_linear1)) eq
> ParityCheckMatrix(C_linear1`Kernel);
true
> GeneratorMatrix(BinaryDual(C_linear2)) eq
> ParityCheckMatrix(C_linear2);
true
```

A.8 New Codes from Existing

`BinaryDirectSum(C, D)`

Given binary codes C and D, construct the direct sum of C and D. The direct sum is a binary code that consists of all vectors of the form (u, v), where $u \in C$ and $v \in D$.

`BinaryDirectSum(Q)`

Given a sequence of binary codes $Q = [C_1, \ldots, C_r]$, construct the direct sum of all these binary codes C_i, $1 \leq i \leq r$. The direct sum is a binary code that consists of all vectors of the form $(u_1, \ldots, u_r)$, where $u_i \in C_i$, $1 \leq i \leq r$.

`BinaryExtendCode(C)`

Given a binary code C form a new binary code C' from C by adding the appropriate extra coordinate to each vector of C such that the sum of the coordinates of the extended vector is zero.

`BinaryPunctureCode(C, i)`

Given a binary code C of length n and an integer i, $1 \leq i \leq n$, construct a new binary code C' by deleting the i-th coordinate from each codeword of C.

`BinaryPunctureCode(C, S)`

Given a binary code C of length n and a set S of distinct integers $\{i_1, \ldots, i_r\}$ each of which lies in the range $[1, n]$, construct a new binary code C' by deleting the components $i_1, \ldots, i_r$ from each codeword of C.

`BinaryShortenCode(C, i)`

Given a binary code C of length n and an integer i, $1 \leq i \leq n$, construct a new binary code from C by selecting only those codewords of C having a zero as their i-th component and deleting the i-th component from these codewords. Thus, the resulting code will have length $n - 1$.

`BinaryShortenCode(C, S)`

Given a binary code C of length n and a set S of distinct integers $\{i_1, \ldots, i_r\}$ each of which lies in the range $[1, n]$, construct a new binary code from C by selecting only those codewords of C having zeros in each of the coordinate positions $i_1, \ldots, i_r$, and deleting these components. Thus, the resulting code will have length $n - r$.

`BinaryPlotkinSum(C, D)`

Given binary codes C and D both of the same length, construct the Plotkin sum of C and D. The Plotkin sum is a binary code that consists of all vectors of the form $(u, u + v)$, where $u \in C$ and $v \in D$.

Example HAE8

We combine binary codes in different ways and look at the length of the new binary codes.

```
> C1 := BinaryRandomCode(5,4);
> C2 := BinaryRandomCode(7,3);
> C1‘IsLinear;
true
> C2‘IsLinear;
false
> C1‘Length;
5
> C2‘Length;
```

```
7
> C3 := BinaryDirectSum(C1,C2);
> C3‘Length;
12
> C4 := BinaryDirectSum([C1,C2,C3]);
> C4‘Length;
24
> C5 := BinaryExtendCode(C2);
> C5‘Length;
8
> C6 := BinaryPunctureCode(C2,4);
> C6‘Length;
6
> C7 := BinaryShortenCode(C2,{4,5});
> C7‘Length;
5
> C8 := BinaryPlotkinSum(C2,C2);
> C8‘Length;
14
```

Apèndix B

Paquet *BinaryCodes*

Firmat: Miriam Gutiérrez Alarcón

Bellaterra, juny de 2010

Resum

L'objectiu principal d'aquest projecte és ampliar la llibreria *BinaryCodes*, iniciada al 2007, que ens permet construir i manipular codis binaris lineals i no lineals. Per aquest motiu, s'han desenvolupat una sèrie de funcions, amb els seus corresponents tests i exemples, en l'entorn de programació matemàtica Magma. Aquestes funcions consisteixen bàsicament en la construcció de nous codis a partir d'altres ja existents.

Resumen

El objetivo principal de este proyecto es ampliar la librería *BinaryCodes*, iniciada en el 2007, que nos permite construir y manipular códigos binarios lineales y no lineales. Por este motivo, se han desarrollado una serie de funciones, con sus correspondientes tests y ejemplos, en el entorno de programación matemática Magma. Estas funciones consisten básicamente en la construcción de nuevos códigos a partir de otros ya existentes.

Abstract

The main goal of this project is to expand the *BinaryCodes* library, started in 2007, which allows us to construct and manipulate binary linear and nonlinear codes. For this reason, it has been developed a set of functions with its corresponding tests and examples, in the Magma mathematical programming environment. These functions basically consist in the construction of new codes from existing ones.