Codis no lineals en Magma: construcció de codis perfectes

Memòria del projecte de final de carrera corresponent als estudis d'Enginyeria Superior en Informàtica presentat per Laura Vidal Marín i dirigit per Mercè Villanueva Gay.

Bellaterra, juny de 2009

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

CERTIFICA:

Que la present memòria ha sigut realitzada sota la seva direcció per Laura Vidal Marín

Bellaterra, juny de 2009

Firmat: Mercè Villanueva Gay

a la meva família

Agraïments

Gràcies a la meva directora de projecte, Mercè Villanueva, aquest projecte ha tirat endavant. En tot moment m'ha guiat i ajudat, m'ha ensenyat el que calia saber per a poder treballar. I tot això ho ha fet amb la millor actitud i ganes; veritablement ha estat una directora de projecte immillorable.

També voldria agrair la participació d'en Jaume Pujol en aquest projecte, que em va proporcionar la seva ajuda amb la funció de Vasilev' i va aportar idees per a millorar el rendiment d'algunes funcions.

Aprofitant que aquest projecte significa la conclusió de la carrera, agrair a totes les persones, companys, professors, amics i familiars que m'han acompanyat en aquest camí llarg i dur, que han fet l'experiència millor i inoblidable.

A tots vosaltres, moltes gràcies.

Índex

Índex de figures

Capítol 1

Introducció

Els codis correctors d'errors es poden classificar, segons les propietats algebraiques de les seves paraules-codi, com a lineals o no lineals. Els codis lineals es poden expressar mitjançant un subespai vectorial, una representació molt adient per a treballar amb ells en un ordinador, ja que permet representar un codi amb una cardinalitat considerable (és a dir, que està format per moltes paraules-codi) mitjançant una base de vectors que pot generar qualsevol paraula del codi. Els codis no lineals, en canvi, no es poden representar mitjançant subespais vectorials, sinó que cal emmagatzemar tot el conjunt de paraules-codi en memòria; per a codis grans, aquesta opció és inviable.

En aquest projecte treballarem amb codis correctors d'errors construïts sobre l'alfabet binari, és a dir, que les seves paraules-codi estan formades per zeros i uns.

Els codis també es poden caracteritzar segons la relació entre la seva capacitat correctora i el seu radi de recobriment: els codis perfectes són aquells tals que la seva capacitat correctora coincideix amb el seu radi de recobriment, és a dir, un codi que pot corregir e errors i tal que qualsevol vector de la mateixa longitud que el codi estigui a una distància màxima d'e d'exactament una paraula-codi, és perfecte.

De codis perfectes n'hi ha de lineals i de no lineals. En aquest projecte ens centrarem en els codis binaris perfectes no lineals que poden corregir un error, anomenats 1-perfectes. Treballarem només amb codis 1-perfectes perquè no existeixen codis binaris perfectes no lineals i no equivalents amb capacitat correctora major que 1.

Per tal de treballar amb codis no lineals amb un ordinador, existeix una representació similar a la dels codis lineals per a minimitzar l'espai ocupat en memòria. Tots els codis, lineals o no, tenen un nucli (*kernel*) que sempre és lineal. Si coneixem el *kernel* d'un codi podem dividir-lo en dues parts: els vectors que formen part del *kernel* i els que no. Aquests darrers es poden partir en classes, i de cada classe s'escull un representant. Així, a partir dels representants escollits i del *kernel* es pot reconstruir el codi, tot i que no sigui lineal. Això ens permet, generalment, reduir les necessitats de memòria per a representar el codi. En el pitjor dels casos, si el *kernel* només conté una paraula-codi, haurem de guardar totes les paraules del codi. Aquesta representació dels codis no lineals s'anomena súper dual, i és una generalització del dual per a codis lineals.

En aquest projecte es desenvoluparà un paquet per a Magma que permeti construir codis binaris perfectes no lineals. La representació del súper dual amb Magma es va implementar en el paquet *BinaryCodes*, en el projecte [Ova08] el curs passat. Per tant, aprofitarem aquesta estructura a l'hora de representar els codis perfectes no lineals que es construiran amb el paquet *BinaryPerfectCodes* desenvolupat en aquest projecte.

1.1 Objectius

Els objectius que es van fixar a principi del projecte, i que s'han anat perfilant i ajustant al llarg del seu desenvolupament, han estat els següents:

1. Estudiar les bases teòriques per a la realització del projecte. La forma-

ció específica que s'ha fet ha estat general sobre codis no lineals, i més específicament sobre codis perfectes i algunes de les seves construccions.

2. Aprofitar propietats matemàtiques dels codis perfectes i dels codis no lineals per tal de poder construir codis perfectes de manera eficient.

3. Desenvolupar funcions que construeixin codis perfectes, funcions per a calcular invariants d'un codi i altres funcions de caire més general aplicables tant als codis no lineals qualssevol com als codis perfectes en particular.

4. Provar totes les funcions desenvolupades per tal de validar que funcionen correctament.

5. Escriure la documentació, tant del propi paquet de funcions desenvolupat, com el capítol d'ajuda de MAGMA, incloent exemples.

6. Escriure la memòria del projecte.

1.2 Contingut de la memòria

Aquesta memòria es divideix en diversos capítols. A continuació es descriu breument el contingut de cada capítol:

Capítol 2: Bases teòriques. Introducció al rerefons teòric del projecte. S'estudia la teoria general de codis lineals i no lineals, propietats i construccions dels codis perfectes.

Capítol 3: Planificació del projecte. Explicació en detall dels objectius del projecte i de com han anat evolucionant al llarg del temps. També s'inclou la planificació temporal del projecte, contraposant la planificació inicial amb la real, i la divisió del projecte en tasques i fites.

Capítol 4: Desenvolupament. Detalls sobre el procés de desenvolupament del *package* de MAGMA. Es descriuen quines eines i sistemes

s'han utilitzat i es discuteixen les decisions de disseny i implementació que s'han pres al llarg del projecte.

Capítol 5: *Handbook of* Magma *functions*. Manual d'ajuda de Magma. Conté una llista de totes les funcions desenvolupades, amb una explicació per a cada funció i exemples. El capítol està escrit en anglès, ja que s'ha d'integrar amb el llibre d'ajuda de Magma.

Capítol 6: Conclusions i resultats. Resum dels resultats del projecte. Estudi del comportament del paquet *BinaryPerfectCodes*.

Capítol 7: Conclusions. Revisió dels objectius del projecte, conclusions extretes a partir de la seva realització i proposta de possibles millores a desenvolupar en futurs projectes.

Bibliografia.

Apèndix. CD amb el codi font, proves, exemples i manual del *package* desenvolupat.

Capítol 2

Bases teòriques

En aquest capítol es veurà sobre quins fonaments teòrics es basa aquest projecte.

Els conceptes generals sobre codis s'han extret de [RH91], la representació de codis no lineals de [Hed08] i la introducció als codis perfectes està basada en les teories exposades a [Vil01]. Els conceptes bàsics de teoria de dissenys estan extrets de [Vil09].

Els exemples que il·lustren les definicions al llarg del capítol s'han realitzat amb MAGMA.

2.1 Codis correctors d'errors

Sigui $\mathbb{F}$ un cos finit. Siguin $\mathbb{F}^k$ i $\mathbb{F}^n$ els espais vectorials definits sobre $\mathbb{F}$ de dimensió k i n, respectivament (i.e. el conjunt de tots els vectors de k (i n) coordenades sobre $\mathbb{F}$, respectivament). Codificar significa transformar la informació $(a_1, a_2, \cdots, a_k) \in \mathbb{F}^k$ en un vector $(b_1, b_2, \cdots, b_n) \in \mathbb{F}^n$ que anomenarem **paraula-codi**. Per tant, definim un **codi** C com la imatge d'una aplicació $f : \mathbb{F}^k \to \mathbb{F}^n$, és a dir, que transforma vectors de longitud k en vectors de longitud n: $C = f(\mathbb{F}^k) \subseteq \mathbb{F}^n$.

Exemple 2.1 ***Codificació d'un vector d'informació***
Suposem que tenim un codi C que transforma vectors de 4 coordenades en vectors de 8 coordenades. Aquest codi C el farem servir al llarg dels exemples dins d'aquest capítol.

Per exemple, el nostre codi C transforma el vector d'informació (1 1 0 1) en la paraula-codi (1 1 0 1 1 0 0 0).

En aquest projecte es treballarà amb codis binaris. Sigui $\mathbb{Z}_2^n$ l'espai vectorial de dimensió n sobre el cos finit $\mathbb{Z}_2 = GF(2)$. Definim un **codi binari** com un subconjunt de $\mathbb{Z}_2^n$, els elements del qual s'anomenen paraules-codi.

Exemple 2.2 ***Conjunt de paraules-codi d'un codi***
Si C és un codi, la comanda "Set(C)" de MAGMA ens retorna totes les paraules-codi d'un codi. Aquestes són les paraules-codi que formen el codi C.

```
> Set(C);
    (1 0 1 1 1 1 1 1),          (0 1 0 1 1 0 1 0),
    (1 1 0 1 0 0 1 1),          (0 0 1 1 0 1 1 0),
    (0 0 0 0 0 0 0 0),          (0 1 1 0 0 1 1 1),
    (1 1 1 0 1 1 1 0),          (1 0 0 0 1 0 0 1),
    (0 0 1 1 1 1 0 1),          (1 1 0 1 1 0 0 0),
    (0 1 0 1 0 0 0 1),          (1 0 1 1 0 1 0 0),
    (0 1 1 0 1 1 0 0),          (1 0 0 0 0 0 1 0),
    (0 0 0 0 1 0 1 1),          (1 1 1 0 0 1 0 1)
```

Suposem que volem enviar una paraula-codi a través d'un canal de comunicació digital. Seria d'esperar que a la sortida del canal hi trobéssim la mateixa paraula-codi, però en les comunicacions digitals es poden produir errors, que es traduiran en alteracions en les coordenades del vector rebut respecte a la paraula-codi enviada. Els **codis correctors d'errors** introdueixen redundància a les dades per tal que, en cas de produir-se errors, el receptor pugui corregir-los i obtenir la informació que s'havia enviat.

2.1.1 Distància mínima i pes mínim

Definim la **distància de Hamming** entre dos vectors binaris $x = (x_1, \cdots, x_n)$, $y = (y_1, y_2, \cdots, y_n) \in \mathbb{Z}_2^n$ com la quantitat de components diferents entre tots dos:

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

La **distància mínima de Hamming** d d'un codi binari C és la menor de totes les distàncies entre cada parella de paraules-codi de C, és a dir: $d = \min\{d_H(u, v) \mid u \neq v,\ u, v \in C\}$. Així, definim un codi binari (n, M, d) com un subconjunt de $\mathbb{Z}_2^n$ amb cardinalitat M i distància mínima de Hamming d.

Exemple 2.3 ***Distància mínima de Hamming d'un codi***
La distància mínima del codi C és 2:

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

Ho podem provar mostrant les distàncies que hi ha entre totes les parelles de paraules codi diferents i buscant-ne el mínim:

```
> d := [Distance(u,v) : u,v in C | u ne v];
> d;
[ 2, 5, 3, 5, 5, 4, 4, 5, 7, 6, ..., 4, 3, 3, 2, 5, 3, 5, 2 ]
> Minimum(d);
2
```

Definim també el **pes de Hamming** d'un vector binari $x \in \mathbb{Z}_2^n$ com el nombre de components del vector diferents de 0, és a dir, $w_H(x) = d_H(x, \mathbf{0})$. El **pes mínim de Hamming** d'un codi binari C és el pes de la paraula-codi de menor pes del codi: $w = \min\{w_H(v) \mid v \in C, v \neq \mathbf{0}\}$.

Exemple 2.4 ***Pes mínim de Hamming d'un codi***
El pes mínim del codi C és 2:

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

Ho podem veure mitjançant la distribució de pesos del codi, que ens mostra per cada pes possible quantes paraules-codi hi ha. Excloent la paraula-codi zero, el menor pes possible és 2:

```
> WeightDistribution(C);
[ <0, 1>, <2, 1>, <3, 3>, <4, 5>, <5, 4>, <6, 1>, <7, 1> ]
```

2.1.2 Capacitat correctora i radi de recobriment

La **capacitat correctora** e d'un codi (n, M, d) depèn de la seva distància mínima:

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

Podem pensar en un codi com un esquema on cada paraula-codi és el centre d'una esfera, i al seu voltant hi haurà tots els vectors que descodificarem com la paraula-codi donada. Aquestes esferes, de radi e, són disjuntes, però en general no recobreixen tot $\mathbb{F}^n$.

Anomenem **radi de recobriment** d'un codi (n, M, d) al valor ρ, definit com el mínim radi que han de tenir les esmentades esferes per tal de recobrir tot $\mathbb{F}^n$:

$$\rho = \max_{x \in \mathbb{F}^n} \{ \min_{v \in C} \{ d_H(x, v) \} \},\ x \in \mathbb{F}^n,\ v \in C.$$

La capacitat correctora del codi sempre és menor o igual que el radi de recobriment.

Exemple 2.5 ***Capacitat correctora i radi de recobriment d'un codi*** Calculem la capacitat correctora del codi C segons la seva distància mínima, calculada en l'exemple 2.3.

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

```
0
```

Això vol dir que el codi C no pot corregir cap error.

Calculem el radi de recobriment del nostre codi C $(8, 16, 2)$:

```
> CoveringRadius(C);
2
```

Com veiem, la capacitat correctora e del codi és menor que el seu radi de recobriment.

2.1.3 Codis lineals i codis no lineals

Els codis binaris **lineals** són aquells que, a més de ser un subconjunt de $\mathbb{Z}_2^n$ són un subespai vectorial. Tot codi binari lineal té una **dimensió** k com a subespai de $\mathbb{Z}_2^n$ i tindrem que la cardinalitat del codi és $M = 2^k$. Un codi binari lineal es denota per $[n, 2^k, d]$, per tant $M = 2^k$. Qualsevol codi lineal, per tant, contindrà la paraula $\mathbf{0}$. Els codis no lineals, en canvi, no són un subespai vectorial.

Definim la **matriu generadora** G d'un codi binari lineal com una matriu les files de la qual formen una base del codi com a subespai vectorial, i per tant ens permet generar qualsevol paraula-codi. La matriu generadora d'un codi binari lineal C $[n, 2^k, d]$ té n columnes i k files.

Donat un codi binari lineal C $[n, 2^k, d]$ definim el **codi ortogonal** (o el **dual**) a C com $C^\perp = \{x \in \mathbb{Z}_2^n \mid x \cdot v = 0,\ \forall v \in C\}$ (on "$\cdot$" denota el producte escalar). Donat un codi binari lineal C $[n, 2^k, d]$ el seu codi ortogonal $C^\perp$ és també un codi binari lineal $C^\perp$ $[n, 2^{n-k}, d']$. Una propietat del codi dual és la següent: $(C^\perp)^\perp = C$.

La matriu generadora H del dual $C^\perp$ és la **matriu de control** del codi C, i ens permet saber si un vector pertany al codi C, ja que els elements del codi s'anul·len en ser operats amb la matriu H. De la mateixa manera, la matriu

generadora G del codi C és la matriu de control del codi ortogonal o dual $C^{\perp}$. La matriu de control d'un codi binari lineal C $[n, 2^k, d]$ té n columnes i $n - k$ files.

Els codis binaris lineals, doncs, pel fet de ser un subespai vectorial, es poden representar mitjançant una base d'aquest subespai, és a dir, a partir de la matriu generadora. A l'hora de treballar amb codis en un ordinador, aquesta propietat resulta molt interessant, ja que és una representació molt compacta. Suposem, per exemple, un codi de longitud n i cardinalitat $M = 2^k$:

- Si el codi és lineal, podem representar-lo mitjançant una base de vectors (la matriu generadora), de manera que necessitarem emmagatzemar k vectors de n coordenades, és a dir, $n \times k$ valors.
- Si el codi és no lineal, haurem de guardar totes les seves paraules-codi, de manera que necessitarem emmagatzemar $M = 2^k$ vectors de n coordenades, és a dir, $n \times 2^k$ valors.

Per a n i M prou grans, els codis no lineals no els podrem emmagatzemar a memòria i per tant no hi podrem treballar. Tanmateix, més endavant veurem que si estudiem les propietats dels codis no lineals podrem, en alguns casos, trobar una representació més compacta per al codi.

Exemple 2.6 ***Matriu generadora i matriu de control d'un codi lineal***

Comprovem que el codi C és lineal:

```
> IsLinear(C);
true
```

Podem conèixer més informació sobre el codi:

```
> C;
[8, 4, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 1 0]
```

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

Com ja hem calculat abans, veiem que C és un codi binari de longitud 8, dimensió 4 i distància mínima 2. Tot i que aquesta comanda ja ens mostra la matriu generadora, la podem calcular amb la comanda "GeneratorMatrix(C)". Calculem també la matriu de control amb la comanda "ParityCheckMatrix(C)":

```
> GeneratorMatrix(C);
[1 0 0 0 0 0 1 0]
[0 1 0 1 0 0 0 1]
[0 0 1 1 0 1 1 0]
[0 0 0 0 1 0 1 1]
```

```
> ParityCheckMatrix(C);
[1 0 0 0 1 1 1 0]
[0 1 0 0 1 0 0 1]
[0 0 1 0 0 1 0 0]
[0 0 0 1 1 1 0 1]
```

La matriu generadora té mida $n \times k = 8 \times 4$, i la matriu de control té mida $n \times (n-k) = 8 \times 4$.

2.1.4 Codis isomorfs i codis equivalents

Dos codis binaris C_1 i C_2 de longitud n són **isomorfs** si existeix una permutació de coordenades $\pi \in S_n$ tal que $C_2 = \pi(C_1) = \{\pi(v) | v \in C_1\}$.

Dos codis binaris C_1 i C_2 de longitud n són **equivalents** si existeix $x \in \mathbb{Z}_2^n$ i una permutació de coordenades $\pi \in S_n$ tal que $C_2 = \{x + \pi(c) | c \in C_1\}$.

2.2 Rang i nucli d'un codi

El rang i el nucli d'un codi ens permeten decidir si un codi és lineal o no, com veurem a continuació.

L'**expansió lineal** o *span* d'un codi C, denotada per $\langle C \rangle$, és el conjunt de

totes les combinacions lineals que es poden fer amb les paraules-codi de C. L'expansió lineal d'un codi sempre és lineal.

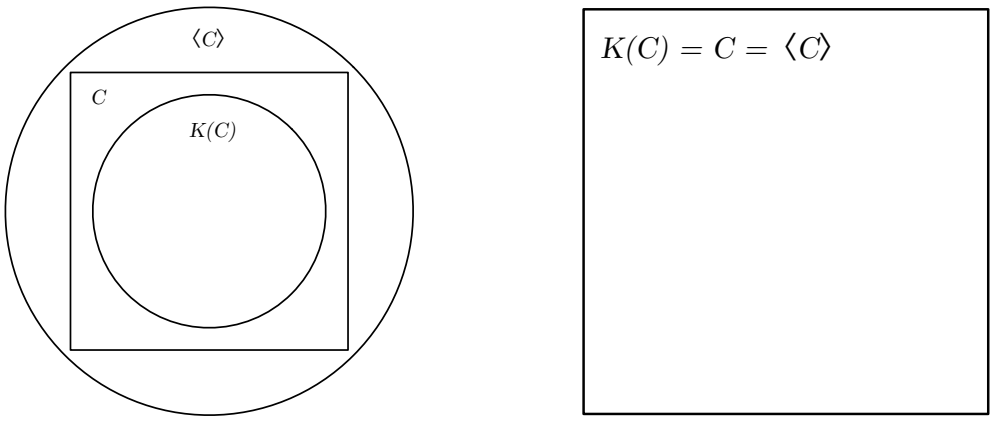
El **rang** r d'un codi C és la dimensió de l'expansió lineal de C.

El **nucli** o *kernel* d'un codi C de longitud n es defineix com $K(C) = \{x \in \mathbb{F}_2^n | C = C + x\}$, en altres paraules, el *kernel* està format per tots aquells vectors de $\mathbb{F}_2^n$ que deixen el codi C invariant per translació.

El *kernel* d'un codi, que conté el zero, sempre és lineal, i la seva dimensió es denota per k.

El rang i la dimensió del *kernel* són **invariants** d'un codi, és a dir, paràmetres del codi que no canvien encara que sotmetem el codi a transformacions isomorfes.

En el cas dels codis lineals, el *kernel* coincideix amb el codi i amb l'expansió lineal, és a dir, $K(C) = C = \langle C \rangle$. Per als codis no lineals, en canvi, $K(C) \subset C \subset \langle C \rangle$.



(a) Codis no lineals (b) Codis lineals

Figura 2.1: Relació entre el codi, el *kernel* i l'expansió lineal

El fet que qualsevol codi (inclús els no lineals) tingui el *kernel* lineal ens ajudarà a l'hora de trobar una representació més compacta per als codis no lineals.

Exemple 2.7 ***Kernel d'un codi lineal***

Calculem el nucli o kernel del codi C:

```
> K := {c: c,d in C | c+d in C};
> K;
{   (1 0 1 1 1 1 1 1), (1 1 0 1 0 0 1 1), (0 0 0 0 0 0 0 0),
    (1 1 1 0 1 1 1 0), (0 0 1 1 1 1 0 1), (0 1 0 1 0 0 0 1),
    (0 1 1 0 1 1 0 0), (0 0 0 0 1 0 1 1), (0 1 0 1 1 0 1 0),
    (0 0 1 1 0 1 1 0), (0 1 1 0 0 1 1 1), (1 0 0 0 1 0 0 1),
    (1 1 0 1 1 0 0 0), (1 0 1 1 0 1 0 0), (1 0 0 0 0 0 1 0),
    (1 1 1 0 0 1 0 1) }
```

Com veiem, la paraula-codi zero pertany al *kernel*. Com que C és un codi lineal, el seu *kernel* hauria de coincidir amb el propi codi:

```
> K eq Set(C);
true
```

A més, tant el *kernel*, com el codi, com l'expansió lineal de C han de coincidir:

```
> span := sub<VectorSpace(GF(2),8) | C>;
> Set(span) eq Set(C);
true
> Set(span) eq K;
true
> Set(C) eq K;
true
```

En canvi, en un codi no lineal, trobem que $K(C) \subseteq C \subseteq \langle C \rangle$. Per a veure el següent exemple, utilitzarem el paquet de MAGMA *BinaryCodes* i generarem un codi aleatori no lineal de longitud 8 i 31 paraules-codi:

```
> Attach("BinaryCodes.m");
> D := BinaryRandomCode(8,31);
> IsBinaryLinearCode(D);
false
```

```
> D‘Kernel;
[8, 0, 8] Cyclic Linear Code over GF(2)
> kernelD := Set(D‘Kernel);
> kernelD;
{
    (0 0 0 0 0 0 0 0)
}
> setD := BinarySet(D); // Conjunt de paraules-codi de D
> spanD := sub<VectorSpace(GF(2),8) | setD>;
```

Comprovem que el *kernel*, l'expansió lineal (*span*) i el codi no coincideixen:

```
> Set(spanD) eq setD;
false
> kernelD eq Set(spanD);
false
> kernelD eq setD;
false
```

I que el *kernel* és un subconjunt del codi, i aquest és un subconjunt de l'expansió lineal:

```
> kernelD subset setD;
true
> setD subset Set(spanD);
true
```

2.3 Classificació i representació de codis no lineals

Per a poder utilitzar els codis no lineals quan treballem en un ordinador hem de trobar una representació que ens permeti emmagatzemar el codi a

memòria.

Una primera opció és guardar la llista de totes les paraules-codi, però això és inviable per a codis grans. A continuació veurem una segona opció.

Sigui $x \in \mathbb{F}^n$ i $A \subseteq \mathbb{F}^n$. Direm que el subconjunt $x + A = \{x + y \mid y \in A\}$ és un **traslladat** del conjunt A. A més, el vector x és un representant o líder de la classe $x + A$.

Podem expressar un codi no lineal com la unió del *kernel* i els traslladats del *kernel* (o classes): $C = \bigcup_{i=0}^{t} (K(C) + c_i)$, on c_i són els representants de les classes.

Definim el **sistema de paritat** de C com la matriu $(H|S)$ de mida $(n-k) \times (n+t)$ tal que:

- H és la matriu generadora del dual del *kernel* de C, $K(C)^{\perp}$.
- S és el producte escalar d'H pels representants de les classes del *kernel*: $(H \cdot c_1^t, H \cdot c_2^t, \cdots, H \cdot c_t^t)$.

Diem que la matriu $(H|S)$ és una generalització de la matriu de control per a codis no lineals, ja que ens permetrà saber si un vector x pertany al codi no lineal C: $x \in C$ si $H \cdot x^t = 0$ o bé $H \cdot x^t$ és una columna de S. A més, si C és lineal, la matriu $(H|S)$ coincideix amb la matriu de control H de C.

La matriu $(H|S)$ és la matriu generadora del **super dual** del codi C, que és una generalització del codi dual per a codis no lineals. Notem, però, que

$$SuperDual(SuperDual(C)) \neq C,$$

a diferència dels codis lineals, on $(C^{\perp})^{\perp} = C$. La dimensió del *kernel* d'un codi C (n, M, d) amb matriu de control generalitzada $(H|S)$ és $k = dim(H) - n$, i el rang del codi és $r = k + dim(S)$.

Aquesta representació dels codis no lineals, així com el càlcul del *kernel*, està implementada en el paquet de Magma *BinaryCodes*, desenvolupat en

el projecte [Ova08].

2.4 Codis perfectes

Els **codis perfectes** són aquells codis tals que la seva capacitat correctora coincideix amb el radi de recobriment. En altres paraules, un codi C de longitud n és perfecte si per algun $e \geq 0$ cada vector de l'espai vectorial sobre el qual està definit el codi (en el nostre cas, pels codis binaris, sobre $\mathbb{Z}_2^n$) està a distància menor o igual a e d'exactament una paraula-codi de C.

Els codis perfectes poden corregir e errors, i s'anomenen $e-$correctors o $e-$perfectes.

Els codis binaris perfectes de longitud n que existeixen són els següents:

- Codis trivials:
  - Per $e = 0$, el codi format per tots els vectors de longitud n.
  - Per $e = n$, el codi format per una sola paraula de longitud n.
- Codi de repetició: $e = \dfrac{n-1}{2}$, amb n senar.
- Codi de Golay: amb capacitat correctora $e = 3$ i longitud $n = 23$.
- Codis 1-perfectes: $e = 1$ i longitud $n = 2^m - 1$, on m s'anomena *paràmetre* del codi. Els codis 1-perfectes lineals s'anomenen codis de Hamming.

Els únics paràmetres pels quals existeixen codis binaris perfectes no lineals i no equivalents són $e = 1$ i $n = 2^m - 1$ per $m \geq 4$.

Els codis binaris 1-perfectes de longitud $n = 2^m - 1$ tenen 2^{n-m} paraules-codi i distància mínima de Hamming 3. Per un codi binari 1-perfecte C tenim que $C^\perp \subseteq K(C)$ i que $k + r \geq n + 1$, on k és la dimensió del *kernel* i r el rang del codi, o dimensió de l'expansió lineal del codi.

Exemple 2.8 ***Codi perfecte***
Sigui C_h un codi de Hamming de paràmetre $m = 3$ definit sobre $GF(2)$:

```
> Ch := HammingCode(GF(2),3);
> Ch;
[7, 4, 3] "Hamming code (r = 3)" Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1 1 0]
[0 1 0 0 0 1 1]
[0 0 1 0 1 1 1]
[0 0 0 1 1 0 1]
```

Comprovem que el codi és perfecte:

```
> IsPerfect(Ch);
true
```

Comprovem les propietats dels codis perfectes, com ara que la capacitat correctora coincideix amb el radi de recobriment:

```
> e := Floor((MinimumDistance(Ch)-1)/2);
> e eq CoveringRadius(Ch);
true
```

També podem veure que té distància mínima 3, i $2^{n-m} = 2^4$ paraules-codi:

```
> MinimumDistance(Ch);
3
> #Ch;
16
```

2.4.1 Construccions de codis perfectes

A continuació presentem una sèrie de construccions de codis binaris 1-perfectes. Les construccions de codis perfectes, permeten, a base de fer operacions i transformacions a les paraules-codi d'un o més codis perfectes, obtenir un nou codi perfecte.

Construcció de Vasil'ev

Donat $x \in \mathbb{F}_2^n$ sigui $p(x) = w_H(x) \mod 2$. Sigui C_n un codi binari 1-perfecte de longitud $n = 2^m - 1$. Sigui $f : C_n \to \{0, 1\}$ una aplicació arbitrària tal que $f(0) = 0$ i $f(c_1) + f(c_2) \neq f(c_1 + c_2)$ (és a dir, que no sigui lineal) per a $c_1, c_2, c_1 + c_2 \in C_n$.

El codi $C_{2n+1} = \{(x|x + u|p(x) + f(u)) \ : \ x \in \mathbb{F}_2^n, \ u \in C_n\}$, on $(\cdot|\cdot)$ denota la concatenació, és binari 1-perfecte de longitud $2n+1 = 2^{m+1} - 1$, i aquesta manera de construir-lo s'anomena **construcció de Vasil'ev**.

Existeixen exactament 19 codis de Vasil'ev de longitud 15 no equivalents, tal i com es demostra a [Her82].

Construcció de Mollard

La construcció de Mollard és una generalització de la de Vasil'ev.

Sigui $x = (x_{11}, x_{12}, \ldots, x_{1n_2}, x_{21}, x_{22}, \ldots, x_{n_1 n_2}) \in \mathbb{F}_2^{n_1 n_2}$. Definim les funcions de paritat generalitzades, $p_1(x) = (\sigma_1, \cdots, \sigma_{n_1}) \in \mathbb{F}_2^{n_1}$, on $\sigma_i = \sum_{j=1}^{n_2} x_{ij}$, i $p_2(x) = (\sigma'_1, \cdots, \sigma'_{n_1}) \in \mathbb{F}_2^{n_2}$, on $\sigma'_j = \sum_{i=1}^{n_1} x_{ij}$.

Siguin C_1 i C_2 dos codis binaris 1-perfectes de longitud n_1 i n_2, respectivament. Sigui $f : C_1 \to \mathbb{F}_2^{n_2}$ una aplicació arbitrària.

El codi $F = \{(x|c_1 + p_1(x)|c_2 + p_2(x) + f(c_1)) : x \in \mathbb{F}_2^{n_1 n_2}, c_1 \in C_1, c_2 \in C_2\}$ és un codi binari 1-perfecte de longitud $n_1 n_2 + n_1 + n_2$. El codi F s'ha obtingut mitjançant la **construcció de Mollard**.

Construcció *doubling*

Igual que amb les dues construccions anteriors, amb la construcció *doubling* obtenim codis de longitud $2n + 1$ a partir de codis de longitud n.

Sigui e_i un vector tal que totes les seves coordenades són 0 excepte la i-èsima, que val 1.

Definim la suma directa $X \oplus Y = \{(x, y) : x \in X, y \in Y\}$ per $X, Y \subset \mathbb{F}_2^n$.

S'anomena codi extès d'un codi C al codi C^* construït de tal manera que a cada paraula-codi se li afegeix una coordenada al final que val 0 si la paraula-codi tenia pes parell i 1 si tenia pes senar. Sigui C_1 un codi binari 1-perfecte de longitud n i C_2^* un codi 1-perfecte estès de longitud $n + 1$. Sigui π una permutació en el conjunt $\{1, 2, \ldots, n\}$.

El codi $C = (C_1 \oplus C_2^*) \bigcup_{i=1}^{n} ((C_1 + e_i) \oplus (C_2 + e_{\pi(i)})^*)$ és un codi binari 1-perfecte de longitud $2n + 1$, i la manera com s'ha obtingut s'anomena **construcció *doubling***.

Construcció *switching*

Donat un codi binari 1-perfecte C, de longitud n, podem obtenir un nou codi C' mitjançant la construcció *switching* intercanviant un conjunt seleccionat de paraules-codi, $S \subset C$ per un altre conjunt de vectors S',

$$C' = (C \backslash S) \cup S'.$$

Els conjunts S i S' han de complir que

$$\{x \in F_2^n : d_H(x, S) \leq 1\} = \{x' \in F_2^n : d_H(x', S') \leq 1\},$$

entenent que la distància d'un vector x a un conjunt S és el mínim entre totes les distàncies d'x a tots els elements d'S. En altres paraules, l'espai que recobreix el conjunt d'esferes de radi 1 al voltant dels elements d'S ha de ser igual a l'espai recobert pel conjunt d'esferes de radi 1 al voltant dels elements d'S'.

El codi C' és un nou codi 1-perfecte de longitud n.

2.5 Teoria de dissenys

2.5.1 Sistemes triples d'Steiner

Un **sistema triple d'Steiner (STS)** és una parella ordenada $(V, \mathcal{B})$, on V és un conjunt finit de punts i $\mathcal{B}$ és un conjunt de subconjunts de 3 elements de V, anomenats triples, tal que cada parella d'elements diferents de V apareix junta en exactament un triple de $\mathcal{B}$.

L'ordre d'un STS $(V, \mathcal{B})$ és la mida del conjunt V, denotada per $|V|$. Escrivim $STS(v)$ per a denotar un STS d'ordre v.

Un STS $(V, \mathcal{B})$ es pot representar gràficament. Hi ha una equivalència entre un $STS(v)$ i una descomposició del graf d'ordre v, K_v, en triangles, tal que cada símbol de V correspon a un vèrtex, i cada triple $\{a, b, c\}$ es representa com a un triangle que uneix els vèrtexs a, b i c. Com que cada parella de símbols apareix en exactament un triple de $\mathcal{B}$, cada aresta pertany a exactament un triangle. Aleshores, un STS és equivalent a un graf complet K_v, on $|V| = v$, en el qual les arestes es particionen en triangles.

Només existeixen STS per $v \equiv 1$ o $v \equiv 3$ $(mod\ 6)$.

Exemple 2.9 ***Sistemes triples d'Steiner***
El sistema triple d'Steiner definit pels conjunts $V = \{1, 2, 3, 4, 5, 6, 7\}$ i $\mathcal{B} = \{\{1, 2, 4\}, \{2, 3, 5\}, \{3, 4, 6\}, \{4, 5, 7\}, \{5, 6, 1\}, \{6, 7, 2\}, \{7, 1, 3\}\}$ té ordre 7 i es pot representar amb el graf de la Figura 2.2.

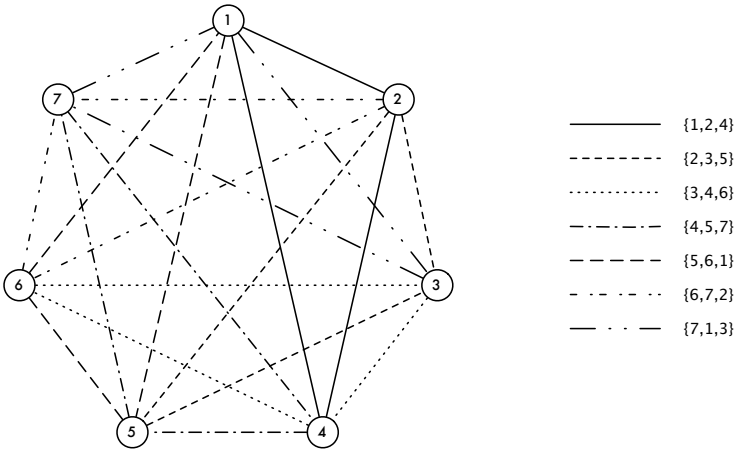


Figura 2.2: Representació gràfica d'un STS

2.5.2 Sistemes quàdruples d'Steiner

Un **sistema quàdruple d'Steiner (SQS)** és una parella ordenada $(V, \mathcal{B})$, on V és un conjunt finit de punts i $\mathcal{B}$ és un conjunt de subconjunts de 4 elements de V, anomenats quàdruples, tal que cada subconjunt de 3 elements diferents de V és un subconjunt d'exactament un quàdruple de $\mathcal{B}$.

L'ordre d'un SQS $(V, \mathcal{B})$ és la mida del conjunt V, denotada per $|V|$. Escrivim $SQS(v)$ per a denotar un SQS d'ordre v.

Només existeixen SQS per $v = 1$ o $v \equiv 2$ o 4 $(mod\ 6)$.

2.5.3 Dissenys combinatoris

Per enters positius $t \leq k \leq v$ i λ, un $t - (v, k, \lambda)$ disseny és una parella $(V, \mathcal{B})$, on V és un conjunt finit de punts i $\mathcal{B}$ és un conjunt de subconjunts de

k elements de V anomenats blocs, tal que qualssevol t punts estan continguts en exactament λ blocs.

Un $t-(v,k,1)$ disseny s'anomena t-disseny d'Steiner. Un $2-(v,k,1)$ disseny tal que $|V| = |\mathcal{B}|$ s'anomena disseny quadrat o simètric.

Exemple 2.10 ***Relació entre dissenys i sistemes d'Steiner***
Un $STS(v)$ és un $2-(v,3,1)$ disseny, i un $SQS(v)$ és un $3-(v,4,1)$ disseny.

2.5.4 Fragments

Donat un $STS(v)$, amb $v \geq 7$, a [Gib76] es defineix un fragment F com a un conjunt de blocs del sistema triple de la següent forma:

$$F = \{\{x_1, x_3, x_5\}, \{x_1, x_4, x_6\}, \{x_2, x_3, x_6\}, \{x_2, x_4, x_5\}\}.$$

Si considerem el conjunt de punts de l'STS, $S = \{x_1, \cdots, x_6\}$, aleshores F és un fragment a S, i es representa per les parelles ordenades $(x_1, x_2), (x_3, x_4)$, (x_5, x_6), anomenades parelles representatives. El fragment F es pot reconstruir totalment a partir de les seves parelles representatives.

Un intercanvi de qualsevol parella representativa constitueix una transformació invariant del sistema triple. Independentment de com s'ordenin les parelles ordenades que formen un fragment, sempre es generarà el mateix fragment. Tot i això, si canviem l'ordre dins les parelles ordenades generarem 4 triples diferents.

Existeixen 80 STS d'ordre 15 no isomorfs, que es poden classificar comptant els seus fragments i observant el patró de quants fragments contenien un punt particular.

2.5.5 Grup d'automorfismes

Grup d'automorfismes d'un disseny

Si $D_1 = (V_1, \mathcal{B}_1)$ i $D_2 = (V_2, \mathcal{B}_2)$ són dos t-dissenys, aleshores una aplicació bijectiva $\alpha : V_1 \longrightarrow V_2$ s'anomena **isomorfisme** de D_1 sobre D_2 si $B \in \mathcal{B}_1 \Longleftrightarrow \alpha(B) \in \mathcal{B}_2$. Aleshores, els dissenys D_1 i D_2 són **isomorfs**.

Un isomorfisme d'un disseny D sobre ell mateix s'anomena **automorfisme**. El conjunt de tots els automorfismes d'un disseny D forma un grup sota composició, el **grup d'automorfismes** de D.

Grup d'automorfismes d'un graf

El grup d'automorfismes d'un graf $G < v|a >$, on v és el conjunt de vèrtexs de G i a el seu conjunt d'arestes, és el conjunt de tots els isomorfismes del graf amb ell mateix, és a dir, de les aplicacions $v \longrightarrow v$, tal que el graf resultant és isomorf amb G.

Dos grafs $G_1 < v_1|a_1 >$ i $G_2 < v_2|a_2 >$ són isomorfs si, i només si, existeix una bijecció $\phi : v_1 \rightarrow v_2$ que conserva les adjacències i les no adjacències. En tal cas, es diu que ϕ és un isomorfisme i $G_1 \cong G_2$.

És una forma de simetria en la qual es preserva la connectivitat entre arestes i vèrtexs.

Grup d'automorfismes d'un codi

El grup d'automorfismes d'un codi és el conjunt de totes les transformacions que fan que el codi, un cop transformat, sigui el mateix. Aquestes transformacions poden ser permutacions de coordenades o translacions del codi (sumar un vector a totes les paraules del codi).

El grup d'automorfismes d'un codi perfecte C coincideix amb el grup d'automorfismes del graf de distàncies mínimes de C. De fet, el codi C es pot reconstruir a partir del seu graf de mínimes distàncies, tal com es demostra a [MÖPS08].

Capítol 3

Planificació del projecte

En aquest capítol aprofundirem en els objectius del projecte que s'han comentat en la introducció. També veurem la planificació temporal establerta al principi del projecte i en què ha derivat finalment.

3.1 Objectius del projecte

L'objectiu principal del projecte és desenvolupar un paquet de funcions per Magma per a treballar amb codis binaris perfectes no lineals, anomenat *BinaryPerfectCodes*. Algunes de les funcions que contindrà el paquet serviran per a

- Fer construccions de codis perfectes (construcció de Vasil'ev, construcció *doubling* i construcció *switching*).
- Fer funcions per a calcular invariants d'un codi binari no lineal, com ara el rang, el nucli, els fragments, el grup d'automorfismes o els STS (Sistemes Triples d'Steiner).
- Construir una base de dades de codis perfectes no equivalents, utilitzant els invariants anteriors.

Per tal d'assolir l'objectiu principal del projecte caldrà complir tot un conjunt de sub-objectius: entendre què són els codis no lineals i com es representen, comprendre què són les construccions de codis perfectes, entendre altres conceptes teòrics (com ara el grup d'automorfismes d'un codi o els STS, entre d'altres) i familiaritzar-se amb l'entorn de treball.

Una gran part d'aquest projecte consistirà a estudiar les bases teòriques que fonamentaran el desenvolupament del paquet. Tanmateix, el projecte també inclou el desenvolupament (disseny, programació, proves i elaboració del manual d'usuari) del *package* per MAGMA.

El paquet *BinaryPerfectCodes* ha de tenir les següents funcionalitats:

- Construccions de codis perfectes:
  - Construcció de tots els codis de Vasil'ev no equivalents de longitud 15.
  - Construcció de Vasil'ev.
  - Construcció *doubling*.
  - Construcció *switching*.
- Base de dades de codis perfectes no equivalents de longitud 15.
- Comprovació de l'equivalència o no equivalència de dos codis perfectes de longitud 15 mitjançant els grups d'automorfismes dels codis.
- Calcular invariants d'un codi:
  - STS.
  - SQS.
  - Grup d'automorfismes.
  - Fragments.

Caldrà desenvolupar tot un seguit de proves per al paquet.

El paquet es desenvoluparà d'una banda respectant les convencions imposades per MAGMA pel que fa a l'estructura de directoris, documentació i proves, i de l'altra respectant els criteris d'estil de programació del CCG, documentats al *CCG Style Guide* [dg09].

La documentació del paquet seguirà el format de MAGMA, que consisteix en manuals en PDF editats en LaTeX. Les funcions s'organitzaran per temes, i s'inclourà una breu introducció teòrica al principi i exemples al final de cada tema.

3.2 Estat de l'art

Els membres del Grup de Combinatòria i Codificació (CCG), del Departament d'Enginyeria de la Informació i de les Comunicacions treballen, entre d'altres projectes, per a estudiar les propietats de famílies de codis no lineals, com ara els codis perfectes, Hadamard o Reed-Muller.

El programa MAGMA no contempla aquest tipus de codis, per la qual cosa va ser necessari desenvolupar un paquet, *BinaryCodes*, expressament per a emmagatzemar i manipular aquests tipus de codis no lineals. El paquet *BinaryCodes* va ser creat pel projectista Víctor Ovalle Arce en el seu projecte final de carrera *"Códigos binarios no lineales en* MAGMA*"* [Ova08], el curs 2007-2008.

En aquest projecte s'utilitzarà el paquet *BinaryCodes* per a poder treballar amb codis no lineals.

D'altra banda, el projectista Joan Cuadros treballarà durant aquest curs sobre el paquet *BinaryCodes*, ampliant-lo per a afegir-hi característiques que permetran que el paquet *BinaryPerfectCodes* pugui aprofitar certes propietats per codis perfectes no lineals.

3.3 Estudi de viabilitat

3.3.1 Viabilitat tècnica

Els requisits indispensables per a poder realitzar el projecte són disposar d'accés a un ordinador amb el programa MAGMA i una llicència en vigor.

El Departament d'Enginyeria de la Informació i de les Comunicacions proporciona als projectistes la possibilitat de treballar en un laboratori amb ordinadors que disposen de connexió a Internet. Des d'aquests ordinadors es pot connectar remotament al servidor *macwilliams*, que té el programa MAGMA instal·lat i en vigor. També es permet accedir als ordinadors del laboratori remotament, sense necessitat d'estar al laboratori de projectistes.

Es disposa de tots aquests recursos, per tant, el projecte és viable tècnicament.

3.3.2 Viabilitat operativa

Per a poder desenvolupar satisfactòriament el projecte caldrà un suport teòric continuat, doncs les funcions que es programaran dins el paquet *BinaryPerfectCodes* requeriran trobar solucions pràctiques a les propietats teòriques dels codis perfectes no lineals. Aquest assessorament teòric és proporcionat pel grup CCG i sobretot per la directora d'aquest projecte, Mercè Villanueva.

D'altra banda, si bé no és un requisit indispensable, es té prevista la integració del paquet *BinaryPerfectCodes* amb el paquet *BinaryCodes* que modificarà el projectista Joan Cuadros. Per tant, l'èxit d'aquesta part del projecte va lligat a l'acompliment dels terminis i els objectius plantejats en el seu projecte per part d'en Joan Cuadros. La planificació d'aquest projecte s'ha fet concordar amb la de l'altre projectista per tal que es puguin integrar els dos paquets sense problemes.

Es disposa de l'assistència del professorat del CCG, per tant en aquest sentit

el projecte és viable. Tanmateix, cal considerar el risc que suposaria que les funcionalitats del paquet *BinaryCodes* necessàries per a la integració dels dos paquets no estiguin acabades a temps.

3.3.3 Viabilitat econòmica

Per a determinar si el projecte és viable econòmicament tindrem en compte les necessitats de *software* i *hardware* per al projecte.

Pel que fa al *software* es necessita una llicència de MAGMA, que té un cost de 1200€ per tres anys, però no cal adquirir-la perquè el Departament d'Enginyeria de la Informació i de les Comunicacions ja en disposa d'una en vigor. La resta d'eines que s'utilitzaran (el sistema operatiu GNU/Linux i l'editor de text Gedit) tenen llicència GPL i no tenen cap cost.

Pel que fa als requeriments de *hardware*, no es requereix cap altre equipament a banda d'un ordinador convencional del qual ja es disposa.

Per tant, no cal fer cap inversió econòmica específica per a aquest projecte, i podem afirmar que aquest és viable econòmicament.

Pressupost

A la Taula 3.1 s'adjunta un pressupost per al projecte, tenint en compte el salari base per als titulats de grau superior, regulat pel conveni col·lectiu d'informàtica [con09], que és d'uns 9.8 €/hora.

3.3.4 Viabilitat legal

MAGMA és un intèrpret orientat al llenguatge matemàtic. És un programa amb llicència privada, del qual el Departament d'Enginyeria de la Informació i de les Comunicacions disposa.

Concepte	**Hores**	**Preu**
Estudi dels fonaments teòrics	28	274.4 €
Elaboració de l'informe previ	3	29.4 €
Disseny de funcions	42.25	414.05 €
Disseny de tests	14	137.2 €
Implementació de funcions en MAGMA	85	833 €
Test	40	392 €
Elaboració de la memòria	70	686 €
TOTAL	282.25	2766.05 €

Taula 3.1: Pressupost per al projecte

El paquet *BinaryPerfectCodes* tindrà llicència de *software* lliure GPLv3. No hi ha incompatibilitats entre les llicències del paquet i del propi MAGMA.

Per tant, el projecte és viable legalment.

3.3.5 Alternatives

Tot i l'existència de programari similar a MAGMA, com ara GAP o SAGE, el grup CCG està treballant amb MAGMA, i per tant s'utilitzarà aquest programa per tal que els investigadors del grup puguin utilitzar el paquet. D'altra banda, el paquet *BinaryCodes* també es va desenvolupar amb MAGMA.

3.4 Planificació temporal del treball

La planificació inicial del treball va començar a finals d'octubre de 2008, quan es van plantejar els objectius i característiques del projecte. La data prevista de finalització, fins a la data d'entrega de la memòria, era el 17 de juny de 2009. En total, es van planificar 280 hores de feina. A la Figura 3.1 es mostra la planificació inicial del projecte i la Taula 3.2 és un resum de les fites més importants.

A la pràctica, però, els objectius del projecte van variar lleugerament, de

Tasca	**Termini**
Presentació del projecte i definició dels objectius	24/10/2008
Estudi dels fonaments teòrics del projecte	Octubre a gener
Elaboració de l'informe previ	Desembre
Entrega de l'informe previ	16/01/2009
Disseny de les funcions del paquet	Finals de desembre - finals de gener
Disseny dels tests del paquet	Finals de gener - Mitjans de març
Codificació de les funcions del paquet	Mitjans de febrer - abril
Test del paquet	Abril
Elaboració de la memòria i la presentació	Finals de novembre - finals de juny (per fases)
Entrega de la memòria	17/06/2009

Taula 3.2: Tasques principals del projecte

manera que:

- Vam decidir desenvolupar només les construccions Vasil'ev i *doubling*, deixant per a futures amplicacions la *switching*.
- Es va substituir tot el bloc de funcions referents a la creació d'una base de dades de codis perfectes no equivalents de longitud 15 per un bloc de funcions complementàries per a treballar amb codis no lineals.

L'eliminació de la construcció *switching* del paquet va ser motivada per la manca de temps, doncs les construccions de Vasil'ev i *doubling* van ocupar la major part de l'etapa de desenvolupament.

La substitució del bloc de base de dades de codis pel grup de funcions de manipulació de codis no lineals es va fer per un major interès en disposar d'aquestes funcions que no pas de les de la base de dades.

A més, l'execució de la planificació temporal del projecte s'ha vist alterada en alguns casos, ja sigui per dificultats en la implementació d'algunes parts com per l'aparició d'imprevistos en la disponibilitat temporal de la projectista.

A la Figura 3.2 es pot veure la planificació real del projecte en contraposi-

ció amb la planificació estimada després del canvi de requisits del paquet. Els blocs de temps de colors més saturats corresponen a l'execució real del projecte, mentre que els blocs de colors més pastel són la planificació inicial. Com es pot veure a la figura, tot i la desviació d'algunes tasques, s'han pogut assolir els objectius plantejats a temps.

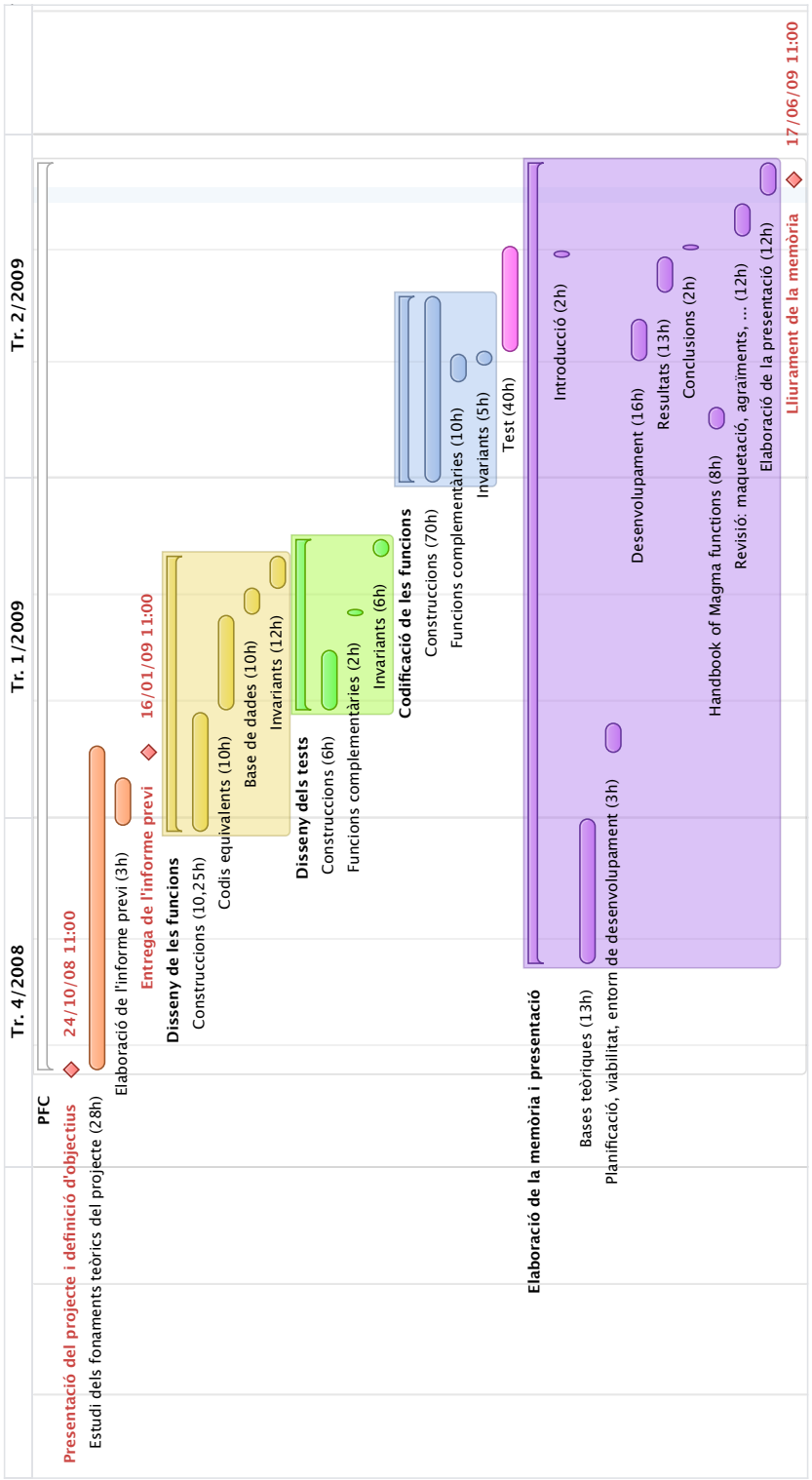


Figura 3.1: Planificació temporal del projecte

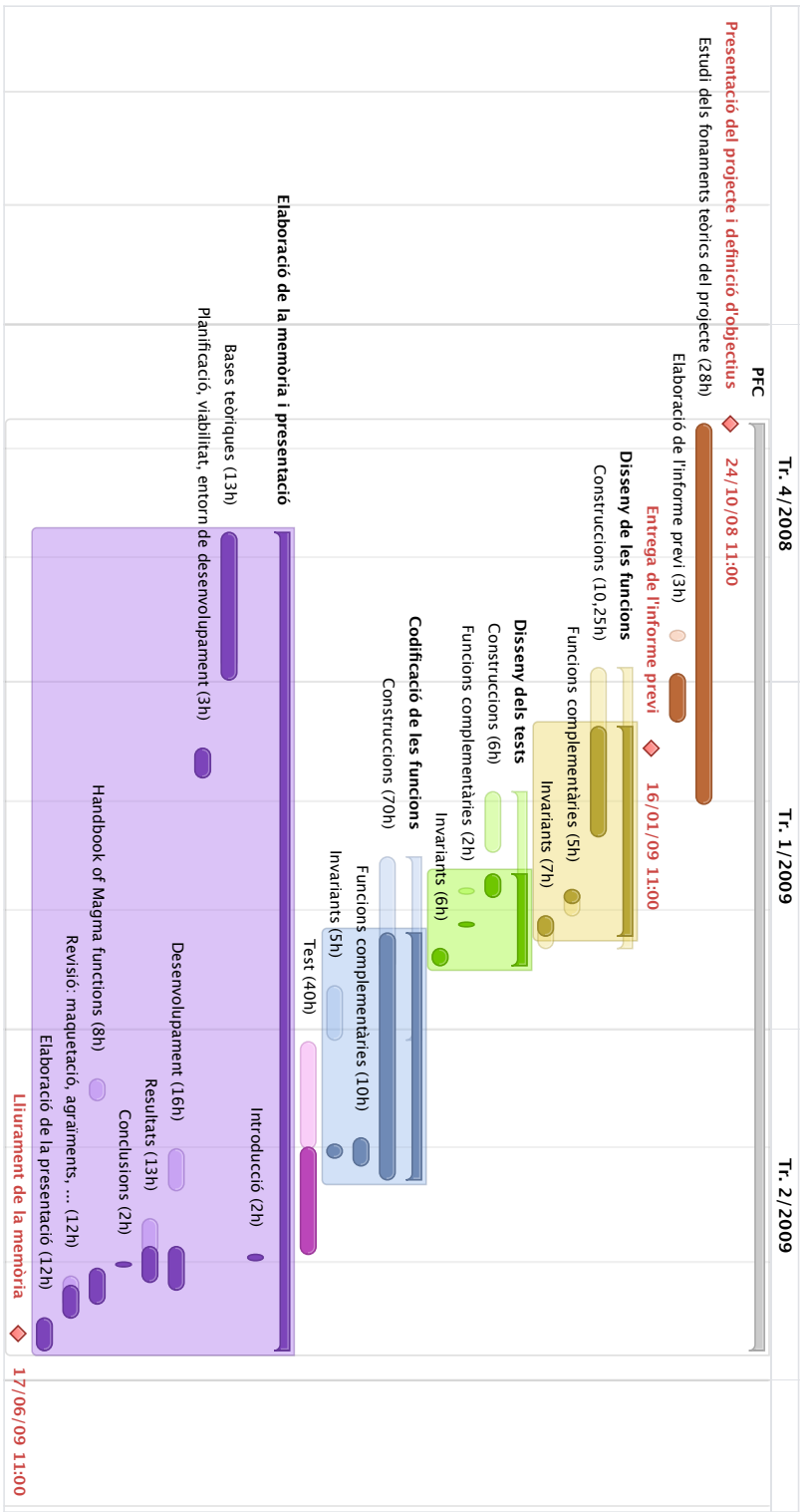


Figura 3.2: Planificació real del projecte

Capítol 4

Desenvolupament del projecte

En aquest capítol es farà una explicació de l'entorn de treball amb el qual s'ha realitzat el projecte i es veuran els detalls de la implementació del paquet per Magma.

4.1 Entorn de desenvolupament

Per a realitzar aquest projecte, el Departament d'Enginyeria de la Informació i de les Comunicacions va posar a disposició de la projectista un ordinador a la sala de projectistes del departament, i un compte en un servidor del Grup de Combinatòria i Codificació.

L'ordinador de la sala de projectistes és un Intel Pentium 4 a 2.00 GHz, amb 2GB de RAM i sistema operatiu Fedora 5. Des de qualsevol ubicació es pot accedir per connexió SSH a aquest ordinador. El servidor Macwilliams també és accessible per SSH, però només des de l'ordinador de projectistes. Aquest servidor té un processador de doble nucli i 2GB de RAM. És en aquest servidor on està instal·lada la còpia de Magma que s'ha utilitzat per a fer el projecte.

Magma [CE08] és un *software* privatiu de gran qualitat desenvolupat per

una universitat australiana. Té moltes funcionalitats en diferents camps, com poden ser la teoria de números, geometria algebraica, teoria de grafs i teoria de codis. El llenguatge que s'utilitza per a treballar amb MAGMA és molt proper al llenguatge formal matemàtic, cosa que el fa especialment adequat als matemàtics. Algunes de les característiques més importants de MAGMA són:

- MAGMA no té interfície gràfica, sinó que és per línia de comandes. Això no ha suposat cap problema en aquest projecte, ja que no es necessitava la interfície gràfica en cap cas.
- MAGMA té un suport limitat per a la computació numèrica i integració simbòlica. L'usuari final no pot definir els seus propis tipus de dades.
- MAGMA permet guardar totes les operacions que s'han fet durant una sessió de treball, però no hi ha cap manera d'emmagatzemar a disc un objecte concret que s'hagi creat.
- La llibreria de MAGMA és gran i molt optimitzada. És molt potent i útil per a la investigació en determinats camps, com ara la teoria de codis. Com que la majoria del programa està escrit en C, es podrien utilitzar les funcionalitats d'algunes llibreries de C existents, però aquest enllaçat només es pot fer recompilant MAGMA, per la qual cosa es requereix el codi font de l'aplicació. En ser un programa privatiu, els usuaris finals no poden fer aquest procés. A més, pel mateix motiu, hi podria haver problemes a causa de restriccions de llicència d'algunes llibreries lliures.

Actualment MAGMA dóna suport a funcionalitats bàsiques per a codis sobre cossos finits i codis sobre anells d'enters i anells de Galois. Tots aquest tipus de codis són lineals. A més, MAGMA disposa de funcions per al cas particular dels codis binaris lineals.

En el projecte [Ova08] es va desenvolupar un paquet per MAGMA, *BinaryCodes*, que permet treballar també amb codis no lineals, i que es farà servir

en aquest projecte.

A més, el projectista Joan Cuadros ha ampliat la funcionalitat del paquet *BinaryCodes* amb algunes funcions que seran necessàries en aquest projecte.

4.2 Implementació del paquet *BinaryPerfectCodes*

El paquet *BinaryPerfectCodes* desenvolupat en aquest projecte es divideix en tres grans blocs de funcions:

- Construcció de codis binaris perfectes no lineals.
- Càlcul d'invariants de codis perfectes.
- Càlculs amb paraules-codi.

La metodologia de desenvolupament que s'ha portat a terme ha estat la següent:

1. Divisió del paquet en diverses funcions.
2. Disseny de cada funció del paquet: decidir el nom, paràmetres d'entrada i retorn de la funció, descripció del funcionament de la funció.
3. Disseny de proves per a cada funció: estudiar com ha de ser el comportament de la funció respecte els paràmetres, i especificar quines condicions i propietats ha de complir el resultat de la funció. A més, en alguns casos ha resultat interessant estudiar el temps d'execució de les funcions.
4. Implementació en Magma de cada funció del paquet i de les seves funcions de prova corresponents.
5. Executar totes les funcions de prova per a cada funció, per tal de comprovar-ne el correcte funcionament.

A continuació aprofundirem en el desenvolupament de cada bloc del paquet *BinaryPerfectCodes*.

4.2.1 Construcció de codis binaris perfectes no lineals

Construcció de Vasil'ev

La construcció de Vasil'ev a partir d'un codi C_n de longitud n es defineix com:

$$C_{2n+1} = \{(v|v+c|\sigma(v)+f(c)) : v \in \mathbb{F}_2^n, c \in C_n\},$$

on $(x|y)$ denota la concatenació d'x i y, $\sigma(v)$ és el pes del vector v mòdul 2, i $f(c)$ és una aplicació $f : C_n \longrightarrow \mathbb{Z}_2$.

La funció *BinaryVasilevCode* es va desenvolupar inicialment seguint l'expressió anterior. Aquesta seria una versió exhaustiva de fer la construcció de Vasil'ev, doncs es generen totes les paraules del codi.

Per a codis de cardinalitat elevada aquesta és una opció inviable, ja que tant el temps de càlcul com la quantitat de memòria necessària per a emmagatzemar totes les paraules-codi són molt grans. El paquet *BinaryCodes* disposa d'una funció que a partir de totes les paraules del codi genera una estructura *BinaryCode* que no emmagatzema totes les paraules-codi, sinó que calcula el súper dual del codi. Per tant, el problema no és en sí emmagatzemar el nou codi Vasil'ev sinó guardar tota la llista de paraules-codi que es passaran al constructor de l'estructura *BinaryCode*.

Per tant, cal buscar una manera alternativa de calcular els codis de Vasil'ev. La solució és estudiar el *kernel* dels codis de Vasil'ev i veure si hi ha alguna particularitat que puguem aprofitar per a construir-los més eficientment.

Efectivament, els codis de Vasil'ev compleixen les següents propietats:

- Si la imatge de la funció f val zero per a qualsevol paraula del codi C_n,

aleshores coneixem tot el *kernel* del codi C_{2n+1}:

$$K(C_{2n+1}) = \{\{(v|v|\sigma(v)) : v \in \mathbb{Z}_2^n\} \cup \{(\mathbf{0}|k|0) : k \in K(C_n)\}\}$$

- Si la imatge de la funció f és $f(c) = b_i \ \forall c \in K(C_n) + x_i$, és a dir, és la mateixa per a totes les paraules-codi pertanyents a una mateixa classe del *kernel* del codi C_n, aleshores el *kernel* del codi C_{2n+1} conté el conjunt anterior

$$\{\{(v|v|\sigma(v)) : v \in \mathbb{Z}_2^n\} \cup \{(\mathbf{0}|k|0) : k \in K(C_n)\}\} \subseteq K(C_{2n+1}).$$

- Si la funció f és qualsevol, aleshores el *kernel* del nou codi C_{2n+1} conté el conjunt

$$\{(v|v|\sigma(v)) : v \in \mathbb{Z}_2^n\} \subseteq K(C_{2n+1}).$$

D'aquí ve la necessitat de disposar d'una funció capaç de construir una estructura *BinaryCode* a partir d'un subconjunt del *kernel* i un subconjunt dels líders del nou codi. Aquesta funció l'ha fet el projectista Joan Cuadros en la seva ampliació del paquet *BinaryCodes*.

Aprofitant aquestes propietats dels codis de Vasil'ev es va desenvolupar la versió no-exhaustiva de la funció *BinaryVasilevCode*.

A més de la construcció de Vasil'ev en general, es va fer una funció per a construir els 19 codis de Vasil'ev de longitud 15 no equivalents a partir de la classificació de Hergert [Her82]. Aquesta funció consisteix en l'adaptació a Magma d'un algorisme desenvolupat per en Jaume Pujol en GAP.

Construcció *doubling*

La construcció *doubling* a partir de dos codis C_1 i C_2 de longitud n, i una permutació de coordenades π es defineix com

$$C = (C_1 \oplus C_2^*) \bigcup_{i=1}^{n} [(C_1 + e_i) \oplus (C_2 + e_{\pi(i)})^*],$$

on e_i és un vector les coordenades del qual són totes zero excepte la i-èsima que val 1.

La funció *BinaryDoublingCode* del paquet *BinaryPerfectCodes* es va desenvolupar inicialment seguint aquesta expressió matemàtica. Aquesta versió és exhaustiva, donat que es generen totes les paraules del codi. Com es pot intuir, això és inviable per a codis de cardinalitat elevada, tant pel que fa al temps de càlcul com a les necessitats d'emmagatzematge.

Així doncs, es van estudiar les propietats dels codis *doubling* per tal de trobar alguna possible millora. Si observem les característiques del *kernel* dels codis *doubling*, en funció dels paràmetres amb el qual els construïm, veiem que:

- Si la permutació és la identitat, $\pi = Id$, aleshores coneixem tot el *kernel* del nou codi:

  $$K(C) = (K_1 \oplus K_2^*) \bigcup_{i \in I} [(K_1 + e_i) \oplus (K_2 + e_{\pi(i)})^*],$$

  on K_1 és el *kernel* de C_1, K_2 el *kernel* de C_2, T_i denota el subespai lineal generat per les paraules de pes mínim de C que tenen un 1 a la coordenada i-èsima, i $I = \{i : T_i \subseteq K_1 \cap K_2\}$.

- Si la permutació no és la identitat, aleshores el *kernel* del nou codi C conté

  $$(K_1 \oplus K_2^*) \subseteq K(C).$$

Aprofitant que coneixem algunes paraules-codi que formen part del *kernel*

del nou codi, s'ha desenvolupat una versió no-exhaustiva de la funció *BinaryDoublingCode*.

4.2.2 Càlcul d'invariants de codis perfectes

Dins el bloc de funcions per al càlcul d'invariants de codis perfectes incloem les següents funcions:

- Càlcul dels STS i SQS d'un codi perfecte i perfecte estès, respectivament.
- Càlcul del grup d'automorfismes d'un codi perfecte.

Les funcions de càlcul dels sistemes d'Steiner (STS i SQS) d'un codi han estat senzilles d'implementar, perquè el MAGMA ja disposa de funcions per a calular t-dissenys a partir d'un conjunt de vectors. Així doncs, les funcions de càlcul dels STS i SQS calculen un disseny a partir de les paraules de pes mínim del codi.

El grup d'automorfismes d'un codi perfecte es pot calcular a partir del seu graf de mínima distància [MÖPS08]. MAGMA disposa d'una funció per a calcular el grup d'automorfismes d'un graf, i com s'ha vist al Capítol 2, el grup d'automorfismes d'un codi perfecte coincideix amb el del seu graf de distàncies mínimes. Per tant, la feina per aquesta funció ha estat construir el graf de mínimes distàncies d'un codi, que veurem a la Secció 4.2.3.

4.2.3 Càlculs amb paraules-codi

Les funcions per a treballar amb paraules-codi que s'han desenvolupat són les següents:

- Obtenir el conjunt de totes les paraules d'un codi.

- Obtenir un conjunt de paraules-codi d'un cert pes, i calcular el nombre de paraules amb aquest pes que hi ha al codi.

- Calcular el pes mínim d'un codi.

- Obtenir una paraula de pes mínim, o totes les paraules de pes mínim d'un codi.

- Construir el graf de mínimes distàncies d'un codi.

El graf de mínimes distàncies d'un codi és un graf tal que els seus vèrtexs es corresponen amb cada una de les paraules del codi, i hi ha una aresta que uneix dues paraules-codi si la distància entre elles es correspon amb la distància mínima del codi.

Exemple 4.1 ***Graf de mínima distància d'un codi***
Suposem que tenim un codi de Hamming C de longitud 7, format per les següents paraules-codi:

```
(1 1 1 0 0 1 0) (0 1 1 1 0 0 1) (0 0 1 0 1 1 1) (1 1 1 1 1 1 1)
(1 0 1 0 0 0 1) (0 1 1 0 1 0 0) (1 0 1 1 1 0 0) (0 0 1 1 0 1 0)
(0 1 0 0 0 1 1) (1 1 0 0 1 0 1) (0 1 0 1 1 1 0) (1 0 0 1 0 1 1)
(1 1 0 1 0 0 0) (0 0 0 0 0 0 0) (0 0 0 1 1 0 1) (1 0 0 0 1 1 0)
```

Podem construir el seu graf de mínima distància, que es pot representar gràficament com a la Figura 4.1.

A banda d'aquestes funcions, s'ha ampliat la funció *BinaryMinimumDistance* del paquet *BinaryCodes* per tal que si el codi és invariant per distància (com és el cas dels codis perfectes) es pugui calcular la distància mínima a partir del pes mínim, un càlcul bastant més lleuger que el de la distància mínima.

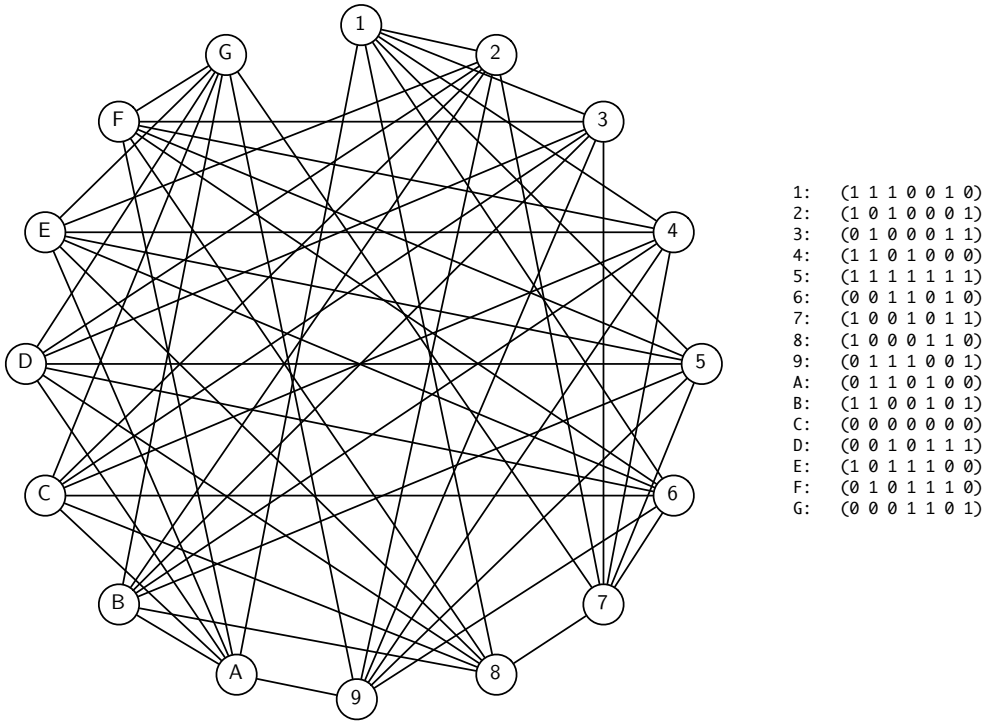


Figura 4.1: Graf de mínima distància d'un codi de Hamming de longitud 7.

4.3 Proves del paquet *BinaryPerfectCodes*

Un cop creat el paquet de funcions per a MAGMA, cal sotmetre'l a un procés de test, un conjunt de proves que permet validar el funcionament del *software* desenvolupat i determinar paràmetres com la qualitat, la fiabilitat, l'estabilitat, l'escalabilitat i la seguretat, entre d'altres. Tanmateix, per exhaustiu que sigui el procés de test, no es pot assegurar totalment que el *software* no tingui cap mal funcionament.

Segons el que es vulgui provar, hi ha diversos tipus de tests: de caixa blanca, de caixa negra, de regressió, d'unitat, de sistema, ...

En aquest projecte s'han fet, principalment, tests de caixa negra, que comproven que la funció a la qual es sotmet a prova retorni el que s'espera que ha de retornar, a partir d'unes certes entrades.

S'han implementat diverses funcions de proves per a comprovar que els codis construïts compleixen les propietats que s'espera que compleixin.

Per a qualsevol dels codis construïts, tant amb la construcció de Vasil'ev com amb la *doubling*, cal provar que:

- El codi resultant sigui perfecte, és a dir:
  - La cardinalitat del codi ha de ser 2^{n-m}, on $n = 2^m - 1$ és la longitud del codi k és la dimensió del *kernel*.
  - La seva distància mínima i el seu pes mínim han de ser 3.
- Si el codi d'entrada és de longitud n, el de sortida ha de ser de longitud $2n + 1$.
- Els codis generats exhaustivament o no exhaustivament han de ser iguals.

A més, per a les funcions de Vasil'ev, concretament haurem de comprovar que el resultat sigui lineal si el codi d'entrada és lineal i la funció d'entrada

és zero.

Per a les funcions relatives a la construcció *doubling*, hem de comprovar que el codi resultant ha de ser lineal si els codis d'entrada són lineals i la permutació és la identitat.

Per a la resta de funcions, que tenen un resultat més senzill i amb molts menys paràmetres, les proves han consistit en comparar el resultat obtingut amb el resultat esperat, que en molts casos ja es coneixia. Per exemple, per a provar les funcions de mínima distància, si el codi d'entrada és perfecte, ja sabem que la sortida ha de ser 3.

A continuació es mostra el resultat d'una prova d'exemple qualsevol. Concretament correspon al cas de la funció *BinaryVasilevN15Code*, que construeix els 19 codis de Vasil'ev de longitud 15 no equivalents segons la classificació de Hergert.

En aquest cas, en tractar-se d'una funció d'exemple, la validació que els resultats són correctes l'hem de fer manualment, mirant que el resultat real coincideixi amb el resultat que s'espera. A la pràctica, però, els tests que s'han desenvolupat per al paquet comproven automàticament que el resultat sigui correcte, mitjançant la comanda "`assert Output eq ExpectedOutput;`" de MAGMA.

El codi de la funció de prova genera tots els codis de Vasil'ev de longitud 15 no equivalents, i mostra els paràmetres dels codis generats:

```
procedure testVasilevN15()
print "== VasilevN15 Test";
  for i in [0..18] do
      print "=== Vasilev Code: ",i;
      c := BinaryVasilevN15Code(i);

      length := c`Length;
      cardinal := #BinarySet(c);
      perfect:=IsBinaryPerfectCode(c);
      linear := IsBinaryLinearCode(c);
```

```
      minDistance := BinaryMinimumDistance(c);

      print "Param   | Expected | Real";
      print "=========================";
      print "L       | 15       | ",length;
      print "#       | 2048     | ",cardinal;
      print "Perfect | true     | ",perfect;
      if i eq 0 then
      print "Linear  | true     | ",linear;
      else
      print "Linear  | false    | ",linear;
      end if;
      print "minDist | 3        | ",minDistance;
  end for;
end procedure;
```

I els resultats obtinguts són, tal i com podem veure a partir de la sortida del test, correctes:

```
== VasilevN15 Test

=== Vasilev Code:  0                 ...  === Vasilev Code:  18
Param   | Expected | Real                 Param   | Expected | Real
=========================               =========================
L       | 15       | 15                   L       | 15       | 15
#       | 2048     | 2048                 #       | 2048     | 2048
Perfect | true     | true                 Perfect | true     | true
Linear  | true     | true                 Linear  | false    | false
minDist | 3        | 3                    minDist | 3        | 3
```

Capítol 5

Handbook of Magma functions

5.1 Introduction

A code $C \subseteq \mathbb{F}_2^n$ is said to be **perfect** if its covering radius is equal to its error correction capacity. In other words, a code C of length n is perfect if, for some $e \geq 0$, any element of $\mathbb{F}_2^n$ has distance less than or equal to e from exactly one codeword in C. A perfect code is called e-perfect, because it can correct e errors.

Perfect codes may be linear or nonlinear. Some properties on perfect codes, valid for both linear and nonlinear codes, are:

- Perfect codes are distance invariant, that is, its weight distribution is equal to its distance distribution. This property is very useful, because the weight distribution is faster to compute than the distance distribution.
- Their minimum distance is equal to their minimum weight, since perfect codes are distance invariant.
- 1-perfect codes have length $n = 2^m - 1$, 2^{n-m} codewords, and its minimum distance is 3.

From now on, in this chapter, the term "perfect code"will refer to binary 1-perfect codes, linear or nonlinear, unless otherwise specified.

MAGMA currently has functions to deal with linear perfect codes, but not with nonlinear perfect codes. This chapter describes functions to construct and work with binary 1-perfect codes. These functions have been developed as a separate package for MAGMA, called *BinaryPerfectCodes*.

5.2 Construction of binary perfect codes

`BinaryVasilevCode(C, f)`

Given a binary 1-perfect code C of length n and a function f, return the binary 1-perfect code D of length $2n+1$ given by the Vasil'ev construction: $D = \{(v|v+c|f(c)+p(c)) : v \in \mathbb{Z}_2^n, c \in C\}$, where $p(c)$ is the even parity bit of c and f is a function $f : C \to \mathbb{Z}_2$.

`BinaryVasilevCode(C, f)`

Given a binary 1-perfect code C of length n and a sequence of integer numbers, return the binary 1-perfect code D of length $2n+1$ given by the Vasil'ev construction: $D = \{(v|v+c|f(c)+p(c)) : v \in \mathbb{Z}_2^n, c \in C\}$, where $p(c)$ is the even parity bit of c and f is a function $f : C \to \mathbb{Z}_2$. The given sequence of integer numbers correspond to the images by the function f for all the different cosets of C using its kernel.

`BinaryVasilevN15Code(g)`

Given an integer g from 0 to 18, return the g-th Vasil'ev code of length 15.

`BinaryDoublingCode(C, D, p)`

Given two binary 1-perfect codes, C and D, both of length n, and a permutation p, return the binary 1-perfect code E of length $2n + 1$ given by the doubling construction: $E = (C \oplus D^*) \bigcup_{i=1}^{n} (C + e_i) \oplus (D + e_{p(i)})^*$, where e_i is a vector which is all zero except the i-th coordinate that is 1, and D^* is the extended code of D.

Example H5E1

```
C := BinaryCode(HammingCode(GF(2), 3));
C‘Length;
7
f := function(x) return 0; end function;

D := BinaryVasilevCode(C, f);
D‘Length;
15
BinaryCardinal(D);
2048
BinaryMinimumDistance(D);
3
IsBinaryPerfectCode(D);
true
IsBinaryLinearCode(D);
true
D‘Kernel;
[15, 11, 3] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 1 1 0 1]
[0 1 0 0 0 0 0 0 0 0 0 0 1 1 1]
[0 0 1 0 0 0 0 0 0 0 0 1 1 1 1]
[0 0 0 1 0 0 0 0 0 0 0 1 0 1 1]
[0 0 0 0 1 0 0 0 0 0 0 1 0 0 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 0 1]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 0 1 0 0 0 1 1 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 1 1 0]
[0 0 0 0 0 0 0 0 0 1 0 1 1 1 0]
[0 0 0 0 0 0 0 0 0 0 1 1 0 1 0]
```

```
D‘CosetLeaders;
[]
```

Example H5E2

```
D := BinaryVasilevN15Code(3);
D‘Length;
15
BinaryCardinal(D);
2048
BinaryMinimumDistance(D);
3
IsBinaryPerfectCode(D);
true
IsBinaryLinearCode(D);
false
D‘Kernel;
[15, 7, 3] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 0 1 1]
[0 1 0 0 0 0 0 0 0 0 0 0 1 0 1]
[0 0 1 0 0 0 0 0 0 0 0 1 0 0 1]
[0 0 0 1 0 0 0 0 0 0 1 0 0 0 1]
[0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
[0 0 0 0 0 1 0 0 1 0 0 0 0 0 1]
[0 0 0 0 0 0 1 1 0 0 0 0 0 0 1]
D‘CosetLeaders;
[
   (1 1 0 1 0 0 1 0 0 0 0 0 0 0 0),
   (0 0 1 1 0 0 1 0 0 0 0 0 0 0 0),
   (1 0 1 0 1 0 1 0 0 0 0 0 0 0 0),
   (0 1 1 0 0 1 1 0 0 0 0 0 0 0 0),
   (0 0 0 1 1 1 1 0 0 0 0 0 0 0 0),
   (0 1 0 0 1 0 0 1 0 0 0 0 0 0 0),
   (1 0 0 0 0 1 0 1 0 0 0 0 0 0 0),
   (1 1 1 1 1 1 0 1 0 0 0 0 0 0 0),
   (1 1 1 0 0 0 1 1 0 0 0 0 0 0 0),
   (1 0 0 1 1 0 1 1 0 0 0 0 0 0 0),
```

```
   (0 1 1 1 1 0 1 1 0 0 0 0 0 0 0),
   (0 1 0 1 0 1 1 1 0 0 0 0 0 0 0),
   (1 0 1 1 0 1 1 1 0 0 0 0 0 0 0),
   (1 1 0 0 1 1 1 1 0 0 0 0 0 0 0),
   (0 0 1 0 1 1 1 1 0 0 0 0 0 0 0)
]
```

Example H5E3

```
C := BinaryCode(HammingCode(GF(2),3));
p := Sym(c‘Length)!(1,2);

E := BinaryDoublingCode(C,C,p);
E;
[18, 6, 6] Linear Code over GF(2)
Generator matrix:
[1 0 0 1 0 1 1 0 1 0 0 0 1 1 1 1 1 1]
[0 1 0 1 1 1 0 0 1 0 0 0 1 1 1 0 0 1]
[0 0 1 0 1 1 1 0 0 0 0 0 0 0 0 1 0 1]
[0 0 0 0 0 0 0 1 1 0 0 1 0 1 0 1 1 0]
[0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 1 0 1]
[0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 1 1]
E‘Length;
15
BinaryCardinal(E);
2048
BinaryMinimumDistance(E);
3
IsBinaryPerfectCode(E);
true
IsBinaryLinearCode(E);
false
```

5.3 Invariants of binary perfect codes

`BinarySTS(C)`

Given a binary 1-perfect code C, return the Steiner triple system associated to the codewords of weight 3 of C.

`BinarySTS(C, u)`

Given a binary 1-perfect code C and a codeword u, return the Steiner triple system associated to the codewords of weight 3 of $C + u$.

`BinarySQS(C)`

Given a binary 1-perfect code C, return the Steiner quadruple system associated to the codewords of weight 4 of the extended code of C.

`BinarySQS(C, u)`

Given a binary 1-perfect code C and a codeword u, return the Steiner quadruple system associated to the codewords of weight 4 of the extended code of $C + u$.

`BinaryAutomorphismGroup(C)`

Given a binary 1-perfect code C, return the automorphism group of C.

Example H5E4

```
C := BinaryCode(HammingCode(GF(2),3));
p := Sym(C`Length)!(1,2);
D := BinaryDoublingCode(C,C,p);

BinarySTS(D);
2-(15, 3, 1) Design with 35 blocks
Point-set of 2-(15, 3, 1) Design with 35 blocks
Block-set of 2-(15, 3, 1) Design with 35 blocks

BinarySTS(D, BinaryMinimumWord(D));
```

```
2-(15, 3, 1) Design with 35 blocks
Point-set of 2-(15, 3, 1) Design with 35 blocks
Block-set of 2-(15, 3, 1) Design with 35 blocks

BinarySQS(D);
3-(16, 4, 1) Design with 140 blocks
Point-set of 3-(16, 4, 1) Design with 140 blocks
Block-set of 3-(16, 4, 1) Design with 140 blocks

BinarySQS(D,BinaryMinimumWord(D));
3-(16, 4, 1) Design with 140 blocks
Point-set of 3-(16, 4, 1) Design with 140 blocks
Block-set of 3-(16, 4, 1) Design with 140 blocks

E := BinaryVasilevN15Code(1);
BinarySTS(E);
2-(15, 3, 1) Design with 35 blocks
Point-set of 2-(15, 3, 1) Design with 35 blocks
Block-set of 2-(15, 3, 1) Design with 35 blocks
BinarySQS(E);
3-(16, 4, 1) Design with 140 blocks
Point-set of 3-(16, 4, 1) Design with 140 blocks
Block-set of 3-(16, 4, 1) Design with 140 blocks



BinaryAutomorphismGroup(D);
Permutation group acting on a set of cardinality 2048
Order = 393216 = 2^17 * 3
GSet{@ (1 1 1 0 1 1 0 1 0 1 0 1 0 1 0), (0 1 0 0 0 1 1 1 0 1 0 0 0 1 1),
..., {(1 1 0 0 1 0 1 1 0 1 0 0 0 1 1), (0 0 0 1 1 0 1 1 0 1 0 0 0 1 1)} @}
Set of all automorphisms of Graph
Vertex  Neighbours

1       7 26 50 ... 2034 2038 ;
2       9 17 53 ... 2039 2043 ;
...
2048    3 10 29 ... 1929 2035 ;
```

```
Mapping from: GrpPerm: $, Degree 2048, Order 2^17 * 3 to
Set of all automorphisms of Graph
Vertex  Neighbours

1       7 26 50 ... 2034 2038 ;
2       9 17 53 ... 2039 2043 ;
...
2048    3 10 29 ... 1929 2035 ;

Graph
Vertex  Neighbours

1       2 3 4 ... 36 ;
2       1 1738 ... 1976 ;
...
2048    8 9 14 ...  1660 ;
```

5.4 Operations on codewords

`BinarySet(C)`

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

`BinaryWords(C, w)`

Given a binary code C and an integer w, return the set of all words of C having weight w.

`BinaryNumberOfWords(C, w)`

Given a binary code C and an integer w, return the number of words of C having weight w.

`BinaryMinimumWeight(C)`

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

`BinaryMinimumWord(C)`

Given a binary code C, return one word of C having minimum weight.

`BinaryMinimumWords(C)`

Given a binary code C, return the set of all words of C having minimum weight.

`BinaryMinimumWords(C, u)`

Given a binary code C and a codeword u, return the set of all words of $C+u$ having minimum weight.

`BinaryMinimumDistanceGraph(C)`

Given a binary code C, return its minimum distance graph. The minimum distance graph of a code is a graph G, where the vertices set is the set of words of C, and two vertices are connected by an edge if the distance between them is the minimum distance of C.

Example H5E5

```
C := BinaryVasilevN15Code(2);
BinarySet(C);
{
   (1 1 0 0 0 0 1 1 0 1 1 1 0 1 1),
   (0 1 0 1 1 1 0 0 1 1 1 1 0 1 0),
   ...
   (0 0 0 1 1 0 1 0 1 0 0 1 1 1 1)
}

#BinaryWords(C,12) eq BinaryNumberOfWords(E,12);
true

BinaryMinimumWords(C) eq BinaryWords(E,BinaryMinimuWeight(E));
true

BinaryMinimumWeight(C);
```

```
3
BinaryMinimumWord(C);
(1 0 1 0 0 0 0 0 0 0 0 0 1 0 0)

BinaryNumberOfWords(c,12);
35

BinaryMinimumDistanceGraph(c);
Graph
Vertex  Neighbours
(1 1 0 0 0 0 1 1 0 1 1 1 0 1 1) (1 1 0 0 0 0 1 0 1 1 1 1 0 0 1)
                              ... (1 1 0 1 1 0 1 1 0 1 1 1 0 0 1) ;
(0 1 0 1 1 1 0 0 1 1 1 1 0 1 0) (0 0 0 1 0 1 1 0 1 1 1 1 0 1 0)
... (0 1 0 1 1 0 1 0 1 1 1 1 0 0 0) ;
...
(0 0 0 1 1 0 1 0 1 0 0 1 1 1 1) (0 0 1 1 0 1 1 0 1 0 0 1 1 1 1)
... (0 1 1 1 1 0 1 0 1 0 0 1 1 0 1) ;
Vertex-set of Graph
Vertex  Neighbours
(1 1 0 0 0 0 1 1 0 1 1 1 0 1 1) (1 1 0 0 0 0 1 0 1 1 1 1 0 0 1)
... (1 1 0 1 1 0 1 1 0 1 1 1 0 0 1) ;
(0 1 0 1 1 1 0 0 1 1 1 1 0 1 0) (0 0 0 1 0 1 1 0 1 1 1 1 0 1 0)
... (0 1 0 1 1 0 1 0 1 1 1 1 0 0 0) ;
...
(0 0 0 1 1 0 1 0 1 0 0 1 1 1 1) (0 0 1 1 0 1 1 0 1 0 0 1 1 1 1)
... (0 1 1 1 1 0 1 0 1 0 0 1 1 0 1) ;

Edge-set of Graph
Vertex  Neighbours
(1 1 0 0 0 0 1 1 0 1 1 1 0 1 1) (1 1 0 0 0 0 1 0 1 1 1 1 0 0 1)
... (1 1 0 1 1 0 1 1 0 1 1 1 0 0 1) ;
(0 1 0 1 1 1 0 0 1 1 1 1 0 1 0) (0 0 0 1 0 1 1 0 1 1 1 1 0 1 0)
... (0 1 0 1 1 0 1 0 1 1 1 1 0 0 0) ;
...
(0 0 0 1 1 0 1 0 1 0 0 1 1 1 1) (0 0 1 1 0 1 1 0 1 0 0 1 1 1 1)
... (0 1 1 1 1 0 1 0 1 0 0 1 1 0 1) ;
```

Capítol 6

Resultats

En aquest capítol es veuran els resultats obtinguts a partir del desenvolupament del paquet per a MAGMA.

S'ha implementat un paquet de *software* per a MAGMA que serveix per a treballar amb codis binaris perfectes no lineals. Aprofitant el paquet *BinaryCodes* desenvolupat en un projecte el curs passat [Ova08] i ampliat aquest curs en un altre projecte, s'ha pogut treballar amb codis no lineals amb MAGMA, ja que de sèrie aquest programa no té aquesta funcionalitat.

S'analitzarà el funcionament de les funcions de construcció de codis Vasil'ev i *doubling*, ja que aquestes conformen el bloc principal de funcions del paquet. La resta de funcions implementades són, principalment, generalitzacions o adaptacions de funcions ja existents en MAGMA per a codis lineals, per tal que acceptin també codis no lineals, representats a través de l'estructura de súper dual.

6.1 Construcció de Vasil'ev

Com s'ha explicat al Capítol 4, s'ha desenvolupat dues versions per a la funció *BinaryVasilevCode*, una que construeix el codi exhaustivament (és

a dir, generant totes les seves paraules-codi) i una altra que aprofita les propietats del *kernel* dels codis de Vasil'ev, i construeix el nou codi a partir de les paraules-codi que coneixem del *kernel*.

A la Taula 6.1 es mostra un conjunt de casos d'execució per a diferents paràmetres d'entrada i per les dues versions de la funció.

Teòricament, la versió no exhaustiva de la funció de Vasil'ev hauria de ser més ràpida i eficient que la versió exhaustiva, o en el pitjor dels casos, igual. A la pràctica, però, resulta que la versió no exhaustiva utilitza una funció del paquet *BinaryCodes* que no és tan eficient com podria ser, i per tant, per a casos de codis petits, com ara els de longitud 7, resulta més ràpida la versió exhaustiva que l'altra.

En canvi, observem que per a construir codis de Vasil'ev de longitud 31 a partir de codis de longitud 15, cas en què el codi resultant tindria $2^{27} = 134217728$ paraules-codi, s'exhaureix tota la memòria virtual del sistema abans que es puguin acabar de construir les paraules del codi. Per tant, per a codis amb una cardinalitat elevada, és inviable construir els codi de Vasil'ev de manera exhaustiva.

Utilitzant la versió no exhaustiva de la funció *BinaryVasilevCode* es poden arribar a construir codis de longitud 31, si com a paràmetres d'entrada utilitzem un codi de longitud 15 i una funció que, o bé retorna zero per a totes les paraules del codi, o bé retorna el mateix valor per a tots els vectors continguts en la mateixa classe del codi, i coneixem la imatge d'aquesta funció per a les classes del codi.

Tanmateix, observem a la Taula 6.1 com, quan els paràmetres d'entrada són un codi de longitud 15 i una funció arbitrària, el temps de generació de les paraules-codi del *kernel* és baix, però el temps de construcció del codi binari resultant, amb l'estructura del súper dual implementada al paquet *BinaryCodes*, és molt elevat. S'ha fet una estimació del temps necessari per a construir el codi, segons la complexitat de l'algorisme i els paràmetres del

codi amb el qual s'ha fet la prova, i s'ha trobat que, pel cap baix, seria necessari un any i mig de càlcul per a poder veure el resultat. Per tant, no té sentit utilitzar aquesta funció per a construir codis de longitud 31 a partir dels paràmetres d'entrada esmentats, mentre no es millori l'eficiència de la funció *BinaryCode*.

6.2 Construcció *doubling*

Igual que en el cas de la funció de Vasil'ev, per a la construcció *doubling* també s'han desenvolupat dues versions: una que construeix rel codi de manera exhaustiva i una que el construeix a partir del conjunt de paraules-codi que sabem que pertanyen al *kernel*.

Aquesta segona versió no exhaustiva és més ràpida i eficient que l'exhaustiva, tot i que per passar de codis de longitud 15 a 31 el temps de càlcul és molt elevat, a causa de la funció *BinaryCodes*. Com es pot veure a la Taula 6.2, on es mostren temps de diverses execucions de la funció *BinaryDoublingCode*, el temps de generació de totes les paraules-codi que coneixem del *kernel* i els líders de les classes, per a codis d'entrada de longitud 15 és raonable, però el temps que es triga en acabar de completar el *kernel*, és a dir, acabar de construir el codi, és molt elevat. El temps que es mostra a la taula per a aquest cas és una estimació, tenint en compte quins són els paràmetres d'entrada i la complexitat de la funció *BinaryDoublingCode*.

Paràmetres d'entrada	**Versió funció**	**Temps 1**	**Temps 2**
• Hamming n=7 • Funció arbitrària	No exhaustiva	0.000 s	0.010 s
• Hamming n=7 • Funció arbitrària	Exhaustiva	0.030 s	0.050 s
• Hamming n=7 • Funció zero	No exhaustiva	0.000 s	0.010 s
• Hamming n=7 • Funció zero	Exhaustiva	0.040 s	0.050 s
• Vasil'ev n=15 • Funció arbitrària	No exhaustiva	0.040 s	$\sim$ 15360 h
• Vasil'ev n=15 • Funció arbitrària	Exhaustiva	No es poden generar totes les paraules-codi	No aplicable
• Vasil'ev n=15 • Funció zero	No exhaustiva	0.010 s	1363.880 s
• Vasil'ev n=15 • Funció zero	Exhaustiva	No es poden generar totes les paraules-codi	No aplicable
• Vasil'ev n=15 • Funció tal que es coneix la seva imatge per a cada classe del codi	No exhaustiva	0.030 s	657.480 s

Llegenda

Temps 1: temps emprat en generar les paraules-codi del codi (versió exhaustiva) o les paraules-codi conegudes del *kernel* (versió no exhaustiva).

Temps 2: temps total de generació del codi.

Taula 6.1: Comparació de temps d'execució de la funció *BinaryVasilevCode*

Paràmetres d'entrada	**Versió funció**	**Temps 1**	**Temps 2**
• C i D: Hamming n=7 • Permutació arbitrària	No exhaustiva	0.030 s	0.250 s
• C i D: Hamming n=7 • Permutació arbitrària	Exhaustiva	0.030 s	0.250 s
• C i D: Hamming n=7 • Permutació identitat	No exhaustiva	0.030 s	0.040 s
• C i D: Hamming n=7 • Permutació identitat	Exhaustiva	0.040 s	0.050 s
• C i D: Vasil'ev n=15 • Permutació arbitrària	No exhaustiva	4.400 s	$\sim$ 14 h
• C i D: Vasil'ev n=15 • Permutació arbitrària	Exhaustiva	No es poden generar totes les paraules-codi	No aplicable
• C i D: Vasil'ev n=15 • Permutació identitat	No exhaustiva	4.370 s	$\sim$ 14 h
• C i D: Vasil'ev n=15 • Permutació identitat	Exhaustiva	No es poden generar totes les paraules-codi	No aplicable

Llegenda

Temps 1: temps emprat en generar les paraules-codi del codi (versió exhaustiva) o les paraules-codi conegudes del *kernel* (versió no exhaustiva).

Temps 2: temps total de generació del codi.

Taula 6.2: Comparació de temps d'execució de la funció *BinaryDoublingCode*

Capítol 7

Conclusions

A continuació veurem les conclusions extretes de la realització d'aquest projecte, així com propostes de millora del *software* desenvolupat.

7.1 Assoliment d'objectius

La conclusió principal del projecte és que s'han assolit els objectius plantejats. Si bé no els inicials, sí que s'ha aconseguit fer tota la feina proposada en la revisió dels objectius. Un enginyer ha de saber adaptar-se i reconduir les situacions canviants, i la realitat és que sovint els projectes pateixen canvis de requisits per causes diverses; és feina de l'enginyer aconseguir superar aquestes circumstàncies i satisfer les expectatives del client o del destinatari del projecte.

A continuació es revisen els objectius del projecte i el grau d'assoliment de cada un d'ells:

1. Estudi de les bases teòriques necessàries per a la comprensió i posterior implementació del paquet de funcions per a Magma. Aquest objectiu s'ha assolit en la seva totalitat, ja que en cas contrari hauria estat impossible desenvolupar les funcions del paquet.

2. Implementació d'un paquet de funcions per a Magma que permet treballar amb codis binaris perfectes no lineals. Desenvolupament de funcions que aprofiten les propietats matemàtiques d'aquests codis per tal de poder-los construir de manera eficient. Aquest objectiu s'ha assolit totalment, doncs s'han implementat funcions de construcció de codis Vasil'ev i *doubling*, a més de funcions per a calcular invariants de codis perfectes i altres funcions generals per a treballar amb codis no lineals.

3. Proves del paquet per a comprovar-ne el funcionament. Aquest objectiu s'ha acomplert satisfactòriament: s'han desenvolupat i executat funcions de test per a cada funció del paquet de Magma.

4. Documentació de tot el paquet de funcions, incloent exemples. Aquest objectiu s'ha acomplert totalment, s'ha desenvolupat un manual del paquet seguint el format del manual de Magma i s'ha documentat el codi del paquet.

5. Redacció de la memòria del projecte. Aquest objectiu s'ha assolit completament, ja que és un requisit indispensable per a poder entregar el projecte.

Pel que fa a la planificació temporal del projecte, quan es va realitzar a l'inici es va deixar un marge de temps bastant gran per a contemplar possibles imprevistos. Finalment, tot i que s'ha hagut de consumir una part d'aquest temps de marge, s'han assolit a temps tots els objectius del projecte. Per tant, podem dir que s'ha respectat la planificació, doncs la realització i assoliment de les fites s'ha fet tal i com es proposava, tot i que lleugerament desplaçada en el temps.

7.2 Conclusions

Aquest projecte té una característica molt important i bastant distintiva respecte altres projectes de final de carrera, i és la necessitat de tenir un

coneixement bastant ampli dels fonaments teòrics sobre els quals es basen totes les funcions que s'han desenvolupat posteriorment. Per això, el projecte ha estat complex, però el fet d'haver après tota aquesta teoria és molt satisfactori.

Per primera vegada en tota la carrera, m'he hagut d'enfrontar a un projecte tota sola, sense un equip de treball, si bé sempre sota la guia de la meva directora de projecte. He hagut d'analitzar la viabilitat del projecte, desenvolupar-lo i documentar-lo, i prendre moltes decisions al llarg de tot el procés. És, per tant, un petit tast del que espero serà la resta de la meva carrera professional com a enginyera.

Per a mi, una de les coses més importants d'haver fet aquest projecte és que el paquet *BinaryPerfectCodes* desenvolupat serveixi als investigadors del Grup de Combinatòria i Codificació del Departament d'Enginyeria de la Informació i de les Comunicacions com a suport en els seus projectes.

7.3 Línies futures

Per a continuar amb aquest projecte, hi ha diversos aspectes que es poden modificar per a millorar el rendiment i ampliar les funcionalitats que ofereix el paquet actualment.

- Afegir noves construccions de codis binaris perfectes no lineals, com poden ser la construcció de Mollard i la construcció *switching*.
- Afegir noves funcions per a emmagatzemar codis de longitud 15 no equivalents, és a dir, fer una base de dades i totes les funcions necessàries per a obtenir codis de la base de dades i afegir-ne de nous.
- Millorar la funció que calcula l'estructura *BinaryCode* a partir del *kernel* i els líders del codi. Tot i que aquesta funció no ha estat desenvolupada dins aquest projecte, la seva millora suposaria un gran avantatge per a les construccions de codis del paquet *BinaryPerfectCodes*.

- Desenvolupar funcions per a comprovar si dos codis perfectes són equivalents o no, utilitzant el grup d'automorfismes dels codis.
- Afegir noves funcions per a calcular altres invariants d'un codi, com ara els fragments.
- Unificar els paquets *BinaryCodes* i *BinaryPerfectCodes*, ja que en un principi havíen de formar part del mateix paquet, però per qüestions pràctiques es va decidir separar-los.

Bibliografia

[CE08] J. J. Cannon and W. Bosma (Eds.). *Handbook of Magma Functions*. 2.15 edition, 2008.

[con09] *XVI Convenio colectivo estatal de empresas de consultoría y estudios de mercados y de la opinión pública*. Number 82. Boletín Oficial del Estado, 4 d'abril de 2009.

[dg09] CCG development group. *CCG Style Guide*. Departament d'Enginyeria de la Informació i les Comunicacions, Març 2009.

[Gib76] Peter D. Gibbons. *Computing Techniques for the Construction and Analysis of Block Designs*. PhD thesis, University of Toronto, 1976.

[Hed08] Olof Heden. Perfect codes from the dual point of view i. *Discrete Mathematics*, 308(24):6141–6156, Desembre 2008.

[Her82] Ferdinand Hergert. The equivalence classes of the vasil'ev codes of length 15. In *Proceedings of Ravishholzhausen Konfoenz*, volume 963. Springer Lecture Notes Series, 1982.

[MÖPS08] I. Yu. Mogilnykh, Patric R. J. Östergård, Olli Pottonen, and F. I. Solov'eva. Reconstructing extended perfect binary codes from their minimum distance graphs. Octubre 2008.

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

[RH91] Josep Rifà and Llorenç Huguet. *Comunicación digital: teoría matemática de la información, codificación algebraica, criptología*. Masson, Barcelona, 1991.

[Vil01] Mercè Villanueva. *On rank and kernel of perfect codes*. PhD thesis, UAB, Juliol 2001.

[Vil09] Mercè Villanueva. *Design Theory, Geometry and Codes. Apunts de Teoria de la Codificació del Màster en Informàtica Avançada*. UAB, Gener 2009.

Apèndix A

Paquet *BinaryPerfectCodes*

A continuació s'adjunta un CD amb el codi font del paquet *BinaryPerfectCodes*, les proves, exemples i el manual.

Firmat: Laura Vidal Marín

Bellaterra, juny de 2009

Resum

La finalitat d'aquest projecte és aconseguir construir codis binaris perfectes no lineals de manera eficient. Per a fer-ho, hem desenvolupat un paquet de *software* per a l'intèrpret MAGMA que conté funcions per a la construcció de codis perfectes, càlcul d'invariants de codis i altres funcions complementàries per a fer càlculs sobre les paraules d'un codi.

Resumen

La finalidad de este proyecto es conseguir construir códigos binarios perfectos no lineales de forma eficiente. A tal efecto, hemos desarrollado un paquete de *software* para el intérprete MAGMA que contiene funciones para la construcción de códigos perfectos, cálculo de invariantes de códigos y otras funciones complementarias para realizar cálculos sobre las palabras de un código.

Abstract

The purpose of this project is to be able to construct nonlinear binary perfect codes in an efficient way. In order to do so, a software package for the interpreter MAGMA has been developed. This package contains functions to construct perfect codes, compute some invariants of these codes and other additional functions to perform operations on codewords.