INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

On Combinatorial Searches for Designs and Codes

Sheridan Houghten

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfilment of the Requirements for
the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

July 1999

© Sheridan Houghten, 1999

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre reference

Our file Notre reference

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-43587-3

Canada

Abstract

On Combinatorial Searches for Designs and Codes

Sheridan Houghten

Concordia University, 1999

Searches for designs and codes may be divided into two broad categories: those which attempt to prove the *existence* of a design or code with a given set of parameters, and those which attempt to *enumerate* designs or codes with a given set of parameters. In this thesis we consider computer searches for several designs and one code. We also consider computational techniques which one may generally employ in such searches.

A 2-(28, 12, 11) design is both quasi-derived and quasi-symmetric, with intersection numbers $a = 4$ and $b = 6$. We consider the problem of embedding such designs with an automorphism of order 7 without fixed points or blocks, as derived designs in a symmetric 2-(64, 28, 12) design. We find that four 2-(28, 12, 11) designs could not be embedded, thus providing the first examples of quasi-derived quasi-symmetric designs which are not derived designs.

The smallest case with the parameters $k = 6$ and $\lambda = 1$ for which it is not known if a $(v, 6, 1)$ design exists is the case $v = 46$. One may partition the search for a (46, 6, 1) design into two cases. A search assuming the first of these cases revealed no designs. We examine the remaining case, presenting detailed information on how one may structure a search, and present estimates of the time required to complete the search.

The largest minimum weight of a self-dual doubly-even binary (n, k, d) code is $d = 4\lfloor n/24 \rfloor + 4$. Of such codes with length divisible by 24, the Golay Code is the only $(24, 12, 8)$ code, the Extended Quadratic Residue Code is the only known $(48, 24, 12)$ code, and there is no known $(72, 36, 16)$ code. For any non-zero weight w of such a code, the codewords of weight w form a 5-design. One may partition the search for a $(48, 24, 12)$ self-dual doubly-even code into 3 cases. A search assuming one of the cases found only the Extended Quadratic Residue Code itself. We consider the remaining 2 cases, giving information on how a search may be structured for each of these cases, and present estimates of the search size and time.

Estimation is of great importance in these types of searches, as it not only indicates if a search is feasible with the given resources, but also may indicate necessary areas of improvement in our algorithms. We examine methods of obtaining estimates of the size of a search and of the time required to complete a search, including a demonstration of a conversion of the former type of estimate to the latter.

Acknowledgments

I would like to thank my supervisor, Dr. Clement Lam, for his invaluable guidance, support and patience throughout my graduate studies. I also thank the members of my thesis committee who have helped to guide me through the last few years.

For allowing me access to their computing facilities, I thank the staff of Computer Services, the Department of Computer Science and the Centre interuniversitaire en calcul mathématique algébrique at Concordia University. I would especially like to thank Mr. Larry Thiel who has provided many programming ideas and who has written many of the computer programs I have used in my research.

There are several people with whom I have collaborated, and who have provided assistance in understanding problems and writing papers. I thank Dr. Vladimir Tonchev, Dr. Jeannette Janssen, Ms. Yuan Ding, Ms. Suzan Smith and Mr. Jeff Parker.

I have been supported by many people during my studies. My family and friends have been especially supportive. I would also like to thank the professors at Concordia. A special thankyou goes to the Science College, especially Dr. Geza Szamosi and Mrs. Ruth Richer, for helping me to decide to take this path.

My final and biggest thanks are for my husband Warren and my daughter Katia, for occupying themselves together for so much time and allowing me to work.

Contents

List of Figures x

List of Tables xi

List of Functions xii

List of Symbols xiii

1 Introduction 1
- 1.1 Designs . . . 1
  - 1.1.1 Symmetric, Residual and Derived Designs . . . 2
  - 1.1.2 Designs with $\lambda = 1$ . . . 3
  - 1.1.3 Isomorphisms and Automorphisms of Designs . . . 3
- 1.2 Error-Correcting Codes . . . 4
  - 1.2.1 Equivalence and Automorphisms of Codes . . . 4
  - 1.2.2 Self-Dual and Extremal Codes . . . 5
- 1.3 Objectives . . . 6
- 1.4 Contributions . . . 7
- 1.5 Organization of Thesis . . . 8

2 Preliminaries 9
- 2.1 Designs and their Incidence Matrices . . . 9
- 2.2 Codes and their Weight Enumerators . . . 9
- 2.3 Quadratic Residue Codes . . . 10
- 2.4 Reed-Muller Codes . . . 11
- 2.5 Depth-First Searches and Backtracking . . . 13
- 2.6 Estimates . . . 14

3 Connections between Designs and Codes 16
- 3.1 Codes from Designs . . . 16
  - 3.1.1 The p-rank of a Design . . . 16
- 3.2 Designs from Codes . . . 17

3.2.1 Assmus-Mattson Theorem 17
3.2.2 Designs from Perfect Codes 18
3.2.3 Symmetric Designs from Codes 18

4 Extensions of 2-(28, 12, 11) Designs **19**
4.1 Enumeration of 2-(28, 12, 11) Designs 19
4.2 Extensions Assuming an Automorphism of Order 7 19
4.3 Non-Embeddable Quasi-Derived Designs 19
4.4 Counting Symmetric and Residual Designs 24
4.5 Summary 28
4.6 Further Work 28

5 Estimates in the Search for a (46, 6, 1) Block Design **30**
5.1 Search Procedure 30
5.2 Incidence Matrix with a "c4" 33
5.3 Incidence Matrix with a "c5" 33
5.3.1 A Column of 5 Ones 36
5.3.2 A Column of 4 Ones 39
5.3.3 Columns of 3 Ones 40
5.4 Summary 40
5.5 Update 42
5.6 Further Work 45

6 Construction of (48, 24, 12) Self-Dual Doubly-Even Codes **46**
6.1 Introduction 46
6.2 Organization of Generator Matrix for $b = 2$ 47
6.3 Organization of Generator Matrix for $b = 3$ 51
6.3.1 Note 56
6.4 Summary 56

7 Search Method and Estimates in the Search for a (48, 24, 12) Code **58**
7.1 Complete Search 59
7.1.1 Complete Search to Row 8 for $b = 3$ 60
7.1.2 Complete Search to Row 7 for $b = 2$ 61
7.2 Estimation of Search Size 63
7.2.1 Starting Information for $b = 3$ 63
7.2.2 Starting Information for $b = 2$ 64
7.2.3 Estimation Method 65
7.2.4 Number of Survivors Chosen for each Row 66
7.3 Refinements to the Search 67
7.3.1 Ordering of Rows 67

7.3.2 Choosing a Block Pattern . . . 67
7.3.3 Rejecting Previous Cases . . . 68
7.3.4 Selecting Particular Row Types . . . 69
7.3.5 Partial Isomorphism Testing . . . 73
7.3.6 Resulting Estimates for $b = 3$ . . . 75
7.3.7 Resulting Estimates for $b = 2$ . . . 76
7.4 Summary . . . 77
7.5 Further Work . . . 77

8 Computational Methods **79**
8.1 Vector Storage and Manipulation . . . 79
8.1.1 Storage and Addition of Codewords . . . 80
8.1.2 Computing the Weight of Codewords . . . 81
8.2 Ordering of Candidates . . . 82
8.3 Generation and Maintenance of Candidate Lists . . . 84
8.4 Testing for Survivors . . . 85
8.5 Compatibility Matrices . . . 86
8.6 Summary . . . 88

9 Estimate Conversion **89**
9.1 The Search for a (48, 24, 12) Self-Dual Doubly-Even Code . . . 90
9.1.1 Operations Performed at Every Level . . . 90
9.1.2 Operations Performed for Every Candidate at Every Level . . . 90
9.1.3 Operations Performed for Every Survivor at Every Level . . . 91
9.2 Adding Codewords to the Buffer . . . 92
9.3 Checking the Buffer . . . 93
9.4 Checking Column-Blocks . . . 95
9.5 Subcode Test . . . 97
9.6 Partial Isomorphism Testing . . . 99
9.6.1 Finding Vectors with the Correct Distribution of Ones . . . 99
9.6.2 Finding "Earliest" Vectors . . . 101
9.6.3 Detailed Poor Woman Algorithm . . . 105
9.7 Operations per Row for $b = 3$ . . . 109
9.8 Operations per Row for $b = 2$ . . . 110
9.9 Summary . . . 111

10 Conclusion **113**

Bibliography **115**

Appendix A: Incidence Matrices of 4 Non-Extendable Quasi-Derived Designs 120

Appendix B: Incidence Matrix of a Quasi-Symmetric 2-(36,16,12) with a Trivial Automorphism Group 121

List of Figures

2.1 Generator Matrix of QR 12
2.2 Generator Matrix for $R(1,2)$ 12
2.3 Generator Matrix for a (24, 3, 12) Self-Dual Doubly-Even Code 12
2.4 Generator Matrix for a (24, 2, 12) Self-Dual Doubly-Even Code 13

4.1 Structure of the Incidence Matrix for a Symmetric 2-(64, 28, 12) Design 20
4.2 Structure of A_{64}, with Column Intersection Information 23

5.1 Structure of Incidence Matrix for a (46, 6, 1) Design 32
5.2 Structure of Incidence Matrix Containing a "c5" 35
5.3 Incidence Matrix for Case 1 38

6.1 Organization of Generator Matrix for C_{48} 47
6.2 Structure of Generator Matrix for $b = 2$ 49
6.3 Structure of Generator Matrix for $b = 3$ 52

List of Tables

4.1 Number of Candidates Per Derived Design 22
4.2 Summary of Extensions and Corresponding Residual Designs 25
4.3 Automorphism Group Statistics, 2-(64, 28, 12) Designs 26
4.4 2-rank Statistics, 2-(64, 28, 12) Designs 27
4.5 Automorphism Group Statistics, Residual Designs 27

5.1 Estimates Assuming a Column of 5 Ones 39
5.2 Estimates Assuming a Column of 4 Ones 41
5.3 Estimates Assuming a Column of 3 Ones 42
5.4 Results Assuming a Column of 5 Ones 43
5.5 Results Assuming a Column of 4 Ones 44
5.6 Results Assuming a Column of 3 Ones 45

7.1 Number of Non-Isomorphic Codes by Row for $b = 3$ 61
7.2 Count of Codes at Row 8 for $b = 3$ 62
7.3 Number of Non-Isomorphic Codes by Row for $b = 2$ 62
7.4 Count of Codes at Row 8 with no (24, 4, 12) Subcode 70
7.5 Estimates of Candidates and Survivors by Row for $b = 3$ 76
7.6 Estimates of Candidates and Survivors by Row for $b = 2$ 77

9.1 Estimates of Candidates and Operations by Row for $b = 3$ 110
9.2 Estimates of Candidates and Operations by Row for $b = 2$ 111

List of Functions

Function	Page
AddToBuf(w, *Buf*, *SizeOfBuf*)	92
CheckBlocks(w, r)	95
CheckBlocksb2(w, r)	97
CheckBuf(*Buf*, *SizeOfBuf*)	93
CheckSubcode()	98
CompleteSearch(*StartRow*, *EndRow*)	60
EarlierThan(x, y)	102
EarlierThanb2(x, y)	103
Extend($\{a_1, \ldots, a_i, *, \ldots, *\}$)	13
Find10_2(r, *Buf*, *SizeOfBuf*)	100
FindWeight(v)	82
NextEarliest(*prev*)	104
PoorWoman(r, *cand*, *Buf*, *SizeOfBuf*)	105

List of Symbols

A_i	an incidence matrix for D_i
C	an (n, k, d) linear code
C^-	the dual of the code C
C_{48}	an extremal (48, 24, 12) self-dual doubly-even code
col_i	the i'th column of the generator matrix for C_{48}
D	a $t-(v, k, \lambda)$ design
D^-	the dual of the design D
D_{64}	a symmetric 2-(64,28,12) design
D_{28}	a quasi-symmetric 2-(28,12,11) design
D_{36}	a quasi-symmetric 2-(36,16,12) design
D_{46}	a 2-(46,6,1) design
QR	the Extended Quadratic Residue code of length 48
row_i	the i'th row of the generator matrix for C_{48}
$v(left)$	the codeword v restricted to its first 24 columns
$v(right)$	the codeword v restricted to its last 24 columns
$v \cdot w$	the inner product of codewords v and w
W_C	the weight enumerator of C
$wt(v)$	the weight of codeword v

Chapter 1

Introduction

In this thesis we examine combinatorial searches, in particular the construction of error-correcting codes and block designs. Some textbooks include [3, 9, 28, 40].

1.1 Designs

A t-design with parameters (v, b, r, k, λ) is a set of v points and b blocks such that every block contains exactly k points, every point is contained in exactly r blocks, and every set of t points is contained in exactly λ blocks. Equivalently, we say that every set of λ blocks *intersect in* exactly t points. Such a design is also referred to as a $t-(v, k, \lambda)$ design.

Let D be a design. Let P be the set of points in D and B be the set of blocks in D. The *dual* of D is denoted by D^-. In D^-, B is the set of points and P is the set of blocks. A point of D^- is incident with a block of D^- iff the corresponding block of D is incident with the corresponding point of D.

The *order* of a design, for $t \geq 2$, is $n = \lambda \begin{pmatrix} v-2 \\ k-1 \end{pmatrix} / \begin{pmatrix} v-t \\ k-t \end{pmatrix}$. In the case of 2-designs, the order is equivalent to $n = r - \lambda$.

A design is called *simple* if it has no repeated blocks. In this thesis, all designs we consider are simple.

A design is called *trivial* if every set of k points is contained in a block. A non-trivial 2-design is also known as a balanced incomplete block design, or BIBD.

Suppose that D is a $t-(v, k, \lambda)$ design. Let the points of D be labelled $p_1, \ldots, p_v$ and the blocks of D be labelled $x_1, \ldots, x_b$. An incidence matrix A of D is a $v \times b$

matrix such that:

$$A[i,j] = \begin{cases} 1 & \text{if } p_i \text{ is in } x_j, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

1.1.1 Symmetric, Residual and Derived Designs

A 2-design is called *symmetric* if $b = v$. In this case, we also have $r = k$, which implies that every pair of blocks intersect in exactly λ points. Symmetric designs are sometimes called *square* designs.

Given a symmetric 2-(v, k, λ) design D, removing any block x of D, together with all its points from the remaining blocks, gives a 2-$(v-k, k-\lambda, \lambda)$ design which is called the *residual* design of D with respect to block x. The points of x and the intersections of x with the remaining blocks form a 2-$(k, \lambda, \lambda-1)$ designs which is called the *derived* design of D with respect to block x. A 2-(v, k, λ) design with $r = k + \lambda$ is thus potentially a residual design and is called *quasi-residual*; similarly, a 2-(v, k, λ) with $\lambda = k - 1$ is potentially a derived design and is called *quasi-derived*.

Note that by replacing a symmetric design with its complementary design, its derived designs are transformed into residual designs, and vice-versa. Hence the problem of embedding a quasi-derived design is essentially the same as that of embedding a quasi-residual design. A trivial reason for which a quasi-residual (or quasi-derived) design might not be embeddable into, or extendable to, its corresponding symmetric design, is the non-existence of the symmetric design. It is also possible that in a quasi-residual 2-(v, k, λ) design, some pair of blocks intersect in more than λ points, which is not possible in a residual design. For more information on other non-embeddable designs, see [47].

Although not every quasi-residual design is residual, all those with k "much" larger than λ are. More precisely, as proved by Bose, Shrikhande and Singhi [7] (see also [9]), any quasi-residual design with $\lambda > 2$ and $k > g(\lambda)$ is residual, where:

$$g(\lambda) = \begin{cases} 76 & \text{if } \lambda = 3, \\ \frac{1}{2}(\lambda-1)(\lambda^4 - 2\lambda^2 + \lambda + 2) & \text{if } 4 \le \lambda \le 9, \text{ and} \\ \frac{(\lambda-1)^2(\lambda-2)}{2}(1 + (\lambda-1)(\lambda-2)) - \lambda + 1 & \\ + \frac{(\lambda-1)(\lambda-2)}{2}\sqrt{(\lambda-1)^2(1+(\lambda-1)(\lambda-2))^2 + 4(\lambda-1)} & \text{if } \lambda > 9. \end{cases}$$

A 2-design is *quasi-symmetric* with intersection numbers a, b, $(0 \leq a < b)$ if any two blocks intersect in either a or b points [43]. The question of whether a quasi-symmetric design is also a quasi-derived (or a quasi-residual) design is of special interest, for any embedding of such a design in a symmetric design gives automatically another quasi-symmetric design as the corresponding residual (respectively, derived) design. Thus, embedding of quasi-symmetric quasi-derived (or quasi-residual) designs can be viewed as a technique for the construction of new quasi-symmetric designs from given ones. For some applications of this technique see [23], [26], [45].

1.1.2 Designs with $\lambda = 1$

Designs with $t \geq 2$ and $\lambda = 1$ are also known as Steiner systems. A Steiner system denoted as $S(t, k, v)$, is a $t - (v, k, 1)$ design.

The existence or non-existence of all designs with $\lambda = 1$ and $k < 6$ is known. In particular, existence results for $k \leq 5$, and for $k = 6$ with $\lambda \neq 1$ are given in [18]. In [2], [14], [17], [18], [33], [34], [35], [36] and [37], the existence of certain designs with $k = 6$ and $\lambda = 1$ is proven. A summary of these results can be found in [1]. As seen in this summary, the smallest remaining case for which the existence question has yet to be answered is the case $(46, 6, 1)$.

In [37], the author also shows that, due to recursive construction, there are only a finite number of designs with $k = 6$ and $\lambda = 1$ which need to be considered before all existence results for designs with these parameters can be proven.

1.1.3 Isomorphisms and Automorphisms of Designs

Let D and D' be $t - (v, k, \lambda)$ designs with sets of points P and P' respectively, and sets of blocks B and B' respectively. An *isomorphism* from D to D' is a one-to-one map f from P to P', carrying B to B', such that $p \in P$ is contained in $x \in B$ if and only if $f(p) \in P'$ is contained in $f(x) \in B'$. D is isomorphic to D' is there exists such an isomorphism.

Two designs must have the same parameters if they are to be isomorphic. Let A and A' be the incidence matrices of D and D' respectively. In terms of these incidence

matrices, an isomorphism from D to D' may be viewed as a permutation of the rows and columns of A to produce A'.

An *automorphism* of a design is an isomorphism from a design to itself. The set of automorphisms of a design forms a group, called the automorphism group of the design.

1.2 Error-Correcting Codes

Let F be $GF(q)$, the finite field with q elements. A code C is a subset of F^n, the set of vectors of length n with components from F. The elements of C are called *codewords*. A code is *linear* if, for any two codewords u and v, $u + v$ is also a codeword. If this is the case, then the code forms a linear subspace. In this thesis, we are interested only in linear codes.

The *weight* $wt(w)$ of a codeword w is the number of its non-zero components. A code is *even* if the weight of any vector in the code is divisible by 2, and *doubly-even* if the weight of any vector is divisible by 4.

The *minimum weight* of a code is the smallest weight of the non-zero vectors in the code. The *distance* between two codewords u and v is the number of components in which they differ. For linear codes, *minimum weight* and *minimum distance* are the same.

An (n, k, d) code is a k-dimensional subspace of F^n with minimum distance d. Up to $\lfloor (d-1)/2 \rfloor$ errors can be corrected by using *maximum likelihood decoding*, in which we choose the closest codeword to the one we receive. High minimum distance is therefore an extremely desirable property in a code.

Let C be an (n, k, d) code over F. The code C is *perfect* if all vectors in F^n are at distance at most $\lfloor (d-1)/2 \rfloor$ from some codeword of C.

1.2.1 Equivalence and Automorphisms of Codes

Two linear codes over F are *equivalent* if one can be obtained from the other by a coordinate permutation of their codewords, followed by a multiplication of each coordinate position by (possibly different) elements of F.

Suppose that such an action upon a linear code C preserves C. Then we consider this action to be an automorphism of C. The set of automorphisms of C forms the automorphism group of C. If σ is an automorphism of C, then it is of *order p* if σ^p is the identity permutation.

1.2.2 Self-Dual and Extremal Codes

Let C be an (n, k, d) code over $GF(2)$. If u and v are two vectors in C, then their *inner product* is defined as

$$u \cdot v = \sum_{i=1}^{n} u_i v_i \pmod 2.$$

If $u \cdot v = 0$ then u and v are *orthogonal*. The *dual* of C is defined as

$$C^{-} = \{u \in GF(2)^n | u \cdot v = 0 \ \forall v \in C\}.$$

A code C is *self-dual* if $C = C^{-}$.

The largest minimum weight of a self-dual, doubly-even $(n, n/2, d)$ code over $GF(2)$ is $d = 4\lfloor n/24 \rfloor + 4$ (see [29]). For $n \geq 74$, there is a further upper bound on minimum weight, namely $d \leq 2\lfloor (n+6)/10 \rfloor$ (see [13]). A self-dual code that has the largest possible minimum weight is an *extremal code*.

Binary self-dual codes of length up to 32 are classified in [11] and [12]. All of these have a non-trivial automorphism group. In [38], it is shown that most binary self-dual codes have only a trivial automorphism group: as their length approaches infinity, the ratio of the number of self-dual codes with non-trivial automorphism group, to the total number of self-dual codes, approaches zero. Thus self-dual codes with a non-trivial automorphism group become relatively rare as the length of these codes increases.

Extremal self-dual, doubly-even codes with length divisible by 24 are of particular interest because for any non-zero weight w, the codewords of weight w form a 5-design [4]. Of such codes, the Golay code is the only $(24, 12, 8)$ code and the Extended Quadratic Residue Code, which we shall call QR, is the only known $(48, 24, 12)$ code.

However, some further properties of these codes are known — in particular, with respect to their automorphism groups. An algorithm for computing the automorphism group of error-correcting codes is given in [27].

In [21], it is shown that any extremal self-dual doubly-even code C of length 48 with a nontrivial automorphism of odd order is equivalent to the Extended Quadratic Residue Code. This is done by looking at the possible values of p, if σ is an automorphism of C of prime order $p > 2$. There are six cases, namely $p = 47, 23, 11, 7, 5$ and 3. Each case is considered in turn to prove the result.

There is no known self-dual doubly-even $(72, 36, 16)$ code [44], but [10] finds all the odd primes p which can divide the order of the group of such a code if it exists. These are $p = 23, 17, 11, 7, 5$ and 3. In [39], [41], and [22], the cases $p = 23, 17$ and 11 are eliminated respectively.

1.3 Objectives

We consider several problems in this thesis.

The smallest non-trivial parameters for a design that is both quasi-derived and quasi-symmetric, are 2-$(22, 6, 5)$, with intersection numbers $a = 0$ and $b = 2$. These parameters correspond to the unique Mathieu-Witt 3-$(22, 6, 1)$ design, which is embeddable in a symmetric 2-$(78, 22, 6)$ design. The residual design thus produced is a 2-$(56, 16, 6)$ design which is quasi-symmetric with intersection numbers $a = 4$ and $b = 6$ [45].

In this thesis we consider the next smallest parameter set, namely 2-$(28, 12, 11)$, a quasi-derived design which is quasi-symmetric with intersection numbers $a = 4$ and $b = 6$. We wish to determine if it is possible to embed all such designs with an automorphism of order 7 without fixed points or blocks, as derived designs in a symmetric 2-$(64, 28, 12)$ design.

The smallest case with the parameters $k = 6$ and $\lambda = 1$ for which it is not known if a design exists is the case $v = 46$. The search for such a design may be divided into two large cases. A search assuming the first of these two cases has previously been completed by Janssen, Lam & Thiel, revealing no designs. In this thesis we examine

the remaining case. Our objective is to determine if a $(46, 6, 1)$ design exists.

We also examine $(48, 24, 12)$ self-dual doubly-even codes, of which the Extended Quadratic Residue Code is the only known example. The search for such a code may be partitioned into 3 cases, namely $b = 2$, 3 or 4, where b is the maximum dimension of a subcode of length 24. A search assuming the case $b = 4$ has previously been completed, finding only the Extended Quadratic Residue Code itself [19]. In [25], the author calculates by hand that no codes are possible if $b = 2$, although this has yet to be verified and thus one of our objectives is to verify this by independent means. In this thesis we consider the cases $b = 3$ and $b = 2$. Our objective is to determine if the Extended Quadratic Residue Code is the only $(48, 24, 12)$ self-dual doubly-even code.

1.4 Contributions

We consider 2-$(28, 12, 11)$ designs with an automorphism of order 7 without fixed points or blocks. We attempt to embed all such designs as derived designs in a symmetric 2-$(64, 28, 12)$ design. We find that four 2-$(28, 12, 11)$ designs could not be embedded, thus providing the first examples of quasi-symmetric quasi-derived designs which are not derived designs. We also provide the first examples of quasi-symmetric 2-$(36, 16, 12)$ designs with trivial automorphism groups, and improve on the numbers of 2-$(64, 28, 12)$ designs and 2-$(36, 16, 12)$ designs as found in the Handbook of Combinatorial Designs [31].

We consider the search for a $(46, 6, 1)$ design. We give detailed information on how one may structure a search assuming the remaining case, and present estimates of the time required to complete the search.

We also consider the enumeration of $(48, 24, 12)$ self-dual doubly-even codes. Our method is to attempt direct construction of the code by attempting to complete a generator matrix. We give information on the structure of the generator matrix for each of the two remaining cases. We also give information on how one may structure a search assuming each of the two remaining cases, and give estimates of the size of these searches.

Finally, we describe several computational methods applicable to these types of searches, and describe how estimates of search size may be translated to estimates of the time required to complete the search, using the search for a $(48, 24, 12)$ self-dual doubly-even code as an example.

1.5 Organization of Thesis

There are 10 chapters in this thesis. The layout of the remainder of the thesis is as follows: Chapter 2 gives some technical definitions and mathematical preliminaries which relate to the research problems. Chapter 3 examines some of the connections which exist between error-correcting codes and block designs. Chapter 4 examines the problem of extending 2-$(28, 12, 11)$ designs to 2-$(64, 28, 12)$ designs. Chapter 5 gives information on how to structure a search for a 2-$(46, 6, 1)$ design, and some estimates of the computing time required to solve this problem. Chapters 6 and 7 examine the problem of constructing $(48, 24, 12)$ self-dual doubly-even codes: Chapter 6 describes the structure of the generator matrix and its rows, while Chapter 7 presents a possible search method and estimates of search size. Chapter 8 describes some computational methods which can be used in solving research problems of the type seen in this thesis, and related problems. We use some research problems in this thesis as examples. Chapter 9 considers the conversion of an estimate of the size of a search to an estimate of the time required to complete the search, using the search for $(48, 24, 12)$ self-dual doubly-even codes as an example. Chapter 10 concludes the thesis with a summary of the results and contributions made in the thesis. A summary of the results of each research problem, together with future related work, may be found at the end of the chapter relating to that problem.

Chapter 2

Preliminaries

This chapter gives some technical definitions and additional information that we require to solve the problems identified in Chapter 1.

2.1 Designs and their Incidence Matrices

Let D be a t-(v, b, r, k, λ) design. The parameters of D are related by the following two equations:

$$b = \lambda \binom{v}{t} / \binom{k}{t} \tag{2.1}$$

and:

$$bk = vr. \tag{2.2}$$

Suppose that A is an incidence matrix for D. Then A^{-} is an incidence matrix for D^{-}, the dual of D.

In 2-designs, the following equation relates A with its transpose:

$$AA^{-} = (k - \lambda)I + \lambda J \tag{2.3}$$

2.2 Codes and their Weight Enumerators

Let C be an (n, k, d) code.

The *weight enumerator* of C is

$$W_C(x, y) = a_0 x^n + a_1 x^{n-1} y + a_2 x^{n-2} y^2 + \cdots + a_n y^n. \tag{2.4}$$

where a_i is the number of vectors in C of weight i. The MacWilliams Identity relates the weight enumerator of a code to that of its dual: [28, page 127]

$$W_{C^\perp}(x, y) = \frac{1}{|C|} W_C(x + y, x - y). \tag{2.5}$$

Let C_{48} be a $(48, 24, 12)$ self-dual doubly-even code. Because C_{48} is doubly-even and has minimum weight 12, the only possible non-zero coefficients in the weight enumerator of C_{48} are a_0, a_{12}, a_{16}, a_{20}, a_{24}, a_{28}, a_{32}, a_{36} and a_{48}.

Define z as the length-48 vector whose components are all 1. Because z is in $C_{48}^\perp$ and C_{48} is self-dual, z is in C_{48}. For each vector v of weight i in C_{48} there is a corresponding vector $v + z$ of weight $48 - i$. Thus the coefficients of the weight enumerator are related as follows: $a_0 = a_{48} = 1$, $a_{12} = a_{36}$, $a_{16} = a_{32}$, $a_{20} = a_{28}$.

From Eq. (2.4) and (2.5) we have

$$W_{C_{48}}(x, y) = x^0 y^{48} + a_{12} x^{12} y^{36} + a_{16} x^{16} y^{32} + \cdots - a_{16} x^{32} y^{16} + a_{12} x^{36} y^{12} + x^{48} y^0$$

and

$$W_{C_{48}}(x, y) = W_{C_{48}^\perp}(x, y) = \frac{1}{2^{24}} W_{C_{48}}(x + y, x - y).$$

By expanding $\frac{1}{2^{24}} W_{C_{48}}(x + y, x - y)$ and equating the coefficients of $x^2 y^{46}$, $x^4 y^{44}$, $x^6 y^{42}$ and $x^{10} y^{38}$ to zero we get a system of equations which, when solved, give:

$$\begin{aligned}
a_0 &= a_{48} = 1, \\
a_{12} &= a_{36} = 17\,296, \\
a_{16} &= a_{32} = 535\,095, \\
a_{20} &= a_{28} = 3\,995\,376, \\
a_{24} &= 7\,681\,680.
\end{aligned}$$

2.3 Quadratic Residue Codes

The only known $(48, 24, 12)$ self-dual doubly-even code is the Extended Quadratic Residue code of length 48, which we call QR. Here we give a brief description of this code. For further information on quadratic residue codes, see [6].

An element $a \in GF(p)$ is a *quadratic residue* modulo p if $x^2 = a$ has non-zero solutions in GF(p) [6. page 171]. For $p = 47$ the set of quadratic residues is

$$S = \{1, 2, 3, 4, 6, 7, 8, 9, 12, 14, 16, 17, 18, 21, 24, 25, 27, 28, 32, 34, 36, 37, 42\}.$$

We define the Hall polynomial as $\theta(x) = \sum_{s \in S} x^s$. The generator polynomial for the (47, 24, 11) quadratic residue code can be obtained by

$$g(x) = \gcd(1 + x + \cdots + x^{46}, \theta(x)).$$

We find that

$$g(x) = 1 + x + x^2 + x^3 + x^5 + x^6 + x^7 + x^9 + x^{10} + x^{12} + x^{13} + x^{14} + x^{18} + x^{19} + x^{23}.$$

From $g(x)$ we construct a generator matrix for the (47, 24, 11) Quadratic Residue code. Label the columns of the generator matrix from left to right as 0 to 46. Row 1 of the matrix has a 1 in column i if x^i is in $g(x)$, and a 0 otherwise. Any other row j, $2 \leq j \leq 24$, is obtained by rotating row $j - 1$ to the right by one position.

To obtain the Extended Quadratic Residue code QR of length $n + 1 = 48$, we annex an overall parity check column to the matrix. We label this column as ∞. Fig. 2.1 gives the resulting generator matrix.

2.4 Reed-Muller Codes

A generator matrix for a 1st-order Reed-Muller Code $R(1, m)$ may be written so that the first m rows and last $2^m - 1$ columns are the binary representations of the numbers between 1 and $2^m - 1$, and the first column consists entirely of zeroes, except in the last row which contains only ones (see [3, p.143]). For example, a generator matrix for the first-order Reed-Muller code $R(1, 2)$ is shown in Fig. 2.2.

If we take 6 copies of the above matrix, and replace the third row by the sum of the second and third row, followed by some row and column permutations, we obtain the generator matrix shown in Fig. 2.3. This is a generator matrix for a (24, 3, 12) self-dual doubly-even code.

∞01	234	567	1 890	111 123	111 456	111 789	222 012	222 345	222 678	233 901	333 234	333 567	334 890	444 123	444 456
111	110	111	011	011	100	011	000	100	000	000	000	000	000	000	000
101	111	011	101	101	110	001	100	010	000	000	000	000	000	000	000
100	111	101	110	110	111	000	110	001	000	000	000	000	000	000	000
100	011	110	111	011	011	100	011	000	100	000	000	000	000	000	000
100	001	111	011	101	101	110	001	100	010	000	000	000	000	000	000
100	000	111	101	110	110	111	000	110	001	000	000	000	000	000	000
100	000	011	110	111	011	011	100	011	000	100	000	000	000	000	000
100	000	001	111	011	101	101	110	001	100	010	000	000	000	000	000
100	000	000	111	101	110	110	111	000	110	001	000	000	000	000	000
100	000	000	011	110	111	011	011	100	011	000	100	000	000	000	000
100	000	000	001	111	011	101	101	110	001	100	010	000	000	000	000
100	000	000	000	111	101	110	110	111	000	110	001	000	000	000	000
100	000	000	000	011	110	111	011	011	100	011	000	100	000	000	000
100	000	000	000	001	111	011	101	101	110	001	100	010	000	000	000
100	000	000	000	000	111	101	110	110	111	000	110	001	000	000	000
100	000	000	000	000	011	110	111	011	011	100	011	000	100	000	000
100	000	000	000	000	001	111	011	101	101	110	001	100	010	000	000
100	000	000	000	000	000	111	101	110	110	111	000	110	001	000	000
100	000	000	000	000	000	011	110	111	011	011	100	011	000	100	000
100	000	000	000	000	000	001	111	011	101	101	110	001	100	010	000
100	000	000	000	000	000	000	111	101	110	110	111	000	110	001	000
100	000	000	000	000	000	000	011	110	111	011	011	100	011	000	100
100	000	000	000	000	000	000	001	111	011	101	101	110	001	100	010
100	000	000	000	000	000	000	000	111	101	110	110	111	000	110	001

Figure 2.1: Generator Matrix of QR

$$\begin{matrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{matrix}$$

Figure 2.2: Generator Matrix for $R(1,2)$

$$\begin{matrix} 111111 & 111111 & 000000 & 000000 \\ 000000 & 000000 & 111111 & 111111 \\ 111111 & 000000 & 111111 & 000000 \end{matrix}$$

Figure 2.3: Generator Matrix for a $(24,3,12)$ Self-Dual Doubly-Even Code

```
111111111111  000000000000
000000000000  111111111111
```

Figure 2.4: Generator Matrix for a (24, 2, 12) Self-Dual Doubly-Even Code

From the corresponding generator matrix for the first-order Reed-Muller code $R(1, 1)$, we may similary obtain the generator matrix for a (24, 2, 12) self-dual doubly-even code which is shown in Fig. 2.4.

2.5 Depth-First Searches and Backtracking

In combinatorial searches, one method which is often used is the depth-first search with backtracking.

In a combinatorial search of the types discussed in this thesis, one is generally searching for any or all N-ruples satisfying a set of given conditions. For example, in the search for a (48, 24, 12) self-dual doubly-even code, the "N-tuples" are a set of 24 vectors of length 48. A condition to be satisfied is that, excluding the vector consisting entirely of zeroes, any linear combination of the vectors in this set must have a weight that is both greater than 12, and divisible by 4.

The method used in such searches often involves a depth-first search. Suppose that a complete solution for a problem is the N-tuple $\{a_1, \ldots, a_N\}$. In a depth-first search, one may commence the *level* i of the search by taking a particular partial solution $\{a_1, \ldots, a_i, *, \ldots, *\}$, in which the first i elements of the N-tuple are known. We then proceed to find all possible values for a_{i+1} satisfying the given conditions. For each possible value of a_{i+1} in turn, we extend the partial solution to $\{a_1, \ldots, a_i, a_{i+1}, *, \ldots, *\}$ and commence the $i+1$'th level of the search. After using each possible value for a_{i+1}, we "backtrack" to level $i-1$, deleting a_i from the partial solution.

The general algorithm employed for level i of a search is as follows:

Extend($\{a_1, \ldots, a_i, *, \ldots, *\}$)
begin

```
    if i = N
    then output the complete solution {a_1, ..., a_N};
    else
    begin
        find all possible elements a_{i+1} for which {a_1, ..., a_i, a_{i+1}, *, ..., *}
            satisfies the given conditions;
        for each such element a_{i+1} do
            Extend({a_1, ..., a_i, a_{i+1}, *, ..., *});
    end;
end;
```

2.6 Estimates

In searches such as those seen in this thesis, it is usually desirable to first make estimates, to ensure that completion of the search is feasible. We may wish to estimate any or all of the following:

- the amount of computer time required,
- the amount of computing memory required,
- the size of the search (number of candidates and survivors) at each level, as each partial solution is found.

These items are of course interrelated. The first two are dependent upon the specific computer used, although they may be translated to a different computer with knowledge of the differences in architecture of the computers in question. The last item, sometimes called the *profile* of the search, is independent of the resources to be used, and is reliant purely upon the algorithm to be used.

Indeed when the profile of the search is known, the amount of time and memory required may be derived by considering the algorithm used. For example, to obtain an estimate of the computer time required, we may look at the most frequently used

operations, multiplying the time required for each by the number of times it must be performed (generally equal either to the number of candidates or to the number of survivors). A rigorous accounting of machine cycles is probably unnecessary. An example of such a conversion is given in Chapter 9.

The advantages of making estimates are fairly obvious. With this knowledge we may make decisions as to the feasibility of completing the search in question with the chosen algorithm and available resources. If we decide that the search is computationally too expensive, then we may use the knowledge to modify the algorithm or find different resources.

In this thesis, examples of making estimates are given in two places. In Chapter 5, we show estimates of the time required to complete a search for a $(46, 6, 1)$ block design. In this example, we chose a small number of cases, and used the desired algorithm on each. Indeed, as different programs were used at different stages of the search, we chose cases at each separate stage.

In Chapter 7, we used estimates of time and memory requirements to decide for how long we could continue to perform a complete search for a $(48, 24, 12)$ self-dual doubly-even code, including checking for all isomorphisms. After a certain point, this complete search was no longer feasible. From then on, we made estimates of the size of the search as each new row was completed. We used these estimates to help us decide upon a course of action for performing the remainder of the search. Finally, in Chapter 9, we converted these estimates of the size of a search to estimates of the time required to complete the search.

Other examples of the use of estimates in computer searches of this type may be found in [8, 19, 24].

Chapter 3

Connections between Designs and Codes

In this chapter we shall examine some of the many connections which exist between designs and codes. For more information on this subject, consult [3, 9, 40].

3.1 Codes from Designs

First we shall examine how a code may be obtained from a design. Let D be a $t-(v,b,r,k',\lambda)$ design. The code of the design D, which we shall denote C_D, is the subspace spanned by the points of D.

The code C_D may thus be considered to be the row space of A, an incidence matrix of D. The incidence matrices of isomorphic designs yield isomorphic codes.

Let D^- be the dual design of D. Then A^T is an incidence matrix for D^-. Therefore C_{D^-}, the code of D^-, is the subspace spanned by the blocks of D, and may be considered to be the column space of A.

3.1.1 The p-rank of a Design

Let D be a 2-(v,b,r,k',λ) design of order n'. The p-rank of D, which we shall denote $rank_p(D)$, is the dimension of C_D over $GF(p)$.

Let p be a prime which does not divide n'. Then $rank_p(D) \geq v-1$ and $rank_p(D) = v-1$ if and only if p divides k' (see [3], p.42).

Let q be a prime which divides n'. Then $rank_q(D) \leq (b+1)/2$. Furthermore if q

does not divide λ and q^2 does not divide n', then $(C_q(D))^{\perp} \subseteq C_q(D)$ and $rank_q(D) \geq v/2$ (see [3], p.47).

For an example of the application of the above, consider D_{64}, which is a $2-(64, 64, 28, 28, 12)$ design. The order of this design is $n' = 28 - 12 = 16$. Consider the 2-rank of D_{64}. Since 2 divides 16, $rank_2(D_{64}) \leq 32$. This in turn implies that the dimension of $C_{D_{64}}$ is ≤ 32.

3.2 Designs from Codes

There are several means by which designs may be obtained from codes. Let C be an (n, k, d) code. In many cases, the codewords of C of any fixed weight form the blocks of a t-design on n points.

3.2.1 Assmus-Mattson Theorem

The following is a well-known result (see [4]). Let C be an (n, k, d) code over $GF(q)$ with dual $C^{\perp}$ which is an $(n, n-k, d^{\perp})$ code also over $GF(q)$.

Let

$$w = \begin{cases} n & \text{when } q = 2 \\ \text{largest integer } w \text{ satisfying } w - ((w+q-2)/(q-1)) < d & \text{otherwise} \end{cases}$$

Let $w^{\perp}$ be defined similarly. Let $W_{C^{\perp}}(x, y) = a_0 x^n + a_1 x^{n-1} y + \ldots a_n y^n$ be the weight enumerator of $C^{\perp}$. Suppose there exists an integer t with $0 < t < d$ for which at most $d - t$ coefficients a_s, $1 \leq s \leq n - t$, in $W_{C^{\perp}}$ are non-zero. Then for each i, $d \leq i \leq w$, the vectors of weight i in C yield a t-design. Similarly, for each j, $d^{\perp} \leq j \leq min(w^{\perp}, n - t)$, the vectors of weight j in $C^{\perp}$ yield a t-design.

For an application of this result, consider C_{48}, a (48, 24, 12) self-dual doubly-even code. Since C_{48} is self-dual, $C_{48} = C_{48}^{\perp}$ and hence $W_{C_{48}} = W_{C_{48}^{\perp}}$. In this case we have $q = 2$ and therefore we require a t, $0 < t < 12$, for which at most $12 - t$ coefficients a_s, $1 \leq s \leq n - t$, of the weight enumerator are non-zero. As seen in Chapter 2, the only non-zero coefficients of $W_{C_{48}}$ are $a_0, a_{12}, a_{16}, a_{20}, a_{24}, a_{28}, a_{32}, a_{36}$ and a_{48}. There are 7 non-zero coefficients in the required range and hence $1 \leq t \leq 12 - 7$. Therefore the vectors of any fixed weight i in C_{48} yield a t-design for $1 \leq t \leq 5$.

3.2.2 Designs from Perfect Codes

Let C be a perfect (n, k, d) code over $GF(q)$. Then the vectors of weight d yield a $t-(n, d, 1)$ design for $t = (d-1)/2 + 1$ (see [40], p.126).

Consider the $(7, 4, 3)$ Hamming code. This binary code has 2^4 codewords. The number of vectors in $GF(2)^7$ which are at distance ≤ 1 from any codeword is 8 — the codeword itself (at distance 0), and 7 others, all at distance 1, obtained by taking the complement of the entry in exactly one position at a time. Note that none of these 7 vectors can be a codeword, since then the minimum-distance criterion would not be satisfied.

Therefore we have a total of $8 \cdot 2^4 = 2^7$ vectors at distance at most 1 from any codeword and hence this is a perfect code. In applying the above theorem, we have $n = 7$, $k = 4$, $d = 3$ and $t = (3-1)/2 + 1 = 2$. Therefore the vectors of weight 3 in this code yield a $2-(7, 3, 1)$ design.

3.2.3 Symmetric Designs from Codes

Let D be a symmetric 2-(v, k, λ) design with incidence matrix A.

Let p be a prime such that $p \mid \lambda$ but $p \nmid (k-\lambda)$. The inner product of any pair of rows of A is $\lambda \equiv 0 \bmod p$. Also, it follows from Eq. (2.3) that $AA^T = (k-\lambda)I \bmod p$ which is not the zero matrix, and thus the row rank of A mod p is v.

Let D have a 2-(v_d, k_d, λ_d) derived design D_d and a 2-(v_r, k_r, λ_r) residual design D_r. Let A_d denote the v_d rows of A corresponding to D_d, and let A_r denote the remaining v_r rows of A. Then any row of A must be in the dual space (or dual code) of A_d over $GF(p)$, for a suitable p [46]. Similarly, any row of A must be in the dual space (or dual code) of A_r over $GF(p)$, for a suitable p. This information is useful in extending derived or residual designs to symmetric designs, as seen in Chapter 4.

Chapter 4

Extensions of 2-$(28, 12, 11)$ Designs

In this chapter, we consider the problem of embedding 2-(28, 12, 11) designs as derived designs in symmetric 2-(64, 28, 12) designs. The full details and results of this joint work may be found in [16]. Personal contributions to this work are described in sections 4.3–4.6 of this chapter.

4.1 Enumeration of 2-$(28, 12, 11)$ Designs

In [15], Ding enumerated all 2-(28, 12, 11) designs with an automorphism of order 7, and with no fixed points or blocks. Such designs can arise from only a few possible orbit matrices. For each orbit matrix, there may be several solutions. Up to isomorphism, there are 246 such designs.

4.2 Extensions Assuming an Automorphism of Order 7

Also in [16], Smith attempted to embed each of the above 246 designs as derived designs in symmetric 2-(64, 28, 12) designs invariant under the initial automorphism of order 7. Of the 246 2-(28, 12, 11) designs, 223 extend and 23 do not.

4.3 Non-Embeddable Quasi-Derived Designs

We now wish to determine if these 23 remaining designs can be extended if we do not make any assumption about an automorphism of order 7.

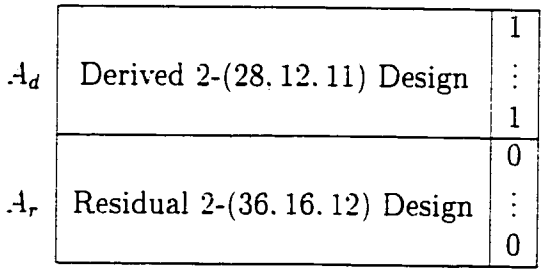

Figure 4.1: Structure of the Incidence Matrix for a Symmetric 2-(64, 28, 12) Design

Let A_{64} be the incidence matrix of a symmetric 2-(64, 28, 12) design. One may permute the rows and the columns of A_{64} so that in the last column, the 28 ones are placed in the first 28 rows. Then, as shown in Fig. 4.1, we have the incidence matrix of a derived 2-(28, 12, 11) design in the first 28 rows and first 63 columns of A_{64}, and the incidence matrix of a residual 2-(36, 16, 12) design in the last 36 rows and first 63 columns of A_{64}.

We wish to determine if each of the 23 remaining 2-(28, 12, 11) designs extend to 2-(64, 28, 12) designs. Therefore we shall place each of these remaining 23 designs, in turn, in the first 28 rows and first 63 columns of A_{64}, and attempt to complete the remainder of A_{64}.

Let A_d denote the first 28 rows of A_{64} and let A_r denote the last 36 rows of A_{64}. Therefore for each of the remaining 23 designs, we must complete A_r. We may attempt to complete A_r row-by-row. Only one column of each row of A_r is known. Each row must contain 28 ones, and hence for each row, there are $\binom{63}{28}$ possibilities to consider. Clearly using this brute-force approach will be very expensive. Instead, we shall use the information given in section 3.2.3.

Let $p = 3$. Note that $p \mid 12$ but $p \nmid (28 - 12)$. Then as seen in section 3.2.3, any row of A_{64} must be in the dual code of A_d over $GF(3)$.

Therefore to find the list of all vectors which are candidates to fill the rest of A_{64}, we take all vectors from the dual code of A_d over $GF(3)$ which have:

- (0, 1) entries only,
- exactly $k = 28$ ones, and

- an inner product of exactly $\lambda = 12$ with each row of A_d.

To obtain this list, we need the generator matrix for the dual code of A_d over $GF(3)$. We shall call this generator matrix $A_d^{\perp}$. Using $A_d^{\perp}$ we generate all possible vectors and retain all those which satisfy the above conditions.

Note that the dimension of $A_d^{\perp}$ is 64-28=36, and hence there are 3^{36} possible vectors. However, every vector in $A_d^{\perp}$ contains 1 or more ones. Multiplying any such vector by 2, we would obtain a vector containing 1 or more twos. Therefore we do not need to consider linear combinations involving vectors which have been multiplied by 2. Hence we now need to consider only 2^{36} possibilities, a reasonable number.

The number of candidates to complete the remainder of A_{64} for each of the 23 starting derived designs, is given in Table 4.1. In this table, "Case" refers to the original orbit matrix and "Solution" to the number of the solution within each case.

With this list of vectors, we attempt to complete the design. The general method is that described in Section 2.5. We use column intersection information to speed up the process. Let a, b, c and d refer to the number of rows in A_r which contain '00', '01', '10' and '11' respectively in columns 1 and 2. We may permute the rows of A_r so that A_{64} is of the form shown in Fig. 4.2. This eliminates isomorphic possibilities.

Since the total number of rows in A_r is 36, we have:

$$a + b + c + d = 36. \tag{4.1}$$

There are exactly 16 ones in any column (excepting the last) of A_r. Therefore we have the following two equations:

$$b + d = 16, \text{ and} \tag{4.2}$$

$$c + d = 16. \tag{4.3}$$

Finally, since any pair of columns intersects in either 4 or 6 places in A_d, and in exactly 12 places in A_{64}, we have:

$$d = 12 - (4 \text{ or } 6). \tag{4.4}$$

There are two solutions to this system of equations, namely $\{a = 12, b = 8, c = 8, d = 8\}$ and $\{a = 10, b = 10, c = 10, d = 6\}$.

Case	Solution	# of Candidates
1	3	1133
	4	1133
	6	1896
	10	2372
	13	1896
	14	2372
	17	2372
	18	2372
26	33	528
	37	1641
	39	556
	67	1774
	68	1774
164	2	419
	5	300
	7	398
171	34	528
237	2	339
	4	339
	5	353
	6	1732
	8	339
	9	339

Table 4.1: Number of Candidates Per Derived Design

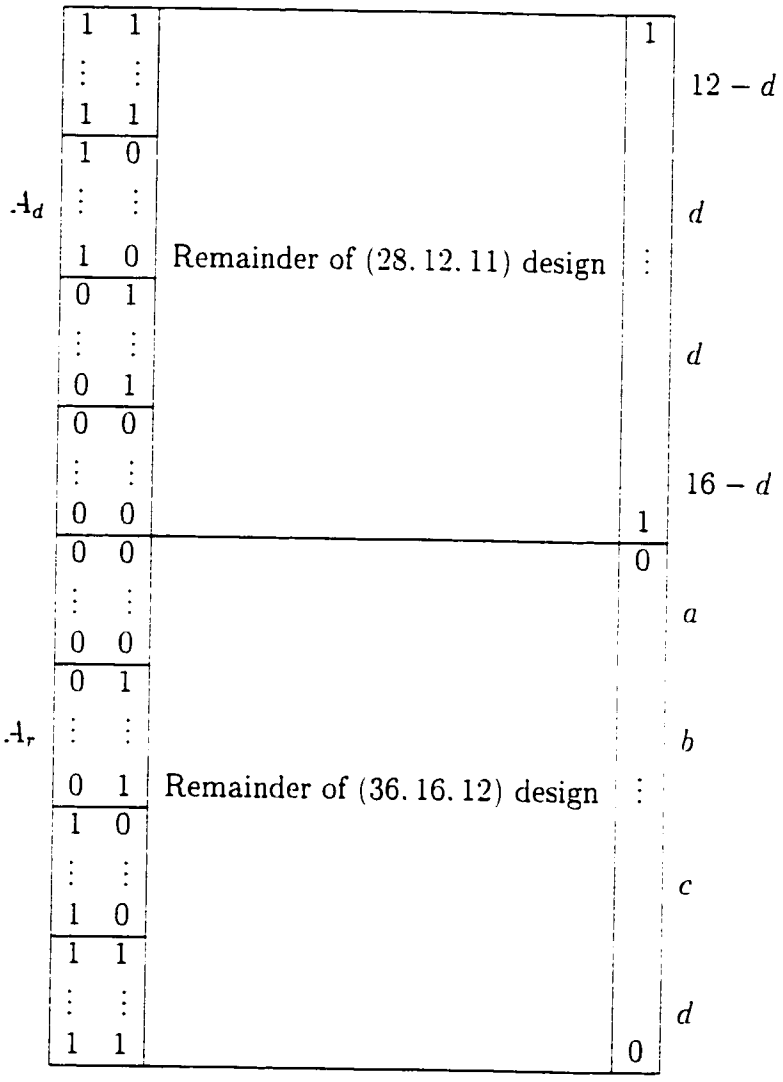

Figure 4.2: Structure of A_{64}. with Column Intersection Information

For each of the remaining 23 2-(28, 12, 11) designs which did not extend to 2-(64, 28, 12) designs assuming an automorphism of order 7, we try to fill the first a rows of A_r with vectors from the candidate list which contain '00' in columns 1 and 2, and then in turn, the next b, c and d rows with vectors which contain '01', '10' and '11' respectively in columns 1 and 2. To further eliminate isomorphic possibilities, we insist that within each of these groups, later rows are filled with vectors which appear later in the candidate list than those used for earlier rows.

During the search, we found exactly four 2-(28, 12, 11) designs which do not extend to 2-(64, 28, 12) designs. They are the first examples of non-embeddable quasi-symmetric designs. The base rows of the incidence matrices for these four designs are given in Appendix A. Each base row gives 6 other rows by cyclically permuting the 7 columns in each of the 9 column blocks.

4.4 Counting Symmetric and Residual Designs

For each of the twenty-three 2-(28, 12, 11) designs, we attempted to count the total number of extensions. The number of extensions for some of the designs is very large: for those designs, it was necessary to stop the search simply upon finding that extensions existed. Accordingly, it is also not feasible to find the total number of non-isomorphic extensions for all 23 designs. For example, testing for isomorphisms between the 61440 extensions to case 164, solution 5 required about two weeks of computing time on an Alpha-based DEC 2100 server model A500M. The results of this process are summarized in Table 4.2. Here, "Autogp. size" is the size of the automorphism group for the quasi-symmetric 2-(28,12,11) design, ext_{tot} is the number of extensions per 2-(28, 12, 11) design, and ext_{ni} is the number of non-isomorphic extensions per 2-(28, 12, 11) design. In particular, our search shows that there are at least 8784 non-isomorphic symmetric 2-(64,28,12) designs, obtained as extensions of solution 5 of Case 164.

Next we considered the residual designs corresponding to these extensions. Recall that these "extensions" are in fact symmetric 2-(64, 28, 12) designs composed of a quasi-symmetric derived 2-(28, 12, 11) design (from which it is "extended"), a quasi-

Case	Solution	Autogp. size	ext_{tot}	ext_{ni}	res_{ni}
1	3	21	0	0	0
	4	21	0	0	0
	6	84	many	?	?
	10	84	92	14	14
	13	84	many	?	?
	14	84	92	14	14
	17	21	92	16	14
	18	21	92	16	14
26	33	21	0	0	0
	37	21	1	1	1
	39	21	129	13	13
	67	21	8	2	2
	68	21	8	2	2
164	2	7	61440	?	?
	5	7	61440	8784	8784
	7	7	61440	?	?
171	34	21	0	0	0
237	2	7	61440	?	?
	4	7	61440	?	?
	5	7	61440	?	?
	6	7	61440	?	?
	8	7	61440	?	?
	9	7	61440	?	?

Table 4.2: Summary of Extensions and Corresponding Residual Designs

Automorphism Group Size	Number of designs
1	6
3	28
7	2
12	16
21	18
84	8

Table 4.3: Automorphism Group Statistics, 2-(64, 28, 12) Designs

symmetric residual 2-(36, 16, 12) design, and an extra column. Again, we considered only those 2-(28, 12, 11) designs for which we know the number of their non-isomorphic extensions.

The number of extensions is of course equal to the number of corresponding quasi-symmetric residual designs obtained in this manner. The number of non-isomorphic residual designs for each of these cases is also shown in Table 4.2, where res_{ni} is the number of non-isomorphic quasi-symmetric residual designs corresponding to each derived design. Amongst these quasi-symmetric 2-(36,16,12) designs are some of the first known examples with trivial automorphism groups. The incidence matrix of one such design is given in Appendix B.

Among the designs resulting in a small number of extensions, there are a total of 78 non-isomorphic extensions: each of the extensions which were non-isomorphic to all other extensions from the same design, were also non-isomorphic to all other extensions from the other "small-case" designs. For each of these 78 extensions, we found its automorphism group and 2-rank. Statistics, in the form of the number of cases having each group size and 2-rank, are found in Tables 4.3 and 4.4.

In many cases, residual designs corresponding to the same derived design were isomorphic to residual designs corresponding to other derived designs. Indeed, there are only 30 non-isomorphic residual designs from amongst the 78 "small-case" designs examined. For each of these 30 designs, we found its automorphism group and 2-rank. Of the thirty, five had a 2-rank of 16 and twenty-five had a 2-rank of 17. Statistics, in the form of the number of cases having each group size, are found in Table 4.5.

2-rank	Number of designs
17	2
18	7
19	5
20	3
21	10
22	7
23	3
24	6
25	3
26	5
27	16
28	11

Table 4.4: 2-rank Statistics. 2-(64, 28, 12) Designs

Automorphism Group Size	Number of designs
1	4
3	8
12	8
21	6
84	4

Table 4.5: Automorphism Group Statistics. Residual Designs

4.5 Summary

We have now the first examples of quasi-symmetric quasi-derived designs which are not derived designs, and the first examples of quasi-symmetric 2-(36.16.12) designs with trivial automorphism groups. We have also improved on the numbers of 2-(64.28.12) designs and 2-(36.16.12) designs as found in the Handbook of Combinatorial Designs [31].

We summarize the above results with the following theorems.

Theorem 4.1 *Not every quasi-symmetric quasi-derived design is a derived design.*

Theorem 4.2 *There are at least 8784 non-isomorphic 2-(64.28.12) designs.*

Theorem 4.3 *There are at least 8784 non-isomorphic 2-(36.16.12) designs.*

Theorem 4.4 *There are at least 4 non-isomorphic 2-(36.16.12) designs with a trivial automorphism group.*

4.6 Further Work

In the process of reviewing a paper based on this work, an anonymous referee made the following observation:

> The 4-block intersection graph of a QSD(28.12.11) is strongly regular with parameters (63.30.13.15). Adding a point to this graph yields a two-graph which can be made regular by Seidel switching. The switching set of size 35 consists of 5 disjoint 7-cliques in the 63-graph. For example, the first non-extendable design (of Appendix A) gives a 63-graph which contains exactly 5 disjoint 7-cliques:
>
> $$(8,\ldots,14),(15,\ldots,21),(22,\ldots,28),(29,\ldots,35),(36,\ldots,44),$$
>
> yielding a srg(64.35.18.20) the complement of which is a symmetric 2-(64.28.12) design with group of order 21 (having a polarity with no absolute points).

We verified that each of the four non-extendable designs yields a symmetric design via this switching. It is possible that each of these designs yields more than one symmetric design in this manner. but this has yet to be verified.

Chapter 5

Estimates in the Search for a $(46, 6, 1)$ Block Design

In this chapter we examine the search for a 2-(46, 6, 1) block design. We have the following conjecture:

Conjecture 5.1 *There is no* (46, 6, 1) *design.*

We show how the search may be divided into two cases. A search assuming the first case had previously been completed by Janssen, Lam & Thiel, and revealed no designs. We thus focus on a method of completing the search assuming the second case, and present estimates of the time required to complete such a search. At the end of the chapter, we include an update on the results of this search.

5.1 Search Procedure

The procedure we shall follow in our search is to attempt to complete the incidence matrix of a (46, 6, 1) design. The incidence matrix of such a design has 46 rows and 69 columns. In each row, there are 9 ones. In each column, there are 6 ones. Each pair of rows intersect in exactly one column.

Any row and column permutations preserve the properties of an incidence matrix. For this reason, we can place the 9 ones of the first row in any columns we wish. Let us place them in columns 1, 2 and 63–69:

$$r1 = 110\,0\cdots0\,0000000\,0000000\,1111111.$$

Now consider the second row. The first two rows must intersect in exactly one column. Using column permutations we can assume that the first two rows intersect in column 1, and that the other ones are placed in columns 3 and 56–62:

$$r2 = 101\,0\cdots0\,0000000\,1111111\,0000000.$$

Consider column 2. Each column must contain 6 ones, so using row permutations we can place a one in row 3. This takes care of the intersection between row 3 and row 1. Using column permutations we can assume that rows 2 and 3 intersect in column 3, and that the other ones for row 3 are placed in columns 49–55:

$$r3 = 011\,0\cdots0\,1111111\,0000000\,0000000.$$

Observe that in the first 3 columns and first $\binom{3}{2} = 3$ rows, each pair of columns intersect exactly once, in distinct rows. We call this type of configuration a "c3" configuration. Next we shall examine if there also must exist a "c4" configuration — 4 columns and $\binom{4}{2} = 6$ rows in which each pair of columns intersect exactly once.

In each of the first three columns, two ones have been placed so far: let us place the remaining ones in each of these columns. In each of the remaining rows, we cannot place more than a single one in the first three columns without creating a situtation in which a pair of rows intersect in more than one column, which is not permitted. Row permutations allow us to place the remaining ones for column 1 in rows 4–7, the remaining ones for column 2 in rows 8–11, and the remaining ones for column 3 in rows 12–15. Thus we have the partially-filled incidence matrix shown in Fig. 5.1.

Consider the intersections between each of rows 4–15. We have to ensure that each of rows 4–7 intersects with each of rows 8–15 and that each of rows 8–11 intersects with each of rows 12–15. Thus we need to ensure that $\binom{3}{2} \cdot 4 \cdot 4 = 48$ row-intersections occur. Clearly these intersections cannot occur in any of the last 21 columns, or, as previously mentioned, any of the first 3 columns. Therefore we are restricted to a total of $69 - 3 - 21 = 45$ columns.

Because we require 48 row-intersections in 45 columns, clearly not all of the row-intersections occur in disjoint columns. More specifically, there must be rows i, j and

110	0 ... 0	0000000	0000000	1111111
101	0 ... 0	0000000	1111111	0000000
011	0 ... 0	1111111	0000000	0000000
100			0000000	0000000
100			0000000	0000000
100			0000000	0000000
100			0000000	0000000
010		0000000		0000000
010		0000000		0000000
010		0000000		0000000
010		0000000		0000000
001		0000000	0000000	
001		0000000	0000000	
001		0000000	0000000	
001		0000000	0000000	
000				
⋮				
000				

Figure 5.1: Structure of Incidence Matrix for a (46, 6, 1) Design

k, $4 \leq i \leq 7$, $8 \leq j \leq 11$, $12 \leq k \leq 15$, which all intersect in the same column, say column c. Observe that column c intersects each of columns 1–3 exactly once. Therefore, rows 1–3 and i, j and k, together with columns 1–3 and c, form a "c4" configuration.

We can therefore assume that the incidence matrix of any (46, 6, 1) design must contain a "c4" configuration. It is also possible for a "c5" configuration to exist, where a "c5" is 5 columns and 10 rows in which each pair of columns intersect exactly once. Therefore the search may be divided into two cases — the first of which assumes there is no "c5", and the second of which assumes there is a "c5". If we divide the search in this manner, then each case has the extra information regarding the existence of a "c5" configuration. This extra information reduces the size of the search, thereby making it easier to complete.

5.2 Incidence Matrix with a "c4"

This case assumes the existence of a "c4" configuration but no "c5" configuration. A search assuming this case, performed by Janssen, Lam & Thiel, uses techniques similar to those described in the next section. During this search, no designs were found. Therefore, if a $(46, 6, 1)$ design exists, its incidence matrix must contain a "c5" configuration.

5.3 Incidence Matrix with a "c5"

We now consider the case in which the incidence matrix contains a "c5" configuration. To aid our search, we would like to ensure that the incidence matrix is in a convenient structure. We can assume that the incidence matrix has the structure shown in Fig. 5.2, as explained below.

We shall place the "c5" configuration in the first ten rows and first five columns of the incidence matrix. We shall call these first ten rows section RA, and these first five columns section CA. We must ensure that each pair of rows in section RA intersect in exactly one column. There are fifteen pairs of rows which do not already intersect in section CA. Since in section RA, the remaining columns can each have at most 2 ones, we require fifteen columns, each containing exactly 2 ones in section RA, to create the required intersections. We can assume that these are columns 6–20, and shall call these columns section CB. Each row in section RA requires 4 more ones, and since no other row intersections in section RA are permitted, we require $10 \cdot 4 = 40$ columns each containing a single one in section RA. We can assume that these are columns 21–60, and shall call these columns section CC. We shall call the remaining $69 - 5 - 15 - 40 = 9$ columns section CD: each of these columns contain only zeroes in section RA.

In section CA, each column contains 4 ones in section RA, thus each of these columns require 2 more ones. In section CA, if we place more than a single one in any row outside of section RA, then that row would intersect a row of section RA in more than one column. Therefore we require $5 \cdot 2 = 10$ rows each containing a single

one in section CA. We can assume that these are rows 11–20: we shall call these rows section RB.

For each row in section RB, let b, c and d be the number of ones in sections CB, CC and CD respectively. Since each row has 9 ones and section CA contains a single one, we have:

$$1 + b + c + d = 9.$$

By counting intersections with the 10 rows in section RA, we have

$$4 + 2b + c = 10.$$

which simplifies to

$$2b + c = 6. \tag{5.1}$$

Since $c \geq 0$, Eq.(5.1) implies $0 \leq b \leq 3$. Solving the equations, we have the following possibilities: $\{b = 0, c = 6, d = 2\}$, $\{b = 1, c = 4, d = 3\}$, $\{b = 2, c = 2, d = 4\}$ and $\{b = 3, c = 0, d = 5\}$.

We shall call the remaining $46 - 10 - 10 = 26$ rows section RC: each of these rows contain only zeroes in section CA.

In the following, the notation (RX, CY) refers to the portion of the incidence matrix confined to the rows of section RX and the columns of section CY, where X=A, B or C and Y=A, B, C or D.

Now let us consider section (RB, CD) further. In this section there are at least 2 ones per row: since there are 10 rows, there must be at least 20 ones in total in this section. Since there are 9 columns, at least 1 column must contain 3 ones or more. Furthermore, there are at most 5 ones per column since 5 (disjoint) pairs of these rows already intersect in section CA.

We divide the search assuming a "c5" configuration into three subcases, based on the maximum number of ones in any column of section (RB, CD):

- At least one column in (RB, CD) contains 5 ones, and all other columns contain at most 5 ones.

- At least one column in (RB, CD) contains 4 ones, and all other columns contain at most 4 ones, and

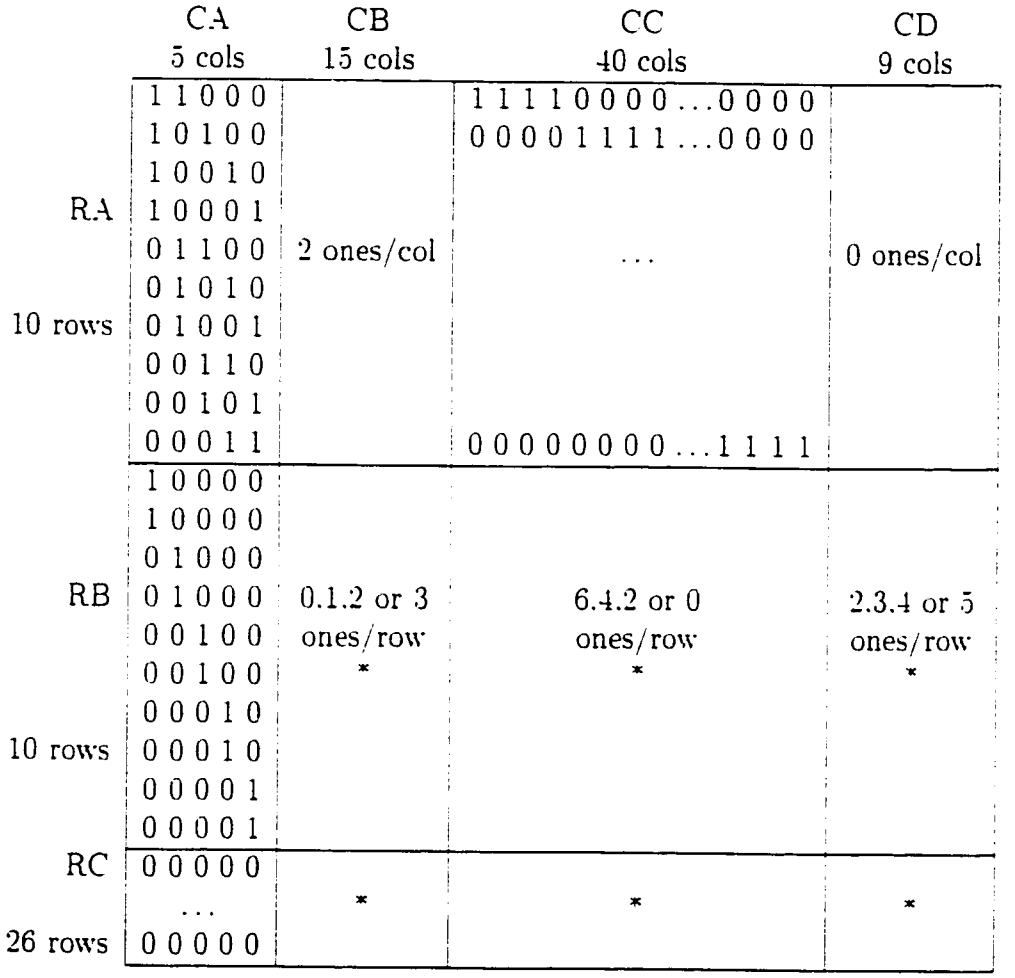

Figure 5.2: Structure of Incidence Matrix Containing a "c5"

- Each column contains at most 3 ones.

The basic procedure to complete the search is as follows. For each of the 3 subcases, we fill in selected portions of the incidence matrix using BDX. This step creates a number of "images", each of which is then passed on to a specialized program, which attempts to complete the remainder of the incidence matrix.

5.3.1 A Column of 5 Ones

We can assume that column 61 in section (RB, CD) contains exactly 5 ones and that all other columns in this section contain at most 5 ones. Up to isomorphism, there is only one way to complete column 61, namely with ones in rows 11, 13, 15, 17, 19 and 21. Completing the remainder of rows 11, 13, 15, 17 and 19 gives 9 non-isomorphic cases.

For each of these cases, we further complete selected portions of the incidence matrix using BDX. These portions are selected based on the orbit structure of the automorphism group of the partially completed matrix, with the goal of ensuring an efficient isomorph rejection. A useful side effect is that it breaks down the search into smaller subcases which can easily be run on a network of processors. More detail follows.

Selecting portions of the incidence matrix

It is of some interest to look at how we decide which portions of the incidence matrix should be completed. We are interested in the part of the incidence matrix which already has some completed entries, since these produce restrictions on the remaining entries in that part.

Specifically, therefore, we consider rows 1–21. Columns 1–5 and 61, and rows 1–10, together with rows 11, 13, 15, 17 and 19, are already completely filled. Hence we shall concentrate on filling portions of rows 12, 14, 16, 18, 20 and 21, in columns 6–60 and 62–69.

We consider the automorphism group on rows 1–21. In a partially completed binary matrix, we can think of each entry having one of 3 possible values, namely 0,

1, or "unknown", where "unknown" indicates that this entry has yet to be completed.

Suppose the automorphism group on rows 1–21 contains the column orbit $(x\ y)$. This indicates that if we exchange the entries in columns x and y, each of rows 1–21 remain unchanged. Specifically, if a given row contains a particular value (0, 1 or "unknown") in column x, then it also must contain the *same* value in column y.

Now consider further the situation in which a given row has "unknown" entries in columns x and y. Suppose we choose to find all possible ways to complete (i.e. place either a 0 or 1 in) those entries. If a case appears in which the row contains different entries in columns x and y, then the orbit $(x\ y)$ is broken, and the size of the automorphism group can be reduced accordingly. Note that if, for example, the row contains a 1 in column x and a 0 in column y, this is isomorphic to the case in which the row contains a 0 in column x and a 1 in column y. We only need to consider one of these 2 cases.

It is important when completing unknown entries, to complete *all* of the entries within a particular orbit. Suppose in our above example, we complete only the entries in column x. Then it will appear as if the orbit $(x,\ y)$ has been broken, since now column x contains only 0's or 1's, and column y still contains "unknown" entries. However, upon later completing the entries in column y, the orbit will magically "mend" itself if we have placed the same entries in column y as we previously did in column x. Furthermore, we have no way of performing isomorphism testing between the "unknown" entries and the entries with 0's or 1's.

The general procedure we use is as follows. First, we complete all entries in rows and columns within a selected orbit, checking for isomorphisms among solutions. In general, we select orbits which are fairly small, and orbits which "match up" against 1's in other rows; there is, however, a certain amount of trial and error in the selection.

Next, we check the automorphism group, again restricted to rows 1–21, of each of the resulting non-isomorphic solutions. When most solutions have an automorphism group of size 1, little improvement can be made by completing other portions of rows 1–21. Specifically, there will be very few solutions rejected by isomorphism testing if we continue further in this manner. Therefore, at this point, it makes more sense for

	11111111112	22222222233333333334444444444555555555556	666666666
12345	678901234567890	12345678901234567890123456789012345678901234567890	123456789
11000	111000000000000	11110000000000000000000000000000000000000000	000000000
10100	000111000000000	00001111000000000000000000000000000000000000	000000000
10010	000000111000000	00000000111100000000000000000000000000000000	000000000
10001	000000000111000	00000000000011110000000000000000000000000000	000000000
01100	000000100100100	00000000000000001111000000000000000000000000	000000000
01010	000100000010010	00000000000000000000111100000000000000000000	000000000
01001	000010010000001	00000000000000000000000011110000000000000000	000000000
00110	100000000001001	00000000000000000000000000001111000000000000	000000000
00101	010000001000010	00000000000000000000000000000000111100000000	000000000
00011	001001000000100	00000000000000000000000000000000000000001111	000000000
10000	000000000000111	00000000000000000000000000000000000000000000	111110000
10000	000000000000000	0000000000000000.....	00000....
01000	000000000000000	00001000100010000000000000000100010001000	100001000
01000	00000.00.00.000	00000...0...0...0000000000000...0...0...	0....0...
00100	000000000000000	10000000010001000000100010000000000000100	100000100
00100	00.0000.00.0000	0...0000.0...0..00000...0...00000000.0..	0.....0..
00010	000000000000000	01000100000000101000000001000000010000000	100000010
00010	0.00.0000.00000	.0...0..0000..0.0...0000.0..0000.0..0000	0......0.
00001	000000000000000	00100010001000000100010000000100000000000	100000001
00001	.00.00.00000000	..0...0...0.0000.0...0..0000.0..00000000	0.......0
00000	000	000.000.000.000.00..00..00..00..00..00..	100000000

Figure 5.3: Incidence Matrix for Case 1

a specialized program to take over.

To demonstrate, we shall look at the first of the 9 non-isomorphic cases in which there is a column of 5 ones in section (RB,CD). The first 21 rows of this partially-completed matrix are shown in Fig. 5.3, in which a '.' indicates an unknown entry.

For this case, the automorphism group is of size $2^4 \cdot 3$. The row orbits are: (1 2 3 4 13 15 17 19), (5 6 7 8 9 10), (12 21) and (14 16 18 20). Row 11 is fixed. The column orbits are: (1 61), (2 3 4 5), (6 7 8 9 10 11 12 13 14 15 16 17 37 38 41 42 45 46 49 50 53 54 57 58), (18 19 20), (21 22 23 25 26 27 29 30 31 33 34 35), (24 28 32 36 66 67 68 69), (39 40 43 44 47 48 51 52 55 56 59 60) and (62 63 64 65).

Since rows 12 and 21 are in the same orbit, we completed these two rows, in all columns. This gave a total of 147 solutions, of which 113 had an automorphism group of size 1. Since the large majority of the solutions had an automorphism group of size 1, we do not attempt to complete any other portions before passing these solutions to the specialized program.

Case	#Subcases	Est. Avg. Time Per Subcase	Est. Total Time
1	147	5.85 s	0.24 hrs
2	449	21.45 s	2.67 hrs
3	1111	36.64 s	11.31 hrs
4	1500	80.59 s	33.58 hrs
5	354	20.88 s	2.05 hrs
6	144	101.00 s	4.04 hrs
7	309	1051.48 s	90.25 hrs
8	1753	289.63 s	141.04 hrs
9	1493	120.24 s	49.87 hrs
Total	7260	166.14 s	335.05 hrs (14 days)

Table 5.1: Estimates Assuming a Column of 5 Ones

Estimates assuming a column of 5 ones

For each of the 9 cases, we used BDX to complete selected portions of the incidence matrix as described. From each case, several subcases were chosen, and passed to the specialized program in order to obtain an estimate of the overall time required. All estimates are based on an Alpha-based DEC 2100 server model A500M and are summarized in Table 5.1.

To compare these estimates to the actual results obtained in a search assuming this case, see section 5.5.

5.3.2 A Column of 4 Ones

We can assume that column 61 in section (RB, CD) contains exactly 4 ones and that all other columns in this section contain at most 4 ones. Up to isomorphism, there is only one way to complete column 61, namely with ones in rows 11, 13, 15, 17, 21 and 22. Completing the remainder of rows 11, 13, 15 and 17 gives 33 non-isomorphic cases.

As before, for each of the 33 cases we used BDX to complete selected portions of the incidence matrix, and then passed several subcases to the specialized program

for estimation purposes. The results of this process are summarized in Table 5.2.

To compare these estimates to the actual results obtained in a search assuming this case, see section 5.5.

5.3.3 Columns of 3 Ones

Next we consider the case in which each column in section (RB, CD) of the incidence matrix contains at most 3 ones. We subdivide this case based on the number of columns in this section containing exactly 3 ones, taking into account the restriction mentioned earlier that the total number of ones in this section is at least 20. Therefore we have eight subcases, since there must be at least two, and at most nine, columns containing exactly 3 ones in section (RB, CD).

For each of these subcases, we used BDX to complete all columns in section (RB, CD) which contain exactly 3 ones. For those subcases which have very few columns of 3 ones, we also filled in a few other columns in this section, creating a number of images. We selected several images as test cases. For each test case, we used BDX to complete several rows — generally rows 11–15. We thereby obtained estimates both of the time required by BDX for all images, and the total number of row completions. These must each be passed to the specialized program, which will attempt to complete the remainder of the matrix. To obtain estimates of the time required by the specialized program, we selected several and passed them to the specialized program.

These estimates are summarized in Table 5.3. The time estimates shown in this table include both the estimate of the time required by BDX, and the estimate of the time required by the specialized program. In this table, the notation 'Ek' is used as shortform for '$\times 10^k$'. For example, '5.5 E4' means '5.5×10^4'.

5.4 Summary

The search for a (46, 6, 1) design may be divided into two cases, the first of which assumes a "c4" but no "c5", and the second of which assumes a "c5". A search assuming the first case found no designs. Therefore if a (46, 6, 1) design exists, it

Case	#Subcases	Est. Avg. Time Per Subcase	Est. Total Time
1	298	729.13 s	60.4 hrs
2	150	206.97 s	8.6 hrs
3	214	2628.08 s	156.2 hrs
4	541	24.12 s	3.6 hrs
5	306	849.53 s	72.2 hrs
6	99	1388.54 s	38.2 hrs
7	404	506.69 s	56.9 hrs
8	48	3051.24 s	46.7 hrs
9	539	4036.06 s	604.3 hrs
10	418	951.37 s	110.5 hrs
11	405	1140.83 s	128.3 hrs
12	106	1868.67 s	55.0 hrs
13	723	4547.80 s	913.3 hrs
14	4750	82.55 s	108.9 hrs
15	5993	67.88 s	113.0 hrs
16	535	6864.46 s	1020.1 hrs
17	44	2238.34 s	27.2 hrs
18	918	88.49 s	22.6 hrs
19	393	985.51 s	107.6 hrs
20	1734	2228.54 s	1073.4 hrs
21	3284	14.75 s	13.5 hrs
22	1575	2217.90 s	970.3 hrs
23	2037	6635.49 s	3754.6 hrs
24	1803	22.04 s	11.0 hrs
25	20462	83.58 s	475.1 hrs
26	3317	54.12 s	49.9 hrs
27	550	7031.11 s	1074.2 hrs
28	5932	919.11 s	1514.5 hrs
29	2422	707.46 s	476.0 hrs
30	2268	9176.29 s	5781.1 hrs
31	4215	1250.84 s	1464.5 hrs
32	2077	30659.16 s	17688.6 hrs
33	6970	1558.19 s	3016.8 hrs
Total	75530	1955.01 s	41017.1 hrs (1709 days)

Table 5.2: Estimates Assuming a Column of 4 Ones

#Cols of 3 Ones	#Column Completions	Est. #Row Completions	Est. Avg. Time Per Subcase	Est. Total Time
9	1499	5.5 E4	255.26 s	106.3 hrs
8	3735	4.2 E6	2233.73 s	2317.5 hrs
7	3730	4.4 E6	9070.41 s	9398.0 hrs
6	2061	5.5 E6	80217.08 s	45924.3 hrs
5	659	2.9 E6	82908.41 s	15176.9 hrs
4	3151	1.1 E7	8682.28 s	7599.4 hrs
3	24005	6.6 E5	90.32 s	602.6 hrs
2	902	2.3 E4	176.70 s	44.2 hrs
Total	39742	2.9 E7	7352.62 s	81169.2 hrs (3382 days)

Table 5.3: Estimates Assuming a Column of 3 Ones

must contain a "c5" configuration.

Estimates indicate that we will require some 5100 machine-days to complete the search assuming this remaining case. Clearly if this search is to be completed, it must be broken up. Fortunately the search may be divided into a large number of subcases. On average, the time to run each subcase is only a few minutes.

5.5 Update

A search assuming the existence of a "c5" configuration has recently been performed by Lam & Thiel. Recall that the search assuming this case may be divided into 3 subcases, based on the maximum number of ones in any column of section (RB, CD):

- At least one column in (RB, CD) contains 5 ones, and all other columns contain at most 5 ones,
- At least one column in (RB, CD) contains 4 ones, and all other columns contain at most 4 ones, and
- Each column contains at most 3 ones.

The results of a search assuming the case in which at least one column in (RB, CD) contains 5 ones, and all others contain at most 5 ones, are shown in Table 5.4.

Case	#Subcases	Total Time
1	147	< 1 hrs
2	449	8 hrs
3	1111	22 hrs
4	1500	57 hrs
5	354	37 hrs
6	144	6 hrs
7	309	163 hrs
8	1753	282 hrs
9	1493	78 hrs
Total	7260	653 hrs

Table 5.4: Results Assuming a Column of 5 Ones

In this table, the time is normalized to an execution on a Sparc-10 computer, which is approximately 2.35 times slower than an Alpha-based DEC 2100 server model A500M.

A search assuming the case in which at least one column in (RB, CD) contains 4 ones, and all others contain at most 4 ones, was further subdivided based on the number of columns (1-5) containing 4 ones. The results are summarized in Table 5.5. In this table, cases 1-33 are the original 33 cases, assuming exactly 1 column containing 4 ones. Cases $i1_j$-$i4_j$ are subcases assuming exactly j columns containing 4 ones. Note that the overall number of subcases considered is 138 955 and the overall execution time was 89 986 hours. Again, the time is normalized to an execution on a Sparc-10 computer.

The results of a search assuming the case in which each column in (RB, CD) contains at most 3 ones are shown in Table 5.6. The method is essentially the same as that described in section 5.3.3, except that the method of generating BDX images for row completions has been automated and improved. The number of subcases in this table is equal to the number of column completions in Table 5.3, except in 2 cases. For the case in which there are 4 columns of 3 ones, *only* those columns were completed in section (RB, CD), rather than also completing other columns in this section as in Table 5.3; this gives 152 subcases rather than 3151 subcases. For

Case	#Subcases	Total Time
1	298	13 hrs
2	150	8 hrs
3	214	66 hrs
4	541	5 hrs
5	306	39 hrs
6	99	33 hrs
7	404	55 hrs
8	48	39 hrs
9	539	677 hrs
10	418	77 hrs
11	405	86 hrs
12	106	21 hrs
13	723	603 hrs
14	901	9 hrs
15	44	24 hrs
16	535	695 hrs
17	44	22 hrs
18	918	13 hrs
19	393	65 hrs
20	1734	1199 hrs
21	3211	10 hrs
22	1575	821 hrs
23	2037	3910 hrs
24	1803	6 hrs
25	535	321 hrs
26	3317	18 hrs
27	550	901 hrs
28	5927	3507 hrs
29	2422	284 hrs
30	2268	5747 hrs
31	4215	1002 hrs
32	2077	11237 hrs
33	6970	3293 hrs
Total	45727	34806 hrs

Case	#Subcases	Total Time
$i1_2$	3093	5417 hrs
$i2_2$	4804	33417 hrs
$i3_2$	3267	1570 hrs
$i4_2$	25204	10781 hrs
$i1_3$	12579	970 hrs
$i2_3$	408	1523 hrs
$i3_3$	408	1388 hrs
$i1_4$	1787	7 hrs
$i2_4$	1748	26 hrs
$i3_4$	38331	79 hrs
$i1_5$	1599	2 hrs
Total	93228	55180 hrs

Table 5.5: Results Assuming a Column of 4 Ones

#Cols of 3 Ones	#Subcases	Total Time
9	1499	400 hrs
8	3735	4159 hrs
7	3730	13937 hrs
6	2061	26107 hrs
5	659	24609 hrs
4	152	9672 hrs
3	25	1390 hrs
2	902	68 hrs
Total	12763	80342 hrs

Table 5.6: Results Assuming a Column of 3 Ones

a similar reason, there are 25 subcases rather than 24 005 subcases for the case in which there are 3 columns of 3 ones. The time given in Table 5.6 is normalized to an execution on a Sparc-10 computer, and includes only the time required for the specialized program.

Note that in the case in which there are 9 columns of 3 ones, the given time is 400 hours. This is in fact an approximation: for each subcase, the time printed by the specialized program is rounded to the nearest hour. For this case, there are a large number of subcases which required less than 30 minutes, and thus the program printed "0 hrs" as the time required for these subcases. We therefore approximate the time for these subcases as 15 minutes each.

No designs were found for any of the 3 cases. Since our search revealed no designs we can conclude that Conjecture 5.1 is true. One may therefore restate it as a theorem:

Theorem 5.1 *There is no* $(46, 6, 1)$ *design.*

5.6 Further Work

The completion of this search answers the question of whether there exists a $(46, 6, 1)$ design. One may now consider the next smallest case with $k = 6$ and $\lambda = 1$, namely $(51, 6, 1)$.

Chapter 6

Construction of $(48, 24, 12)$ Self-Dual Doubly-Even Codes

We now examine the problem of enumerating $(48, 24, 12)$ self-dual doubly-even codes. Our desired method of enumeration is to directly construct these codes, by way of attempting to complete their generator matrices.

The Extended Quadratic Residue Code is the only known $(48, 24, 12)$ self-dual doubly-even code. We have the following conjecture:

Conjecture 6.1 *The Extended Quadratic Residue Code is the only* $(48, 24, 12)$ *self-dual doubly-even code.*

6.1 Introduction

As shown in [19, 20], the generator matrix can be organized as shown in Fig. 6.1, where RM is a $(24, b, 12)$ code obtained by taking $3 \cdot 2^{4-b}$ copies of the 1st-order Reed-Muller code $R(1, b-1)$. The code B, together with the code RM, form the dual of RM, therefore all vectors in B are chosen from RM^{-}.

In the same work it is shown that $2 \leq b \leq 4$. One may therefore divide the search into 3 cases, namely $b = 2$, $b = 3$ and $b = 4$. The results of a search assuming $b = 4$ are reported in [19, 20]. In this search, only codes isomorphic to the Extended Quadratic Residue code QR were found. The Extended Quadratic Residue code therefore remains the only known $(48, 24, 12)$ self-dual doubly-even code.

In this thesis, we examine the two remaining cases, namely $b = 2$ and $b = 3$.

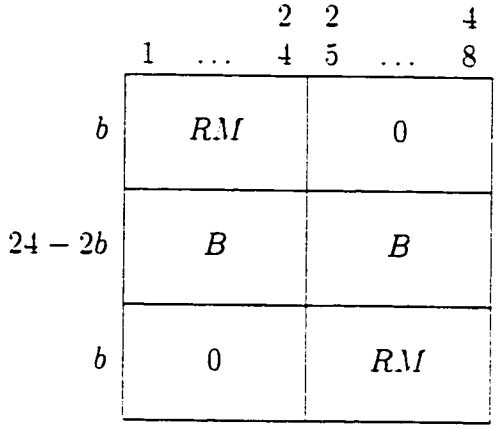

Figure 6.1: Organization of Generator Matrix for C_{48}

6.2 Organization of Generator Matrix for $b = 2$

Let C_{48} be a (48, 24, 12) self-dual doubly-even code in which the generator matrix is organized as shown in Fig. 6.1, with $b = 2$.

In this case, the weight enumerator of RM is:

$$a_0 = a_{24} = 1.$$
$$a_{12} = 2.$$

From the MacWilliams Identity (Eq. (2.5)), we can obtain the weight enumerator of RM^-, namely:

$$a_0 = a_{24} = 1.$$
$$a_2 = a_{22} = 132.$$
$$a_4 = a_{20} = 5\ 346.$$
$$a_6 = a_{18} = 67\ 188.$$
$$a_8 = a_{16} = 367\ 983.$$
$$a_{10} = a_{14} = 980\ 232.$$
$$a_{12} = 1\ 352\ 540.$$

Let M be the generator matrix of C_{48}.

Two sections of M remain to be filled in. The first of these sections is the subspace generated by rows 3–22, restricted to their first 24 columns. Call these restricted rows

$row_i(left)$, $3 \leq i \leq 22$. The second of these sections is the subspace generated by rows 3–22, restricted to their last 24 columns. Call these restricted rows $row_i(right)$, $3 \leq i \leq 22$.

Note that in both of these sections, all vectors are in RM^-. Therefore in particular, $row_i(left)$ and $row_i(right)$ must both be chosen from RM^-, for $3 \leq i \leq 22$.

Because the number of weight-12 codewords in C_{48} is non-zero, we use as many of these as possible when trying to complete the remainder of M. This use of weight-12 codewords is aimed at reducing the number of possibilities we must consider for each row.

Similarly, we would like to reduce the number of possibilities for $row_i(left)$ and $row_i(right)$. Therefore we shall attempt to complete the remainder of M using weight-10 words from RM^- for $row_i(left)$, and weight-2 words from RM^- for $row_i(right)$.

In fact, we may assume that M has the form shown in Fig. 6.2. The reasoning is as follows.

First consider $row_i(right)$, $3 \leq i \leq 22$. Based on the configuration in the last two rows, we may divide $row_i(right)$ into 2 column-blocks, each of length 12. There are 132 codewords of weight 2 in RM^-, and we shall attempt to use these to complete $row_i(right)$, $3 \leq i \leq 22$. If we place a single one in each of the two column-blocks of $row_i(right)$, then $row_i + row_{23} + row_{24}$ is a vector of weight 22, which is not doubly-even. Therefore we must place both ones in the same column-block. There are $\binom{12}{2} = 66$ ways to place 2 ones in a column-block, and thus 66 codewords of weight 2 in RM^- may be obtained from each column-block. Exactly 11 of these 66 codewords are linearly independent. All 66 codewords in the first column-block may be generated by $row_3(right), \ldots, row_{12}(right)$ and $row_{23}(right)$. All 66 codewords in the second column-block may be generated by $row_{13}(right), \ldots, row_{22}(right)$ and $row_{24}(right)$. All of the required 132 codewords of weight 2 in RM^- may thus be obtained from the 2 column-blocks. One may now verify that RM^- is generated by $row_3(right), \ldots, row_{24}(right)$.

Observe that we have divided the remaining incomplete rows of the generator

<table>
<tr><td>111111111111
000000000000</td><td>000000000000
111111111111</td><td>000000000000
000000000000</td><td>000000000000
000000000000</td></tr>
<tr><td colspan="2">wt 10</td><td>110000000000
101000000000
100100000000
100010000000
100001000000
100000100000
100000010000
100000001000
100000000100
100000000010</td><td></td></tr>
<tr><td colspan="2">wt 10</td><td></td><td>110000000000
101000000000
100100000000
100010000000
100001000000
100000100000
100000010000
100000001000
100000000100
100000000010</td></tr>
<tr><td>000000000000
000000000000</td><td>000000000000
000000000000</td><td>111111111111
000000000000</td><td>000000000000
111111111111</td></tr>
</table>

Figure 6.2: Structure of Generator Matrix for $b = 2$

matrix into 2 row-blocks. These are $\{row_3, \ldots, row_{12}\}$, and $\{row_{13}, \ldots, row_{22}\}$.

Now consider $row_i(left)$, $3 \leq i \leq 22$. Based on the configuration in the first two rows, we may divide $row_i(left)$ into 2 column-blocks, each of length 12. Since $wt(row_i(right)) = 2$, and the minimum weight of C_{48} is 12, then $wt(row_i(left)) \geq 12 - 2 = 10$. Furthermore, if $wt(row_i(left)) > 14$ then $wt(row_i + row_1 + row_2) < 12$. Therefore $wt(row_i(left)) = 10$ or 14. However if $wt(row_i(left)) = 14$ then $wt(row_i(left) + row_1(left) + row_2(left)) = 10$. Therefore we may assume that $wt(row_i(left)) = 10$ for $3 \leq i \leq 22$.

After $row_3(left), \ldots, row_{22}(left)$ are filled in, one may verify that RM^- is generated by $row_1(left), \ldots, row_{22}(left)$.

Let us further consider $row_i(left)$, $3 \leq i \leq 22$. Let b_j be the number of column-blocks in $row_i(left)$ which have j ones. Clearly $0 \leq j \leq 10$. Since $wt(row_i(left)) = 10$, we have $\sum_{j=0}^{10} j \cdot b_j = 10$.

Suppose there is a column-block of j ones, $j \geq 7$ in $row_i(left)$. Then the other column-block of $row_i(left)$ must contain $10 - j$ ones. If the column-block of j ones is the first block of $row_i(left)$, then add row_i to row_1. Otherwise, add row_i to row_2. The result of the addition is a vector of weight $((12 - j) + (10 - j) + 2) = 24 - 2j$ which is ≤ 10 when $j \geq 7$. This is less than the minimum weight of the code, and therefore $b_j = 0$ for $j \geq 7$.

Now suppose that there is a column-block of j ones, $j \leq 3$ in $row_i(left)$. If the column-block of j ones is the first column-block of $row_i(left)$, then add row_i to row_2. Otherwise, add row_i to row_1. The result of the addition is a vector of weight $(j) + (12 - (10 - j)) + 2 = 4 + 2j$, which is ≤ 10 when $j \leq 3$. Therefore $b_j = 0$ for $j \leq 3$.

We now have two cases to consider. First consider the case in which there are 2 column-blocks of 5 ones in $row_i(left)$. By adding row_i to row_1, we obtain a vector of weight $(12 - 5) + (5) + 2 = 14$, which is not doubly-even. Therefore there must be one column-block of 6 ones, and one column-block of 4 ones, in $row_i(left)$.

This concludes our examination of the general organization of the generator matrix for the case in which RM has $b = 2$ rows. We shall now consider the remaining case.

namely $b = 3$.

6.3 Organization of Generator Matrix for $b = 3$

Let C_{48} be a (48, 24, 12) self-dual doubly-even code in which the generator matrix is organized as shown in Fig. 6.1, with $b = 3$.

In this case, the weight enumerator of RM is:

$$\begin{aligned} a_0 &= a_{24} = 1, \\ a_{12} &= \quad 6. \end{aligned}$$

From the MacWilliams Identity (Eq. (2.5)), we can obtain the weight enumerator of RM^-, namely:

$$\begin{aligned} a_0 &= a_{24} = 1, \\ a_2 &= a_{22} = 60, \\ a_4 &= a_{20} = 2\ 706, \\ a_6 &= a_{18} = 33\ 484, \\ a_8 &= a_{16} = 184\ 239, \\ a_{10} &= a_{14} = 489\ 720, \\ a_{12} &= \quad 676\ 732. \end{aligned}$$

We shall use the same notation as for $b = 2$. Namely, let M be the generator matrix of C_{48}. As for $b = 2$, two sections of M remain to be filled in. The first of these sections is the subspace generated by rows 4–21, restricted to their first 24 columns. Call these restricted rows $row_i(left)$, $4 \leq i \leq 21$. The second of these sections is the subspace generated by rows 4–21, restricted to their last 24 columns. Call these restricted rows $row_i(right)$, $4 \leq i \leq 21$.

As before, $row_i(left)$ and $row_i(right)$ must both be chosen from RM^-, for $4 \leq i \leq 21$. We would again like to complete the remainder of M using weight-10 words from RM^+ for $row_i(left)$, and weight-2 words from RM^+ for $row_i(right)$. However, as we shall see, it is not possible in this case. In fact, there must be one row composed of a weight-8 or weight-12 word from RM^- for $row_i(left)$, and a weight-4 word from

111111	111111	000000	000000	000000	000000	000000	000000
000000	000000	111111	111111	000000	000000	000000	000000
111111	000000	111111	000000	000000	000000	000000	000000
				110000			
				101000			
	wt	10		100100			
				100010			
				100001			
					110000		
					101000		
	wt	10			100100		
					100010		
						110000	
						101000	
	wt	10				100100	
						100010	
							110000
							101000
	wt	10					100100
							100010
	wt	8 or 12		100000	100000	100000	100000
000000	000000	000000	000000	111111	111111	000000	000000
000000	000000	000000	000000	000000	000000	111111	111111
000000	000000	000000	000000	111111	000000	111111	000000

Figure 6.3: Structure of Generator Matrix for $b = 3$

RM^- for $row_i(right)$. Furthermore we may assume that M has the form shown in Fig. 6.3, as explained below.

First consider $row_i(right)$, $4 \leq i \leq 21$. Based on the configuration in the last three rows, we may divide $row_i(right)$ into 4 column-blocks, each of length 6. There are 60 codewords of weight 2 in RM^-, and we shall attempt to use these to complete $row_i(right)$, $4 \leq i \leq 21$. If we place a single one in separate column-blocks of $row_i(right)$, then by adding a linear combination of the last 3 rows, we obtain a vector of weight 22, which is not doubly-even. Therefore we must place both ones in the same column-block.

There are $\binom{6}{2} = 15$ ways to place 2 ones in a column-block, and thus 15 codewords of weight 2 in RM^- may be obtained from each column-block. Ex-

actly 5 of these 15 codewords are linearly independent. All 15 codewords in the first column-block may be generated by $row_4(right), \ldots, row_8(right)$. Note that $row_4(right) + \ldots + row_8(right) + row_{22}(right)$ is a vector with exactly 6 ones, all in the second column-block. Therefore all 15 codewords in the second column-block may be generated by $row_9(right), \ldots, row_{12}(right)$ together with $row_4(right) + \ldots + row_{12}(right) + row_{22}(right)$. Similarly, all 15 codewords in the third column-block may be generated by $row_{13}(right), \ldots, row_{16}(right)$ and $row_4(right) + \cdots + row_8(right) + row_{13}(right) + \cdots + row_{16}(right) + row_{24}(right)$. Finally, all 15 codewords in the fourth column-block may be generated by $row_{17}(right), \ldots, row_{20}(right)$ and $row_4(right) + \cdots + row_{12}(right) + row_{17}(right) + \cdots + row_{20}(right) + row_{23}(right) + row_{24}(right)$. All of the required 60 codewords of weight 2 in RM^- have thus been generated. Therefore $row_{21}(right)$ cannot have weight 2, and so we choose the next most restrictive possibility, namely weight 4.

Observe that we have divided the remaining incomplete rows of the generator matrix into 5 sections. There are 4 row-blocks:

$\{row_4, \ldots, row_8\}$.

$\{row_9, \ldots, row_{12}\}$.

$\{row_{13}, \ldots, row_{16}\}$.

$\{row_{17}, \ldots, row_{20}\}$.

and a single row, $\{row_{21}\}$.

Now consider $row_i(left)$, $4 \leq i \leq 21$. Based on the configuration in the first three rows, we may divide $row_i(left)$ into 4 column-blocks, each of length 6.

Since $wt(row_i(right)) = 2$ for $4 \leq i \leq 20$, and the minimum weight of C_{48} is 12, then $wt(row_i(left)) \geq 12 - 2 = 10$ for $4 \leq i \leq 20$. Furthermore, if $wt(row_i(left)) > 14$ then $wt(row_i + row_1 + row_2) < 12$. Therefore $wt(row_i(left)) = 10$ or 14. However if $wt(row_i(left)) = 14$ then $wt(row_i(left) + row_1(left) + row_2(left)) = 10$. Therefore we may assume that $wt(row_i(left)) = 10$ for $4 \leq i \leq 20$.

Similarly, since $wt(row_{21}(right) = 4$, and the minimum weight of C_{48} is 12, then $wt(row_{21}(left)) \geq 12 - 4 = 8$. Furthermore, if $wt(row_{21}(left)) > 16$ then $wt(row_{21} + row_1 + row_2) < 12$. Therefore we may assume that $wt(row_{21}(left)) = 8$, 12 or 16.

Observe that if $wt(row_{21}(left)) = 16$ then $wt((row_{21} + row_1 + row_2)left) = 8$ and therefore we may consider these two cases to be equivalent. Therefore we only need to consider the cases when $wt(row_{21}(left)) = 8$ or 12.

Now let b_j be the number of blocks in $row_i(left)$ which have j ones. Clearly $0 \leq j \leq 6$.

First let us show that either all blocks of $row_i(left)$ have an even number of ones, or all blocks of $row_i(left)$ have an odd number of ones. Clearly it is not possible for there to be an odd number of blocks each containing an odd number of ones, for if there were, then row_i would have an odd weight.

Therefore suppose that the blocks of $row_i(left)$ contain e_1, e_2, o_1 and o_2 ones, where e_1 and e_2 are even, and o_1 and o_2 are odd. Then one may always match up one even block, together with one odd block, against one of the linear combinations of the first 3 rows, creating a vector which is not doubly-even. For example, suppose that the first block of $row_i(left)$ contains e_1 ones, the second block contains e_2 ones, the third block contains o_1 ones, and the fourth block contains o_2 ones. The total weight of row_i is $e_1 + e_2 + o_1 + o_2 + 2$. Adding row_i and row_3, we obtain a vector of weight $(6-e_1)+e_2+(6-o_1)+o_2+2$. Consider the difference between the weight of row_i and the weight of row_i+row_3. Since e_1 is even, $e_1-(6-e_1) \equiv 2 \pmod 4$. Since o_1 is odd, $o_1-(6-o_1) \equiv 0 \pmod 4$. Therefore $(e_1+e_2+o_1+o_2)-((6-e_1)+e_2+(6-o_1)+o_2) \equiv 2 \pmod 4$. Therefore row_i and $row_i + row_3$ cannot both be doubly-even. This is the difference between the weight of row_i and the weight of $row_i + row_3$, and therefore either all blocks of row_i are odd, or all blocks of row_i are even.

Since we have 4 blocks, we have:

$$b_0 + b_1 + b_2 + b_3 + b_4 + b_5 + b_6 = 4.$$

First we consider the case when $row_i(left)$ has weight 10. In this case, we have $\sum_{j=0}^{6} j \cdot b_j = 10$. If $b_6 = 1$ or $b_5 > 0$, then the block containing the largest number of ones has 5 or 6 ones and the block which contains the next-largest number of ones has at least 2 ones. We can always match up a block of 5 or 6 ones, together with the block which contains the next-largest number of ones, against 2 blocks of 6 ones from a linear combination of the first 3 rows, creating a vector of low weight. For example,

suppose the first block of $row_i(left)$ contains 6 ones, and the second block contains x ones. $x > 0$. Then the last 2 blocks contain $10 - 6 - x$ ones. Adding row_i and row_1, we obtain a vector of weight $2 + (6 - 6) + (6 - x) + 10 - 6 - x = 10 - 2x < 10$. This is less than the minimum weight of the code.

In the case $b_4 = 2$, one may similarly match up these 2 blocks of 4 ones against 2 blocks of 6 ones from a linear combination of the first 3 rows, creating a vector of low weight.

Therefore we have $b_6 = 0$, $b_5 = 0$, and $b_4 \leq 1$. Furthermore, either all blocks have an odd number of ones, or all blocks have an even number of ones. Therefore the only possibilities for the case when $row_i(left)$ has weight 10 are: $\{b_0 = 0, b_1 = 0, b_2 = 3, b_3 = 0, b_4 = 1, b_5 = 0, b_6 = 0\}$ and $\{b_0 = 0, b_1 = 1, b_2 = 0, b_3 = 3, b_4 = 0, b_5 = 0, b_6 = 0\}$.

Now let us consider the case when $row_i(left)$ has weight 12. In this case, we have $\sum_{j=0}^{6} j \cdot b_j = 12$. If $b_6 > 1$ or $b_5 > 1$ then we can always match up the blocks of 5 or 6 ones against 2 blocks of 6 ones from a linear combination of the first 3 rows, creating a vector of low weight. If $b_6 = 1$ or $b_5 = 1$ then the block containing the largest number of ones has 5 or 6 ones, and the block containing the next-largest number of ones has at least 2 ones. We can always match up a block of 5 or 6 ones, together with the block which contains the next-largest number of ones, against 2 blocks of 6 ones from a linear combination of the first 3 rows, creating a vector v such that $v(left)$ has weight 8. Therefore we can consider this to be part of the case in which $row_i(left)$ has weight 8. Therefore if $row_i(left)$ has weight 12, then $b_6 = 0$ and $b_5 = 0$.

If $b_4 > 0$ then since either all blocks contain an even number of ones, or all blocks contain an odd number of ones, there are at least 2 blocks of 4 ones. We can match up 2 of these blocks of 4 ones against 2 blocks of 6 ones from a linear combination of the first 3 rows, again creating a vector v such that $v(left)$ has weight 8. Therefore if $row_i(left)$ has weight 12, $b_4 = 0$.

Hence the only possibility when $row_i(left)$ has weight 12 is $\{b_0 = 0, b_1 = 0, b_2 = 0, b_3 = 4, b_4 = 0, b_5 = 0, b_6 = 0\}$.

Finally let us consider the case when $row_i(left)$ has weight 8. In this case, we have $\sum_{j=0}^{6} j \cdot b_j = 8$. If we have only two blocks containing ones, then we can match up these two non-zero blocks against two blocks of 6 ones from a linear combination of the first 3 rows, creating a vector of low weight. As explained earlier, we can also eliminate all possibilities in which some blocks have an even number of ones, and some blocks have an odd number of ones. Therefore when $row_i(left)$ has weight 8, the only possibilities are $\{b_0 = 0, b_1 = 3, b_2 = 0, b_3 = 0, b_4 = 0, b_5 = 1, b_6 = 0\}$, $\{b_0 = 1, b_1 = 0, b_2 = 2, b_3 = 0, b_4 = 1, b_5 = 0, b_6 = 0\}$, $\{b_0 = 0, b_1 = 0, b_2 = 4, b_3 = 0, b_4 = 0, b_5 = 0, b_6 = 0\}$ and $\{b_0 = 0, b_1 = 2, b_2 = 0, b_3 = 2, b_4 = 0, b_5 = 0, b_6 = 0\}$.

We use these distributions of ones on the left-hand side of the generator matrix in the next chapter, to estimate the search size.

6.3.1 Note

In a private communication from Christine Bachoc [5], it was shown that one can use a weight 12 codeword for row_{21}, in which $row_{21}(left)$ has weight 4 and $row_{21}(right)$ has weight 8. In this case, $row_{21}(right)$ has one column-block containing 5 ones, and 3 column-blocks each containing a single one.

6.4 Summary

One may partition the search for a $(48, 24, 12)$ self-dual doubly-even code into three subcases, based on the maximum dimension b of a $(24, b, 12)$ subcode. The results of a search assuming $b = 4$ are reported in [19, 20]. In this search, only codes isomorphic to the known code were found.

For the case $b = 2$, we may assume that the generator matrix for such a code has the form shown in Fig. 6.2, with one column-block of 6 ones and one column-block of 4 ones in $row_i(left)$, for $3 \leq i \leq 22$.

For the case $b = 3$, we may assume that the generator matrix for such a code has the form shown in Fig. 6.3. In $row_i(left)$, $4 \leq i \leq 20$, there must be either 3 column-blocks of 2 ones and 1 column-block of 4 ones, or 3 column-blocks of 3 ones and 1 column-block of a single one. If $wt(row21(left)) = 12$ then $row_{21}(left)$ must

have 4 column-blocks of 3 ones. If $wt(row21(left) = 8$ then $row_{21}(left)$ must have 3 column-blocks of a single one and one column-block of 5 ones, or 2 column-blocks of 2 ones, 1 column block of 4 ones and 1 column-block of 0 ones, or 4 column-blocks of 2 ones, or 2 column-blocks of 3 ones and 2 column-blocks of a single one.

With this information, we proceed to the next chapter, in which we develop a search method and estimate the search size for the two remaining cases, $b = 3$ and $b = 2$.

Chapter 7

Search Method and Estimates in the Search for a $(48, 24, 12)$ Code

We shall now continue to look at (48, 24, 12) self-dual, doubly-even codes. Two cases remain. In the first remaining case, there is a (24, 3, 12) subcode, and we assume that C_{48} has a generator matrix of the form given in Fig. 6.3. In the final remaining case, there is a (24, 2, 12) subcode, and we assume that C_{48} has a generator matrix of the form given in Fig. 6.2. In both cases, we shall refer to the specified subcode as RM.

A search assuming either case would consist of attempting to complete a generator matrix of the specified form: we would attempt to complete this generator matrix row by row, starting with row_{b-1} and ending with row_{24-b}, where b is the dimension of the subcode RM. The first b rows and last b rows of the generator matrix are already known.

As we proceed, we shall monitor the search to determine its size. For each row in turn, we must look at:

- the number of candidates for the row, and
- the number of survivors for the row.

A candidate is a survivor if it is compatible with all previously completed rows. A candidate is compatible with all previous rows if:

- it is linearly-independent from all previous rows, and

- all linear combinations of the candidate and previous rows are themselves valid codewords.

While it is still manageable, we shall find all non-isomorphic survivors for the current row as we proceed row by row. For later rows, it is necessary to first obtain estimates of the number of candidates and number of survivors, before deciding how best to proceed.

7.1 Complete Search

As shown in Chapter 6, we know the types of vectors which can be used to complete each row. For example, when $b = 3$ we know that for row 4, either there is 1 block of 4 ones and 3 blocks of 2 ones, or there are 3 blocks of 3 ones and 1 block of a single one. When $b = 2$ we know that every remaining row has 1 block of 4 ones and 1 block of 6 ones.

Our first step, therefore, is to create a list of candidates, consisting of all vectors of the appropriate types. Note that each of these vectors is of length 24; as we consider each row, we must concatenate a length-24 vector with the appropriate "right-hand side" for that row as shown in either Fig. 6.2 or Fig. 6.3.

This initial list is the list of candidates for row_{b-1}. We examine each candidate in turn to determine if it is compatible with all previous rows, and if so we add it to the list of survivors for row_{b-1}. We then check for isomorphisms among the subcodes generated by the completed rows. Since isomorphic codes have the same properties, we are only interested in non-isomorphic codes.

Since the same set of types of vectors must be used to fill both row_{b-1} and row_{b-2}, and since row_{b-2} must be compatible with all of the rows with which row_{b-1} is compatible, our list of candidates for row_{b-2} consists of all survivors from row_{b-1}, with of course the appropriate right-hand side $row_{b-2}(right)$. Our list of survivors for row_{b-2} consists of all those candidates which are also compatible with row_{b-1}.

For the same reasons, the above is true for each subsequent row up to the end of the first row-block. Namely, the list of candidates for row_i is the list of survivors

for row_{i-1}, and the list of survivors for row_i is all those candidates which are also compatible with row_{i-1}.

We shall now consider a method to find all possible non-isomorphic completions of a set of rows of the generator matrix. The rows in question must be consecutive rows within the same row-block. The algorithm we employ is as follows:

CompleteSearch(*StartRow*, *EndRow*)
for i = *StartRow* to *EndRow* *do*
begin
    for each possible non-isomorphic completion to row_{i-1} *do*
    begin
        for each vector $v(left)$ in the list of survivors for row_{i-1} *do*
        begin
            concatenate $v(left)$ with $row_i(right)$, creating v:
            if v is compatible with $row_1, \ldots, row_{i-1}$ and $row_{25-b}, \ldots, row_{24}$
            then add v to the list of survivors for row_i:
            if the code C generated by v, $row_1, \ldots, row_{i-1}$ and $row_{25-b}, \ldots, row_{24}$ is
                non-isomorphic to all previously seen codes
            then C is a non-isomorphic completion to row_i:
        end
    end
end

7.1.1 Complete Search to Row 8 for $b = 3$

The results of employing the above method, for the case $b = 3$, are shown in Table 7.1. From this table, we can see that the number of non-isomorphic codes increases very rapidly to 52 161 at row_8. It was at this point that we started to encounter difficulties.

Specifically, the running program required more than 165 megabytes of virtual memory, which was the limit on the computer being used. This was due to the large number of non-isomorphic codes generated, each of which require a certain amount

Row	Non-Isomorphic Codes
4	2
5	4
6	35
7	877
8	52 161

Table 7.1: Number of Non-Isomorphic Codes by Row for $b = 3$

of storage space.

Therefore we needed to break up the work for row_8 into several runs. Note that codes with different weight enumerators are non-isomorphic. After finding all survivors for row_8, we compute the weight enumerator of each code generated, then divide the work so that codes in different runs have different weight enumerators.

We encountered 12 different weight enumerators. Each weight enumerator had a different number of minimum-weight codewords. In addition, we know that at this point, the minimum-weight codewords generate the code as the basis consists entirely of minimum-weight codewords. Therefore while checking for isomorphisms among codes, we save additional space by comparing only the minimum-weight codewords from each code, rather than the entire code.

The results of this division of labour are shown in Table 7.2. In this table, we show the number of codes, and number of non-isomorphic codes, which occur for each weight enumerator. Each weight enumerator is identified by its number of minimum-weight codewords. There are several cases which occur only rarely. Since there are only a small number of codes to compare for each of these cases, we decided to check for isomorphisms among this entire group as a whole rather than case by case.

7.1.2 Complete Search to Row 7 for $b = 2$

The results of employing the same method, for the case $b = 2$, are shown in Table 7.3. As before, the number of non-isomorphic codes increases rapidly, to 20 875 at row_7. In this case, we do not attempt to push the complete search further by dividing these cases based on weight enumerator, as the number of non-isomorphic codes at row_7 is already very large.

Min-wt Codewords	Codes	Non-Isomorphic Codes
89	291 401	25 080
93	167 051	16 329
88	108 487	4 107
92	76 064	3 824
97	19 361	1 918
96	8 649	.
87	5 037	.
91	2 743	.
101	504	.
95	188	.
100	158	.
85	44	.
		903
Total	679 687	52 161

Table 7.2: Count of Codes at Row 8 for $b = 3$

Row	Non-Isomorphic Codes
3	1
4	3
5	17
6	269
7	20 875

Table 7.3: Number of Non-Isomorphic Codes by Row for $b = 2$

7.2 Estimation of Search Size

The computational difficulties encountered in the above searches illustrate the necessity of monitoring the amount of computer memory and time required as each new row is completed. If we do not do this then we run the risk of exceeding the physical limits of the computer(s) being used and taking an unreasonable amount of time. Also if we monitor carefully then we can decide in which areas we can make the most savings, and try to make improvements in those specific areas. We discuss estimates more fully in Chapter 9.

7.2.1 Starting Information for $b = 3$

Our work has now been divided into 6 cases, based upon the weight enumerator of the code up to row_8. The first 4 of these cases are relatively large. There are 25 080 non-isomorphic codes with 89 minimum-weight codewords, 16 329 non-isomorphic codes with 93 minimum-weight codewords, 4 107 non-isomorphic codes with 88 minimum-weight codewords and 3 824 non-isomorphic codes with 92 minimum-weight codewords. The remaining cases are relatively small. There are 1 918 non-isomorphic codes with 97 minimum-weight codewords and 903 non-isomorphic codes with 96, 87, 91, 101, 95, 100 or 85 minimum-weight codewords.

For estimation purposes, we use 3 codes from each of the large cases and 1 code from each of the other cases. We use codes from each case because codes from different cases may behave differently. We use more than one code from each of the larger cases because of the large number of codes from each of these cases, making accuracy for these cases more important.

The basic method of estimation used is that described in [19]. We use the method of attempting to complete the matrix which is described earlier in this chapter, except that we do not do full isomorphism testing as it is now too expensive.

The estimation is performed on a row-by-row basis. However, the candidate list for row_{i-1} is not necessarily the survivor list for row_i. Recall that the generator matrix as shown in Fig. 6.3 is divided into 5 sections:
$\{row_4, \ldots, row_8\}$,

$\{row_9, \ldots, row_{12}\}$.

$\{row_{13}, \ldots, row_{16}\}$.

$\{row_{17}, \ldots, row_{20}\}$.

and $\{row_{21}\}$.

We call the first four of these the "row-blocks" of the generator matrix. There are also "column-blocks", each 6 columns wide. The column-blocks of the "left-hand side" of the matrix are:

$\{col_1, \ldots, col_6\}$.

$\{col_7, \ldots, col_{12}\}$.

$\{col_{13}, \ldots, col_{18}\}$. and

$\{col_{19}, \ldots, col_{21}\}$.

On the "right-hand side" of the matrix, the column-blocks are:

$\{col_{25}, \ldots, col_{30}\}$.

$\{col_{31}, \ldots, col_{36}\}$.

$\{col_{37}, \ldots, col_{42}\}$. and

$\{col_{43}, \ldots, col_{48}\}$.

Since row_4 and row_9 must both be vectors of the same two types, and since row_9 must be compatible with all of the rows with which row_4 is compatible, our list of candidates for row_9 consists of all survivors from row_4, with the appropriate right-hand side $row_9(right)$.

For the same reasons, the candidate list for row_i, $i = 13$ or 17, is the survivor list for row_{i-4}. Within each row-block, the candidate list for row_i, where row_i is *not* the first row of the row-block, is the survivor list for row_{i-1}. For the last section, consisting only of row_{21}, we must create a special candidate list since that row is different from all others.

7.2.2 Starting Information for $b = 2$

Recall that there are 20 875 non-isomorphic codes complete to row_7. We take 10 of these partially-completed matrices for estimation purposes. We use the same method of estimation as for $b = 3$.

Recall that the generator matrix as shown in Fig. 6.2 is divided into two row-blocks, namely:

$\{row_3, \ldots, row_{12}\}$, and

$\{row_{13}, \ldots, row_{22}\}$.

There are also 4 column-blocks, each 12 columns wide. The column-blocks of the left-hand side of the matrix are:

$\{col_1, \ldots, col_{12}\}$, and

$\{col_{13}, \ldots, col_{24}\}$.

The column-blocks of the right-hand side of the matrix are:

$\{col_{25}, \ldots, col_{36}\}$, and

$\{col_{37}, \ldots, col_{48}\}$.

Since row_3 and row_{13} must both be vectors of the same two types, and since row_{13} must be compatible with all of the rows with which row_3 is compatible, our list of candidates for row_{13} consists of all survivors from row_4, with the appropriate right-hand side $row_{13}(right)$. Within each row-block, the candidate list for row_i, where row_i is *not* the first row of the row-block, is the survivor list for row_{i-1}.

7.2.3 Estimation Method

Suppose we find s_i survivors for row_i. Instead of attempting to complete row_{i-1} for each of those survivors, we use only a certain number, say t_i. Suppose from these chosen survivors for row_i, we find s_{i-1} survivors for row_{i-1}. Then the expected total number of survivors for row_{i-1} is $s_{i-1} \cdot s_i/t_i$.

For example, suppose we take one of the initial 52 161 starting matrices for $b = 3$, in which $row_1, \ldots, row_8$ and $row_{22}, \ldots, row_{24}$ have already been completed. Further suppose that given this starting matrix, we have found 1000 survivors for row_9. If we take 3 of these, and find a total of 150 survivors for row_{10}, then the expected total number of survivors for row_{10}, for the given starting matrix, is $150 \cdot 1000/3 = 50000$.

For more information on this method, see [19].

7.2.4 Number of Survivors Chosen for each Row

In the most simple form of this method of estimation, we choose only one survivor for each row of the generator matrix, counting the number of survivors for the next row if our chosen survivor were placed in the current row. For a more accurate estimate, we may choose more survivors at each row. For example, we may choose 3 or 5 survivors at each row; this generally gives a fairly accurate estimate.

However, the size of the search varies according to a certain pattern. The largest part of the search, which we call the "bulge", is generally around the mid-point, that is, when we have filled approximately half of the remaining rows. The size of the search then falls off because as more rows are filled, greater restrictions are placed on the candidates. In addition, within each row-block, there is a "local bulge". For the first row of the row-block, there is almost no restriction from the intersection pattern within the appropriate column-block. As more rows are filled within the row-block, the intersection pattern within the column-block places a greater restriction on the candidates. The local bulge occurs around the middle of each row-block. There is then a large drop at the end of the row-block. Early in the row-block, it is almost guaranteed that we will be able to continue to fill the next row, no matter which candidates we choose for the current row. However, this is not true later in the row-block. Given all the previous rows of the row-block, there are often no survivors at all for the last row of a row-block.

If we choose exactly 3 survivors at each level, then the probability of reaching the last row of a row-block is rather low. For example, consider the case $b = 3$. In this case, we would have chosen only 3 of a large number of possible third rows. Perhaps more importantly, if there are no possible fourth rows (given our choices of third rows) then it is impossible to continue the search into the next row-block.

For these reasons, we may obtain more accurate estimates by choosing a low number of survivors for each row early in the row-block, and a high number at the local bulge around the middle of the row-block. For example, for the case $b = 3$, we may choose 3 survivors for row_9, 4 survivors for row_{10}, 50 for row_{11}, and 3 for row_{12}. Therefore there are $3 \cdot 4 \cdot 50 = 600$ possibilities in which we *may* be able to reach

row_{12} and the next row-block. Of course if there are $s_{11} < 50$ survivors for row_{11}, then we can only choose a maximum of s_{11} survivors at row_{11}.

7.3 Refinements to the Search

The method described earlier in this chapter is clearly too expensive. As previously mentioned, we encountered problems even as early as row_8 for the case $b = 3$ and row_7 for the case $b = 2$. As we eliminate full isomorphism testing, the search becomes much larger. We shall now consider several refinements which we can make to the search.

7.3.1 Ordering of Rows

Within each row-block remaining to be filled, suppose that v_1 is an acceptable vector to fill row_i, where row_i is *not* the last row of a row-block. Further suppose that v_2 is an acceptable vector to fill row_{i-1}. Then since v_1 and v_2 must obey the same constraints, v_1 must also be an acceptable vector to fill row_{i-1} (with the appropriate right-hand side $row_{i-1}(right)$) and v_2 must also be an acceptable vector to fill row_i (with the appropriate right-hand side $row_i(right)$). These two cases are isomorphic. Therefore when looking through the candidate list for row_{i-1}, we only need to begin looking *after* the point at which row_i occurs in the list. It is of course necessary that the candidate list remain ordered.

7.3.2 Choosing a Block Pattern

Now consider the pattern in each column-block on the right-hand side of the matrix. For example, for the case $b = 3$, within each of the remaining row-blocks, there is a column-block on the right-hand side of the matrix with the following pattern:

row_i 110000
row_{i-1} 101000
row_{i-2} 100100
row_{i-3} 100010.

where $i = 9$, 13 or 17.

Let w_1 be the codeword obtained by adding row_i, row_{i-1} and row_{i-2}. This vector

has 4 ones in that column-block, where previously all vectors had 2 ones in that column-block. Let w_2 be the codeword obtained by adding w, $row_4, \ldots, row_8$, and either row_{22} (for $i = 9$), row_{24} (for $i = 13$), or $row_{23} + row_{24}$ (for $i = 17$). This vector has 2 ones in that column-block. This is in fact the first time, for that particular row-block, that we have been able to "reduce" the weight of a vector in a column-block. This has the effect of requiring w_2 to have a higher weight on its left-hand side than w_1. It is therefore easier for w_2 to have a total weight which is lower than the minimum-weight of the code, resulting in the rejection of the current candidate for row_{i-2}. This is a benefit seen starting only at row_{i-2}.

Suppose within each of these row-blocks, we replace row_{i-1} with $row_{i-1} + row_{i-2}$, so that the pattern in the appropriate block of the right-hand side becomes:

row_i	110000
$row_{i-1} + row_{i-2}$	001100
row_{i-2}	100100
row_{i-3}	100010.

Now the benefit of row_{i-2} is seen 1 row earlier, reducing the "local bulge" within each row-block.

Note that if we use this modification, then the list of candidates for row_{i-2} is now the list of survivors for row_i. We cannot use the list of survivors for the new row_{i-1} because of the intersection pattern in the column-block. This in turn implies that we can no longer insist that row_{i-2} appears later in the candidate list than the new row_{i-1}: we can only insist that row_{i-2} appears later in the candidate list than row_i. For this reason we chose not to use this particular method during our search.

7.3.3 Rejecting Previous Cases

Recall that we have divided the search for a (48, 24, 12) self-dual doubly-even code into 3 large cases, based upon the dimension b of a subcode. The results of the first case, namely $b = 4$, are already known: this case assumes the existence of a (24, 4, 12) subcode.

For the case $b = 3$, we assume the existence of a (24, 3, 12) subcode but no (24, 4, 12) subcode. Therefore if, in a search assuming this case, we find a code with a (24, 4, 12) subcode, then we can reject this code as it is part of the case already seen.

Similarly, for the case $b = 2$, we assume the existence of a $(24, 2, 12)$ subcode but no $(24, 3, 12)$ subcode. Therefore if, in a search assuming this case, we find a code with a $(24, 3, 12)$ subcode, then we can reject this code.

We can test for this possibility as follows. If a $(24, 4, 12)$ subcode exists, then it has 14 codewords of weight 12. A $(24, 3, 12)$ subcode has only 6 codewords of weight 12. A $(24, 2, 12)$ subcode has only 2 codewords of weight 12.

As we check the candidates for each new row in turn, we find all vectors which are linear combinations of that candidate and previously-completed rows. We are particularly interested in vectors with weight 12 or 24. For each vector w with weight 24, we count the number of weight-12 vectors which intersect w in 12 places. If there are more than 2 such vectors, then the code generated by the current candidate, together with the previously-completed rows, has a $(24, 3, 12)$ subcode. If there are more than 6 such vectors, then the code has a $(24, 4, 12)$ subcode. Therefore in the case $b = 3$, we can reject the candidate if there are more than 6 such vectors; in the case $b = 2$, we can reject the candidate if there are more than 2 such vectors.

This is a fairly expensive test and may not be worthwhile at every row, particularly for the case $b = 3$. In this case, a greater number of intersecting vectors must be found and therefore an "illegal" subcode is less likely. However as more rows are completed, the likelihood increases that an illegal subcode exists. In general we find that in the case $b = 2$, this test is useful during most of the search, but for the case $b = 3$, it may be most useful later in the search.

For the case $b = 3$, a few of the 52 161 codes completed to row_8 already contain a $(24, 4, 12)$ subcode. The updated count of codes complete to row_8, after checking for such a subcode, is shown in Table 7.4.

7.3.4 Selecting Particular Row Types

Recall that we already insist that for the case $b = 3$, the generator matrix has the form given in Fig. 6.3, and for the case $b = 2$, the generator matrix has the form given in Fig. 6.2. We also know that only certain types of vectors may be used to fill the remaining rows, as discussed in Chapter 6.

Min-wt Codewords	Non-Isomorphic Codes	Non-Isomorphic Codes without (24, 4, 12) Subcode
89	25 080	25 080
93	16 329	16 329
88	4 107	4 107
92	3 824	3 270
97	1 918	1 918
others	903	660
Total	52 161	51 364

Table 7.4: Count of Codes at Row 8 with no (24, 4, 12) Subcode

Upon further examination of the problem, we may be able to further restrict the choice of types of vectors for each row. This will of course further reduce the number of possibilities to consider for each row, and thereby reduce the overall size of the search.

Row Types for $b = 3$

Recall that each of $row_4, \ldots, row_{20}$ must be one of two possible types: within the first 24 columns, either the row has one column-block containing 4 ones, and 3 containing 2 ones, or the row has 3 column-blocks containing 3 ones, and 1 containing a single one. We shall call the first of these possibilities *type* $4 \cdot 2^3$, and the second of these possibilities *type* $3^3 \cdot 1$.

First consider rows of type $4 \cdot 2^3$. Suppose we have a row with 4 ones in the first column-block. Then we can add that row to row_1 and create another vector of type $4 \cdot 2^3$, except with 4 ones in the second column-block. In general we may move the block of 4 ones to any column-block by adding an appropriate linear combination of the first 3 rows of the generator matrix. Therefore we can insist that all rows of type $4 \cdot 2^3$ have 4 ones in the *first* column-block.

Now consider rows of type $3^3 \cdot 1$. Suppose we have a row with 3 ones in each of the first two column-blocks. By adding this row to row_1, we take the complement of the entries in the first two column-blocks. In general we may take complements of the entries in any two column-blocks simultaneously by adding an appropriate linear combination of the first 3 rows of the generator matrix. Therefore we can insist that

all rows of type $3^3 \cdot 1$ have ones in the *first* column of the first two column-blocks containing 3 ones.

Recall that there are 3 row-blocks remaining to be filled, each with 4 rows. Also recall that we have created a candidate list containing all possible vectors of type $4 \cdot 2^3$ and type $3^3 \cdot 1$. In fact we generated this candidate list in a particular order, with all vectors of type $4 \cdot 2^3$ appearing before those of type $3^3 \cdot 1$. Suppose the first row of the row-block is of type $3^3 \cdot 1$. Then the second row of the row-block must also be of type $3^3 \cdot 1$, since we insist that the second row appears later in the candidate list than the first row. Adding these two rows, possibly together with a linear combination of the first 3 rows, we obtain a vector of type $4 \cdot 2^3$. This vector appears earlier in the candidate list than either of the vectors from those two rows, and could be chosen to fill either of the rows.

For example, suppose that we have:

$$row_9(left) = 111000\ 111000\ 111000\ 100000$$
$$row_{10}(left) = 100110\ 100110\ 100110\ 010000.$$

Then:

$$row_9(left) + row_{10}(left) = 011110\ 011110\ 011110\ 110000.$$

Adding $row_1(left)$:

$$100001\ 100001\ 011110\ 110000.$$

This last vector appears earlier in the candidate list than either $row_9(left)$ or $row_{10}(left)$.

We can therefore insist that the first row of any row-block is of type $4 \cdot 2^3$.

For each of these row-blocks, there are 15 possible vectors of weight 12, with a weight of 10 on the left-hand side and a weight of 2 on the right-hand side, in the appropriate column-block.

Note that by adding two vectors of type $3^3 \cdot 1$, and possibly a linear combination of the first 3 rows of the generator matrix, we obtain a vector of type $4 \cdot 2^3$. Suppose 3 rows of the row-block, say row_a, row_b and row_c, are of type $3^3 \cdot 1$. Let v be the

vector of type $4 \cdot 2^3$ obtained by adding row_a, row_b, and possibly a linear combination of the first 3 rows of the generator matrix. Let w be the vector of type $4 \cdot 2^3$ obtained by adding row_a, row_c, and possibly a linear combination of the first 3 rows of the generator matrix. Observe that v and w intersect in exactly one column on the right-hand side of the generator matrix, and hence they could be used as the first two rows of the row-block. Therefore we can insist that *both* of the first two rows of each row-block are of type $4 \cdot 2^3$.

Row Types for $b = 2$

Recall that for this case, there is only one possible type of vector we use to fill each of $row_3, \ldots, row_{22}$. In particular, within the first 24 columns of each of these rows, we must have 1 column-block containing 6 ones and 1 column-block containing 4 ones. Let us call a vector containing 6 ones in the first column-block and 4 ones in the second column-block a vector of *type* $6 \cdot 4$. Similarly, let us call a vector containing 4 ones in the first column-block and 6 ones in the second column-block a vector of *type* $4 \cdot 6$.

Observe that we can exchange the contents of the column-blocks of the left-hand side of the generator matrix by permuting these column-blocks and then permuting the first two rows of the matrix. Therefore, for any particular row, say row_i, $3 \leq i \leq 22$, if row_i is of type $4 \cdot 6$, then after this process, row_i will be of type $6 \cdot 4$.

Recall that there are 2 row-blocks remaining to be filled, namely:
$\{row_3, \ldots, row_{12}\}$ and
$\{row_{13}, \ldots, row_{22}\}$.
If the first row-block contains j rows of type $6 \cdot 4$, then after this process, it will contain j rows of type $4 \cdot 6$. Therefore since there are 10 rows in each row-block, we can insist that the first row-block contains at least 5 rows of type $6 \cdot 4$. In particular, we can insist that the *first* 5 rows of the first row-block are all of type $6 \cdot 4$.

Now consider rows of type $6 \cdot 4$. By adding any such row to row_1, we have a vector of type $6 \cdot 4$, except with the complement of its original entries in the first column-block. Similarly, by adding any row of type $4 \cdot 6$ to row_2, we obtain a vector of type

$4 \cdot 6$, except with the complement of its original entries in the second column-block. Therefore we can insist that for row_i, $3 \leq i \leq 22$, there must be a one in the *first* column in the block containing 6 ones.

7.3.5 Partial Isomorphism Testing

After a certain point in the search, it is too computationally expensive to perform full isomorphism testing. Therefore we attempt to detect some, but not all, isomorphic solutions using a cheaper test. We call this method *Poor Woman's Isomorphism Testing.*

This test may be used for both remaining cases. In reality, however, we found during the estimation process that the test was of little use for the case $b = 2$, although it may become more useful as the search progresses into the second row-block for this case. Therefore since we use this test primarily for the case $b = 3$, we shall describe its use for this case.

Recall that each of the remaining row-blocks has the following pattern in a column-block on the right-hand side of the generator matrix:

row_i	110000
row_{i-1}	101000
row_{i-2}	100100
row_{i-3}	100010
special	100001.

where the "special" row is obtained by adding $row_i, \ldots row_{i-3}, row_4, \ldots row_8$ and one of the linear combinations of the last 3 rows of the generator matrix.

As previously mentioned, we may insist that within each row-block, each row other than row_i itself appears later in the candidate list than the previous row. This eliminates some isomorphic possibilities. In particular, at row_{i-1} we save a factor of 2 by eliminating half of the possibilities, at row_{i-2} a further factor of 3, and at row_{i-3} a further factor of 4. We therefore obtain a full saving of a factor of 4! within each row-block, which is the maximum that may be obtained by considering the rows alone.

There are, however, 6 columns within each row-block. If we can use these, we can save a further factor of $5 \cdot 6 = 30$ within each row-block. To do this, we shall use

a method which we shall refer to as "Poor woman's isomorphism testing", since it is cheaper computationally than full isomorphism testing, but does not eliminate as many possibilities.

For an example of this method, suppose that we are testing a candidate v for row_{11} of the generator matrix. Suppose that we have:

row_9	=	111100	110000	110000	110000	...	110000 ...
row_{10}	=	011101	001100	011000	011000	...	101000 ...and
v	=	001111	101000	101000	100100	...	100100 ...

Note:

$v + row_9$	=	110011	011000	011000	010100	...	010100 ...

Clearly v does not appear earlier in the candidate list than either row_9 or row_{10}, and therefore could not be rejected as a possible row_{11} on this basis. However, $v+row_9$ appears earlier in the candidate list than row_{10}, and thus with the appropriate column permutations on the right-hand side, could be used as row_{10}. Furthermore, we would have already seen this isomorphic case. Therefore we can reject v as a candidate for row_{11}.

The algorithm we employ when testing a candidate v for row_{11} is as follows:

1. find all linear combinations of v and previously completed rows which have weight 10 on the left-hand side and weight 2 in the correct block of the right-hand side.

2. if the smallest of these, say w_9, is not row_9, then reject v.

3. of the linear combinations from step 1, find the smallest, say w_{10}, which also intersects row_9 in 1 column on the right-hand side.

4. if w_{10} is not row_{10}, then reject v.

5. of the linear combinations from step 3, find the smallest, say w_{11}, which intersects both row_9 and row_{10} in the *same* column of the right-hand side.

6. if w_{11} is not v, then reject v.

In general, when we are testing a candidate v for row_j of a row-block, we use the following algorithm:

find all linear combinations of v and previously completed rows
    which have weight 10 on the left-hand side and weight 2 in the
    correct block of the right-hand side;
for each previous row row_k of the row-block
begin
    find the smallest of the above linear combinations, say w_k, which
        intersects the rows of the row-block, to row_{k-1}, in the correct
        pattern if it were row_k;
    if w_k is not row_k
    then reject v;
end
find the smallest of the above linear combinations, say w_j, which
    intersects the previous rows of the row-block in the correct
    pattern if it were row_j;
if w_j is not v
then reject v.

In this case "smallest" means that the vector in question appears earliest in the candidate list of all the vectors being considered. The reason that we can reject based on these criteria is that if such a case exists, then we have already seen the case in question, or more specifically an isomorphic case, earlier in the search.

There may be one necessary modification to the above. If we are insisting, as shown in section 7.3.4, that the first two rows of a row-block are both of type $4 \cdot 2^3$, then we must ensure that when comparing the smallest linear combination against the first row of a row-block, we only reject the candidate if there exists another vector of type $4 \cdot 2^3$. This vector would be used as the second row of the row-block in the isomorphic case.

7.3.6 Resulting Estimates for $b = 3$

In turn we take each of the 14 starting matrices chosen from our 6 cases. There are 3 row-blocks remaining to be filled, and row_{21}. In the first two row-blocks, to "fill

Row	#Candidates	#Survivors
9	2 E10	5 E7
10	5 E10	2 E9
11	2 E12	4 E10
12	8 E11	5 E7
13	4 E10	2 E9
14	7 E10	1 E9
15	1 E12	2 E8
16	3 E8	0

Table 7.5: Estimates of Candidates and Survivors by Row for $b = 3$

out" the search and to try to obtain reliable estimates as described in section 7.2.4. we choose a maximum of 10. 20. 50 and 5 survivors for the first. second. third and fourth rows of each row-block respectively. We choose a maximum of 3 survivors for each of the remaining rows. If less than the requested maximum number of survivors exist for any particular row. then we take all remaining survivors for that row. We use the general algorithm described in this chapter. together with the "refinements" described in sections 7.3.1. 7.3.4 and 7.3.5. Our estimates. for the total of all 51 364 starting matrices. are shown in Table 7.5.

As can be seen from this table. the search is largest at two points. namely around the point at which we are trying to complete row_{11} and around the point at which we are trying to complete row_{15}. For a discussion of how these estimates of the size of the search may be converted to estimates of the amount of time required to complete the search. together with the results of the conversion. see Chapter 9.

7.3.7 Resulting Estimates for $b = 2$

For this case. we have 10 starting matrices chosen from our original 20 875 non-isomorphic codes complete to row_7. We take each of these in turn. again trying to "fill out" the search as described in section 7.2.4. As before. if less than the requested maximum number of survivors exist for any particular row. then we take all remaining survivors for that row. We use the general method described in this chapter. together with the "refinements" described in sections 7.3.1. 7.3.3 and 7.3.4. Our estimates for the total of all 20 875 starting matrices. are shown in Table 7.6.

Row	#Candidates	#Survivors
8	3 E8	9 E7
9	4 E11	3 E10
10	1 E13	7 E10
11	4 E11	0

Table 7.6: Estimates of Candidates and Survivors by Row for $b = 2$

As can be seen from this table, the search is largest at the point at which we are trying to complete row_{10}. Again, for a discussion of how these estimates of the size of the search may be converted to estimates of the amount of time required to complete the search, together with the results of the conversion, see Chapter 9.

7.4 Summary

In this chapter and the previous chapter, we have presented a method of organizing the search for a (48, 24, 12) self-dual doubly-even code. There are three cases to consider, based on the maximum dimension b of a (24, b, 12) subcode. The results of the case $b = 4$ are already known. We examined the remaining 2 cases, namely $b = 3$ and $b = 2$, in some detail, presenting a methodology and possible improvements which could be used in a search assuming either case. We have also presented estimates of the number of candidates and number of survivors we expect to see for each row of the generator matrix.

7.5 Further Work

There are two remaining cases in the search for a (48, 24, 12) self-dual doubly-even code.

To complete a search assuming either remaining case, we must first make a full analysis of the various improvements discussed in this chapter, in particular an analysis of the "cost" of each algorithm (in terms of number of operations) as seen in Chapter 9. The results of an analysis of the type shown in Chapter 9 may help to indicate which of the prospective improvements are worthwhile, and which algorithms require further optimization.

Upon implementation of the search, if any $(48, 24, 12)$ codes are found then we must verify if any of these are isomorphic to the known code, the Extended Quadratic Residue code QR.

Upon completion of these two remaining cases, we will know the exact number of $(48, 24, 12)$ self-dual doubly-even codes, and hence have an answer to Conjecture 6.1. As previously mentioned, such extremal codes with length divisible by 24 are of particular interest. It is not known if any $(72, 36, 16)$ self-dual doubly-even codes exist, and therefore this is the next case to consider.

Chapter 8

Computational Methods

In this chapter we discuss several techniques which can be used in searches of the types discussed in this thesis. For each technique we use an example to illustrate its use. Several of these techniques have been used during the searches discussed in this thesis.

8.1 Vector Storage and Manipulation

Many mathematical structures are collections of vectors over some field F^n, where F is $GF(q)$, the finite field with q elements. In this thesis we have been primarily concerned with binary structures. In these cases F is $GF(2)$. Therefore the vectors in question have some length n, with each of the n components having the value 0 or 1.

If we wish to implement a search for a binary structure then it is important to be able to efficiently store and manipulate binary vectors. In this section we shall examine this problem, using as an example the search for a $(48, 24, 12)$ self-dual doubly-even code.

If C_{48} is such a code then each codeword in C_{48} is a binary vector of length 48. Each non-zero codeword must have a weight of at least 12 and which is divisible by 4. There are a total of 2^{24} such vectors in C_{48}.

In practice we store a basis of C_{48} in a generator matrix. Therefore to obtain all codewords in C_{48}, we must compute all linear combinations of the vectors in the generator matrix. In our search we attempt to fill a generator matrix for C_{48}.

To verify that C_{48} has the required properties, we must verify that every codeword has a valid weight, and therefore we must compute, and possibly store, every linear combination of vectors in the generator matrix.

Therefore each of the following items are important in our search:

- The efficient storage of binary vectors.
- An efficient method of addition of such vectors over $GF(2)$, and
- An efficient method of computing the weight of such vectors.

8.1.1 Storage and Addition of Codewords

Because we are concerned with binary vectors, we make use of the binary nature of computers. Specifically, we can use 48 bits to store a codeword of length 48, with each bit representing a single component of the vector. Many computers have a 32-bit word length. In such a case we must use 2 words to store each vector. The leftmost 32 components can be stored in the 32 bits of the first word, and the rightmost 16 components can be stored in any 16 bits of the second word. We choose to use the leftmost 16 bits of the second word. If using a computer with a 64-bit word length, we can use 1 word to store a vector. We choose to use the leftmost 48 bits of the word to store the vector. Note that with either of these organizations, for each vector of length 48 we must use a total of 64 bits and therefore 16 bits are unused.

Suppose v and w are codewords stored in such a fashion. To add v and w over $GF(2)$, we use a bitwise exclusive-or function. This function takes the exclusive-or of all bits simultaneously. The ith bit of the resulting vector is the exclusive-or of the ith bit of v with the ith bit of w. If we are using two 32-bit words to store each length-48 vector, then to compute $v + w$ we must first compute the bitwise exclusive-or of the first word of v and the first word of w, then the bitwise exclusive-or of the second word of v and the second word of w.

8.1.2 Computing the Weight of Codewords

There are 2^{48} possible binary vectors of length 48. Not all of these are valid codewords since many have an invalid weight. To determine the weight of a codeword we must count the number of its non-zero components.

One method of computing the weight of a codeword v stored as described above is to isolate each bit of v in turn, incrementing the computed weight if that bit is a 1. Clearly this can be very time-consuming. It is more efficient to pre-compute the weight if possible.

If we were to pre-compute the weight of all possible binary vectors of length 48 we would require a table with 2^{48} entries. Each entry corresponds to a particular binary vector of length 48. If stored bitwise as described above, we can think of the vector as an integer stored in base 2, with 48 binary digits. Therefore the entry in position i of our table would be the weight of the vector corresponding to the integer i stored in base 2. A table of 2^{48} entries, with each entry requiring one word, requires 2^{48}, or approximately 28 trillion, words.

Clearly it is not reasonable to store a table of such a size. A more reasonable approach is to divide each vector into 3 sections consisting of 16 components each. We precompute the weight of all possible binary vectors of length 16, storing these weights in a table. This table requires only 2^{16}, or approximately 65 thousand, words. To check the weight of the codeword w, we add together the weights of w restricted to its first 16 components, w restricted to its middle 16 components, and w restricted to its last 16 components, by looking up each of these in the precomputed table.

In reality, we are using computers which have either a 32-bit word length or a 64-bit word length. In either case, we use 64 bits to store each codeword. We found it convenient to use an interface which assumed vectors of 64 bits, and which would function on computers which have either 32-bit or 64-bit words. As described in section 8.1.1, since our codewords only require 48 bits, $64 - 48 = 16$ bits remain unused. Our interface computes the weight as described above, except assuming a 64-bit vector. An algorithm to compute and return the weight of a 64-bit vector v, assuming a weight-table *WeightTab* for 16-bit vectors, is:

```
FindWeight(v)
begin
    let wt = WeightTab[v AND FFFF16] +
             WeightTab[RightShift(v.16) AND FFFF16] +
             WeightTab[RightShift(v.32) AND FFFF16] +
             WeightTab[RightShift(v.48) AND FFFF16]:
    return wt:
end
```

Note that "v AND $FFFF_{16}$" computes the bitwise 'and' of v with the 16-bit vector of 16 ones. This has the effect of "masking off" the rightmost 16 bits of v. The function RightShift(v, i) shifts the bits of v to the right by i positions. Since our codewords are 48-bit vectors, the first table reference, together with its associated RightShift and AND, is in fact unnecessary.

8.2 Ordering of Candidates

In searching for a code, one method which we have used is to attempt to directly construct such a code, by completing its generator matrix. Similarly one can search for a block design by attempting to complete an incidence matrix for the design.

In each of these cases we attempt to place certain constraints upon the matrix to speed up the search. For example, we may be able to force a certain structure upon the matrix such as knowing the number of ones which must be placed in a row or in a column. We may also be able to divide the matrix into several sections, with each section having a particular structure.

Once this structure is in place we must try all possibilities which respect the structure. Generally, as described in section 2.5, we attempt to fill the matrix one section at a time, ensuring that the section in question has the required structure. We then try all possibilities with the required structure for every other section in turn, additionally ensuring that sections completed later do not conflict with sections completed earlier.

For an example of this general method we shall use the search described in Chapter 4, in which we attempted to embed (28, 12, 11) designs as derived designs in (64, 28, 12) designs. In this case we are attempting to complete an incidence matrix for a (64, 28, 12) design which has the structure shown in Fig. 4.2, where the values a, b, c and d provide extra information about the intersection between columns 1 and 2, as described in section 4.3.

In general when completing a generator matrix or an incidence matrix we have a certain list of candidates to fill each row, or perhaps each section, of the matrix. In our example we obtained a list of candidates for any of the last A_r rows by the method described in section 4.3. The size of each of these candidate lists is given in Table 4.1.

In our example the first row we shall attempt to complete is row 29. Suppose we are considering case 164, solution 5. Then there are 300 candidates. 85 of these contain zeroes in both of the first two columns and thus are candidates to fill the first a rows. 78 of these contain a zero in the first column and a one in the second column and thus are candidates to fill the second b rows. 77 of these contain a one in the first column and a zero in the second column and thus are candidates to fill the third c rows. 60 of these contain ones in both of the first two columns and thus are candidates to fill the last d rows.

Consider any two rows r_i and r_j, $i < j$ and in which r_i and r_j both appear in the same section as defined by the first two columns. Suppose that v_i is used to fill r_i and v_j to fill r_j. Then since the vectors used to fill r_i and r_j must both obey the constraints imposed by $r_1, \ldots, r_{i-1}$, the vector v_j could be used to fill r_i and the vector v_i to fill r_i. Furthermore, these two cases are isomorphic. Therefore, we can insist that v_i appears earlier in the candidate list than v_j. This eliminates from our search an isomorphic completion in which v_i is used to fill r_j and v_j is used to fill r_i. If we insist that r_j appears later in the candidate list than k previous rows, then we eliminate $k-1$ isomorphic possibilities in this manner: the number of possibilities to be considered is therefore reduced by a factor of k.

In this thesis, there are two examples in which we have insisted upon such an

ordering. During the search for a (64, 28, 12) design, the last A_r rows of the incidence matrix are divided into 4 sections, of a, b, c and d rows respectively, as described above and in section 4.3. Within each of these sections, we insist that the vector used to fill row r_j must come later in the candidate list than the vector used to fill row r_i, if $i < j$. During the search for a (48, 24, 12) self-dual doubly-even code as described in Chapter 7, we similarly insist that within a row-block, the vector used to fill row_j must come later in the candidate list than the vector used to fill row_i, for $i < j$.

8.3 Generation and Maintenance of Candidate Lists

One method which can be used to complete a generator matrix or incidence matrix is to perform a depth-first search row by row, as described in section 2.5. We choose an order in which to try to complete rows, often in order from the top to the bottom of the matrix. We find all possibilities for completing the current row. We choose each of those possibilities in turn, proceeding to the next row following the same method.

In general for each row we have a certain number of *candidates* which must be tested to determine if they can indeed be used to fill that row. If they can then they become *survivors*. At some point we must obtain the list of candidates for each row.

Let us use the search for a (48, 24, 12) doubly-even code, in which RM has 3 rows, as an example. At the start of the search, the first row which we attempt to complete is row_4. We already know row_1–row_3 and row_{22}–row_{24}. Based upon this known configuration, we can insist that row_4 must have either 1 block of 4 ones and 3 blocks of 2 ones, or 3 blocks of 3 ones and 1 block of a single one, as shown in Chapter 6. There are a total of $4 \cdot \binom{6}{4} \cdot \binom{6}{2}^3 + 4 \cdot \binom{6}{3}^3 \cdot 6 = 394500$ possible vectors of these two types and it is simple to generate a list of these vectors.

As seen in Chapter 7, within a row-block the list of survivors for row_i becomes the list of candidates for row_{i+1}. Also the list of survivors from the first row of a row-block becomes the list of candidates for the first row of the following row-block.

So for this example, as we proceed in the depth-first search we use the required survivor list from a previous row for the candidate list for our current row. If at any

point we do not find any survivors for our current row, this means that it is impossible to complete the generator matrix in its current form and we must backtrack to the previous row. Similarly if we run through all survivors for the current row, and none allow any survivors for the following row, then we must again backtrack. Therefore as the depth-first search proceeds the candidate list for any particular row changes based on the configuration given by the previously completed rows.

For example, consider row_{11}. The survivors for this row, which are the candidates for row_{12}, must satisfy the constraints imposed by $row_1, \ldots, row_{10}$ and $row_{22}, \ldots, row_{24}$. Upon backtracking and replacing the contents of row_{10} by some row'_{10}, the candidate list for row_{12} changes, as it must now consist of all vectors satisfying the constraints imposed by $row_1, \ldots, row'_{10}$ and $row_{22}, \ldots, row_{24}$.

8.4 Testing for Survivors

If we proceed as above then we test the candidates for each row in turn. For the above example, the testing consists of:

- adding the candidate to all previously known codewords, to find all possible linear combinations, and
- checking if every linear combination has a weight that is both doubly-even and greater than the minimum weight of the code.

We may also use some of the refinements mentioned in Chapter 7, such as testing if a (24, 4, 12) subcode exists. If the candidate passes all of the tests then it becomes a survivor.

For codes there is only a restriction on the number of ones which must be present in any row of the generator matrix. However, this restriction is also applied to all codewords generated by the partially-completed matrix. When attempting to complete the incidence matrix of a design, there is also a restriction on the number of ones which must be present in any column. For an incidence matrix the number of ones present in a row or in a column is fixed. For this reason we may choose to

complete an incidence matrix column by column instead of row by row. Of course the two methods may be mixed.

It is sometimes beneficial to delay testing a candidate at a certain level (row or column), or alternatively to precompute the results of some of these tests if possible. As a search progresses the amount of work which must be done generally increases with each level until the restrictions imposed by previously completed areas make it difficult to complete any further areas. After this point, which can be called the *bulge* of the search, there will be very few survivors. It is extremely desirable to try to avoid doing too much work at the bulge.

If we can delay work which would normally be done at the bulge until later, we may be able to save some computing time. For an example of this technique, see section 7.3.2.

8.5 Compatibility Matrices

When attempting to complete a generator matrix of a code, or an incidence matrix of a design, compatibility matrices may be useful. For the purposes of this explanation, we assume that we are completing entire rows of a generator or incidence matrix B, although we may use a compatibility matrix when completing regions of any shape.

Suppose we have already partially completed B. Based on this initial configuration of B, further suppose that we have c_i candidates for row_i of B and c_j candidates for row_j of B.

A candidate must pass several tests to become a survivor, as described in the previous section. Suppose that r_i is a candidate for row_i and r_j is a candidate for row_j. Then r_i and r_j are considered *compatible* if all of the above tests would be passed for both r_i and r_j, if r_i were used for row_i and r_j were used for row_j. A $c_i \times c_j$ compatibility matrix $M_{i,j}$ may be built which contains this information. Specifically, $M_{i,j}[r_i][r_j]$ is TRUE if and only if r_i and r_j are compatible. To determine the set of all candidates for row_j which are compatible with a particular row_i, we find all entries in the row $M_{i,j}[row_i]$ which are TRUE.

A compatibility matrix can be very useful as the search becomes large. We can

precompute a compatibility matrix based on a configuration which exists before the bulge, and then use it at the bulge. At the bulge, our matrix B will be more complete than at the time the compatibility matrix was created; however, it is extremely expensive to do extensive testing at this point. Therefore, instead of conducting tests at the bulge we simply check the precomputed information, and can do more extensive tests after the bulge if necessary. Note that this implies that generally the information contained in a compatibility matrix is most useful if it is related to those rows which would be completed at or near the bulge.

Suppose that we have not already chosen a row_i. Compatibility matrices may also be used to find the set of all possible compatible pairs $\{r_i, r_j\}$ for row_i and row_j. For an example, see [19].

Several compatibility matrices may be used at once. For example, suppose that we wish to precompute information not only for row_i and row_j, but also for some other row row_k, which has c_k candidates. Then we will have 3 compatibility matrices:

- $M_{i,j}$, a $c_i \times c_j$ matrix, for compatibility between candidates for row_i and row_j.
- $M_{i,k}$, a $c_i \times c_k$ matrix, for compatibility between candidates for row_i and row_k, and
- $M_{j,k}$, a $c_j \times c_k$ matrix, for compatibility between candidates for row_j and row_k.

A set of candidates $\{r_i, r_j, r_k\}$, where r_i is a candidate for row_i, r_j is a candidate for row_j, and r_k is a candidate for row_k, is compatible if every subset of size 2 is compatible. Therefore to determine the set of all candidates for row_k which are compatible with a particular row_i and a particular row_j, we must find all entries which are TRUE in the row:

$$M_{i,k}[row_i] \text{AND } M_{j,k}[row_j].$$

Note that we must have already determined that row_i and row_j are compatible by checking $M_{i,j}$.

8.6 Summary

In this chapter we examined computational techniques, several of which have been employed during the searches discussed in this thesis. Although we used specific examples to demonstrate their use, these techniques may be applicable to many other searches of a similar nature.

Chapter 9

Estimate Conversion

We shall now look at how an estimate of the size of a search may be converted to an estimate of the time required for the search. In this case, the amount of time required is directly related to the number of computer operations required, and the speed of the computer being used.

For this chapter, assume that we are performing a backtrack search, as described in section 2.5. Generally, the amount of time required to complete a particular level of a search may be obtained by considering:

- the number of operations required for items which are performed at every level (say x cycles).
- the number of operations required for items which are performed for every candidate at level i (say y_i cycles), and
- the number of operations required for items which are performed for every survivor at level i (say z_i cycles).

A simplisitc view gives the total number of cycles required to complete level i of the search as:

$$x + y_i \cdot c_i + z_i \cdot s_i,$$

where c_i denotes the number of candidates at level i and s_i denotes the number of survivors at level i.

One should note that although we wish our estimates to be as accurate as possible, a rigorous accounting of machine cycles is unnecessary. Specifically, within each algorithm one may safely ignore certain details such as the overhead (in cycles) required for looping constructs, without these details making a significant impact upon the accuracy of the resulting estimates.

The goal of this chapter is simply to present a method of converting estimates, using a specific example. The algorithms presented for this example are indeed *algorithms*, and when written in any specific programming language, certain changes will be necessary. Again, these changes should not make a significant impact upon the accuracy of the estimates.

Finally, the resulting estimates may be most useful as a diagnostic tool, for identifying the specific algorithms which may need improvement. Indeed we shall see specific examples of this at the end of this chapter.

9.1 The Search for a $(48, 24, 12)$ Self-Dual Doubly-Even Code

For an example, we shall consider the search for a $(48, 24, 12)$ self-dual doubly-even code containing a subcode RM with $b = 2$ or 3 rows, as described in Chapter 7.

9.1.1 Operations Performed at Every Level

In this search, the only operation performed for each row, independent of the number of candidates or number of survivors, is the selection of the candidate list.

9.1.2 Operations Performed for Every Candidate at Every Level

In this search we find it convenient to have a buffer storing all linear combinations of the currently filled rows of the generator matrix, that is, all vectors which are currently in the code. The following items may be performed for each candidate for a row:

- add the candidate to each vector in the buffer, storing the result in the buffer;

- check each vector in the buffer, rejecting the candidate if a vector is not doubly-even or has a low weight;
- check the contents of the column-blocks of the candidate, rejecting the candidate if it is of the wrong row type (see section 7.3.4);
- for $b = 2$, determine the maximum value b' for which the partial code stored in the buffer contains a $(24, b', 12)$ subcode, rejecting the candidate if $b' > b$;
- for $b = 3$, perform "poor-woman's isomorphism testing", possibly rejecting the candidate.

Note that it may not be necessary to perform all of the above items for some candidates, since if a candidate has been rejected by one of the earlier tests, the later tests will never be performed. However, we may still wish to count the total amount of time to perform all of the tests, as this is a ceiling on the actual amount of time required for each candidate.

Also, for simplicity in this discussion we have logically separated the addition of vectors to the buffer, and the weight-check of vectors in the buffer. In reality it is almost certainly more efficient to check the weight of vectors as they are being added to the buffer, since if ever a vector with invalid weight is generated, we can immediately reject the candidate without generating any further vectors.

9.1.3 Operations Performed for Every Survivor at Every Level

In this search, the following items are performed for each survivor s for row_i:

- store the survivor in the appropriate survivor-list, and
- continue to the next level (row_{i+1}), with $row_i = s$.

In continuing to the next level, we also have to update the buffer by adding s to each vector already in the buffer. Note that we would have done this once already, when s was only a candidate, not a survivor. However, we find all survivors for row_i before proceeding to row_{i+1} as the list of survivors for row_i usually becomes a candidate

list for some later row. Therefore part of the buffer will almost always have been overwritten while checking a different candidate, and we must restore this part.

In general, the number of candidates for any given row far exceeds the number of survivors for that row, and hence the total time required to perform the search is dominated by the items performed for each candidate for a row. For this reason, we shall concentrate our efforts on obtaining a reliable estimate of the time required to perform these items. We shall now give algorithms for each of these items, and analyze the number of operations required for each algorithm. For the purposes of this analysis, assume that we are using a computer with a 64-bit word length; in this case, each codeword is stored in exactly one such word, as discussed in section 8.1.1.

9.2 Adding Codewords to the Buffer

Assume that at the start of the program, we compute all linear combinations of the currently-filled rows of the generator matrix, storing them in *Buf*; the number of vectors stored in *Buf* is *SizeOfBuf*. An algorithm for adding a codeword w to each vector in *Buf* is:

```
AddToBuf(w, Buf, SizeOfBuf)
begin
    for i = 1 to SizeOfBuf do
    begin
        let Buf[SizeOfBuf + i] = Buf[i] + w;
    end;
end
```

At the start of the program, 11 rows have already been filled for the case $b = 3$, namely $row_1, \ldots, row_8$ and $row_{22}, \ldots, row_{24}$. Therefore at the start of the program, $SizeOfBuf = 2^{11} = 2048$. When testing candidates for row_j, we have just completed row_{j-1}, and therefore $SizeOfBuf = 2^{(j-1)+3}$. The loop is thus executed a maximum of 2^{j+2} times for every candidate at row_j.

For the case $b = 2$, we have initially filled 9 rows, namely $row_1, \ldots, row_7, row_{23}$, and row_{24}. Therefore when testing candidates for row_j, the loop is executed a maximum of 2^{j-1} times for every candidate at row_j.

Here, and for the remainder of the analysis in this chapter, we shall ignore the overhead involved in any looping structure itself. Therefore we count 6 memory references and 2 additions at each iteration of the loop. This gives a maximum of $8 \cdot 2^{j-2}$ operations to be executed for every candidate for row_j for the case $b = 3$ and a maximum of $8 \cdot 2^{j-1}$ operations to be executed for every candidate for row_j for the case $b = 2$.

9.3 Checking the Buffer

An algorithm for checking the weight of each vector in *Buf*, returning TRUE if any vector has invalid weight and FALSE otherwise, is:

```
1   CheckBuf(Buf, SizeOfBuf)
2   begin
3       for i = SizeOfBuf+1 to 2*SizeOfBuf do
4       begin
5           let wt = weight of Buf[i];
6           if (wt < 12) or wt is not divisible by 4
7           then return TRUE;
8           if wt = 12
9           then begin
10              let MinwtVec[NumMinwt] = Buf[i];
11              increment NumMinwt;
12          end;
13          if wt = 24
14          then begin
15              let SubcodeLenVec[NumSubcodeLen] = Buf[i];
16              increment NumSubcodeLen;
```

```
17              end:
18          end:
19          return FALSE:
20      end
```

First let us consider how the weight of a codeword is calculated. As discussed in section 8.1.2, we may use a table *WeightTab* to store the weight of 16-bit sections of a 64-bit vector. The algorithm used to compute weight is that shown in section 8.1.2. To compute the weight of any particular codeword requires 4 references to the table and 3 additions. We also must select each of these 16-bit sections in turn by using a mask involving a logical operation; except for the leftmost 16-bit section of the codeword, we must also use a shift to move the 16-bit section of interest all the way to the left. Therefore to compute the weight of a codeword requires 4 memory references, 4 logical operations and 3 shifts, followed by 4 more memory references and 3 additions, for a total of 18 operations per codeword.

We require 3 memory references to execute line 5 of CheckBuf, on top of the operations required to compute the weight. We require 2 memory references, 2 comparisons and 2 logical operations to execute line 6 of CheckBuf. Lines 8–16 are necessary if we are checking the the existence of a $(24, b', 12)$ subcode, as we need these lists of weight 12 and weight 24 vectors in the CheckSubcode algorithm shown later in this chapter. Lines 8 and 13 each require 1 memory reference and 1 comparison. At most one of lines 10 and 15 will be executed. Each of these lines requires 4 memory references, therefore these 2 lines account for a total of at most 4 operations. At most one of lines 11 and 16 will be executed. Each of these lines requires 1 memory reference and 1 arithmetic operation, therefore these 2 lines account for a total of at most 2 operations. Therefore for each iteration of the loop, we require 37 operations.

For this algorithm, *Buf* contains all linear combinations of previously completed rows together with the current candidate. The loop in CheckBuf iterates through the last *half* of the linear combinations in the buffer. Note that due to the method of computing *Buf* given in AddToBuf, all linear combinations involving the current

candidate will occur in this last half of *Buf*. Also note that we must reset NumMinwt and NumSubcodeLen to their appropriate previous values if a candidate is later rejected.

For the case $b = 3$, when testing candidates for row_j, we have already completed $j+2$ rows, namely $row_1, \ldots, row_{j-1}$ and $row_{22}, \ldots, row_{24}$. Since *Buf* contains all linear combinations of these rows and the current candidate, $SizeOfBuf = 2^{j-3}$. Therefore the loop in this algorithm is executed a maximum of 2^{j-2} times for every candidate for row_j, and thus we require a maximum of $37 \cdot 2^{j-2}$ operations per candidate.

Similarly, for the case $b = 2$, $SizeOfBuf = 2^{j-2}$ and therefore we require a maximumof $37 \cdot 2^{j-1}$ operations per candidate.

9.4 Checking Column-Blocks

We shall now consider an algorithm to check the column-blocks of a codeword. To implement this, we should initialize several constant values at the start of the program. Assume that at the start of the program, we initialize *LeftBlockMask*[1...] to values which will respectively "mask off" each of the column-blocks of the left-hand side. For example, we would initialize *LeftBlockMask*[3] to $000FC0000000_{16}$ to mask off the third column-block on the left-hand side for the case $b = 3$. Also assume that we initialize *BlockRow*[i] to contain the position of row_i within its row-block. For example, we would initialize *BlockRow*[11] to 3 for the case $b = 3$, since for this case row_{11} is the third row within its row-block.

An algorithm for the case $b = 3$, for checking the column-blocks of a codeword w being tested as a candidate for row r, returning TRUE if w can be rejected based on invalid column-blocks and FALSE otherwise, is:

```
1    CheckBlocks(w, r)
2    begin
3        let wt = weight of (w AND LeftBlockMask[1]):
4        if wt = 2
5        then return TRUE;
```

```
6          if wt is odd
7          then begin
8               if BlockRow[r]≤ 2
9               then return TRUE:
10              find the first 2 column-blocks of 3 ones in w:
11              if the first column of each of these does not contain a 1
12              then return TRUE:
13         end:
14         return FALSE:
15    end
```

As previously shown, we require 18 operations to compute the weight of a codeword. To compute the weight of a particular section of a codeword, we must first "mask off" the section of interest by taking the bitwise AND of the codeword with a vector containing ones in that section; we then compute the weight of the resulting vector. As shown in line 3 of the above algorithm, we can perform this extra step using 2 memory references and 1 logical operation; storing the result gives a total of 22 operations.

Line 4 requires 1 memory reference and 1 comparison. Line 6 requires 1 memory reference and a logical operation. Line 8 requires 2 memory references and 1 comparison. To execute line 10, we need to find the weight of at most 3 column-blocks. Since we already know the weight of *BlockRow*[r] from line 8, in the worst case in line 10 we will need to find and store the weight of 2 more column-blocks; we also require 2 comparisons. This gives a maximum of 46 operations for line 10. To execute line 11, we can find the weight of the first column of each selected column-block using a mask on that column. If using this method, line 11 will require $2 \cdot 21$ operations to find the weight of each of these single bits, plus 2 comparisons and a logical operation, for a total of 45 operations. Therefore the above algorithm will require a maximum of 120 operations total.

Since this is executed only once per candidate and no loops are involved, this is a

cheap algorithm to implement and its cost is not significant in comparison with the other algorithms. For this reason, we could ignore the cost of executing this algorithm in our analysis. However, as we shall see, this algorithm is also required for the case $b = 3$ as part of "poor-woman's" isomorphism testing. As part of this testing, its cost becomes more significant as it is executed frequently.

For the case $b = 2$, the algorithm is as follows:

```
1    CheckBlocksb2(w, r)
2    begin
3         let wt = weight of (w AND LeftBlockMask[1]):
4         if wt = 4 and BlockRow[r] ≤ 5
5         then return TRUE:
6         find the first column-block of 6 ones in w:
7         if its first column does not contain a 1
8         then return TRUE:
9         return FALSE:
10   end
```

We shall not present a full analysis of the cost of the above algorithm for $b = 2$, as its cost is not significant in comparison with the other algorithms and thus may safely be ignored.

9.5 Subcode Test

As mentioned in section 7.3.3, we may reject the current candidate if the partial code stored in *Buf* contains a $(24, b', 12)$ subcode, $b' > b$. For $b = 2$ there should be $ValidSubMinwtCount = 2$ codewords of weight 12; for $b = 3$ there should be $ValidSubMinwtCount = 6$ codewords of weight 12. The following algorithm CheckSubcode returns TRUE if a subcode with more than this number of weight 12 vectors exists and FALSE otherwise:

```
1    CheckSubcode()
2        for i = 1 to NumSubcodeLen do
3        begin
4            let SubMinwtCount = 0:
5            for j = 1 to NumMinwt do
6            begin
7                if (NOT(SubcodeLenVec[i]) AND MinwtVec[j]) = 0)
8                then increment SubMinwtCount:
9            end:
10           if SubMinwtCount > ValidSubMinwtCount
11           then return TRUE:
12       end:
13       return FALSE:
14   end
```

Line 7 is used to determine if $MinwtVec[j]$ has a weight of 12 in the subspace defined by the ones of $SubcodeLenVec[i]$ — namely, it determines if all ones in $MinwtVec[j]$ match up against ones in $SubcodeLenVec[i]$. It requires 4 memory references, 2 logical operations and 1 comparison. Line 8, if executed, requires 1 memory reference and 1 arithmetic operation. We shall ignore the cost of the remaining lines as they either do not appear in the innermost loop, or are concerned only with control structure.

The outermost loop executes at most *NumSubcodeLen* times and the innermost loop executes at most *NumMinwt* times. In fact, we only use this algorithm for the case $b = 2$. In this case, $NumSubcodeLen = NumMinwt$ for the first column-block of the right-hand side. We approximate both *NumSubcodeLen* and *NumMinwt* as 1/8 of the total number of codewords at each level. Since $SizeOfBuf = 2^{j-2}$, we therefore approximate both *NumSubcodeLen* and *NumMinwt* as 2^{j-1}.

Therefore we approximate the total cost of the CheckSubcode algorithm as $9 \cdot 2^{2j-2}$.

9.6 Partial Isomorphism Testing

An algorithm for partial isomorphism testing, which we call "poor-woman's" isomorphism testing, is given in Chapter 7. As mentioned in Chapter 7, we only used this method for the case $b = 3$. Therefore although we shall explain the algorithms for both case $b = 2$ and case $b = 3$, we shall only count operations which apply to the case $b = 3$.

We shall expand our explanation of this algorithm here. First we need to consider several other algorithms, used in "poor-woman's" isomorphism testing, to proceed with our analysis.

9.6.1 Finding Vectors with the Correct Distribution of Ones

First we require an algorithm for finding all vectors with weight 10 on the left-hand side and weight 2 in the correct column-block on the right-hand side. Assume that at the start of the program, we initialize *LeftMask* to $FFFFFF000000_{16}$, which "masks off" the left-hand side of the generator matrix. Also assume that we initialize *RightBlockMask*$[i]$ to a value which will "mask off" the right-hand side column-block containing 2 ones in row_i. For example, for the case $b = 3$ we would initialize *RightBlockMask*$[11]$ to $00000003F000_{16}$, which masks off the second column-block on the right-hand side.

Later we shall see that we frequently need to check the "type" of a particular vector. In particular, for the case $b = 3$ we are interested in whether a particular vector is of type $4 \cdot 2^3$ or of type $3^3 \cdot 1$. For the case $b = 2$, we also divide vectors into 2 groups. For this case, we are interested in whether a particular vector has 6 ones in the first column-block ("type $6 \cdot 4$") or in the second column-block ("type $4 \cdot 6$"). For each case, we shall refer to the first of the possibilities as *type 1* and the second of the possibilities as *type 2*.

We shall store this information for each vector we find. Specifically, *Type1*$[i]$ is TRUE if and only if *PWList*$[i]$ is of type $4 \cdot 2^3$ (for the case $b = 3$) or of type $6 \cdot 4$ (for the case $b = 2$).

An algorithm to find all vectors with weight 10 on the left-hand side and weight

2 in the column-block for row r, storing them in *PWList*, follows.

```
1    Find10_2(r, Buf, SizeOfBuf)
2    begin
3        let PWListSize = 0;
4        for each vector v with 2 ones in the column-block for row r do
5        begin
6            let LeftWt = weight of (v AND LeftMask);
7            let BitsOk = CheckBlocks(v, r);
8            if LeftWt = 10 and BitsOk = TRUE then
9            begin
10               increment PWListSize;
11               PWList[PWListSize] = v;
12               if weight of (v AND LeftBlockMask[1]) = 4 (for b = 3) or 6 (for b = 2)
13               then let Type1[PWListSize] = TRUE;
14               else let Type1[PWListSize] = FALSE;
15           end;
16       end;
17   end
```

Note that in this version, we only keep those vectors which, based on their column-blocks, could be used to fill row r, that is those vectors w for which the algorithm CheckBlocks(w, r) would return FALSE.

Also note that we only consider those vectors with weight 2 in the column-block for row r. It is straightforward to find these vectors as each one is in fact a linear combination of a predetermined set of rows. For example, for the case $b = 3$ when testing a candidate for row_{10}, the only vectors which can have weight 2 in the correct column-block are:

- linear combinations of $row_1, \ldots, row_3$ and row_9.
- linear combinations of $row_1, \ldots, row_3$ and the current candidate for row_{10}, and

- linear combinations of $row_1, \ldots, row_3$, row_9 and the current candidate for row_{10}.

Since the rows in these linear combinations are predetermined, we know in exactly which locations of *Buf* these linear combinations appear. As usual, we shall not consider the overhead for the looping structure in this algorithm. In particular, we shall not consider the overhead involved in finding the necessary locations of *Buf*.

In general, if r is the i'th row of a row-block, then for the case $b = 3$, there are $8 \cdot (2^i - 1)$ vectors with 2 ones in the correct column-block. Therefore the loop in this algorithm is executed $8 \cdot (2^i - 1)$ times for the i'th row of a row-block. Of these vectors, exactly half have weight 10 on the left-hand side; the other half have weight 14 on the left-hand side. Exactly one quarter of those with weight 10 will satisfy the CheckBlocks algorithm. Therefore if r is the i'th row of a row-block, then the statements from lines 9–15 are executed at most $2^i - 1$ times each and at the end of this algorithm, $PWListSize = 2^i - 1$.

Line 6 requires 18 operations to compute the weight of v, plus 3 memory references and 1 logical operation, for a total of 22 operations. As shown, the algorithm CheckBlocks requires a maximum of 120 operations; line 7 requires an additional 3 memory references, for a maximum of 123 operations. Line 8 requires 2 memory references, 2 comparisons and a logical operation. If executed, line 10 requires 1 memory reference and 1 arithmetic operation, and line 11 requires 3 memory references. If executed, line 12 requires 18 operations to compute the weight, plus 2 memory references, 1 logical operation and 1 comparison, for a total of 22 operations. At most one of lines 13 and 14 will be executed. Each of these use 2 memory references and thus these two lines combine for a total of 2 operations. We therefore estimate the total number of operations required as $(150 \cdot 8 + 29) \cdot (2^i - 1) = 1229 \cdot (2^i - 1)$ when r is the i'th row of a row-block. We shall use this number as the "cost" of the algorithm Find10_2.

9.6.2 Finding "Earliest" Vectors

The previous algorithm finds all vectors with weight 10 on the left-hand side and weight 2 in the column-block for row r. We also need to be able to find the "earliest" vector from such a list. In this case, a vector x would be considered "earlier than"

a vector y if x would appear earlier in the candidate list than y. An explanation of how we order the candidate list is given in Chapter 7.

An algorithm for the case $b = 3$ which returns TRUE if x is "earlier than" y, and FALSE otherwise, is:

```
1    EarlierThan(x, y)
2    begin
3        if weight of (x AND LeftBlockMask[1]) is even
4        then let IsEvenx = TRUE;
5        else let IsEvenx = FALSE;
6        if weight of (y AND LeftBlockMask[1]) is even
7        then let IsEveny = TRUE;
8        else let IsEveny = FALSE;
9        if IsEvenx = TRUE and IsEveny = FALSE
10       then return TRUE;
11       if IsEvenx = FALSE and IsEveny = TRUE
12       then return FALSE;
13       if x > y
14       then return TRUE;
15       else return FALSE;
16   end
```

To execute either of lines 3 or 6 requires 2 memory references, 3 logical operations and a comparison, plus the 18 operations required to compute weight, for a total of 24 operations each. To execute any of lines 4, 5, 7 or 8 requires a single memory reference; exactly 2 of these lines will be executed. Lines 9 and 11 each require 2 memory references, 2 logical operations and 2 comparisons. Line 13 requires 2 memory references, 1 comparison and a logical operation. This gives a maximum of 66 operations for this algorithm.

The algorithm for the case $b = 2$ is identical, except that "even" vectors in the case $b = 3$ are treated in the same way as vectors of *type* $6 \cdot 4$ in the case $b = 2$.

We include it here for completeness but do not analyze its cost:

```
1    EarlierThanb2(x, y)
2    begin
3        if weight of (x AND LeftBlockMask[1]) = 6
4        then let xIs6 = TRUE;
5        else let xIs6 = FALSE;
6        if weight of (y AND LeftBlockMask[1]) = 6
7        then let yIs6 = TRUE;
8        else let yIs6 = FALSE;
9        if xIs6 = TRUE and yIs6 = FALSE
10       then return TRUE;
11       if xIs6 = FALSE and yIs6 = TRUE
12       then return FALSE;
13       if x > y
14       then return TRUE;
15       else return FALSE;
16   end
```

Since in our estimates we use the additional conditions described in section 7.3.4, in our "poor-woman's" isomorphism testing we can only reject a candidate for the first row of a row-block in the case $b = 3$ if, given the "earliest" vector in *PWList* (computed in the algorithm Find10_2), this vector is *not* the candidate *and* if there exists another vector of type $4 \cdot 2^3$. However, we could still reject the candidate if any vector "earlier than" the candidate satisfies the condition, because this would still be isomorphic to a case which had already been seen. Given this additional constraint, we need the following algorithm for the case $b = 3$, which, given the previous earliest vector, finds the next-earliest vector.

In this algorithm, we use *Valid*[1.....*PWListSize*], which is TRUE if and only if *PWList*[*i*] would be a valid choice for the current row, specifically if it intersects all previous rows of the row-block in the correct pattern. We will show how

Valid[1,...,*PWListSize*] is initialized and maintained when we present the full algorithm for "poor-woman's" isomorphism rejection.

An algorithm for the case $b = 3$ which, given the previous earliest vector *prev*, finds the next-earliest vector in *PWList*, is:

```
1    NextEarliest(prev)
2    begin
3        let earliest = 1;
4        while Valid[earliest] = FALSE or (PWList[earliest] = prev) do
5            increment earliest;
6        for j =earliest+1 to PWListSize do
7        begin
8            if Valid[j] = TRUE and PWList[j] ≠ prev and
                 EarlierThan(PWList[j], PWList[earliest]) = TRUE
9            then earliest = j;
10       end;
11       return earliest;
12   end
```

First note that in this algorithm, there must be at least one vector in *PWList* which is both "valid" and not equal to *prev*. Therefore, since the second loop runs through all vectors in *PWList* which follow this vector, the two loops combine for a total of exactly *PWListSize*-1 iterations. Thus in our analysis we obtain a ceiling on the actual number of operations performed by the two loops by counting the number of operations performed by the most expensive loop, should it execute the maximum number of times. We shall therefore count the number of operations performed by the second loop, if it iterates *PWListSize*-1 times.

A ceiling on the number of operations used by this second loop is obtained by assuming both lines 8 and 9 are executed at every iteration. Line 8 uses 9 memory references, 3 comparisons, 3 logical operations, and uses the EarlierThan algorithm.

which as already shown uses a maximum of 66 operations. Line 8 therefore accounts for a maximum of 78 operations. Line 9 makes 2 memory references. We can ignore the cost of line 3 of this algorithm, which uses a single memory reference, as this is insignificant. Therefore we count the total number of operations performed by this algorithm as at most $80 \cdot (PWListSize-1)$. Recall that *PWListSize* is in fact a count of the number of codewords with weight 10 on the left-hand side, weight 2 in the correct block of the right-hand side, and which could be used to fill the current row. For the i'th row of a row-block, there are at most $2^i - 1$ such vectors, and thus at most $80 \cdot (2^i - 2)$ operations are performed by this algorithm for the i'th row of a row-block.

To use the NextEarliest algorithm to find the earliest vector of *all* those in *PWList*, we can initialize *prev* to zero, since a vector with a zero in every component will always be considered "earlier than" any other vector according to the algorithm EarlierThan.

9.6.3 Detailed Poor Woman Algorithm

We now present a more detailed algorithm to perform "poor-woman's" isomorphism rejection than that shown in Chapter 7. Given the current row r, the current candidate *cand* and buffer *Buf* of size *SizeOfBuf*, this algorithm will return TRUE if and only if *cand* can be rejected by the "poor-woman's" method. In this algorithm, we wish to be able, given any particular row, to access all of the rows of the row-block containing that row. To accomplish this, assume that at the start of the program, we initialize *BlockStart*[i] to contain the number of the first row of the row-block containing row_i. For example, *BlockStart*[11]= 9 in the case $b = 3$. The detailed algorithm follows.

```
1    PoorWoman(r, cand, Buf, SizeOfBuf)
2    begin
3        let prev = 0;
4        Find10_2(r, Buf, SizeOfBuf);
5        for i = 1 to PWListSize do
6            let Valid[i] = TRUE;
```

```
7       for i = BlockStart[r] to r do
8       begin
9           let earliest = NextEarliest(prev):
10          while EarlierThan(PWList[earliest], row_i) = TRUE do
11          begin
12              if BlockRow[i] > 1
13              then return TRUE:
14              for j = 1 to PWListSize do
15              begin
16                  if weight of (PWList[earliest] AND PWList[j]
                        AND RightBlockMask[i]) = 1
17                  then let Valid[j] = TRUE:
18                  else let Valid[j] = FALSE:
19                  if Valid[j] = TRUE and Even[j] = TRUE
20                  then return TRUE:
21              end:
22              let earliest = NextEarliest(PWList[earliest]):
23          end:
24          for j = 1 to PWListSize do
25          begin
26              let Valid[j] = TRUE:
27              let BlockAnd = PWList[j]:
28              for k = BlockStart[r] to i do
29              begin
30                  if row_k = PWList[j]
31                  then let Valid[j] = FALSE:
32                  let BlockAnd = BlockAnd AND row_k:
33              end:
34              if weight of (BlockAnd AND RightBlockMask[i]) ≠ 1
35              then let Valid[j] = FALSE:
```

```
36              end;
37          end;
38          return FALSE;
39      end
```

We can now consider the total number of operations required by the PoorWoman algorithm for the case $b = 3$. Since this algorithm is long, we shall only consider the most expensive lines, namely those which appear in the body of a loop, or which refer to another algorithm. This will make our analysis easier to understand and to verify. The number of operations used by these lines dominates the number of operations used in total.

Line 4 refers to the algorithm Find10_2, which requires $1229 \cdot (2^i - 1)$ operations for the i'th row of a row-block. It also requires 3 memory references, which are insignificant and which we shall ignore. The loop given on lines 5 and 6 iterates *PWListSize* times, but since there is only a single, cheap statement in the body of this loop, we shall ignore its cost in our analysis.

The outermost loop starting on line 7 iterates i times for the i'th row of a row-block. The body of this loop includes line 9, and two inner loops, one from lines 10–23, and the other from lines 24–36. Line 9 refers to the algorithm NextEarliest, which uses at most $80 \cdot (2^i - 2)$ operations for the i'th row of a row-block; we shall ignore the 2 extra memory references for this line.

The first inner loop, from lines 10–23, attempts to replace row_i by the current "earliest" vector. Often, this loop is never entered, as the earliest vector is not earlier than row_i. There is only more than one iteration of this loop in the rare case that row_i is the first row of a row-block, and there does not exist another "even" vector which could be used as the second row of the row-block. Therefore if we assume that this loop iterates once, with lines 10, 12 and 13 executing exactly once, and the remainder executing only 25% of the time, then we should have a fairly good approximation of the total number of operations used. Line 10 refers to the algorithm EarlierThan, which requires 66 operations. Line 12 uses 2 memory references, a logical operation

and a comparison, for a total of 4 operations.

The innermost loop from lines 14–21 iterates $2^i - 1$ times for the i'th row of a row-block. Line 16 requires 18 operations to compute the weight, plus 6 memory references, 3 logical operations and 1 comparison, for a total of 28 operations. Lines 17 and 18 both use 2 memory references, but since exactly one of these lines is executed for any iteration of the loop, they combine for a total of 2 operations. Line 19 uses 4 memory references, 2 comparisons and 2 logical operations, for a total of 8 operations. The total number of operations for this innermost loop is therefore $38 \cdot .25 \cdot (2^i - 1)$ for the i'th row of a row-block.

Line 22 refers to the NextEarliest algorithm, which uses at most $80 \cdot (2^i - 2)$ operations for the i'th row of a row-block: we shall ignore the 3 extra memory references for this line. Since we assume that this line is executed only 25% of the time, line 22 will account for $80 \cdot .25 \cdot (2^i - 2)$ operations.

Therefore the total number of operations for lines 10–23 is $\simeq 70 + 118 \cdot .25 \cdot (2^i - 1)$ per iteration of the outer loop, for the i'th row of a row-block.

Now consider the loop from lines 24–36. This loop iterates $PWListSize = 2^i - 1$ times at the i'th row of a row-block. Line 26 makes 2 memory references and line 27 makes 3 memory references.

The innermost loop from lines 28–33 iterates i times for the i'th row of a row-block. Line 30 uses 3 memory references, 1 logical operation and 1 comparison, for a total of 5 operations. Line 31 executes only a fraction of the time and uses only 2 memory references, so is not significant and we shall ignore it in our analysis. Line 32 uses 3 memory references and 1 logical operation, for a total of 4 operations. Therefore the total number of operations for this innermost loop is $9 \cdot i$ for the i'th row of a row-block.

Line 34 requires 18 operations to compute the weight, plus 3 memory references, 2 logical operations and 1 comparison, for a total of 24 operations. Line 35 is executed the majority of the time, and uses 2 memory references.

Therefore the total number of operations for lines 24–36 is $(31 + 9 \cdot i) \cdot (2^i - 1)$ per iteration of the outer loop, for the i'th row of a row-block. Recall that there

are i iterations of the outer loop for the i'th row of a row-block. Therefore the total number of operations executed by this algorithm for every candidate for row_j, the i'th row of a row-block, is approximately:

$$(1229 \cdot (2^i - 1)) + ((80 \cdot (2^i - 2) + (70 + 118 \cdot .25 \cdot (2^i - 1)) + ((31 + 9 \cdot i) \cdot 2^{i-1}) \cdot i.$$

which is approximately

$$(1260 + 119 \cdot i) \cdot (2^i - 1) + 70 \cdot i.$$

Since $70 \cdot i$ is insignificant compared to the remainder of the expression, we shall eliminate this from the expression and estimate the total cost of the PoorWoman algorithm as

$$(1260 + 119 \cdot i) \cdot (2^i - 1).$$

Since the value of i is at most 4, we shall not eliminate any further parts of the expression.

9.7 Operations per Row for $b = 3$

From the above analysis, we approximate the number of operations per candidate for row_j by adding the number of operations used by the AddToBuf, CheckBuf, and PoorWoman algorithms. Since in our program we execute PoorWoman last, this algorithm is executed for only approximately one tenth of the candidates. Therefore the total number of operations per candidate for row_j is approximately:

$$(8 \cdot 2^{j-2}) + (37 \cdot 2^{j-2}) + .1 \cdot (1260 + 119 \cdot i) \cdot (2^i - 1).$$

Note that since i refers to the position of row_j within its row-block, i and $2^i - 1$ are both always insignificant in comparison with 2^{j+2}. For example, at row_{12}, $j = 12$ and $i = 4$. In fact this is the largest possible value of i, and yet $(1260+119 \cdot i) \cdot (2^i - 1)$ is completely dominated by $45 \cdot 2^{j-2}$. We can therefore approximate the number of operations per candidate for row_j for the case $b = 3$ as $45 \cdot 2^{j-2}$, which is totally accounted for by the AddToBuf and CheckBuf algorithms.

Row	#Candidates	#Operations
9	2 E10	2 E15
10	5 E10	9 E15
11	2 E12	7 E17
12	8 E11	6 E17
13	4 E10	6 E16
14	7 E10	2 E17
15	1 E12	6 E18
16	3 E8	5 E15
Total		8 E18

Table 9.1: Estimates of Candidates and Operations by Row for $b = 3$

This suggests, among other things, that if improvements are to be made to our overall program, we should concentrate our efforts on improving or restricting the use of these algorithms.

Given the estimated number of candidates per row shown in Fig. 7.5, we combine this number with our above estimate of the number of operations per candidate, to obtain the results shown in Fig. 9.1.

In our estimates we never find survivors for row_{16}. Therefore we can estimate the total number of operations by the sum of the number of operations for each of $row_9, \ldots, row_{16}$, namely 8 E18 operations. On a machine which could execute 1 billion operations per second, this program would require about 9 E4 days to execute.

9.8 Operations per Row for $b = 2$

As above, we approximate the number of operations per candidate for row_j by adding the number of operations used by the appropriate algorithms. In this case, we use the AddToBuf, CheckBuf, and CheckSubcode algorithms. Since in our program we execute CheckSubcode last, this algorithm is executed for only approximately one tenth of the candidates. Therefore the total number of operations per candidate for row_j is approximately:

$$(8 \cdot 2^{j+1}) + (37 \cdot 2^{j-1}) + .1 \cdot (9 \cdot 2^{2j-2}).$$

Clearly the factor of 2^{2j-2} dominates the above equation. We can therefore ap-

Row	#Candidates	#Operations
8	3 E8	5 E12
9	4 E11	3 E16
10	1 E13	3 E18
11	4 E11	4 E17
Total		3 E18

Table 9.2: Estimates of Candidates and Operations by Row for $b = 2$

proximate the number of operations per candidate for row_j for the case $b = 3$ as 2^{2j-2}. which is totally accounted for by the CheckSubcode algorithm. Clearly if improvements are to be made to the overall program used for $b = 2$. we should concentrate our efforts on the CheckSubcode algorithm.

Given the estimated number of candidates per row shown in Fig. 7.6. we combine this number with our above estimate of the number of operations per candidate. to obtain the results shown in Fig. 9.2.

In our estimates we never find survivors for row_{11}. Therefore we can estimate the total number of operations by the sum of the number of operations for each of $row_8, \ldots, row_{11}$. namely 3 E18 operations. On a machine which could execute 1 billion operations per second. this program would require about 4 E4 days to execute.

9.9 Summary

The goal of this chapter was to present a method of converting estimates of search size to estimates of the time required to complete a search. using the search for a (48. 24. 12) self-dual doubly-even code as an example.

For this example. the rough estimates calculated indicate that to complete the search for a (48. 24. 12) self-dual doubly-even code. we require approximately 1 E5 days of computing time on a machine executing 1 billion operations per second.

These estimates. and the analysis required to obtain them. are probably most useful if taken primarily as a diagnostic tool. Indeed. during our analysis we discovered that for the case $b = 3$. almost all of the execution time is due to two algorithms. AddToBuf and CheckBuf: for the case $b = 2$. almost all of the execution time is due

to only one algorithm, CheckSubcode. With this information, we can concentrate our efforts on improving or restricting the use of these algorithms as this will have the greatest impact on the execution time.

Chapter 10

Conclusion

In this chapter we conclude the thesis with an examination of the objectives and main results.

In Chapter 4 we considered 2-(28, 12, 11) designs with an automorphism of order 7 without fixed points or blocks. Our objective was to determine if all such designs could be embedded as derived designs in a symmetric 2-(64, 28, 12) design. We found that four 2-(28, 12, 11) designs could not be embedded, thus providing the first examples of quasi-symmetric quasi-derived designs which are not derived designs. We also provided the first examples of quasi-symmetric 2-(36, 16, 12) designs with trivial automorphism groups, and improved on the numbers of 2-(64, 28, 12) designs and 2-(36, 16, 12) designs as found in the Handbook of Combinatorial Designs [31].

In Chapter 5 we considered the search for a 2-(46, 6, 1) design. Our objective was to determine if a 2-(46, 6, 1) design exists. We gave detailed information on how one may structure a search assuming the remaining case, and presented estimates of the time required to complete the search. Finally we compared these estimates to the results obtained from an actual search which employed essentially the same method. According to the results of this search, a 2-(46, 6, 1) design does not exist.

In Chapters 6 and 7 we examined (48, 24, 12) self-dual doubly-even codes, of which the Extended Quadratic Residue Code is the only known example. Our objective was to determine if the Extended Quadratic Residue Code is the only (48, 24, 12) self-dual doubly-even code. The search for such a code may be partitioned into 3 cases, namely $b = 2$, 3 or 4, where b is the maximum dimension of a subcode of length 24. A search

assuming the case $b = 4$ had previously been completed, finding only the Extended Quadratic Residue Code itself [19]. In this thesis we considered the cases $b = 3$ and $b = 2$, giving information on the structure of the generator matrix for each of the two remaining cases. We also gave information on how one may structure a search assuming each of these cases, and gave estimates of the size of these searches.

In Chapter 8 we described several computational methods which one may apply to these types of searches in general. Finally, in Chapter 9 we described how estimates of search size may be translated to estimates of the time required to complete the search, using the search for a (48, 24, 12) self-dual doubly-even code as an example. We identified that 3 algorithms would be responsible for a great majority of the execution time in this search. Therefore a clear next step is to focus on improving or restricting the use of these algorithms.

At the end of some of these chapters we also identified several related problems which remain to be solved. The methods described in this thesis may be applied to these problems, and to others of a similar type.

Bibliography

[1] Abel, R.J.R. and Grieg, M., BIBDs with Small Block Size. CRC Handbook of Combinatorial Designs. C. Colbourn and J. Dinitz eds., CRC Press, New York 1996, 41–47.

[2] Abel, R.J.R. and Mills, W.H., Some New BIBDs with $k = 6$ and $\lambda = 1$, *J. Combin. Des.* 3 (1995), 381–391.

[3] Assmus, E.F. Jr. and Key, J.D., "Designs and their Codes", Cambridge University Press, Cambridge, 1992.

[4] Assmus, E.F. Jr. and Mattson, H.F. Jr., New 5-designs, *J. Combin. Theory* 6 (1969), 122–151.

[5] Bachoc, C., private communication.

[6] Berlekamp, E. R., *Algebraic Coding Theory*, McGraw-Hill, 1968.

[7] Bose, R.C., Shrikhande, S.S. and Singhi, N.M., Edge regular multigraphs and partial geometric designs with an application to the embedding of quasi-residual designs, in *Thieorie Combinatorie, Tomo I*, Atti Dei Convegni Lincei 17 (1976), 49–81.

[8] Carter, J.L. "On the Existence of a Projective Place of Order Ten", Ph.D. Dissertation, University of California, Berkeley, 1974.

[9] Cameron, P.J. and van Lint, J.H., "Designs, Graphs, Codes and their Links", London Mathematical Society, Student Texts 22, Cambridge University Press, Cambridge, 1991.

[10] Conway, J. H. and Pless, V., On Primes Dividing the Group Order of a Doubly-Even (72, 36, 16) code and the Group Order of a Quaternary (24, 12, 10) Code. *Discrete Mathematics* 38 (1982), 143–156.

[11] Conway, J. H. and Pless, V., On the Enumeration of Self-Dual Codes. *J. Combin. Theory* 28 (1980), 26–53.

[12] Conway, J. H., Pless, V. and Sloane, N. J. A., The Binary Self-Dual Codes of Length up to 32: A Revised Enumeration. *J. Combin. Theory* 60 (1992), 183–195.

[13] Conway, J. H. and Sloane, N. J. A., A New Upper Bound on the Minimal Distance of Self-Dual Codes. *IEEE Trans. Inform. Theory* 36 (1990), 1319–1332.

[14] Denniston, R.H.F., A Steiner system with a maximal arc. *Ars Combin.* 9 (1980), 247–248.

[15] Ding, Y., "A Study of Balanced Incomplete Block Designs". Master's Thesis, Concordia University, 1993.

[16] Ding, Y., Houghten, S., Lam, C., Smith S., Thiel, L. and Tonchev, V.D., Quasi-symmetric 2-(28, 12, 11) Designs with an Automorphism of Order 7. *J. Combin. Des.* 6 (1998), 213–223.

[17] Greig, M., Some balanced incomplete block design constructions. *Congr. Numer.* 77 (1990), 121–134.

[18] Hanani, H., Balanced incomplete block designs and related designs. *Discrete Math.* 11 (1975), 255–369.

[19] Houghten, S.K., "Construction of Extremal (48, 24, 12) Doubly-Even Codes". Master's Thesis, Concordia University, 1993.

[20] Houghten, S.K., Lam, C. and Thiel, L., Construction of (48, 24, 12) Doubly-Even Self-Dual Codes. *Congr. Numer.* 103 (1994), 41–53.

[21] Huffman, W. C., Automorphisms of Codes with Applications to Extremal Doubly Even Codes of Length 48. *IEEE Trans. Inform. Theory* 28 (1982), 511–521.

[22] Huffman, W. C. and Yorgov, V. Y., A (72.36.16) Doubly-Even Code Does Not Have an Automorphism of Order 11. *IEEE Trans. Inform. Theory* 33 (1987), 749–752.

[23] Jungnickel, D. and Tonchev, V.D., Exponential number of quasi-symmetric SDP designs and codes meeting the Grey-Rankin bound, Designs. *Codes and Cryptography* 1 (1991), 247–253.

[24] Knuth, D.E., Estimating the Efficiency of Backtrack Programs. *Math. Comp.* 29 (1975), 121–136.

[25] Koch, H.V., Unimodular Lattices and Self-dual Codes, Proceedings of the International Congress of Mathematicians, 1986.

[26] Lam, C., Thiel, L. and Tonchev, V.D., On quasi-symmetric 2-(28.12.11) and 2-(36.16.12) designs. *Designs, Codes and Cryptography* 5 (1995), 43–56.

[27] Leon, J. S., Computing Automorphism Groups of Error-Correcting Codes. *IEEE Trans. Inform. Theory* 28 (1982), 496–511.

[28] MacWilliams, F.J. and Sloane, N.J.A., "The Theory of Error-Correcting Codes", North-Holland, 1977.

[29] Mallows, C.L. and Sloane, N.J.A., An upper bound for self-dual codes. *Info. and Control* 22 (1973), 188–200.

[30] Mathon, R., Computational Methods in Design Theory, in *Surveys in Combinatorics*, London Mathematical Society, Lecture Notes Series 166, Cambridge University Press, Cambridge 1991, 101–118.

[31] Mathon, R. and Rosa, A., 2-(v,k,λ) Designs of small order, CRC Handbook of Combinatorial Designs, C. Colbourn and J. Dinitz eds., CRC Press, New York 1996, 3–41.

[32] McKay, B.D., autoson — a distributed batch system for UNIX workstation networks (version 1.3), *Technical Report TR-CS-96-03*, Joint Computer Science Technical Report Series, Australian National University.

[33] Mills, W.H., A new block design, *Congr. Numer.* 14 (1975), 461–465.

[34] Mills, W.H., Two new block designs, *Utilitas Math.* 7 (1975), 73–75.

[35] Mills, W.H., The construction of balanced incomplete block designs with $\lambda = 1$, *Congr. Numer.* 20 (1977), 131–148.

[36] Mills, W.H., The construction of BIBDs using nonabelian groups, *Congr. Numer.* 21 (1978), 519–526.

[37] Mills, W.H., The construction of balanced incomplete block designs, *Congr. Numer.* 23 (1979), 73–86.

[38] Oral, H. and Phelps, K. T., Almost all Self-Dual Codes are Rigid, *J. Combin. Theory* 60 (1992), 264–276.

[39] Pless, V., 23 Does Not Divide the Order of the Group of a (72, 36, 16) Doubly-Even Code, *IEEE Trans. Inform. Theory* 28 (1982), 113–117.

[40] Pless, V., "Introduction to the Theory of Error-Correcting Codes", John Wiley and Sons, 1989.

[41] Pless, V. and Thompson, J. G., 17 Does Not Divide the Order of the Group of a (72, 36, 16) Doubly-Even Code, *IEEE Trans. Inform. Theory* 28 (1982), 537–541.

[42] Shrikhande, M.S., Quasi-symmetric Designs, CRC Handbook of Combinatorial Designs, C. Colbourn and J. Dinitz eds., CRC Press, New York 1996, 430–434.

[43] Shrikhande, M.S. and Sane, S.S., "Quasi-Symmetric Designs", London Math. Soc. Lecture Note Ser. 164, Cambridge 1991.

[44] Sloane, N. J. A., Is There a (72, 36) $d = 16$ self-dual code?, *IEEE Trans. Inform. Theory* 19 (1973), 251.

[45] Tonchev, V.D., Embedding of the Witt-Mathieu system S(3.6.22) in a symmetric 2-(78, 22, 6) design, *Geometriae Dedicata* 22 (1987), 49–75.

[46] Tonchev, V.D., Codes, CRC Handbook of Combinatorial Designs, C. Colbourn and J. Dinitz eds., CRC Press, New York 1996, 517–543.

[47] van Lint, J.H. and Tonchev, V.D., A Class of Non-embeddable Designs, *J. Combin. Theory, series A* 62 (1993), 252–260.

Appendix A: Incidence Matrices of 4 Non-Extendable Quasi-Derived Designs

Case 1, solution 3, base rows:

0000000 1101000 1101000 1101000 1101000 1101000 0010111 0111001 0111001
0001111 1010001 1001100 0110100 0101001 0011010 0000000 1100101 1001011
0011011 0011010 1010100 0001101 1000011 1101000 1001011 0000000 1100101
0101011 1101000 0011100 1000110 0110010 1010001 1011100 0010111 0000000

Case 1, solution 4, base rows:

0000000 1101000 1101000 1101000 1101000 1101000 0010111 0111001 0111001
0001111 1001010 1000101 0101100 0011001 0010110 0000000 1101001 1110100
0011011 1000011 1011000 0010110 0101010 0110001 1010011 0000000 1010011
0101011 1001001 1100010 0110001 0000111 0101100 0100111 1001110 0000000

Case 26, solution 33, base rows:

0000000 0010110 0101100 1101000 1000101 0010110 1110100 0100111 1110100
0010111 1000000 1110000 1101000 1110000 0111011 1001000 1001000 0110101
0010111 0010011 0011001 0001101 1010111 0000001 0100001 0111100 0101000
0010111 1100111 1001010 0110100 0000100 1010010 0011011 0000110 1000001

Case 171, solution 34, base rows:

1000000 1010000 1010000 1101000 1101000 1101000 1101100 1110100 1111100
1101000 1001000 0100111 1010001 0011010 1101011 0100100 0001111 0010000
0110111 1100000 1101010 0011010 1000110 0100000 1001110 0001100 1010001
1001010 0111111 0110000 1000110 0110100 1100001 0001010 1000100 0010011

Appendix B: Incidence Matrix of a Quasi-Symmetric 2-(36,16,12) with a Trivial Automorphism Group

000001111011010100000101101100111000101011010000011001101100001
000011000001001110101011101111010010001011111101100010000010000
000011110010101101000100011010010101011101100100010100010111000
001001101001100011101100001001001101110010001101011110001000010
001011011100010101001011000010001100100100100111100011111100000
010101000111010000111100100010001111011000111100000000011100101
010101110000011000100011110010111101000000001001111111000001100
011001000101110110010000011010010011100100011010111000101011000
011001100110110000001111000101110000010000110010110110010010011
011111100001101001001000010100100000101011001011100000011101101
100001011011110000010010101011100010110001000111100100000101110
100101010000000011011110110001110001001110100110101100101000001
100101100111001011000001100001000011010110010011010011110101000
101011010100000000101100111100101010011100001010001011000111010
101101011000011110010019000000100100011111011000010010011010110
111011101000000000110110001110010001000110010101000001110001111
000000110111001000011111010111001101100111001010000000100010110
000100001000111010011101011111010001111010110000100101101000001
000110000111000111010110001101111100010000000001101001011011100
001010011011001110101000010100100111010000010110110101100000101
001110100010100110110001100011101000000101101110010001001001011
010010101110011011010010000100010011001001001110001111001100010
010110110000011110100100001001001010100110110010101100010100110
011100001010101001001001101010011010011110000000100101101010110
011100011000000011110101100100010100110001111011000100100111000
011110000101010100000010111001000101111011100000000111100001011

1000101111011000110000001100100110000100011110001010101100001111
1001000001101101011000010111101001000010100111110011001100000101
1001101010000100010110010110010111101010000101100100100000011101
1010000011000111011110001001110010010000110000001111000101110011
1010100000111000001100111101000101101001101000000111100111001001
1100000011100011101011000100111100001010001000010100101011011101
1100001101001001011010110000001000111100111101000010010010111001
1100100000011100111001110000001011110011010000011100001001100111
1101000111011010000000100111010010000001001111010101010011100011
1110001100101011000101001000010001101010000011011010111100100011